A BRIEF INTRODUCTION TO PROGRAMMING LANGUAGES

Programming Languages

Generations

There are well over 2500 programming languages and their number continues to increase. The different generations of programming languages include:

- **Machine languages:** These are the **native languages** that the CPU processes. Each manufacturer of a CPU provides the instruction set for its CPU.
- **Assembly languages:** These are **readable versions** of the native machine languages. Assembly languages simplify coding considerably. Each manufacturer provides the assembly language for its CPU.
- **Third-generation languages:** These languages are procedural: they identify the instructions or procedures involved in reaching a result. The instructions are **NOT tied to any particular machine**. Examples include **C, C++, and Java.**
- **Fourth-generation languages:** These languages describe what is to be done without specifying how it is to be done. These instructions are NOT tied to any particular machine. Examples include **SQL, Prolog, and Matlab.**
- **Fifth-generation languages:** We use these languages for **artificial intelligence, fuzzy sets, and neural networks.**

The **third, fourth, and fifth-generation languages are high-level languages**. They exhibit **no direct connection to any machine language**. Their instructions are **more human-like** and **less machine-like**. A program written in a high-level language is relatively **easy to read** and relatively **easy to port across different platforms.**

Features of C

- C is English-like.
- C is quite compact - has a small number of keywords
- C is the lowest in level of the high-level languages
- C can be faster and more powerful than other high-level languages
- C programs that need to be maintained are large in number
- C is used extensively in high-performance computing
- UNIX, Linux, and Windows operating systems are written in C and C++

- C programming language is case-sensitive.

The C Compiler

THE C PROGRAM COMPILATION TAKES PLACE IN THE FOLLOWING WAY

- The code written in the C language (also known as **source code** - this is the input or program instructions for the compiler) is converted into **machine-level equivalents (or Binary Code**, this is the output of the compiler) an **exe file** is generated by the compiler
- The OS loads the Binary Code (given by the compiler) into the RAM and starts executing the code based on the inputs given by the user and generates the output for it.

SUMMARY

C PROGRAM -------> COMPILER -------> MACHINE LANGUAGE PROGRAM -------> USER

C COMPILER FOR LINUX

gcc (GNU Compiler Collection)

- Command to create the Binary code version of our Source Code – **gcc hello.c** [assuming our source code is written in a file named hello.c]
- The executable/output file produced by the gcc compiler by default will be – **a.out** [This contains all the machine language instructions needed to execute the program]
- To produce an executable/output file with a name of your choice (other than the a.out), we have to replace **gcc hello.c** with **gcc hello.c -o hello**

C COMPILER FOR WINDOWS

cl (Clang)

- Command to create the Binary code version of our Source Code **– cl hello.c**
- The executable/output file produced by the gcc compiler by default will be – **hello.exe** [This contains all the machine language instructions needed to execute the program]
- To produce an executable/output file with a name of your choice (other than the a.out), we have to replace **cl hello.c** with **cl hello.c -o myprogram**

Compiler VS Interpreter

They perform similar tasks, which involve **translating human-readable code into machine-executable code**, but they do so in different ways.

- **Compiler**: Compilers **translate the entire source code into machine code** (or an intermediate code) **all at once**. This results in the **creation of an executable file** that can be **run independently** of the original source code.

Interpreter: Interpreters **translate the source code line by line** or **statement by statement**, **executing each line immediately after it's translated**. There is **no separate output file generated.**

Whitespace

C compilers **ignore whitespace altogether**. Whitespace refers to any of the following:

- Blank Space
- Newline
- Horizontal tab
- Vertical tab
- Form feed
- Comments

Boilerplate Code

```
#include <stdio.h>    // Pre-processor Directive – information about the printf identifier
int main(void)        // program execution starts with this line – Program's entry point
{
        printf("Hello World, Welcome to C programming!");
        return 0;      // return to operating system
}
```

What does this #include – does?

#include is simply a command to the compiler to **copy-and-paste**, it pastes all the contents of what we are including **before the compilation begins (pre-processor).**

Program Startup

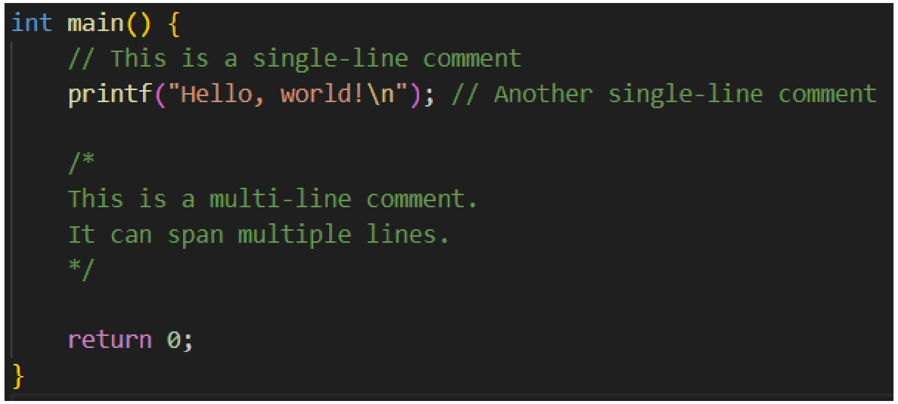
```
int main(void)        // program execution starts with this line – Program's entry point
{
        return 0;      // return to operating system
```

}

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 - int main(void)**. The last statement (**return 0;**) before the closing brace **transfers control back to the operating system.**

Comments

Comments in programming languages are **non-executable portions of code** used to **annotate** or **document the code** for humans reading it. They are **ignored by the compiler** or interpreter when the code is executed.
Comments in C and be of 2 types – **i. Single Line ii. Multi-Line**



```c
int main() {
    // This is a single-line comment
    printf("Hello, world!\n"); // Another single-line comment

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

    return 0;
}
```

Types

C is a **typed programming language.**

- A type is the **rule** that defines **how to store values in memory** and **which operations are admissible** on those values.
- A type defines the **number of bytes** available for storing values and hence the **range of possible values**.

There are **4 most common types** in C language and they can be divided into **2 Categories** as follows:

Integral Types

Floating Point Types

char
int

float
double

char – Occupies 1 byte [can store a **single character** or a **single symbol**]

int – Occupies 4 bytes

float – Typically occupies 4 bytes [can store a **single-precision**, floating-point number]
double – Typically occupies 8 bytes [can store a **double-precision,** floating-point number]

Units (bits/bytes)

Bits - The **most fundamental unit** of a modern computer is the binary digit or bit. A bit is either **on or off. One (1) represents on, while zero (0) represents off.**
Bytes - The fundamental **addressable unit of RAM** is the byte **[1 byte = 8 bits]**

Size Specifiers

Size specifiers **adjust the size of the int and double types.**

int Type Specifiers

- **short int (or short) –** Contains at least 16 bits
- **long int (or long) –** Contains at least 32 bits
- **long long int (or long long) –** Contains at least 64 bits

double Type Specifiers

- **long double –** it ensures that it contains at least **as many bits as a double**. The C language does not require a long double to contain a minimum number of bits.

Simple Calculation

1 byte = 8 bits
Total number of values we can use for a bit (2) = 1 or 0
Total number of possible values in a byte = 2^^{8} (We can use either of the 2 values in a bit and there are 8 bits) = **256 values/options in one byte**

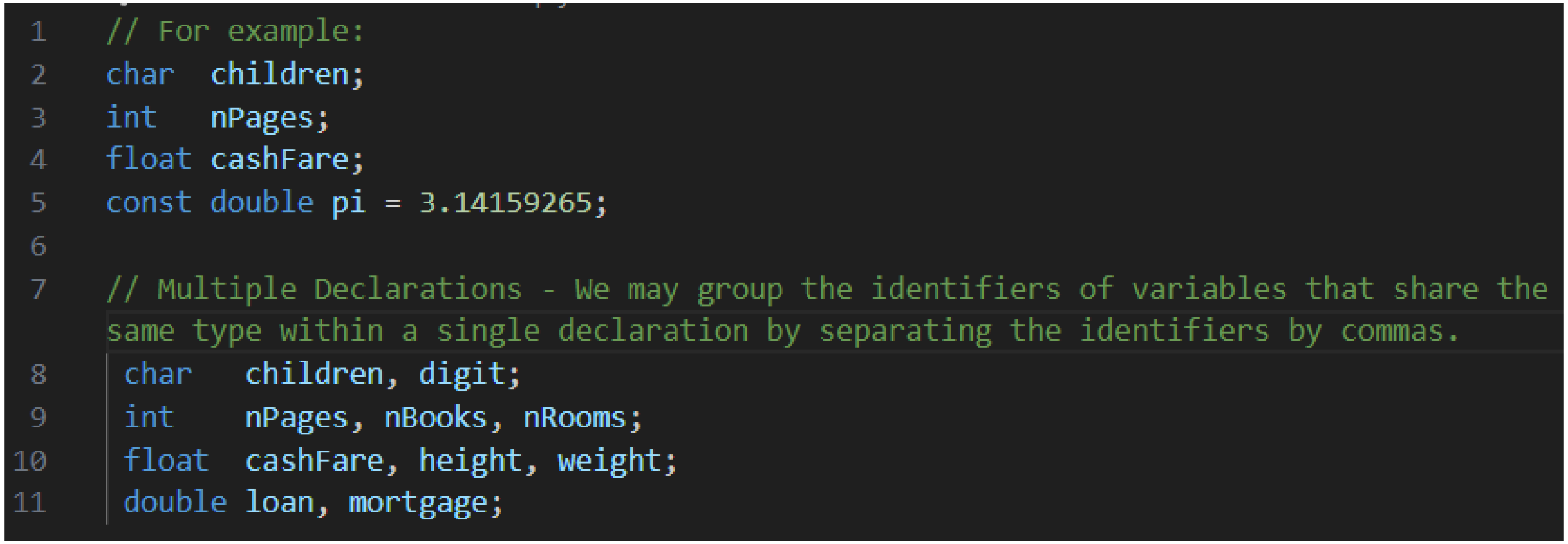
int has 4 bytes, Therefore, 256^^{4} (since there can be 256 options in 1 byte and we have a total of 4 bytes)
So the total number of possible options for int type = 4,294,967,296

Variable Declarations

In C, a declaration takes the form:

[const] type identifier [= initial value];

The brackets denote an optional part of the syntax. We select a **meaningful name for the identifier** and optionally set the variable's initial value. We conclude the declaration with a semi-colon, making it a complete statement.



```c
// For example:
char   children;
int    nPages;
float  cashFare;
const double pi = 3.14159265;

// Multiple Declarations - We may group the identifiers of variables that share the same type within a single declaration by separating the identifiers by commas.
char   children, digit;
int    nPages, nBooks, nRooms;
float  cashFare, height, weight;
double loan, mortgage;
```

Const Qualifier

Any type can hold a constant value. A constant value **cannot be changed**. To qualify a type as holding a constant value we use the keyword **const**. A type qualified as const is **unmodifiable**. That is, if a program instruction **attempts to modify** a const-qualified type, **the compiler will report an error.**

For example:

const num = 100; // num is un-modifiable

num = 10; // This is not possible

Naming Conventions

We may select any identifier for a variable that satisfies the following naming conventions:

- starts with a letter or an underscore (_)
- contains any combination of letters, digits and underscores (_)
- contains less than 32 characters (some compilers allow more, others do not)
- is not a **C-reserved word (**see below**)**

GOOD VARIABLE NAMING TECHNIQUES

Variable names (identifiers) should...

- be self-documenting (should not require comments to describe what they are used for)
- be concise but not so short that it is cryptic
- accurately describes the data being stored
- help with the reading of the code use "camelNotation" (first letter of each word is capitalized with the exception of the first word)
- not use underscore (_) characters in order to avoid conflicts with system libraries

Keywords – Reserved Words

Reserved words are predefined keywords in a programming language **that have special meanings and purposes**. They are **reserved by the language** and **cannot be used for naming variables, functions, or other identifiers**. These words typically serve **specific syntactical or operational roles within the language**.

The C language reserves the following words for its own use:

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

while

Operators VS Operands

Operators are **symbols or keywords** that **perform operations** on **one or more operands**. Operands are the **values or variables that operators act upon**.

In simpler terms:

- **Operators**: They do the actions (like addition, and subtraction).
- **Operands**: They are the values or variables the actions are done on (like numbers or variables).

Expressions

The expressions can be of the following types:

- Arithmetic
- Relational
- Logical

Arithmetic Expressions

Arithmetic Expressions consist of:

- Integral operands
- Floating point operands

Integral Operands Arithmetic Expressions

The C language supports **5 Binary** and **2 Unary arithmetic operations** on integral (int and char) operands.

5 Binary Operations

- Addition – **operand + operand**
- Subtraction – **operand – operand**
- Multiplication – **operand * operand**
- Division – **operand / operand**

- Remaindering – **operand % operand**

For the Integer division expression, a **whole number is the output.** If the division is not exact, the operation discards the remainder.
For Example:
4 / 2 = 2
4 / 3 = 1 (0.5 will get discarded)

The remaindering operation yields the remainder of the 2 number.
For Example:
4 / 2 = 0
5 / 3 = 2 (2 being the remainder)

2 Unary Operations

The unary arithmetic operations are **identity** and **negation**.

- **+ operand –** evaluates to the operand
- **- operand –** changes the sign of the operand

Floating-point Operands Arithmetic Expressions

The C language supports **4 Binary** and **2 Unary arithmetic operations** on floating-point (float and double) operands.

Every functionality of the Binary Expressions and Unary Expressions remains the same as Internal types except the remaindering binary expressions **(floating-point operands do not have remaindering expressions).**

Shorthand Assignments

The C language also supports shorthand operators that **combine an arithmetic expression with an assignment expression**. These operators store the result of the **arithmetic expression** in **the left operand.**

Like **Arithmetic Expressions,** C has **5 binary and 2 unary shorthand assignment operators** for **integral** (int and char) operands + C **has 4 binary (no remaindering) and 2 unary shorthand assignment operators** for **floating-point** (float and double) operands.

Integral Operands

5 Binary Operators

- **operand += operand**
- **operand -= operand**
- **operand *= operand**
- **operand /= operand**
- **operand %= operand**

2 Unary Operators

- **++operand (prefix) OR operand++ (postfix)**
- **--operand (prefix) OR operand -- (postfix)**

Floating-Point Operands – Everything remains the same for floating-point as well except that it does not have any modulus operator (%)

Relational Expressions

The C language supports **6 relational operations (for both integral and floating-point types)**.
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.**

Relational Expressions take one of the forms below:

- **operand == operand – returns true (1) if** operands are equal
- **operand > operand – returns true (1) if** left is greater than the right
- **operand >= operand – returns true (1) if** left is greater than or equal to the right
- **operand < operand – returns true if (1)** left is less than the right
- **operand <= operand – returns true if (1)** left is less than or equal to the right
- **operand != operand – returns true (1) if** left is not equal to the right

Logical Expressions

C supports **3 logical operators (for both integral and floating-point types)**. Logical expressions yield **1 if the result is true and 0 if the result is false.**

Logical Expressions take one of the forms below:

- **operand && operand – returns true (1) if** both operands evaluate to true

- **operand || operand – returns true (1) if** any of the operands evaluate to true
- **! operand – converts true to false and false to true.**

Casting

The C language **supports conversions from one type to another**. To convert the type of an operand, we **precede the operand with the target type enclosed within parentheses**. We call such an **expression a cast.**

Casting expressions take one of the forms listed below:

- **(long double)** operand
- **(double)** operand
- **(float)** operand
- **(long long)** operand
- **(long)** operand
- **(int)** operand
- **(short)** operand
- **(char)** operand

Mixed-Type Expressions

For expressions with **operands of different types**, we need **rules for converting operands of one type to another type**.

The C language uses the following ranking:

long double	**higher**
double	**...**
float	**...**
long long	**...**
long	**...**
int	**...**
short	**...**
char	**lower**

There are **two distinct kinds of expressions** to consider with respect to type coercion:

- assignment expressions
- arithmetic and relational expressions

ASSIGNMENT EXPRESSIONS

- **Promotion –** If the left operand in an assignment expression is of a higher type than the right operand, the compiler promotes the right operand to the type of the left operand.

  For Example:

  ```
  int num1 = 10;
  double num2;

  num2 = num1; // promotion (num1 gets promoted to type 'double')
  ```

- **Narrowing –** If the left operand in an assignment expression is of a lower type than the right operand, the compiler truncates the right operand to the type of the left operand.

  For Example:

  ```
  double num1 = 10;
  int num2;

  num2 = num1; // truncation (num1 gets truncated to type 'int')
  ```

ARITHMETIC AND RELATIONAL EXPRESSIONS

C compilers **promote the operand of the lower type in an arithmetic or relational expression to an operand of the higher type before evaluating** the expression.

For Example:

```
1034 * 10   evaluates to 10340       // an int result
1034 * 10.0 evaluates to 10340.0   // a double result
1034 * 10L  evaluates to 10340L     // a long result
1034 * 10.f evaluates to 10340.0f  // a float result
```

Compound Expressions

Magic Numbers

Values that appear out of nowhere in program code are known as magic numbers. These may be mathematical constants, standard rates or default values.

We avoid magic numbers by identifying them with symbolic names and using those names throughout the code - We set their value in either of the two ways:

1. Using an **unmodifiable variable**

   These are variables declared with the 'const' keyword. Once a variable is declared as const, its value cannot be changed after initialization. Attempting to modify a const variable will result in a compilation error.

2. Using a **macro directive**

   A macro is NOT a variable but is used for substitution at compile-time

Flag

Flags are variables that determine whether an iteration continues or stops. A flag is either true or false. Flag helps ensure that no paths cross one another. By introducing a flag, we avoid jump and multiple exit, obtain a flow chart where no path crosses any other and hence improved design.

A flag variable can enforce the single entry/exit principle for a loop by initially setting the flag to a specific value, modifying the flag within the loop based on a condition, and then checking the flag's value after the loop to ensure that the loop executed as expected. This approach ensures that the loop has a clear entry point before it starts and a clear exit point after it finishes, allowing for controlled and predictable loop behaviour.

```
#include <stdio.h>

int main(void)
{
  int i, value;
  int done = 0;  // flag
  int total = 0; // accumulator

  for (i = 0; i < 10 && done == 0; i++)
  {
    printf("Enter integer (0 to stop) ");
    scanf("%d", &value);

    if (value == 0)
    {
      done = 1;
    }
    else
    {
      total += value;
    }
  }

  printf("Total = %d\n", total);

  return 0;
}
```

Auto Sizing Vs Explicit sizing of the arrays

Auto sizing of the arrays refers to initialising an array with 1 or more elements while declaring it without specifying its size. The array will automatically store space in the memory according to the number of variables we initialise it with.
For example:
int arr[] = {2,4,5};
Here, we do not mention the size of the array but we initialise it with 3 elements, which will define an array of size 3 (12 bytes in this case).

Explicit sizing of the arrays refer to defining the size of the array while declaring it. After we explicitly define the size of the array, we cannot change the size of the array later on during the program's execution.
For example:

int arr[5];
Here, we are explicitly defining the size of the array as 5.

It is better to work with currency as integers rather than floating point. Why?

Working with currency as integers is often preferred over floating point numbers primarily due to **precision**, and **rounding issues.**

Integers are whole numbers and so they do not have any inherent precision issues associated with floating-point numbers. Floating point numbers are limited in their precision due to the way they are stored in binary, which lead to small rounding errors in calculations. By working with integers we can eliminate rounding errors.

Style Guidelines

1. **Readability and Maintainability:** Adhering to style guidelines improves code readability and maintainability. Clear and consistent code is easier to understand, reducing the chances of introducing bugs during development and making it simpler to update or modify code in the future.
2. **Consistency:** Style guidelines ensure that the codebase maintains a consistent and uniform appearance, making it easier for multiple developers to work on the same project. Consistency simplifies code review, debugging, and maintenance, as programmers can quickly understand and follow established conventions.

1. A well written code is easy to read and maintain.
2. The coding style is consistent and clear throughout.
3. It is easy to modify or upgrade code.
4. It becomes easy to spot bugs, find errors and debug the code

Absence of Style Guidelines

1. **Inconsistent Code -** Developers write code in their preferred style leading to lack of consistency within the codebase making it more difficult among the team mates to understand and work with each other's code, slowing down development and increasing the likelihood of errors.
2. **Code Maintenance Challenges** - Without guidelines, maintaining and updating the codebase becomes more challenging, as there is no standard to follow.
3. **Quality and Debugging Issues:** Inconsistent code may introduce quality issues and make debugging more challenging. Developers might overlook errors or spend extra time debugging due to code that deviates from expected practices.

Initialization of Variables - It ensures that the variable has defined initial value before they are used. This helps prevent unexpected behaviour and bugs due to uninitialized variables.
If a variable is not initialised it may contain arbitrary or garbage values left in memory, leading to unpredictable results, making code more challenging to debug and maintain.

Type - A type is a rule that defines how to store values in memory and which operations are admissible on those values. It defines the number of bytes available for storing values and hence the range of possible values.

What is the significance of the number preceding the (.) in the display of the floating point numbers? Provide explanations with reference to %3.5 lf and %0.5lf formatting?

The number preceding the (.) in the format specifier for displaying floating-point numbers controls the **minimum width of the field** that the number is printed in.
In this case,

1. **%3.5lf** - The number before the . (in this case, 3) specifies the minimum width of the field. It means that the floating-point number will be printed in a field at least 3 characters wide. The 5 after the (.) specifies the number of digits to display after the decimal point (the precision). If the number has more than 3 characters (including the integer part, the decimal point, and the fractional part), it will be displayed as is. If it's shorter, it will be padded with leading spaces to meet the minimum width requirement.
   For example, if you have the number 12.34500 and use %3.5lf, it will be printed as "12.34500" because it already has more than 3 characters.

2. **%0.5lf -** The number before the . (in this case, 0) specifies the minimum width of the field. It means that there is no minimum width requirement, and the output will expand to accommodate the number. The 5 after the . specifies the number of digits to display after the decimal point (5 in this case). If there are less than 5 digits after the (.) the compiler will fill the remaining places with 0's in most of the cases.

In C programming, casting is used to explicitly convert a value from one data type to another. This is done to ensure that the data is interpreted and processed correctly by the program. Casting can be useful when you need to perform operations on variables of different data types or when you want to store a value in a different type of variable.

```
 include <stdio.h>
int main()
{
double num1 = 3.14; int num2;                              num2 = num1;                                        num2 = (int)num1;
printf("num1 (double): %f\n", num1); printf("num2 (int): %d\n", num2); return 0;
}
```

A structured program is a type of computer program designed with clear modules and organized control structures, making the code more readable and maintainable. It emphasizes a top-down design approach, breaking the program into smaller, manageable parts. This approach helps reduce errors and makes it easier to understand and modify the code.

UPLING

pling describes the degree of interrelatedness of a module with other modules. Low coupling in the context of a function call refers to minimising the dependencies between th
ng function and the called function, promoting modular and independent design by limiting the knowledge each function has about the internal workings of the other.

DULAR DESIGN

lular design Involves breaking down a complex task into smaller, independent blocks (modules) that can be developed and tested separately. These modules can then be eas
bined to create a larger, well-organised program, making it easier to understand, maintain, and update.

NST QUALIFIER USED FOR A FUNCTION

const qualifier used for a function parameter indicates that the function promises not to modify the value of that parameter within the function, providing a clear contract and
renting unintended modifications to the argument. This enhances code clarity, readability, and helps catch accidental modifications during compilation.

SS BY REFERENCE

sing by address allows functions to directly access and modify the original data, avoiding the overhead of creating a duplicate copy of the entire variable. This approach is
icularly beneficial for large data structures, as it reduces memory consumption and improves performance. Additionally, passing by address is essential when you want a func
odify the original value, as changes made to a copy in pass by value would not affect the original variable outside the function.