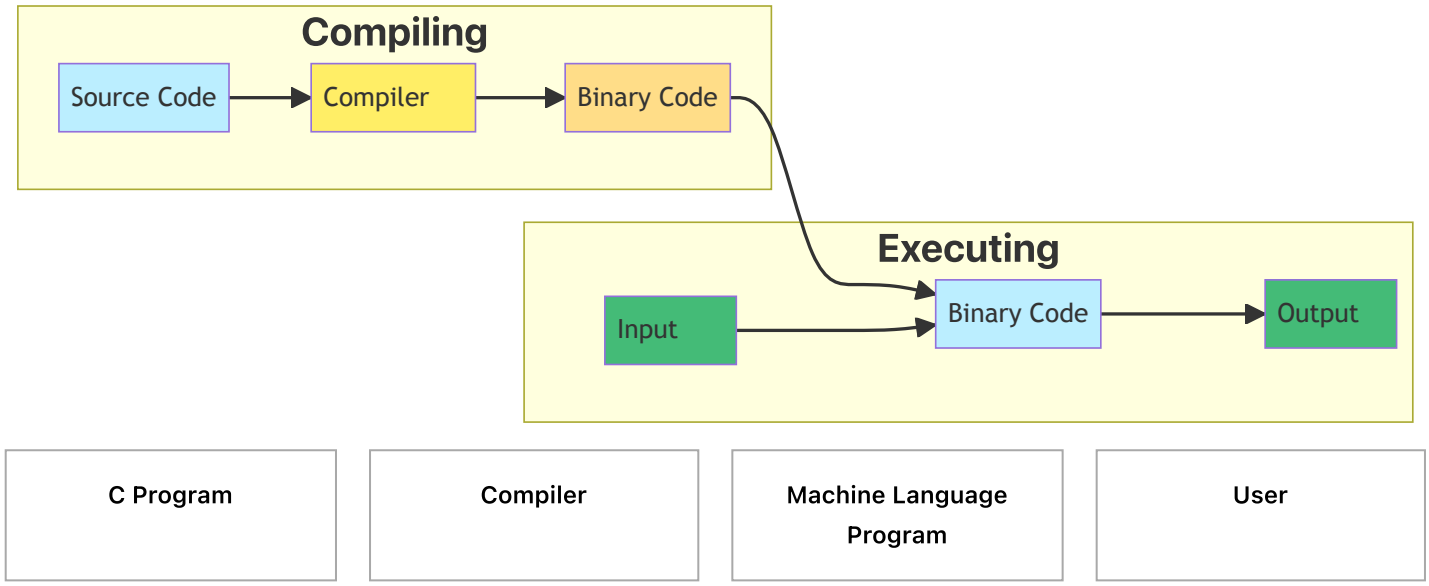
To run our program, we direct the operating system to load the binary code into RAM and start executing that code. The user of our program provides input while that code is executing and receives output from it.



Compilers can optimize the program instructions that we provide to improve execution times.

Examples

Let us write a program that displays the phrase "This is C" and name our source file `hello.c`. Source files written in the C language end with the extension `.c`.

Copy and paste the following statements into a txt editor of your choice:

```
/* My first program          // comments introducing the source file
   hello.c          */

 #include <stdio.h>          // information about the printf identifier

 int main(void)              // the starting point of the program
 {
    printf("This is C"); // send output to the screen

    return 0;            // return control to the operating system
 }
```

Each line is explained in more detail in the following section.

Linux

The C compiler that ships with Linux operating systems is called gcc.

To create a binary code version of our source code, enter the following command:

```
gcc hello.c
```

By default, the gcc compiler produces an output file named `a.out`. `a.out` contains all of the machine language instructions needed to execute the program.

To execute these machine language instructions, enter the command:

```
a.out
```

The output of the executed binary will display:

```
This is C
```

Windows

The C compiler that runs on Windows platforms is called cl. To access this compiler, open a Developer command prompt for Visual Studio window. To create a binary code version of our source code, enter the following command:

```
cl hello.c
```

By default, the cl compiler produces a file named `hello.exe`. `hello.exe` contains all of the machine language instructions needed to execute the program.

To execute these machine language instructions, enter the command:

```
hello
```

The output of the executed binary will display:

```
This is C
```

Basic C Syntax

The source code listed above includes syntax found in the source code for nearly every C program.

Documentation

The comments self-document our source code and enhance its readability. Comments are important in the writing of any program. C supports two styles: multi-line and inline. C compilers ignore all comments.

Multi-Line Comments

```
/* My first program
    hello.c             */
```

`/*` and `*/` delimit comments that may extend over several lines. Within these delimiters, we may include any character, except the `/*` and `*/` pair.

Inline Comments

```
int main(void)                  // the starting point of the program
```

`//` indicates that the following characters until the end of the line.

Whitespace

Whitespace enhances program readability, for instance, by displaying the structure of a program. C compilers ignore whitespace altogether. Whitespace refers to any of the following:

- blank space
- newline
- horizontal tab
- vertical tab
- form feed
- comments

We may introduce whitespace anywhere except within an identifier or a pair of double-quotes. In the sample above, `main` and `printf` are identifiers. The blank spaces within "This is C" are not whitespace but characters within the literal string.

We preface our source code with comments to identify the program and provide distinguishing information.

Indentation

By indenting the **`printf("This is C")`** statement and placing it on a separate line, we show that **`printf("This is C")`** is part of something larger, which we have called **`int main(void)`**

Program Startup

Every C program includes a clause like `int main(void)`. Program execution starts at this line. We call it the program's entry point. The pair of braces that follow this clause encompasses the program instructions.

```
int main(void)           // program startup
{
        return 0;         // return to operating system
}
```

When the users or we load the executable code into RAM (`a.out` or `hello.exe`), the operating system transfers control to this entry point. The last statement (`return 0;`) before the closing brace transfers control back to the operating system.

Program Output

The following statement outputs "This is C" to the standard output device (for example, the screen).

```
printf("This is C");
```

The line before **`int main(void)`** includes information that tells the compiler that **`printf`** is a valid identifier.

```
#include <stdio.h>          // information about the printf identifier
```

Case Sensitivity

The C programming language is case sensitive. That is, the compiler treats the character 'A' as different from the character 'a'. If we change the identifier **`printf()`** to **`PRINTF()`** and recompile, the compiler will report a syntax error. However, if we change `"This is C"` to `"THIS IS C"` and recompile the source code, the compiler will accept the change and not report any error.

Computers

Learning Outcomes

After reading this section, you will be able to:

- Describe the major components of a modern computer
- Describe the software that controls a modern computer
- Outline the contents of these notes

Introduction

Computers are available in many flavours: mobile devices, smart phones, laptops, tablets, desktops, workstations and servers to name a few. All of these devices control their operations through software. Programmers create this software. Users rely on this software to operate their devices.

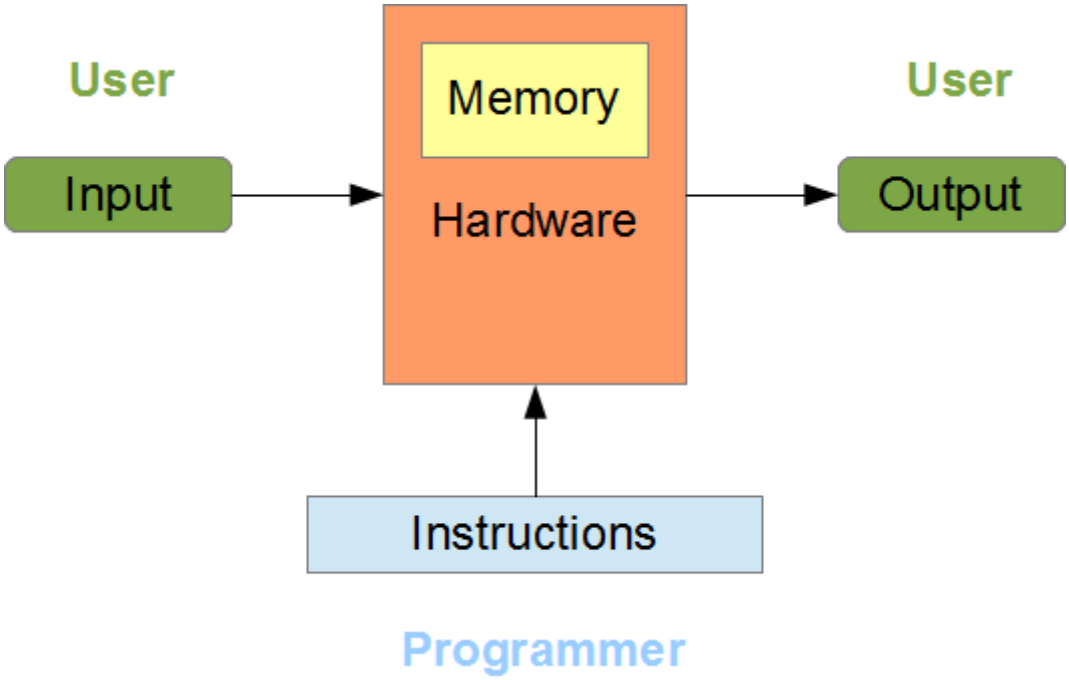
We refer to the software with which a user operates their device as application software. Application software consists of one or more programs. Each program is a full set of instructions that performs a well-defined task on the host device. Programmers code these instructions in a programming language.

The instructions that programmers code represent algorithms. An algorithm is a step-by-step procedure that describes how to achieve a specified task. Examples include searching, sorting and mixing. Application programmers study different algorithms and create their own if required. They find the code for some algorithms in libraries and write the code for their own algorithms.

This introductory chapter describes the major components of a modern computer, components to which programmers often refer. Subsequent chapters show how to write programs to use the principal features of these components efficiently.

Hardware

The figure below illustrates the relation of the programmer to a user of a software application. The boxes identify the principal elements for any programmer. The green boxes identify the elements of concern to the user.

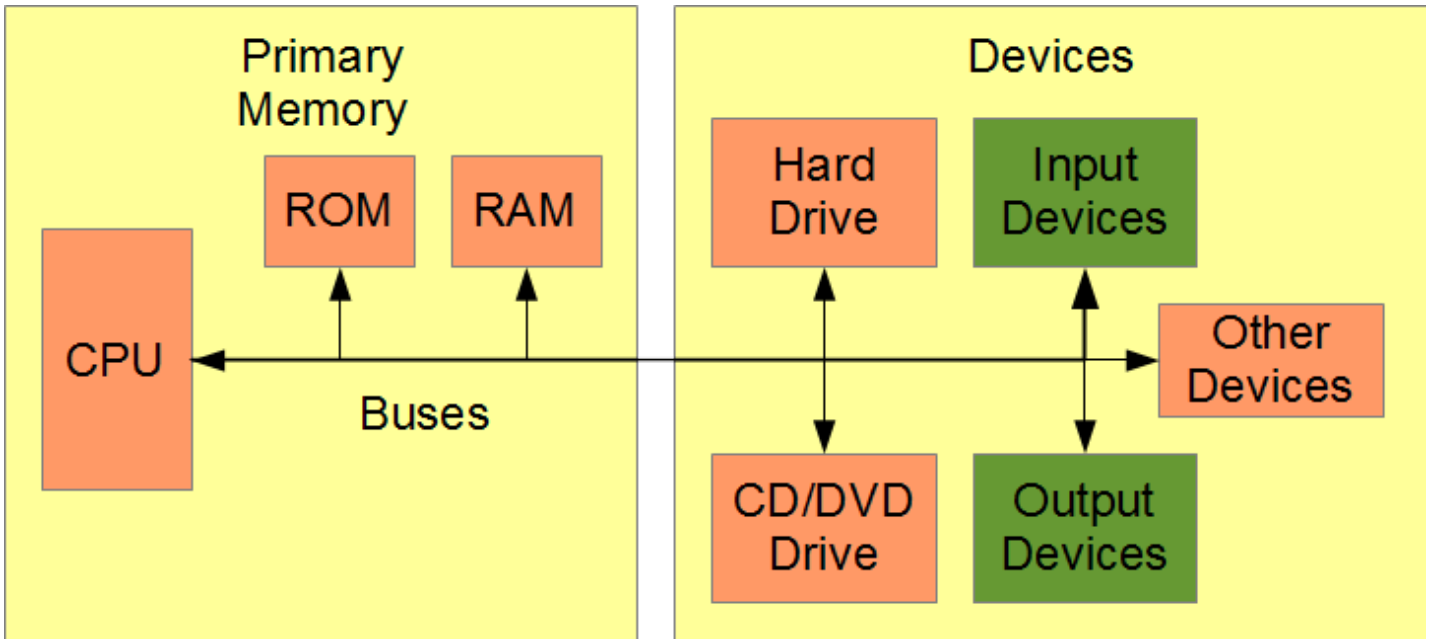


Computer hardware stores the program instructions in its own memory, accepts input from the user, processes that input according to the stored instructions, and generates the output for the user. The user can rerun the program to process different inputs and produce corresponding outputs.

Modern Computers

In 1945, John von Neumann, noting that instructions are pieces of information just like data, proposed a new computer architecture, in which instructions and data are stored alongside one another. We call this idea the stored-program concept. All modern computers are stored-program computers.

The figure below shows the components of a stored-program computer. They include a central processing unit (CPU), a clock, primary memory and a set of devices. Buses interconnect these components and are part of the motherboard. The CPU, primary memory and a clock are also part of the motherboard. The clock controls the rate at which the CPU executes the instructions.



Primary Memory

Primary memory is memory directly accessible by the CPU. Primary memory includes read-only memory (ROM), random-access memory (RAM) and memory within the CPU itself.

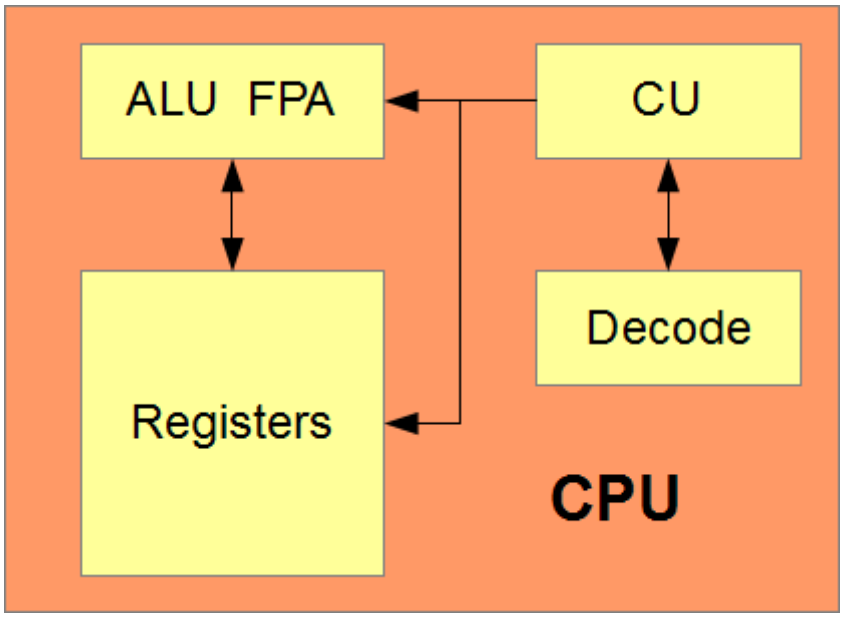
ROM holds the instructions for starting the system. ROM is not volatile: it persists if we turn off power.

RAM holds the program instructions and the program data. RAM is volatile: its contents are lost if we turn off power.

Central Processing Unit

The CPU is the work-horse of any modern computer. The CPU executes program instructions serially (one at a time). A modern CPU consists of:

- registers
- a decode unit
- a control unit (CU)
- an arithmetic and logic unit (ALU)
- a floating-point accelerator (FPA)



Registers are the CPU's internal memory. They hold the data used by the ALU and FPA and any new data that the ALU and FPA produce. Register data is volatile: we lose the contents of each register as soon as we turn off power.

The Decode unit extracts each incoming instruction from the instruction queue and decodes that instruction. The CU moves data between registers, RAM and the devices, and passes the decoded instruction to the ALU or the FPA for processing. The CU manages the data, but does not change it.

The ALU performs the comparisons and integer calculations, changes the data and creates new data as directed by the CU. The FPA performs the calculations on floating-point data. The ALU and FPA work solely with register memory inside the CPU.

Devices

The devices of a modern computer include peripheral and other devices. Peripheral devices include a keyboard, a mouse, and a monitor, which provide user interfaces for input and output. Hard drives, USB keys and DVD/CD-ROMs constitute secondary memory, which provides persistent storage of program instructions and program data.

Memory Comparison

Secondary memory is inexpensive compared to primary memory, but considerably slower. Compare the following data transfer rates:

- Registers ~10 nanoseconds
- ROM and RAM ~60 nanoseconds
- Hard disk ~12,000,000 nanoseconds

A nanosecond is 1e-9 seconds. To appreciate the differences, consider the following analogy. The ratio of the time that the CPU takes to transfer data between registers to the time that a hard disk takes to transfer that same information is the ratio of the width of an average-sized room to the distance once around the earth along the equator.

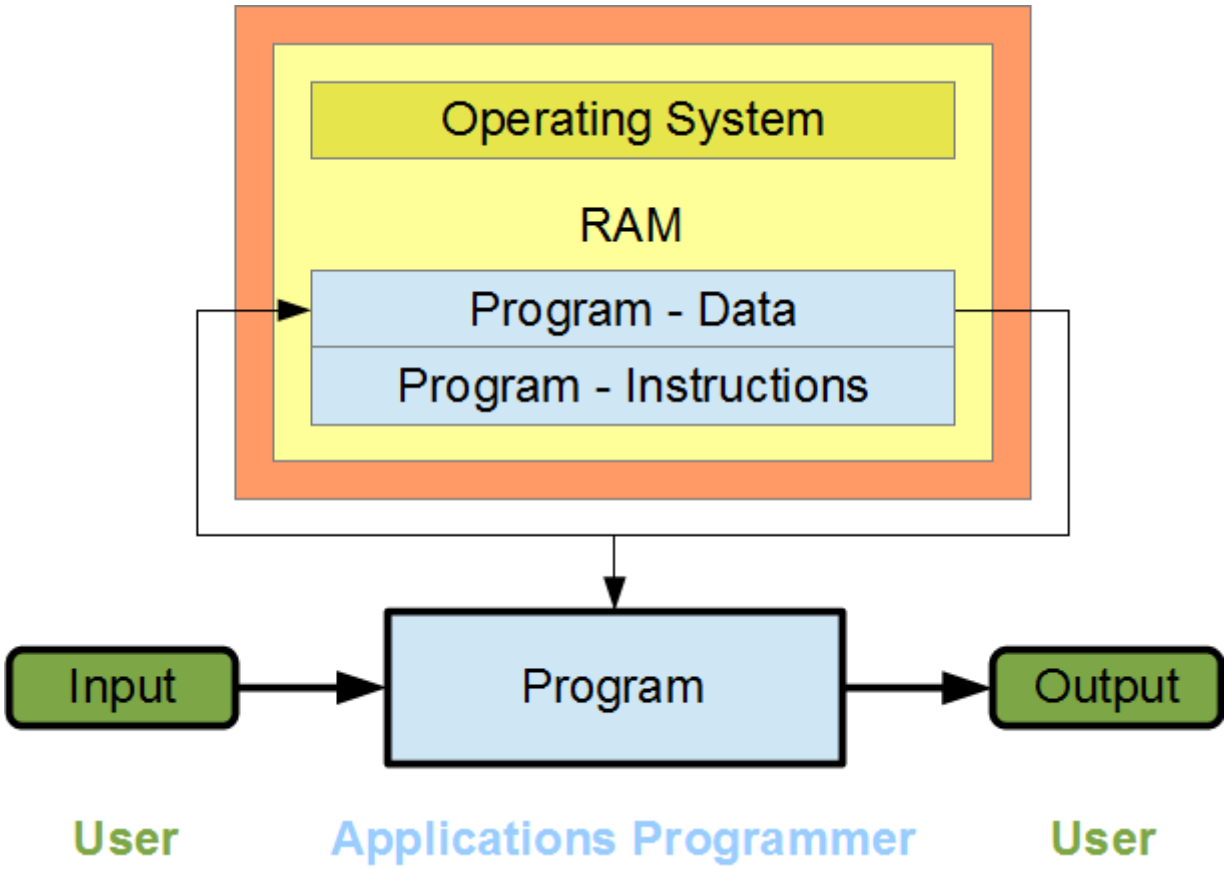
Software

The software that controls a modern computer includes the programs that are currently executing and the operating system that manages them. The operating system is a program that executes as long as power is on. The operating system resides in RAM along with the other currently executing programs.

When a user starts an application program, the operating system loads that program's instructions into part of the RAM and transfers control to the program. The program starts executing, requests data from the user, sends output to the user, and eventually terminates its own execution and returns control to the operating system.

An application program transforms raw data from the user (the input) into equivalent data stored in RAM, operates on the data in RAM and transforms the resultant data into some user-readable form (the output). Programming languages define how internal data is stored in RAM.

As application developers, we focus on input and output processing and transformation of input data into output data. The figure below relates this focus to the software that we code.



Outline

These notes introduce the fundamental concepts of software development. The chapters are grouped into six parts:

1. Introduction
2. Computations
3. Data Structures
4. Modularity
5. Secondary Storage
6. Refinements

The introductory part describes modern computers, the storage of information in memory, and how to create your first program.

The computations part introduces the concept of type, shows how to describe program variables using types, shows how to handle basic input and output, how to calculate a new value from existing values, how to create optional paths through a program, how to write code in a friendly

readable style, and how to test and debug sets of program instructions, including how to determine the output produced by those instructions.

The data structures part describes how to organize groups of values into data types in memory. This part introduces the grouping concepts of arrays and structures.

The modules part describes how to organize program instructions into self-contained cohesive units, called functions, where each function implements a single algorithm. This part shows how to divide a program into independent modules and how to pass information from one module to another.

The secondary storage part describes how to move text data between an installed device and RAM. This part introduces files and describes the syntax for working with secondary memory.

The refinements part elaborates on concepts introduced in the other parts. It covers character strings as specialized versions of arrays and shows how to work with the library functions that handle them. This part describes the relation between pointers and arrays, including arrays of structures. This part also covers two-dimensional arrays and shows how to store tabular data using strings. This part concludes with introductions to some of the standard algorithms and the guidelines for portability of program code.

Optional Sections

Some chapters include sections or sub-sections marked optional. These sections contain information that elaborates specific details related to the topic covered in these notes. Feel free to skip these sections on first reading, without disrupting presentation flow. Subsequent mandatory sections do not rely on any information covered in these optional sections. These optional sections simply add depth to the material.

Expressions

Learning Outcomes

After reading this section, you will be able to:

- Code various expressions that apply operations on operands of program type

Introduction

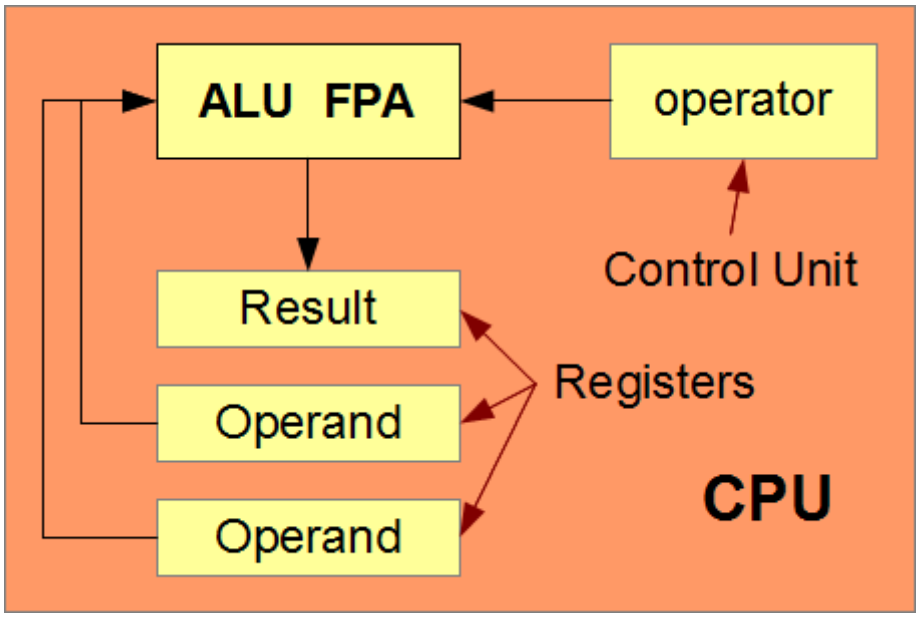
Programming languages support operators that combine variables and constants into expressions for transforming existing data into new data. These operators take one or more operands. The operands may be variables, constants or other expressions. The C language supports a comprehensive set of these operators for arithmetic, relational and logical expressions.

This chapter describes the supported operators in detail, what happens when the operands are of different types, how to change the type of an operand and the order of evaluation of sub-expressions within expressions. The introduction to this detailed description is a brief overview of the hardware components that evaluate expressions. These are the ALU and FPA inside the CPU.

Evaluating Expressions

The ALU evaluates the simplest of instructions on integer values: for instance, additions where the operands are of the same type. The FPA evaluates the simplest of instructions on floating-point values. C compilers simplify C language expressions into sets of hardware instructions that either the ALU or the FPA can process.

The ALU receives the expression's operator from the Control Unit, applies that operator to integer values stored in the CPU's registers and places the result in one of the CPU's registers. The FPA does the same but for floating-point values.



The expressions that the ALU can process on integer types are:

- arithmetic
- relational
- logical

The FPA can processes these same kinds of expressions on floating-point types.

Arithmetic Expressions

Arithmetic expressions consist of:

- integral operands - destined for processing by the **ALU**
- floating-point operands - destined for processing by the **FPA**

Integral Operands

The C language supports 5 binary and 2 unary arithmetic operations on integral (`int` and `char`) operands. Here, the term *binary* refers to two operands; *unary* refers to one operand.

Binary Operations

The ***binary*** arithmetic operations on integers are addition, subtraction, multiplication, division and remaindering. Expressions take one of the forms listed below:

Arithmetic Expression	Meaning
operand `+` operand	add the operands
operand `-` operand	subtract the right from the left operand
operand `*` operand	multiply the operands
operand `/` operand	divide the left by the right operand
operand `%` operand	remainder of the division of left by right

Division of one integer by another yields a whole number. If the division is not exact, the operation discards the remainder. The expression evaluates to the truncated integer result; that is, the whole number without any remainder. The expression with the modulus operator (%) evaluates to the remainder alone.

For example:

```
34 / 10    // evaluates to 3 (3 groups of 10 people)
34 % 10    // evaluates to 4 (4 person left without a group)
```

Unary Operations

The ***unary*** arithmetic operations are identity and negation. Expressions take one of the forms listed below:

Arithmetic Expression	Meaning
`+` operand	evaluates to the operand
`-` operand	changes the sign of the operand

The plus operator leaves the value unchanged and is present for language symmetry.

Floating-Point

Operands

The C language supports 4 binary and 2 unary arithmetic operations on the floating-point (`float` and `double`) operands.

Binary

The binary arithmetic operations on floating-point values are addition, subtraction, multiplication and division. Expressions take one of the forms listed below:

Arithmetic Expression	Meaning
operand `+` operand	add the operands
operand `-` operand	subtract the right from the left operand
operand `*` operand	multiply the operands
operand `/` operand	divide the left by the right operand

The division operator (/) evaluates to a floating-point result. There is no remainder operator for floating-point operands.

Unary

The unary operations are identity and negation. Expressions take the form listed below:

Arithmetic Expression	Meaning
`+` operand	evaluates to the operand
`-` operand	change the sign of the operand

The plus operator leaves the value unchanged and is present for language symmetry.

Limits (Optional)

The result of any operation is an expression of related type. Arithmetic operations can produce values that are outside the range of the expression's type.

Consider the following program, which multiplies two `int`'s and then two `double`'s

```c
// Limits on Arithmetic Expressions
// limits.c
#include <stdio.h>

int main(void)
{
        int i, j, ij;
        double x, y, xy;

        printf("Enter an integer : ");
        scanf("%d", &i);
        printf("Enter an integer : ");
        scanf("%d", &j);
        printf("Enter a floating-point number : ");
        scanf("%lf", &x);
        printf("Enter a floating-point number : ");
        scanf("%lf", &y);

        ij = i * j;
        xy = x * y;

        printf("%d * %d = %d\n", i, j, ij);
        printf("%le * %le = %le\n", x, y, xy);

        return 0;
}
```

Compile this program and execute it inputting different values. Try some very small numbers. Try some very large numbers. When does this program give incorrect results? When it does, explain why?

Relational Expressions

The C language supports 6 relational operations. A relational expression evaluates a condition. It compares two values and yields ***1*** if the condition is ***true*** and ***0*** if the condition is ***false***. The value

of a relational expression is of type `int`. Relational expressions take one of the forms listed below:

Relational Expression	Meaning
operand `==` operand	operands are equal
operand `>` operand	left is greater than the right
operand `>=` operand	left is greater than or equal to the right
operand `<` operand	left is less than the right
operand `<=` operand	left is less than or equal to the right
operand `!=` operand	left is not equal to the right

The operands may be integral types or floating-point types.

Example

The following program, accepts two `int`'s and outputs 1 if they are equal; 0 otherwise:

```
// Relational Expressions
 // relational.c
 #include <stdio.h>

 int main(void)
 {
        int i, j, k;

        printf("Enter an integer : ");
        scanf("%d", &i);
        printf("Enter an integer : ");
        scanf("%d", &j);

        k = i == j; // compare i to j and assign result to k

        printf("%d == %d yields %d\n", i, j, k);
```

```
        return 0;
}
```

The first conversion specifier in the format string of the last `printf()` corresponds to `i`, the second corresponds to `j` and the third corresponds to `k`.

Logical Expressions

The C language does not have reserved words for true or false. It interprets the value 0 as false and any other value as true. C supports 3 logical operators. Logical expressions yield 1 if the result is true and 0 if the result is false. The value of a logical expression is of type int. Logical expressions take one of the forms listed below:

Logical Expression	Meaning
operand `&&` operand	both operands are true
operand `||` operand	one of the operands is true
`!` operand	the operand is not true

The operands may be integral types or floating-point types.

Example

The following program, accepts three `int`'s and outputs ***1*** if the second is greater than or equal to the first and less than or equal to the third; ***0*** otherwise:

```
// Logical Expressions
// logical.c
#include <stdio.h>

int main(void)
{
        int i, j, k, m;
```

```
        printf("Enter an integer : ");
        scanf("%d", &i);
        printf("Enter an integer : ");
        scanf("%d", &j);
        printf("Enter an integer : ");
        scanf("%d", &k);

        m = j >= i && j <= k; // store the value of this expression in m

        printf("%d >= %d and %d <= %d yields %d\n", j, i, j, k, m);

        return 0;
}
```

The conversion specifiers in the last `printf()` correspond to the arguments in the same order (first to ***j***, second to ***i***, etc.).

deMorgan's Law

deMorgan's law is a handy rule for converting conditions in logical expressions. The law states that:

> **NOTE**
>
> The opposite of a compound condition is the compound condition with all sub-conditions reversed, all &&'s changed to ||'s and all ||'s to &&'s.

Consider the following definition of an adult:

```
adult = !child && !senior;
```

This definition is logically identical to:

```
adult = !(child || senior);
```

The parentheses direct the compiler to evaluate the enclosed expression first.