Compilers

Learning Outcomes

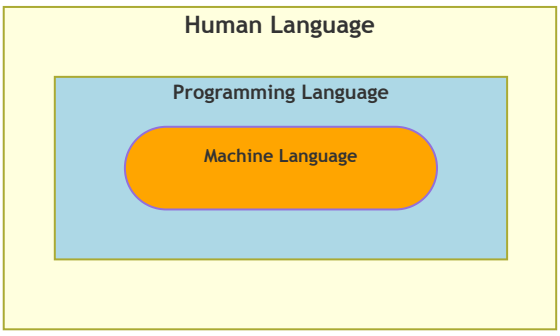
After reading this section, you will be able to:

- Describe the generations of programming languages
- Highlight the features of the C language
- Describe the general compilation process of a C compiler
- Edit, compile and run a computer program written in the C language
- Self-document a computer program using comments

Introduction

We transform the program instructions and program data into the bits and bytes that a computer understands using a compiler. We write the instructions and provide the data in a programming language that the compiler understands.

Programming languages demand completeness and greater precision than human languages. The ultimate interpreter of any computer program is the hardware. Hardware cannot interpret intent or nuance, and we need to provide much more detail in our instructions than we do in casual conversations amongst ourselves. In this sense, programming is a more arduous task than writing a formal report that someone else will read.



This chapter describes the generations of programming languages, identifies some key features of the C language, describes the compilers that we use to convert programs written in C into

binary instructions that hardware can execute and explains the basic syntax found in any C program.

Programming Languages

Generations

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

1. **Machine languages**. These are the native languages that the CPU processes. Each manufacturer of a CPU provides the instruction set for its CPU.
2. **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.
3. **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.
4. **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.
5. **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.

> **NOTE**
>
> - Eric Levenez maintains an up-to-date map of 50 of the more popular languages.
> - TIOBE Software tracks the most popular languages and the long-term trends based on world-wide availability of software engineers, courses and third-party vendors as calculated from Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube and Baidu search engines.

Features of C

C is one of the more popular third-generation languages. As a procedural language, C requires systematically ordered sets of instructions to perform a computational task and supports the collection of sets of instructions into so-called functions, which can be accessed from multiple points in the same program as required.

C serves as an excellent, first programming language for several reasons:

- 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++

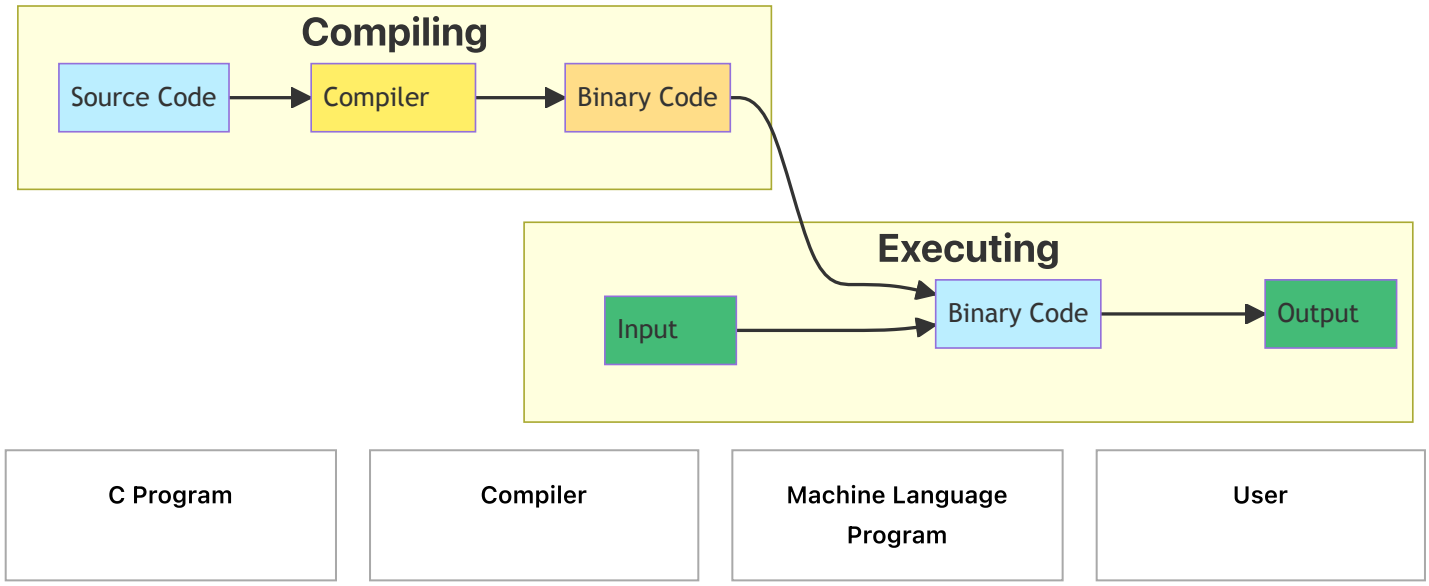
C programs execute quickly. Comparative times for a standard test (Sieve of Eratosthenes) are:

Language	Time to Run
Assembler	0.18 seconds
C	2.7 seconds
Basic	10 seconds

The C Compiler

A C compiler is an operating system program that converts C language statements into machine language equivalents. A compiler takes a set of program instructions as input and outputs a set of machine language instructions. We call the input to the compiler our source code and the output from the compiler the binary code. Once we compile our program, we do not need to recompile it, unless we have changed the source code in the interim.

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.