CSC-241 Intro to Computer Organization
Notes

Qixin Deng

Contents

1 C Make & CLion

CMake is an open-source, cross-platform family of tools designed to build, test, and package software. It is primarily used to control the software compilation process using simple platform and compiler-independent configuration files, which then generate native build environments that you can use in your preferred development environment.

This document outlines the steps to build a simple C++ project using CMake. The project includes a main program file and a library. The project has the following structure:

```
MyProject/
|-- CMakeLists.txt
|-- src/
|   |-- main.c
|   |-- library.c
|-- include/
    |-- library.h
```

In the root directory, create a 'CMakeLists.txt' file with the following content:

```
cmake_minimum_required(VERSION 3.27)
project(C_Project C)

set(CMAKE_C_STANDARD 11)
include_directories(${PROJECT_SOURCE_DIR}/include)
add_library(MyLib STATIC src/library.c)
add_executable(MyExecutable src/main.c)

target_link_libraries(MyExecutable MyLib)
```

- `cmake_minimum_required(VERSION 3.27)`: Specifies that the minimum required version of CMake is 3.10.
- `project(C_Project C)`: Defines the project name as `C_Project` using C programming language.
- `set(CMAKE_C_STANDARD 11)`: Specifies that the C standard should be C11.
- `include_directories(${PROJECT_SOURCE_DIR}/include)`: Specifies the directory for header files.
- `add_library(MyLib src/library.c)`: Creates a library named `MyLib` from the source file `mylib.c`.
- `add_executable(MyExecutable src/main.c src/library.c)`: Creates an executable named `MyExecutable` from the source file `main.c library.c`.
- `target_link_libraries(MyExecutable MyLib)`: Links the `MyLib` library to the `MyExecutable` executable.

1. Create a build directory:

```
mkdir build
cd build
```

2. Configure the project:

```
cmake ..
```

3. Compile the project:

```
make
```

4. Run the executable:

```
./MyExecutable
```

5. Clean the build files:

```
make clean
```

   The command make clean is used in a build system that utilizes make. It typically removes all files generated during the build process, such as object files (.o), executables, and other intermediate files. This command is useful for cleaning up the working directory, especially when you want to rebuild the project from scratch to ensure that no outdated or corrupted files interfere with the build.

For the CLion, the only thing you need to do is modify "Run/Debug Configurations", switch Target to "All targets" and specify the executable file.

2 Hello, World!

The first step in learning any programming language is to write a simple program. Below is a simple C program that prints "Hello, World!" to the screen.

```c
#include <stdio.h> //preprocessor

int main() {
    // My first C program
    printf("Hello, World!\n");
    return 0;
}
```

Listing 1: Hello World Program

- "All C programs need to include the **main()** function. The code execution starts from the main() function.
- /* ... */ is used for adding comments.
- **printf()** is used to format and output data to the screen. The printf() function is declared in the 'stdio.h' header file.
- **stdio.h** is a header file (standard input-output header file), and #include is a preprocessor directive used to include header files. If the compiler encounters the **printf()** function without finding the stdio.h header file, a compilation warning or error will occur.
- The **return 0;** statement is used to indicate that the program exits successfully."
- A **semicolon** ";" must be at the end of the statement.
- Functions must be **declared before use**.

2.1 C Preprocessor

The C preprocessor is not part of the compiler, but it is a separate step in the compilation process. In short, the C preprocessor is merely a text substitution tool that instructs the compiler to perform necessary preprocessing before actual compilation. We abbreviate the C Preprocessor as CPP.

All preprocessor commands begin with a hash symbol (`#`). It must be the first non-whitespace character, and for readability, preprocessor directives should start in the first column. Below is a list of all important preprocessor directives:

Directive	Description
`#define`	Defines a macro
`#include`	Includes a source code file
`#undef`	Undefines a previously defined macro
`#ifdef`	Returns true if the macro is defined
`#ifndef`	Returns true if the macro is not defined
`#if`	Compiles the code below if the given condition is true
`#else`	Provides an alternative to `#if`
`#elif`	Compiles the code below if the previous `#if` condition is false and the current condition is true
`#endif`	Ends a `#if`...`#else` conditional compilation block
`#error`	Outputs an error message when encountering a standard error
`#pragma`	Issues special commands to the compiler using standardized methods

```c
#include <stdio.h>
#include "myheader.h"
```

These directives tell the CPP to fetch `stdio.h` from the system library and add its content to the current source file. The next line instructs the CPP to fetch `myheader.h` from the local directory and include its content in the current source file.

2.2 C Header Files

Header files in C are files with the extension `.h` that contain declarations of C functions and macro definitions, which can be shared among multiple source files. There are two types of header files: those written by the programmer and those provided by the compiler.

To use a header file in a program, you must include it using the C preprocessor directive `#include`. We have already seen the `stdio.h` header file, which is a compiler-provided header file. Including a header file is equivalent to copying the contents of the header file into the source file, but this is not done manually because it is prone to errors, especially in programs consisting of multiple source files.

A simple practice in C programs is to place all constants, macros, system-wide global variables, and function prototypes in header files and include these header files wherever needed.

2.2.1 Syntax for Including Header Files

The `#include` preprocessor directive is used to include both user and system header files. There are two forms:

- `#include <file>`
  - This form is used to include system header files. It searches for the file named `file` in the standard list of system directories. When compiling the source code, you can prepend directories to this list using the `-I` option.
- `#include "file"`
  - This form is used to include user header files. It searches for the file named `file` in the directory containing the current file. When compiling the source code, you can prepend directories to this list using the `-I` option.

2.2.2 How `#include` Works

The `#include` directive instructs the C preprocessor to treat the specified file as input. The output of the preprocessor includes the output generated by the included file and the text following the `#include` directive. For example, if you have a header file `header.h` as follows:

```
char *test (void);
```

And a main program `program.c` that uses the header file as follows:

```
int x;
#include "header.h"

int main (void)
{
   puts (test ());
}
```

The compiler will see the following code:

```
int x;
char *test (void);

int main (void)
{
   puts (test ());
}
```

2.2.3 Including a Header File Only Once

If a header file is included twice, the compiler will process the contents of the header file twice, leading to errors. To prevent this, the standard practice is to enclose the entire content of the header file in conditional compilation statements, as follows:

```
#ifndef HEADER_FILE
#define HEADER_FILE

/* the entire header file content */

#endif
```

This structure is commonly referred to as an `#ifndef` wrapper. When the header file is included again, the condition evaluates to false because `HEADER_FILE` is already defined. At this point, the preprocessor skips the entire content of the file, and the compiler ignores it.
Consider a scenario where one header file includes another:

```
// a.h
#include "functions.h"
// b.h
#include "functions.h"
// main.c
#include "a.h"
#include "b.h"
```

Without include guards, functions.h would be included twice in main.c (once via a.h and once via b.h), leading to multiple declarations. However, with #ifndef guards in functions.h, the preprocessor will ensure that the contents of functions.h are only included once, regardless of how many times it is indirectly included.

While repeating the same function declaration multiple times in the same scope does not cause any errors or have any additional effect beyond the first declaration. This is because a function declaration simply tells the compiler about the existence of a function, specifying its name, return type, and parameters, without providing the actual implementation (body) of the function. But, the issue of multiple inclusions causing errors typically arises when there are definitions, such as variable definitions or inline function definitions, in the header file. For instance, if the header file contained something like:

```
int global_variable = 0;  // Definition
```

3 Types of Tokens

C tokens mainly include:

- Keywords
- Identifiers
- Constants
- String Literals
- Operators
- Separators

3.1 Keywords

```
auto      break     case       char      const          continue  default     do
double  else       enum      extern   float           for              goto          if
int long register return short signed sizeof static
struct   switch    typedef  union    unsigned  void            volatile   while
```

On December 16, 1999, the C99 standard introduced 5 additional keywords:

```
inline    restrict    _Bool   _Complex    _Imaginary
```

On December 8, 2011, the C11 standard introduced 7 additional keywords:

```
_Alignas  _Alignof  _Atomic  _Static_assert  _Noreturn  _Thread_local  _Generic
```

Keywords are reserved words in C that have special meanings. They cannot be used as names for variables, constants, or other identifiers.The following are the standard keywords in the C language:

3.2 Identifiers

Identifiers are the names of variables, functions, arrays, etc. An identifier starts with a letter (A-Z or a-z) or an underscore (_), followed by zero or more letters, underscores, and digits (0-9). C is case-sensitive, so ‘Manpower‘ and ‘manpower‘ are different identifiers. Below are examples of valid identifiers in C:

```
mohd       zara    abc   move_name  a_123
myname50   _temp   j     a23b9      retVal
```

3.3 Constants

Constants are fixed values that do not change during program execution. They can be of various types, such as integer constants, floating-point constants, character constants, etc. Below are examples of valid constants in C:

```c
const int MAX = 100;  // Integer constant
const float PI = 3.14;  // Floating-point constant
const char NEWLINE = '\n';  // Character constant
```

3.4 String Literals

String literals are sequences of characters enclosed in double quotes. They automatically end with a null character '\0'.

```c
char greeting[] = "Hello, World!";
```

3.5 Operators

Operators perform various operations like arithmetic, logical, and comparison operations. Common C operators include:

- Arithmetic operators: `+`, `-`, `*`, `/`, `%`
- Relational operators: `==`, `!=`, `>`, `<`, `>=`, `<=`
- Logical operators: `&&`, `||`, `!`
- Bitwise operators: `&`, `|`, `^`, `~`, `<<`, `>>`
- Assignment operators: `=`, `+=`, `-=`, `*=`, `/=`, `%=`
- Other operators: `sizeof`, `?:`, `&`, `*`, `->`, `.`

Example:

```c
int a = 5, b = 10;
int sum = a + b;  // Using the arithmetic operator +
int isEqual = (a == b);  // Using the relational operator ==
```

```c
#include <stdio.h>

int main() {
   int c;
   int a = 10;
   c = a++;
   printf("Assignment before increment:\n");
   printf("Line 1 - Value of c: %d\n", c );
   printf("Line 2 - Value of a: %d\n", a );

   a = 10;
   c = a--;
   printf("Line 3 - Value of c: %d\n", c );
   printf("Line 4 - Value of a: %d\n", a );

   printf("Increment before assignment:\n");
   a = 10;
   c = ++a;
   printf("Line 5 - Value of c: %d\n", c );
   printf("Line 6 - Value of a: %d\n", a );

   a = 10;
   c = --a;
   printf("Line 7 - Value of c: %d\n", c );
   printf("Line 8 - Value of a: %d\n", a );
}
```

```c
#include <stdio.h>

int main() {
   int a = 4;
   short b;
   double c;
   int* ptr;

```

```c
    /* sizeof operator example */
    printf("Line 1 - Size of variable a = %lu\n", sizeof(a));
    printf("Line 2 - Size of variable b = %lu\n", sizeof(b));
    printf("Line 3 - Size of variable c = %lu\n", sizeof(c));

    /* & and * operators example */
    ptr = &a;    /* 'ptr' now contains the address of 'a' */
    printf("Value of a is %d\n", a);
    printf("*ptr is %d\n", *ptr);
    printf("Address of a is %p\n", ptr);

    /* Ternary operator example */
    a = 10;
    b = (a == 1) ? 20 : 30;
    printf("Value of b is %d\n", b);

    b = (a == 10) ? 20 : 30;
    printf("Value of b is %d\n", b);
}
```

```
Line 1 - Size of variable a = 4
Line 2 - Size of variable b = 2
Line 3 - Size of variable c = 8
Value of a is 4
*ptr is 4
Address of a is 0x16ce1358c
Value of b is 30
Value of b is 20
```

This code demonstrates several operators in C:

- `sizeof`: Determines the size of a variable.
- `&` and `*`: Used for pointer operations, `&` gets the address of a variable, and `*` accesses the value at the address.
- Ternary operator `? :`: A conditional expression that assigns a value based on a condition.

3.6 Separators

Separators are used to separate statements and expressions. Common separators include:

- Comma , Separates variable declarations or function parameters.
- Semicolon ; is a statement terminator, indicating the end of a logical entity.
- Parentheses (), curly braces {}, square brackets [ ]: Used for grouping expressions, defining code blocks, and array indices.

3.7 Comments

C supports two types of comments:

- `//` Single-line comments
- `/* */` Multi-line comments

Example:

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

/*
This is a multi-line comment
It can span multiple lines
*/

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

4 C Data Types

In C, data types refer to an extensive system used for declaring variables or functions of different types. The type of a variable determines the amount of space it occupies in storage and how the bit pattern stored is interpreted.
An integral data type is fundamentally an integer, meaning it has no fractional portion. Integral types come in different sizes, which determine how many bytes are required to store values and how large those values can be. Integral types can be either signed or unsigned:

- **Signed** values can include positive and negative numbers (including zero).
- **Unsigned** values only include positive numbers (including zero).

4.1 Integral Types

This table shows the range of values for the integral types on a typical, modern 64-bit (LP64) computer:

Type	Size (bytes)	Range (Binary)	Range (Decimal)
`signed char`	1	-2^7 to $2^7 - 1$	-128 to 127
`unsigned char`	1	0 to $2^8 - 1$	0 to 255
`signed short int`	2	-2^{15} to $2^{15} - 1$	$-32,768$ to $32,767$
`unsigned short int`	2	0 to $2^{16} - 1$	0 to $65,535$
`signed int`	4	-2^{31} to $2^{31} - 1$	$-2,147,483,648$ to $2,147,483,647$
`unsigned int`	4	0 to $2^{32} - 1$	0 to $4,294,967,295$
`signed long int`	8	-2^{63} to $2^{63} - 1$	$-9,223,372,036,854,775,808$ to $9,223,372,036,854,775,807$
`unsigned long int`	8	0 to $2^{64} - 1$	0 to $18,446,744,073,709,551,615$

Table 1: Size and Range of Integral Types

4.2 Floating Point Types

Unlike the integral types, floating point types are not divided into signed and unsigned. All floating point types are signed only. Floating point numbers follow the IEEE-754 Floating Point Standard. Here are the approximate ranges of the IEEE-754 floating point numbers on Intel x86 computers, but the actual size is heavily dependent on the compiler; for long double, it could be 8 bytes, 12 bytes, or 16 bytes. In particular, all that you are guaranteed is that double doesn't have less precision and range than float, and that long double doesn't have less precision and range than double.:

Type	Size (bytes)	Smallest Positive Value	Largest Positive Value
`float`	4	1.1754×10^{-38}	3.4028×10^{38}
`double`	8	2.2250×10^{-308}	1.7976×10^{308}
`long double`	8	3.3621×10^{-4932}	1.1897×10^{4932}

Table 2: Approximate Ranges of IEEE-754 Floating Point Numbers

```c
#include <stdio.h>
#include <float.h>

int main()
{
    printf("Number of bytes: %lu bytes\n", sizeof(float));
    printf("Minimum value of float: %E\n", FLT_MIN);
    printf("Maximum value of float: %E\n", FLT_MAX);
    printf("Precision: %d\n", FLT_DIG);

    return 0;
}
```

```
Number of bytes: 4 bytes
Minimum value of float: 1.175494E-38
Maximum value of float: 3.402823E+38
Precision: 6
```

Another example:

```c
int main()


```

```c
{
    hello();
    float f = 3.1415926535f;  // float typically holds 6-7 significant digits
    double d = 3.1415926535;  // double typically holds 15-16 significant digits

    printf("Float:  %.10f\n", f);
    printf("Double: %.10lf\n", d);

    return 0;
}
```

```
Float:  3.1415927410
Double: 3.1415926535
```

In C, floating-point numbers are typically represented using the IEEE 754 standard. This standard defines a binary format for both single-precision (`float`) and double-precision (`double`) numbers. here, we will focus on understanding the binary representation of a single-precision `float`.

4.2.1 IEEE 754 Single-Precision Format

A 32-bit float is divided into three parts:

- **Sign bit**: 1 bit
- **Exponent**: 8 bits
- **Significand (Mantissa)**: 23 bits

$$\text{Floating-point number} = (-1)^{\text{sign}} \times 1.\text{mantissa} \times 2^{(\text{exponent}-127)}$$

4.2.2 Example: Representing Float in Binary

Let's go through the process of representing 8.125 as a 32-bit float.
Step 1: Convert Integer and Fractional Parts to Binary. 8.125 has an integer part `8` and a fractional part `0.125`.

- **Integer part (8):**

$$8_{10} = 1000_2$$

- **Fractional part (0.125):**

$$\begin{aligned} 0.125 \times 2 &= 0.25 \quad \text{(integer part: 0)} \\ 0.25 \times 2 &= 0.5 \quad \text{(integer part: 0)} \\ 0.5 \times 2 &= 1.0 \quad \text{(integer part: 1)} \end{aligned}$$

At this point, the fractional part becomes 0, so we stop. Thus,

$$8.125_{10} = 1000.001_2$$

Step 2: Normalize the Binary Number. To normalize, adjust the binary number so it is of the form:

$$1.\text{mantissa} \times 2^{\text{exponent}}$$

For 8.125:

$$1.000001 * 2^3$$

Step 3: Determine the Exponent. The exponent in the IEEE 754 format is biased by 127, because the exponent can be positive and negtive. Here the exponent is 1, so the biased exponent is:

$$\text{Exponent} = 3 + 127 = 130$$

In binary, $130_{10} = 10000010_2$.
Step 4: Pack the Number into IEEE 754 Format. Now, combine the sign bit, exponent, and mantissa:

- **Sign bit**: 0 (positive)
- **Exponent**: 10000010_2
- **Mantissa**: The first 23 bits of the mantissa $00000100000000000000000_2$

Final binary representation:

$$0\ 10000010\ 00000100000000000000000$$

4.3 Literal Constants

Literal constants such as ‘42‘ are of type ‘int‘, and constants like ‘42.0‘ are of type ‘double‘. If a literal is too large for an ‘int‘, its type will be ‘long int‘. To force a smaller literal number to a particular type, append a suffix:

- Append ‘f‘ to a floating-point value to make it a ‘float‘, e.g., ‘42.0f‘.
- Append ‘L‘ to an integral value to make it a ‘long int‘, e.g., ‘42L‘.
- Append ‘U‘ to an integral value to make it an ‘unsigned int‘, e.g., ‘42U‘.
- Combine them to make an ‘unsigned long int‘, e.g., ‘42UL‘ or ‘42LU‘.

4.4 Void Type

The `void` type specifies that no value is available. It is typically used in the following three cases:

- **Function Returns as Void**: C functions may not return a value, or they may return void. For example:

```
void exit(int status);
```

- **Function Arguments as Void**: C functions may not accept any parameters. A function with no parameters can accept a void. For example:

```
int rand(void);
```

- **Pointers to Void**: A pointer of type `void *` represents the address of an object, but not its type. For example, the memory allocation function `void *malloc(size_t size);` returns a pointer to void, which can be cast to any data type.

4.5 Type Casting

Type casting is the process of converting a variable from one data type to another. For example, if you want to store a `long` type value into a simple integer, you need to cast the `long` type to an `int` type. You can use the type casting operator to explicitly convert a value from one type to another, as shown below:

```
(type_name) expression
```

Here is an example:

```
#include <stdio.h>

int main() {
   int sum = 17, count = 5;
   double mean;

   mean = (double) sum / count;
   printf("Value of mean : %f\n", mean);
   mean = sum / count;
   printf("Value of mean : %f\n", mean);
}
```

When the above code is compiled and executed, it will produce the following result:

```
Value of mean : 3.400000
Value of mean : 3.000000
```

It is important to note that the precedence of the type casting operator is higher than that of division, so the value of `sum` is first cast to `double` and then divided by `count`, resulting in a value of type `double`.

Type casting can be implicit, performed automatically by the compiler, or explicit, specified by using the type casting operator. It is a good programming practice to use explicit type casting when type conversion is needed.

4.5.1 Integer Promotion

Integer promotion is the process of converting an integer type that is smaller than `int` or `unsigned int` to `int` or `unsigned int`. Consider the following example where a character is added to an `int`:

```c
#include <stdio.h>

int main() {
   int i = 17;
   char c = 'c'; /* ASCII value is 99 */
   int sum;

   sum = i + c;
   printf("Value of sum : %d\n", sum);
}
```

When the above code is compiled and executed, it will produce the following result:

```
Value of sum : 116
```

Here, the value of `sum` is 116 because the compiler performed integer promotion, converting the value of `'c'` to its corresponding ASCII value before performing the actual addition.

4.5.2 Usual Arithmetic Conversions

Usual arithmetic conversions are implicit conversions that are applied to operands to bring them to a common type. The compiler first performs integer promotion. If the operand types are different, they are converted to the highest-ranking type in the following hierarchy:

- `long double`
- `double`
- `float`
- `unsigned long`
- `long`
- `unsigned int`
- `int`

Usual arithmetic conversions do not apply to the assignment operator, or to the logical operators `&&` and `||`. Let's look at the following example to understand this concept:

```c
#include <stdio.h>

int main() {
   int i = 17;
   char c = 'c'; /* ASCII value is 99 */
   float sum;

   sum = i + c;
   printf("Value of sum : %f\n", sum);
}
```

When the above code is compiled and executed, it will produce the following result:

```
Value of sum : 116.000000
```

Here, `c` is first converted to an integer, but since the final value is of type `float`, the usual arithmetic conversions are applied. The compiler converts both `i` and `c` to `float` and adds them together to obtain a floating-point result.

5 Variable Definition

Variable definition tells the compiler where and how much storage to create for the variable. It specifies a data type and includes a list of one or more variables, as shown below:
`type variable_list;`
Here, `type` refers to the data type of the variable, which can be integer, floating-point, character, pointer, or a user-defined object.
The `variable_list` can consist of one or more variable names separated by commas. Variables are made up of letters, digits, and underscores, starting with a letter or an underscore.

```c
int age; //Defining an Integer Variable
float salary; //Defining a Floating-Point Variable
char grade; //Defining a Character Variable
int *ptr; //Defining a Pointer Variable
int i, j, k; //Defining Multiple Variables
```

5.1 Variable Initialization

In C, variable initialization is assigning an initial value to a variable at the time of definition. Initialization can be done during definition or later in the code. The initializer consists of an equal sign followed by a constant expression:
`type variable_name = value;`
Here, `type` represents the data type, `variable_name` is the variable's name, and `value` is the initial value.

```
int x = 10;         // Integer variable x initialized to 10
float pi = 3.14;    // Floating-point variable pi initialized to 3.14
char ch = 'A';      // Character variable ch initialized to 'A'
int d = 3, f = 5;   // Initialize d and f
byte z = 22;        // Initialize z
```

5.2 Later Initialization

Variables can be initialized later using the assignment operator `=`.

```
int x;          // Define integer variable x
x = 20;         // Initialize x to 20
float pi;       // Define floating-point variable pi
pi = 3.14159;   // Initialize pi to 3.14159
char ch;        // Define character variable ch
ch = 'B';       // Initialize ch to 'B'
```

Note: Variables should be initialized before use. Uninitialized variables have undefined values and may contain garbage. To avoid unpredictable behavior, always initialize variables before use.

5.3 Uninitialized Variables

In C, the default value of an uninitialized variable depends on its type and scope. Global variables and static variables have a default value of zero.
Default values for different types:

- Integer variables (`int`, `short`, `long`, etc.): Default value is 0.
- Floating-point variables (`float`, `double`, etc.): Default value is 0.0.
- Character variables (`char`): Default value is `'\0'`, the null character.
- Pointer variables: Default value is `NULL`, indicating the pointer does not point to a valid memory address.
- Composite types (arrays, structures, unions): Their elements or members are initialized according to the default rules.

Note: Local variables (non-static variables defined inside functions) do not automatically get initialized to default values. Their initial values are undefined (contain garbage). Therefore, you should explicitly assign an initial value before use.

```
int a;
int main()
{
    int b;
    printf("a's value is %d\n",a);
    printf("b's value is %d\n",b);
    return 0;

}
```

```
a's value is 0
b's value is 69000288
```

5.4 #define

`#define` allows you to define a constant that is replaced by its corresponding value at compile time.

```
#define CONSTANT_NAME CONSTANT_VALUE
```

Example:

```
#define PI 3.14159
```

In the program, the compiler will replace all instances of `PI` with `3.14159`.
Example:

```c
#include <stdio.h>

#define LENGTH 10
#define WIDTH  5
#define NEWLINE '\n'

int main() {
   int area;
   area = LENGTH * WIDTH;
   printf("value of area : %d", area);
   printf("%c", NEWLINE);
   return 0;
}
```

```
value of area : 50
```

5.5 Using the `const` Keyword

You can declare constants (read-only variable) of a specified type using the `const` keyword:

```
const type CONSTANT_NAME = CONSTANT_VALUE;
```

Example:

```c
const int MAX_VALUE = 100;
```

The value of `MAX_VALUE` will always be `100` and cannot be modified.
Example:

```c
#include <stdio.h>

int main() {
   const int LENGTH = 10;
   const int WIDTH  = 5;
   const char NEWLINE = '\n';
   int area;
   area = LENGTH * WIDTH;
   printf("value of area : %d", area);
   printf("%c", NEWLINE);
   return 0;
}
```

```
value of area : 50
```

5.6 Difference Between `#define` and `const`

- **Replacement Mechanism:** `#define` is a text substitution, formally called macro, while `const` declares a typed read-only variable.
- **Type Checking:** `#define` does not involve type checking, while `const` does.
- **Scope:** `#define` constants have no scope restrictions, whereas `const` constants have block scope.

`#define` is a preprocessor directive, while `const` is a variable definition. The macro defined by `#define` is expanded during preprocessing, while the `const` variable is used during compilation and runtime.
In conclusion, `const` defines variables with a specific type, allowing type safety checks, whereas `#define` only performs textual substitution without any type information, which can lead to issues like "macro pitfalls" or "parenthesis problems.", which will not be discussed in this class.

6 Conditional Statements

Conditional statements require the programmer to specify one or more conditions to evaluate or test. Additionally, they specify statements to execute if the condition is true (mandatory) and optionally, statements to execute if the condition is false. In C, any non-zero and non-null value is considered `true`, while zero or null is considered `false`.

- **if statement** An `if` statement consists of a boolean expression followed by one or more statements.

```c
//if(boolean_expression)
//{
//   /* Statements to execute if the boolean expression is true */
//}

#include <stdio.h>

int main ()
{
   int a = 10;
   if( a < 20 )
   {
       printf("a is smaller than 20\n" );
   }
   printf("a's value is %d\n", a);

   return 0;
}
```

- **if...else statement** An `if` statement can be followed by an optional `else` statement, which executes when the boolean expression is false.

```c
//if(boolean_expression){
//    /* Statements to execute if the boolean expression is true */
//}
//else {
//    /* Statements to execute if the boolean expression is false */
//}

#include <stdio.h>

int main () {
   /* Local variable definition */
   int a = 100;

   /* Check boolean condition */
   if( a < 20 ) {
       /* If condition is true, print the following */
       printf("a is less than 20\n" );
   } else {
       /* If condition is false, print the following */
       printf("a is greater than 20\n" );
   }
   printf("Value of a is %d\n", a);

   return 0;
}
```

- **Nested if statement**: You can use an `if` or `else if` statement inside another `if` or `else if` statement.

```c
//if( boolean_expression 1) {
//  /* Executes when boolean expression 1 is true */
//   if(boolean_expression 2){
//      /* Executes when boolean expression 2 is true */
//   }
//}

#include <stdio.h>

int main () {
   /* Local variable definition */
   int a = 100;
   int b = 200;

   /* Check boolean condition */
   if( a == 100 ) {
       /* If condition is true, check the following condition */
       if( b == 200 ) {
          /* If condition is true, print the following */
          printf("a is 100, and b is 200\n" );
       }
   }
   printf("Exact value of a is %d\n", a );
   printf("Exact value of b is %d\n", b );

   return 0;
}
```

- **switch statement**: A `switch` statement allows testing a variable against multiple values. A `switch` statement tests a variable against multiple values, each called a `case`, and executes the corresponding block.

  The syntax of a `switch` statement in C:

```
switch(expression) {
    case constant-expression:
       statement(s);
       break; /* Optional */
    case constant-expression:
       statement(s);
       break; /* Optional */

    /* You can have any number of case statements */
    default: /* Optional */
       statement(s);
}
```

  - The expression following `switch` is compared with the constant values of each `case` until a match is found or until the `default` case (if present) is reached.
  - If a match is found, the corresponding block is executed, and the `switch` statement is exited.
  - If no match is found and a `default` case exists, the code block following `default` is executed.
  - If no match is found and there is no `default` case, the `switch` statement is skipped entirely.

  Rules for `switch` Statements:

  - The expression in `switch` must be of integer type (e.g., `char`, `short`, `int`, or enum).
  - Each `case` label must be unique within the `switch` statement.
  - The `default` label is optional and provides a block of code to execute if no `case` matches.
  - The `case` label must be a constant value, not a variable or expression.
  - The order of `case` labels does not matter.
  - The `break` statement is typically used to terminate the execution of a `case` block. Without `break`, the program will continue to execute the following `case` blocks.
  - `switch` statements can be nested within other `switch` statements, but this can make the code harder to read and understand.

  Example:

```
#include <stdio.h>

int main () {
   /* Local variable definition */
   char grade = 'B';

   switch(grade) {
   case 'A' :
      printf("Excellent!\n" );
      break;
   case 'B' :
   case 'C' :
      printf("Well done\n" );
      break;
   case 'D' :
      printf("You passed\n" );
      break;
   case 'F' :
      printf("Better try again\n" );
      break;
   default :
      printf("Invalid grade\n" );
   }
   printf("Your grade is %c\n", grade );

   return 0;
}
```

7 Loop Statements

Loop statements allow us to execute a statement or a group of statements multiple times.

7.1 Loops

7.1.1 while loop

`while` Loop: Repeats a statement or a group of statements while a given condition is true. It tests the condition before executing the loop body. The syntax of the `while` loop in C language:

```
while(condition)
{
   statement(s);
}
```

Here, `statement(s)` can be a single statement or a block of code consisting of multiple statements. `condition` can be any expression and is true when it evaluates to any non-zero value. The loop executes as long as the condition is true. When the condition becomes false, the loop exits, and the program flow continues with the next statement immediately following the loop.

The key point about the `while` loop in C is that the loop may not execute at all. If the condition is false, the loop body is skipped, and the program flow directly continues with the next statement following the `while` loop.

```c
#include <stdio.h>

int main ()
{
   /* Local variable definition */
   int a = 10;

   /* while loop execution */
   while( a < 20 )
   {
      printf("The value of a: %d\n", a);
      a++;
   }

   return 0;
}
```

The output will be:

```
The value of a: 10
The value of a: 11
The value of a: 12
The value of a: 13
The value of a: 14
The value of a: 15
The value of a: 16
The value of a: 17
The value of a: 18
The value of a: 19
```

7.1.2 for loop

`for` Loop: Repeats a sequence of statements multiple times and simplifies the code that manages the loop variable. The syntax of the `for` loop in C language:

```
for ( init; condition; increment )
{
   statement(s);
}
```

Below is the control flow of the `for` loop:

- `init` is executed first and only once. This step allows you to declare and initialize any loop control variables. You can also leave this step empty, as long as a semicolon appears.
- Next, the `condition` is evaluated. If it is true, the loop body is executed. If it is false, the loop body is not executed, and the control flow jumps to the next statement following the `for` loop.
- After executing the loop body, the control flow jumps back to the `iteration` statement. This statement allows you to update the loop control variable. This step can also be left empty, as long as a semicolon appears after the condition.

- The condition is evaluated again. If it is true, the loop continues to execute—loop body, increment, and then condition evaluation again. When the condition becomes false, the `for` loop terminates.

```c
#include <stdio.h>

int main ()
{
   /* for loop execution */
   for( int a = 10; a < 20; a = a + 1 )
   {
      printf("The value of a: %d\n", a);
   }

   return 0;
}
```

The output will be:

```
The value of a: 10
The value of a: 11
The value of a: 12
The value of a: 13
The value of a: 14
The value of a: 15
The value of a: 16
The value of a: 17
The value of a: 18
The value of a: 19
```

In the iteration section, you can include multiple expressions separated by commas. These expressions are evaluated in left-to-right order, and they all execute as part of the loop's iteration step.Another example:

```c
#include <stdio.h>

int main()
{
    for(int i = 0, j = 0; i < 10; ++i, ++j){
        printf("%d %d\n", i, j);
    }

    return 0;
}
```

7.1.3 do...while Loop

`do...while` Loop: Unlike `for` and `while` loops, which test the loop condition at the beginning of the loop, the `do...while` loop in C language checks its condition at the end of the loop.
The syntax of the `do...while` loop in C language:

```
do
{
   statement(s);
}while( condition );
```

Note that the condition expression appears at the end of the loop, so the `statement(s)` inside the loop will be executed at least once before the condition is tested. If the condition is true, the control flow jumps back to the `do` statement and the `statement(s)` inside the loop are executed again. This process repeats until the given condition becomes false.

```c
#include <stdio.h>

#include <stdio.h>

int main() {
    int correctNumber = 7; // Suppose the correct number is 7
    int guess;

    // Using a do-while loop to repeatedly prompt the user for a guess
    do {
        printf("Guess the number (1 to 10): ");
        scanf("%d", &guess);

        if (guess != correctNumber) {
            printf("Incorrect, try again!\n");
```

```
        }

    } while (guess != correctNumber);

    printf("Congratulations! You guessed the correct number.\n");
    return 0;
}
```

7.1.4 Nested Loops

Nested Loops: The C language allows the use of one loop inside another loop. Below are some examples demonstrating this concept.

```
#include <stdio.h>

int main ()
{
   /* Local variable definition */
   int i, j;

   for(i=2; i<10; i++) {
      for(j=2; j <= (i/j); j++)
        if(!(i%j)) break; // If a divisor is found, it is not a prime number
      if(j > (i/j)) printf("%d is a prime number\n", i);
   }

   return 0;
}
```

The output will be:

```
2 is a prime number
3 is a prime number
5 is a prime number
7 is a prime number
```

```
#include <stdio.h>
int main()
{
    int i=1,j;
    while (i <= 5)
    {
        j=1;
        while (j <= i )
        {
            printf("%d ",j);
            j++;
        }
        printf("\n");
        i++;
    }
    return 0;
}
```

The output will be:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

7.2 Loop Control Statements

Loop control statements change the execution sequence of your code. Through them, you can implement code jumps.

7.2.1 break

`break` Statement: Terminates the loop or `switch` statement, and the program flow continues with the next statement immediately following the loop or `switch`. If you are using nested loops (i.e., one loop inside another loop), the `break` statement will stop the execution of the innermost loop, and then the control will move to the next line of code after that loop block.

```c
#include <stdio.h>

int main ()
{
   /* Local variable definition */
   int a = 10;

   /* while loop execution */
   while( a < 20 )
   {
      printf("The value of a: %d\n", a);
      a++;
      if( a > 15)
      {
         /* terminate the loop using break statement */
          break;
      }
   }

   return 0;
}
```

When the above code is compiled and executed, it produces the following results:

```
The value of a: 10
The value of a: 11
The value of a: 12
The value of a: 13
The value of a: 14
The value of a: 15
```

7.2.2 continue

`continue` Statement: Causes the loop to immediately stop its current iteration and start the next iteration of the loop. In the case of a `for` loop, after the `continue` statement is executed, the increment statement is still executed. For `while` and `do...while` loops, the `continue` statement causes the condition to be re-evaluated.

```c
#include <stdio.h>

int main ()
{
   /* Local variable definition */
   int a = 10;

   /* do loop execution */
   do
   {
      if( a == 15)
      {
         /* skip the iteration */
         a = a + 1;
         continue;
      }
      printf("The value of a: %d\n", a);
      a++;

   }while( a < 20 );

   return 0;
}
```

The result will be:

```
The value of a: 10
The value of a: 11
The value of a: 12
The value of a: 13
The value of a: 14
The value of a: 16
The value of a: 17
The value of a: 18
The value of a: 19
```

7.2.3 goto

`goto` Statement: Transfers control to the labeled statement. However, it is not recommended to use `goto` statements in your programs.

Note: The use of `goto` statements is generally discouraged in any programming language. It makes the program's control flow difficult to follow, which can make the code harder to understand and maintain. Any program that uses `goto` can be rewritten without it. The syntax of the `goto` statement in C language:

```
goto label;
..
.
label: statement;
```

Here, `label` can be any plain text identifier, except for C keywords, and it can be placed before or after the `goto` statement in the C program.

```c
#include <stdio.h>

int main ()
{
   /* Local variable definition */
   int a = 10;

   /* do loop execution */
   LOOP:do
   {
      if( a == 15)
      {
         /* skip the iteration */
         a = a + 1;
         goto LOOP;
      }
      printf("The value of a: %d\n", a);
      a++;

   }while( a < 20 );

   return 0;
}
```

The output will be:

```
The value of a: 10
The value of a: 11
The value of a: 12
The value of a: 13
The value of a: 14
The value of a: 16
The value of a: 17
The value of a: 18
The value of a: 19
```

This label can be either before or after the goto statement.

```c
#include <stdio.h>

int main() {
    int x = 0;

    // Example where goto is before the label
    goto label1;  // Jumps forward to 'label1'

    // This part of the code is skipped by the goto above
    printf("This line will be skipped.\n");

label1:
    printf("Jumped to label1.\n");

    x++;

    if (x < 2) {
        // Example where goto is after the label
        goto label2;  // Jumps backward to 'label2'
```

```
    }

    return 0;

label2:
    printf("Jumped to label2.\n");
    goto label1;  // Jumps back to 'label1'
}
```

The output will be:

```
Jumped to label1.
Jumped to label2.
Jumped to label1.
```

7.3 Infinite Loops

If the condition never becomes false, the loop will become an infinite loop. The `for` loop can traditionally be used to implement an infinite loop. Since none of the three expressions that form the loop are required, you can leave some conditional expressions empty to form an infinite loop.

```
#include <stdio.h>

int main ()
{
   for( ; ; )
   {
      printf("This loop will run forever!\n");
   }
   return 0;
}
```

When the condition expression is absent, it is assumed to be true. You can also set an initial value and increment expression, but in general, C programmers tend to use the `for(;;)` structure to represent an infinite loop.
Note: You can terminate an infinite loop by pressing `Ctrl + C`.

```

#include <stdio.h>

int main() {
    int choice;

    // Infinite loop using while (1)
    while (1) {
        // Display menu options
        printf("Menu:\n");
        printf("1. Option 1\n");
        printf("2. Option 2\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");

        // Read user input
        scanf("%d", &choice);

        // Process the user's choice
        switch (choice) {
            case 1:
                printf("You selected Option 1.\n");
                break;
            case 2:
                printf("You selected Option 2.\n");
                break;
            case 3:
                printf("Exiting the program.\n");
                return 0; // Proper exit from the program
            default:
                printf("Invalid choice! Please try again.\n");
                break;
        }
    }

    return 0;
}
```

This pattern is commonly used in menu-driven programs, interactive applications, and scenarios where the program needs to keep running until a specific exit condition is met.

8 C Pointers

Memory addressing refers to the process by which a computer's CPU accesses data stored in its memory. Memory can be thought of as a large array of bytes, where each byte (memory block) has a unique address (similar to a house having a unique postal address). The CPU uses these addresses to locate and access the data it needs to execute programs.

As you know, every variable has a memory location, and each memory location is defined by an address that can be accessed using the `&` operator, which represents an address in memory.
Consider the following example, which will output the address of a defined variable:

```c
#include <stdio.h>

int main ()
{
    int num = 10;
    int *p;               // Define a pointer variable
    p = &num;

    printf("The address of num: %p\n", p);
    return 0;
}
```

The output will be:

```
The address of num: 0x16ae67588
```

Through the above example, we learned what a memory address is and how to access it. Next, let's look at what a pointer is.

8.1 What is a Pointer?

A pointer is a memory address, and a pointer variable is a variable used to store a memory address. Like other variables or constants, you must declare a pointer before using it to store the address of another variable. The general form of a pointer variable declaration is:

`type *var_name;`

Here, `type` is the base type of the pointer; it must be a valid C data type, and `var_name` is the name of the pointer variable. The asterisk (`*`) used to declare a pointer is the same as the one used in multiplication, but in this statement, the asterisk specifies that the variable is a pointer. The following are valid pointer declarations:

```c
int    *ip;    /* A pointer to an integer */
double *dp;    /* A pointer to a double */
float  *fp;    /* A pointer to a float */
char   *ch;    /* A pointer to a char */
```

All actual data types, whether they are integers, floating-point numbers, characters, or other data types, have the same type for their corresponding pointer values, which is a long hexadecimal number representing a memory address.

8.2 How to Use Pointers?

When using pointers, the following operations are frequently performed: defining a pointer variable, assigning the address of a variable to the pointer, and accessing the value available at the address in the pointer variable. These are accomplished using the unary operator `*`, which returns the value of the variable located at the address specified by the operand. The following example involves these operations:

```c
#include <stdio.h>

int main ()
{
   int  var = 20;   /* Actual variable declaration */
   int  *ip;        /* Pointer variable declaration */

   ip = &var;  /* Store the address of var in the pointer variable */

   printf("The address of var: %p\n", &var);

   /* Address stored in the pointer variable */
   printf("The address stored in ip: %p\n", ip);

   /* Access the value using the pointer */
   printf("The value of *ip: %d\n", *ip);

   return 0;
}
```

The output will be:

```
The address of var: 0x16f93f588
The address stored in ip: 0x16f93f588
The value of *ip: 20
```

8.3 NULL Pointers

When declaring a variable, if there is no specific address to assign, it is a good programming practice to assign a `NULL` value to the pointer variable. A pointer assigned with a `NULL` value is called a null pointer. A `NULL` pointer is a constant defined in the standard library with a value of zero. Consider the following program:

```c
#include <stdio.h>

int main ()
{
   int  *ptr = NULL;

   printf("The address in ptr is %p\n", ptr);

   return 0;
}
```

The output will be:

```
The address in ptr is 0x0
```

In most operating systems, programs are not allowed to access memory at address 0, as this memory is reserved by the operating system. However, the memory address 0 has special significance; it indicates that the pointer does not point to a valid memory location. By convention, if a pointer contains a null value (zero), it is assumed not to point to anything. To check for a null pointer, you can use an `if` statement as follows:

```c
if(ptr)     /* If ptr is not null, proceed */
if(!ptr)    /* If ptr is null, proceed */
```

8.4 Pointer Arithmetic in C

Pointers in C are addresses represented by numeric values. Therefore, you can perform arithmetic operations on pointers. There are four types of arithmetic operations that can be performed on pointers: `++`, `--`, `+`, and `-`. Suppose `ptr` is an integer pointer pointing to the address 1000, which is a 32-bit integer. Let's perform the following arithmetic operation on the pointer:

```
ptr++
```

After executing the above operation, `ptr` will point to the location 1004 because each increment of `ptr` will move it to the next integer location, which is 4 bytes ahead. This operation moves the pointer to the next memory location without affecting the actual value stored at the memory location. If `ptr` points to a character at address 1000, the above operation will cause the pointer to point to 1001, as the next character location is at 1001.

- Each time a pointer is incremented, it actually points to the storage unit of the next element.
- Each time a pointer is decremented, it points to the storage unit of the previous element.
- The number of bytes a pointer jumps during increment or decrement depends on the data type length of the variable the pointer points to. For example, `int` is 4 bytes.
- When you subtract two pointers, C computes the difference in terms of the number of elements between them, not the raw memory addresses.

8.5 Pointer Comparison

In C, pointers can be compared to determine their relationship. Pointer comparisons are primarily used to check if two pointers point to the same memory location or to determine if one pointer is before or after another pointer. Pointers can be compared using relational operators such as `==`, `!=`, `<`, `>`, `<=`, and `>=`. If `p1` and `p2` point to two related variables, such as different elements in the same array, you can compare `p1` and `p2`.

```c
#include <stdio.h>

int main() {
    int a = 5;
    int b = 10;
    int *ptr1 = &a;
    int *ptr2 = &a;
    int *ptr3 = &b;

    if (ptr1 == ptr2) {
        printf("ptr1 and ptr2 point to the same memory address\n");  // This line will be printed
    } else {
        printf("ptr1 and ptr2 point to different memory addresses\n");
    }

    if (ptr1 != ptr3) {
        printf("ptr1 and ptr3 point to different memory addresses\n");  // This line will be printed
    } else {
        printf("ptr1 and ptr3 point to the same memory address\n");
    }

    return 0;
}
```

The output will be:

```
ptr1 and ptr2 point to the same memory address
ptr1 and ptr3 point to different memory addresses
```

8.6 Pointer to a Pointer

A pointer to a pointer is a form of multiple levels of indirection, or a chain of pointers. Typically, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, and the second pointer points to the location that holds the actual value.
A pointer to a pointer variable must be declared with two asterisks placed before the variable name. For example, the following declares a pointer to a pointer of type `int`:

```
int **var;
```

When a target value is indirectly pointed to by one pointer to another pointer, accessing this value requires using the double asterisk operator, as shown in the following example:

```c
#include <stdio.h>

int main ()
{
   int  V;
   int  *Pt1;
   int  **Pt2;

   V = 100;

   /* Get the address of V */
   Pt1 = &V;

   /* Use the & operator to get the address of Pt1 */
   Pt2 = &Pt1;

   /* Access the value using Pt2 */
   printf("var = %d\n", V );
   printf("Pt1 = %p\n", Pt1 );
   printf("*Pt1 = %d\n", *Pt1 );
   printf("Pt2 = %p\n", Pt2 );
   printf("**Pt2 = %d\n", **Pt2);

   return 0;
}
```

The output will be:

```
var = 100
Pt1 = 0x7ffee2d5e8d8
*Pt1 = 100
Pt2 = 0x7ffee2d5e8d0
**Pt2 = 100
```

9 C Functions

A function is a group of statements that together perform a task. Every C program has at least one function, the main function `main()`, and all the most trivial programs can define additional functions.

You can divide your code into different functions. How you divide your code into different functions is up to you, but logically, the division is typically based on each function performing a specific task.

A function declaration tells the compiler the function's name, return type, and parameters. The function definition provides the actual body of the function.

The C standard library provides a large number of built-in functions that programs can call. For example, the function `strcat()` is used to concatenate two strings, and the function `memcpy()` is used to copy memory from one location to another.

9.1 Defining a Function

The general form of a function definition in C language is as follows:

```
return_type function_name( parameter list )
{
   body of the function
}
```

In C language, a function consists of a function header and a function body. Below are all the components of a function:

- **Return Type:** A function may return a value. The `return_type` is the data type of the value the function returns. Some functions perform the required operations without returning a value. In this case, the `return_type` is the keyword `void`.
- **Function Name:** This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters:** Parameters are like placeholders. When a function is invoked, you pass a value to the parameter. This value is called the actual parameter or argument. The parameter list refers to the type, order, and number of parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body:** The function body contains a collection of statements that define what the function does.

Here is the source code of the `max()` function. This function has two parameters, `num1` and `num2`, and it returns the larger of the two:

```
/* Function returns the larger of two numbers */
int max(int num1, int num2)
{
   /* Local variable declaration */
   int result;

   if (num1 > num2) {
      result = num1;
   } else {
      result = num2;
   }
   return result;
}
```

9.2 Function Declaration

A function declaration tells the compiler the function's name and how to call the function. The function's actual body can be defined separately. A function declaration has the following parts:

```
return_type function_name( parameter list );
```

For the function `max()` defined above, the function declaration would be:

```
int max(int num1, int num2);
```

In the function declaration, the parameter names are not important, only the type of the parameters is required. Therefore, the following is also a valid declaration:

```
int max(int, int);
```

When you define a function in one source file and call it in another file, the function declaration is necessary. In such cases, you should declare the function at the top of the file where the function is called.

9.3 Calling a Function

When creating a C function, you define what the function does, and then you call the function to accomplish the defined task.

When a program calls a function, control of the program is transferred to the called function. The called function performs its defined task, and when its `return` statement is executed or when it reaches the closing brace, control is passed back to the main program.

When calling a function, pass the required parameters, and if the function returns a value, you can store the return value. For example:

```c
#include <stdio.h>
/* Function declaration */
int max(int num1, int num2);

int main ()
{
   /* Local variable definition */
   int a = 100;
   int b = 200;
   int ret;

   /* Call the function to get the maximum value */
   ret = max(a, b);

   printf("Max value is: %d\n", ret);

   return 0;
}

/* Function returns the larger of two numbers */
int max(int num1, int num2)
{
   /* Local variable declaration */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

The output will be:

```
Max value is: 200
```

9.4 Function Parameters

If a function is to use parameters, it must declare variables that accept the values of the parameters. These variables are called the function's formal parameters. Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit. When calling a function, there are two ways to pass parameters to the function:

- **Call by Value**: This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

  By default, C uses the call by value method to pass arguments. Generally, this means that the code inside a function cannot alter the actual arguments used to call the function. The `swap()` function is defined as follows:

```c
/* Function definition */
void swap(int x, int y)
{
```

```
   int temp;

   temp = x; /* Save the value of x */
   x = y;    /* Assign the value of y to x */
   y = temp; /* Assign the value of temp to y */

   return;
}

```

Now, let's call the `swap()` function by passing actual arguments:

```
#include <stdio.h>

/* Function declaration */
void swap(int x, int y);

int main ()
{
   /* Local variable definition */
   int a = 100;
   int b = 200;

   printf("Before swapping, the value of a: %d\n", a );
   printf("Before swapping, the value of b: %d\n", b );

   /* Call the function to swap the values */
   swap(a, b);

   printf("After swapping, the value of a: %d\n", a );
   printf("After swapping, the value of b: %d\n", b );

   return 0;
}
```

The output will be:

```
Before swapping, the value of a: 100
Before swapping, the value of b: 200
After swapping, the value of a: 100
After swapping, the value of b: 200
```

The above example shows that although the values of `a` and `b` are changed within the function, the actual values of `a` and `b` remain unchanged.

- **Call by Reference**: This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

```
/* Function definition */
void swap(int *x, int *y)
{
   int temp;
   temp = *x;    /* Save the value at address x */
   *x = *y;      /* Assign the value of y to x */
   *y = temp;    /* Assign the value of temp to y */

   return;
}
```

Now, let's call the `swap()` function using call by reference:

```
#include <stdio.h>

/* Function declaration */
void swap(int *x, int *y);

int main ()
{
   /* Local variable definition */
   int a = 100;
   int b = 200;

   printf("Before swapping, the value of a: %d\n", a );
   printf("Before swapping, the value of b: %d\n", b );

   /* Call the function to swap the values
```

```
     * &a represents the pointer to a, i.e., the address of variable a
     * &b represents the pointer to b, i.e., the address of variable b
    */
    swap(&a, &b);

    printf("After swapping, the value of a: %d\n", a );
    printf("After swapping, the value of b: %d\n", b );

    return 0;
}
```

The output will be:

```
Before swapping, the value of a: 100
Before swapping, the value of b: 200
After swapping, the value of a: 200
After swapping, the value of b: 100
```

The above example demonstrates that, unlike call by value, call by reference changes the values of `a` and `b` inside the function and also affects the values of `a` and `b` outside the function.

By default, C uses call by value to pass arguments. In general, this means that the code within a function cannot alter the arguments used to call the function.

9.5 Function Pointers

A function pointer is a pointer variable that points to a function. Typically, when we talk about pointer variables, we refer to pointers that point to variables of types such as integers, characters, or arrays. However, function pointers point to functions. Function pointers can be used to call functions and pass parameters, just like regular functions.

To declare a function pointer, you need to specify the return type, use the * symbol, provide the pointer's name (enclosed in parentheses), and then provide the parameter list that matches the function signature.

`return_type (*pointer_name)(parameter_types);`

In C, the function name itself is already considered a pointer to the function's address.

The following example declares a function pointer variable `p` that points to the function `max`:

```
#include <stdio.h>

int max(int x, int y)
{
    return x > y ? x : y;
}

int main(void)
{
    /* p is a function pointer */
    int (* p)(int, int) = & max; // The & can be omitted
    int a, b, c, d;

    printf("Please enter three numbers: ");
    scanf("%d %d %d", &a, &b, &c);

    /* Equivalent to calling the function directly, d = max(max(a, b), c) */
    d = p(p(a, b), c);

    printf("The largest number is: %d\n", d);

    return 0;
}
```

The output will be:

```
Please enter three numbers: 1 2 3
The largest number is: 3
```

9.6 Callback Functions

Function pointer variables can be used as parameters for other functions. A callback function is a function that is invoked through a function pointer. Simply put, a callback function is a function that is called by another function

when it is executed.

In the example, the `populate_array()` function defines three parameters, with the third parameter being a pointer to a function that is used to set the values of the array.

In the example, we define a callback function `getNextRandomValue()`, which returns a random value and is passed as a function pointer to the `populate_array()` function.

`populate_array()` will call the callback function 10 times and assign the callback function's return values to the array.

In C, when passing a function as a callback (i.e., when assigning a function to a function pointer or passing it as an argument), you do not need to use the & (address-of) operator because the function name itself is already considered a pointer to the function's address.

```c
#include <stdlib.h>
#include <stdio.h>

void populate_array(int *array, size_t arraySize, int (*getNextValue)(void))
{
    for (size_t i = 0; i < arraySize; i++)
        array[i] = getNextValue();
}

// Get a random value
int getNextRandomValue(void)
{
    return rand();
}

int main(void)
{
    int myarray[10];
    /* Do not add parentheses to getNextRandomValue,
       otherwise it won't compile because adding parentheses
       would pass an int instead of a function pointer */
    populate_array(myarray, 10, getNextRandomValue);
    for(int i = 0; i < 10; i++) {
        printf("%d ", myarray[i]);
    }
    printf("\n");
    return 0;
}
```

The output will be:

```
16807 282475249 1622650073 984943658 1144108930 470211272 101027544 1457850878 1458777923 2007237709
```

9.7 Why Use Callback Functions?

Callback functions are used in scenarios where a function needs to perform different operations based on the situation or input. Instead of writing multiple versions of the same function with slight variations, you can write one function that accepts a callback, allowing you to define the specific operation externally. This makes your code more modular, reusable, and easier to maintain.

The example below shows we only need to write one quick sort function which can do ascending sort and descending sort based on the different callback functions.

```c
#include <stdio.h>
#include <stdlib.h>

// Callback function to compare two integers for ascending order
int compareAscending(const void *a, const void *b) { // match the qsort function.
    return (*(int*)a - *(int*)b);
}

// Callback function to compare two integers for descending order
int compareDescending(const void *a, const void *b) {
    return (*(int*)b - *(int*)a);
}

// Function to print the array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
```

```c
    }
    printf("\n");
}

// Function to sort the array using a callback function for comparison
void sortArray(int arr[], int size, int (*compare)(const void*, const void*)) {
    qsort(arr, size, sizeof(int), compare);
}

int main() {
    int arr[] = {3, 1, 4, 1, 5, 9, 2, 6, 5};
    int size = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: \n");
    printArray(arr, size);

    // Sort the array in ascending order
    sortArray(arr, size, compareAscending);
    printf("Array sorted in ascending order: \n");
    printArray(arr, size);

    // Sort the array in descending order
    sortArray(arr, size, compareDescending);
    printf("Array sorted in descending order: \n");
    printArray(arr, size);

    return 0;
}
```

9.8 C Variable Number of Arguments

Sometimes, you may encounter situations where you want a function to accept a variable number of arguments rather than a predefined number of arguments. C provides a solution for this, allowing you to define a function that can accept a variable number of arguments as needed.

```c
int func_name(int arg1, ...);
```

Here, the ellipsis `...` represents a variable argument list.

```c
int func(int, ... )  {
   // function implementation
}

int main() {
   func(2, 2, 3);
   func(3, 2, 3, 4);
}
```

Notice that the last parameter of the function `func()` is written as an ellipsis, which is three dots (`...`). The parameter before the ellipsis is of type `int` and represents the total number of variable arguments to be passed. To use this feature, you need to include the `stdarg.h` header file, which provides the functions and macros needed to implement variable arguments. The steps are as follows:

1. Define a function with the last parameter as an ellipsis. Custom parameters can be set before the ellipsis.
2. In the function definition, create a variable of type `va_list`, which is defined in the `stdarg.h` header file.
3. Use the `va_start()` macro with an `int` parameter to initialize the `va_list` variable as a list of arguments. The `va_start()` macro is defined in the `stdarg.h` header file.
4. Use the `va_arg()` macro and the `va_list` variable to access each item in the argument list.
5. Use the `va_end()` macro to clean up the memory allocated to the `va_list` variable.

Common macros include:

- `va_start(ap, last_arg)`: Initializes the variable argument list. `ap` is a variable of type `va_list`, and `last_arg` is the name of the last fixed argument (i.e., the argument before the variable argument list). This macro sets `ap` to point to the first argument in the variable argument list.
- `va_arg(ap, type)`: Retrieves the next argument in the variable argument list. `ap` is a variable of type `va_list`, and `type` is the type of the next argument. This macro returns a value of type `type` and moves `ap` to the next argument.

- `va_end(ap)`: Ends the traversal of the variable argument list. `ap` is a variable of type `va_list`. This macro sets `ap` to `NULL`.

Now, let's write a function with a variable number of arguments that returns their average according to the steps above:

```c
#include <stdio.h>
#include <stdarg.h>

double average(int num, ...) {

    va_list valist;
    double sum = 0.0;
    int i;

    /* Initialize valist for num number of arguments */
    va_start(valist, num);

    /* Access all the arguments assigned to valist */
    for (i = 0; i < num; i++) {
       sum += va_arg(valist, int);
    }
    /* Clean up the memory reserved for valist */
    va_end(valist);

    return sum / num;
}

int main() {
   printf("Average of 2, 3, 4, 5 = %f\n", average(4, 2, 3, 4, 5));
   printf("Average of 5, 10, 15 = %f\n", average(3, 5, 10, 15));
}
```

In the example above, the `average()` function takes an integer `num` and an arbitrary number of integer arguments. Inside the function, the variable `valist` of type `va_list` is used to access the variable argument list. In the loop, the `va_arg()` macro is used to fetch the next integer argument and accumulate the sum. Finally, the `va_end()` macro is used to end the traversal of the variable argument list.

When the above code is compiled and executed, it will produce the following output. It should be noted that the `average()` function is called twice, and in each call, the first argument represents the total number of variable arguments passed. The ellipsis is used to pass a variable number of arguments.

```
Average of 2, 3, 4, 5 = 3.500000
Average of 5, 10, 15 = 10.000000
```

Note:in C, when you use a va_list to handle a variable number of arguments in a function, certain types undergo **default argument promotion**. Specifically, float arguments are promoted to double when passed to a variadic function (a function that takes a variable number of arguments, like those using ... and va_list). If you use va_arg to retrieve a float argument, the C standard does not define what happens because the function actually received a double. Thus, using va_arg to get a float instead of a double results in **undefined behavior**, which is why the compiler warns you.

10 C Scope

In any programming language, scope refers to the region of the program where a defined variable exists and beyond which the variable cannot be accessed. In C language, there are three places where variables can be declared:

- **Local variables** inside a function or block
- **Global variables** outside all functions
- **Formal parameters** in the function parameter definition

Let's look at what local variables, global variables, and formal parameters are.

10.1 Local Variables

Variables that are declared inside a function or block are called local variables. They can only be used by the statements inside that function or block. Local variables are not known outside the function. Here is an example of using local variables. In this example, all the variables `a`, `b`, and `c` are local to the `main()` function.

```c
#include <stdio.h>

int main ()
{
   /* Local variable declaration */
   int a, b;
   int c;

   /* Actual initialization */
   a = 10;
   b = 20;
   c = a + b;

   printf ("value of a = %d, b = %d and c = %d\n", a, b, c);

   return 0;
}
```

10.2 Global Variables

Global variables are defined outside of all functions, usually at the top of the program. Global variables are accessible throughout the program's lifecycle and can be accessed within any function. Global variables can be accessed by any function. That is, once a global variable is declared, it is available throughout the entire program. Here is an example using both global and local variables:

```c
#include <stdio.h>

/* Global variable declaration */
int g;

int main ()
{
   /* Local variable declaration */
   int a, b;

   /* Actual initialization */
   a = 10;
   b = 20;
   g = a + b;

   printf ("value of a = %d, b = %d and g = %d\n", a, b, g);

   return 0;
}
```

In a program, local and global variables can have the same name, but within a function, if the names are the same, the local variable's value is used, and the global variable is not accessed. Here is an example:

```c
#include <stdio.h>

/* Global variable declaration */
int g = 20;

int main ()
{
   /* Local variable declaration */
   int g = 10;

   printf ("value of g = %d\n",  g);

   return 0;
}
```

The output will be:

```
value of g = 10
```

10.3 Formal Parameters

The parameters of a function, also known as formal parameters, are treated as local variables within that function. If they share the same name as a global variable, the local variable takes precedence. Here is an example:

```c
#include <stdio.h>

/* Global variable declaration */
```

```c
int a = 20;

int main ()
{
  /* Local variable declaration in main function */
  int a = 10;
  int b = 20;
  int c = 0;
  int sum(int, int);

  printf ("value of a in main() = %d\n",  a);
  c = sum(a, b);
  printf ("value of c in main() = %d\n",  c);

  return 0;
}

/* Function to add two integers */
int sum(int a, int b)
{
    printf ("value of a in sum() = %d\n",  a);
    printf ("value of b in sum() = %d\n",  b);

    return a + b;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a in main() = 10
value of a in sum() = 10
value of b in sum() = 20
value of c in main() = 30
```

10.4 What is `static`?

`static` is a commonly used modifier in C/C++, employed to control the storage and visibility of variables.
We know that variables defined inside a function are allocated space on the stack by the compiler when the program executes to that point. The space allocated for a function on the stack is released when the function execution ends, which raises a problem: how can we preserve the value of a variable in a function until the next time it is called? The most straightforward solution is to define the variable as global, but defining a global variable has many drawbacks, the most obvious being that it disrupts the variable's access scope (i.e., a variable defined in a function would no longer be controlled solely by that function). The `static` keyword provides a good solution to this problem.

```c
#include <stdio.h>

// Function that demonstrates the use of a static variable
void counterFunction() {
    // Static variable is initialized only once and retains its value between function calls
    static int count = 0;

    // Increment the count each time the function is called
    count++;

    // Print the current value of count
    printf("Function called %d times\n", count);
}

int main() {
    // Call the function multiple times to demonstrate the static variable behavior
    counterFunction(); // Output: Function called 1 times
    counterFunction(); // Output: Function called 2 times
    counterFunction(); // Output: Function called 3 times

    return 0;
}
```

10.4.1 Static Global Variables

1. Global variables are global variables not explicitly modified by `static`. By default, global variables have external linkage, meaning they are accessible throughout the project. A global variable defined in one file can be used in another file by declaring it with `extern`.

```c
// file1.c
int count = 10;  // Definition of the variable

```

```c
// file2.c
extern int count;  // Declaration of the same variable
```

Here, file2.c is telling the compiler that there is an int variable called count that is defined elsewhere, specifically in file1.c. The linker will resolve the reference to count between these files.

2. Global static variables are global variables explicitly modified by `static`. Their scope is limited to the file in which they are declared, and they cannot be used in other files, even if declared with `extern`.

Static global variables have the following characteristics:

- Static variables are allocated memory in the global data area, including static local variables.
- Uninitialized static global variables are automatically initialized to 0 by the program.
- Static global variables are visible throughout the file in which they are declared but are invisible outside the file.

10.4.2 Static Local Variables

1. These variables are allocated memory in the global data area.
2. Static local variables are initialized the first time the program execution reaches the declaration of the object. Subsequent function calls do not reinitialize them.
3. Static local variables are typically initialized at the point of declaration. If not explicitly initialized, the program automatically initializes them to 0.
4. They reside permanently in the global data area until the program ends. However, their scope is local; when the function or block in which they are defined ends, their scope ends as well.

10.4.3 what are Stack, Heap and Global Storage Area?

In C programming, memory is managed in different regions, each with a specific purpose. These regions include the stack, heap, and global storage area. Below is an explanation of each:

10.5 Stack

- **Purpose:** The stack is used for static memory allocation. It stores local variables, function parameters, return addresses, and control data during function calls.
- **Memory Management:** The stack operates on a Last In, First Out (LIFO) basis. When a function is called, its local variables and parameters are pushed onto the stack. When the function returns, this data is popped off the stack, freeing the memory.
- **Size and Limitations:** The stack has a limited size, typically set by the operating system. If too much stack memory is used (e.g., through deep recursion or large local variables), a stack overflow can occur, leading to a program crash.
- **Lifetime of Variables:** Variables stored on the stack are automatically destroyed when the function in which they were declared exits.

10.6 Heap

- **Purpose:** The heap is used for dynamic memory allocation. This is where memory is allocated and freed explicitly using functions like `malloc()`, `calloc()`, `realloc()`, and `free()`.
- **Memory Management:** Unlike the stack, the heap does not follow a strict order for allocating and freeing memory. Memory can be allocated at any time and freed when no longer needed. However, it's up to the programmer to manage this memory correctly; failing to free allocated memory can result in memory leaks.
- **Size and Limitations:** The heap is generally larger than the stack, but it is also finite. Excessive allocation without freeing memory can exhaust the heap, leading to a program crash or failure to allocate more memory.
- **Lifetime of Variables:** Variables or objects allocated on the heap persist until they are explicitly freed by the programmer. This allows data to persist even after the function that created it has returned.

10.7 Global Storage Area

- **Purpose:** This area stores global variables and static variables, which are variables declared outside of any function or declared with the `static` keyword inside functions or files.
- **Memory Management:** The global storage area is divided into two segments: Data Segment and Code Segment. The **Data segment** can be further splitter into two parts, one for initialized global and static variables, usually we just call data segment, the other is **BSS segment (Block Started by Symbol)**, which is for uninitialized global and static variables. **Code Segment (Text Segment)** is for code (instructions) and read-only data, including string literals. The system automatically initializes uninitialized global and static variables to zero.
- **Size and Limitations:** The size of the global storage area is determined at compile time and is typically not very large compared to the heap.
- **Lifetime of Variables:** Variables in the global storage area exist for the lifetime of the program. They are initialized when the program starts and remain in memory until the program terminates.

```c
#include <stdio.h>
#include <stdlib.h>
int k1 = 1;
int k2;
static int k3 = 2;
static int k4;
int main()
{
    static int m1 = 2, m2;
    int i = 1;
    char* p;
    char str[10] = "hello";
    char* q = "hello";
    p = (char *)malloc(100);
    free(p);
    printf("Stack area - variable address    i: %p\n", &i);
    printf("Stack area - variable address   p: %p\n", &p);
    printf("Stack area - variable address str: %p\n", str);
    printf("Stack area - variable address   q: %p\n", &q);
    printf("Heap area - dynamically allocated: %p\n", p);
    printf("Global with initial value k1: %p\n", &k1);
    printf("Global without initial value k2: %p\n", &k2);
    printf("Static with initial value k3: %p\n", &k3);
    printf("Static without initial value k4: %p\n", &k4);
    printf("Static internal with initial value m1: %p\n", &m1);
    printf("Static internal without initial value m2: %p\n", &m2);
    printf("String literal address: %p, %s\n", q, q);
    printf("Program area address: %p\n", &main);
    return 0;
}
```

The output will be:

```
Stack area - variable address    i: 0x16bceb570
Stack area - variable address   p: 0x16bceb568
Stack area - variable address str: 0x16bceb578
Stack area - variable address   q: 0x16bceb560
Heap area - dynamically allocated: 0x6000026a8000
Global with initial value k1: 0x10411c000
Global without initial value k2: 0x10411c014
Static with initial value k3: 0x10411c008
Static without initial value k4: 0x10411c010
Static internal with initial value m1: 0x10411c004
Static internal without initial value m2: 0x10411c00c
String literal address: 0x104117dca, hello
Program area address: 0x104117b80
```

From the output, we can see i, p, str[], and q are located in Stack area starting from 0x16bceb...; the memory space allocated to p is located in Heap area starting from 0x6000026a8000; the k1, k2, k3, k4, m1, m2, "hello" and our program are located in the Global Storage Area starting from 0x10411.....

10.8 Initializing Local and Global Variables

When a local variable is defined, the system does not initialize it. You must initialize it yourself. When a global variable is defined, the system automatically initializes it as shown below:
Properly initializing variables is a good programming habit, as failing to do so can sometimes lead to unexpected results because uninitialized variables may contain garbage values from already available memory locations.

Data Type	Default Initialization Value
int	0
char	'\0'
float	0.0
double	0.0
pointer	NULL

Table 3: Default Initialization Values for Data Types

11 C Arrays and Strings

The C language supports the array data structure, which can store a fixed-size sequential collection of elements of the same type. Arrays are used to store a series of data, but they are often thought of as a series of variables of the same type. All arrays consist of contiguous memory locations. The lowest address corresponds to the first element, and the highest address corresponds to the last element.

Specific elements in an array can be accessed by an index, with the first index value being 0. The C language also allows us to handle arrays using pointers, making array operations more flexible and efficient.

11.1 Declaring Arrays

To declare an array in C, you need to specify the type of its elements and the number of elements, as shown below:

```
type arrayName [ arraySize ];
```

This is called a one-dimensional array. `arraySize` must be an integer constant greater than zero, and `type` can be any valid C data type. For example, to declare an array named `balance` that contains 10 elements of type `double`, the declaration statement is as follows:

```c
double balance[10];
```

Now, `balance` is an available array that can hold 10 numbers of type `double`.

11.2 Initializing Arrays

In C, you can initialize an array element by element or use an initialization statement. The number of values between the braces { } cannot exceed the number of elements specified in the brackets `[ ]` when the array was declared.

```c
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

`balance` is initialized all the elements of the array to 0.

```c
int balance[10] = {0};
```

The first five elements are initialized, the rest will be set to 0.

```c
int balance[50] = {1, 2, 3, 4, 5};
```

If you omit the array size, the array size will be the number of elements during initialization. Therefore, if:

```c
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

You will create an array identical to the one created in the previous example. Below is an example of assigning a value to a specific element in an array:

```c
balance[4] = 50.0;
```

If you want more control over the initialization process, you can use a loop to initialize each element manually:

```c
int balance[50];
for (int i = 0; i < 50; i++) {
    balance[i] = i;
}
```

The above statement assigns the value 50.0 to the fifth element of the array. All arrays start with an index of 0, which is called the base index. The last index of an array is the total size of the array minus 1.

11.3 Getting the Length of an Array

You can use the `sizeof` operator to get the length of an array, as shown below:

```c
#include <stdio.h>

int main() {
    int array[] = {1, 2, 3, 4, 5};
    int length = sizeof(array) / sizeof(array[0]);

    printf("Array length: %d\n", length);

    return 0;
}
```

11.4 Array Name

In C language, an array name represents the address of the array, i.e., the address of the first element of the array. When we declare and define an array, the array name represents the address of the array. For example, in the following code:

```c
int myArray[5] = {10, 20, 30, 40, 50};
```

Here, `myArray` is the array name, which represents an integer array containing 5 elements. `myArray` also represents the address of the array, i.e., the address of the first element. The array name itself is a constant pointer, meaning its value cannot be changed. Once defined, it cannot point to any other location. We can use the `&` operator to get the address of the array, as shown below:

```c
int myArray[5] = {10, 20, 30, 40, 50};
int *ptr = &myArray[0]; // or simply int *ptr = myArray;
```

In the example above, the pointer variable `ptr` is initialized to the address of `myArray`, i.e., the address of the first element of the array.

It's important to note that although the array name represents the address of the array, in most cases, the array name will automatically convert to a pointer to the first element of the array. This means that we can directly use the array name in pointer operations, such as when passing parameters to functions or traversing the array:

```c
void printArray(int *arr, int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]); // Array name arr is used as a pointer
        // same to: printf("%d ", *(arr++)); // Array name arr is used as a pointer
    }
}

int main() {
    int myArray[5] = {10, 20, 30, 40, 50};
    printArray(myArray, 5); // Pass the array name to the function
    return 0;
}
```

In the code above, the `printArray` function accepts an integer array and the size of the array as parameters. We pass the `myArray` array name to the function, and inside the function, the `arr` array name can be used like a pointer.

11.5 Multidimensional Arrays

The C language supports multidimensional arrays. The general form of declaring a multidimensional array is as follows:

`type name[size1][size2]...[sizeN];`

The simplest form of a multidimensional array is a two-dimensional array. A two-dimensional array is essentially a list of one-dimensional arrays. To declare a two-dimensional array with `x` rows and `y` columns, the form is as follows:

`type arrayName[x][y];`

Here, `type` can be any valid C data type, and `arrayName` is a valid C identifier. A two-dimensional array can be thought of as a table with `x` rows and `y` columns. Below is an example of a two-dimensional array with 3 rows and 4 columns:

```c
int x[3][4];
```

Thus, each element in the array is identified using an element name in the form `a[i,j]`, where `a` is the array name, and `i` and `j` are the indices that uniquely identify each element in `a`.
A multidimensional array can be initialized by specifying values for each row within braces. Here is an array with 3 rows and 4 columns:

```
int a[3][4] = {
 {0, 1, 2, 3},   /* Initializing row with index 0 */
 {4, 5, 6, 7},   /* Initializing row with index 1 */
 {8, 9, 10, 11}  /* Initializing row with index 2 */
};
```

The inner nested braces are optional; the following initialization is equivalent to the above:

```
int a[3][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
```

Elements in a two-dimensional array are accessed using indices (i.e., the row index and column index). For example:

```
int val = a[2][3];
```

The above statement will retrieve the element from the 3rd row and the 4th column of the array. You can verify this using the illustration above. Let's look at the following program where we will use nested loops to process a two-dimensional array:

```
#include <stdio.h>

int main ()
{
   /* An array with 5 rows and 2 columns */
   int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6}, {4,8} };
   int i, j;

   /* Output the value of each element in the array */
   for ( i = 0; i < 2; i++ )
   {
      for ( j = 0; j < 2; j++ )
      {
         printf("a[%d][%d] = %d\n", i, j, a[i][j] );
      }
   }
   return 0;
}
```

The output will be:

```
a[0][0] = 0
a[0][1] = 0
a[1][0] = 1
a[1][1] = 2
```

The following definition is also correct:

```
int arr[][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

When you provide an initializer list for a 2D array, the compiler can infer the size of the first dimension (the number of rows) based on the number of initializer elements provided. Here's what's happening in your declaration:

- `int arr[][3]`: The number of columns is explicitly specified as `3`, but the number of rows is left unspecified (`[]`).
- **Initializer List**: The initializer list `{{1, 2, 3}, {4, 5, 6}}` provides two rows, each containing three elements.

This works because:

- The size of the first dimension (number of rows) can be omitted when the array is initialized since the compiler can deduce it from the initializer list.
- The size of the second dimension (number of columns) must be explicitly stated so the compiler knows how many elements are in each row to correctly calculate the memory offsets.

The interesting thing is, the following statements are wrong:

```c
int arr[2][] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

In C, unlike the first dimension, the second dimension (the number of columns) of an array **must always be specified** when you declare it. Why?

1. **Memory Layout Requirement:**
   - In C, a 2D array is stored in a contiguous block of memory in row-major order. To correctly compute the memory offset for each element, the compiler needs to know the exact number of columns.
   - If you only specify the number of rows (e.g., `2` in `int arr[2][]`), the compiler doesn't know how many columns are in each row. Therefore, it cannot determine the amount of memory to allocate for each row or compute the memory addresses of the array elements.
2. **Compiler Needs to Compute Element Addressing:**
   - For any element `arr[i][j]` in a 2D array, the compiler computes the address using the formula:

     $$\text{address of } \texttt{arr[i][j]} = \text{base address of } \texttt{arr} + (i \times \text{number of columns} + j) \times \text{size of each element}$$

   - In this formula, the number of columns is essential to calculate the memory offset. Without the number of columns, the compiler cannot perform this calculation.

In short, you cannot declare an array like `int arr[2][]` because the compiler requires a fixed number of columns to correctly compute the memory allocation and element access.

11.6 Passing Arrays to Functions

If you want to pass a one-dimensional array as a parameter to a function, you must declare the function's formal parameter in one of the following three ways. The result of these three declarations is the same because each tells the compiler to expect an integer pointer. Similarly, you can pass a multidimensional array as a formal parameter.

11.6.1 Method 1-Using Pointer

```c
//1D array
#include <stdio.h>

void myFunction(int *param, int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", param[i]);
    }
    printf("\n");
}

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    myFunction(arr, 5);  // Passing the 1D array to the function
    return 0;
}
```

```c
//2D array
#include <stdio.h>

void myFunction(int arr[][3], int rows) {
//void print2DArrayPointer(int (*arr)[3], int rows) { also correct, using pointer notation
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int arr[2][3] = {
        {1, 2, 3},
        {4, 5, 6}
    };
    myFunction(arr, 2);  // Passing the 2D array to the function
    return 0;
}
```

11.6.2 Method 2-Using Known Size

```c
//1D array
#include <stdio.h>

void myFunction(int param[10]) {
    for (int i = 0; i < 10; i++) {
        printf("%d ", param[i]);
    }
    printf("\n");
}

int main() {
    int arr[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    myFunction(arr);  // Passing the 1D array to the function
    return 0;
}
```

```c
//2D array
#include <stdio.h>

void myFunction(int arr[3][3]) {
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int arr[3][3] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };
    myFunction(arr);  // Passing the 2D array to the function
    return 0;
}
```

Another way:

```c
//2D array (99 or later)
#include <stdio.h>

void myFunction(int rows, int cols, int arr[rows][cols]) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int arr[2][3] = {
        {1, 2, 3},
        {4, 5, 6}
    };
    myFunction(2, 3, arr);  // Passing the 2D array to the function
    return 0;
}
```

Why we need to specify the size of the array? In C, when you pass an array to a function, it decays to a pointer. This means that the size information is lost; you can no longer use sizeof to determine the size of the array inside the function. When you declare an array like int arr[5], it has a fixed size and its type can be expressed as int[5], and sizeof(arr) will give you the total size of the array in bytes (e.g., 5 elements * size of int). However, when you pass this array to a function like this:

```c
void exampleFunction(int arr[]) {
    // Here, arr is treated as a pointer, not an array
    printf("%zu\n", sizeof(arr)); // This will print the size of a pointer, not the array
}
```

arr is actually a pointer (int *arr). So sizeof(arr) gives you the size of the pointer (e.g., 4 or 8 bytes, depending on the architecture), not the size of the original array. This is why sizeof cannot be reliably used to determine the size of an array when it has been passed to a function.

```c
#include <stdio.h>

void f1(int *arr, int (*arr2)[3]) { // Use int (*arr2)[3] to match the type of arr2
    printf("arr[0]: %d\n", arr[0]);
    printf("arr2[0][0]: %d\n", arr2[0][0]);
}

int main() {
    int arr[2] = {1, 2};
    int arr2[2][3] = {{1, 2, 3}, {4, 5, 6}};
    f1(arr, arr2); // Now the function call matches the corrected function signature

    return 0;
}
```

Let's dig deeper. Consider the following array:

```c
int arr[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

This array is stored in a **contiguous** block of memory as follows:

$$[1, 2, 3, 4, 5, 6]$$

The indexing operator [ ] is used to access elements of an array. In the case of a 2D array, you use two sets of [ ] to access a specific element. When accessing elements using `arr[i][j]`, the C language calculates the correct memory position by treating the array as an *array of arrays*. For this specific example:

- `arr` is a pointer to an array of 3 integers `(int(*)[3])`.
- `arr[0]` points to the first array {`1, 2, 3`}.
- `arr[1]` points to the second array {`4, 5, 6`}.

```c
#include <stdio.h>

int main() {
    int arr[2][4] = { {1, 2, 3, 4}, {5, 6, 7, 8} };
    printf("%p\n", arr);// base address
    printf("%p\n", (arr + 1));// arr decays to int(*)[4], arr + 1 moves to next row arr[1], address
    of arr[1]
    printf("%p\n", &arr[1]); // explicitly takes the address of the second row, arr[1]
    printf("%p\n", *(arr + 1)); // *(arr + 1) dereferences the pointer to get the second row (arr
    [1]), which is an array. Print an array as pointer, so the type decays.

    printf("%p\n", &arr[1][0] );  // the address of first element in the second row
    printf("%d\n", *(arr + 1)[0]); // access the first element in the second row

    // The nature of continuous storage
    printf("%d\n", arr[0][1] );  // 1
    printf("%d\n", *(arr[0] + 1) );  // 1
    printf("%d\n", *(*(arr + 0) + 1) );  // 1
    //Pointer arithmetic is based on its data type int(*)[4]
    printf("%d\n", *(*(arr + 1) + 1) );  // 5

    return 0;
}
```

The output could be:

```
0x16dc63540
0x16dc63550
0x16dc63550
0x16dc63550
0x16dc63550
5
2
2
2
6
```

When arr is used in a context where it decays to a pointer, it decays to a pointer to its first element. Since arr is a 2D array, its first element is arr[0], which is an array of int[4]. arr decays to a pointer to arr[0], which is of type int[4]. Therefore, arr decays to a pointer of type int (*)[4].

Why it cannot decay to `int**`? You probably think this would result in the following memory layout:

`int**: [ pointer_to_row_0 ] [ pointer_to_row_1 ]`

where

`pointer_to_row_0` $\rightarrow [1, 2, 3]$ `pointer_to_row_1` $\rightarrow [4, 5, 6]$

However, this is **not** how a 2D array is stored in memory. A real 2D array is laid out contiguously in memory:

`arr[2][3]: [ 1, 2, 3, 4, 5, 6 ]`

In a `int**` setup, there is **an extra level** of indirection, where each row would need to be dynamically allocated and could exist in different locations in memory. This is why casting a contiguous 2D array to `int**` will not work—it changes how memory access is calculated.

Let's see what happens when you cast. Consider the following code example:

```
int arr[2][3] = { {1, 2, 3}, {4, 5, 6} };
```

If you attempt to cast `arr` to `int**`:

```
int **ptr = (int **) arr;
```

the compiler will assume `ptr` is a pointer to a pointer. Let's see the case below:

```c
#include <stdio.h>

int main() {
    int arr[2][4] = { {1, 2, 3, 4}, {5, 6, 7, 8} };

    int **ptr = (int**)arr;
    printf("%p\n", arr);
    printf("%p\n", ptr);
    printf("%p\n", (arr + 1));
    printf("%p\n", (ptr + 2));
    printf("%p\n", *(arr + 1)); // int[4] decays to int*
    printf("%p\n", *(ptr + 2));// interpreted as int*
    return 0;
}
```

The output could be:

```
0x16d51f540
0x16d51f540
0x16d51f550
0x16d51f550
0x16d51f550
0x600000005
```

Thus, from the above you can see, *(arr + 1) produce the correct address, because it represents an array, and we print it as a pointer, so the array decays to a pointer. But *(ptr + 2) is a totally different story, it will grab the first 8 bytes from the address and interpret it as a pointer! Thus, its 0x600000005, exactly the number stored in the array *(arr + 1), 5 and 6 (each 4 bytes, total 8 bytes)! Because of Little Endian, lower significant bytes are stored at lower memory addresses, thus you see the result 0x600000005. The advantage of this approach is that the least significant byte (which is often the byte you want to access first in many computations) is stored at the lowest address, making certain operations faster or more convenient.

Note: We can use (int **) to declare a dynamic 2D array, we will see it in memory management chapter.

11.7 Returning Arrays from Functions

You cannot declare a local pointer variable inside a function and return it because local variables are stored on the stack, and their memory is automatically deallocated when the function returns. This makes the memory used by local variables invalid outside of the function. Attempting to return such a variable and access it elsewhere leads to undefined behavior, such as accessing garbage values or causing a program crash. Here is what will happen for a function call in C:

1. A new "stack frame" is created on the stack for that function.
2. Local variables are allocated memory within this stack frame.

3. When the function returns, its stack frame is destroyed, and all the memory used for local variables is reclaimed.
4. The pointer returned points to a memory location that is no longer valid, which is undefined behavior.

C does not support returning the address of a local variable from a function unless the local variable is defined as a `static` variable.

```c
#include <stdio.h>

// Function to return a pointer to a static 1D array
int* myFunction() {
    static int arr[5] = {1, 2, 3, 4, 5};  // Static array
    return arr;  // Return pointer to the array
}

int main() {
    int *arr = myFunction();  // Get the returned array

    // Print the array elements
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Returning an existing static 2D array is not covered by this course.

11.8 C Strings

In C, a string is actually a one-dimensional array of characters terminated by a null character `\0`. Therefore, `\0` is used to mark the end of a string. The null character, also known as the terminator, abbreviated as NUL, is a control character with a value of 0. `\0` is an escape character, meaning it tells the compiler that this is not the character '0' but the null character.

The following declaration and initialization create a string `Wabash`. Because the null character `\0` is stored at the end of the array, the character array's size is one more than the number of characters in the word `Wabash`.

```c
char college[7] = {'W', 'a', 'b', 'a', 's', 'h', '\0'};
```

In fact, you don't need to put the null character at the end of a string literal. The C compiler automatically places the `\0` at the end of the string when initializing the array. Let's try to output the string defined above:

```c
char college[] = "wabash";
```

In C, Arrays are not assignable after declaration because they are treated as constant pointers to their first element (means you cannot assign "wabash" address to it!).

```c
char college[20];
college = "wabash";//wrong!
```

You can only assign values to arrays element by element, or use functions like strcpy to copy a string.

```c
strcpy(college, "wabash");
```

String literals are stored in read-only memory in most implementations of C. Writing to this memory (attempting to change "Qixin Deng" to "Eixin Deng") is not allowed, and the program might crash or behave unpredictably.

```c
char *name;

name = "Qixin Deng";
name[0] = 'E';//wrong!
```

If you want to modify a string, you should dynamically allocate memory or use a mutable array instead of a string literal.

```c
char name[] = "Qixin Deng"; // or any array with suitable size char name[10] for example.
name[0] = 'E';
```

C provides a large number of functions to operate on strings:

No.	Function & Purpose
1	`strcpy(s1, s2);` Copies string `s2` to string `s1`.
2	`strcat(s1, s2);` Concatenates string `s2` to the end of string `s1`.
3	`strlen(s1);` Returns the length of string `s1`.
4	`strcmp(s1, s2);` Returns 0 if `s1` and `s2` are identical; returns a value less than 0 if `s1 < s2`; returns a value greater than 0 if `s1 > s2`.
5	`strchr(s1, ch);` Returns a pointer to the first occurrence of character `ch` in string `s1`.
6	`strstr(s1, s2);` Returns a pointer to the first occurrence of string `s2` in string `s1`.

```c
#include <stdio.h>
#include <string.h>

int main ()
{
    char str1[20] = "wabash";
    char str2[20] = "college";
    char str3[20];
    int  len;

    /* Copy str1 to str3 */
    strcpy(str3, str1);
    printf("strcpy(str3, str1):  %s\n", str3);

    /* Concatenate str1 and str2 */
    strcat(str1, str2);
    printf("strcat(str1, str2):   %s\n", str1);

    /* Total length of str1 after concatenation */
    len = strlen(str1);
    printf("strlen(str1):  %d\n", len);

    return 0;
}
```

The output will be:

```
strcpy(str3, str1):  wabash
strcat(str1, str2):   wabashcollege
strlen(str1):  13
```

Strings as Pointers vs Arrays

When you declare a string as a pointer (e.g., `char *str = "Hello";`), the string literal is stored in read-only memory. Attempting to modify it will result in undefined behavior or a crash.

When you declare a string as an array (e.g., `char str[] = "Hello";`), the string is stored in writable memory, allowing you to modify it.

Multidimensional Character Arrays (Array of Strings)

You can create an array of strings by using a 2D character array:

```c
char arr[3][10] = {
    "Hello",
    "World",
    "C"
};
```

In this case, each row is a separate string, and each string can hold up to 9 characters (plus the null terminator). For the given example, **it occupies 30 bytes!** Even though some bytes are left empty.

- Strings in C are just character arrays with a null terminator (`'\0'`).
- You need to use `string.h` functions to manipulate strings easily.
- Always ensure the array has enough space for the string and the null terminator when dealing with character arrays.

12 C enum (Enumeration)

An enumeration (`enum`) is a basic data type in C used to define a set of constants with discrete values. It allows data to be more concise and easier to read. Enumeration types are typically used to name a set of related constants in a program, thereby improving the program's readability and maintainability.

To define an enumeration type, the `enum` keyword is used, followed by the name of the enumeration type and a set of enumeration constants enclosed in curly braces {}. Each enumeration constant can be represented by an identifier, and you can also assign an integer value to them. If not specified, the values start at 0 and increment by 1. The syntax for defining an enumeration is as follows:

```
enum EnumName {EnumElement1, EnumElement2, ...};
```

For example, consider a week with 7 days. Without using an enumeration, you would need to use `#define` to define an alias for each integer:

```
#define MON  1
#define TUE  2
#define WED  3
#define THU  4
#define FRI  5
#define SAT  6
#define SUN  7
```

This code is quite verbose. Now, let's see how it looks using an enumeration:

```
enum DAY
{
      MON=1, TUE, WED, THU, FRI, SAT, SUN
};
```

Note: The default value of the first enumeration member is 0, and subsequent members increment by 1 from the previous member. In this example, we set the value of the first enumeration member to 1, so the second member is 2, and so on.
You can change the values of enumeration elements when defining the enumeration type:

```
enum season {spring, summer=3, autumn, winter};
```

In this case, `spring` has a value of 0, `summer` is 3, `autumn` is 4, and `winter` is 5.

12.1 Defining Enumeration Variables

Previously, we only declared the enumeration type. Now, let's look at how to define enumeration variables. Enumeration variables can be defined in the following three ways:

```
enum DAY
{
      MON=1, TUE, WED, THU, FRI, SAT, SUN
};
enum DAY day;

//////////////////////

enum DAY
{
      MON=1, TUE, WED, THU, FRI, SAT, SUN
} day;

//////////////////////

enum
{
      MON=1, TUE, WED, THU, FRI, SAT, SUN
} day;
```

In C, enumeration types are treated as `int` or `unsigned int`, so according to the C standard, it is not possible to iterate over enumeration types directly. However, in some specific cases where enumeration types are continuous, it is possible to iterate conditionally.
The following example uses a `for` loop to iterate over the enumeration elements:

```c
#include <stdio.h>

enum DAY
{
      MON=1, TUE, WED, THU, FRI, SAT, SUN
} day;
int main()
{
    // Iterate over the enumeration elements
    for (day = MON; day <= SUN; day++) {
        printf("Enumeration element: %d \n", day);
    }
}
```

The output will be:

```
Enumeration element: 1
Enumeration element: 2
Enumeration element: 3
Enumeration element: 4
Enumeration element: 5
Enumeration element: 6
Enumeration element: 7
```

The following enumeration type is not continuous and cannot be iterated:

```c
enum
{
    ENUM_0,
    ENUM_10 = 10,
    ENUM_11
};
```

12.2 Using Enumerations in a `switch` Statement

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    enum color { red=1, green, blue };

    enum color favorite_color;

    /* User inputs a number to choose a color */
    printf("Enter your favorite color: (1. red, 2. green, 3. blue): ");
    scanf("%u", &favorite_color);

    /* Output result */
    switch (favorite_color)
    {
    case red:
        printf("Your favorite color is red");
        break;
    case green:
        printf("Your favorite color is green");
        break;
    case blue:
        printf("Your favorite color is blue");
        break;
    default:
        printf("You did not choose a color");
    }

    return 0;
}
```

The output will be:

```
Enter your favorite color: (1. red, 2. green, 3. blue): 1
Your favorite color is red
```

The main purpose of enum type is to make the code more readable. By using an enum variable, you ensure that its value will be one of the predefined constants from that enumeration:

```c
#include <stdio.h>

```

```c
enum TrafficLight {
    RED,
    YELLOW,
    GREEN
};

int main() {
    enum TrafficLight signal;
    signal = RED;

    if (signal == RED) {
        printf("Stop!\n");
    }

    if (signal == GREEN) {
        printf("Go!\n");
    }

    return 0;
}
```

13 C Structures

13.1 typedef and define

The C language provides the `typedef` keyword, which allows you to give a new name to an existing type. The following example defines the term `BYTE` for a single-byte number:

```c
typedef unsigned char BYTE;
```

After this type definition, the identifier `BYTE` can be used as a shorthand for `unsigned char`. For example:

```c
BYTE b1, b2;
```

By convention, type names defined using `typedef` are often written in **uppercase** letters to remind users that the type name is a symbolic abbreviation.

In C, both `#define` and `typedef` are used to create aliases or shortcuts, but they serve different purposes and work in distinct ways. Here's a detailed comparison, along with examples to illustrate their differences:

- `#define`:
  - `#define` is a preprocessor directive used to create macros. It can be used to define constants, aliases for expressions, or simple code snippets.
  - `#define` replaces the text in the code before the compilation starts (during the preprocessing phase).
- `typedef`:
  - `typedef` is a keyword used to create a new name (alias) for an existing data type. It is processed by the compiler during the compilation phase.
  - `typedef` is primarily used to simplify complex type declarations or to create meaningful type names.

Considering the scope:

- `#define`:
  - `#define` has a global effect within the file unless undefined using `#undef`.
  - It can lead to unintended consequences if the same name is used elsewhere, as it performs a literal text replacement.
- `typedef`:
  - `typedef` is confined to the scope in which it is declared (e.g., within a function or globally).
  - It does not cause text replacement, so the alias does not interfere with other code.

```c
#include <stdio.h>

#define PI 3.14
#define SQUARE(x) ((x) * (x))

int main() {
    printf("PI: %f\n", PI);
    printf("Square of 5: %d\n", SQUARE(5));
    return 0;
}
```

- `PI` is replaced by `3.14` wherever it appears in the code.
- `SQUARE(x)` is a macro that replaces `SQUARE(5)` with `((5) * (5))`, which then evaluates to `25`.

```c
#include <stdio.h>

typedef unsigned long int ULONG;
typedef struct {
    char name[50];
    int age;
} Person;

int main() {
    ULONG bigNumber = 1000000;
    Person person = {"Alice", 30};

    printf("Big Number: %lu\n", bigNumber);
    printf("Name: %s, Age: %d\n", person.name, person.age);
    return 0;
}
```

- `ULONG` is an alias for `unsigned long int`, making the code more concise.
- `Person` is a new type that represents a structure with `name` and `age` fields, improving readability.

Aspect	`#define`	`typedef`
Purpose	Define constants, macros, and replace text before compilation	Create type aliases during compilation
Scope	Global within the file unless undefined	Scoped to where it's declared
Syntax Use Cases	Constants, expressions, code snippets, simple text replacement	Type definitions, improving code readability
Type Safety	Not type-safe, can lead to unexpected results	Type-safe, checked by the compiler
Flexibility	More flexible but risky due to text replacement	Less flexible but safer and more readable

13.2 Defining a Structure

A structure is user-defined data type available in C that allows you to store different types of data items. The data members within a structure can be basic data types like `int`, `float`, `char`, etc., or they can be other structure types, pointer types, and so on.

Structures are used to represent a record for example. Suppose you want to keep track of books in a library, you might want to track the following attributes for each book:

- Title
- Author
- Subject
- Book ID

A structure definition consists of the keyword `struct` followed by the structure name, which can be defined as needed. The `struct` statement defines a new data type that contains multiple members. The format of the `struct` statement is as follows:

```
struct tag {
    member-list
    member-list
    member-list
    ...
} variable-list;
```

Here, `tag` is the structure tag. `member-list` is the standard variable definition, such as `int i;` or `float f;`, or any other valid variable definition. `variable-list` defines the structure variables, specified at the end of the structure before the final semicolon. You can specify one or more structure variables. Below is an example of how to declare a `Book` structure:

```
struct Course
{
   char title[50];
   char professor[50];
   char subject[50];
   int course_id;
} course;
```

In general, at least two of the following three parts, `tag`, `member-list`, and `variable-list`, should be present. Consider the following examples:

```
// This declaration defines a structure with three members: an int, a char, and a double,
// and also declares a structure variable s1. This structure does not have a tag.
struct
{
    int a;
    char b;
    double c;
} s1;
```

```
// This declaration defines a structure with three members: an int, a char, and a double.
// The structure's tag is named SIMPLE, but no variables are declared.
struct SIMPLE
{
    int a;
    char b;
    double c;
};
// Structure with SIMPLE tag, additionally declaring variables t1, t2, and t3.
struct SIMPLE t1, t2[20], *t3;
```

```
// You can also create new types using typedef.
typedef struct
{
    int a;
    char b;
    double c;
} Simple2;
// Now, you can use Simple2 as a type to declare new structure variables.
Simple2 u1, u2[20], *u3;
```

In the declarations above, the first and second declarations are treated as completely different types by the compiler, even though their member lists are identical. It would be illegal to assign `t3 = &s1;`. You should also see the difference between variable declarations using typedef and regular struct.

13.3 Nested Structures and Pointers

A structure's members can include other structures, or pointers to its own structure type. These pointers are typically used to implement more advanced data structures like linked lists and trees.

```
// This structure declaration includes another structure.
struct COMPLEX
{
    char string[100];
    struct SIMPLE a;
};
```

```
// This structure declaration includes a pointer to its own type.
struct NODE
{
    char string[100];
    struct NODE *next_node;
};
```

If two structures contain each other, an incomplete declaration must be used for one of the structures, as shown below:

```
struct B; // Incomplete declaration of structure B

// Structure A contains a pointer to structure B.
struct A
{
    struct B *partner;
    // other members;
};

// Structure B contains a pointer to structure A. After A is declared, B is also declared.
struct B
{
    struct A *partner;
    // other members;
};
```

13.4 Initializing Structure Variables

Like other types of variables, structure variables can be initialized when they are defined.

```
#include <stdio.h>

struct Course
{
   char title[50];
   char professor[50];
   char subject[50];
   int course_id;
} c = {"C Programming", "Qixin Deng", "CSC", 241};
```

13.5 Accessing Structure Members

To access the members of a structure, we use the member access operator (“.”). The member access operator is a period between the structure variable name and the member we want to access. You can define variables of the structure type using the `struct` keyword. The following example demonstrates the use of structures:

```
#include <stdio.h>
#include <string.h>

struct Course
{
   char title[50];
   char professor[50];
   char subject[50];
   int course_id;
};

int main( )
{
   struct Course C1;

   /* Detailed information about C1 */
   strcpy(C1.title, "C Programming");
   strcpy(C1.professor, "Qixin Deng");
   strcpy(C1.subject, "CSC");
   C1.course_id = 241;

   printf("C1 Title : %s\n", C1.title);

   return 0;
}
```

13.6 Structures as Function Arguments

You can pass structures as function arguments. The method of passing is similar to other types of variables or pointers. You can access structure variables using the method described above.

```
void printCourse(struct Course C);
```

13.7 Pointers to Structures

You can define a pointer to a structure in the same way as you define a pointer to any other variable type, as shown below:

```
struct Books *struct_pointer;
```

Now, you can store the address of a structure variable in this pointer variable. To find the address of a structure variable, use the `&` operator in front of the structure name, as shown below:

```
struct_pointer = &Book1;
```

To access the members of the structure pointed to by this pointer, you must use the `->` operator, as shown below:

```
struct_pointer->title;
```

13.8 Calculating Structure Size

In C, you can use the `sizeof` operator to calculate the size of a structure. `sizeof` returns the size in bytes of the given type or variable. For structures, `sizeof` returns the total number of bytes occupied by the structure, including the size of all member variables and any possible padding bytes. The following example demonstrates how to calculate the size of a structure:

```
#include <stdio.h>

struct Person {
    char name[20];
    int age;
    float height;
};

int main() {
    struct Person person;
    printf("The size of the Person structure is: %zu bytes\n", sizeof(person));
    return 0;
}
```

In the above example, we define a structure named `Person`, which contains a character array `name`, an integer `age`, and a float `height`. In the `main` function, we declare a `Person` type variable `person`, and then use the `sizeof` operator to get the size of the `person` structure. Finally, we use the `printf` function to print out the size of the structure, which produces the following output:

```
The size of the Person structure is: 28 bytes
```

Note that the size of a structure can be affected by compiler optimizations and alignment rules. The compiler may insert extra padding bytes into the structure to align its member variables for more efficient memory access. Therefore, the actual size of the structure may be larger than the sum of the sizes of its member variables. If you need to understand the memory layout and alignment of a structure more precisely, you can use macros like `offsetof` and attributes like `__attribute__((packed))` to further control and query the structure's size and alignment.

13.9 Bit Fields

In C, a bit-field is a special structure member that allows you to define members with a specified number of bits. This feature is useful when you need to manage multiple variables that represent switches or flags (i.e., variables with TRUE/FALSE values), like the following:

```
struct {
  unsigned int widthValidated;
  unsigned int heightValidated;
} status;
```

This structure would typically require 8 bytes of memory, but in reality, we are only storing 0 or 1 in each variable. C provides a more memory-efficient way to handle such variables by allowing you to define the width of each variable within the structure, indicating to the compiler that only a few bits are needed. The above structure can be rewritten as:

```
struct {
  unsigned int widthValidated : 1;
  unsigned int heightValidated : 1;
} status;
```

Now, the `status` variable in the above structure will occupy 4 bytes of memory, but only 2 bits will be used to store values. If you use 32 variables, each with a width of 1 bit, the `status` structure will use 4 bytes. However, if you add one more variable (33 in total), the next memory segment will be allocated, and 8 bytes will be used. Let's examine the following example to understand this concept:

```c
#include <stdio.h>
#include <string.h>

/* Define a simple structure */
struct {
  unsigned int widthValidated;
  unsigned int heightValidated;
} status1;

/* Define a bit-field structure */
struct {
  unsigned int widthValidated : 1;
  unsigned int heightValidated : 1;
} status2;

int main() {
   printf("Memory size occupied by status1: %d\n", sizeof(status1));
   printf("Memory size occupied by status2: %d\n", sizeof(status2));

   return 0;
}
```

When the above code is compiled and executed, it will produce the following results:

```
Memory size occupied by status1: 8
Memory size occupied by status2: 4
```

The characteristics and usage of bit-fields are as follows:

- When defining a bit-field, you can specify the width of the member, i.e., the number of bits it occupies.
- The width of a bit-field cannot exceed the size of its data type, as the bit-field must fit within the integer type used.
- The data type of a bit-field can be `int`, `unsigned int`, `signed int`, or an enum type.
- Bit-fields can be used independently or in combination with other structure members.
- Access to bit-fields is achieved using the dot operator (“.”), similar to accessing normal structure members.

The bit-field definition is similar to a structure definition and follows this format:

```
struct bitFieldStruct {
    type [member_name] : width;
};
```

The following table describes the elements of a bit-field:

- `type`: Must be one of `int`, `unsigned int`, or `signed int`, which determines how the bit-field values are interpreted.
- `member_name`: The name of the bit-field.
- `width`: The number of bits in the bit-field. The width must be less than or equal to the width of the specified type.

Variables with predefined widths are called bit-fields. A bit-field can store more than one bit of a number. For example, to store a value from 0 to 7, you can define a 3-bit wide bit-field:

```c
struct {
    unsigned int age : 3;
} Age;
```

The above structure definition instructs the C compiler that the `age` variable will only use 3 bits to store its value. If you attempt to use more than 3 bits, it won't be possible, or say it will lead to **implicit truncation**, the data stored in the corresponding attribute will depend on the actual value stored in the corresponding bits.

```c
struct bs {
    int a:8;
    int b:2;
    int c:6;
} data;
```

The above code defines a structure named `struct bs` with a structure variable `data`, which occupies four bytes. For bit-fields, their width cannot exceed the size of their data type. In this case, an `int` type typically occupies 4 bytes (32 bits). Adjacent bit-fields with the same type, whose combined bit-width is less than the size of the type, will be stored together until no more space is available.

```c
#include <stdio.h>

struct example1 {
    int a : 4;
    int b : 5;
    int c : 7;
};

int main() {
    struct example1 ex1;

    printf("Size of example1: %lu bytes\n", sizeof(ex1));

    return 0;
}
```

The struct will use the largest data type as its unit:

```
```

In the above example, the `example1` structure contains three bit-field members `a`, `b`, and `c`, occupying 4 bits, 5 bits, and 7 bits, respectively. The `sizeof` operator calculates the size of the `example1` structure in bytes, and the result is output as:

```
Size of example1: 4 bytes
```

The struct bit field will always use the largest data type as its unit to expand if necessary:

```c
struct Data {
    int i : 31;
    short s : 6;
} data;

int main(){
    printf("%d\n", sizeof(data));// 6 bytes or 8 bytes?
    return 0;
}
```

The output will be:

```
8
```

13.10 C Union

A union is a special data type that allows you to store different data types in the same memory location. You can define a union with many members, but only one member can hold a value at any given time. Unions provide an efficient way of using the same memory location for multiple purposes.

13.11 Defining a Union

To define a union, you must use the `union` statement, which is similar to defining a structure. The `union` statement defines a new data type with multiple members. The format of the `union` statement is as follows:

```
union [union tag] {
    member definition;
    member definition;
    ...
    member definition;
} [one or more union variables];
```

Here, `union tag` is optional, and each `member definition` is a standard variable definition such as `int i;` or `float f;` or any other valid variable definition. At the end of the union definition, before the final semicolon, you can optionally specify one or more union variables. Below is an example of a union type named `Data` that has three members: `i`, `f`, and `str`:

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

Now, a variable of type `Data` can store an integer, a floating-point number, or a string. This means that one variable (the same memory location) can store data of multiple types. You can use any built-in or user-defined data type within a union as needed.

13.12 Memory Usage of a Union

A union occupies memory sufficient to store the largest member in the union. For example, in the above instance, `Data` will occupy 20 bytes of memory, because among the members, the string occupies the most space. The following example will show the total memory size occupied by the union:

```c
#include <stdio.h>
#include <string.h>

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

int main() {
    union Data data;

    printf("Memory size occupied by data: %d\n", sizeof(data));

    return 0;
}
```

The output will be:

```
Memory size occupied by data: 20
```

13.13 Accessing Union Members

To access the members of a union, we use the member access operator (“.”). The member access operator is a period between the union variable name and the member you want to access. You can use the `union` keyword to define variables of the union type. The following example demonstrates the usage of a union:

```c
#include <stdio.h>
#include <string.h>

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

int main() {
    union Data data;

    data.i = 10;
    data.f = 220.5;
    strcpy(data.str, "C Programming");

    printf("data.i: %d\n", data.i);
    printf("data.f: %f\n", data.f);
    printf("data.str: %s\n", data.str);

    return 0;
}
```

When the above code is compiled and executed, it will produce the following results:

```
data.i: 1917853763
data.f: 4122360580327794860452759994368.000000
data.str: C Programming
```

Here, we can see that the values of the `i` and `f` members of the union are corrupted because the last assigned value occupies the memory location, which is why the `str` member can be output correctly. Now let’s look at the same example, but this time we use only one variable at a time, which also demonstrates the main purpose of using a union:

```c
#include <stdio.h>
#include <string.h>

union Data {
    int i;
```

```c
    float f;
    char str[20];
};

int main() {
    union Data data;

    data.i = 10;
    printf("data.i: %d\n", data.i);

    data.f = 220.5;
    printf("data.f: %f\n", data.f);

    strcpy(data.str, "C Programming");
    printf("data.str: %s\n", data.str);

    return 0;
}
```

When the above code is compiled and executed, it will produce the following results:

```
data.i: 10
data.f: 220.500000
data.str: C Programming
```

In this example, all members output correctly because only one member is used at a time.

14 C Input & Output

When we refer to input, it means providing some data to the program. Input can be in the form of files or from the command line. The C language provides a set of built-in functions to read the given input and populate it into the program as needed. When we refer to output, it means displaying some data on the screen, printer, or any file. The C language provides a set of built-in functions to output data to the computer screen and to save data into text or binary files.

14.1 Standard Files

In C, **all devices are treated as files**. Therefore, devices (such as monitors) are handled the same way as files. The following three files are automatically opened when a program is executed to allow access to the keyboard and screen:

Standard File	File Pointer	Device
Standard Input	`stdin`	Keyboard
Standard Output	`stdout`	Screen
Standard Error	`stderr`	Screen

The file pointer is the way to access a file, and this section will explain how to read values from the keyboard and output results to the screen. I/O (Input/Output) in C is typically performed using the `printf()` and `scanf()` functions. The `scanf()` function is used to read and format input from standard input (keyboard), while the `printf()` function sends formatted output to standard output (screen). The 'f' in printf and scanf means "formatted".

```c
int printf(const char *format, ...);
int scanf(const char *format, ...);
```

```c
#include <stdio.h>  // Required for printf() function

int main() {
    int age;
    scanf("%d", &age);
    printf("age: %d", age);  // Display the content in quotes
    return 0;
}
```

The scanf function should not include a prompt message inside its format string. **The format string of scanf should only contain the format specifiers** (like %d for integers), and the prompt message should be handled separately using printf.

14.2 Formatting Output in C

In C, the `printf` function is used to print output to the standard output stream, usually the console. The `printf` function allows you to format the output by using format specifiers, which define how the values are presented.

- `%d` **or** `%i` - Used for printing integers.
  - Example: `printf("%d", 10);` will output `10`.
- `%f` - Used for printing floating-point numbers.
  - Example: `printf("%f", 3.14);` will output `3.140000`.
  - You can control the number of digits after the decimal point by specifying precision. For example, `printf("%.2f", 3.14);` will output `3.14`.
- `%c` - Used for printing characters.
  - Example: `printf("%c", 'A');` will output `A`.
- `%s` - Used for printing strings.
  - Example: `printf("%s", "Hello");` will output `Hello`.
- `%x` **or** `%X` - Used for printing hexadecimal numbers. `%x` uses lowercase letters, and `%X` uses uppercase.
  - Example: `printf("%x", 255);` will output `ff`.
- `%p` - used in C to print a pointer (i.e., the memory address stored in a pointer variable). When you use `%p` in a `printf` function, it outputs the memory address in a platform-dependent hexadecimal format, typically preceded by `0x`.
- `%%` - Used for printing a percentage sign.
  - Example: `printf("%%");` will output `%`.

You can also specify width and precision:

- **Width**: Specifies the minimum number of characters to print. If the value to be printed is shorter than the width, it is padded with spaces.
  - Example: `printf("%5d", 10);` will output `  10` (three spaces before `10`).
- **Precision**: Specifies the number of digits to be printed after the decimal point for floating-point numbers or the maximum number of characters to be printed for strings.
  - Example: `printf("%.3f", 3.14159);` will output `3.142`.

You can also specify flags:

- `-` **(Left Justify)**: The output is left-justified within the given field width.
  - Example: `printf("%-5d", 10);` will output `10  ` (two spaces after `10`).
- `0` **(Zero Padding)**: Pads the output with zeros instead of spaces.
  - Example: `printf("%05d", 10);` will output `00010`.

14.3 `getchar()` & `putchar()` Functions

The `int getchar(void)` function reads the next available character from the screen and returns it as an integer. This function reads only one character at a time. You can use this method in a loop to read multiple characters from the screen. The `int putchar(int c)` function outputs a character to the screen and returns the same character. This function outputs only one character at a time. You can use this method in a loop to output multiple characters to the screen.

```c
#include <stdio.h>

int main() {
    int c;

    printf("Enter a value :");
    c = getchar();

    printf("\nYou entered: ");
    putchar(c);
    printf("\n");
    return 0;
}
```

When the above code is compiled and executed, it will wait for you to enter some text. After you input text and press the Enter key, the program will continue and will read only a single character, displaying the following:

```
Enter a value :aasdfasdf

You entered: a
```

14.4 gets() & puts() Functions

The `char *gets(char *s)` function reads a line from `stdin` into the buffer pointed to by `s` until a terminating character or `EOF`. The `int puts(const char *s)` function writes the string `s` and a trailing newline character to `stdout`.

```c
#include <stdio.h>

int main() {
    char str[100];

    printf("Enter a value :");
    gets(str);

    printf("\nYou entered: ");
    puts(str);
    return 0;
}
```

When the above code is compiled and executed, it will wait for you to enter some text. After you input text and press the Enter key, the program will continue and will read the entire line until it ends, displaying the following:

```
Enter a value :asdfasdf

You entered: asdfasdf
```

The gets() function is unsafe because it does not check the size of the buffer into which it is reading data, making it prone to buffer overflow attacks. It has been deprecated in the C standard and should be replaced with safer alternatives like fgets(), which allows you to specify the maximum number of characters to read, ensuring that your program is protected from this common vulnerability.

14.5 File Input and Output in C

A file, whether it is a text file or a binary file, represents a sequence of bytes. The C language provides functions to access these files at a high level, as well as low-level (OS) calls to handle files on storage devices. Clion uses debug folder as the default project directory.

Files are handled by **using a pointer to a FILE object**, which is defined in the stdio.h library. The FILE structure contains information used by the operating system to manage the file, such as the file's current position, error state, and the buffering mechanism. When you open a file using the fopen() function, it returns a file pointer of type **FILE *****. You use this pointer to perform various file operations such as reading, writing, and closing the file.

You can use the `fopen()` function to create a new file or open an existing file. This call initializes an object of type `FILE`, which contains all the necessary information to control the stream. The prototype for this function is as follows:

```
FILE *fopen(const char *filename, const char *mode);
```

Here, `filename` is a string used to name the file, and the access `mode` can be one of the following values:

Mode	Description
`r`	Opens an existing text file for reading.
`w`	Opens a text file for writing. If the file does not exist, a new file is created. The content is written from the beginning of the file. If the file exists, it is truncated to zero length.
`a`	Opens a text file in append mode for writing. If the file does not exist, a new file is created. The content is appended to the end of the file.
`r+`	Opens a text file for reading and writing.
`w+`	Opens a text file for reading and writing. If the file exists, it is truncated to zero length. If the file does not exist, a new file is created.
`a+`	Opens a text file for reading and writing. If the file does not exist, a new file is created. Reading starts from the beginning, and writing is only in append mode.

If you are dealing with binary files, you need to use the following access modes instead of the above:
`"rb"`, `"wb"`, `"ab"`, `"rb+"`, `"r+b"`, `"wb+"`, `"w+b"`, `"ab+"`, `"a+b"`

To close a file, use the `fclose()` function. The prototype of the function is as follows:

```
int fclose(FILE *fp);
```

If the file is successfully closed, the `fclose()` function returns zero. If an error occurs while closing the file, the function returns `EOF`. This function actually flushes the data in the buffer, closes the file, and releases all memory used by the file. `EOF` is a constant defined in the header file `stdio.h`.

The following is the simplest function to write a character to a stream:

```
int fputc(int c, FILE *fp);
```

The `fputc()` function writes the character value of `c` to the output stream pointed to by `fp`. If the write is successful, it returns the written character. If an error occurs, it returns `EOF`. You can use the following function to write a null-terminated string to the stream:

```
int fputs(const char *s, FILE *fp);
```

The `fputs()` function writes the string `s` to the output stream pointed to by `fp`. If the write is successful, it returns a non-negative value. If an error occurs, it returns `EOF`. You can also use the `int fprintf(FILE *fp, const char *format, ...)` function to write a string to a file. Try the following example:

```c
#include <stdio.h>

int main() {
    FILE *fp = NULL;

    fp = fopen("/tmp/test.txt", "w+");
    fprintf(fp, "This is testing for fprintf...\n");
    fputs("This is testing for fputs...\n", fp);
    fclose(fp);
}
```

The following is the simplest function to read a single character from a file:

```
int fgetc(FILE *fp);
```

The `fgetc()` function reads a character from the input file pointed to by `fp`. The return value is the character read, or `EOF` if an error occurs. The following function allows you to read a string from a stream:

```
char *fgets(char *buf, int n, FILE *fp);
```

The `fgets()` function reads `n - 1` characters from the input stream pointed to by `fp`. It copies the read string into the buffer `buf` and appends a null character to terminate the string.

If this function encounters a newline character `'\n'` or the end of the file `EOF` before reading the last character, it will only return the characters read, including the newline. You can also use the `int fscanf(FILE *fp, const char *format, ...)` function to read strings from a file, but **it will stop reading when it encounters the first space or newline character**.

```c
#include <stdio.h>

int main() {
    FILE *fp = NULL;
    char buff[255];

    fp = fopen("test.txt", "r");
    fscanf(fp, "%s", buff);
    printf("1: %s\n", buff);

    fgets(buff, 255, fp);
    printf("2: %s\n", buff);

    fgets(buff, 255, fp);
    printf("3: %s\n", buff);
    fclose(fp);
}
```

The output will be:

```
1: This
2: is testing for fprintf...

3: This is testing for fputs...
```

First, the `fscanf()` method only reads `This` because it encounters a space afterward. Next, the `fgets()` call reads the remaining part of the line until the end. Finally, the second `fgets()` call reads the second line completely. The file pointer fp is used to manage the current reading position within the file. The file pointer keeps track of where you are reading from in the file, so each file operation (like fscanf or fgets) updates the pointer to the next position after the operation is completed.

14.6 C Command-Line Arguments

When executing a program, you can pass values from the command line to a C program. These values are called command-line arguments. They are important for the program, especially when you want to control the program externally rather than hard-coding these values within the code.

In C, command-line arguments are a way to get input from the command line and can be used to pass information to the program at runtime. Command-line arguments are passed to the program via the parameters of the `main` function. The prototype of the `main` function can be one of the following two forms:

```c
int main(int argc, char *argv[]);
```

or:

```c
int main(int argc, char **argv);
```

argc (argument count): Represents the number of command-line arguments, including the program name itself. Therefore, `argc` is at least 1.

argv (argument vector): Is **a pointer to an array of strings**, where each string is a command-line argument. The first element of the array (`argv[0]`) is usually the name of the program. The subsequent elements are the command-line arguments passed to the program.

Here is a simple example that checks whether any command-line arguments are provided and performs actions based on the arguments:

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
   if(argc == 2)
   {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if(argc > 2)
   {
      printf("Too many arguments supplied.\n");
   }
   else
   {
      printf("One argument expected.\n");
   }
}
```

Using one argument, compile and execute the above code, and it will produce the following result:

```
$./MyExecutable testing
The argument supplied is testing
```

Using two arguments, compile and execute the above code, and it will produce the following result:

```
$./MyExecutable testing1 testing2
Too many arguments supplied.
```

Not passing any arguments, compile and execute the above code, and it will produce the following result:

```
$./MyExecutable
One argument expected
```

It should be noted that `argv[0]` stores the program's name, `argv[1]` is a pointer to the first command-line argument, and `*argv[n]` is the last argument. If no arguments are provided, `argc` will be 1; otherwise, if one argument is passed, `argc` will be set to 2.

Multiple command-line arguments are separated by spaces. However, if an argument itself contains spaces, you should enclose the argument in double quotes `""` or single quotes `''`. Let's rewrite the above example to pass a command-line argument enclosed in double quotes:

```
$./MyExecutable "testing1 testing2"

The argument supplied is testing1 testing2
```

Command-line arguments are useful in many scenarios, such as:

- Configuration file paths
- Mode selection (e.g., debug mode)
- Input and output file names
- Runtime options and flags (e.g., `-v` for verbose mode)

15 C Memory Management

The C language provides several functions for memory allocation and management, which can be found in the `<stdlib.h>` header file. In C, memory is managed through pointer variables. A pointer is a variable that stores a memory address, which can point to a variable of any data type, including integers, floating-point numbers, characters, and arrays. C provides some functions and operators that allow programmers to perform memory operations, including allocation, deallocation, movement, and copying.

15.1 memcpy() and memmove()

In C, `memcpy` and `memmove` are two functions provided by the standard library that allow you to copy blocks of memory from one location to another. While both are used for copying memory, they behave differently in scenarios where the source and destination memory regions overlap. Both functions are declared in the `<string.h>` header file:

```
#include <string.h>

void *memcpy(void *dest, const void *src, size_t n);
void *memmove(void *dest, const void *src, size_t n);
```

- **dest**: A pointer to the destination memory block where the data should be copied.
- **src**: A pointer to the source memory block from which data will be copied.
- **n**: The number of bytes to copy.
- Both functions return a pointer to the destination memory block (`dest`).

`memcpy` directly copies data **byte-by-byte** from the source memory to the destination memory. Since it does not check if the memory regions overlap, it can lead to data corruption when the regions overlap.

Aspect	memcpy	memmove
Overlapping Regions	Does not handle overlap and can cause data corruption.	Safely handles overlap between source and destination.
Performance	Typically faster due to lack of checks.	May be slightly slower because it has to ensure safety with overlapping memory.
Use Case	Ideal when you know that source and destination don't overlap. Example: Copying a buffer of data to a separate memory location.	Use when source and destination might overlap. Example: Shifting memory blocks within the same array.

Table 4: Comparison of `memcpy` and `memmove`

```c
#include <stdio.h>
#include <string.h>

int main() {
    char str[] = "Hello, World!";

    memcpy(str + 4, str, 6);  // Copy "Hello," to the part starting at str[7]
    printf("After memcpy: %s\n", str); // Hello, Hello,

    return 0;
}
```

The output will be:

```
After memcpy: HellHellHeld!
```

`memmove` is designed to handle cases where the source and destination memory regions overlap. It ensures that data is copied safely, usually by copying to a **temporary buffer** if necessary. This prevents data from being corrupted if part of the memory overlaps with the source.

```c
#include <stdio.h>
#include <string.h>

int main() {
    char str[] = "Hello, World!";

    // Safely move memory with overlap
    memmove(str + 4, str, 6);  // Copy "Hello," to the part starting at str[7]
    printf("After memmove: %s\n", str); // Hello, Hello,

    return 0;
}
```

The output will be:

```
After memmove: HellHello,ld!
```

In C, both `memcpy` and `memmove` perform a **shallow copy**, meaning they only copy the raw bytes of the data rather than creating a deep copy of the structures, which would involve duplicating objects that the pointers point to. This example demonstrates how both functions operate on a structure containing pointers. Consider the following example where a structure contains a pointer to dynamically allocated memory. When copying this structure with `memcpy` or `memmove`, only the pointer is copied (shallow copy), not the data it points to (deep copy).

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
    int id;
    char *name;  // Pointer to a dynamically allocated string
} Person;

int main() {
    // Create and allocate memory for the first person
    Person p1;
    p1.id = 1;
    p1.name = (char *)malloc(20 * sizeof(char));  // Allocate memory for the name
    strcpy(p1.name, "Alice");

    // Create a second person
    Person p2;

```

```
    // Shallow copy using memcpy, try memmove you will get the same result
    memcpy(&p2, &p1, sizeof(Person));

    // Modify the name in p2
    strcpy(p2.name, "Bob");

    // Print both persons
    printf("After memcpy:\n");
    printf("Person ID: %d, Name: %s\n", p1.id, p1.name);  // This will now print "Bob", as p1 and p2
     share the same name pointer
    printf("Person ID: %d, Name: %s\n", p2.id, p2.name);

    free(p1.name); //do we need to free p2.name? NO! They are pointing to the same memory address.
    return 0;
}
```

The output will be:

```
After memcpy:
Person ID: 1, Name: Bob
Person ID: 1, Name: Bob
```

But think about the modification below, why we print the totally different result?

```
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <string.h>


typedef struct {
    int id;
    char *name;
} Person;

int main() {
    // Create and allocate memory for the first person
    Person p1;
    p1.id = 1;
    p1.name = "Alice";

    // Create a second person
    Person p2;

    memcpy(&p2, &p1, sizeof(Person));

    p2.name = "Bob";

    // Print both persons
    printf("Person ID: %d, Name: %s\n", p1.id, p1.name);
    printf("Person ID: %d, Name: %s\n", p2.id, p2.name);
    return 0;
}
```

The output will be:

```
Person ID: 1, Name: Alice
Person ID: 1, Name: Bob
```

This is because we do not directly modify the content stored in a memory space, instead, we let p2.name point to a different memory address in the Global data area.

To implement a **deep copy**, you would need to manually allocate new memory for the `name` field and copy the contents of the string rather than just copying the pointer.

```
// Deep copy for name field
p2.name = (char *)malloc(strlen(p1.name) + 1);  // Allocate new memory for name in p2
strcpy(p2.name, p1.name);  // Perform deep copy of the string
```

15.2 Dynamic Memory Allocation

Note: The `void *` type represents a pointer of an undetermined type. In C and C++, a `void *` pointer can be typecast to any other pointer type.

No.	Function and Description
1	`void *calloc(int num, int size);` Dynamically allocates memory for an array of `num` elements, each of `size` bytes, and initializes all bytes to 0. The result is a block of memory of `num*size` bytes, with all bytes set to 0.
2	`void free(void *address);` Frees the memory block pointed to by `address`; the memory space must have been dynamically allocated.
3	`void *malloc(int num);` Allocates a block of memory of specified size in the heap, used to store data. The memory block is not initialized, so its value is unknown.
4	`void *realloc(void *address, int newsize);` Reallocates memory, expanding the memory block to `newsize`.

When programming, if you know the size of an array in advance, defining the array is straightforward. For example, an array that stores names and can hold up to 100 characters can be defined as follows:

```
char name[100];
```

However, if you do not know the length of the text to be stored in advance, such as when storing a detailed description of a subject, you need to define a pointer that points to a character array with an undefined memory size. Memory is allocated later as needed, as shown below:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
   char name[100];
   char *description;

   strcpy(name, "Qixin Deng");

   /* Dynamically allocate memory */
   description = (char *)malloc(200 * sizeof(char));
   if(description == NULL) {
      fprintf(stderr, "Error - unable to allocate required memory\n");
   } else {
      strcpy(description, "Qixin Deng, An assistant professor at Wabash College");
   }
   printf("Name = %s\n", name);
   printf("Description: %s\n", description);
}
```

The above program can also be written using `calloc()` instead of `malloc()`, as shown below:

```
calloc(200, sizeof(char));
```

When dynamically allocating memory, you have full control and can pass any size value. **Arrays with predefined sizes cannot change their size once defined.**

15.3 Difference between `calloc (clear allocation)` and `malloc (memory allocation)` in C

In C programming, both `calloc` and `malloc` are used for dynamic memory allocation, but they have key differences in how they allocate and initialize memory. Here's a breakdown of their differences:

- `malloc`:

  ```
  void* malloc(size_t size);
  ```

  - Allocates a block of memory of the specified size (in bytes).
  - The size is provided as a single parameter, representing the total number of bytes needed.
  - Allocates memory but does **not initialize** the memory content. The allocated memory contains **garbage values** (whatever data was previously in that memory location).
  - You need to manually initialize the memory if required.
  - Generally faster because it only allocates memory without initializing it.
  - `malloc` allocate contiguous blocks of memory

- `calloc`:

```
void* calloc(size_t num, size_t size);
```

  - Allocates memory for an array of `num` elements, each of `size` bytes.
  - Takes two parameters: the number of elements and the size of each element.
  - Allocates memory and **initializes all bits to zero**. This means the allocated memory is set to **0** (for integers and floating-point numbers) or `NULL` (for pointers).
  - This makes `calloc` useful when you need zero-initialized memory.
  - May be slower than `malloc` because it zeroes out the allocated memory. However, the difference is usually negligible unless allocating large amounts of memory frequently.
  - `calloc` allocate contiguous blocks of memory

```c
#include <stdio.h>
#include <stdlib.h>
//find an online compiler to avoid OS advance feature of Zeroing Memory Pages for security
int main() {
    // Allocate using malloc and initialize manually
    int *arr_malloc = (int*)malloc(5 * sizeof(int));
    for (int i = 0; i < 5; i++) {
        arr_malloc[i] = i + 1;
    }

    free(arr_malloc);

    // Reallocate the memory with malloc
    arr_malloc = (int*)malloc(5 * sizeof(int));
    printf("Values after re-allocating with malloc (uninitialized):\n");
    for (int i = 0; i < 5; i++) {
        printf("malloc arr[%d] = %d\n", i, arr_malloc[i]);
    }

    // Allocate using calloc and check zero initialization
    int *arr_calloc = (int*)calloc(5, sizeof(int));
    printf("Values after calloc (initialized to zero):\n");
    for (int i = 0; i < 5; i++) {
        printf("calloc arr[%d] = %d\n", i, arr_calloc[i]);
    }

    free(arr_malloc);
    free(arr_calloc);

    return 0;
}
```

If you're not planning to read from the allocated memory before writing to it, malloc() is typically the better choice. **This is because it can reuse "dirty" memory that has been previously freed, which avoids the need to request new pages from the operating system or zero-initialize the memory.** This can be more efficient, especially for small allocations.

15.4 Resizing and Freeing Memory

When a program exits, the operating system automatically frees all memory allocated to the program. However, it is recommended to call the `free()` function to release memory when it is no longer needed.

Alternatively, you can resize the allocated memory block by calling the `realloc()` function. Let's revisit the previous example using `realloc()` and `free()`:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    char name[100];
    char *description;

    strcpy(name, "Qixin Deng");

    /* Allocate enough memory for the initial string */
    description = (char *)malloc((strlen("Qixin Deng is a professor") + 1) * sizeof(char));
```

```
    if(description == NULL) {
        fprintf(stderr, "Error - unable to allocate required memory\n");
        return 1;
    } else {
        strcpy(description, "Qixin Deng is a professor");
    }

    /* Reallocate memory for the additional string */
    description = (char *)realloc(description, (strlen(description) + strlen(" at Wabash College") +
     1) * sizeof(char));
    if(description == NULL) {
        fprintf(stderr, "Error - unable to allocate required memory\n");
        return 1;
    } else {
        strcat(description, " at Wabash College");
    }

    printf("Name = %s\n", name);
    printf("Description: %s\n", description);

    /* Free the allocated memory */
    free(description);

    return 0;
}
```

When the above code is compiled and executed, it will produce the following output:

```
Name = Qixin Deng
Description: Qixin Deng is a professor at Wabash College
```

If you try not reallocating additional memory, the `strcat()` function will generate an error because there will not be enough memory available to store `description`.

You will be curious why there is no error report if I use the statement below:

```
description = (char *)malloc(1 * sizeof(char));
```

I allocate only 1 byte of memory, which is clearly insufficient for storing the entire string "Qixin Deng is a professor". However, since the malloc() function successfully allocates this 1 byte, it doesn't trigger the description == NULL check, so no error message is printed. The string "Qixin Deng is a professor" is much longer than 1 byte, which leads to buffer overflow. **The strcpy() function doesn't perform any bounds checking**, so it continues writing the string into memory beyond the allocated 1 byte, potentially overwriting other memory. This is **undefined behavior**, meaning the program may continue to run, crash, or produce corrupted output, depending on what was overwritten.

In C, you **do not** need to manually free memory before calling `realloc`. When you use `realloc`, it automatically handles the memory management for you.

- If `realloc` successfully resizes the memory block, it either extends the current block in place or allocates a new block, copying the contents from the old block to the new one. In the case where a new block is allocated, `realloc` automatically frees the old memory block.
- If `realloc` fails (e.g., if there isn't enough memory to allocate), the original memory block remains unchanged, and it is your responsibility to free it later if it is no longer needed.

Therefore, calling `free()` before `realloc` is unnecessary and would lead to undefined behavior if you attempt to `realloc` memory that was already freed. Here's an example of using `realloc` correctly:

```
int *arr = malloc(5 * sizeof(int)); // Allocate memory
if (arr == NULL) {
    // handle malloc failure
}

// Reallocate memory to resize the array
int *new_arr = realloc(arr, 10 * sizeof(int)); // Resize to hold 10 integers
if (new_arr == NULL) {
    // handle realloc failure
    free(arr); // If realloc fails, free the original block
} else {
    arr = new_arr; // Point to the new block
}
```

In this example, `realloc` either extends the array or allocates a new one, freeing the old memory automatically if necessary. No need to manually call `free()` before using `realloc`.

15.5 Dynamic 2D array

A dynamic 2D array in C allows you to allocate memory at runtime, making it possible to handle data whose size is not known at compile time. There are several different ways to create dynamic 2D array.

In this method, you allocate memory for an array of pointers, where each pointer points to a dynamically allocated row.

- Allocate memory for an array of pointers, where each pointer represents a row.
- Allocate memory for each row (one-dimensional arrays) separately.
- Access elements using array[i][j].
- Deallocate memory for each row, followed by deallocating the array of pointers.

```c
#include "library.h"
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <string.h>


int main() {
    int rows = 2, cols = 3;
    int **array;

    // Step 1: Allocate memory for the array of pointers
    // pointer to pointer to integer
    // an array (pointer) that has int * elements
    array = (int **)malloc(rows * sizeof(int *));
    if (array == NULL) {
        perror("Failed to allocate memory for row pointers");
        exit(EXIT_FAILURE);
    }

    // Step 2: Allocate memory for each row
    for (int i = 0; i < rows; i++) {
        array[i] = (int *)malloc(cols * sizeof(int));
        if (array[i] == NULL) {
            perror("Failed to allocate memory for a row");
            exit(EXIT_FAILURE);
        }
    }

    // Step 3: Initialize and access elements
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            array[i][j] = i * cols + j;
            printf("%d ", array[i][j]);
        }
        printf("\n");
    }

    printf("%p\n", array);//base address
    printf("%p\n", (array + 1));// array is pointer-to-pointer, thus move to next int* (8 bytes)
    printf("%p\n", &array[1]); // based on the definition, move to the second int *
    printf("%p\n", *(array + 1)); // assigned int* in the for loop
    printf("%p\n", &array[1][0]); // The address of the first element in the row

    printf("%d\n", *(array + 1)[0]); // access the first element in the second row

    // The nature of continuous storage
    printf("%d\n", array[0][1] );  // 1
    printf("%d\n", *(array[0] + 1) );  // 1
    printf("%d\n", *(*(array + 0) + 1) );  // 1
    //Pointer arithmetic is based on its data type int(*)[3]
    printf("%d\n", *(*(array + 1) + 1) );  // 4

    // Step 4: Deallocate memory and  free the array of pointers
    for (int i = 0; i < rows; i++) {
        free(array[i]);
    }

    free(array);

    return 0;
}
```

The output could be:

```
0 1 2
3 4 5
0x600000038030
0x600000038038
0x600000038038
0x600000038050
0x600000038050
3
1
1
1
4
```

Compared with the previous static 2D array, you can see the difference in memory allocation. **Pay attention to the order of memory release.** What will happen if you release `array` before freeing `array[i]`?

15.6 C Undefined Behavior

In C, "undefined behavior" refers to the execution of code that may produce unpredictable results because the C language standard does not define what should happen. This means that when a program exhibits undefined behavior, it may produce unexpected outcomes, including program crashes, data corruption, security vulnerabilities, or it may appear to run normally.
Undefined behavior is a critical concept in C because it directly impacts the correctness and safety of a program.

Below are some common scenarios that can lead to undefined behavior:

15.6.1 Array Out-of-Bounds Access

When we try to access elements outside the bounds of an array, such as accessing an element before the 0th element or beyond the length of the array, the compiler cannot determine what is stored in the accessed memory space, leading to undefined behavior. For example:

```c
int arr[3] = {1, 2, 3};
printf("%d\n", arr[5]); // Out-of-bounds access, result is undefined
```

15.6.2 Dereferencing a Null Pointer

When we attempt to dereference a null pointer, the compiler cannot determine the contents of the memory space being accessed, leading to undefined behavior. For example:

```c
int *ptr = NULL;
printf("%d\n", *ptr); // Dereferencing a null pointer, result is undefined
```

15.6.3 Uninitialized Local Variables

Using uninitialized local variables results in undefined behavior because their value is undefined. For example:

```c
int x;
printf("%d\n", x); // x is uninitialized, result is undefined
```

15.6.4 Division by Zero (Floating-Point)

Attempting to divide a floating-point number by zero results in undefined behavior. For example:

```c
float x = 1.0;
float y = x / 0.0; // Floating-point division by zero, result.
```

15.6.5 Division by Zero (Integer)

Attempting to divide an integer by zero results in undefined behavior. For example:

```c
int x = 10;
int y = x / 0; // Integer division by zero, result is undefined
```

15.6.6 Signed Integer Overflow

When integer operations produce a result that exceeds the range of the integer type, the result is undefined. For example:

```
signed char x = 127;
x = x + 1; // Signed char overflow, result is undefined
```

15.6.7 Excessive Shift Amount

When performing a shift operation where the shift amount is greater than or equal to the number of bits in the operand, the result is undefined. For example:

```
int x = 1;
int y = x << 32; // Shift amount too large, result is undefined
```

15.6.8 Incorrect Type Casting

When performing unsafe type casting, the result is undefined. For example:

```
int *ptr = (int *)malloc(sizeof(int));
float *fptr = (float *)ptr; // Incorrect type casting, result is undefined
```

15.6.9 Memory Out-of-Bounds Access

Writing to memory that has been freed or was never allocated results in undefined behavior. For example:

```
int *ptr = (int *)malloc(sizeof(int));
free(ptr);
*ptr = 10; // Out-of-bounds memory access, result is undefined
```

15.6.10 Undefined Floating-Point Behavior

Comparing two NaN (Not-a-Number) values for equality is undefined behavior. For example:

```
float x = sqrt(-1);
float y = sqrt(-1);
if (x == y) {
    printf("NaN values are equal\n");
}
```

15.6.11 Other Undefined Behaviors

Here are some additional scenarios that can lead to undefined behavior:

- **Using Undefined Floating-Point Features:** Relying on hardware-specific or implementation-specific floating-point behavior, such as precision or rounding behavior.
- **Mismatched Function Parameters:** Calling a function with a number of arguments that does not match its definition, such as `printf("%s %d", "Name")`.
- **Modifying String Literals:** Attempting to modify the contents of a string literal, such as `char *str = "Hello"; str[0] = 'h';`.
- **Using Undefined Program States:** Relying on undefined program states, such as the initial value of a global variable.
- **Violating Strict Syntax Rules:** Violating strict C language syntax rules, such as using undeclared identifiers.
- **Race Conditions in Multithreading:** Unsynchronized access to shared resources in a multithreading environment can lead to undefined behavior.
- **Using Undefined Standard Library Function Behavior:** Certain standard library functions may exhibit undefined behavior under specific conditions, such as `fscanf()` when no input matches.

These are situations to avoid in programming because they can lead to unpredictable behavior in different environments, making the code non-portable and potentially causing errors in the program.

15.6.12 How to Avoid Undefined Behavior

To avoid undefined behavior, follow these best practices:

- **Read and Follow the C Language Standard:** Understand which operations may lead to undefined behavior and avoid them.
- **Use Static Analysis Tools:** These tools can help detect potential undefined behavior.
- **Perform Thorough Testing:** Test different execution paths of the program to ensure it runs correctly under all conditions.
- **Avoid Relying on Undefined Behavior:** Do not assume that undefined behavior will produce specific results.
- **Use Safe Functions and Libraries:** Use well-defined functions provided by the standard library, and avoid using non-standard or unsafe functions that may lead to undefined behavior.

Undefined behavior is a complex and dangerous concept in C, requiring developers to have a deep understanding of C language rules and to take appropriate measures to avoid it. By following best practices and using appropriate tools, you can minimize the risk of undefined behavior and improve the reliability and security of your program.

16 Cache

16.1 Motivation for Cache Memory

Cache memory is a crucial component in computer architecture, designed to bridge the speed gap between the processor (CPU) and the main memory (RAM). The primary motivation for introducing cache memory is to improve overall system performance by reducing the time it takes for the CPU to access data.

16.1.1 Speed Difference Between CPU and Main Memory

- **CPU Speeds vs. Memory Speeds:** Modern CPUs operate at incredibly high speeds, often measured in gigahertz. In contrast, main memory (usually Dynamic RAM, or DRAM) is much slower, measured in hundreds of megahertz. DRAM access typically costs around 100 to 200 CPU cycles or more. This discrepancy in speed creates a bottleneck when the CPU has to wait for data from the main memory.
- **Problem of Slow Memory Access:** When the CPU requests data that is not already in its internal registers, it has to fetch it from the main memory. The time taken for this operation can be several CPU cycles or more, causing the CPU to idle and wait, which significantly degrades performance.

16.1.2 The Principle of Locality

Cache memory leverages the **Principle of Locality** to enhance performance. The Principle of Locality consists of two types:

- **Temporal Locality (Locality in Time):**
  - This principle states that if a data item (like an instruction or piece of data) is accessed, it is likely to be accessed again in the near future.
  - For example, loops in a program repeatedly access the same set of instructions, and frequently used variables tend to be accessed repeatedly.
- **Spatial Locality (Locality in Space):**
  - This principle states that if a particular data item is accessed, other data items with nearby memory addresses are also likely to be accessed soon.
  - For example, when accessing an element in an array, it is likely that nearby elements will be accessed next.

By storing recently accessed data and nearby data in a smaller, faster memory (cache), the system can reduce the time the CPU spends waiting for data.

- **Servicing Most Accesses from a Small, Fast Memory:** Cache memory is typically made from faster but smaller and more expensive types of memory (like Static RAM, or SRAM), which can be accessed much faster than DRAM.
- **Minimizing the Number of Times the CPU Accesses Main Memory:** By storing frequently accessed data and instructions in the cache, the system reduces the number of times it needs to access slower main memory.
- **Bandwidth Optimization:** Cache memory also helps reduce the amount of data transferred between the CPU and the main memory. By servicing most of the CPU's data and instruction requests, the cache reduces the traffic on the memory bus, thereby optimizing bandwidth usage and minimizing contention between different parts of the system that might need to access main memory.

16.2 Levels of the Memory Hierarchy

The memory hierarchy in computer architecture is organized **to balance speed, cost, and size**. Each level of the hierarchy serves a different purpose and has its unique characteristics. This section provides a detailed description of each level.

- **CPU Registers** are the **fastest** and **smallest** type of memory available in a computer system. They are located **inside the processor** and typically hold **hundreds of bytes**, and their access time is in the order of **a few nanoseconds** (less than 10 ns). Because they are embedded within the CPU, they are incredibly fast but also expensive and limited in size. Registers are used to **hold temporary data, intermediate results of calculations, and addresses needed for immediate execution by the CPU**. Since they have the shortest access time, they are at the top of the memory hierarchy.

- **Cache Memory** is a **small, high-speed memory located close to the CPU**. Cache memory is designed to **store copies of frequently accessed data from the main memory** to reduce the average time required to access data. The cache can be divided into several levels, such as Level 1 (L1), Level 2 (L2), and Level 3 (L3), with L1 being the fastest and smallest, and L3 being larger but slower. Cache memory sizes typically range from **kilobytes (KB) to megabytes (MB)**, and their access time is between 10 to 100 nanoseconds. The cost per bit is higher than the main memory but significantly lower than CPU registers.
- **Main Memory**, or Random Access Memory (RAM), is the primary storage used to **store the operating system, application programs, and data that the CPU needs in real-time.**. It is larger than cache memory and provides a reasonable balance between speed, size, and cost. Main memory is typically measured in **megabytes (MB) to gigabytes (GB)**, with access times ranging from **100 nanoseconds to 1 microsecond**. The cost per bit for RAM is lower than that of cache memory, making it suitable for storing larger amounts of data. However, RAM is still **volatile**, meaning that it loses all stored information when the power is turned off.
- **Disk Storage** (such as Hard Disk Drives (HDDs) or Solid-State Drives (SSDs)) provides a **much larger storage capacity** than main memory but with significantly slower access times. Disk storage is used for long-term data storage and can hold **gigabytes (GB) to terabytes (TB) of data**. Access times for disk storage are typically measured in **milliseconds (ms)**, making it considerably slower than main memory. The cost per bit is much lower than that of RAM, which allows for large storage capacities. Disk storage is **non-volatile**, meaning it retains data even when the computer is powered off. It is used to store files, applications, and data that do not need to be accessed as quickly as data in main memory.
- **Tape Storage** provides virtually infinite storage capacity at a very low cost per bit, but with the **slowest access times of all memory types in the hierarchy**. Access times for tape storage are typically in the range of **seconds to minutes**. Tape is a sequential access medium, meaning that data **must be read in a specific order**, making random access operations very slow. It is ideal for storing large volumes of data that do not need to be accessed frequently, such as **backup copies, logs, and historical records.**

Based on the discussion above, we can summarize the characteristics of the memory hierarchy:

- **Speed:** The closer the memory is to the CPU, the faster it is. Registers and cache memory are extremely fast, while disk and tape storage are much slower.
- **Size:** As we move down the hierarchy from registers to tape storage, the size of the memory increases significantly. Registers are very small, while disk and tape storage can hold massive amounts of data.
- **Cost:** The cost per bit decreases as we move down the hierarchy. Registers are the most expensive, while tape storage is the cheapest.
- **Volatility:** Memory closer to the CPU, like registers and cache, is volatile and loses data when power is lost. Disk and tape storage are non-volatile and retain data permanently.

16.3 Cache Block (Cache Line)

In the context of cache memory, a block (also known as a cache block or cache line) is a **small, fixed-size unit of data** that is transferred between the cache and the main memory. **It is the minimum unit of information that can either be present or not present in a cache at any given time.** The size of a cache block is typically a power of two, such as 16, 32, 64, or 128 bytes.

Each block in the cache is associated with a specific range of addresses in the main memory. When the CPU accesses a particular memory address, the cache checks whether the corresponding block is present in the cache. If the block is present, it is called a **cache hit**; if not, it is a **cache miss**, and the block must be fetched from the main memory.

Cache blocks can have different states based on the cache coherence protocol or the cache replacement policy in use. Common states include:

- **Valid**: The block contains valid data.
- **Invalid**: The block does not contain valid data or has been marked for replacement.
- **Dirty**: The block has been modified in the cache but not yet written back to the main memory (relevant in write-back caches).

16.3.1 Why Are Blocks Important in Cache Design?

- **Improving Access Times**: By transferring data in blocks, caches take advantage of spatial locality, as accessing one memory address makes it likely that nearby addresses will also be accessed soon. This reduces the number of accesses to slower main memory.
- **Minimizing Memory Bandwidth Usage**: Fetching data in blocks, rather than individual bytes, reduces the overhead associated with memory access and minimizes memory bandwidth usage.
- **Cache Hit Rate**: The size of the cache block directly impacts the cache hit rate. Smaller block sizes may lead to a higher number of cache misses if the data being accessed is spread out across different memory locations. Conversely, larger block sizes may increase the likelihood of cache hits but can also cause cache pollution (unused data occupying cache space).

16.3.2 Example of Cache Block Usage

Consider a scenario where the CPU needs to read data from memory address 0x100. The following steps outline how cache blocks are used:

1. Cache Check: The CPU checks if the data at address 0x100 is in the cache. The cache controller uses the address's tag to determine whether the corresponding block is present.
2. Cache Miss: If the block containing address 0x100 is not in the cache, a cache miss occurs. The cache controller fetches the block of data that includes address 0x100 from the main memory. For example, if the block size is 64 bytes, the block may contain addresses from 0x100 to 0x13F.
3. Block Load: The fetched block is loaded into the cache. Future accesses to any address within this block (e.g., 0x101, 0x102, ..., 0x13F) can now be serviced by the cache, resulting in faster access times.
4. Cache Hit: If the CPU later accesses an address within the loaded block (such as 0x104), the cache provides the data directly, avoiding a slow access to main memory.

16.4 Direct Mapped Cache

In a direct-mapped cache, each block from the main memory can be mapped to exactly one location in the cache. This is a simple and efficient way to organize cache memory but may lead to more cache misses in certain scenarios due to conflicts. Let's assume 1 kilobyte cache with a 32-byte cache block size. So there are a total of 32 blocks.

16.4.1 Address Breakdown for Direct Mapped Cache

To understand how the cache maps memory addresses, we need to consider how the **memory address** is divided:

- **Block Offset**: The block offset determines the specific byte within a block. Since each block is 32 bytes (or 2^5 bytes), the block offset requires 5 bits to uniquely identify any byte within a block.
- **Index**: The index determines which cache block the data will be placed in. Since there are 32 blocks in the cache (or 2^5 blocks), the index requires 5 bits to identify which block to use.
- **Tag**: The remaining bits of the memory address are used as the tag. The tag is used to verify whether the data stored in a cache block corresponds to the requested memory address. The number of tag bits depends on the total number of address bits minus the index and block offset bits. If the system uses a 32-bit address, the address breakdown would be as 32-(5+5)=22bits.

16.4.2 Example of Cache Operation

Let's walk through a simplified example to see how this direct-mapped cache works:
Suppose the CPU needs to access the memory address 0x00400020. The 32-bit address 0x00400020 in binary format looks like this:

```
0000 0000 0100 0000 0000 0000 0010 0000
```

From this address, we extract the relevant bits for the tag, index, and block offset:

- Block Offset: The last 5 bits (00000) represent the block offset.
- Index: The next 5 bits (00001) represent the cache index.
- Tag: The remaining 22 bits (0000 0000 0100 0000 0000 00) represent the tag.

The cache controller uses the index (00000) to locate the appropriate block in the cache. In this case, it checks block number 0 in the cache. The cache controller then compares the tag of the stored block at index 0 with the requested tag (0000 0000 0100 0000 0000 00). If the tags match, it means the requested data is present in the cache, resulting in a cache hit. If the tags do not match, it results in a cache miss. The block containing the requested address is then fetched from the main memory and loaded into the cache block at index 0.

16.4.3 Metadata in Cache

A cache is indeed used to store data that the CPU frequently accesses to reduce the time required to fetch this data from the slower main memory. However, the cache also needs to store some additional information, known as metadata, to efficiently manage and locate the data. This metadata includes the tag and some other control bits.

Each cache block (cache line) in this cache can store 32 bytes of actual data. However, each line also has an additional area to store the tag and other control bits. For example, if each tag is 22 bits (as in our example with a 32-bit address), each cache line will store 22 bits (approximately 3 bytes) of metadata. **When the CPU accesses a memory address, it breaks the memory address down into the tag, index, and offset. The index is used to select which cache line to check. The tag stored in that line is compared with the tag portion of the memory address. If they match (and the valid bit is set), it's a cache hit; otherwise, it's a cache miss. The offset is then used to locate the exact byte within the 32-byte block of data stored in that cache line.** Here's a simplified diagram of what each cache line contains:

```
+-----------------------------------------------------+
|     Tag     |   Valid Bit   |     Data Block (32 B)   |
|   (22 bits) |   (1 bit)     |   (32 bytes of data)    |
+-----------------------------------------------------+
```

16.4.4 Actual Size of The Cache

The actual size of the cache is indeed slightly larger than the stated capacity of **1 KB (1024 bytes)**. This is because, in addition to storing the **data blocks** (the 1 KB of actual data), the cache must also store **metadata** for each cache line. This metadata includes the **tag** bits and other control bits, such as the **valid** bit or **dirty** bit.

In the above example, the cache size is **1 KB** (kilobyte), which refers only to the space allocated for **data storage**.

- The cache is divided into **32 blocks**, each **32 bytes** in size:

$$\text{Number of blocks} = \frac{\text{Cache size}}{\text{Block size}} = \frac{1024\,\text{bytes}}{32\,\text{bytes per block}} = 32\,\text{blocks}$$

- Thus, the cache can hold **1024 bytes of data** from the main memory, but this does not include the additional storage required for metadata.

Each cache line (block) in a direct-mapped cache needs to store some metadata to keep track of which part of the main memory it corresponds to:

- **Tag Bits:**
  - In our example, there are **22 tag bits** per cache line (derived from a 32-bit address with 5 bits for index and 5 bits for offset).
  - To calculate the total number of tag bits in the cache:

$$\text{Total Tag Bits} = \text{Number of blocks} \times \text{Tag bits per block} = 32 \times 22 = 704\,\text{bits}$$

- **Control Bits:**
  - Each cache line typically includes at least one **valid bit** to indicate whether the cache line contains valid data.
  - If the cache uses a **write-back policy**, it might also include a **dirty bit** to indicate whether the data in the cache line has been modified but not yet written back to the main memory.
  - Assuming **1 valid bit** per block, the total number of control bits would be:

$$\text{Total Control Bits} = \text{Number of blocks} \times \text{Control bits per block} = 32 \times 1 = 32\,\text{bits}$$

To find the actual total size of the cache, we need to consider both the data storage and the metadata:

1. **Convert Tag and Control Bits to Bytes:**

- Total Tag Bits: **704 bits** = $\frac{704}{8} = 88$ bytes
- Total Control Bits: **32 bits** = $\frac{32}{8} = 4$ bytes

2. **Compute Total Cache Size:**
   - Data Size: **1024 bytes**
   - Metadata Size: **88 bytes (tags) + 4 bytes (control bits) = 92 bytes**
3. **Actual Cache Size:**

$$\text{Actual Cache Size} = \text{Data Size} + \text{Metadata Size} = 1024\,\text{bytes} + 92\,\text{bytes} = 1116\,\text{bytes}$$

Thus, the **actual size** of the cache, considering both data and metadata, is **1116 bytes** — which is slightly larger than the stated **1 KB (1024 bytes)**.

16.4.5 What Is A Conflict Miss?

A conflict miss (or collision miss) occurs when two or more different memory addresses map to the same cache line due to the direct mapping scheme. When this happens, every time a new block is loaded into that specific cache line, it replaces the existing block, even if the existing block might still be needed soon. **Why Does Direct Mapped Cache Lead to More Cache Misses Due to Conflicts?**

- **Fixed Mapping of Memory Blocks:**
  - In a direct-mapped cache, every memory block has only one possible cache line it can occupy. This fixed mapping increases the likelihood of multiple memory blocks competing for the same cache line.
  - For example, if two frequently accessed memory addresses (e.g., `0x00400020` and `0x00C00020`) both map to cache line 5, accessing one will cause the other to be evicted. When the second address is accessed again, a **cache miss** occurs because it was evicted when the first address was loaded.
- **Thrashing:**
  - Direct-mapped caches lack flexibility in placing memory blocks because each block can only go to one specific cache line. If two or more blocks map to the same line, they will continually replace each other, causing frequent cache misses (this phenomenon is known as **thrashing**).

16.5 Two-Way Set Associative Cache

A **two-way set associative cache** is a type of cache organization that provides a compromise between a **direct-mapped cache** and a **fully associative cache**. In this type of cache, the cache is divided into sets, and each set contains **two cache lines** (or blocks). A memory block can be placed in either of the two lines within a particular set, providing more flexibility than a direct-mapped cache.

- **Cache Organization:**
  - The cache is divided into multiple **sets**, with each set containing **two cache lines**.
  - Each memory address maps to a specific set, but within that set, the memory block can be placed in **either of the two cache lines**.
  - This reduces the likelihood of conflict misses compared to a direct-mapped cache, as each block has two possible locations.
- **Address Breakdown:** A memory address is typically divided into three parts: **tag**, **set index**, and **block offset**.
  - **Set Index:** Determines which set the memory block maps to.
  - **Tag:** Identifies the specific memory block within the set.
  - **Block Offset:** Identifies the specific byte within the block.

16.5.1 Example of Cache Operation

Example Configuration: 1 KB Two-Way Set Associative Cache with 32 Byte Blocks:

- **Cache Size:** 1 KB (1024 bytes)
- **Block Size:** 32 bytes
- **Associativity:** 2-way (each set has 2 lines)
- **Number of Sets:**

$$\text{Number of Sets} = \frac{\text{Cache Size}}{\text{Block Size} \times \text{Number of Ways}} = \frac{1024\,\text{bytes}}{32\,\text{bytes per block} \times 2} = 16\,\text{sets}$$

- The cache has **16 sets**, each containing **2 cache lines** of 32 bytes.

Consider a system with a **32-bit memory address** and a cache configuration as described above:

- **Block Offset:**
  - Each block is **32 bytes** (2^5), so the block offset requires **5 bits** to address any byte within the block.
- **Set Index:**
  - There are **16 sets** (2^4), so the set index requires **4 bits** to identify the set.
- **Tag:**

$$\text{Tag Bits} = 32 - (\text{Set Index Bits} + \text{Block Offset Bits}) = 32 - (4 + 5) = 23\,\text{bits}$$

Suppose the CPU needs to access memory address `0x00400028`:

- **Address Breakdown:** The 32-bit address `0x00400028` in binary is:

$$0000\,0000\,0100\,0000\,0000\,0000\,0010\,1000$$

- **Block Offset:** Last 5 bits (`01000`) — identifies the byte within the block. - **Set Index:** Next 4 bits (`0000`) — identifies the set within the cache. - **Tag:** Remaining 23 bits (`0000 0000 0100 0000 0000 00`) — identifies the memory block.

- **Cache Mapping:**
  - The cache controller uses the **set index** (`0000`) to locate the corresponding set in the cache (set 0 in this case).
  - Both cache lines in set 0 are checked simultaneously to see if the tag (`0000 0000 0100 0000 0000 00`) matches.
- **Cache Hit or Miss:**
  - **Cache Hit:** If the tag matches in one of the two lines, it is a cache hit, and the data is accessed.
  - **Cache Miss:** If neither line matches the tag, it is a cache miss, and the data is fetched from the main memory. One of the two lines in set 0 will be replaced based on the replacement policy (e.g., LRU).

16.5.2 actual Size of The Cache

To determine the actual size of the cache, we consider:

1. **Data Storage Size:**
   - The cache can hold **1024 bytes** of data.
   - Each block (cache line) is **32 bytes**.
   - There are **32 cache lines** in total (16 sets × 2 lines per set).

   The total data storage size is:

$$\text{Data Storage Size} = 32\,\text{cache lines} \times 32\,\text{bytes per line} = 1024\,\text{bytes}$$

2. **Metadata Storage Size:**
   - **Tag Bits:**

- Each memory address is **32 bits**.
- We have **5 bits** for the block offset and **4 bits** for the set index.
- The number of tag bits per cache line is:

$$\text{Tag Bits} = 32 - (\text{Set Index Bits} + \text{Block Offset Bits}) = 32 - (4 + 5) = 23\,\text{bits}$$

- Each set has **2 cache lines**, so each set requires $2 \times 23 = 46$ tag bits.
- There are **16 sets** in total, so the total number of tag bits is:

$$\text{Total Tag Bits} = 16\,\text{sets} \times 46\,\text{tag bits per set} = 736\,\text{bits}$$

- Convert the total tag bits to bytes:

$$\text{Total Tag Bytes} = \frac{736\,\text{bits}}{8} = 92\,\text{bytes}$$

- **Control Bits:**
  - Assume **1 valid bit** per line and **1 dirty bit** per line (total 2 control bits per line).
  - There are **32 cache lines** in total, so the total number of control bits is:

$$\text{Total Control Bits} = 32\,\text{lines} \times 2\,\text{bits per line} = 64\,\text{bits}$$

  - Convert the control bits to bytes:

$$\text{Total Control Bytes} = \frac{64\,\text{bits}}{8} = 8\,\text{bytes}$$

- **Total Metadata Size:**

$$\text{Total Metadata Size} = \text{Total Tag Bytes} + \text{Total Control Bytes} = 92\,\text{bytes} + 8\,\text{bytes} = 100\,\text{bytes}$$

The **actual size** of the cache is the sum of the data storage size and the metadata storage size:

$$\text{Actual Cache Size} = \text{Data Storage Size} + \text{Metadata Size} = 1024\,\text{bytes} + 100\,\text{bytes} = 1124\,\text{bytes}$$

16.5.3 Comparison to Direct Mapped Cache

- **Flexibility:** Unlike a direct-mapped cache, where each memory block has only one possible cache line, a two-way set associative cache provides two possible locations for each memory block, reducing the likelihood of conflict misses.
- **Performance:** Generally, a two-way set associative cache will have a lower cache miss rate than a direct-mapped cache due to the additional flexibility in placing blocks.
- **Cost and Complexity:** A two-way set associative cache is more complex than a direct-mapped cache but less complex than a fully associative cache.

16.6 Fully Associative Cache

A **Fully Associative Cache** is a cache structure where any block of memory can be placed in any cache line. There are no restrictions or fixed mappings that determine which memory block goes into which cache line. Instead, every cache line can store any memory block, providing maximum flexibility. This organization reduces the number of conflict misses to zero since all memory blocks can be placed anywhere in the cache.

16.6.1 Key Characteristics of a Fully Associative Cache

- **Cache Organization:**
  - The cache is organized as a single set containing multiple cache lines. Each cache line can store any block from the main memory.
  - The cache controller must compare the tag of every cache line to find a match, leading to increased hardware complexity and power consumption.
- **Address Breakdown:**
  - A memory address is divided into two parts: the **tag** and the **block offset**.
    - **Tag:** Used to identify which memory block is stored in a particular cache line.

    * **Block Offset:** Used to identify the exact byte within a cache line.

- **Mapping of Memory Addresses:**
  - When the CPU accesses a memory address, the **tag** part of the address is used to search through all cache lines to determine if the block is present in the cache.
  - The cache controller compares the tag of the requested address with the tag stored in each cache line simultaneously.
- **Cache Hit and Miss:**
  - **Cache Hit:** If the tag of the requested address matches the tag stored in any cache line, it is a **cache hit**, and the data is accessed from the cache.
  - **Cache Miss:** If no tags match, it is a **cache miss**, and the data must be fetched from the main memory and loaded into one of the cache lines.

16.6.2 Example of Cache Operation

Consider a system with a **32-bit memory address** and a cache configuration as described: **1 KB Fully Associative Cache with 32 Byte Blocks**:

- **Cache Size:** 1 KB (1024 bytes)
- **Block Size:** 32 bytes
- **Number of Cache Lines:**

$$\text{Number of Cache Lines} = \frac{\text{Cache Size}}{\text{Block Size}} = \frac{1024\,\text{bytes}}{32\,\text{bytes per block}} = 32\,\text{lines}$$

- The cache has **32 cache lines**, each capable of holding any block of 32 bytes from the main memory.
- **Block Offset:**
  - Each block is **32 bytes** (2^5), so the block offset requires **5 bits** to address any byte within the block.
- **Tag:**

$$\text{Tag Bits} = 32 - \text{Block Offset Bits} = 32 - 5 = 27\,\text{bits}$$

Suppose the CPU needs to access memory address `0x00400028`:

- **Address Breakdown:** The 32-bit address `0x00400028` in binary is:

$$0000\,0000\,0100\,0000\,0000\,0000\,0010\,1000$$

- **Block Offset:** Last 5 bits (`01000`) — identifies the byte within the block. - **Tag:** Remaining 27 bits (`0000 0000 0100 0000 0000 000`) — identifies the memory block.

- **Cache Mapping:**
  - The cache controller searches all cache lines simultaneously to find a match for the tag (`0000 0000 0100 0000 0000 000`).

16.6.3 Advantages and Disadvantages of a Fully Associative Cache

Advantages:

- **Minimized Conflict Misses:** Since any block can be placed in any cache line, fully associative caches virtually eliminate conflict misses.
- **Flexibility:** Fully associative caches offer maximum flexibility for data placement, which can significantly reduce the miss rate.
- **Higher Hit Rate:** Due to the ability to place any data block anywhere, fully associative caches generally have the highest hit rate.

Disadvantages

- **Increased Complexity and Cost:** Fully associative caches require hardware capable of searching all cache lines in parallel to find a match, which is more complex and expensive.
- **Higher Power Consumption:** The parallel search required in fully associative caches consumes more power.
- **Longer Access Time:** The time to search all cache lines can be longer, which may result in slightly longer access times.

16.7 Multi-Level Cache

Modern computer systems use a **multi-level cache architecture** to optimize memory access and improve overall performance. This architecture consists of several levels of caches, typically labeled as **L1 (Level 1), L2 (Level 2),** and **L3 (Level 3) caches**. Each level is designed to provide different trade-offs between speed, size, and complexity to reduce the time the CPU spends waiting for data from the slower main memory.

16.7.1 Key Characteristics of Multi-Level Caches

- **Multi-Level Cache Hierarchy:**
  - **L1 Cache:** The fastest and smallest cache, located closest to the CPU cores.
  - **L2 Cache:** Larger and slower than L1, serves as an intermediate buffer.
  - **L3 Cache:** The largest and slowest among the caches, shared among multiple cores.
- **Purpose of Multi-Level Caches:**
  - To balance the trade-offs between speed, size, and cost.
  - Allows frequently accessed data to be kept close to the CPU in a small, fast cache (L1), while less frequently accessed data can reside in larger, slower caches (L2 or L3).
- **General Architecture:**
  - Data is first checked in the L1 cache. If it is not found (a **cache miss**), the search moves to the L2 cache, and finally, the L3 cache.
  - This approach helps reduce the average memory access time and minimizes the number of accesses to the slower main memory.

Specifically,

1. L1 Cache
   - **Type:** Typically split into two parts:
     - **L1 Data Cache (L1d):** Stores frequently accessed data.
     - **L1 Instruction Cache (L1i):** Stores frequently accessed instructions (program code).
   - **Organization:**
     - The L1 cache is usually **small** (16 KB to 64 KB) but very **fast**.
     - Often implemented as a **4-way or 8-way set-associative cache**.
   - **Purpose:** Provides the **fastest possible** access times, typically in the range of 1-3 CPU cycles.
   - **Access Latency:** Very low (1-3 cycles).
2. L2 Cache
   - **Type:** Unified cache (stores both data and instructions).
   - **Organization:**
     - Larger than L1 cache (typically 256 KB to 1 MB per core) but still relatively fast.
     - Generally implemented as an **8-way to 16-way set-associative cache**.
   - **Purpose:** Serves as an intermediate buffer between the fast L1 cache and the larger, slower L3 cache or main memory.
   - **Access Latency:** Moderate (3-10 cycles).
3. L3 Cache
   - **Type:** Shared cache among multiple CPU cores.
   - **Organization:**
     - Much larger than L1 and L2 caches (ranging from 2 MB to 64 MB or more).
     - Typically implemented as a **16-way to 32-way set-associative cache**.
   - **Purpose:** Stores data that may be shared across multiple cores, reducing access to main memory.
   - **Access Latency:** Higher (10-20 cycles).

16.7.2 Example of Cache Operation

For each of the levels, the cache performs the same as N-Way set associate cache. Suppose the CPU core needs to access data at a specific memory address:

1. **L1 Cache Check:**
   - If the data is found, it is a **cache hit**, and the CPU accesses it with minimal delay.
   - If not, it results in an **L1 cache miss**.
2. **L2 Cache Check:**
   - Upon an L1 miss, the CPU checks the **L2 cache**.
   - If the data is found, it is fetched and copied to the L1 cache for faster future access.
   - If not, it results in an **L2 cache miss**.
3. **L3 Cache Check:**
   - The CPU checks the **L3 cache**.
   - If the data is found, it is a **cache hit**, and the data is fetched to the L1 cache.
   - If not, it results in an **L3 cache miss**.
4. **Main Memory Access:**
   - On an L3 cache miss, the CPU accesses the **main memory** to fetch the data.
   - The data is loaded into the L1 cache as appropriate.

16.7.3 Exclusive vs. Inclusive Cache Hierarchy

The way data is shared across cache levels also depends on whether the system uses an inclusive or exclusive cache hierarchy.

- **Inclusive Cache Hierarchy**: In this system, data in L1 is always present in L2 and L3. So, when data is loaded into L1, it is also typically present in the lower levels (L2 and L3).However, when the data is initially fetched (due to an L3 miss), it still typically goes only to L1 first. If the L1 line is evicted, it might then be "pushed down" to L2/L3. Example: Intel processors often use inclusive caches.
- **Exclusive Cache Hierarchy**: In this system, data exists in only one level at a time, so L1, L2, and L3 do not store duplicates. For example, if a line is moved from L1 to L2, it is evicted from L1. This avoids data duplication and maximizes cache space at each level. Example: AMD processors often use exclusive caches.

16.7.4 Advantages and Disadvantages of Multi-Level Cache Architecture

Advantages:

- **Reduced Latency:** Multi-level caches reduce average memory access time.
- **Improved Throughput:** Balances the load and prevents bottlenecks, improving performance.
- **Efficient Use of Cache Space:** Ensures smaller, faster caches are used for the most critical data.
- **Support for Multi-Core Processors:** L3 (and optional L4) caches are shared across multiple cores.

Disadvantages:

- **Increased Complexity:** Managing multiple levels of caches adds complexity to the CPU design.
- **Higher Cost and Power Consumption:** More levels require more transistors and power.
- **Diminishing Returns:** Benefits diminish beyond a certain point for some applications.

16.8 Cache Block Replacement Policy

16.8.1 Least Recently Used (LRU) Replacement Policy

The **Least Recently Used (LRU)** replacement policy evicts the cache block that has not been accessed for the longest time. The idea is based on the principle of **temporal locality**, which suggests that data accessed recently is more likely to be accessed again soon.

16.8.2 How It Works

- Maintain a list or stack of all the cache blocks. Each time a block is accessed, it is moved to the top of the list.
- When a replacement is needed, the block at the bottom of the list (the least recently used) is selected for eviction.

16.8.3 Implementation

- **Counter-Based Implementation:** Each cache block is associated with a counter that is updated every time the block is accessed. When a replacement is needed, the block with the smallest counter value (indicating the least recent access) is evicted.
- **Stack-Based Implementation:** Use a linked list or stack to keep track of the order of access. When a block is accessed, it is moved to the top of the stack.

16.8.4 Advantages and Disadvantages of LRU

Advantages:

- Effectively reduces cache misses by evicting the least useful block.
- Well-suited for workloads with strong temporal locality.

Disadvantages:

- Implementation can be complex, especially in hardware.
- Maintaining and updating the LRU list or counters can be costly in terms of time and resources.

16.8.5 Comparison of Cache Replacement Policies

Policy	Complexity	Performance	Overhead	Suitable For
LRU	High	Good	High	Workloads with strong temporal locality
FIFO	Low	Moderate	Low	Simple applications or hardware implementations
LFU	High	Moderate	High	Workloads with long-term access patterns
Random	Low	Low to Moderate	Low	Simple hardware implementations, unpredictable workloads
OPT	Very High	Optimal	Not Implementable	Theoretical analysis only
PLRU	Moderate	Good	Moderate	Systems where LRU is too costly to implement

Table 5: Comparison of Cache Replacement Policies

16.9 Cache Write Policy

In computer architecture, a **cache write policy** determines how modifications to data in the cache are propagated to the main memory. When the CPU writes data to the cache, it must eventually ensure that the changes are reflected in the main memory. There are two primary cache write policies used to manage this process:

- **Write-Through**
- **Write-Back**

Each of these policies has its own advantages and disadvantages, and the choice of policy can significantly affect the performance and consistency of a computer system.

16.9.1 Write-Through Cache Policy

s The **Write-Through** cache policy ensures that every write operation to the cache is immediately followed by a write to the main memory. This means that the main memory always contains the most recent copy of the data.

- When the CPU writes data to the cache, the data is simultaneously written to the main memory.
- The cache and main memory are always kept in sync, which simplifies data consistency.
- This policy typically uses a mechanism called a **write buffer**(controlled by memory controller) to store write operations temporarily, reducing the delay caused by waiting for the main memory write to complete.

Advantages:

- **Data Consistency:** The main memory always has the most up-to-date data, which simplifies the design of the memory hierarchy and avoids inconsistencies.
- **Simple Implementation:** Easier to implement in hardware since every write operation is immediately propagated to the main memory.
- **Suitable for Multi-Processor Systems:** Ideal for systems where multiple processors may read from or write to the same memory location.

Disadvantages:

- **Higher Latency:** Every write operation involves writing to both the cache and the main memory, which can slow down the overall write performance.
- **Increased Memory Bandwidth Usage:** Generates more traffic to the main memory, as every write to the cache results in a write to the memory.
- **Inefficient for High Write Frequency:** May lead to performance bottlenecks in applications with frequent write operations.

16.9.2 Write-Back Cache Policy

The **Write-Back** cache policy allows the CPU to write data to the cache without immediately writing it to the main memory. The modified data is written to the main memory only when it is evicted from the cache.

- When the CPU writes data to the cache, the data is marked as **dirty**, indicating that it has been modified.
- The modified data is kept in the cache until the cache block is evicted (replaced with new data).
- When a cache block marked as dirty is evicted, the data is then written back to the main memory.

Advantages:

- **Reduced Memory Bandwidth Usage:** Multiple writes to the same cache block only require a single write to the main memory when the block is evicted, reducing the total number of memory writes.
- **Lower Latency:** Write operations are faster since they do not immediately require writing to the main memory.
- **Efficient for High Write Frequency:** Well-suited for applications with frequent write operations, as it reduces the number of memory accesses.

Disadvantages:

- **Data Inconsistency:** The main memory may not always have the most recent data, which can lead to inconsistencies if not managed carefully.
- **Complex Implementation:** More complex hardware is required to track which cache blocks are dirty and to handle the writing back of data to the main memory.
- **Not Ideal for Multi-Processor Systems:** Requires additional mechanisms, such as cache coherence protocols, to maintain consistency across multiple caches in a multi-processor environment.

16.9.3 Comparison of Write-Through and Write-Back Policies

Aspect	Write-Through	Write-Back
Data Consistency	Always consistent with main memory	May be inconsistent until write-back
Memory Bandwidth Usage	High, due to frequent writes to main memory	Low, as multiple writes may result in a single memory write
Latency	Higher latency for writes	Lower latency for writes
Implementation Complexity	Simpler to implement	More complex due to tracking of dirty blocks
Suitability for Multi-Processor Systems	More suitable due to data consistency	Less suitable; requires cache coherence protocols
Performance with High Write Frequency	Lower performance due to high memory bandwidth usage	Higher performance due to reduced memory writes

Table 6: Comparison of Write-Through and Write-Back Cache Policies

16.9.4 MESI Cache Coherence Protocol

The MESI (Modified, Exclusive, Shared, Invalid) Cache Coherence Protocol is a widely used mechanism in multiprocessor systems to maintain consistency among local caches. It ensures that when multiple processors access shared memory, they see a coherent view of data, preventing inconsistencies caused by updates. Each cache line can be in one of four states: Modified (the line has been changed and is not yet written back to memory), Exclusive (the line is in only one cache and is identical to memory), Shared (the line may be in multiple caches and matches memory), or Invalid (the line is outdated or no longer valid). The protocol facilitates transitions between these states based on cache access, reads, and writes, thus ensuring proper data synchronization across different caches.

MESI Protocol States:

1. **Modified (M)**: The cache line is present only in the current cache, has been modified, and is different from main memory. The cache is responsible for writing the data back to memory when it's evicted.
2. **Exclusive (E)**: The cache line is present only in the current cache, but it is clean (unmodified) and identical to the copy in main memory. No other cache holds this line.
3. **Shared (S)**: The cache line is present in the current cache and possibly in other caches. The line is identical to main memory and has not been modified by any cache.
4. **Invalid (I)**: The cache line is not valid; it contains no useful data.

MESI State Transitions: Below are the common state transitions that happen during various cache operations (read, write) and based on interactions with other caches (via snooping):

1. Read Hit
   - M → M: The cache already holds a modified copy of the data. No changes needed.
   - E → E: The cache already holds an exclusive clean copy. No changes needed.
   - S → S: The cache already holds a shared copy. No changes needed.
   - I → S: If the cache line is invalid, it transitions to S after fetching the data from memory or another cache (read-for-own state). This happens if other caches have the same line in S or E.
2. Read Miss
   - I → S: If the cache performs a read miss and no other cache has the line in M, it fetches the data from memory and enters the S state.
   - I → E: If no other cache holds the data (no other cache responds during snooping), the cache can fetch the data and move into the E state, indicating that it holds the exclusive copy of the clean data.
3. Write Hit
   - M → M: The cache already holds a modified copy, and it writes the data to its cache without additional transitions.
   - E → M: The cache transitions from E to M when it writes to the cache line, since the cache line has now been modified.

- S → M: The cache transitions from S to M on a write, but before writing, it must broadcast an invalidate message to all other caches, causing them to invalidate their copies.

4. Write Miss
   - I → M: If a cache wants to write to an invalid cache line, it fetches the data from memory or another cache and transitions directly to the M state, after invalidating copies in other caches (write-for-own state).
5. Invalidate
   - S → I: If another cache writes to a line that the current cache holds in the S state, the current cache invalidates its copy and transitions to I.
   - E → I: If another cache writes to a line that the current cache holds in the E state, the current cache invalidates its copy and transitions to I.
6. Cache-to-Cache Transfer
   - M → S: When another cache requests the line and the current cache holds it in the M state, the current cache downgrades the line to S and writes it back to main memory. The requesting cache will get the line in S as well.
   - E → S: If the cache holds an exclusive clean copy and another cache requests the line, the current cache transitions to S because now the line is shared between caches.

Event	**M**	**E**	**S**	**I**
Read Hit	M → M	E → E	S → S	I → S (or E)
Read Miss	-	-	-	I → S (or E)
Write Hit	M → M	E → M	S → M	-
Write Miss	-	-	-	I → M
Invalidate	-	E → I	S → I	-
Cache-to-Cache Transfer	M → S	E → S	-	-

Table 7: Summary of MESI State Transitions

Additional Notes:

- Snooping: MESI uses snooping on a shared bus to maintain coherence between caches. If a cache wants to read or write to a block, it will snoop (broadcast a request) to check other caches' states and take appropriate action based on the response.
- Invalidate: For write-back caches, an invalidation request is sent to other caches when a cache writes to a line, forcing them to invalidate their copies of the data.

Here is an example:

indx	**Local Request**	**P1**	**P2**	**P3**
0	**Initially**	-	-	-
1	R1	E	-	-
2	W1	M	-	-
3	R3	S	-	S
4	W3	I	-	M
5	R1	S	-	S
6	R3	S	-	S
7	R2	S	S	S

Table 8: An example of how MESI works. All operations are to the same cache block (Example: "R3" means read block by processor 3).

17 MIPS Architecture Overview

One key benefit of learning assembly language is gaining a deeper understanding of how to write more efficient code in high-level languages. Additionally, it helps understand the role of a compiler and the internal organization of computers at a fundamental level, potentially opening up opportunities in the growing field of embedded processors.

MIPS (Microprocessor without Interlocked Pipeline Stages) is a **RISC** (Reduced Instruction Set Computer) architecture developed in the 1980s. It is widely used in embedded systems, networking equipment, and educational environments. MIPS is known for its simplicity and efficient pipeline design, making it a good platform for learning assembly language and computer architecture concepts.

17.1 RISC vs CISC

RISC (Reduced Instruction Set Computer) is a type of computer architecture that focuses on simplifying the instruction set of a CPU to achieve higher performance. RISC uses a smaller number of simpler instructions that can be executed more quickly, as opposed to complex instructions used in other architectures like CISC (Complex Instruction Set Computer).

Key Features of RISC Architecture

- **Simplified Instructions**: RISC processors use a small, highly optimized set of instructions. Each instruction typically performs a very basic operation (e.g., adding two numbers, loading data from memory, or storing data in memory). These instructions are executed in a single clock cycle, making them faster and easier to implement in hardware.
- **Fixed Instruction Length**: RISC instructions typically have a fixed length, often **32** bits. This uniformity simplifies instruction decoding and pipelining, leading to better performance.
- **Load/Store Architecture**: Memory is accessed only through specific instructions: `load` (to load data from memory into registers) and `store` (to store data from registers into memory). All operations are performed only between data in registers, reducing complexity and speeding up execution.
- **Large Number of Registers**: RISC architectures provide a larger set of general-purpose registers (e.g., 32) to reduce the frequency of memory access. Since accessing registers is much faster than accessing memory, this improves efficiency.
- **Pipelining**: RISC processors are designed to take advantage of instruction pipelining, where multiple instructions are overlapped in execution. This helps in maximizing CPU utilization and improves throughput.
- **Efficient Compiler Use**: RISC architectures are designed to work efficiently with compilers. Because the instruction set is simpler, compilers can optimize code more easily, reducing overhead and improving performance.

RISC vs CISC

RISC is often contrasted with **CISC** (Complex Instruction Set Computer), which has the following characteristics:

- CISC processors have a large number of complex instructions, some of which can perform multiple tasks in a single instruction.
- In CISC, one instruction might take several clock cycles to execute, as the instructions are more complex.
- CISC architectures historically aimed to reduce the number of instructions by having more powerful, complex instructions, but this made the hardware more complex.

Summary of Differences

Features	RISC	CISC
Instruction Set	Small, simple set of instructions	Large, complex set of instructions
Execution Time	Single clock cycle per instruction	Multiple clock cycles for some instructions
Memory Access	Separate `load/store` instructions	Instructions can directly access memory
Registers	Large number of general-purpose registers	Fewer general-purpose registers
Instruction Length	Fixed-length instructions	Variable-length instructions
Hardware Complexity	Simple, efficient hardware design	Complex hardware design
Pipelining	Designed for pipelining	Less pipelining efficiency

Examples of RISC Processors

- **MIPS**: A widely used RISC architecture, especially in educational environments and embedded systems.
- **ARM**: One of the most popular RISC architectures, commonly used in mobile devices, such as smartphones and tablets.
- **SPARC**: Developed by Sun Microsystems, used in workstations and servers.
- **PowerPC**: Used in older Apple computers, and some gaming consoles like the Xbox 360.

Comparison of RISC and CISC

Feature	RISC	CISC
Instruction Set Simplicity	Simple, small set of instructions that execute quickly and easily.	Complex instructions that can perform multiple tasks, reducing the number of instructions per program.
Execution Speed	Most instructions execute in a single clock cycle, leading to faster performance.	Some instructions take multiple clock cycles to execute, leading to slower performance.
Pipelining	Efficient pipelining due to uniform instruction length and simpler operations.	Harder to implement efficient pipelining because of complex, variable-length instructions.
Hardware Complexity	Simpler hardware design due to fewer, basic instructions.	More complex hardware design to accommodate a wide variety of instructions.
Memory Usage	Programs tend to use more memory because more instructions are required to perform tasks.	Programs tend to use less memory as fewer instructions are needed to perform tasks.
Energy Efficiency	More energy-efficient due to simpler instruction set and reduced hardware complexity. Ideal for mobile and embedded systems.	Higher power consumption due to complex hardware and instruction set. Less suitable for energy-efficient systems.
Software Development	Compilers need to optimize more, making software development more complex.	Easier software development since complex instructions map more closely to high-level languages.

Although RISC architectures (like ARM) are gaining ground, especially in mobile and embedded systems due to their energy efficiency and simpler design, CISC architectures remain dominant in many computing domains (desktops, servers, and general-purpose computing) because of backward compatibility, a rich software ecosystem, and continuous advancements in CISC design. The industry's long-standing reliance on x86, its versatility, and its performance in high-computational tasks have cemented CISC's popularity in modern computing.

17.2 The Datapath Diagram

Having a visual representation of a datapath diagram that outlines the key components and features of the MIPS architecture is extremely helpful. It's important to note that there are various ways to implement an architecture in hardware. Today, pipelined and superscalar designs are common in high-performance processors. The initial diagram Figure 1 provides a basic and straightforward depiction of a MIPS datapath. While it is not a fully accurate representation of the MIPS architecture, it serves as a valuable starting point for understanding its structure.

- The PC sends the address to the instruction memory to fetch the next instruction.
- The fetched instruction is sent to the Instruction Register (IR), which splits it into different parts (opcode, registers, etc.).
- The control unit generates signals based on the instruction type.
- Data from the register file is processed by the ALU to perform the required operation (arithmetic/logical or address calculation).
- Depending on the instruction, the result may be written back to the register file or memory.
- The PC is updated to either the next sequential instruction or a new address for branches and jumps.

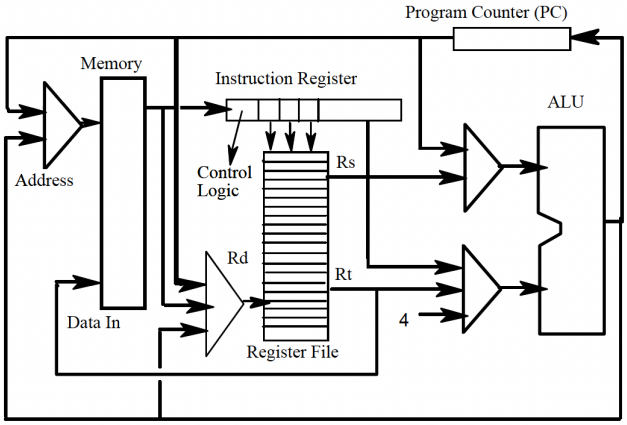


Figure 1: The Datapath Diagram

17.3 The MIPS Register File

Register	Name	Purpose
`$0`	Zero	Always holds the value 0. Cannot be written to.
`$at`	Assembler Temporary	Reserved for assembler use, not intended for general-purpose programming. This register is reserved for the assembler and is used internally for pseudo-instructions or during macro expansion.
`$v0-$v1`	Return Values	Used to store return values from functions.
`$a0-$a3`	Arguments	Used to pass arguments to functions.
`$t0-$t7`	Temporaries	Used for intermediate computations. These registers are not preserved across function calls.
`$s0-$s7`	Saved Registers	They are preserved across function calls, meaning if a function modifies one of these registers, it must save the old value and restore it before returning.
`$t8-$t9`	Temporaries	Additional temporary registers, not preserved across function calls.
`$k0-$k1`	Kernel Registers	Reserved for the operating system. User-level programs should not access these registers.
`$gp`	Global Pointer	Points to the middle of the global data segment for quick access to global variables.
`$sp`	Stack Pointer	Points to the top of the stack, used for function calls and managing local variables.
`$fp`	Frame Pointer	Points to the base of the current function's stack frame, useful for dynamic stack sizes.
`$ra`	Return Address	Stores the return address for function calls, set by the `jal` instruction. The address is where the program should continue execution after a function call completes.

17.4 Arithmetic Logic Unit

The ALU (Arithmetic Logic Unit) is responsible for performing arithmetic (Addition, subtraction, multiplication, division), logical (AND, OR, NOT, XOR), and comparison operations (Less than, greater than, equal to). It takes input from the register file, performs the required operation, and outputs the result back to registers or to memory, such as storing results back into registers, address calculations for memory access, determining the outcome of conditional branch instructions, etc. The ALU control signals determine which specific operation the ALU should perform, based on the instruction.

17.5 Program Counter

The Program Counter (PC) in MIPS **holds the address of the next instruction to be executed.** The PC is initialized by the operating system to the address of the first instruction of the program in memory. It increments by **4** (since each MIPS instruction is 32 bits) after each instruction in sequential execution. In branch or jump instructions, the PC is modified to point to a new address, enabling non-sequential execution. The PC plays a critical role in controlling program flow by determining the order in which instructions are fetched and executed. Key instructions

like beq (branch if equal), j (jump), and jal (jump and link) update the PC to specific target addresses based on conditions or function calls.

17.6 Cache Memory

Most contemporary processors, including those in the **MIPS architecture**, are designed with integrated cache memory directly on the **CPU chip**. This integration allows for significantly faster access to recently used data and instructions compared to accessing them from **main memory** (RAM). Cache memory acts as a temporary storage buffer that holds data the CPU is likely to need again soon, reducing the delay that would otherwise occur when retrieving data from slower, external memory.

In terms of **hardware implementation**, cache memory is typically organized as a **large array of storage locations**, similar to an array in software, where each location holds a block of information. For simplicity, this block of information can be fetched or written back to the cache one **"word"** at a time. In the context of MIPS, a **word** is defined as a 32-bit quantity. Cache memory operates by storing words at specific locations and retrieving them efficiently when required by the CPU.

Each word in cache corresponds to a specific **32-bit memory address**. In the **MIPS architecture**, the range of memory addresses spans from `0` to `4,294,967,295` (representing 4 GB of addressable memory), with each memory location being addressable at the **byte** level. In MIPS, a **byte** is defined as an **8-bit** unit of data, and since a word consists of four bytes, the address of a word in memory refers to the address of its **first byte**.

Another important aspect of MIPS architecture is the alignment of **instructions**. Every instruction in MIPS is exactly **32 bits** in length, which corresponds to a single word. Therefore, the **program counter (PC)**, which keeps track of the next instruction to be executed, increments by **4** after each instruction fetch, ensuring that the processor moves to the next word-aligned instruction in memory.

- **Cache memory** is integrated on the CPU chip for faster access to frequently used data and instructions.
- Cache stores data in **32-bit words**, and memory addresses in MIPS range from `0` to `4,294,967,295`.
- A **word** in MIPS is 32 bits, and a **byte** is 8 bits, with each word consisting of four bytes.
- The **program counter (PC)** increments by 4 after each instruction, as all instructions in MIPS are 32 bits long.

17.7 MIPS Instruction Set Architecture (ISA) and Assembler

The MIPS ISA defines the set of instructions that a MIPS processor can execute, while MIPS assembly provides a human-readable way of writing these instructions.
The MIPS ISA specifies the instructions available for the processor, how they are encoded in binary, and how they interact with hardware components such as registers and memory. MIPS is a RISC (Reduced Instruction Set Computer) architecture, designed to use a small, highly optimized set of simple instructions.

MIPS assembly language is a textual representation of the binary machine instructions defined in the MIPS ISA. Assembly mnemonics, such as `add`, `sub`, `lw`, and `sw`, correspond directly to specific machine instructions in the ISA.

An assembler is a software tool that translates assembly language, which is human-readable, into binary machine instructions that can be executed by the processor. The binary instructions correspond to the processor's instruction set architecture (ISA), such as MIPS or x86.

- The assembler reads each assembly instruction and converts it into the corresponding machine instruction according to the instruction set architecture (ISA).
- It resolves labels (used for branches and jumps) and symbolic addresses (used for data) into actual memory addresses.
- The assembler handles macros, directives, and pseudoinstructions that may not correspond directly to a single machine instruction.
- The assembler generates an *object file*, which contains the binary machine code that the processor can execute.

An assembler directly translates assembly language into machine code, while a compiler translates high-level programming languages (such as C or Java) into either assembly or machine code. While both processes ultimately produce executable machine code, assembly language is much closer to the hardware level and gives the programmer fine-grained control over the processor's instructions.

17.8 Instruction Register (IR) and Instruction Formats

The **Instruction Register (IR)** is a 32-bit register that holds a copy of the most recently fetched instruction. In the MIPS architecture, every instruction is 32 bits long, and these instructions are stored in the IR before being decoded and executed by the control unit. The role of the IR is crucial as it temporarily holds the current instruction so that subsequent stages of the pipeline can operate on it.

MIPS defines three different instruction formats, each suited for different types of operations: **R-format**, **I-format**, and **J-format**. These formats dictate how the 32 bits in the instruction are divided into fields, such as operation codes (opcodes), register identifiers, and immediate values. Each format is designed for specific types of instructions based on their operation and data requirements.

The **R-format (Register-format)** is used for arithmetic and logical instructions that operate on registers. These instructions include operations such as addition, subtraction, and bitwise AND/OR. The structure of the R-format instruction is as follows:

Opcode (6 bits)	rs (5 bits)	rt (5 bits)	rd (5 bits)	shamt (5 bits)	funct (6 bits)

- **Opcode (6 bits)**: Specifies the operation (usually 0 for R-format, with the actual operation determined by the funct field).
- **rs, rt, rd (5 bits each)**: These fields specify the source and destination registers.
- **shamt (5 bits)**: Specifies the shift amount for shift operations.
- **funct (6 bits)**: Further specifies the exact operation to be performed (e.g., add, subtract).

Let's take the example of the `add` instruction in MIPS, which adds two registers and stores the result in a third register.

```
add $t1, $t2, $t3
```

This instruction means: `$t1 = $t2 + $t3`, where `$t1`, `$t2`, and `$t3` are registers.
Binary Representation:

```
000000 01010 01011 01001 00000 100000
```

Breakdown:

- **Opcode (6 bits)**: `000000` (R-format instructions always have opcode `000000`).
- **rs (5 bits)**: `01010` (This is register `$t2`).
- **rt (5 bits)**: `01011` (This is register `$t3`).
- **rd (5 bits)**: `01001` (This is register `$t1`).
- **shamt (5 bits)**: `00000` (Shift amount is 0 for the `add` instruction).
- **funct (6 bits)**: `100000` (Specifies the `add` function).

The **I-format(Immediate Format)** is used for instructions that involve immediate values or memory access. These include load/store instructions and arithmetic operations with an immediate operand. The structure of the I-format instruction is as follows:

Opcode (6 bits)	rs (5 bits)	rt (5 bits)	Immediate (16 bits)

- **Opcode (6 bits)**: Specifies the operation, such as load, store, or immediate arithmetic.
- **rs (5 bits)**: Specifies the source register.
- **rt (5 bits)**: Specifies the destination register.
- **Immediate (16 bits)**: Contains a constant value or address offset used by the instruction. This offset can be positive or negative and is **sign-extended** to 32 bits to ensure it can be added correctly to the base register's address.

Let's take the example of the `lw` (load word) instruction in MIPS, which loads a value from memory into a register.

```
lw $t1, 32($t2)
```

This instruction means: Load the word from memory address `$t2 + 32` into register `$t1`.
Binary Representation:

`100011 01010 01001 0000000000100000`

Breakdown:

- **Opcode (6 bits)**: `100011` (Opcode for `lw`).
- **rs (5 bits)**: `01010` (This is register `$t2`, the base register).
- **rt (5 bits)**: `01001` (This is register `$t1`, the destination register).
- **Immediate (16 bits)**: `0000000000100000` (The immediate value is 32, representing the offset).

What if the Offset Exceeds 16 Bits?
If the value you want to use as an offset exceeds the range of 16 bits (i.e., greater than ±32,767), you need to load the large value into a register first. Use the lui (load upper immediate) instruction to load the higher 16 bits of a large constant into a register. Then, use ori (or immediate) or another instruction to combine this with a lower 16-bit immediate value. Example for Large Offsets:

```
lui $t0, 0x1234      # Load upper 16 bits with 0x1234 (0000 1234 0000 0000)
ori $t0, $t0, 0x5678 # Combine with the lower 16 bits 0x5678 (result: 0x12345678)
lw $t1, 0($t0)       # Load word from memory address 0x12345678
```

In this example: lui loads the upper half of the address (16 bits) into the register. ori (or immediate) combines it with a 16-bit lower part, effectively creating a full 32-bit address. This is how MIPS handles large offsets that exceed 16 bits.
The **J-format (Jump Format)** is used for jump instructions that alter the flow of control by jumping to a specific address in memory. These instructions include jumps and function calls. The structure of the J-format instruction is as follows:

Opcode (6 bits)	Address (26 bits)

- **Opcode (6 bits)**: Specifies the jump operation.
- **Address (26 bits)**: Contains the target address to jump to.

Let's take the example of the `j` (jump) instruction in MIPS, which jumps to a specific address in memory.

```
j 0x00400000
```

This instruction means: Jump to the memory address `0x00400000`.
Binary Representation:

`000010 00000000010000000000000000`

Breakdown:

- **Opcode (6 bits)**: `000010` (Opcode for the `j` instruction).
- **Address (26 bits)**: `00000000010000000000000000` (The target address is `0x00400000`, represented in 26-bit form).

Why 26 bits are sufficient for addressing?
In a J-format instruction (e.g., j or jal), the 26-bit address specifies part of the target address. The remaining 6 bits needed to form a full 32-bit address come from the most significant bits of the current program counter (PC). Here's how the process works:

1. 26-bit Immediate Address: The J-format instruction provides a 26-bit address. This is the lower portion of the address, but it is not enough to directly address a 32-bit memory space.
2. Shifting the Address: The 26-bit immediate address is first left-shifted by 2 bits to align it on a 4-byte boundary. This is because instructions in MIPS are always word-aligned (i.e., addresses are multiples of 4).
3. Combining with the Upper PC Bits: The upper 4 bits of the program counter (PC) are taken from the current value of the PC. These bits are concatenated with the shifted 26-bit immediate value to form the full 32-bit address.

Why Shift by 2 Bits in J-Format?
In MIPS, instructions are 32 bits long (4 bytes), and memory is **word-addressable**. This means that instruction addresses are always multiples of 4. As a result, the two least significant bits of an instruction address are always `00` in binary.

- **Compact Representation**: Since instruction addresses are word-aligned, the two least significant bits are always `00`. Instead of wasting these 2 bits in the instruction encoding, MIPS omits them and shifts the 26-bit address left by 2 bits to recover the full 28-bit word-aligned address.
- **Upper 4 Bits from PC**: The upper 4 bits of the current PC are concatenated with the shifted address to form a 32-bit address. The J-format can only jump within the 256 MB block (segment) in which the current PC resides.

To jump to outside current segment, you need to use `lui` and `ori` we mentioned before.

18 Assembly Programming

18.1 Hello World!

Let's first see the hello world program in C:

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

Rewrite it in MIPS assembly:

```
.data
hello: .asciiz "Hello World!"

.text
main:
  li $v0, 4 # load immediate, v0 = 4 (4 is print string system call)
  la $a0, hello # load address of string to print into a0
  syscall

  li $v0, 10 # exit syscall
  syscall
```

Labels in MIPS follow the same naming conventions as C variables and functions—they must start with a letter or underscore, followed by letters, underscores, or digits.

In MIPS assembly, `li` and `la` are two commonly used instructions to load values into registers. The `li` instruction stands for "Load Immediate," and it is used to load an immediate (constant) value directly into a register. This instruction is typically used when you want to load small integers or constant values.

`li $register, immediate_value`

- `$register` is the destination register where the immediate value will be stored.
- `immediate_value` is the constant value being loaded.

The `la` instruction stands for "Load Address," and it is used to load the memory address of a label (like a variable or a string) into a register. This is useful when you need to work with addresses in memory, such as accessing strings or arrays.

`la $register, label`

- `$register` is the destination register where the address of the label will be stored.
- `label` is the memory location (or variable) whose address is being loaded.

In this example:

- `li $v0, 4` loads the syscall number for printing a string into register `$v0`.
- `la $a0, hello` loads the address of the string `hello` into register `$a0`, which is used by the syscall.
- Finally, the `syscall` executes the system call to print the string.

Let's open the MARS simulator to assemble and execute our hello world program. Write the code in the *Edit* window then click *Assemble* under *Run* entry. Click *run* and switch to *Execute page* and check the Hexadecimal Addresses box, Hexadecimal Values box and ASCII box in the Data Segment sub-window. Note that the data segment of MARS Simulator starts from `0x10010000`, you should see the value below:

Address	Value (+0)	Value (+4)	Value (+8)	Value (+C)
0x10010000	l l e H	o W o	! d l r	\0 \0 \0 \0
0x10010020	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10010040	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0

Table 9: Data Segment for Hello World Program

In MIPS assembly programming, data, including strings, is typically split into **4-byte chunks** when stored in memory. This is due to the architecture's word size and memory alignment requirements. Additionally, the endianness of the system (whether it is little-endian or big-endian) determines how these chunks are stored in memory.

MIPS is a 32-bit architecture, the size of each register in MIPS is 32 bits, which directly corresponds to the word size of 4 bytes. Memory is accessed in 4-byte units, known as words, because this is the most efficient way for the processor to handle data. Thus, when a string or any other data is stored in memory, it is split into 4-byte chunks for processing.

To optimize performance, memory is often aligned on word boundaries. This means that memory addresses for data are multiples of 4 bytes. When storing a string in memory, it is divided into 4-byte segments to match the processor's word size and maintain alignment. Accessing aligned memory is faster because the processor can read or write 4 bytes at a time.

Consider the string "Hello World!\n". This string is stored in memory in chunks of 4 bytes (32 bits). Each 4-byte chunk corresponds to part of the string:

Chunk	**Characters**	**Hex Representation**
1	Hell	0x48656C6C
2	o Wo	0x6F20576F
3	rld!	0x726C6421
4	\n (newline)	0x0A

The order in which these 4-byte chunks are stored in memory depends on the system's endianness:

In **little-endian** systems, the least significant byte (LSB) is stored first (MARS simulator for example). This means that each 4-byte chunk is reversed when stored in memory. For example, the chunk `0x48656C6C` (representing "Hell") would be stored as `6C 6C 65 48` in memory, resulting in reversed byte order within each 4-byte word.

In **big-endian** systems, the most significant byte (MSB) is stored first, which matches the natural order of the bytes. In this case, the chunk `0x48656C6C` would be stored in memory as `48 65 6C 6C`, with no byte reversal.

Why little-endian is more popular?

In a little-endian system, the address of a value remains consistent across different data widths (like 8-bit, 16-bit, or 32-bit), which simplifies casting and memory access significantly. Here's how this works and why it benefits data handling:

In little-endian memory layout, the least significant byte of a value is stored at the lowest memory address, with more significant bytes following at higher addresses. This layout means that the same starting address can be used to fetch data of different widths (e.g., 8-bit, 16-bit, or 32-bit) without modifying the address.

Consider a memory layout where we store a 16-bit value `0x0016` in little-endian order:

Address	Data (Hex)
0x00f0	16
0x00f1	00

If we interpret the value at address `0x00f0` as an 8-bit integer, we get `0x16`. If we interpret it as a 16-bit integer (e.g., C `short`), we get `0x0016`. The address `0x00f0` remains the same regardless of the data width—only the instruction used to fetch it changes.

This uniformity means that "casting is a no-op" in little-endian systems: casting to a smaller or larger type doesn't require any changes to the address, just a different instruction to read the desired number of bytes.

In a big-endian system, the most significant byte is stored at the lowest address. Using the same example:

Address	Data (Hex)
0x00f0	00
0x00f1	16

If you want to read an 8-bit value that represents `0x16`, you need to adjust the pointer to `0x00f1`. For casting or accessing different widths, you'd need to adjust the pointer or use additional logic to locate the least significant byte, making casts more complex.

In little-endian systems, consistent addressing across data widths allows for simple and efficient data manipulation. Truncation, casting, and expansion operations all benefit because the least significant byte is always at the starting address, making pointer adjustments unnecessary and ensuring that casting is effectively a no-op. This simplification is one reason why little-endian order is widely favored in modern architectures like Intel x86 and ARM.

Let's further check the content in the Text Segment window. From the table, we can see each of the assembly commands was translated to a binary code and stored in the memory. The "Basic" column helps visualize the exact MIPS instruction being executed at a particular memory address, making it easier to trace and debug the execution of your MIPS program:

Address	Code	Basic
0x00400000	0x24020004	`addiu $2,$0,0x00000004`
0x00400004	0x3c011001	`lui $1,0x00001001`
0x00400008	0x34240000	`ori $4,$1,0x00000000`
0x0040000c	0x0000000c	`syscall`
0x00400010	0x2402000a	`addiu $2,$0,0x0000000a`
0x00400014	0x0000000c	`syscall`

Table 10: Text Segment for Hello World Program

For most programming tasks in MIPS using the MARS simulator, **you don't have to manually distinguish between real instructions, pseudo-instructions, or macros** because the assembler (like MARS) handles that for you. Here's how it works:

Real instructions (MIPS Instructions) are the basic, standard instructions defined by the MIPS instruction set, such as `add`, `sub`, `lw` (load word), and `sw` (store word). These instructions are directly translated into machine code by the assembler without any modification.

Pseudo-instructions are shorthand instructions that **aren't part of the official MIPS instruction set** but make programming easier. The assembler translates them into real MIPS instructions.

In some assemblers, macros allow for more complex code generation from a single line of input. However, in MARS, you generally don't write or deal with macros directly. You mostly focus on real instructions and pseudo-instructions. If macros are used, MARS would expand them into their equivalent MIPS instructions as part of the assembly process. We will talk about macros later.

```
1 li $t0, 10        # Pseudo-instruction
2 addi $t1, $t0, 5  # Real instruction
```

You don't need to worry about explicitly making the distinction, as MARS will ensure the correct translation and execution.
On the right side of the window, we can check values of all 32 registers, let's see the value stored in `v0`, `a0` and `pc`. `v0` stored the hex number `0xa`, which is exactly the integer number 10; `a0` stored an address `0x10010000`, where stores our hello world string; pc store the next command address `0x00400018`:

Name	Idx	Value
$v0	2	0x0000000a
$a0	4	0x10010000
pc	-	0x00400018

Table 11: MIPS Register Values for Hello Word Program

Note: `Instruction: lui $1, 0x00001001` The "Load Upper Immediate" instruction loads the upper half of register $1 with the value 0x1001. This is part of the process to load the address of the string to be printed into $a0. `Instruction: ori $4, $1, 0x00000000` The "OR Immediate" instruction combines the lower half of register $1 (which was set by the previous instruction) with 0x0000 and stores the result in $4. Register $a0 ($4) now contains the full address of the string "Hello World!".

In MARS, you will see some special registers such as pc, hi, lo and register of coproc 0. In MIPS assembly, the `HI` and `LO` registers and *coprocessor 0* play special roles in operations and system controls:

The HI and LO registers are used specifically to store results from multiplication and division operations. HI stores the remainder from division (DIV), and LO stores the quotient. For multiplication, HI stores the upper 32 bits of the result, while LO stores the lower 32 bits.

Coprocessor 0 in MIPS, handles system-level functions, particularly for exceptions and memory management. COPROC0 contains control and status registers that manage exception handling, virtual memory, and interrupt controls. Key COP0 registers include:

- **Status Register**: Manages interrupt enabling and exception handling.
- **Cause Register**: Holds information about the cause of an exception.
- **EPC (Exception Program Counter)**: Stores the address of the instruction causing an exception.

18.2 .data Section

In MIPS assembly, you can declare global variables in the .data section. This section is typically where you define any literal strings your program will print, as most programs have at least one or two such strings. When declaring a variable in the .data section, the format follows:

```
variable_name: .directive value(s)
```

Whitespace between these elements is arbitrary. The available directives are summarized in Table 12:

Directive	Size (bytes)	C Equivalent
.byte	1	char
.half	2	short
.word	4	int, all pointer types
.float	4	float
.double	8	double
.ascii	NA	stores a string without automatically appending a ('\0') at the end.
.asciiz	NA	stores the string with a null terminator '\0'
.space	NA	typeless, uninitialized space (can be used for any type or array)

Table 12: MIPS Data Directives

1. .asciiz is needed when working with system calls or functions (like printf or string manipulation routines) that expect strings to be null-terminated. This is common in many programming environments that work with C-style strings.
2. .ascii, on the other hand, can be used when you don't need a null terminator, such as when working with raw character data, fixed-length strings, or other data formats.
3. .space The .space directive in MIPS assembly is used to allocate a specific number of bytes in memory for storing data. It reserves space without initializing it with any specific value.

Modify the directive of string "Hello World!", check the Data Segment and see the difference:

```
.data
hello: .ascii "Hello World!"
name: .asciiz "Qixin Deng"

.text
main:
  li $v0, 4 # load immediate, v0 = 4 (4 is print string system call)
  la $a0, hello # load address of string to print into a0
  syscall
  la $a0, name # load address of string to print into a0
  syscall

  li $v0, 10 # exit syscall
  syscall
```

As you can see, the data declarations are fairly straightforward. However, there are some important details regarding their usage, so let's explore an example. Consider the following simple C program:

Address	Value (+0)	Value (+4)	Value (+8)	Value (+C)	Value (+10)	Value (+14)
0x10010000	l l e H	o W o	! d l r	i x i Q	e D n	\0 \0 g n
0x10010020	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0

Table 13: Use .ascii Directive

Address	Value (+0)	Value (+4)	Value (+8)	Value (+C)	Value (+10)	Value (+14)
0x10010000	l l e H	o W o	! d l r	x i Q \0	D n i	\0 g n e
0x10010020	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0

Table 14: Use .asciiz Directive

```c
#include <stdio.h>

int main() {
    char name[30];
    int age;
    printf("What's your name and age?\n");
    scanf("%s %d", name, &age);
    printf("Hello %s, nice to meet you!\n", name);
    return 0;
}
```

When converting a high-level program like this into MIPS assembly (or any assembly language), the goal is to ensure functional equivalence rather than replicating the code structure exactly. In this case, it means recognizing that the literal strings and the local variables `name` and `age` should be treated as global variables in MIPS. Here is the MIPS equivalent data for this program:

```
.data
age:         .word 0                # Space for the age (integer) 4 bytes and init to 0
ask_name:    .asciiz "What's your name and age?\n"
hello_space:.asciiz "Hello "
nice_meet:   .asciiz "nice to meet you!\n"
name:        .space 30              # Space for the name (string)

.text


main:
    # Print "What's your name and age?\n"
    li      $v0, 4                  # syscall for print_string
    la      $a0, ask_name           # load the address of the ask_name string
    syscall                         # make the system call

    # Read name (string) and age (integer)
    li      $v0, 8                  # syscall for reading a string
    la      $a0, name               # load address of name into £a0
    li      $a1, 30                 # set maximum length of input (30 bytes)
    syscall                         # make the system call to read name

    li      $v0, 5                  # syscall for reading an integer
    syscall                         # make the system call
    sw      $v0, age                # store the read integer (age) into the age variable

    # Print "Hello " (Greeting part 1)
    li      $v0, 4                  # syscall for print_string
    la      $a0, hello_space        # load the address of the "Hello " string
    syscall                         # make the system call

    # Print name (Greeting part 2)
    la      $a0, name               # load the address of the name string
    syscall                         # make the system call

    # Print ", nice to meet you!\n" (Greeting part 3)
    la      $a0, nice_meet          # load the address of the "nice to meet you" string
```

```
    syscall                         # make the system call
    
    # Exit the program
    li      $v0, 10                 # syscall for exit
    syscall                         # make the system call
```

In this example, we extract the string literals, the character array `name`, and the integer `age`, and declare them as global variables in the `.data` section. Notice the second `printf` call in the C code. Since it prints the variable `name` using a conversion specifier (`%s`), we split the literal string into two parts: one before the variable and one after. Since there is no built-in `printf` function in MIPS, you need to manually handle printing variables by using the appropriate system calls.

Note: Compilers have stricter rules for following specific code structures, but for manual assembly, functional equivalence is the key priority.

18.3 Array

While strings are special cases that can be handled using `.ascii` or `.asciiz` for literals, how do we declare and handle other types or user-input strings?

The first approach, as demonstrated in the earlier example, is to use the `.space` directive to declare an array with the required number of bytes. It's important to note that the size is specified in bytes, not elements, so this method is suitable for character arrays. For arrays of other types, such as integers, floats, or doubles, you need to multiply by the *sizeof(type)*.

You might wonder, "But `.space` only lets you declare uninitialized arrays, how can I initialize them?" Interestingly, `.space` seems to initialize everything to zero, much like global or static data in C. However, this behavior isn't explicitly documented.

There are two ways to declare initialized arrays, depending on whether you want to initialize each element to the same value or to different values:

- **Different Values:** You can extend the syntax for declaring a single variable of a particular type by listing the values in a comma-separated format.
- **Same Value for All Elements:** If you want all elements to be initialized to the same value, you can use a more convenient option. After specifying the type, write the value, followed by a colon, and then the number of elements. For instance, `a:  .word 123 :  10` would declare an integer array of 10 elements, each initialized to 123.

Consider the following C code:

```c
int a[20];
double b[20];
int c[10] = { 9,8,7,6,5,4,3,2,1,0 };
int d[5] = { 42, 42, 42, 42, 42 };
char e[3] = { 'a', 'b', 'c' };
```

This would translate to the following MIPS assembly:

```
.data
a: .space 80                # 20 integers, 4 bytes each
b: .space 160               # 20 doubles, 8 bytes each
c: .word 9,8,7,6,5,4,3,2,1,0
d: .word 42 : 5             # 5 integers, all initialized to 42
e: .byte 'a', 'b', 'c'      # character array
```

18.4 Load and Store Instructions

In MIPS assembly, access to memory (RAM) is **only allowed through specific load and store instructions**. All other instructions operate on register operands.

18.4.1 Direct Memory Access

The `lw` (load word) instruction copies a word (4 bytes) from a memory address into a register.

```
lw register_destination, RAM_source
# Copy word (4 bytes) from RAM source to destination register.
```

The `lb` (load byte) instruction copies a byte from a memory address into the low-order byte of a destination register. The value is sign-extended to fill the higher-order bytes of the register.

```
lb register_destination, RAM_source
# Copy byte from RAM source to destination register, sign-extended.
```

The `sw` (store word) instruction stores the contents of a register into a specified memory location.

```
sw register_source, RAM_destination
# Store word from register source to RAM destination.
```

The `sb` (store byte) instruction stores the low-order byte of a register into a memory location.

```
sb register_source, RAM_destination
# Store byte from register source to RAM destination.
```

The `li` (load immediate) instruction loads a constant value directly into a register.

```
li register_destination, value
# Load immediate value into destination register.
```

Here is an example that demonstrates the use of `lw`, `sw`, and `li` instructions in a MIPS assembly program:

```c
#include <stdio.h>

int main() {
    int var1 = 23;    // Declare a variable and assign it an initial value of 23.
    int t0, t1;

    t0 = var1;        // Load the value of var1 into t0.
    t1 = 5;           // Load the immediate value 5 into t1.
    var1 = t1;        // Store the value of t1 back into var1.

    printf("var1 = %d\n", var1);  // Print the final value of var1 (should be 5).
    return 0;
}
```

```
.data
var1:   .word 23          # Declare var1 with an initial value of 23.

.text
main:

    la $t0, var1          # Load the address of var1 in £t0£
    lw $t1, 0($t0)        # Load value from the address stored in t0
                          # in MARS, lw £t0, var1 will also give you the correct execution
    li $t1, 5             # Load the immediate value 5 into register £t1.
    sw $t1, var1          # Store the contents of £t1 into var1.

    # Exit program
    li $v0, 10            # System call for exit
    syscall
```

In the MARS simulator, you can indeed use a label directly with instructions like lw without needing to explicitly load the address first into a general-purpose register (e.g., using la). This behavior is specific to MARS and simplifies loading values from memory. In real MIPS hardware or in other simulators, you might need to load the address of the memory into a general-purpose register before accessing the value.

18.4.2 Indirect Memory Access

The `la` (load address) instruction is used to load the address of a variable (usually a label defined in the data section) into a register.

```
la $t0, var1
# Copy the address of var1 into register £t0
```

This instruction loads the base address of the memory location (i.e., ‘var1‘) into the specified register (‘$t0‘).

In indirect addressing, the contents of a register are treated as a memory address. You can load or store data to/from memory using the address stored in a register.

Loads data from the memory address contained in a register into another register.

```
lw $t2, ($t0)
# Load the word at the memory address in £t0 into £t2
```

Stores the value from a register into the memory address contained in another register.

```
sw $t2, ($t0)
# Store the word in £t2 at the memory address in £t0
```

Based addressing adds an offset to the value contained in a register to calculate the effective memory address. This mode is useful for accessing arrays and stack elements.

Loads data from the memory address calculated as the value in the register plus the offset.

```
lw $t2, 4($t0)
# Load the word at memory address (£t0 + 4) into £t2
```

Stores the value in a register into the memory address calculated as the value in the register plus the offset.

```
sw $t2, -12($t0)
# Store the word in £t2 at the memory address (£t0 - 12)
```

Negative offsets are allowed, making this mode particularly useful for accessing stack elements and arrays. Here is an example that demonstrates load address, indirect addressing, and based addressing with an array in a MIPS assembly program:

```c
#include <stdio.h>

int main() {
    int array1[3];     // Declare an array of 3 integers
    int *t0;

    t0 = array1;       // t0 points to the base address of array1

    // Store values in the array using pointer arithmetic
    *t0 = 5;           // First element = 5
    *(t0 + 1) = 13;    // Second element = 13
    *(t0 + 2) = -7;    // Third element = -7

    return 0;
}
```

```
.data
array1:  .space 12         # Declare 12 bytes of storage for an array of 3 integers

.text
main:
    # Load address of array1 into £t0 (direct addressing)
    la $t0, array1

    # Store values in the array using indirect and based addressing

    # Store 5 in the first element of the array (indirect addressing)
    li $t1, 5
    sw $t1, ($t0)

    # Store 13 in the second element of the array (based addressing)
    li $t1, 13
    sw $t1, 4($t0)

    # Store -7 in the third element of the array (based addressing)
    li $t1, -7
    sw $t1, 8($t0)

    # Exit the program
    li $v0, 10
    syscall
```

18.5 System Calls

In MIPS assembly, system calls provide a way to interact with the operating system, allowing for basic input/output operations, memory allocation, and file handling. **These system calls are invoked by placing a specific code in register** `$v0` and, depending on the system call, placing the necessary arguments in other registers.

Name ($v0)	Arguments	Result
1. print integer	$a0 = integer to print	
2. print float	$f12 = float to print	
3. print double	$f12 = double to print	
4. print string	$a0 = address of string	
5. read integer		$v0 = integer read
6. read float		$f0 = float read
7. read double		$f0 = double read
8. read string	$a0 = address of input buffer, $a1 = buffer size	works like C's fgets
9. sbrk	$a0 = size in bytes to allocate	$v0 = address of allocated memory
10. exit		program terminates
11. print character	$a0 = character to print (ASCII value)	
12. read character		$v0 = character read
13. open file	$a0 = address of filename, $a1 = flags, $a2 = mode	$v0 = file descriptor (negative if error)
14. read from file	$a0 = file descriptor, $a1 = address of input buffer, $a2 = max characters to read	$v0 = number of characters read, 0 for EOF, negative for error
15. write to file	$a0 = file descriptor, $a1 = address of output buffer, $a2 = number of characters to write	$v0 = number of characters written, negative for error
16. close file	$a0 = file descriptor	

System-Call Steps:

1. The system call number is placed in the register $v0.

2. If the system call requires arguments, they are placed in registers $a0-$a3.
3. The `syscall` instruction is executed, which triggers the system call.

Here is a C and MIPS assembly program that demonstrates the use of these system calls.

```c
#include <stdio.h>

int main() {
    // Print a string
    printf("Hello, world!\n");

    // Print an integer
    printf("%d\n", 42);

    // Read an integer from user input
    int num;
    scanf("%d", &num);

    // Print the integer read from input
    printf("%d\n", num);

    // Print a character
    printf("%c\n", 'A');

    // Exit the program
    return 0;
}
```

```
.data
message:    .asciiz "Hello, world!\n"
buffer:     .space 100  # Buffer for reading strings

.text
main:
    # Print a string
    li $v0, 4              # Load system call code for print string
    la $a0, message         # Load the address of the string to print
    syscall                 # Make system call

    # Print an integer
    li $v0, 1              # Load system call code for print integer
    li $a0, 42             # Load the integer to print
    syscall                 # Make system call

    # Read an integer
    li $v0, 5              # Load system call code for read integer
    syscall                 # Make system call
    move $t0, $v0          # Store the read integer in £t0

    # Print the read integer
    li $v0, 1              # Load system call code for print integer
    move $a0, $t0          # Load the read integer into £a0
    syscall                 # Make system call

    # Print a character
    li $v0, 11             # Load system call code for print character
    li $a0, 65             # ASCII for 'A'
    syscall                 # Make system call

    # Exit the program
    li $v0, 10             # Load system call code for exit
    syscall                 # Make system call
```

18.6 Integer Arithmetic Instructions

MIPS arithmetic instructions typically use three operands, and all operands must be registers. These instructions do not directly access RAM or use indirect addressing. The operand size is 4 bytes (a word), and there are different types of addition and subtraction operations for signed and unsigned integers.

18.6.1 Addition and Subtraction

Adds two registers as signed integers and stores the result in a third register.

```
add $t0, $t1, $t2
# £t0 = £t1 + £t2; add as signed integers
```

Subtracts one register from another as signed integers.

```
sub $t2, $t3, $t4
# £t2 = £t3 - £t4; subtract as signed integers
```

Adds an immediate value to a register.

```
addi $t2, $t3, 5
# £t2 = £t3 + 5; add immediate value
```

Unsigned arithmetic treats all values as non-negative integers.

Adds two registers as unsigned integers.

```
addu $t1, $t6, $t7
# £t1 = £t6 + £t7; add as unsigned integers
```

Subtracts one register from another as unsigned integers.

```
subu $t1, $t6, $t7
# £t1 = £t6 - £t7; subtract as unsigned integers
```

18.6.2 Multiplication and Division

MIPS supports multiplication and division using special registers for the results. The product of two 32-bit values results in a 64-bit value stored in two special registers: ‘Lo‘ (lower 32 bits) and ‘Hi‘ (upper 32 bits).
Multiplies two 32-bit values and stores the result in ‘Hi‘ and ‘Lo‘.

```
mult $t3, $t4
# (Hi, Lo) = £t3 * £t4; store the 64-bit result in special registers Hi and Lo
```

Divides one register by another, storing the quotient in ‘Lo‘ and the remainder in ‘Hi‘.

```
div $t5, $t6
# Lo = £t5 / £t6; Hi = £t5 % £t6 (remainder)
```

To access the results of multiplication or division, the `mfhi` and `mflo` instructions are used to move the contents of the special registers to general-purpose registers.
Moves the contents of ‘Hi‘ to a register.

```
mfhi $t0
# £t0 = Hi; move the value in Hi to £t0
```

Moves the contents of ‘Lo‘ to a register.

```
mflo $t1
# £t1 = Lo; move the value in Lo to £t1
```

The below is an example of a multiplication result that exceeds 32-bit:

```
.text
main:
    # Load the numbers into registers
    li $t1, 12345            # Load num1 into £t1
    li $t2, 67891              # Load num2 into £t2

    # Perform multiplication
    mult $t1, $t2             # Multiply £t1 and £t2
    mflo $t0                  # Move lower 32 bits of result (LO) to £t0
    mfhi $t3                  # Move upper 32 bits of result (HI) to £t3

    # Print the upper 32 bits (HI) if non-zero
    bnez $t3, print_hi        # Branch if HI is non-zero

print_lo:
    # Print the lower 32 bits directly
    move $a0, $t0             # Move LO result to £a0 for printing
    li $v0, 1                 # Syscall for printing integer
    syscall                   # Print LO (lower 32 bits)

    # Exit program
    li $v0, 10                # Syscall for exit
    syscall

print_hi:
    # Print the upper 32 bits (HI) in decimal form
    move $a0, $t3             # Move HI result to £a0
    li $v0, 1                 # Syscall for printing integer
    syscall                   # Print HI (upper 32 bits)

    # Now print the lower 32 bits
    j print_lo                # Jump to print_lo to print LO as well
```

Below is an example of integer division:

```
.data
newline: .asciiz "\n"     # Newline character

.text
main:
    # Load numbers into registers
    li $t0, 50           # Load dividend into £t1
    li $t1, 7           # Load divisor into £t2

    # Perform division
    div $t0, $t1           # Divide £t1 by £t2
    mflo $t2               # Move quotient (LO) to £t0
    mfhi $t3               # Move remainder (HI) to £t3

    # Print the quotient
    move $a0, $t2          # Move quotient to £a0
    li $v0, 1              # Syscall for printing integer
    syscall                # Print quotient

```

```
    # Print a newline after the quotient
    li $v0, 4                 # Syscall for printing string
    la $a0, newline           # Load address of newline character
    syscall                   # Print newline

    # Print the remainder
    move $a0, $t3          # Move remainder to £a0
    li $v0, 1              # Syscall for printing integer
    syscall                # Print remainder

    # Exit the program
    li $v0, 10             # Syscall for exit
    syscall
```

18.6.3 Data Movement

The ‘move‘ instruction is used to copy the contents of one register into another.
Copies the value from one register to another.

```
move $t2, $t3
# £t2 = £t3; move the value in £t3 to £t2
```

18.6.4 Example

Here is an example that demonstrates the use of various arithmetic instructions in a C and MIPS assembly program:

```c
#include <stdio.h>

int main() {
    int t1 = 10, t2 = 20, t0;

    // Basic arithmetic
    t0 = t1 + t2;          // t0 = 30
    t2 = t0 - t1;          // t2 = 20

    // Addition with immediate
    t2 = t2 + 5;           // t2 = 25

    // Unsigned arithmetic (assuming values are non-negative)
    unsigned int u1 = t0 + t2;  // unsigned addition
    unsigned int u2 = t0 - t2;  // unsigned subtraction

    // Multiplication and division
    int t3 = 10, t4 = 5;
    int product = t3 * t4;      // product = 50
    int quotient = t3 / t4;     // quotient = 2
    int remainder = t3 % t4;    // remainder = 0

    // Move operation
    t2 = remainder;             // t2 = remainder (move)

    return 0;
}
```

```
.text
main:
    # Basic arithmetic
    li $t1, 10             # £t1 = 10
    li $t2, 20             # £t2 = 20
    add $t0, $t1, $t2      # £t0 = £t1 + £t2; £t0 = 30
    sub $t2, $t0, $t1      # £t2 = £t0 - £t1; £t2 = 20

    # Addition with an immediate value
    addi $t2, $t2, 5       # £t2 = £t2 + 5; £t2 = 25

    # Unsigned addition and subtraction
```

```
    addu $t1, $t0, $t2    # $t1 = $t0 + $t2; unsigned addition
    subu $t1, $t0, $t2    # $t1 = $t0 - $t2; unsigned subtraction

    # Multiplication and division
    li $t3, 10            # $t3 = 10
    li $t4, 5             # $t4 = 5
    mult $t3, $t4         # (Hi, Lo) = $t3 * $t4
    mflo $t0              # $t0 = Lo; get the product

    div $t3, $t4          # Lo = $t3 / $t4; Hi = $t3 % $t4
    mflo $t0              # $t0 = quotient (Lo)
    mfhi $t1              # $t1 = remainder (Hi)

    # Move instruction
    move $t2, $t1         # $t2 = $t1; move remainder to $t2

    # Exit program
    li $v0, 10
    syscall
```

Another example:

```c
#include <stdio.h>

int main() {
    int array[] = {10, 20, 30};  // Define an array of three integers
    int sum;

    // Sum the elements in the array
    sum = array[0] + array[1] + array[2];

    // Print the result
    printf("%d\n", sum);  // Expected output: 60

    return 0;
}
```

```
.data
    array: .word 10, 20, 30   # Array of three integers

.text
main:
    lw $t0, array             # Load first element (10) into $t0
    lw $t1, array+4           # Load second element (20) into $t1
    lw $t2, array+8           # Load third element (30) into $t2
    add $t3, $t0, $t1         # Add first two elements
    add $t3, $t3, $t2         # Add the third element to the sum

    move $a0, $t3             # Move result to $a0 for printing
    li $v0, 1                 # Syscall for print integer
    syscall                   # Print the sum (10 + 20 + 30 = 60)
```

18.7 Bitwise Instructions

The following C program demonstrates basic bitwise operations: AND, OR, XOR, left shift, and right shift.

```c
int a = 12;        // 1100 in binary
int b = 10;        // 1010 in binary
int result_and = a & b;  // AND operation
int result_or = a | b;   // OR operation
int result_xor = a ^ b;  // XOR operation
int result_left = a << 2;  // Left shift (multiply by 4)
int result_right = a >> 1; // Right shift (divide by 2)
```

The following MIPS assembly code performs the same bitwise operations in the MARS simulator:

```
.data
result_and: .word 0  # To store the result of AND operation
result_or: .word 0   # To store the result of OR operation
result_xor: .word 0  # To store the result of XOR operation
result_left: .word 0 # To store the result of left shift operation
result_right: .word 0 # To store the result of right shift operation

.text
main:
    # Load the value of a into $t0
    li $t0, 12
    # Load the value of b into $t1
    li $t1, 10

    # AND operation: result_and = a & b
    and $t2, $t0, $t1
    sw $t2, result_and

    # OR operation: result_or = a | b
    or $t3, $t0, $t1
    sw $t3, result_or

    # XOR operation: result_xor = a ^ b
    xor $t4, $t0, $t1
    sw $t4, result_xor

    # Left shift: result_left = a << 2
    sll $t5, $t0, 2   # Shift left by 2 bits (multiply by 4)
    sw $t5, result_left

    # Right shift: result_right = a >> 1
    srl $t6, $t0, 1   # Shift right by 1 bit (divide by 2)
    sw $t6, result_right

    # Exit the program
    li $v0, 10
    syscall
```

1. The values of `a` and `b` are loaded into registers `$t0` and `$t1`.
2. Bitwise AND (`and`), OR (`or`), and XOR (`xor`) operations are performed and stored in memory.
3. Left shift (`sll`) multiplies the value by powers of 2 by shifting bits to the left.
4. Right shift (`srl`) divides the value by powers of 2 by shifting bits to the right.

18.8 Conditional Branches

In high-level languages, this is done using the `if` statement. In assembly, it is slightly more complex. The only tool you have is the ability to jump to a label based on the result of various comparisons. The relevant instructions are shown in the following table:
Let's begin with a simple `if` statement. The actual code inside the `if` block is arbitrary:

```
#include <stdio.h>

int main() {
    int a = 5;  // Example value for a

    if (a > 0) {
        a++;
    }
    a *= 2;

    printf("The value of a is: %d\n", a);
    return 0;
```

Name	Opcode	Format	Operation
Branch On Equal	`beq`	`beq rs, rt, label`	if (rs == rt) goto label
Branch On Not Equal	`bne`	`bne rs, rt, label`	if (rs != rt) goto label
Branch Less Than	`blt`	`blt rs, rt, label`	if (rs < rt) goto label
Branch Greater Than	`bgt`	`bgt rs, rt, label`	if (rs > rt) goto label
Branch Less Than Or Equal	`ble`	`ble rs, rt, label`	if (rs $\leq$ rt) goto label
Branch Greater Than Or Equal	`bge`	`bge rs, rt, label`	if (rs $\geq$ rt) goto label
Set Less Than	`slt`	`slt rd, rs, rt`	rd = (rs < rt) ? 1 : 0
Set Less Than Immediate	`slti`	`slti rd, rs, imm`	rd = (rs < imm) ? 1 : 0
Set Less Than Immediate Unsigned	`sltiu`	`sltiu rd, rs, imm`	rd = (rs < imm) ? 1 : 0
Set Less Than Unsigned	`sltu`	`sltu rd, rs, rt`	rd = (rs < rt) ? 1 : 0

Table 15: MIPS Branching Instructions

```
}
```

Assuming `a` is stored in $t0, the corresponding MIPS assembly code would look like this:

```
.data
    result_msg: .asciiz "The value of a is: "

.text

main:
    li $t0, 5               # Initialize a = 5

    # if (a > 0)
    blez $t0, skip_increment # If a <= 0, skip the increment

    addi $t0, $t0, 1        # a++ (increment a)

skip_increment:
    sll $t0, $t0, 1         # a *= 2 (shift left by 1 to multiply by 2)

    # Print the result
    li $v0, 4               # Syscall for printing a string
    la $a0, result_msg      # Load address of result_msg
    syscall

    li $v0, 1               # Syscall for printing integer
    move $a0, $t0           # Move the result of a to £a0 for printing
    syscall

    # Exit the program
    li $v0, 10              # Syscall to exit
    syscall
```

There are a few things to note here. First, in assembly, **we test for the opposite condition compared to the C code**. This is because we are jumping over the block of code if the condition is false, whereas in C, we enter the block if the condition is true. Translating from C to assembly can be easier if we rewrite the C code using jump logic:

```
#include <stdio.h>

int main() {
    int a = 5;  // Example value for a

    if (a <= 0)
        goto less_eq_0;

    a++;

    less_eq_0:
    a *= 2;

    printf("The value of a is: %d\n", a);
```

```c
    return 0;
}
```

Another thing to notice is the multiplication operation. This isn't directly related to branching but is an important concept. As a human compiler, your goal is to match the behavior, not the exact structure of the high-level code (unless specifically required). In MIPS, we use the `sll` (shift left logical) instruction instead of multiplication by 2, because it is more efficient in terms of performance.

What if we want to use the same condition?

```
.data
    result_msg: .asciiz "The value of a is: "

.text

main:
    li $t0, 5               # Initialize a = 5

    # if (a > 0)
    bgtz $t0, do_increment  # If a > 0, jump to do_increment

    j skip_increment        # Additional branch to skip_increment

do_increment:
    addi $t0, $t0, 1        # a++ (increment a)

skip_increment:
    sll $t0, $t0, 1         # a *= 2 (shift left by 1 to multiply by 2)

    # Print the result
    li $v0, 4               # Syscall for printing a string
    la $a0, result_msg      # Load address of result_msg
    syscall

    li $v0, 1               # Syscall for printing integer
    move $a0, $t0           # Move the result of a to $a0 for printing
    syscall

    # Exit the program
    li $v0, 10              # Syscall to exit
    syscall
```

You can see we need to use one more jump operation, so it is less efficient that the previous version.

18.8.1 if-else

Let's now look at an `if-else` example. Assume `a` and `b` are stored in $t0 and $t1, respectively:

```c
#include <stdio.h>

int main() {
    int a = 5;   // Example value for a
    int b = 20;  // Example value for b

    if (a > 0) {
        b = 100;
    } else {
        b -= 50;
    }

    printf("The value of b is: %d\n", b);
    return 0;
}
```

Here are two possible MIPS implementations:

```
.data
    result_msg: .asciiz "The value of b is: "

.text
.globl main

main:
    # Initialize variables
    li $t0, -5           # a = 5 (example value for a)
    li $t1, 20          # b = 20 (initial value for b)

    # if (a > 0)
    bgez $t0, true_part # If a >= 0, jump to greater_0 (true part)

    # Else part (a < 0)
    addi $t1, $t1, -50  # b -= 50
    j end_if         # Jump to the end of the if-else structure

true_part:
    # True part (a >= 0)
    li $t1, 100         # b = 100

end_if:
    # Print the result
    li $v0, 4           # Syscall for printing a string
    la $a0, result_msg  # Load address of result_msg
    syscall

    li $v0, 1           # Syscall for printing integer
    move $a0, $t1       # Move the result of b to £a0 for printing
    syscall

    # Exit the program
    li $v0, 10          # Syscall to exit
    syscall

```

Or:

```
.data
    newline: .asciiz "\n"
    result_msg: .asciiz "The value of b is: "

.text

main:
    # Initialize variables
    li $t0, -5           # a = 5
    li $t1, 20          # b = 20

    # if (a > 0)
    blez $t0, false_part # If a <= 0, jump to else_part

    # True part (a > 0)
    li $t1, 100         # b = 100
    j end_if            # Jump to end of if statement

false_part:
    # False part (a <= 0)
    subi $t1, $t1, 50   # b -= 50

```

```
end_if:
    # Print the result
    li $v0, 4           # Syscall for printing a string
    la $a0, result_msg  # Load address of result_msg
    syscall

    li $v0, 1           # Syscall for printing integer
    move $a0, $t1       # Move the result of b to £a0 for printing
    syscall

    # Exit the program
    li $v0, 10          # Syscall to exit
    syscall

```

In the first version, the order of the actual code is swapped to keep the conditions the same as in C, while the second version inverts the condition to maintain the same block order as in C. In both cases, an unconditional jump is required to avoid falling into the `else` block, which makes the code more complicated than necessary.

We can simplify the logic by rewriting the C code as:

```c
#include <stdio.h>

int main() {
    int a = 5;   // Example value for a
    int b = 20;  // Example value for b

    b -= 50;     // Decrement b by 50 first

    if (a > 0) {
        b = 100; // If a > 0, override b to 100
    }

    printf("The value of b is: %d\n", b);
    return 0;
}
```

This translates to:

```
.data
    result_msg: .asciiz "The value of b is: "

.text

main:
    # Initialize variables
    li $t0, 5               # a = 5 (example value)
    li $t1, 20              # b = 20 (initial value for b)

    # Perform b -= 50
    addi $t1, $t1, -50      # b -= 50 (decrement b by 50)

    # Check if a <= 0
    ble $t0, $zero, end_if  # If a <= 0, jump to less_eq_0

    # True part (a > 0)
    li $t1, 100             # b = 100

end_if:
    # Print the result
    li $v0, 4               # Syscall for printing a string
    la $a0, result_msg      # Load address of result_msg
    syscall

    li $v0, 1               # Syscall for printing integer
    move $a0, $t1           # Move the result of b to £a0 for printing
```

```
    syscall

    # Exit the program
    li $v0, 10              # Syscall to exit
    syscall
```

This is an example of rearranging code to simplify the assembly. You save on instructions and avoid unnecessary jumps and labels.

18.8.2 Compound Conditions

We have covered simple conditions, but how do we handle compound conditions in MIPS? Let's start with an `and` condition. Assume `a`, `b`, and `c` are stored in $t0, $t1, and $t2, respectively:

```c
#include <stdio.h>

int main() {
    int a = 15;     // Example value for a
    int b = 200;    // Example value for b
    int c = 50;     // Example initial value for c

    if (a > 10 && a < b) {
        c += 20;
    }

    b &= 0xFF;      // Mask b with 0xFF (only keep the lower 8 bits)

    printf("The value of b is: %d\n", b);
    printf("The value of c is: %d\n", c);

    return 0;
}
```

To handle this in MIPS, we need to invert the condition:

```c
#include <stdio.h>

int main() {
    int a = 15;     // Example value for a
    int b = 200;    // Example value for b
    int c = 50;     // Example initial value for c

    if (a <= 10 || a >= b)
        goto no_add20;

    c += 20;

no_add20:
    b &= 0xFF;      // Mask b with 0xFF (only keep the lower 8 bits)

    printf("The value of b is: %d\n", b);
    printf("The value of c is: %d\n", c);

    return 0;
}
```

This becomes:

```
.data
    result_b_msg: .asciiz "The value of b is: "
    result_c_msg: .asciiz "The value of c is: "
    newline: .asciiz "\n"

.text

main:
    # Initialize variables
    li $t0, 15          # a = 15 (example value for a)
    li $t1, 200         # b = 200 (example value for b)
    li $t2, 50          # c = 50 (initial value for c)

```

```
    # if (a <= 10 || a >= b) goto no_add20;
    li $t3, 10           # Load constant 10
    ble $t0, $t3, no_add20  # if (a <= 10) goto no_add20

    bge $t0, $t1, no_add20  # if (a >= b) goto no_add20

    # c += 20;
    addi $t2, $t2, 20    # Increment c by 20

no_add20:
    # b &= 0xFF;
    andi $t1, $t1, 0xFF  # Mask b with 0xFF (keep only the lower 8 bits)

    # Print the result of b
    li $v0, 4            # Syscall for printing a string
    la $a0, result_b_msg # Load address of result_b_msg
    syscall

    li $v0, 1            # Syscall for printing integer
    move $a0, $t1        # Move the value of b to $a0 for printing
    syscall

    # Print newline
    li $v0, 4
    la $a0, newline
    syscall

    # Print the result of c
    li $v0, 4            # Syscall for printing a string
    la $a0, result_c_msg # Load address of result_c_msg
    syscall

    li $v0, 1            # Syscall for printing integer
    move $a0, $t2        # Move the value of c to $a0 for printing
    syscall

    # Exit the program
    li $v0, 10           # Syscall to exit
    syscall

```

This approach works by using short-circuit evaluation. `or` conditions, unlike `and`, do not need to remember past states; reaching a line in a multi-term `or` expression means that all previous checks were false. In contrast, `and` requires more jumps and labels.

```
.data
    result_b_msg: .asciiz "The value of b is: "
    result_c_msg: .asciiz "The value of c is: "
    newline: .asciiz "\n"

.text

main:
    # Initialize variables
    li $t0, 15           # a = 15 (example value for a)
    li $t1, 200          # b = 200 (example value for b)
    li $t2, 50           # c = 50 (initial value for c)

    # if (a > 10 && a < b)
    li $t3, 10           # Load constant 10 into $t3
    bgt $t0, $t3, check_b_less_than_a   # if (a > 10), proceed to check a < b
```

```
17
18      j skip_add_to_c      # If a <= 10, skip adding 20 to c
19
20  check_b_less_than_a:
21      blt $t0, $t1, add_to_c  # if (a < b), proceed to add 20 to c
22
23  skip_add_to_c:
24      # b &= 0xFF;
25      andi $t1, $t1, 0xFF  # Mask b with 0xFF (keep only the lower 8 bits)
26
27      # Print the result of b
28      li $v0, 4            # Syscall for printing a string
29      la $a0, result_b_msg # Load address of result_b_msg
30      syscall
31
32      li $v0, 1            # Syscall for printing integer
33      move $a0, $t1        # Move the value of b to $a0 for printing
34      syscall
35
36      # Print newline
37      li $v0, 4
38      la $a0, newline
39      syscall
40
41      # Print the result of c
42      li $v0, 4            # Syscall for printing a string
43      la $a0, result_c_msg # Load address of result_c_msg
44      syscall
45
46      li $v0, 1            # Syscall for printing integer
47      move $a0, $t2        # Move the value of c to $a0 for printing
48      syscall
49
50      # Exit the program
51      li $v0, 10           # Syscall to exit
52      syscall
53
54  add_to_c:
55      # c += 20;
56      addi $t2, $t2, 20    # Increment c by 20
57      j skip_add_to_c      # Jump to skip_add_to_c
58
```

Finally, unless your professor specifically requires you to use the `slt` family of opcodes, you can avoid using pseudoinstructions like `beq`, `bne`, and `j` in favor of more efficient alternatives.

18.9 Loops

Before diving into MIPS assembly, it's important to cover a concept that might be obvious to some but new to others: any loop structure can be converted into another loop structure (with the possible addition of an `if` statement). For example, a `for` loop can be written as a `while` loop and vice versa. Even a `do-while` loop can be rewritten as a `for` or `while` loop. Let's explore some of these equivalencies:

```c
#include <stdio.h>

int main() {
    int a = 5;

    // For loop
    printf("For loop:\n");
    for (int i = 0; i < a; i++) {
        printf("i = %d\n", i);  // do_something
    }

    // While loop
```

```
    printf("\nWhile loop:\n");
    int i = 0;
    while (i < a) {
        printf("i = %d\n", i);  // do_something
        i++;
    }

    // Do-while loop
    printf("\nDo-while loop:\n");
    i = 0;
    if (i < a) {
        do {
            printf("i = %d\n", i);  // do_something
            i++;
        } while (i < a);
    }

    return 0;
}
```

When writing assembly code, it's often helpful to think in terms of `while` or `do-while` loops rather than `for` loops. These loops tend to resemble assembly code more closely in terms of where the instructions are placed. Just as in the previous chapter, where we discussed converting `if-else` statements into "jump form" or "branch form", we can do the same with loops, converting a `for` loop into a `while` loop in our minds before translating it to assembly.

Let's now apply "**branch form**" to a loop:

```
#include <stdio.h>

int main() {
    int a = 5; // Example value for 'a'
    int i = 0;

    if (i >= a)
        goto done_loop;  // If initial condition fails, skip the loop

loop:
    printf("i = %d\n", i);  // do_something
    i++;
    if (i < a)
        goto loop;          // Continue looping if condition holds

done_loop:
    printf("Loop is done.\n");

    return 0;
}
```

Let's now apply "**jump form**" to a loop:

```
#include <stdio.h>

int main() {
    int a = 5; // Example value for 'a'
    int i = 0;

loop:
    if (i >= a)
        goto done_loop;  // Exit the loop if i >= a

    printf("i = %d\n", i);  // do_something
    i++;
    goto loop;               // Jump back to the start of the loop

done_loop:
    printf("Loop is done.\n");

    return 0;
}
```

In assembly language and lower-level programming, the choice between **placing the condition check at the top** (jump form) versus **at the bottom** (branch form) of a loop structure can have meaningful effects on readability and performance.

When the condition check is placed **at the bottom** of the loop (branch form), the loop structure more naturally matches high-level constructs like a `do-while` loop, which is guaranteed to execute at least once. This approach has two key advantages:

1. The condition is placed at the bottom, matching the structure of a high-level `do-while` loop.
2. It requires only one jump and one label, rather than two, since the jump back to the beginning is only needed if the condition remains true after the first iteration.

Let's look at an example to demonstrate this using both jump and branch forms. Here, we'll use a C program that prints values of `i` from `0` to `a - 1`, where `a` is a positive integer.

```c
#include <stdio.h>

int main() {
    int a = 5; // Example value for 'a'
    int i = 0;

top:
    if (i >= a)  // Check the condition at the top
        goto done;

    printf("i = %d\n", i);  // do_something
    i++;
    goto top;                // Jump back to the top of the loop

done:
    printf("Loop is done.\n");

    return 0;
}
```

Suppose the body will be executed one time:

- **Condition Check at the Top**: The `if (i >= a)`
- **Goto top**: one jump
- **Condition Check again**: The `if (i >= a)`
- **Goto done**: one jump

```c
#include <stdio.h>

int main() {
    int a = 5; // Example value for 'a'
    int i = 0;

loop:
    printf("i = %d\n", i);  // do_something
    i++;

    if (i < a)               // Check the condition at the bottom
        goto loop;           // Branch back to loop if condition holds

    printf("Loop is done.\n");

    return 0;
}
```

Suppose the body will be executed one time:

- **Condition Check at the Bottom**: if (i < a)
- no jump (loop is useful for more than one body execution)

In general, the **branch form** (condition at the bottom) is preferable **when you know the loop will run at least once**. It requires only one jump and one label, whereas the **jump form** (condition at the top) requires both an initial jump to start the loop and another jump after each iteration, resulting in extra instructions.

Now that we've discussed the theory and structure, let's look at a simple example in MIPS assembly.

```c
int sum = 0;
for (int i = 0; i < 100; i++) {
    sum += i;
}
```

This basic loop adds the numbers from 0 to 99. The equivalent MIPS assembly code looks like this:

```
li $t0, 0        # sum = 0
li $t1, 1        # i = 1 (starting from 1 because adding 0 is trivial)
li $t2, 100
loop:
    addi $t0, $t0, $t1  # sum += i
    addi $t1, $t1, 1    # i++
    blt $t1, $t2, loop  # while (i < 100)
```

18.10 Iterate on Fixed-size 1D Array

Consider an array `int numbers[10];` that is filled with 10 random integers. The following C code computes the sum of the array elements:

```
int total = 0;
for (int i = 0; i < 10; i++) {
    total += numbers[i];
}
```

There are various ways to translate this into MIPS assembly code. The first approach is a literal translation:

```
li $t0, 0        # total = 0
li $t1, 0        # i = 0
la $t2, numbers  # t2 = numbers
li $t3, 10
sum_loop:
    sll $t4, $t1, 2  # t4 = i * sizeof(int) == i * 4
    add $t4, $t4, $t2  # t4 = &numbers[i]
    lw $t4, 0($t4)  # t4 = numbers[i]
    add $t0, $t0, $t4  # total += numbers[i]
    addi $t1, $t1, 1  # i++
    blt $t1, $t3, sum_loop  # while (i < 10)
```

Here, we initialize the relevant variables beforehand (loading `numbers` and `10` during every iteration would be inefficient). The multiplication by 4 in the line `i * 4` is required because each array element occupies 4 bytes (the size of an `int` in MIPS). When accessing the i-th element of an array, high-level languages automatically multiply `i` by the size of the element's type to compute the correct memory address.
Once we calculate the element's address, we load its value from memory using `lw`, since this is an array of words (4-byte integers). Afterward, we add the value to `total`, increment `i`, and check the loop condition.
This direct translation is functional but not necessarily the simplest or most efficient. A more efficient approach is to use a pointer to the current element and increment the pointer during each iteration. The equivalent C code is:

```
int* p = &numbers[0];
int i = 0, total = 0;
do {
    total += *p;
    i++;
    p++;
} while (i < 10);
```

In this case, we initialize `p` to point to the first element and increment it on each iteration to point to `numbers[i]`. Since pointer arithmetic in C increments by the size of the type (4 bytes for integers), `p++` increments the pointer by `sizeof(int)`. The corresponding MIPS code is:

```
li $t0, 0        # total = 0
li $t1, 0        # i = 0
la $t2, numbers  # p = numbers
li $t3, 10
sum_loop:
    lw $t4, 0($t2)  # t4 = *p
    add $t0, $t0, $t4  # total += *p
    addi $t1, $t1, 1  # i++
```

```
    addi $t2, $t2, 4  # p++ i.e., p += sizeof(int)
    blt $t1, $t3, sum_loop  # while (i < 10)
```

Finally, a further optimization involves controlling the loop using a pointer and an "end" pointer, as shown in this C code:

```
int* p = &numbers[0];
int* end = &numbers[10];
int total = 0;
do {
    total += *p;
    p++;
} while (p < end);
```

In this version, we use p and an "end" pointer to iterate through the array, eliminating the need for an index variable. The equivalent MIPS code is:

```
li $t0, 0        # total = 0
la $t2, numbers  # p = numbers
addi $t3, $t2, 40  # end = numbers + 10 * sizeof(int)
sum_loop:
    lw $t4, 0($t2)  # t4 = *p
    add $t0, $t0, $t4  # total += *p
    addi $t2, $t2, 4  # p++ i.e., p += sizeof(int)
    blt $t2, $t3, sum_loop  # while (p < end)
```

This version reduces the instruction count from 10 to 7, and the savings are even greater if multiplication is involved. This optimization becomes even more significant when dealing with multi-dimensional arrays, which we will discuss in the next section.

In certain situations, **unrolling the loop** (manually expanding the loop body to perform multiple iterations in one cycle) can reduce the overhead of branching and improve performance by taking advantage of instruction-level parallelism. **Unrolled loops can reduce the overhead of branching and increase performance, especially for large datasets.** For example, instead of processing one array element per loop iteration, you could process two or four at a time.

The following is an example of loop unrolling by 2 in MIPS assembly, including boundary checking to handle cases where the array size is not divisible by 2:

```
# Assume array size is stored in £t3, and the base address of numbers is in £t2
# Unroll the loop to process two elements at a time

li $t0, 0              # total = 0
la $t2, numbers        # p = numbers

# Calculate how many full iterations we can do
srl $t4, $t3, 1        # t4 = size / 2, for two elements per iteration

# Loop to process two elements per iteration
unrolled_loop:
    lw  $t5, 0($t2)    # load numbers[i]
    lw  $t6, 4($t2)    # load numbers[i+1]
    add $t0, $t0, $t5  # total += numbers[i]
    add $t0, $t0, $t6  # total += numbers[i+1]
    addi $t2, $t2, 8   # p += 2 * sizeof(int)
    addi $t4, $t4, -1  # decrement loop counter
    bgtz $t4, unrolled_loop  # while (i < size / 2)

# Check if there's one leftover element
andi $t5, $t3, 1       # check if size is odd (size % 2 != 0)
                       # andi £t5, £t3, 3 check if size is not divisible by 4, why?
beqz $t5, done         # if no leftover, we're done
```

```
24
25    # Handle the leftover element
26    lw  $t6, 0($t2)         # load numbers[size - 1]
27    add $t0, $t0, $t6       # total += numbers[size - 1]
28
29    done:
30    # Total is in £t0
```

- The loop is unrolled to process two elements per iteration by loading and summing two array elements.
- After the main loop, a check is performed to see if the array size is odd (i.e., whether there's a leftover element).
- If the size is odd, the last element is handled separately.
- The total sum is stored in register `$t0`.

18.11 Iterate on Fixed-size 2D Array

The first step in understanding fixed-size 2D arrays in C is recognizing how they are laid out in memory. A fixed-size 2D array is stored in **row-major order**, meaning that all elements from the first row are followed by the elements of the second row, and so on. Essentially, a fixed-size 2D array is equivalent to a fixed-size 1D array with `rows` × `cols` elements arranged in the same order. For example:

```c
#define ROWS 2
#define COLS 4
// The memory of these two arrays is identical
int array[ROWS][COLS] = { { 1, 2, 3, 4 }, { 5, 6, 7, 8 } };
int array1d[ROWS*COLS] = { 1, 2, 3, 4, 5, 6, 7, 8 };
```

This means that when we declare a fixed-size 2D array, it is effectively a fixed-size 1D array with a size equal to `rows` × `columns`. Therefore, when looping through a fixed-size 2D array, it is possible to treat it like a fixed-size 1D array using a single loop. Let's look at an example:

```c
#define ROWS 2
#define COLS 4

int main(){

int array[ROWS][COLS];

for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; ++j) {
        array[i][j] = i + j;
    }
}

return 0;
}
```

Assuming `rows` and `cols` are stored in `$a0` and `$a1` (and both are nonzero), the equivalent MIPS assembly code would look like this:

```
1     la $t0, array       # p = &array[0]
2     li $t1, 0           # i = 0
3     looprows:
4         li $t2, 0       # j = 0
5     loopcols:
6         add $t3, $t1, $t2  # t3 = i + j
7         # Calculate the byte offset of element array[i][j]
8         mult $t1, $a1
9         mflo $t4        # i * cols
10        add $t4, $t4, $t2  # t4 = i * cols + j
11        sll $t4, $t4, 2   # t4 = (i * cols + j) * sizeof(int)
12        add $t4, $t4, $t0  # t4 = &array[i][j]
13        sw $t3, 0($t4)  # array[i][j] = i + j
14        addi $t2, $t2, 1  # j++
15        blt $t2, $a1, loopcols  # while (j < cols)
16        addi $t1, $t1, 1  # i++
17        blt $t1, $a0, looprows  # while (i < rows)
```

In this literal translation, the chunk of code calculating the offset for each element is not only much slower than iterating through the array via pointers but also much more complex. Imagine how much worse it would be for a 3D array with three nested loops.

For comparison, we can take advantage of the 1D memory arrangement and pointer iteration:

```c
#define ROWS 2
#define COLS 4

int main() {
    int array[ROWS][COLS];  // Define the 2D array
    int *p = &array[0][0];  // Pointer to the first element of the array

    // Nested for loops for rows and columns
    for (int i = 0; i < ROWS; i++) {
        for (int j = 0; j < COLS; j++) {
            int t3 = i + j;  // Calculate t3 = i + j
            *p = t3;         // Assign value to the array at position (i, j) using pointer
            p++;             // Move pointer to the next element in the array
        }
    }
```

The equivalent MIPS assembly code would look like this:

```
  la $t0, array       # p = &array[0]
  li $t1, 0           # i = 0
  looprows:
      li $t2, 0       # j = 0
  loopcols:
      add $t3, $t1, $t2  # t3 = i + j
      sw $t3, 0($t0)  # array[i][j] = *p = i + j
      addi $t2, $t2, 1  # j++
      addi $t0, $t0, 4  # p++ (keep pointer in sync with i and j)
      blt $t2, $a1, loopcols
      addi $t1, $t1, 1  # i++
      blt $t1, $a0, looprows
```

This version is shorter, but the extra label and branch instructions may not be worth the complexity in all cases. We can further simplify our code below:

```c
for (int i = 0; i < rows * cols; i++) {
    int r = i / cols;
    int c = i % cols;
    array[i] = r + c;
}
```

The equivalent MIPS assembly code would look like this:

```
  la $t0, array       # p = &array[0]
  li $t1, 0           # i = 0
  mult $a0, $a1       # a0 = rows, a1 = cols
  mflo $t2            # t2 = rows * cols
  loop:
      div $t1, $a1
      mflo $t3        # r = i / cols
      mfhi $t4        # c = i % cols
      add $t3, $t3, $t4  # t3 = r + c
      sw $t3, 0($t0)  # array[i] = *p = r + c
      addi $t1, $t1, 1  # i++
      addi $t0, $t0, 4  # p++ (keep pointer in sync with i, p = &array[i])
      blt $t1, $t2, loop  # while (i < rows * cols)
```

You might wonder whether it is worth converting to a single loop, especially when you still need the original `i` and `j` as if using nested loops. In general, it is preferable to avoid nested loops in assembly if possible.

18.12 1D Dynamic Array

In C, dynamic memory is typically allocated using functions like malloc, and the equivalent in MIPS is done using a system call to allocate memory. Let's first look at an example of dynamically allocating an array in C using malloc and then the corresponding MIPS assembly code.

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int i;
    int size = 5;

    // Dynamically allocate an array of 5 integers
    int *arr = (int *)malloc(size * sizeof(int));

    // Check if allocation succeeded
    if (arr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    // Initialize the array with values 1, 2, 3, 4, 5
    for (i = 0; i < size; i++) {
        arr[i] = i + 1;
    }

    // Print the array
    for (i = 0; i < size; i++) {
        printf("arr[%d] = %d\n", i, arr[i]);
    }

    // Free the allocated memory
    free(arr);

    return 0;
}
```

In MIPS, there is no built-in malloc function, so we use the sbrk system call to allocate memory dynamically. The sbrk system call adjusts the program's heap space to allocate memory. The heap grows upward in memory (towards higher addresses), opposite to the stack.

```
.data
msg_index:  .asciiz "arr["          # First part of the message: "arr["
msg_equals: .asciiz "] = "          # Middle part of the message: "] = "
msg_fail:   .asciiz "Memory allocation failed\n"  # Error message
newline:    .asciiz "\n"            # Newline character

.text

main:

    li $s0, 5                       # size = 5
    li $a0, 20                      # a0 = size * 4

    # System call: sbrk (Allocate heap memory)
    li $v0, 9                       # syscall code for sbrk
    syscall                         # Allocate memory (returns address in $v0)

    # Check if memory allocation was successful (v0 != -1)
    bltz $v0, alloc_failed          # If v0 is -1, allocation failed

    # Save the allocated array base address in $s1
    move $s1, $v0                   # Save the base address of the array in $s1

    # Initialize the array with values (1, 2, 3, 4, 5)
    li $t0, 0                       # i = 0
    move $t1, $s0                   # size is 5

init_loop:
```

```
    bge $t0, $t1, print_loop        # If i >= size, break out of init loop
    addi $t3, $t0, 1                # t3 = i + 1
    mul $t4, $t0, 4                 # t4 = i * 4 (offset for integers)
    add $t5, $s1, $t4               # t5 = base address + offset
    sw $t3, 0($t5)                  # Store (i + 1) into the array
    addi $t0, $t0, 1                # i++
    j init_loop                     # Jump back to continue initialization

# Print the array
print_loop:
    li $t0, 0                       # i = 0
    move $t1, $s0                   # size
    move $t2, $s1                   # base address
print_loop_start:
    bge $t0, $t1, end_program       # If i >= size, break out of print loop
    mul $t3, $t0, 4                 # t4 = i * 4 (offset)
    add $t4, $t2, $t3               # t5 = base address + offset
    lw $a1, 0($t4)                  # Load the value of arr[i]

    # Print "arr["
    li $v0, 4                       # syscall: print string
    la $a0, msg_index               # Load address of "arr["
    syscall

    # Print the index (i)
    move $a0, $t0                   # a0 = i (array index)
    li $v0, 1                       # syscall: print integer
    syscall

    # Print "] = "
    li $v0, 4                       # syscall: print string
    la $a0, msg_equals              # Load address of "] = "
    syscall

    # Print the value of arr[i]
    move $a0, $a1                   # a0 = value of arr[i]
    li $v0, 1                       # syscall: print integer
    syscall

    # Print newline
    li $v0, 4                       # syscall: print string
    la $a0, newline                 # Load address of newline
    syscall

    addi $t0, $t0, 1                # i++
    j print_loop_start              # Jump back to continue printing

# Memory allocation failed
alloc_failed:
    li $v0, 4                       # syscall: print string
    la $a0, msg_fail                # Load address of the failure message
    syscall                         # Print "Memory allocation failed"
    j end_program                   # Jump to end

# End program
end_program:
    li $v0, 10                      # syscall: exit
    syscall
```

Dynamic Memory Allocation Using `sbrk`: The `sbrk` system call (code 9 in $v0) is used to allocate memory dynamically on the heap. The size of memory to be allocated (in bytes) is passed in $a0, and the address of the allocated memory is returned in $v0. If the allocation fails, $v0 will be -1. The heap memory is accessed by calculating the

appropriate address (base address + offset). In this example, the base address of the array is stored in $s1, and offsets are calculated based on the element index.

Since MARS doesn't support shrinking the heap, you must either:

- Manage heap memory usage in your program by keeping track of allocated areas.
- Restart the program to reset the memory state if necessary.

You can reuse portions of the previously allocated heap memory. This means you might keep track of allocated memory addresses and manually repurpose that memory for other data. In MARS, the memory is automatically reclaimed after program execution ends.

18.13 Functions

In assembly language, a function is essentially a label with an associated return instruction. For example, the C function `void func1() {}` would be written in assembly as follows:

```
# void func1()
func1:
  # body goes here
  jr $ra
```

To call a function in assembly, you use the `Jump and Link (jal)` instruction. For example, let's modify a simple "Hello World" program to call a function:

```
.data
hello: .asciiz "Hello World!\n"

.text
main:
  jal hello_world

  li $v0, 10  # exit syscall
  syscall

# void hello_world()
hello_world:
  li $v0, 4  # print string system call
  la $a0, hello  # load the address of the string to print into £a0
  syscall
  jr $ra
```

To avoid the "address out of range" error, always ensure that the main section is defined correctly at the top of the text segment, before any function definitions or data declarations.

The `jal` instruction saves the address of the next instruction into the `$ra` (return address) register and performs an unconditional jump to the specified function label. Open the MARS simulator, the text segment is listed below:

Address	Machine Code	Basic	Source
0x00400000	0x0c100003	jal 0x0040000c	jal hello_world
0x00400004	0x2402000a	addiu $2,$0,0x0000000a	li $v0, 10
0x00400008	0x0000000c	syscall	syscall
0x0040000c	0x24020004	addiu $2,$0,0x00000004	li $v0, 4
0x00400010	0x3c011001	lui $1,0x00001001	la $a0, hello
0x00400014	0x34240000	ori $4,$1,0x00000000	
0x00400018	0x0000000c	syscall	syscall
0x0040001c	0x03e00008	jr $31	jr $ra

and then you check the value stored in the *ra*, it is `0x00400004`, which is the address of the next instruction after function call. Instruction `jr` means "jump register".

18.14 Stack and Stack Frames

MIPS has 32 general-purpose registers, each with a specific convention for its use. Understanding how each register is conventionally used will help you write correct and efficient MIPS programs. Here is an overview of the key registers and their usage:

Register	Name	Usage Convention
$0	`zero`	Always holds the constant 0.
$v0-$v1	`values`	Used for function return values and syscall codes.
$a0-$a3	`arguments`	Used to pass arguments to functions (up to 4 arguments).
$t0-$t9	`temporaries`	Temporary registers, caller-saved (not preserved across function calls).
$s0-$s7	`saved`	Callee-saved registers (preserved across function calls).
$ra	`return address`	Holds the return address for function calls.
$sp	`stack pointer`	Points to the top of the stack in memory.
$fp	`frame pointer`	Used as a frame pointer in some function call conventions.
$gp	`global pointer`	Points to the middle of the static data segment.
$at	`assembler temporary`	Reserved for the assembler.
$k0-$k1	`kernel`	Reserved for operating system (kernel) use.

Table 16: MIPS General-Purpose Registers and Conventions

Choosing Between $t and $s Registers:

- Temporary Registers ($t0 to $t9):
  - Caller-Saved: Temporary registers are caller-saved, meaning the caller must save them if it wants to keep their values across a subroutine call. The callee does not need to preserve their values. Use temporary registers when you don't need to keep the value after calling another subroutine. If you do need the value after a subroutine call, the caller (the calling function) should save the temporary registers on the stack before making the subroutine call and restore them afterward.

```
add $t0, $a0, $a1     # Store result in $t0 (temporary)
jal some_function     # Call a function ($t0 is not guaranteed to be preserved)
```

- Saved Registers ($s0 to $s7):
  - Callee-Saved: Saved registers are callee-saved, meaning the callee (the function being called) must preserve their values. The callee must save the original values on the stack and restore them before returning control to the caller. Saved registers are used when you need to preserve values across subroutine calls. The function itself is responsible for saving and restoring the saved registers.

```
# Using $s registers for persistent values
addi $s0, $a0, 5      # Store a value in $s0 (saved)
jal some_function     # Call a subroutine
# $s0 is guaranteed to hold the value after the subroutine returns
```

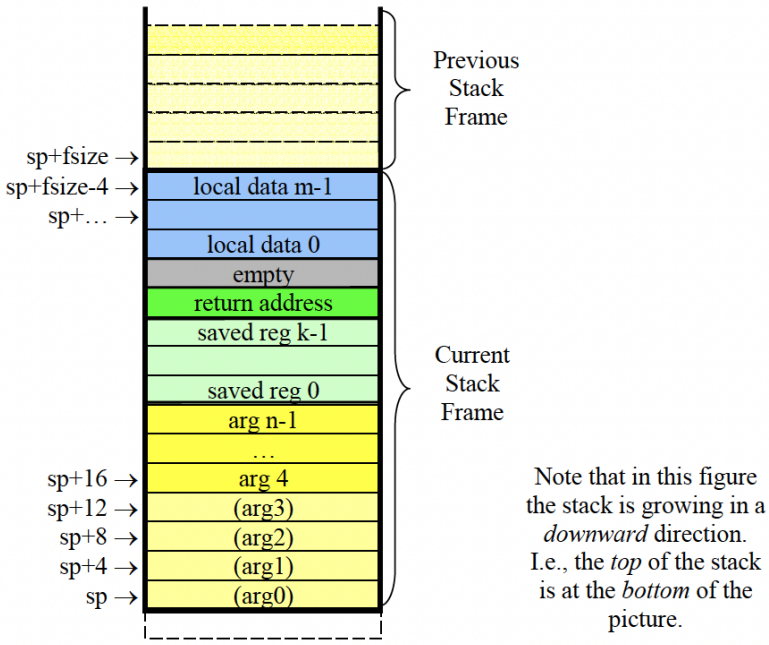
Each time a subroutine is invoked, a unique stack frame is created for that specific instance of the subroutine call. In the case of a recursive subroutine call, multiple stack frames are created—one for each instance. The organization of the stack frame is significant for two reasons:

1. It establishes a contract between the caller and the callee, defining how arguments are passed between them, how the function's result is returned, and how registers are shared between the caller and callee.
2. It specifies the organization of storage local to the callee within its stack frame.

In general, the stack frame for a subroutine may contain the following:

- A place to store the values of saved registers ($s0 to $s7).
- A place to store the subroutine's return address ($ra).
- Space for local data storage.
- Space for arguments.

The following figure illustrates the general organization of a stack frame, which is divided into five regions:



Subroutines do not always need every section in their stack frame. We can classify subroutines as follows:

- **Simple Leaf Subroutines:** These do not call other subroutines and do not use memory space on the stack. They do not require a stack frame and do not modify $sp.
- **Leaf Subroutines with Data:** These subroutines do not call other subroutines but require stack space for local variables or saved registers. They push a stack frame (size being a multiple of 4) but do not save $ra in the stack.
- **Non-leaf Subroutines:** These call other subroutines and generally have all sections in their stack frame.

Below are examples for each of these cases.

18.14.1 Simple Leaf Subroutine

```c
#include <stdio.h>

int square(int x) {
    return x * x;
}

int main() {
    int result = square(5);
    printf("Result: %d\n", result);
    return 0;
}
```

```
.data
msg: .asciiz "Result: "     # Message for printing result

.text

main:
    li $a0, 5               # Load the argument x = 5
    jal square              # Call square function
    move $a1, $v0           # Store the result in $a1

    # Print the message
    li $v0, 4               # syscall: print string
    la $a0, msg             # Load address of the message
```

```
    syscall

    # Print the result
    li $v0, 1                   # syscall: print integer
    move $a0, $a1               # Move result to $a0
    syscall

    # Exit the program
    li $v0, 10                  # syscall: exit
    syscall

# Simple leaf subroutine
square:
    mul $v0, $a0, $a0           # $v0 = $a0 * $a0
    jr $ra                      # Return to caller
```

18.14.2 Leaf Subroutine with Stack Usage

```c
#include <stdio.h>

int sum_of_squares(int x, int y) {
    int square_x = x * x;  // Local variable square_x
    int square_y = y * y;  // Local variable square_y
    return square_x + square_y;  // Return the sum of squares
}

int main() {
    int result = sum_of_squares(3, 4);
    printf("Result: %d\n", result);
    return 0;
}
```

```
.data
msg:    .asciiz "Result: "      # Message for printing result

.text

main:
    li $a0, 3                   # Argument x = 3
    li $a1, 4                   # Argument y = 4

    # Call sum_of_squares function
    jal sum_of_squares
    move $a1, $v0               # Move result to $a0

    # Print the message
    li $v0, 4                   # syscall: print string
    la $a0, msg                 # Load address of the message
    syscall

    # Print the result
    li $v0, 1                   # syscall: print integer
    move $a0, $a1               # Move result to $a0
    syscall

    # Exit the program
    li $v0, 10                  # syscall: exit
    syscall

# Leaf subroutine with stack usage and local variables
sum_of_squares:
    addi $sp, $sp, -8           # Allocate 8 bytes on the stack for local variables
    sw $s0, 0($sp)              # Save callee-saved register $s0
```

```
    sw $s1, 4($sp)              # Save callee-saved register £s1

    # Use £s0 and £s1 as local variables
    # square_x = x * x
    mul $s0, $a0, $a0           # s0 = x * x (store in local variable)

    # square_y = y * y
    mul $s1, $a1, $a1           # s1 = y * y (store in local variable)

    # sum = square_x + square_y
    add $v0, $s0, $s1           # v0 = s0 + s1 (sum result)

    # Restore callee-saved registers and clean up stack
    lw $s0, 0($sp)              # Restore £s0
    lw $s1, 4($sp)              # Restore £s1
    addi $sp, $sp, 8            # Free the stack space
    jr $ra                      # Return to caller
```

18.14.3 Nonleaf Subroutine

```c
#include <stdio.h>

// Leaf subroutine that computes the square of a number
int square(int x) {
    return x * x;
}

// Nonleaf subroutine that calls the 'square' subroutine
int sum_of_squares(int x, int y) {
    int a[3];                  // Local 1D array to store intermediate results

    a[0] = square(x);           // Store square(x) in a[0]
    a[1] = square(y);           // Store square(y) in a[1]
    a[2] = a[0] + a[1];           // Store square(y) in a[1]

    return a[2];        // Return the sum of squares
}

int main() {
    int result;

    // Call sum_of_squares with arguments 3 and 4
    result = sum_of_squares(3, 4);

    // Print the result
    printf("Result: %d\n", result);

    return 0;
}
```

```
.data
msg:    .asciiz "Result: "      # Message for printing the result

.text

main:
    # Set up arguments for sum_of_squares
    li $a0, 3                   # Argument x = 3
    li $a1, 4                   # Argument y = 4
    jal sum_of_squares           # Call sum_of_squares
    move $t0, $v0               # Move result to £t0

    # Print the message
    li $v0, 4                   # syscall: print string
    la $a0, msg                 # Load address of message
    syscall

```

```
    # Print the result
    li $v0, 1                   # syscall: print integer
    move $a0, $t0               # Move result to £a0
    syscall

    # Exit the program
    li $v0, 10                  # syscall: exit
    syscall


    # Leaf subroutine: square
square:
    mul $v0, $a0, $a0           # v0 = a0 * a0 (square the input)
    jr $ra                      # Return to caller

# Non-leaf subroutine: sum_of_squares
sum_of_squares:
    # Prologue - setup stack frame and save return address
    addiu $sp, $sp, -24         # Allocate 24 bytes for local variables (12 bytes for array
                                #+ 8 bytes for saved registers + 4 bytes for ra register)
    sw $ra, 20($sp)             # Save return address
    sw $s0, 16($sp)             # Save callee-saved register £s0
    sw $s1, 12($sp)             # Save callee-saved register £s1

    # Array 'a' will be a 1D array of size 3, starting at 0(£sp)
    # Use 12 bytes of stack for local array (three 4-byte integers)

    # Save arguments (x and y) to saved registers
    move $s0, $a0               # Save x into £s0
    move $s1, $a1               # Save y into £s1

    # First call to square(x)
    move $a0, $s0               # Pass x as argument to square
    jal square                  # Call square
    sw $v0, 0($sp)              # Store result of square(x) in array a[0] (0(£sp))

    # Second call to square(y)
    move $a0, $s1               # Pass y as argument to square
    jal square                  # Call square
    sw $v0, 4($sp)             # Store result of square(y) in array a[1] (4(£sp))

    # Sum the results stored in the array
    lw $t0, 0($sp)              # Load a[0] (square(x)) into £t0
    lw $t1, 4($sp)             # Load a[1] (square(y)) into £t1
    add $t2, $t0, $t1           # t2 = a[0] + a[1] (sum of squares)
    sw $t2, 8($sp)             # Store result in a[2] (8(£sp))

    move $v0, $t2               # Return the value of a[2] in £v0

    # Epilogue - restore registers and stack frame
    lw $ra, 20($sp)             # Restore return address
    lw $s0, 16($sp)             # Restore saved register £s0
    lw $s1, 12($sp)             # Restore saved register £s1
    addiu $sp, $sp, 24          # Deallocate stack space
    jr $ra                      # Return to caller
```

Key Differences Between Using Saved Registers and Pushing to the Stack:

1. Using Saved Registers ($s0, $s1):
    - Convention: The $s0-$s7 registers are callee-saved, meaning the callee (subroutine) is responsible for saving and restoring their values if they are used.

- Usage: You can load values into these registers and rely on them being preserved across subroutine calls (assuming you follow the convention of saving and restoring the registers in the prologue and epilogue).
- Performance: Using registers is generally faster than accessing memory (stack) because register access is direct and does not involve load/store instructions, which access memory.

2. Pushing Arguments to the Stack:

- Convention: When more than four arguments are passed to a subroutine, or when you have more data than can be stored in registers, it is common to use the stack to store the extra arguments or local variables.
- Usage: Pushing values to the stack ensures that they are available for later use. However, accessing these values requires extra load and store instructions.
- Flexibility: Using the stack can be useful when you have more arguments or local variables than you have registers for, or when you need temporary storage for arrays, structures, or large data.
- Example:If a function f(int a, int b, int c, int d, int e) has five arguments, the first four arguments (a, b, c, d) go into $a0, $a1, $a2, $a3, and the fifth argument (e) is pushed onto the stack.

```
    li $a0, 10       # first argument
    li $a1, 20       # second argument
    li $a2, 30       # third argument
    li $a3, 40       # fourth argument
    li $t0, 50       # fifth argument
    addi $sp, $sp, -4 # allocate stack space
    sw $t0, 0($sp)   # push fifth argument onto the stack
    jal f            # call the function f
```

In the function f, you can then retrieve the fifth argument from the stack:

```
f:
    lw $t0, 0($sp)   # get the fifth argument
    addiu $sp, $sp, 4 # restore the stack
    ...
```

- Use Saved Registers ($s0-$s7) when:
  1. You need to preserve values across multiple function calls.
  2. You have a small number of variables (typically up to 8) and you want faster access.
  3. You want to follow the convention that the callee is responsible for saving and restoring those registers.
- Use the Stack when:
  1. You have more local variables than available registers.
  2. You need to store large data (e.g., arrays, structs) that won't fit in registers.
  3. You want to preserve arguments for later use (e.g., across multiple subroutine calls or function returns).
  4. You are working with dynamic data or variable-length data structures that need to be allocated and deallocated.

18.15 Recursive Function

```
#include <stdio.h>

// Recursive function to calculate Fibonacci of n
int fib(int n) {
    if (n <= 1) {
        return n;  // Base case: fib(0) = 0, fib(1) = 1
    } else {
        return fib(n - 1) + fib(n - 2);  // Recursive case: fib(n) = fib(n-1) + fib(n-2)
    }
}

int main() {
    int num = 6;          // Example: compute Fibonacci of 6
    int result = fib(num);  // Call the Fibonacci function

```

```c
    // Print the result in a format similar to the MIPS assembly output
    printf("Fibonacci of %d is %d\n", num, result);

    return 0;  // End the program
}
```

```
.data
msg1:    .asciiz "Fibonacci of "   # First part of the message
msg2:    .asciiz " is "            # Second part of the message
newline: .asciiz "\n"              # Newline character

.text

# Main function
main:
    # Initialize the Fibonacci input number
    li $a0, 6                  # Example: compute Fibonacci of 6
    jal fib                    # Call fib(6)
    move $a1, $v0

    # Print "Fibonacci of "
    li $v0, 4                  # syscall: print string
    la $a0, msg1               # Load address of "Fibonacci of "
    syscall

    # Print the number (6)
    li $v0, 1                  # syscall: print integer
    li $a0, 6                  # Load the number (6) to print
    syscall

    # Print " is "
    li $v0, 4                  # syscall: print string
    la $a0, msg2               # Load address of " is "
    syscall

    # Print the result (Fibonacci value)
    li $v0, 1                  # syscall: print integer
    move $a0, $a1
    syscall                    # The result is already in $v0 from fib

    # Print newline
    li $v0, 4                  # syscall: print string
    la $a0, newline            # Load address of newline character
    syscall

    # Exit the program
    li $v0, 10                 # syscall: exit
    syscall

# Recursive Fibonacci function in MIPS
fib:
    # Prologue - save return address and $s0
    addi $sp, $sp, -12          # Allocate 12 bytes on the stack
    sw $ra, 0($sp)             # Save return address
    sw $s0, 4($sp)             # Save $s0
    sw $s1, 8($sp)             # Save $s1

    # Base case: if (n <= 1), return n
    # as the Fibonacci sequence for 0 and 1 is defined as 0 and 1, respectively.
    move $v0, $a0              # Prepare to return n (copy n into $v0)
    li $t0, 1                  # Load 1 into $t0
    ble $a0, $t0, exit_fib     # If n <= 1, exit the function

```

```
    # Recursive case: fib(n) = fib(n-1) + fib(n-2)
    move $s0, $a0               # Save n into £s0

    # Compute fib(n-2)
    addi $a0, $s0, -2           # a0 = n - 2
    jal fib                     # Call fib(n-2)
    move $s1, $v0                # save return of fib(n-2) in s1

    # Compute fib(n-1)
    addi $a0, $s0, -1            # a0 = n - 1 (restore n from £s0 and decrement)
    jal fib                      # Call fib(n-1)
    # Add fib(n-1) and fib(n-2)
    add $v0, $v0, $s1           # v0 = fib(n-1) + fib(n-2)

exit_fib:
    # Epilogue - restore saved registers and return address
    lw $ra, 0($sp)              # Restore return address
    lw $s0, 4($sp)              # Restore £s0
    lw $s1, 8($sp)              # Restore £s0
    addi $sp, $sp, 12            # Deallocate stack space
    jr $ra                      # Return to the caller
```

18.16 Floating Point

In 32-bit MIPS assembly language, floating-point operations are handled separately from integer operations using a set of floating-point registers and instructions. These floating-point registers are part of the Coprocessor 1 (CP1), which is dedicated to floating-point arithmetic. In MIPS, there are 32 floating-point registers, labeled as $f0 through $f31. These registers are used for single-precision (32-bit) and double-precision (64-bit) floating-point numbers.

- Single-Precision (32-bit) Floating Point: Each register holds one 32-bit floating-point value.
- Double-Precision (64-bit) Floating Point: A pair of consecutive registers hold a single 64-bit floating-point value. For example, $f0$ and $f1$ together can hold one 64-bit floating-point number.

Register(s)	**Use**	**Precision**
$f0$, $f2$	Function Return	Single
$f1$, $f3$	Paired with $f0$, $f2$ for Double Precision	Double
$f4$–$f10$	Temporary Registers	Single
$f5$–$f11$	Paired with $f4$–$f10$ for Double Precision	Double
$f12$, $f14$	Function Arguments (single)	Single
$f13$, $f15$	Paired with $f12$, $f14$ for Double Precision	Double
$f16$–$f18$	Temporary Registers	Single
$f17$, $f19$	Paired with $f16$, $f18$ for Double Precision	Double
$f20$–$f30$	Saved Registers	Single
$f21$–$f31$	Paired with $f20$–$f30$ for Double Precision	Double

Table 17: MIPS Floating-Point Registers and Their Uses

the x is either `s` for single precision or `d` for double precision. The y in the Compare instructions are one of `eq, lt, le`. Naturally, op would be the matching $==, <, <=$. Unfortunately, you don't get not equal, greater than, or greater equal, even as pseudo instructions, but it's easy enough to flip the order of operands or branch on the opposite result.

Name	Opcode	Format	Operation
Load Word to Coproc 1	lwc1 (or l.s)	lwc1 ft, n(rs)	F[ft] = M[R[rs]+n]
Store Word from Coproc 1	swc1 (or s.s)	swc1 ft, n(rs)	M[R[rs]+n] = F[ft]
Load Double to Coproc 1	ldc1 (or l.d)	ldc1 ft, n(rs)	F[ft] = M[R[rs]+n] F[ft+1] = M[R[rs]+n+4]
Store Double from Coproc 1	sdc1 (or s.d)	sdc1 ft, n(rs)	M[R[rs]+n] = F[ft] M[R[rs]+n+4] = F[ft+1]
Move From Coproc 1	mfc1	mfc1 rd, fs	R[rd] = F[fs]
Move To Coproc 1	mtc1	mtc1 rd, fs	F[fs] = R[rd]
Convert Word To Single Precision	cvt.s.w	cvt.s.w fd, fs	F[fd] = (float)F[fs]
Convert Single Precision To Word	cvt.w.s	cvt.w.s fd, fs	F[fd] = (int)F[fs]
Convert Word To Double Precision	cvt.d.w	cvt.d.w fd, fs	F[fd] = (double)F[fs]
Convert Double Precision To Word	cvt.w.d	cvt.w.d fd, fs	F[fd] = (int)F[fs]
Branch on FP True	bc1t	bc1t label	if (FPcond) goto label;
Branch on FP False	bc1f	bc1f label	if (!FPcond) goto label;
FP Compare	c.y.x	c.y.x fs, ft [y is one of `eq, lt, le`]	FPcond = (F[fs] op F[ft]) ? 1 : 0
Absolute Value	abs.x	abs.x fs, ft	F[fs] = (F[ft] > 0) ? F[ft] : -F[ft]
Add	add.x	add.x fd, fs, ft	F[fd] = F[fs] + F[ft]
Subtract	sub.x	sub.x fd, fs, ft	F[fd] = F[fs] - F[ft]
Multiply	mul.x	mul.x fd, fs, ft	F[fd] = F[fs] * F[ft]
Divide	div.x	div.x fd, fs, ft	F[fd] = F[fs] / F[ft]
Negation	neg.x	neg.x fs, ft	F[fs] = -F[ft]
Move	mov.x	mov.x fd, fs	F[fd] = F[fs]

Table 18: MIPS Floating-Point Instructions

Let's see an example:

```
.data
    val1: .float 5.5          # Single precision float value 1
    val2: .float 2.5          # Single precision float value 2
    result: .float 0.0        # Space to store result of the operation
    int_result: .word 0       # Space to store an integer result
    label_true: .asciiz "Comparison was true.\n"
    label_false: .asciiz "Comparison was false.\n"

.text

main:
    # Load the address of 'val1' into $t0
    la $t0, val1

    # Load the single-precision float (5.5) from memory into $f4
    lwc1 $f4, 0($t0)

    # Load the address of 'val2' into $t0
    la $t0, val2

    # Load the single-precision float (2.5) from memory into $f6
    lwc1 $f6, 0($t0)

    # Perform floating-point addition: $f8 = $f4 + $f6
    add.s $f8, $f4, $f6

    # Store the result of the addition into saved register for persistence
    mov.s $f20, $f8          # Move result into saved register $f20
```

```
29
30      # Store the result back into memory
31      swc1 $f20, result        # Store £f20 (saved register) to 'result' in memory
32
33
34      # Compare two floating-point values (temporary registers)
35      c.lt.s $f4, $f6          # Set condition flag if £f4 < £f6
36
37      # Branch based on comparison result
38      bc1t true_label          # If £f4 < £f6, jump to true_label
39      bc1f false_label         # If £f4 >= £f6, jump to false_label
40
41  true_label:
42      # Load string and print "Comparison was true."
43      la $a0, label_true       # Load address of label_true
44      li $v0, 4                # Syscall for printing string
45      syscall
46
47      # Convert a floating-point number to an integer
48      cvt.w.s $f24, $f4        # Convert single-precision float in £f4 to integer in £f24
49      mfc1 $t0, $f24           # Move converted integer from £f24 to £t0
50
51      # Store the converted integer into memory
52      sw $t0, int_result       # Store £t0 (the integer result) to 'int_result' in memory
53
54      j end                    # Jump to the end of the program
55
56  false_label:
57      # Load string and print "Comparison was false."
58      la $a0, label_false      # Load address of label_false
59      li $v0, 4                # Syscall for printing string
60      syscall
61
62  end:
63      # Exit the program
64      li $v0, 10               # Syscall for program exit
65      syscall
66
```

Branching with floating-point values differs slightly from normal branching. Instead of using a single instruction to both test and jump, you must first perform the test and then use a second instruction to jump based on whether the test result was true or false. The test modifies a special control or flag register (or specific bits within a register), and the jump instructions rely on the status of this register.

In the MARS register file, you will find, for example, $f4 holds the value `0x40b00000`, if you remember the float number representation, you will figure out it is 5.5 in decimal format. Below is a review of this computation:

The process of converting the 32-bit hexadecimal value `0x40b00000` to its decimal form involves using the IEEE 754 standard for single-precision floating-point numbers. A 32-bit floating-point number consists of three parts:

- **1 bit for the sign**
- **8 bits for the exponent**
- **23 bits for the mantissa (fractional part)**

The hexadecimal value `0x40b00000` in binary is:

$$0x40b00000 = 0100\ 0000\ 1011\ 0000\ 0000\ 0000\ 0000\ 0000$$

- **Sign bit (1 bit)**: The first bit is `0`, which indicates that the number is positive.

- **Exponent (8 bits)**: The next 8 bits are `10000001`, which represents the exponent in biased form.

$$\begin{aligned}\text{Binary exponent} &= 10000001\\ \text{Decimal exponent} &= 129\\ \text{Actual exponent} &= 129 - 127 = 2\end{aligned}$$

- **Mantissa (23 bits)**: The remaining 23 bits are `011 0000 0000 0000 0000 0000`, which represents the mantissa. The mantissa has an implicit leading 1, so the actual mantissa is:

$$1.011 \text{ (binary)}$$

The binary mantissa `1.011` is converted to decimal as follows:

$$1.011_2 = 1 + \frac{1}{4} + \frac{1}{8} = 1.625$$

The formula for a floating-point number is:

$$\text{value} = (-1)^{\text{sign}} \times 2^{\text{exponent}} \times \text{mantissa}$$

Substituting the known values:

$$\begin{aligned}\text{sign} &= 0 \text{ (positive)}\\ \text{exponent} &= 2\\ \text{mantissa} &= 1.375\end{aligned}$$

We can now calculate the final value:

$$\text{value} = 1 \times 2^2 \times 1.375 = 4 \times 1.375 = 5.5$$

Therefore, the floating-point number represented by `0x40b00000` is 5.5 in decimal.
The function also needs to be careful when you are dealing with float points. After moving an integer value into a floating-point register using mtc1(like when we are calculating fractions), you need to use `cvt.s.w` to convert it into a single-precision floating-point number. This is because `mtc1` simply moves the bits from the integer register into the floating-point register without interpreting them as a floating-point number. Without the conversion, the floating-point unit (FPU) would treat the bits as though they are already in floating-point format, which could lead to incorrect results. For example integer 32 is `0x20`, but float point 32.0 is `0x42000000`.

Let's write a program to convert a Fahrenheit temperature to Celsius: $C = \frac{5}{9} \cdot (F - 32)$

```
.data
    prompt_f: .asciiz "Enter temperature in Fahrenheit: "
    result_msg: .asciiz "Temperature in Celsius: "
    newline: .asciiz "\n"

.text

main:
    # Prompt for Fahrenheit input
    li $v0, 4               # Syscall for printing string
    la $a0, prompt_f        # Load address of prompt
    syscall

    # Read Fahrenheit input (as an integer)
    li $v0, 5               # Syscall for reading integer
    syscall
    move $t0, $v0           # Move Fahrenheit input into $t0

    # Convert integer Fahrenheit to floating point (using $f12)
    mtc1 $t0, $f12          # Move the integer Fahrenheit value to $f12
    cvt.s.w $f12, $f12      # Convert integer to floating-point (single precision)

    # Call the fahrenheit_to_celsius function
    jal fahrenheit_to_celsius
```

```
25
26      # Print the result
27      li $v0, 4               # Syscall for printing string
28      la $a0, result_msg       # Load address of result message
29      syscall
30
31      # Print the Celsius result
32      li $v0, 2               # Syscall to print float
33      mov.s $f12, $f0         # Move result from £f0 to £f12 for printing
34      syscall
35
36      # Exit the program
37      li $v0, 10              # Syscall for program exit
38      syscall
39
40  # Function to convert Fahrenheit to Celsius
41  # Takes Fahrenheit in £f12, returns Celsius in £f0
42  fahrenheit_to_celsius:
43      li $t0, 32              # Load 32 into £t0 (to subtract from Fahrenheit)
44      mtc1 $t0, $f6           # Move 32 into £f6 (convert integer to float)
45      cvt.s.w $f6, $f6        # Convert £f6 (integer 32) to floating-point
46      sub.s $f4, $f12, $f6    # Subtract 32 from Fahrenheit (£f12 - £f6)
47
48      li $t1, 5               # Load 5 into £t1
49      mtc1 $t1, $f8           # Move 5 into £f8 (convert integer to float)
50      cvt.s.w $f8, $f8        # Convert £f8 (integer 5) to floating-point
51      mul.s $f4, $f4, $f8     # Multiply by 5: £f4 = (£f4 * 5)
52
53      li $t2, 9               # Load 9 into £t2
54      mtc1 $t2, $f10          # Move 9 into £f10 (convert integer to float)
55      cvt.s.w $f10, $f10      # Convert £f10 (integer 9) to floating-point
56      div.s $f0, $f4, $f10    # Divide by 9: £f0 = (£f4 / £f10) (final Celsius value)
57
58      jr $ra                  # Return to the caller
```

One more thing that needs your attention is that you need to move your value to $f12 (arguments) before you call the system call to print the float number.

18.17 Special Topic: Integer to String & String to Integer

In C, we can easily achieve this by using atoi() to convert a string to an integer and sprintf() to convert an integer to a string. However, instead of relying on these functions, we will manually implement both conversions, which will provide us with a deeper understanding of how to write this process in assembly code.

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int string_to_integer(const char *str) {
5      int result = 0;
6      int sign = 1;
7      int i = 0;
8
9      // Handle negative numbers
10     if (str[0] == '-') {
11         sign = -1;
12         i++;  // Start conversion after the negative sign
13     }
14
15     // Loop through each character of the string
16     for (; str[i] != '\0'; i++) {
17         result = result * 10 + (str[i] - '0');  // Convert character to number and accumulate
18     }
19
20     return sign * result;  // Return the result with the appropriate sign
21 }
22
23 void integer_to_string(int num, char *buffer) {
24     int i = 0;
```

```c
    int is_negative = 0;

    // Handle negative numbers
    if (num < 0) {
        is_negative = 1;
        num = -num;  // Work with positive version of the number
    }

    // Convert each digit into the buffer (in reverse order)
    do {
        buffer[i++] = (num % 10) + '0';  // Get the last digit and convert it to a character
        num /= 10;                       // Remove the last digit
    } while (num > 0);

    // If the number was negative, append the minus sign
    if (is_negative) {
        buffer[i++] = '-';
    }

    // Null-terminate the string
    buffer[i] = '\0';

    // Reverse the buffer to get the correct order
    int start = 0;
    int end = i - 1;
    while (start < end) {
        // Swap the characters
        char temp = buffer[start];
        buffer[start] = buffer[end];
        buffer[end] = temp;
        start++;
        end--;
    }
}

int main() {
    // Example: String to Integer
    const char *str = "-1234";  // Input string
    int num = string_to_integer(str);  // Manually convert string to integer
    printf("String to Integer: %d\n", num);  // Output: -1234

    // Example: Integer to String
    int number = -5678;  // Input integer
    char buffer[20];     // Buffer to hold the resulting string
    integer_to_string(number, buffer);  // Manually convert integer to string
    printf("Integer to String: %s\n", buffer);  // Output: -5678

    return 0;
}
```

We will follow the similar idea to write our assembly code:

```assembly
.data
    str_input:    .asciiz "-1234"    # Input string for string to integer conversion
    buffer:       .space 20          # Buffer for integer to string conversion
    newline:      .asciiz "\n"       # Newline for output formatting

.text
main:
    # String to Integer conversion
    la $a0, str_input        # Load address of input string
    jal string_to_integer    # Call string_to_integer
    move $a0, $v0            # Move the result to $t0 for later use

    # Print the integer result
    li $v0, 1                # Syscall to print integer
    syscall

    # Print newline
    li $v0, 4
    la $a0, newline
    syscall

```

```
    # Integer to String conversion
    li $a0, 5678            # Load integer to convert (example: 5678)
    la $a1, buffer          # Address of buffer to store string
    jal integer_to_string   # Call integer_to_string

    # Print the string result (it will be stored in the buffer)
    li $v0, 4               # Syscall to print string
    la $a0, buffer          # Load address of the buffer (result)
    syscall

    # Exit the program
    li $v0, 10
    syscall

# Function: string_to_integer
# Converts a string of digits into an integer
# $a0: Address of the input string
# $v0: The integer result
string_to_integer:
    li $v0, 0               # Initialize result to 0
    li $t2, 1               # Initialize sign as positive (1)

    lb $t0, 0($a0)          # Load first character of the string into $t0
    beq $t0, $zero, end_str_to_int # If the first character is null, return 0
    li $t1, '-'             # Load ASCII value of '-'
    beq $t0, $t1, negative  # If the first character is '-', go to negative

str_to_int_loop:
    lb $t0, 0($a0)          # Load the current character
    beq $t0, $zero, end_str_to_int  # If we reach the null terminator, end the loop

    subi $t0, $t0, 48       # Convert ASCII character to integer value by subtracting '0'
    mul $v0, $v0, 10        # Multiply current result by 10
    add $v0, $v0, $t0       # Add the current digit

    addi $a0, $a0, 1        # Move to the next character in the string
    j str_to_int_loop       # Repeat the loop

negative:
    li $t2, -1              # Set sign to negative (-1)
    addi $a0, $a0, 1        # Skip the '-' character
    j str_to_int_loop       # Go back to conversion

end_str_to_int:
    mul $v0, $v0, $t2       # Apply the sign
    jr $ra                  # Return with the result in $v0

# Function: integer_to_string
# Converts an integer into a string of digits (handles negative numbers)
# $a0: The integer to convert
# $a1: Address of the buffer to store the result
integer_to_string:
    move $t0, $a0           # Move the integer into $t0
    la $t1, buffer          # Pointer to the start of buffer
    li $t3, 0               # Index for buffer

    # Handle negative numbers
    bltz $t0, make_negative

int_to_str_loop:
    div $t0, $t0, 10        # Divide integer by 10
    mfhi $t4                # Get remainder (digit)

```

```
    addi $t4, $t4, 48        # Convert digit to ASCII (add '0')
    sb $t4, 0($t1)           # Store the digit in the buffer
    addi $t1, $t1, 1         # Increment buffer index

    mflo $t0                 # Update the quotient in £t0

    bnez $t0, int_to_str_loop # Repeat if quotient is not 0

    sb $zero, 0($t1)          # Null-terminate the string
    j reverse_string          # Reverse the string to correct order

make_negative:
    neg $t0, $t0             # Convert number to positive
    li $t4, 45               # ASCII of '-'
    sb $t4, 0($t1)           # Store '-' sign in buffer
    addi $t1, $t1, 1         # Move buffer pointer forward
    j int_to_str_loop        # Continue with positive number

reverse_string:
    la $t0, buffer           # Pointer to the start of buffer
    sub $t1, $t1, 1          # Adjust index to last character in the buffer

    # Reverse the string by swapping characters
reverse_loop:
    bge $t0, $t1, end_reverse # Stop when indices meet or cross

    lb $t4, 0($t0)           # Load character from the front of the buffer
    lb $t5, 0($t1)           # Load character from the back of the buffer

    sb $t4, 0($t1)           # Store front character in the back
    sb $t5, 0($t0)           # Store back character in the front

    addi $t0, $t0, 1         # Move forward
    subi $t1, $t1, 1         # Move backward
    j reverse_loop           # Repeat

end_reverse:
    jr $ra                   # Return from the function
```

18.18 Macros

In MIPS assembly programming, two useful directives to make the code more readable and maintainable are `.eqv` and macros. Both serve different purposes but aim to improve clarity and reduce redundancy in code.

The `.eqv` directive in MIPS assembly defines a symbolic constant. It allows you to assign a value to a name, making the code more readable and easier to maintain. The assembler replaces all occurrences of the symbolic name with the value at assembly time.

Below is a simple example where `.eqv` is used to define a constant for a system call to print a string. The symbolic name `sys_print_str` is assigned the value `4`, which is the system call code for printing a string.

```
    .eqv sys_print_str 4   # Define sys_print_str as 4

    .data
    message: .asciiz "Hello, MIPS!\n"

    .text
    .globl main

    main:
        li $v0, sys_print_str  # Load syscall code for printing a string
        la $a0, message        # Load address of the message string
```

```
        syscall                 # Execute the syscall

        li $v0, 10              # Syscall to exit the program
        syscall
```

In C, a similar feature is provided using the `#define` directive, which allows you to define symbolic constants. Below is the corresponding C code for the same task:

```c
#define SYS_PRINT_STR 4
```

In MIPS assembly, macros allow you to create reusable blocks of code. They can take arguments and are expanded inline wherever they are invoked, reducing redundancy in code. Unlike `.eqv`, macros can contain multiple instructions and perform more complex tasks.

The following example defines a macro to load a string and print it using a system call. Macros can have parameters, making them versatile.

```
.macro PRINT_STR(%str)
    li $v0, 4            # System call to print a string
    la $a0, %str          # Load the address of the string
    syscall
.end_macro

.data
    message1: .asciiz "Hello, World!\n"
    message2: .asciiz "Welcome to MIPS!\n"

.text

    main:
        PRINT_STR message1    # Call the macro with message1
        PRINT_STR message2    # Call the macro with message2

        li $v0, 10            # Syscall to exit
        syscall
```

Here, the `PRINT_STR` macro takes one argument, `str`, which is the label for the string to be printed. The macro simplifies the task of printing strings, making the code more concise.

In C, the `#define` directive can be used to define macros. Below is the equivalent C program that uses macros to print two different messages.

```c
#include <stdio.h>

#define PRINT_STR(msg) printf("%s", msg)

int main() {
    PRINT_STR("Hello, World!\n");
    PRINT_STR("Welcome to MIPS!\n");

    return 0;
}
```

In this C example, the `#define PRINT_STR(msg)` macro simplifies the call to `printf` for printing a message, just like the MIPS macro.

18.18.1 Macros VS Functions

In programming, functions and macros serve to reduce redundancy and make code reusable, but they operate in fundamentally different ways. Here's a breakdown of the differences:

- Purpose and Usage
  - Functions: Encapsulate a block of code that performs a specific task, allowing it to be called multiple times with different arguments. Functions are commonly used for logical operations, computations, or tasks that need to be repeated with different values.

  - Macros: Allow for inline text substitution at compile or assembly time, effectively inserting the macro's code wherever it is called. Macros are often used to simplify repetitive tasks, create constants, or define small reusable code blocks.

- Execution and Compilation
  - Functions: The code for a function is executed at runtime. When a function is called, program execution jumps to the function's code, executes it, and then returns to the calling code. Function calls typically involve overhead for pushing arguments onto the stack, jumping to the function, and returning.
  - Macros: The code in a macro is expanded directly in place at compile time (in languages like C) or assembly time (in assembly languages). Since macros are expanded inline, they have no runtime overhead, but they can increase the compiled code size if used extensively.

- Argument Handling
  - Functions: Use parameter passing (by value, reference, or pointer) to handle arguments. This allows functions to handle expressions, ensure data type correctness, and perform type checking.
  - Macros: Perform a straightforward text substitution, without evaluating expressions or enforcing type checking. Macro arguments are directly substituted as they are written, so they lack the data validation that functions provide, which can lead to unintended behavior if expressions are used.

- Scope and Lifetime
  - Functions: Have their own scope and can declare local variables. Variables declared inside a function exist only within the function, providing controlled access and minimizing errors due to variable scope.
  - Macros: Have no scope in the traditional sense. Since they are text substitutions, macros do not create a separate variable scope, which can lead to naming conflicts if the same variable names are used outside the macro.

- Debugging and Maintenance
  - Functions: Generally easier to debug and maintain because they produce compact machine code and have a clear call structure. Modern debuggers can step through functions and inspect variables, which helps with identifying issues.
  - Macros: Can make debugging harder, as the code they generate is inserted inline. The expanded code from macros can lead to confusing error messages or unexpected behaviors, especially if macro parameters contain complex expressions.

- Use Cases
  - Functions: Ideal for tasks requiring modularity, encapsulation, and frequent reuse. They're best suited for tasks that involve calculations, data processing, or any task where execution flow control (like recursion or conditional execution) is needed.
  - Macros: Useful for small, repetitive code patterns, constant definitions, or inline code expansion where efficiency is critical and runtime overhead from function calls is undesirable.