C/C++ Programming Techniques

Introduction

Thanh-Hai Tran

Electronics and Computer Engineering
School of Electronics and Telecommunications

Hanoi University of Science and Technology
1 Dai Co Viet - Hanoi - Vietnam

Outline

- **General information of the course**
- **Course syllabus**
- **Overview**
  - Basic concepts: Computer, Program, Programming
  - Languages: Machine Language, Programming language
  - Software development cycle
  - Errors
- **Introduction to C/C++ language**
  - C/C++ history
  - Stages of the program’s lifetime
  - Some features of C/C++
  - Introduction to VS Code : A source code editor

General information of the course

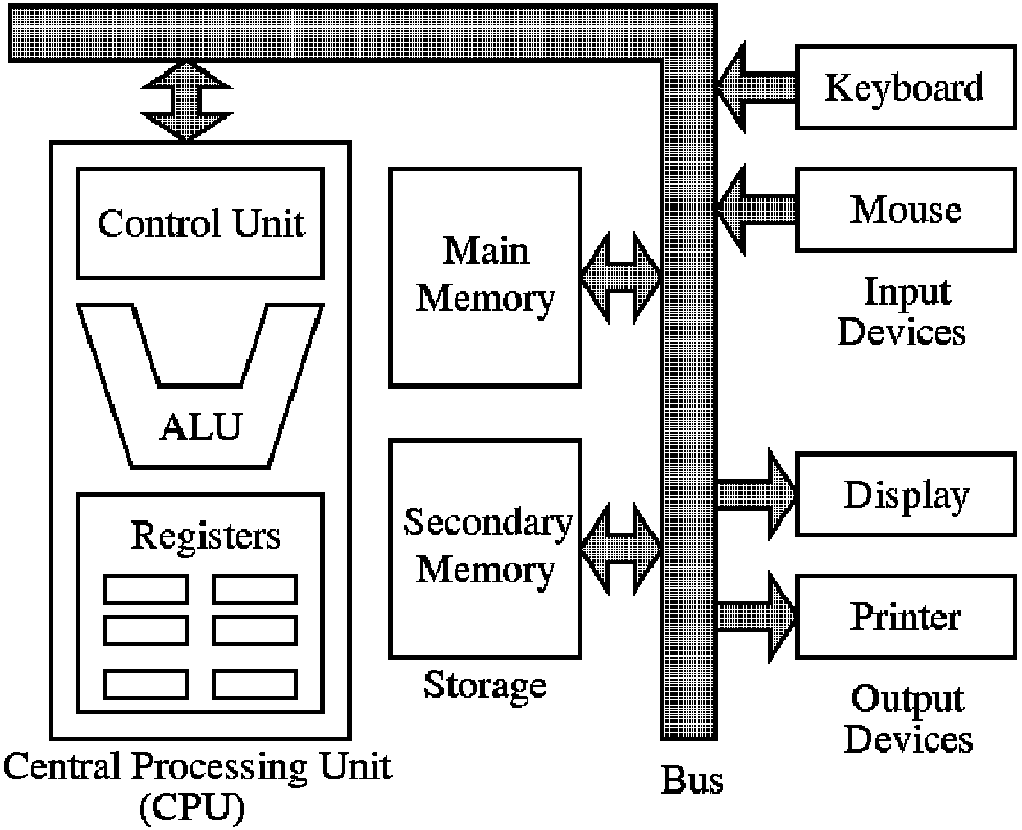
- **Course:**
  - Course: C/C++ Programming techniques
  - Course code: ET2031 2(2-0-1-4)
  - Class code:
  - Lectures: Friday 12h30-14h55 – D6 106
- **Instructor:**
  - Assoc. Prof. Thanh-Hai Tran
  - Electronics and Computer Engineering Dept. – 406 D9
  - Email: hai.tranthithanh1@hust.edu.vn

Course syllabus

Week	Lecture
1-2	The basic concepts
3	Excercise
4	Arrays and Pointers
5-6	Function Oriented Programming
7	Excercise
8	Data type
9-10	Object-Oriented Programming
11	Excercise
12	Inheritance
13	Standard Template Library
14	Other techniques: file / exception
15	Project presentation

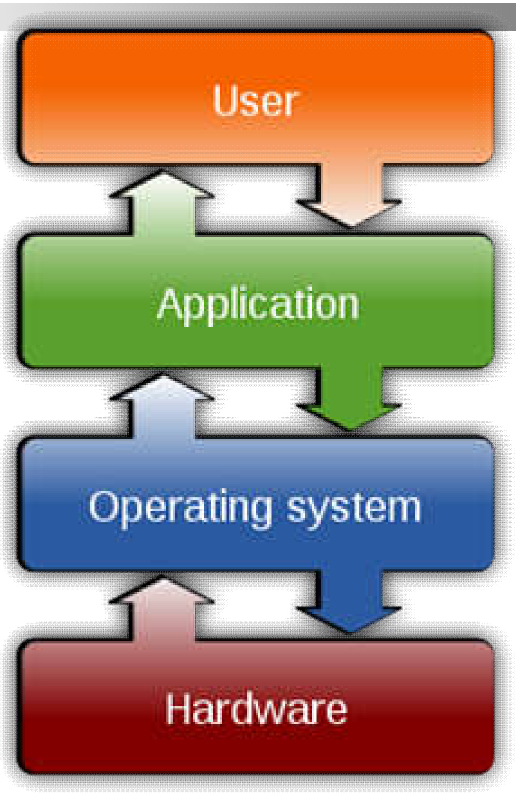
Basic concepts

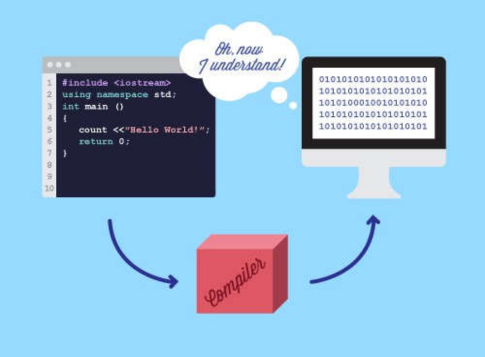
- Computer
- Main components:
  - Hardware
  - Computer Programming
  - Software
- Logic components:
  - Input unit, Output unit,
  - Memory unit,
  - Arithmetic and Logic Unit (ALU),
  - Central Processing Unit (CPU),
  - Second Storage unit



Basic concepts

- **Operating system:** is a software which acts as an interface between the end user and computer hardware
- **Algorithms:** a finite sequence of well-defined, computer-implementable instructions, typically to solve a class of problems or to perform a computation
- **Compiler:** a computer program that translates computer code written in one programming language (the source language) into another language (the target language).

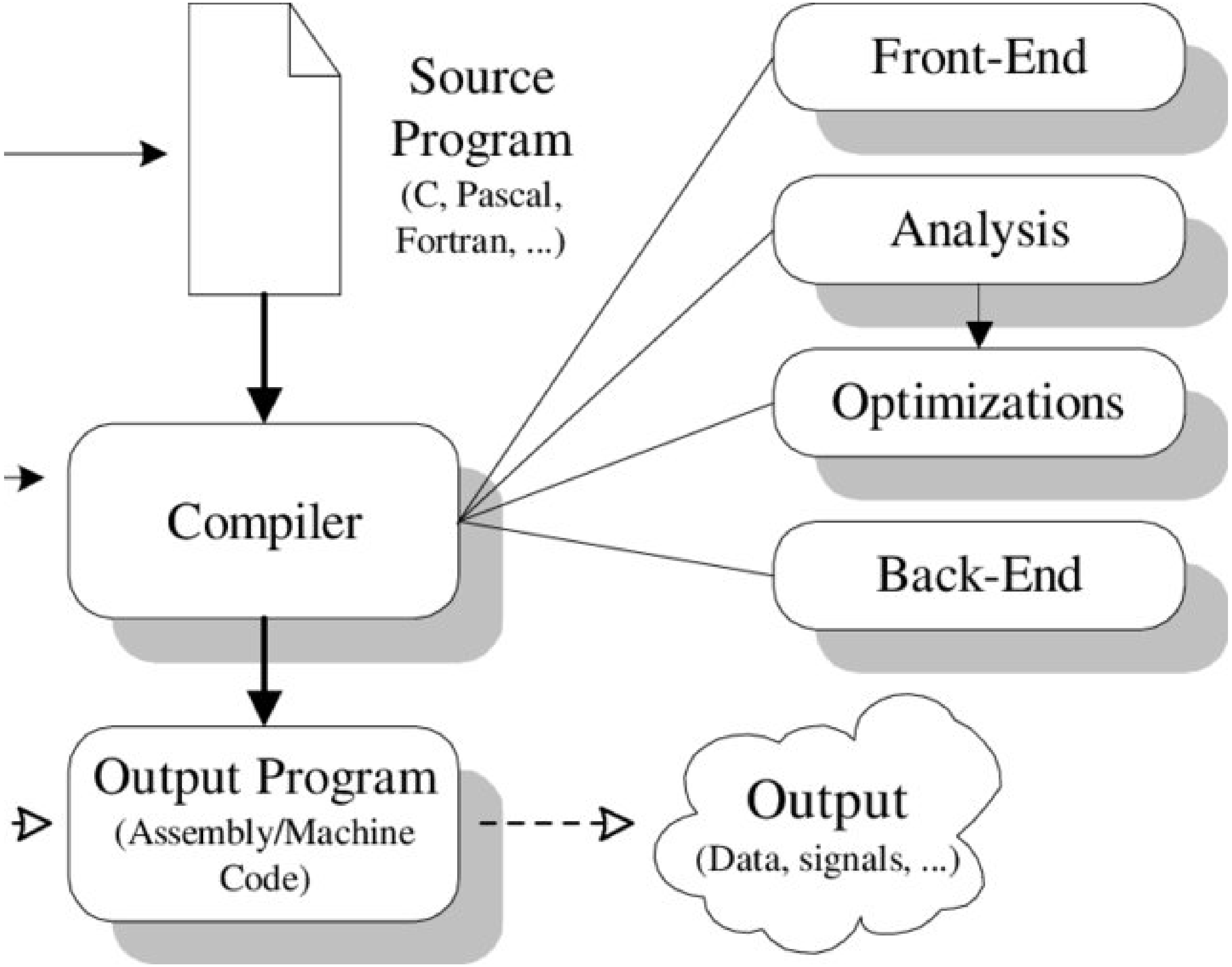




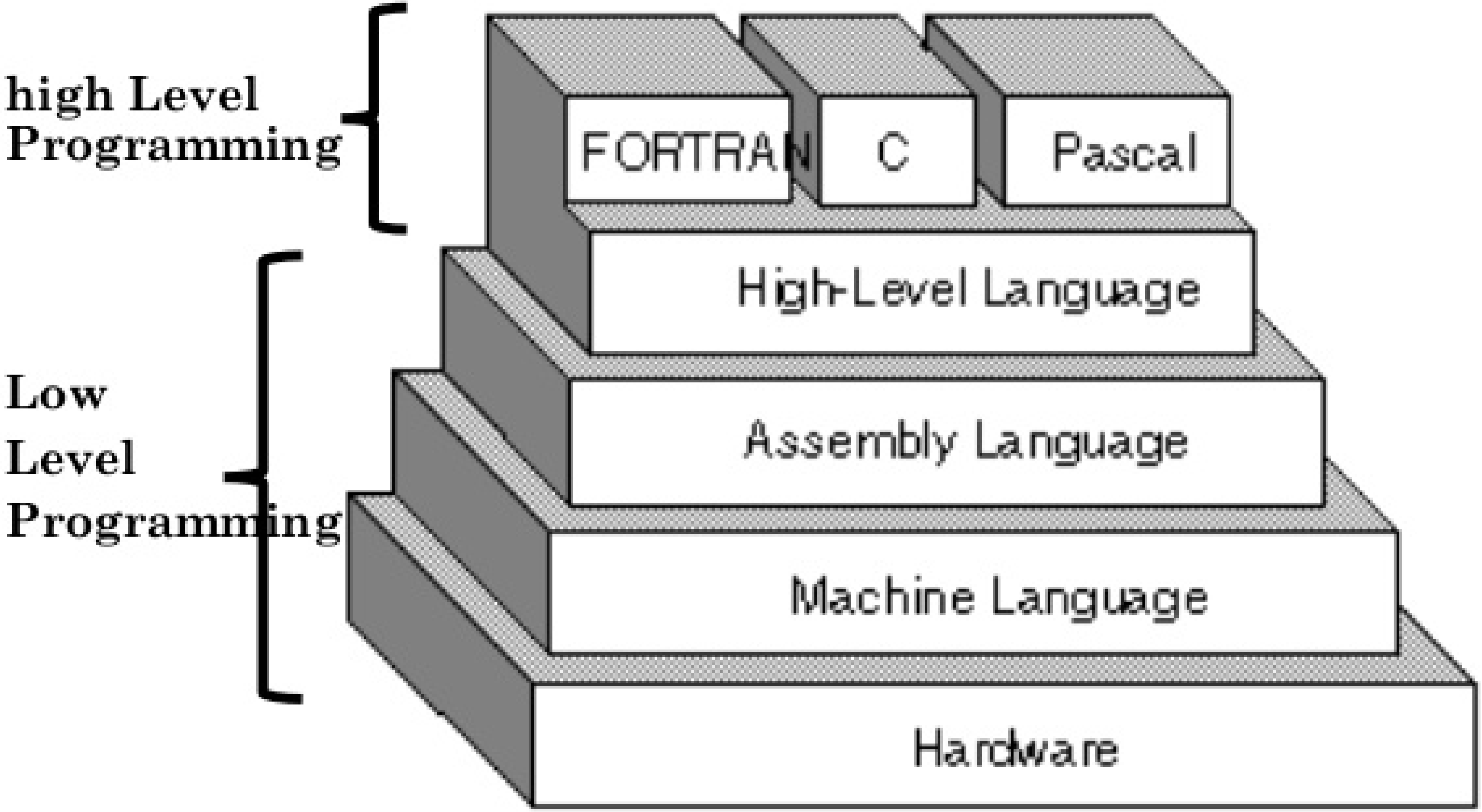
```
#include <iostream>
using namespace std;
int main ()
{
   count <<"Hello World!";
   return 0;
}
```

```
01010101010101010101010
10101010101010101010101
10101000100101010101010
10101010101010101010101
10101010101010101010101
```

Basic concepts



Programming languages

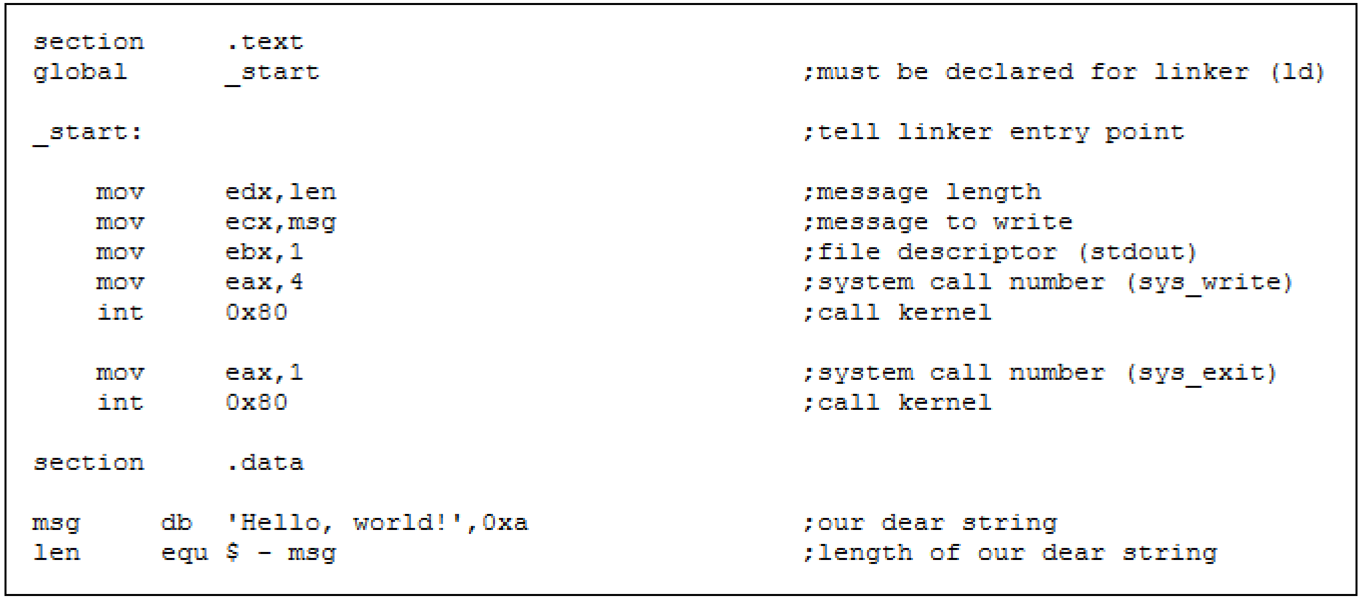


Machine language

- **Main features:**
  - the only **language** a computer is capable of understanding
  - can differ by operating system
  - Defined by hardware designer
- **Computer understands only binary code**
  - A binary code composes of 0/1 bits
  - The letter ‘A’ has binary code: 01000001
  - The number 65 has binary code: 1000001
- **How can computer understand what does “1000001” mean ?**
  - Depends on the command
  - The programmer must understand the allocated memory contain which kind of values
- **Computer memory contains both commands and data**

Assembly language

- low-level programming language in which there is a very strong correspondence between the instructions in the language and the architecture's machine code instructions
- Composes of
  - Simple code
  - Understandable by programmer
  - Need to be complied into machine code



```
section     .text
global      _start                              ;must be declared for linker (ld)

_start:                                         ;tell linker entry point

    mov     edx,len                             ;message length
    mov     ecx,msg                             ;message to write
    mov     ebx,1                               ;file descriptor (stdout)
    mov     eax,4                               ;system call number (sys_write)
    int     0x80                                ;call kernel

    mov     eax,1                               ;system call number (sys_exit)
    int     0x80                                ;call kernel

section     .data

msg     db  'Hello, world!',0xa                 ;our dear string
len     equ $ - msg                             ;length of our dear string
```

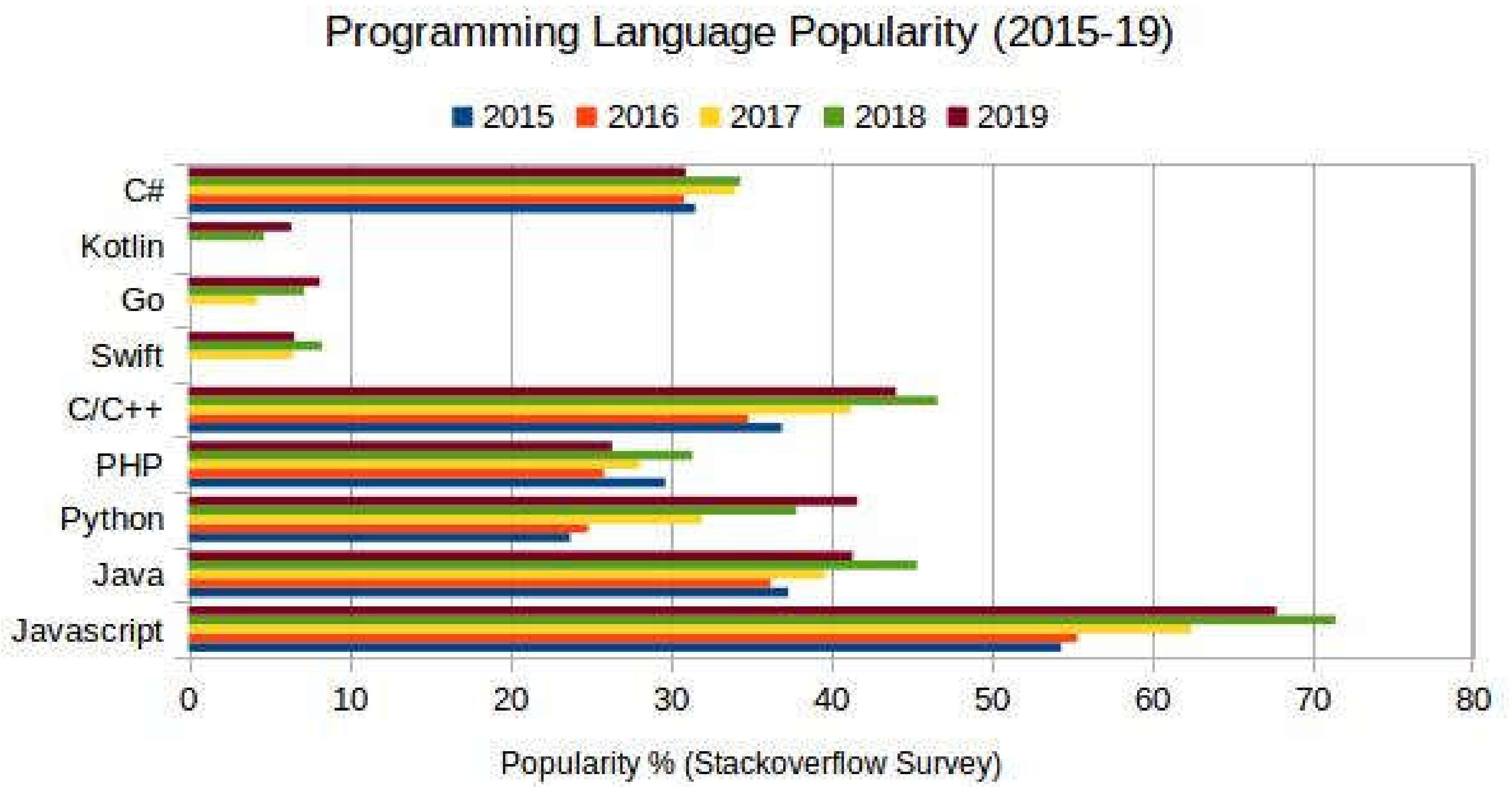
High level language

- **Similar to English, using common math's syntaxes**
- **Each command corresponds to a task**
- **To be understood and executed by machine, it needs to be compiled**
  - Compiler: converts to machine code
  - Interpreter: execute directly the high level program

Programming language

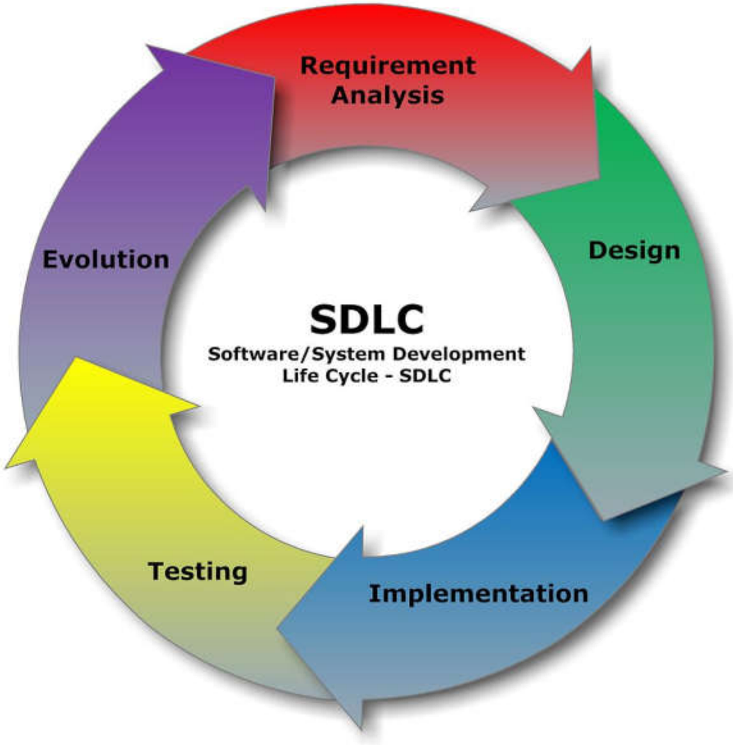
- **Concept:** a formal language comprising a set of instructions that produce various kinds of output.
- **Developement:**
  - Machine code: binary code, not need to be compiled, depends on micro-processor
  - 2nd generation (assembly): need to be compiled, understable, depends on micro-processor
  - 3rd generation: control structure, data structure, package: Fortran, C/C++, COBOL, PASCAL, ...
  - 4th generation: improve efficiency, reduce errors: SQL, LabVIEW, ColdFusion,…

Language popularity



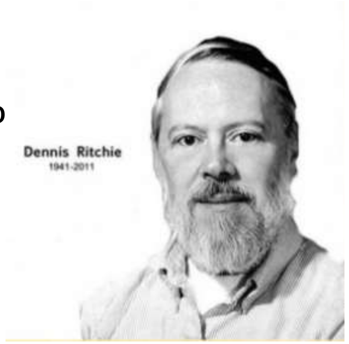
Software developement cycle

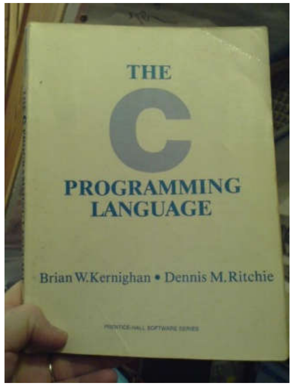
- **Different steps:**
  - Problem definition
  - Design
  - Coding
  - Evaluation
  - Maintenance
- **Evaluation:** evaluate the fucntionalitis of program
- **Debug:** find out the cause of errors and correct them.

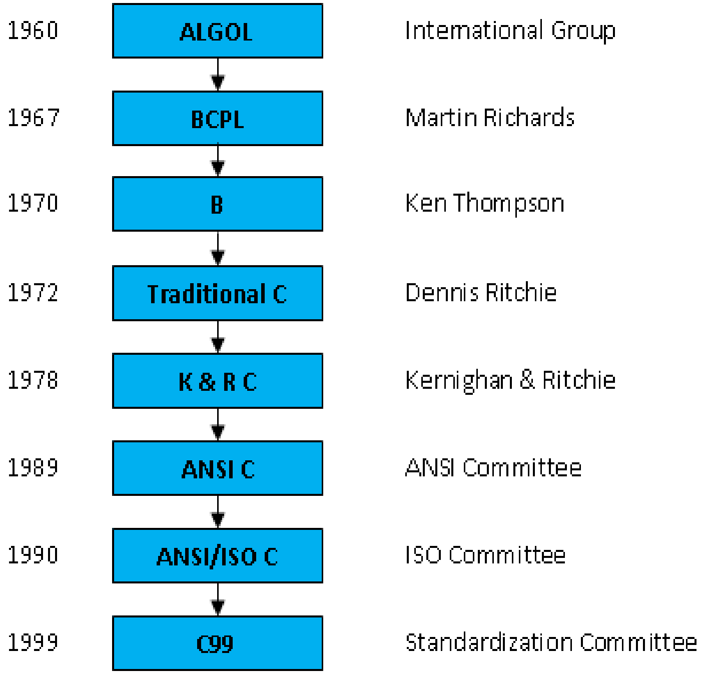


Introduction to C/C++

- **History:**
  - Was born in 1970, parallel with Unix OS (90% of UNIX is written in C)
  - Creator: Dennis Ritchie (Bell Labs.)
- **Goal:**
  - Focus on efficiency
  - Able to access to low-level hardware
  - Structured language (instead of assembly language programming)
- **C is a language between low-level .**
  - Able to access directly to memory
  - Simple syntax, keywords
- **… and high level**
  - Independent of hardware
  - Structure, Function, package
  - Data type cheking







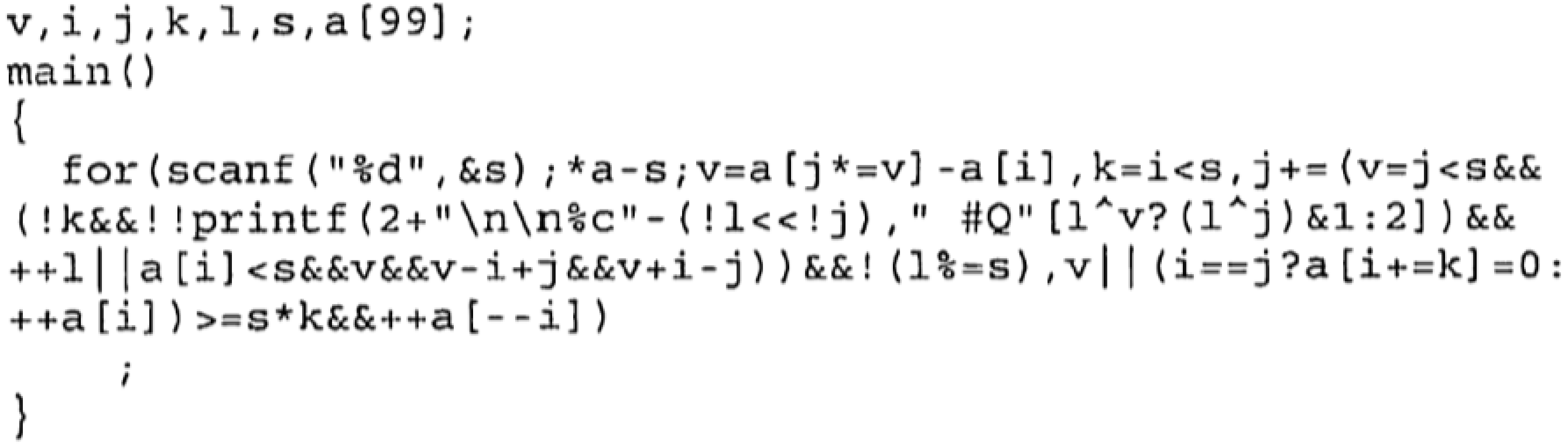
1960	ALGOL	International Group
1967	BCPL	Martin Richards
1970	B	Ken Thompson
1972	Traditional C	Dennis Ritchie
1978	K & R C	Kernighan & Ritchie
1989	ANSI C	ANSI Committee
1990	ANSI/ISO C	ISO Committee
1999	C99	Standardization Committee

C-based languages

- **C++** includes all features of C, but adds classes and other features to support object-oriented programming
- **Java:** is based on C++ and therefore inherits many C features
- **C#:** is a more recent language derived from C++ and java
- **Perl:** is originally a fairy simple scripting language and overtime it has grown and adopted many of the features of C

Strength and weakness of C

- **Strengths**
  - Efficiency
  - Portability
  - Power
  - Flexibility
  - Standard libraries
  - Integration with Unix
- **Weakness: C can be**
  - Error-prone
  - Difficult to understand Power
  - Difficult to modify

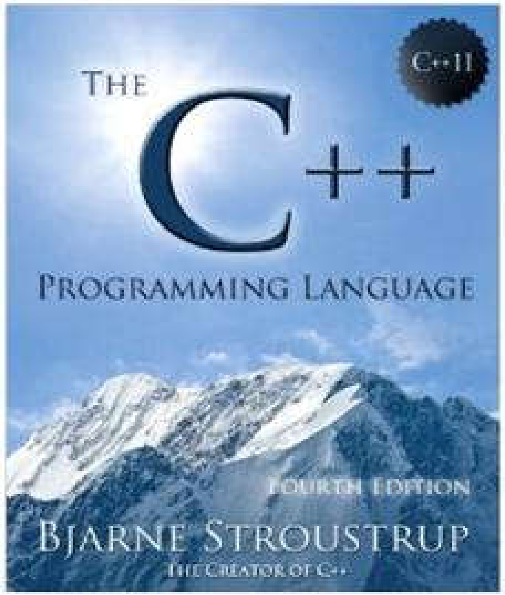


```
v,i,j,k,l,s,a[99];
main()
{
  for(scanf("%d",&s);*a-s;v=a[j*=v]-a[i],k=i<s,j+=(v=j<s&&
(!k&&!!printf(2+"\n\n%c"-(!l<<!j)," #Q"[l^v?(l^j)&1:2])&&
++l||a[i]<s&&v&&v-i+j&&v+i-j))&&!(l%=s),v||(i==j?a[i+=k]=0:
++a[i])>=s*k&&++a[--i])
      ;
}
```

C++ history

- **History:** created in 1979 by expanding the C language. Author: Bjarne Stroustrup (Bell Labs.)

- **Target:**
  - Add new features
  - Overcoming some of the disadvantages of C
- **Additional new features compared to C:**
  - Object Oriented Programming (OOP)
  - General programming (template)
  - Many small features make programming more flexible (add bool type, declare variable anywhere, strong type, define function stack, namespace, handle exception, ...)



Stage of program’s lifetime

- Creating the source code
- Compiling
- Linking
- Loading
- Executing

Creating source code

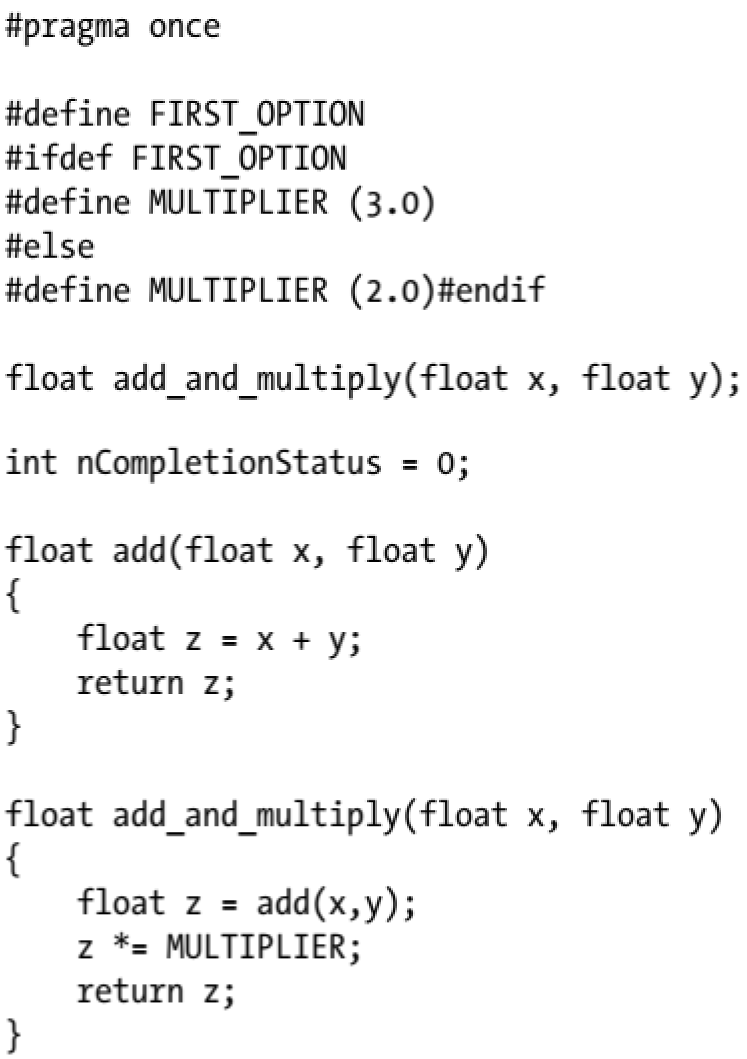
- Source file **main.c**, which contains the main() function.
- Header file **function.h**, which declares the functions called and the data accessed by the main() function.
- Source file **function.c**, which contains the source code implementations of functions and
  instantiation of the data referenced by the main() function.

Compiling

- Compiling is a process of transforming source code written in one programming language into another programming language
- The process of compiling is performed by the program called the **compiler**
- The input for the compiler is a **translation unit**. A typical translation unit is a text file containing the source code
- A program is typically comprised of many translation units

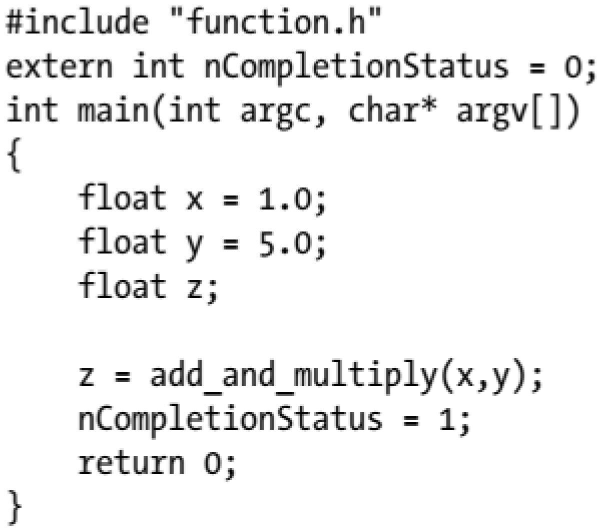
Example

function.h



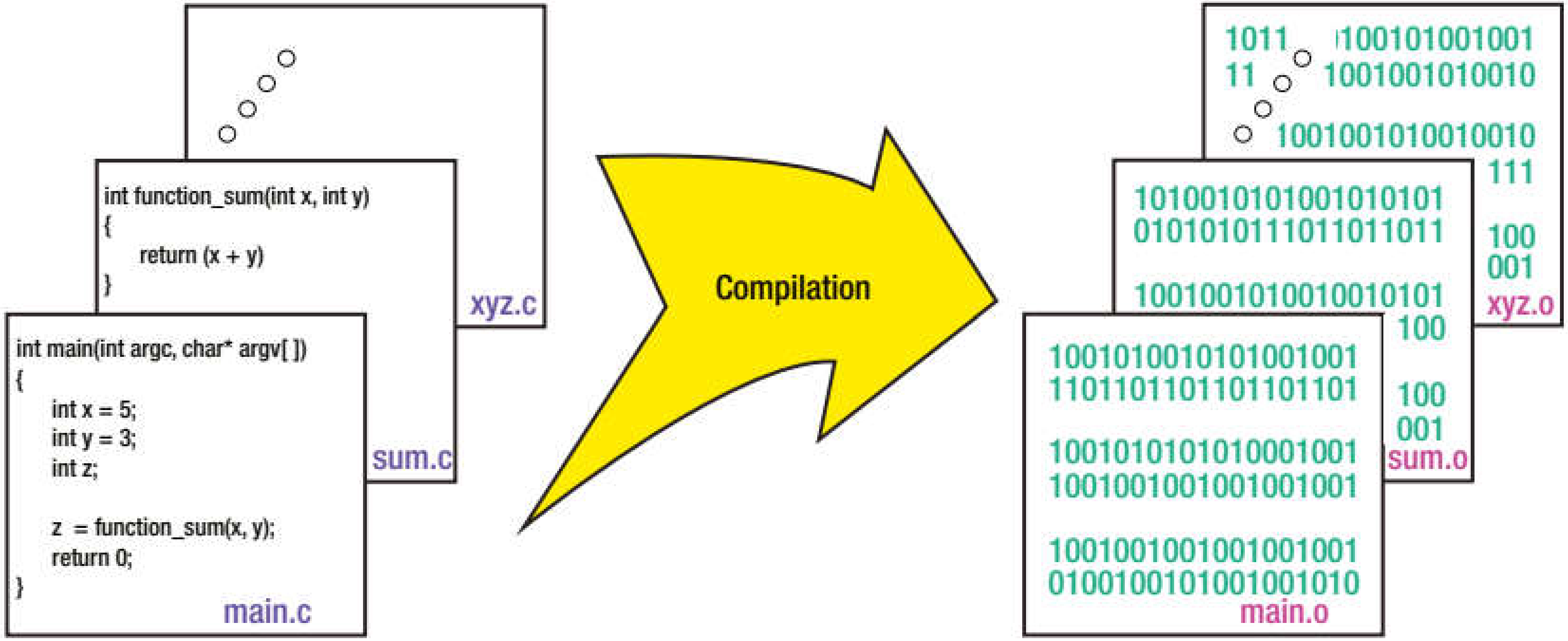
```
#pragma once

#define FIRST_OPTION
#ifdef FIRST_OPTION
#define MULTIPLIER (3.0)
#else
#define MULTIPLIER (2.0)#endif

float add_and_multiply(float x, float y);

int nCompletionStatus = 0;

float add(float x, float y)
{
    float z = x + y;
    return z;
}

float add_and_multiply(float x, float y)
{
    float z = add(x,y);
    z *= MULTIPLIER;
    return z;
}
```

main.c



```
#include "function.h"
extern int nCompletionStatus = 0;
int main(int argc, char* argv[])
{
    float x = 1.0;
    float y = 5.0;
    float z;

    z = add_and_multiply(x,y);
    nCompletionStatus = 1;
    return 0;
}
```

Compiling



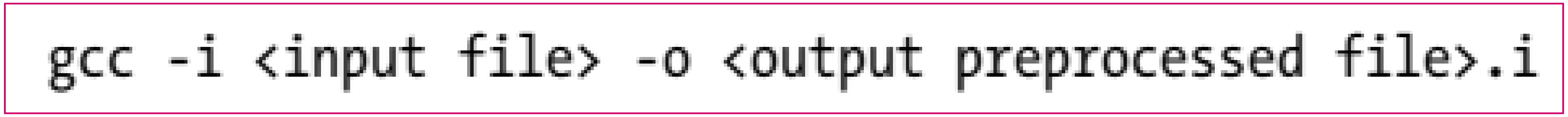
```
int function_sum(int x, int y)
{
     return (x + y)
}
```

```
int main(int argc, char* argv[ ])
{
     int x = 5;
     int y = 3;
     int z;

     z  = function_sum(x, y);
     return 0;
}
```

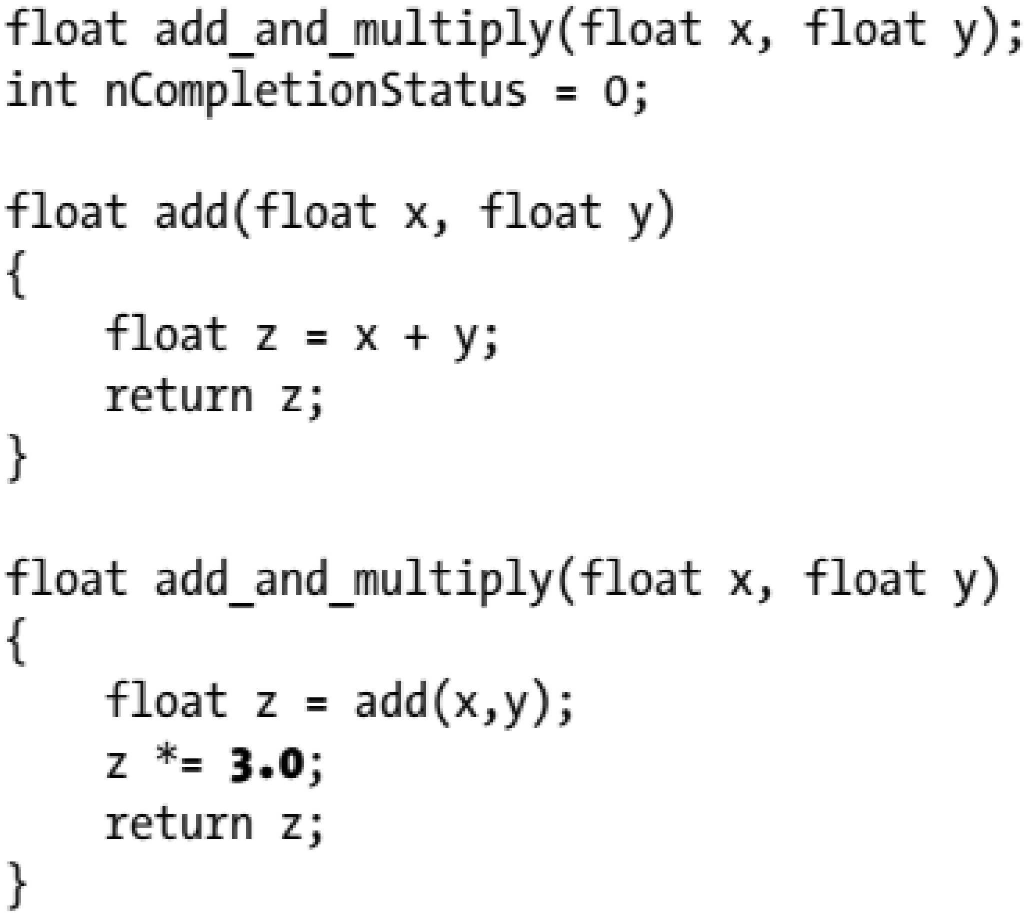
Compiling: 1) pre-processing

- The standard first step in processing the source files is running them through the special text processing program called a **preprocessor**, which performs one or more of the following actions:
  - Includes the files containing definitions (include/header files) into the source files, as specified by the #include keyword.
  - Converts the values specified by using #define statements into the constants.
  - Converts the macro definitions into code at the variety of locations in which the macros are invoked.
  - Conditionally includes or excludes certain parts of the code, based on the position of #if, #elif, and #endif directives.
- The output of the preprocessor is the C/C++ code in its final shape, which will be passed to the next stage, syntax analysis.

Compiling: 1) pre-processing



```
gcc -i <input file> -o <output preprocessed file>.i
```



```
float add_and_multiply(float x, float y);
int nCompletionStatus = 0;

float add(float x, float y)
{
    float z = x + y;
    return z;
}

float add_and_multiply(float x, float y)
{
    float z = add(x,y);
    z *= 3.0;
    return z;
}
```

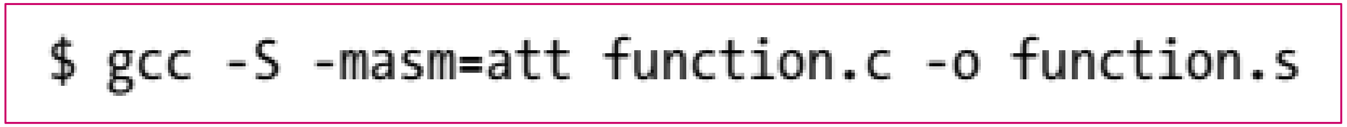
Compiling: 2) Linguistic Analysis

- **Lexical analysis**, which breaks the source code into non-divisible tokens.
- **Parsing/syntax analysis** concatenates the extracted tokens into the chains of tokens, and verifies that their ordering makes sense from the standpoint of programming language rules.
- **Semantic analysis** is run with the intent to discover whether the syntactically correct statements actually make any sense.

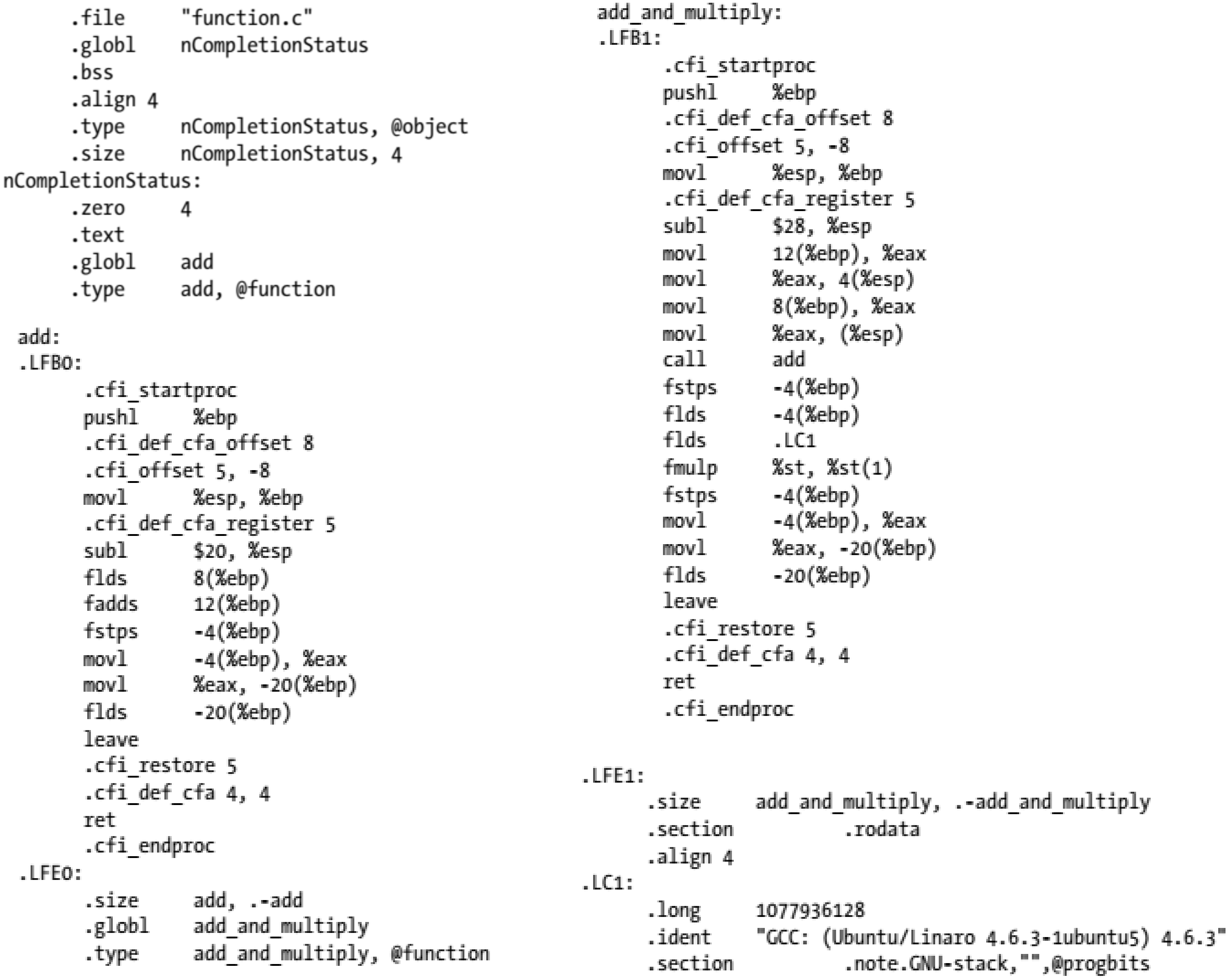
Compiling: 3) Assembling

- The compiler reaches this stage only after the source code is verified to contain no syntax errors.
- In this stage, the compiler tries to convert the standard language constructs into the constructs specific to the actual CPU instruction set.
- Different CPUs feature different functionality treats, and in general different sets of available instructions, registers, interrupts, which explains the wide variety of compilers for an even wider variety of processors.

Compiling: 3) Assembling



```
$ gcc -S -masm=att function.c -o function.s
```



```
      .file     "function.c"
      .globl    nCompletionStatus
      .bss
      .align 4
      .type     nCompletionStatus, @object
      .size     nCompletionStatus, 4
nCompletionStatus:
      .zero     4
      .text
      .globl    add
      .type     add, @function

 add:
 .LFB0:
       .cfi_startproc
       pushl     %ebp
       .cfi_def_cfa_offset 8
       .cfi_offset 5, -8
       movl      %esp, %ebp
       .cfi_def_cfa_register 5
       subl      $20, %esp
       flds      8(%ebp)
       fadds     12(%ebp)
       fstps     -4(%ebp)
       movl      -4(%ebp), %eax
       movl      %eax, -20(%ebp)
       flds      -20(%ebp)
       leave
       .cfi_restore 5
       .cfi_def_cfa 4, 4
       ret
       .cfi_endproc
 .LFE0:
       .size     add, .-add
       .globl    add_and_multiply
       .type     add_and_multiply, @function
```

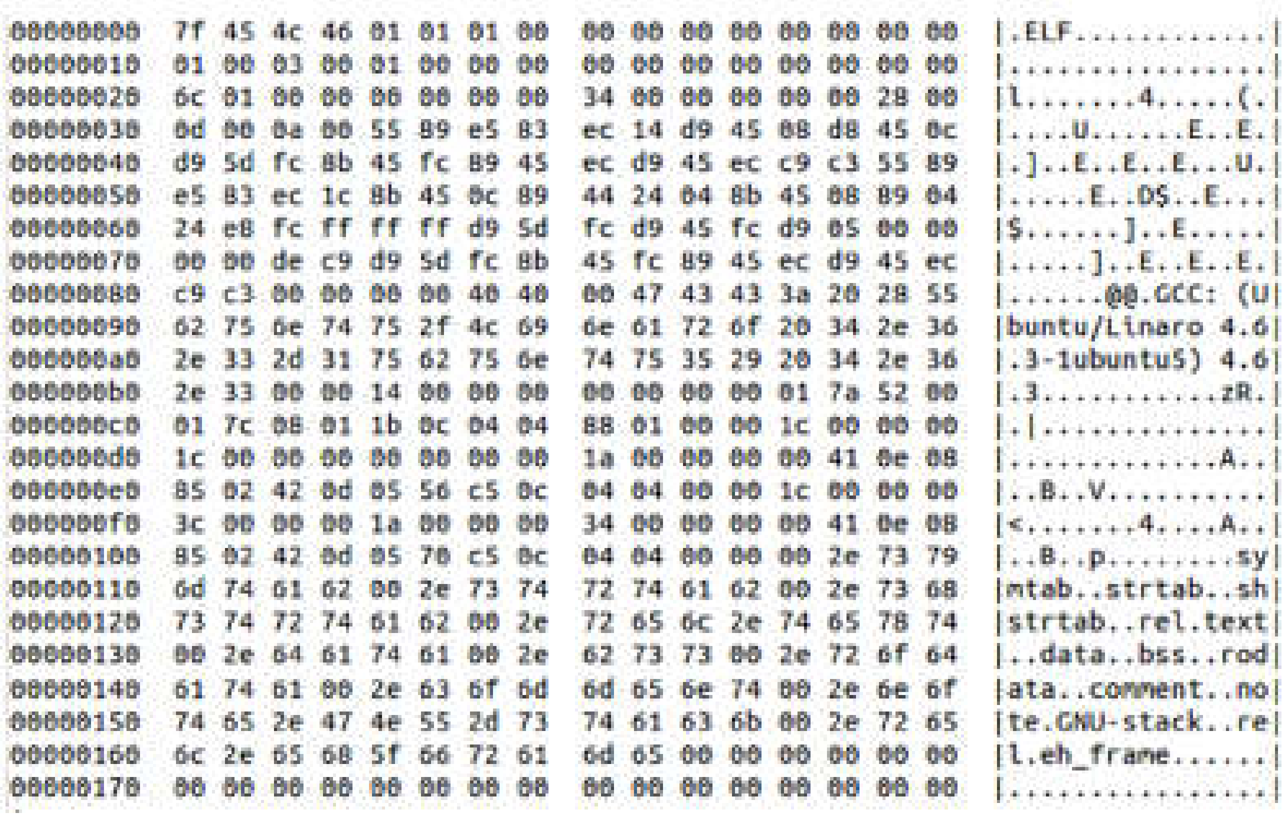
```
 add_and_multiply:
 .LFB1:
       .cfi_startproc
       pushl     %ebp
       .cfi_def_cfa_offset 8
       .cfi_offset 5, -8
       movl      %esp, %ebp
       .cfi_def_cfa_register 5
       subl      $28, %esp
       movl      12(%ebp), %eax
       movl      %eax, 4(%esp)
       movl      8(%ebp), %eax
       movl      %eax, (%esp)
       call      add
       fstps     -4(%ebp)
       flds      -4(%ebp)
       flds      .LC1
       fmulp     %st, %st(1)
       fstps     -4(%ebp)
       movl      -4(%ebp), %eax
       movl      %eax, -20(%ebp)
       flds      -20(%ebp)
       leave
       .cfi_restore 5
       .cfi_def_cfa 4, 4
       ret
       .cfi_endproc


.LFE1:
      .size     add_and_multiply, .-add_and_multiply
      .section          .rodata
      .align 4
.LC1:
      .long     1077936128
      .ident    "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
      .section          .note.GNU-stack,"",@progbits
```

Compiling: 4) Optimization

- Once the first assembler version corresponding to the original source code is created, the optimization effort starts, in which usage of the registers is minimized.
- Additionally, the analysis may indicate that certain parts of the code do not in fact need to be executed, and such parts of the code are eliminated.

Compiling:

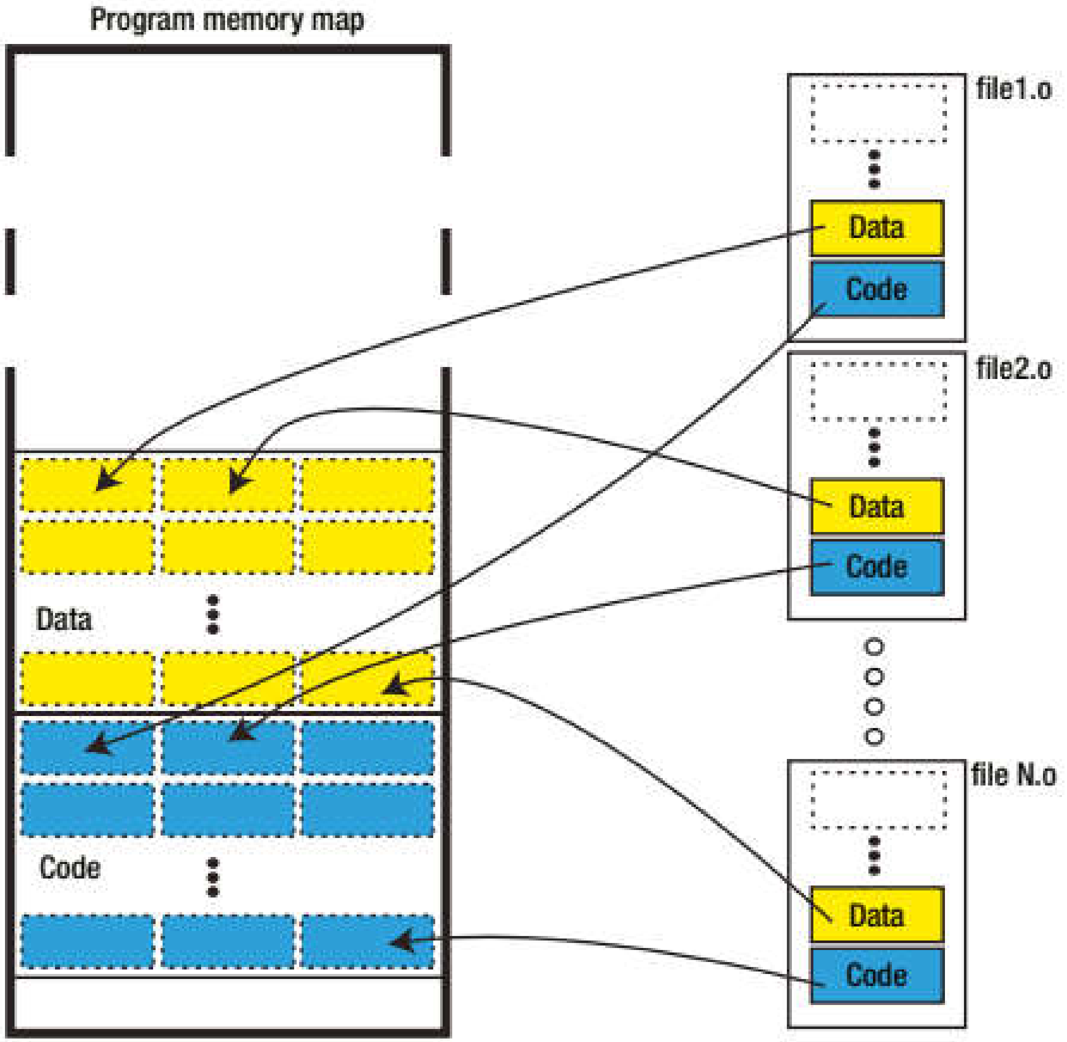
- The gcc could perform the complete compilation that generate the binary object file (standard extension .o)
- Binary contents of an object file:



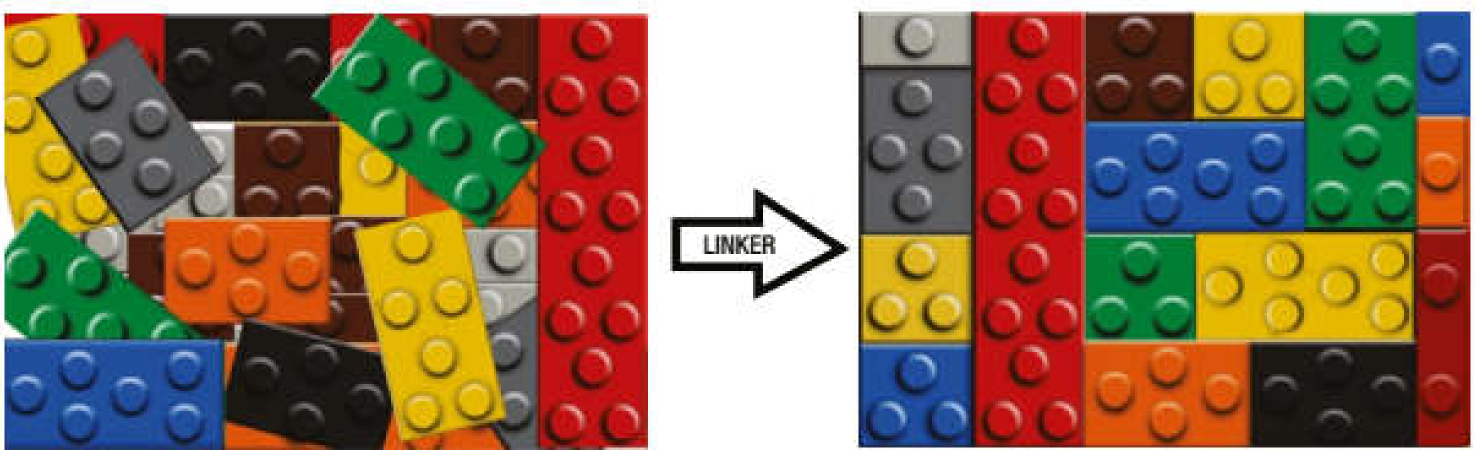
```
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  01 00 03 00 01 00 00 00  00 00 00 00 00 00 00 00  |................|
00000020  6c 01 00 00 00 00 00 00  34 00 00 00 00 00 28 00  |l.......4.....(.|
00000030  0d 00 0a 00 55 89 e5 83  ec 14 d9 45 08 d8 45 0c  |....U......E..E.|
00000040  d9 5d fc 8b 45 fc 89 45  ec d9 45 ec c9 c3 55 89  |.]..E..E..E...U.|
00000050  e5 83 ec 1c 8b 45 0c 89  44 24 04 8b 45 08 89 04  |.....E..D$..E...|
00000060  24 e8 fc ff ff ff d9 5d  fc d9 45 fc d9 05 00 00  |$......]..E.....|
00000070  00 00 de c9 d9 5d fc 8b  45 fc 89 45 ec d9 45 ec  |.....]..E..E..E.|
00000080  c9 c3 00 00 00 00 40 40  00 47 43 43 3a 20 28 55  |......@@.GCC: (U|
00000090  62 75 6e 74 75 2f 4c 69  6e 61 72 6f 20 34 2e 36  |buntu/Linaro 4.6|
000000a0  2e 33 2d 31 75 62 75 6e  74 75 35 29 20 34 2e 36  |.3-1ubuntu5) 4.6|
000000b0  2e 33 00 00 14 00 00 00  00 00 00 00 01 7a 52 00  |.3...........zR.|
000000c0  01 7c 08 01 1b 0c 04 04  88 01 00 00 1c 00 00 00  |.|..............|
000000d0  1c 00 00 00 00 00 00 00  1a 00 00 00 00 41 0e 08  |.............A..|
000000e0  85 02 42 0d 05 56 c5 0c  04 04 00 00 1c 00 00 00  |..B..V..........|
000000f0  3c 00 00 00 1a 00 00 00  34 00 00 00 00 41 0e 08  |<.......4....A..|
00000100  85 02 42 0d 05 70 c5 0c  04 04 00 00 00 2e 73 79  |..B..p........sy|
00000110  6d 74 61 62 00 2e 73 74  72 74 61 62 00 2e 73 68  |mtab..strtab..sh|
00000120  73 74 72 74 61 62 00 2e  72 65 6c 2e 74 65 78 74  |strtab..rel.text|
00000130  00 2e 64 61 74 61 00 2e  62 73 73 00 2e 72 6f 64  |..data..bss..rod|
00000140  61 74 61 00 2e 63 6f 6d  6d 65 6e 74 00 2e 6e 6f  |ata..comment..no|
00000150  74 65 2e 47 4e 55 2d 73  74 61 63 6b 00 2e 72 65  |te.GNU-stack..re|
00000160  6c 2e 65 68 5f 66 72 61  6d 65 00 00 00 00 00 00  |l.eh_frame......|
00000170  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

Compilation Process Limitations

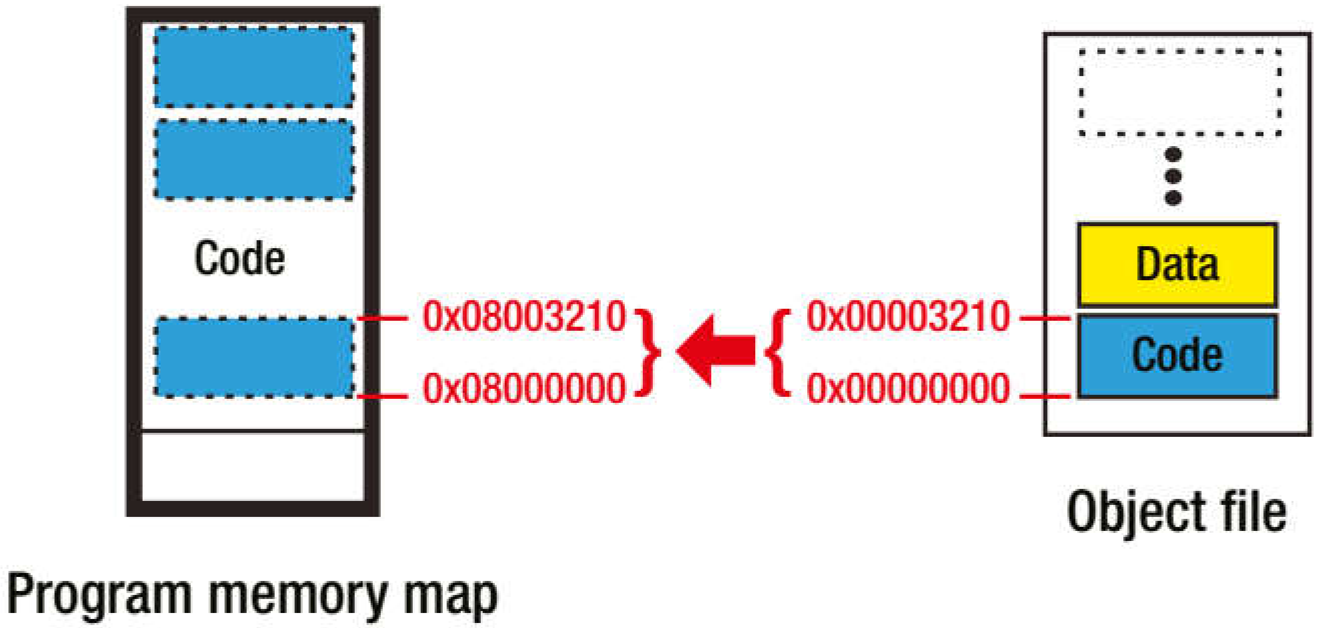
- The compilation process translates the ASCII source files into the corresponding collection of binary object files.
- Each of the object files contains sections, the destiny of each is to ultimately become a part of gigantic puzzle of the program's memory map



Need a linker

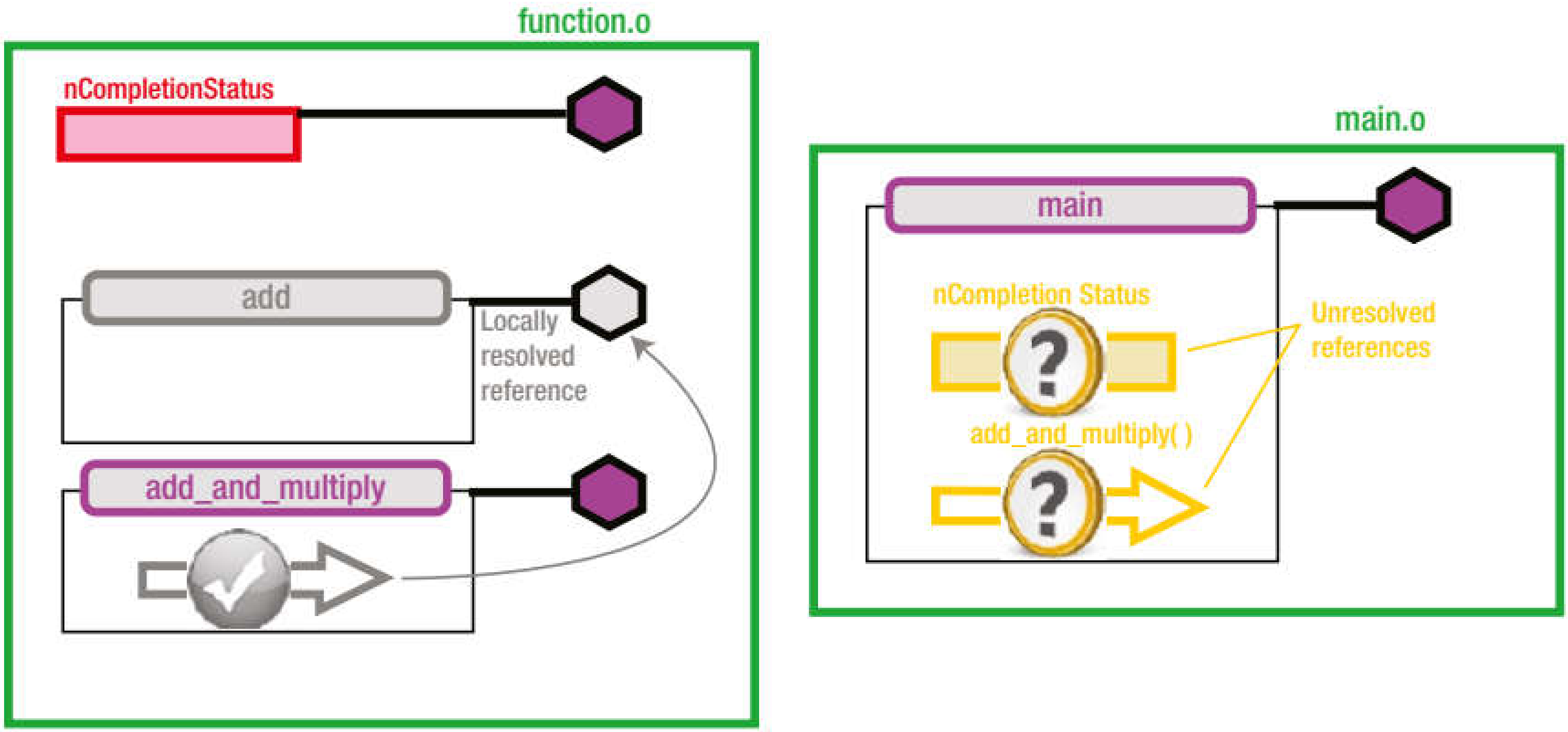


Linking stages: 1) Relocation

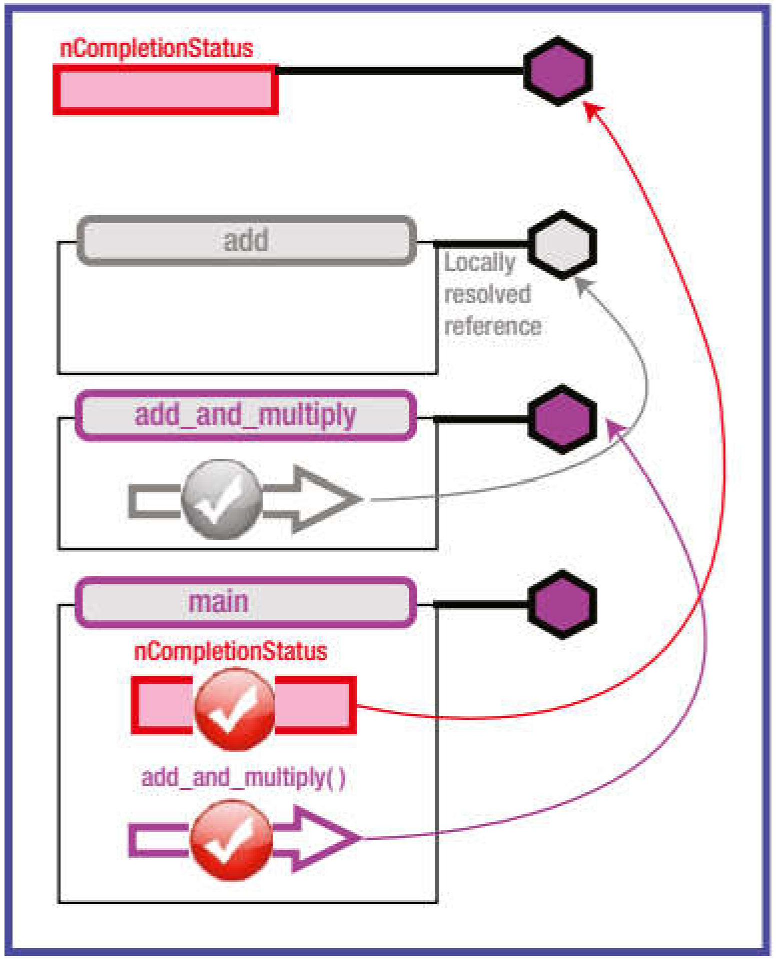


Resolving References

The problem of unresolved references in its essential form

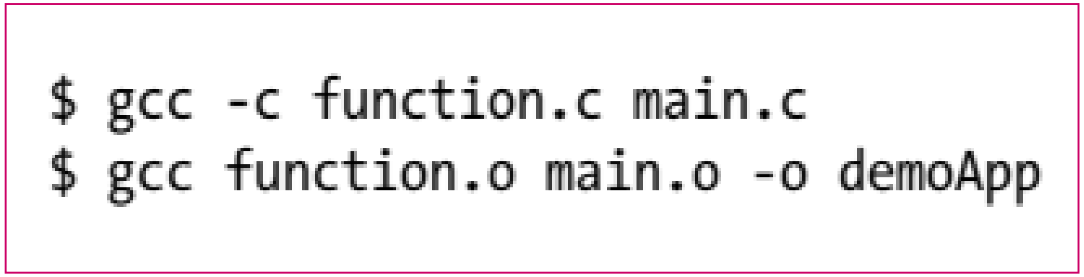


Resolved References



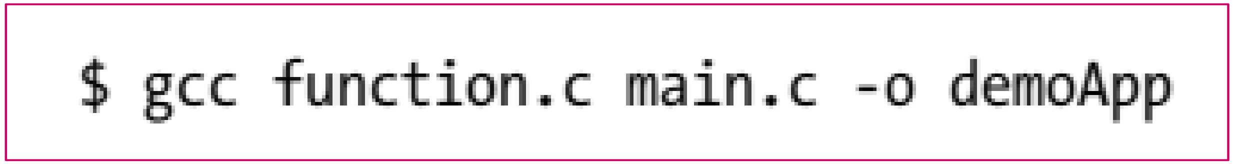
Example

- In the step-by-step approach, you will first invoke the compiler on both of the source files to produce the object files. In the subsequent step, you will link both object files into the output executable



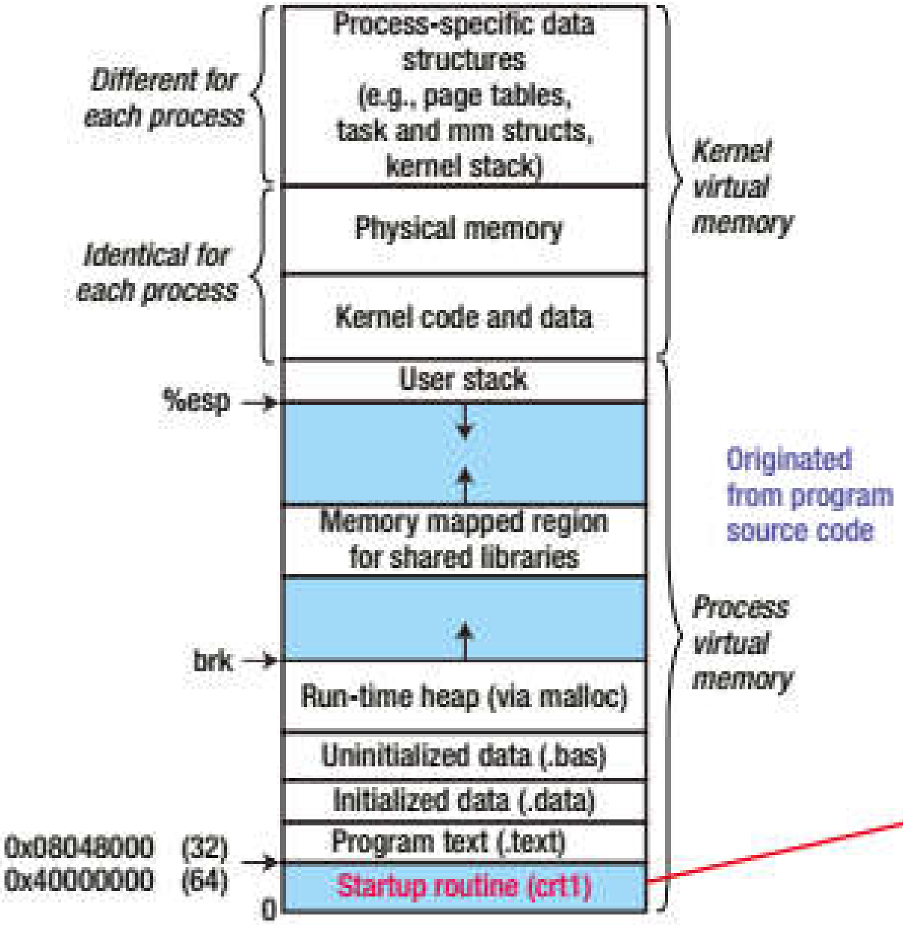
```
$ gcc -c function.c main.c
$ gcc function.o main.o -o demoApp
```

- In the all-at-once approach, the same operation may be completed by invoking the compiler and linker with just one command.



```
$ gcc function.c main.c -o demoApp
```

Overall structure of an executable file



Different for each process

Identical for each process

Process-specific data structures (e.g., page tables, task and mm structs, kernel stack)

Physical memory

Kernel code and data

Kernel virtual memory

%esp

User stack

Memory mapped region for shared libraries

brk

Run-time heap (via malloc)

Uninitialized data (.bas)

Initialized data (.data)

Program text (.text)

Startup routine (crt1)

0x08048000 (32)

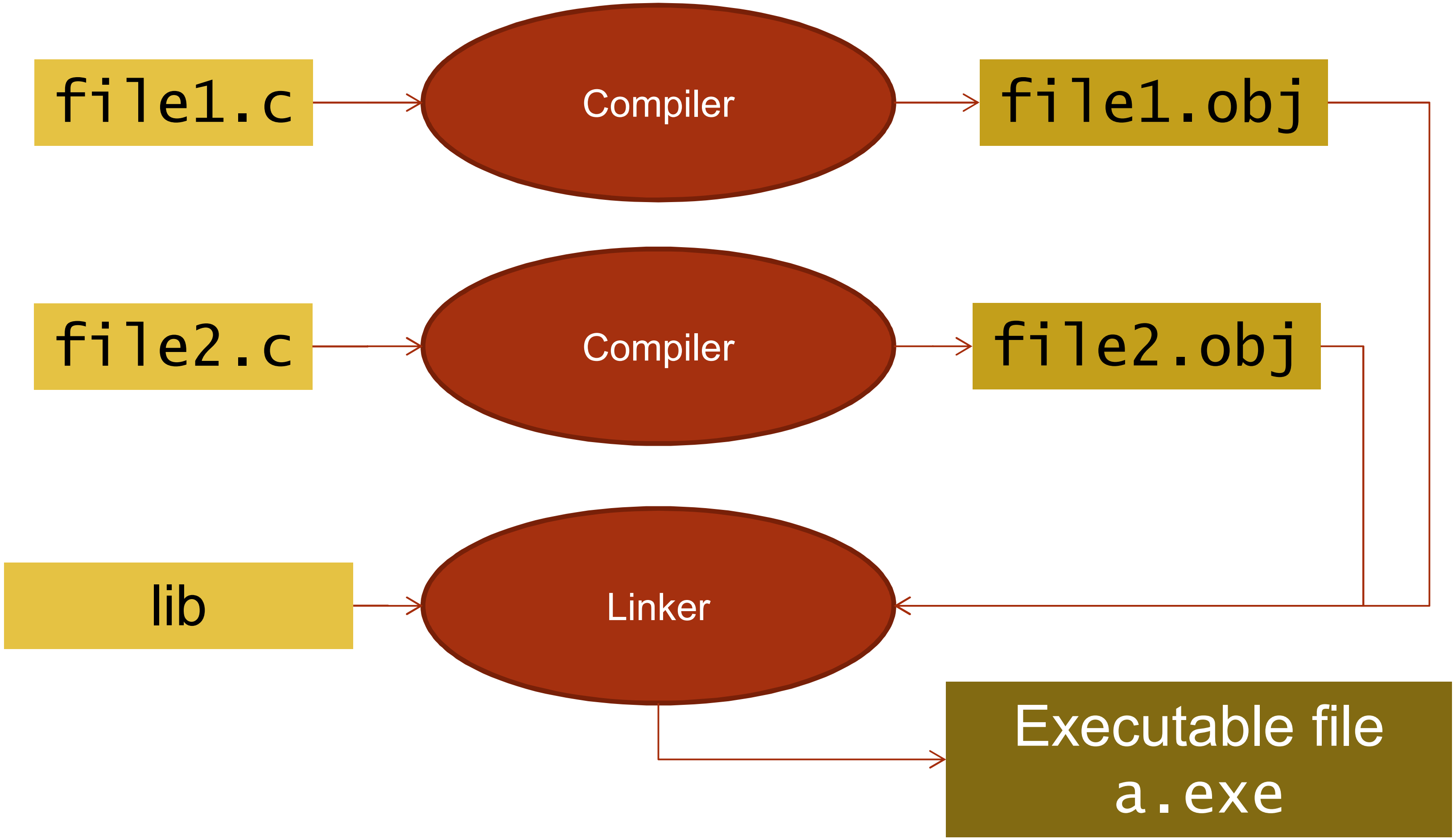
0x40000000 (64)

0

Process virtual memory

Originated from program source code

The whole process



C/C++ compilers

- **Allows to translate each file separately to help:**
  - Easy to divide and manage each part of the program
  - When it is necessary to make changes, just modify the associated file
    - reduce maintenance and modification time
  - Just re-translate files with changes as needed
    - Reduce translation time
- **Modern compilers also allow optimization of data and code**
- **Some common compilers: MS Visual C ++, gcc, Intel C ++ Compiler, Watcom C / C ++, ...**

Notice

- The syntax is case sensitive: int, Int, INT are completely different
- “;” is used to separate single statements
- The {...} sign is used to specify a statement block
- Do not name the variable / constant / function ... to match the keyword (void, int, char, struct, const, ...)
- In a block of statements with no oriented structure (if, for, while, ...), the statements will be executed sequentially from top to bottom.
- Comments:
  - in C is equal to: / *… * /
  - in C ++ there is an extra symbol // to comment to the end of the line