Chapter 1: Getting Started

The first example

Let us start with a very simple example. You can save the following source code into a CPP file named `hello.cpp`. Then you can compile it into an executable program by the command: `g++ hello.cpp`. Since the source file is in C++11 standard and the default setting of the compiler `g++` may be lower than C++11, you may get an error message. By adding the option `--std=c++11`, you should compile the source file successfully by `g++ hello.cpp --std=c++11`. The default file name of the output file should be `a.out`. You can find the output file in the working directory.

C or C++ are not fixed languages. They keep evolving over the past decades. It is impossible to convert all the details of the C and C++ standards here. Please remember that the features you use may be in different standards, and you can specify them by option `--std` to let the compiler compile the source code in different standards.

```
//hello.cpp
#include <iostream>
#include <vector>
#include <string>

using namespace std;

int main()
{
    vector<string> msg {"Hello", "C++", "World", "!"};

    for (const string& word : msg) //C++11 standard
    {
        cout << word << " ";
    }
    cout << endl;
}
```

To run the generated program `a.out`, you can simply run `./a.out` in the command window. The output of the program will be

```
$ ./a.out
Hello C++ World !
```

You can specify the output file name by an option `-o`. If you compile the source with `g++ hello.cpp --std=c++11 -o hello`, the output file will be `hello` nor `a.out`.

If you are using Linux or macOS, you can easily compile source files using `g++` or `gcc` after you install them. If you are using Windows, you can install Windows Subsystem for Linux (WSL). Then you can have a Linux environment in your Windows, and then you can use `g++`, `gcc` and some other similar tools inside it.

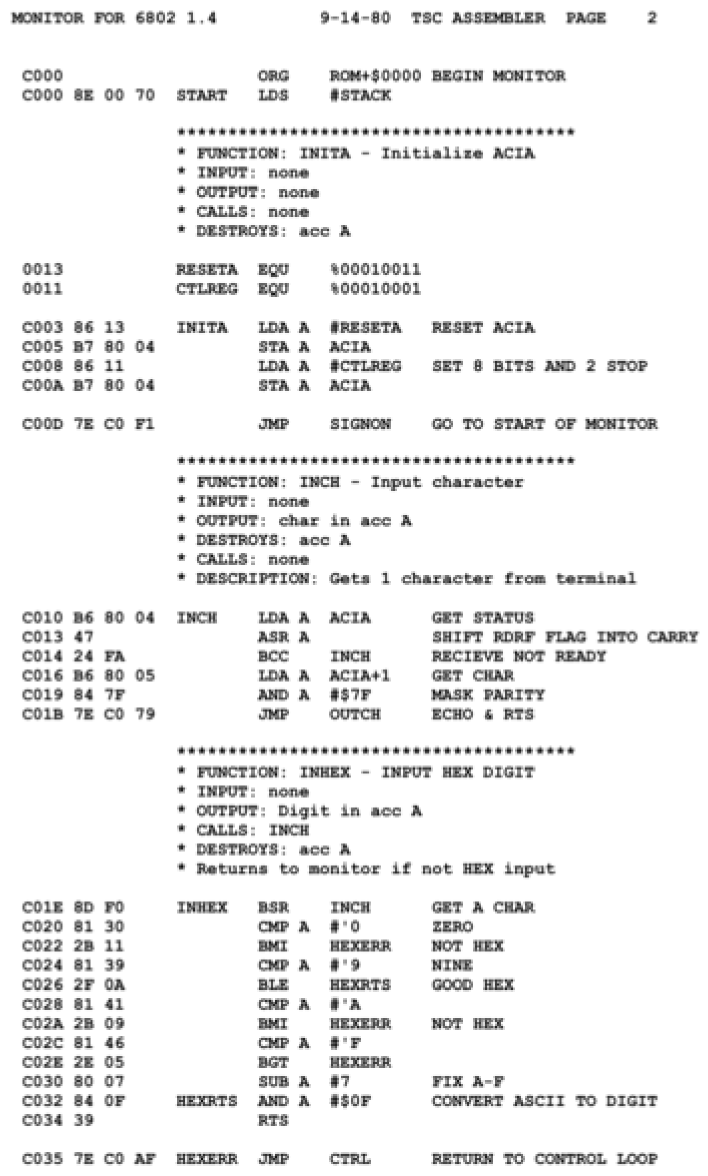
Different Programming Languages

Binary Instructions

There are some different instruction set architectures (ISA). The instructions are in binary and executed by CPUs. A binary instruction can be like `1011000001100001`. Different ISAs are not compatible with each other. Some programs were recorded as binary instructions on punched tapes as shown in the following figure. Anyway, it is very difficult to program complex software from binary instructions.

Assembly Languages

Compared with binary instructions, an assembly language (ASM) is more human-readable even it is still a low-level programming language. The ASM instruction `MOV AL, 61h` means to load number 97 (61 in hexadecimal) into register AL. There is a strong correspondence between the ASM instructions and the CPU code instructions. So you can regard ASM commands as human-readable machine instructions that can only be compiled and run on a specific CPU architecture. Nowadays, few programmers use assembly languages. Assembly languages are only used for hardware drivers, low-level parts of an operating system, real-time systems and some others.



```
MONITOR FOR 6802 1.4          9-14-80  TSC ASSEMBLER  PAGE    2


 C000                ORG    ROM+$0000 BEGIN MONITOR
 C000 8E 00 70  START   LDS    #STACK

                ***************************************
                * FUNCTION: INITA - Initialize ACIA
                * INPUT: none
                * OUTPUT: none
                * CALLS: none
                * DESTROYS: acc A

 0013           RESETA  EQU    %00010011
 0011           CTLREG  EQU    %00010001

 C003 86 13     INITA   LDA A  #RESETA   RESET ACIA
 C005 B7 80 04          STA A  ACIA
 C008 86 11             LDA A  #CTLREG   SET 8 BITS AND 2 STOP
 C00A B7 80 04          STA A  ACIA

 C00D 7E C0 F1          JMP    SIGNON    GO TO START OF MONITOR

                ***************************************
                * FUNCTION: INCH - Input character
                * INPUT: none
                * OUTPUT: char in acc A
                * DESTROYS: acc A
                * CALLS: none
                * DESCRIPTION: Gets 1 character from terminal

 C010 B6 80 04  INCH    LDA A  ACIA      GET STATUS
 C013 47                ASR A            SHIFT RDRF FLAG INTO CARRY
 C014 24 FA             BCC    INCH      RECIEVE NOT READY
 C016 B6 80 05          LDA A  ACIA+1    GET CHAR
 C019 84 7F             AND A  #$7F      MASK PARITY
 C01B 7E C0 79          JMP    OUTCH     ECHO & RTS

                ***************************************
                * FUNCTION: INHEX - INPUT HEX DIGIT
                * INPUT: none
                * OUTPUT: Digit in acc A
                * CALLS: INCH
                * DESTROYS: acc A
                * Returns to monitor if not HEX input

 C01E 8D F0     INHEX   BSR    INCH      GET A CHAR
 C020 81 30             CMP A  #'0       ZERO
 C022 2B 11             BMI    HEXERR    NOT HEX
 C024 81 39             CMP A  #'9       NINE
 C026 2F 0A             BLE    HEXRTS    GOOD HEX
 C028 81 41             CMP A  #'A
 C02A 2B 09             BMI    HEXERR    NOT HEX
 C02C 81 46             CMP A  #'F
 C02E 2E 05             BGT    HEXERR
 C030 80 07             SUB A  #7        FIX A-F
 C032 84 0F     HEXRTS  AND A  #$0F      CONVERT ASCII TO DIGIT
 C034 39                RTS

 C035 7E C0 AF  HEXERR  JMP    CTRL      RETURN TO CONTROL LOOP
```

High-level Languages

C, C++, Java, Python, SQL and most programming languages we use nowadays are all high-level languages. Some programmers who are only familiar with scripting languages such as Python may think C and C++ are more difficult and are low-level languages. You can think C and C++ are lower than scripting languages, but they are not low-level languages. C and C++ are user-friendly, written in simple English words and can be compiled on different platforms.

Scripting Languages 脚本语言

Many scripting languages such as Python are very popular for their simple syntax. Scripting languages are usually interpreted, but not compiled. It is the interpreter to run and act by following the commands in scripting languages. For a compiled language, the compiler will generate an executable file from the source code. Then the generated file is executed, not the compile to be executed. Scripting languages normally have much lower efficiency than compiled languages. Someone may argue that Python can run very efficiently even for deep learning. The truth is that Python heavily depends on various modules which are written in C or C++ for efficiency.

Java is an exception. It tries to take the advantages of scripting languages and compiled languages. Java can be compiled into bytecode, but executed on the Java Virtual Machine (JVM) not the operating system directly. Actually, some other scripting languages also take similar strategies to speed up their execution. Sometimes the boundary between scripting languages and compiled languages is not so clear.

Visual Languages

A visual programming language can let the users manipulate some graphical elements to create programs. This kind of languages is very friendly to users, especially to users who are not computer experts. Scratch is a successful visual language for kids to learn programming.

Scratch programming GUI

Why We Still Need C and C++

C and C++ are not easy languages. The program developed by beginners tends to crash. Beginners are prone to frustration at the beginning of learning C or C++. This book tries to make a smooth start. It will be easier if the learner has a better understanding of computer architectures.

Even though we have many `easy` languages, C and C++ are still important languages, and there is no good substitution for their tasks. Most fundamental software is developed in C or C++. Linux, Windows and most OS are written in C. The software in C or C++ can be a very long list, Apache web server, Oracle, MySQL, Adobe Photoshop, Google Chrome, Microsoft Office, OpenCV, gcc compiler, ... If you care about efficiency, C or C++ may be your best choice, sometimes your only choice. C and C++ are must-learn languages for students in computer science.

Compile and Link

Now we have a longer example. In the example, function `main()` calls function `mul()` , and `mul()` can return the product argument `a` and argument `b` by multiplying them.

```cpp
//mainmul.cpp
#include <iostream>

using namespace std;

int mul(int a, int b)
{
    return a * b;
}
int main()
{
    int a, b;
    int result;

    cout << "Pick two integers:";
    cin >> a;
    cin >> b;

    result = mul(a, b);

    cout << "The result is " << result << endl;
    return 0;
}
```

Since the file name of the source file is `mainmul.cpp` , we can compile it using `g++ mainmul.cpp -o mainmul` and generate an executable file `mainmul` . If the source code is long, we can still put all source code into one file. But it is not convenient to manage the source code. To better manage the source code, we can put the source code into different files according to their functions. Such as we can put function `main()` into file `main.cpp` , and `mul()` into `mul.cpp` . Then they can be organized as follows.

```
//main.cpp
#include <iostream>
#include "mul.hpp"

using namespace std;
int main()
{
    int a, b;
    int result;

    cout << "Pick two integers:";
    cin >> a;
    cin >> b;

    result = mul(a, b);

    cout << "The result is " << result << endl;
    return 0;
}
```

```
//mul.hpp
#pragma once
int mul(int a, int b);
```

```
//mul.cpp
#include "mul.hpp"

int mul(int a, int b)
{
    return a * b;
}
```

Then we can compile the two source files by `g++ -c main.cpp` and `g++ -c mul.cpp` specifically. The object files `main.o` and `mul.o` will be generated. We can link the two object files into one executable file by `g++ main.o mul.o -o mul` . Here the option `-c` is to tell the compiler `g++` to compile the source file only. To compile and link multiple files together, you can simply use `g++ main.cpp mul.cpp -o mul` . But if you have many files in your project, it may take a long time to compile all files. It is better to compile the changed files only. If you do not want to type the commands one by one, you can use a `Makefile` to save the rules for compilation and link, and then a command `make` can help you do everything.

You may notice that we have three files here. `*.c` and `*.cpp` files are source files. `*.h` and `*hpp` are header files. We normally use a header file to store the declarations of functions. A function declaration introduces the function name and its type (arguments). If there is not a header file, the compiler will not know what `mul` is when we compile `main.cpp`, and it will pop an error message like the following. `mul` is an undeclared identifier, and the compiler does not know how to handle it.

```
$ g++ -c main.cpp
main.cpp:14:14: error: use of undeclared identifier 'mul'
    result = mul(a, b);
             ^
1 error generated.
```

`#include` macro is to insert the contents in `mul.hpp` into `main.cpp`. Then the source file after preprocessing will be like the following. Even the compiler cannot find the body (definition) of function `mul()`, it will know its interface. The interface defines how it takes arguments and what type will be returned. A linker will find the body of function `mul.o` and link the two object files together, and then generate an executable file.

```
#include <iostream>
int mul(int a, int b); //#include "mul.hpp" is replaced by the contents in mul.hpp

using namespace std;
int main()
{
    int a, b;
    int result;

    cout << "Pick two integers:";
    cin >> a;
    cin >> b;

    result = mul(a, b);

    cout << "The result is " << result << endl;
    return 0;
}
```

Different Errors

It can be easy to find the problem if you know the type of errors when some errors appear.

Compilation Errors

Compilation errors will appear when the compiler cannot `understand` your source code. Such as in the following example, `n1` is multiplied by `n2`. But what are they? The compiler can not find their definitions. Surely the compiler will not know how to multiply them, and can only pop an error message, a compilation error message. Please note that `;` is also missed in the following source code. It will also cause another compilation error.

```
int mul(int a, int b)
{
    return n1 * n2
}
```

The compilation error messages generated by the prevous source code.

```
$ g++ -c mul.cpp
mul.cpp:5:12: error: use of undeclared identifier 'n1'
    return n1 * n2
           ^
mul.cpp:5:17: error: use of undeclared identifier 'n2'
    return n1 * n2
                ^
mul.cpp:5:19: error: expected ';' after return statement
    return n1 * n2
                  ^
                  ;
3 errors generated.
```

If there are compilation errors, it is better to check the source code to make sure all spells are correct, no `;` or `(` missing. For peole in East Asia, they should also be very careful with characters like `；` `，` `）` , etc. They are different from `;` and `)` . Another Chinese character `　` which is a wider space is easy to be miss used with the normal space.

Link Errors

Link errors are caused in the link stage. The most common link error is `undefined symbol` . It will appear if the linker cannot find the definition of a function in object files. In the previous example if you misspell the function name from `mul` to `Mul` as follows.

```
//mul.cpp
#include "mul.hpp"

int Mul(int a, int b) //mul() is callled in main() but here it is Mul()
{
    return a * b;
}
```

You will get a link error as follows even you can comile the two files `main.cpp` and `mul.cpp` separately.

```
$ g++ main.o mul.o -o mul
Undefined symbols for architecture arm64:
  "mul(int, int)", referenced from:
      _main in main.o
ld: symbol(s) not found for architecture arm64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

If a link error appears, please check the link option first. You should also check the function name that is complained.

Runtime Errors

Even there are many kinds of runtime errors, the most common runtime error is caused by pointers. The following source code can be compiled and generated to an executable file successfully. The reason is that the grammar of the source code is correct.

```cpp
#include <iostream>
int main()
{
    int * p = nullptr;
    p[0] = 5;
    std::cout << p[0] << std::endl;
    return 0;
}
```

But if you run the program it generates, you will get the following error. Segmentation fault normally is caused by reading or writing some unpermitted memory regions, and it is the nightmare of most beginners. If it appears, please check the pointer accessing. There are some more kinds of runtime errors, but I will not introduce them here because of the limitation of space.

```
$ ./a.out
segmentation fault  ./a.out
```

Preprocessor and Macros

The compilation is briefly introduced in the previous part. There are several steps to the compilation before a compiler really compiles the source code. The previous step before the compilation is preprocessing.

Preprocessing directives begin with a `#` character, and each directive occupies one line. The most commonly used preprocessing directives are `include` , `define` , `undef` , `if` , `ifdef` , `ifndef` , `else` , `elif` , `endif` , `line` , `error` and `pragma` . `include` is just introduced, and it appears in almost all source files.

`define` is another popular directive, and can be used to define some macros. In the following source code, `PI` is defined as `3.14` by using `#define PI 3.14` .

```
#define PI 3.14
double len(double r)
{
    return 2.0 * PI * r;
}
```

After preprocessing, the source code will be as the following. `PI` has been replaced with `3.14` . Then it will be sent to a compiler. Macro `PI` is not a variable. The preprocessing for macros is like text replacement.

```
double len(double r)
{
    return 2.0 * 3.14 * r;
}
```

Since macros behaviors like text replacement, sometimes it is dangerous and cause bugs. If we define `PI` as `2.14+1.0` , the statement is grammatically correct.

```
#define PI 2.14+1.0
double len(double r)
{
    return 2.0 * PI * r;
}
```

But after preprocessing the return value of the function will be `4.28+r` , not `2.0*3.14*r` . It may be not what you expected. Anyway, the source code will be compiled successfully, and the compiler will not report any warning or error. You should be very careful when you use macros.

```
double len(double r)
{
    return 2.0 * 2.14+1.0 * r; //= 4.28 + r
}
```

We can even define a macro that works like a function. Sometimes macros can achieve better efficiency than functions since macros have no overhead of function callings.

```
#define MAX(a,b) (((a)>(b))?(a):(b))

int main()
{
    //...
    float a = 2.0f;
    float b = 3.0f;
    float m;
    m = MAX(a, b);
    //...
}
```

Simple Input and Output

The following source code is C style output. Function `printf()` can take several arguments. The first one is a string to specify how to interpret the data. `%d` will let the function interpret `v` as an integer. Another similar function is `fprintf()`. You can find its usage by searching online.

```
int v = 100;
printf("Hello, value = %d\n", v);
```

The C++ style source code for output is different. You can use an overloaded operator `<<`. Here `cout` is an object of data type ostream in namespace `std`. It is declared in the header file `iostream`. So you should include `iostream` before you use it. `std::endl` is an output-only I/O manipulator. It will output a new line character and flushes.

```
int v = 100;
std::cout << "Hello, value = " << v << std::endl;
```

To take data from the standard input stream, you can call a C function `scanf()` as follows. Be careful that you should take the address of the variable `v` using `&`. Otherwise, the program will try to write the address which is the value of `v`. It may be a random number here, and you may get a `segmentation fault` runtime error.

```
int v;
int ret = scanf("%d", &v);
```

The C++ style input is safer. You can use the overloaded operator `>>` as follows. The object `std::cin` will convert the input data into the format of `v` automatically.

```
int v;
std::cin >> v;
```

The input and output are just introduced briefly here. You can learn more in the following chapters of the book.

Command Line Arguments

We can pass some values to our programs through the command line. The command we use previously `g++ hello.cpp -o hello` send three command line arguments `hello.cpp` , `-o` and `hello` to the compiler `g++` .

The command line arguments can be handled using `main()` arguments. The function argument `argc` refers to the number of the command arguments, and `argv[]` is an array that contains pointers. Each element in `argv[]` points to a `char` array string.

```
//argument.cpp
#include <iostream>

using namespace std;
int main(int argc, char * argv[])
{
    for (int i = 0; i < argc; i++)
        cout << i << ": " << argv[i] << endl;
}
```

When the program from the above code is executed with some arguments, it will print the arguments one by one with their indices.

```
$ ./argument hello.cpp -o hello
0: ./argument
1: hello.cpp
2: -o
3: hello
```

Exercises

A simple example

Please write a simple `Hello world` example using C and C++ style output separately, then compile and run the example.

Find bugs

Put the following code into 3 files, and compile them together into an executable file. Find the bugs if there are any.

```
//main.cpp
#include <iostream>
#include "Add.h"

using namespace std;
int main()
{
    int num1 =  2147483647;
    int num2 =  1;
    int result =  0;

    result = add(num1, num2);

    cout << “The result is ” << result << endl;
    return 0;
}
```

```
//add.h
#pragma once
int add(int n1, int n2);
```

```
//add.cpp
#include "add.h"
int Add(int number1, int number2);
{
    return n1 + n2;
}
```