LECTURE NOTES IN CIS342

YUZHE TANG, AMIN FALLAHI

SPRING, 2017

SECTION 2: PROGRAMMING IN C/C++

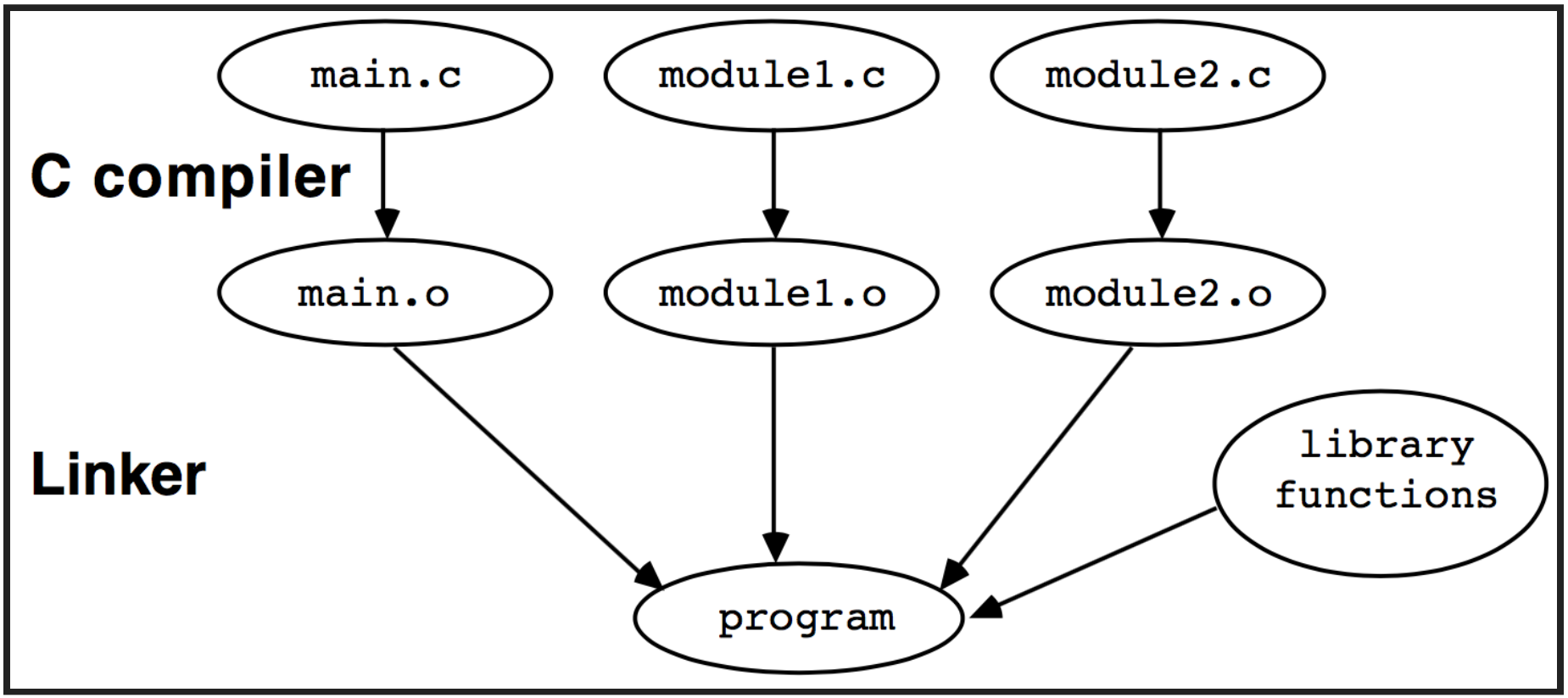
GCC & MAKE (MAR. W4)

REFERENCES

- "Unix Programming Tools", [link]
- Computer Systems: A Programmer's Perspective, Randal E. Bryant and David R. O'Hallaron, Chapter 1, [online pdf]
- Source files: [src directory]

COMPILATION OVERVIEW

- Two steps of compilation:
  - *compiling*: text `.c` file to relocatable `.o` (object) file
  - *linking*: multiple relocatable `.o` files to one executable `.o` file
    - *symbol*: reference to link construct (declaration) in one `.o` file to construct (definition) in another `.o` file


main.c
module1.c
module2.c
C compiler
main.o
module1.o
module2.o
Linker
library
functions
program

Linker

C: BASICS

```c
#include <stdio.h> //preprocessor
int y = 3; //global var. (def. & init.)
//extern int y; //global var. (dec.)
int main() //function (def.)
{
    int x = 0; //local var. (def. & init.), literal,
    printf("y = %d\n",y); //function (invocation)
    return 0;
}
```

LIFE OF A C CONSTRUCT

	variable	function
declare	`extern int x;`	`void foo();`
define	`int x;`	`void foo(){}`
initialize	`int x = 6;`	
reference	`y = x;x=1;`	`foo();` (invocation)
destroy		

GCC: FLAGS

- `-c` for compile, `-o` for output; demo
- `-Wall`, w for warning; demo
- `-I` for `#include`; demo
  - header file (storing declarations)
- `-Ldir`/`-lmylib` for library to link
  - search library for unsolved symbols (functions, global variables) when linking
- `-g` for debug (later)
- ref [link]

MAKE

```
all: link
\t./a.out

linklib: compilelib
\tgcc h1.o -L. -lx #-L. is necessary

compilelib: compile
\tmv h2.o libx.a

link: compile
\tgcc h1.o h2.o

compile:
\tgcc -c h1.c
\tgcc -c h2.c
```

```
SRCS = h1.c h2.c
OBJS = $(SRCS:.c=.o)

all: link
\t./a.out

link: $(OBJS)
\t$(CC) $(OBJS)
```

MAKEFILE: VARIABLES

- variable represents strings of text
- standard variable: `CC`, `CFLAGS`, `LDFLAGS`
  - `LDFLAGS` library search path (`-L`)
  - `OBJS = $(SRCS:.c=.o)`:
    - This incantation says that the object files have the same name as the .c files, but with .o

MAKEFILE: DEPENDENCY RULES

- dependency rules: tells how to make a target based on changes to a list of certain files.
- If-this-then-that
  - dependency line: a trigger that says when to do something
  - command line: specifies what to do

GDB (MAR. W5)

REFERENCES

- "Reviewing gcc, make, gdb, and Linux Editors", [pdf]
- "Unix Programming Tools", [link]

A BUGGY PROGRAM

```c
#include<stdio.h> //printf
int main(){
  int array_stack[] = {0,1,2};
  int sum = 0;
  for(int i=0; i<=3; i++){
    sum += array_stack[i];
  }
  printf("sum = %d\n", sum);
  return 0;
}
```

USE GDB TO FIND BUG

- Installing gdb
- Compile: `gcc -g`
- Run gdb: `gdb a.out`

GDB COMMAND: CONTROL EXECUTION

- C execution model
- breakpoints
  - `break/b file:n|fn|file:fn`
  - `disable/enable/delete bkpt`: `bkpt=file:n|fn|file:fn`
- stepping
  - `run/r`: run
  - `next/n`: next statement (step over a function call)
  - `continue/c`: continue till breakpoint

GDB COMMAND: EXAMINE RUNTIME

- examine runtime data
  - `print v`/`p v`: print variable `v`
- examine code (with `gcc -g`)
  - `list`/`l`
- examine execution environment: e.g. stack (later)

GDB FUNCTIONALITY

functionality	commands
breakpoints	`b`,`disable/enable/delete breakpoi`
stepping	`r`,`s`,`n`,`c`,`finish`,`return`
examine_data	`p/i v`,`display/undisplay`,`watch`,`set`
examine_code	`list`
examine_stack	`bt`,`where`,`info`,`up/down`,`frame`
misc.	`editmode vi`, `b fn if expression`, `h` disassembler, `shell cmd`

DEMO & EXERCISE

- Exercise: Debug the following program using gdb, upload the correct program to BB.

```c
#include<stdio.h> //printf
int main(){
  int a1[] = {2,1,0};
  int sum = 0;
  int i;
  for(i=0; i<=2; i++){
    sum += a1[i]/a1[2-i];
  }
  printf("sum = %d\n", sum);
  return 0;
}
```

POINTER IN C (APR. W1)

REFERENCES

- Pointer Basics: [http://cslibrary.stanford.edu/106/]

POINTER (C SYNTAX)

- a pointer is a variable that stores a reference to something.
  - "something", called pointee, is usually another variable.
- e.g.: a pointer variable named `x` referencing to a "pointee" variable of value `42`.

pointer pointee

POINTER OPERATIONS

- definition/initialization: `int *p1 = p2;`
- dereference: `*p`
- get reference of: `& a`
  - get the *address* (memory location) of variable `a`

```c
#include<stdio.h>
int main(){
    int a = 10;
    int * p = & a;
    int b = *p;
    printf("a=%d,b=%d,*p=%d,p=%p\n",a,b,*p,p);
}
```

LIFE OF A C POINTER/SYMBOL

	pointer	variable	functic
declare	`extern int * p`	`extern int x`	`void`
define	`int *p;`	`int x`	`void`
initialize	`int *p=&a;`	`int x=6`	
	`int*q=malloc(7)`		
reference	`*p=x;x=*p`	`y=x`	`foo()`
destroy	`delete p`		

EXERCISE

- Do the following to complete the code snippet at the bottom. Then compile and execute your program. Submit the completed program to BB.
  1. define two pointers `p1` and `p2`, both pointing to variable `x`.
  2. Use `p1` to update `x`'s value to `5`.
  3. Then use `p2` to read the value of variable `x` and `printf` it on terminal.

```c
#include<stdio.h>
int main(){
    int x = 4;
    // To complete the program below:


}
```

VIRTUAL MEMORY IN C (APR. W1)

INTRODUCTION

- we talked about pointers
- but **where does a pointer point to?**
- this is related to the *virtual memory* model in C/C++.
- References
  - "Using GNU's GDB Debugger: Memory Layout And The Stack", by Peter Jay Salzman [link]

FOUR VARIABLE "TYPES" IN C

1. *global* variable: defined outside a function
2. *local* variable: defined inside a function
3. *dynamically-allocated* variable: allocated by `malloc()`
   - `int * p = malloc(2*sizeof(int));`
4. *static (local)* variable: defined inside a function, with keyword `static`
   - `static int x;`, [example]

```c
#include<stdio.h>
#include<stdlib.h>
// global variable x
int x = 1;
void foo(){
  static int t = 5;
  printf("t in foo = %d\n", t);
}

int main(){
  // local variable y
  int y = 2;
  // dynamically-allocated variable pz
  int * pz = malloc(2*sizeof(int));
  // static local variable t
  static int t = 4;
```

VARIABLE TYPE: SCOPE AND VISIBILITY

variable type	scope	visibility	memory location
global variable	global	global	`.rodata`/`.bss`
static variable	global	nested local	`.rodata`/`.bss`
local variable	local	nested local	`stack`
dynamically-allocated var	dynamic	global	`heap`

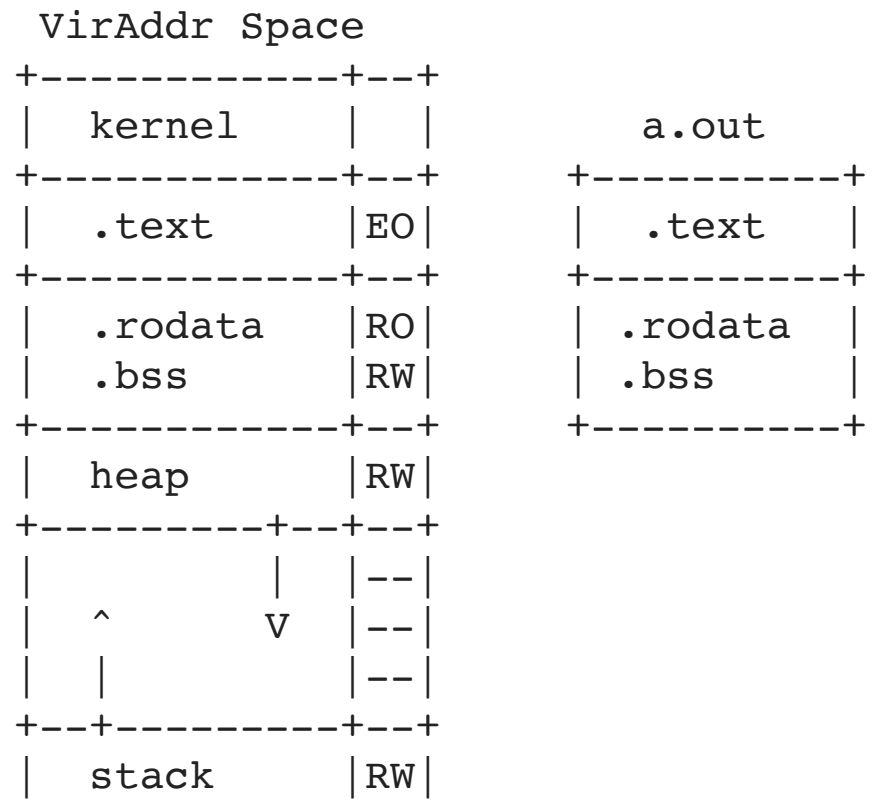
- static local variable
  - Possible to define multiple static local variables of the same name, defined in different functions.
  - They represent different memory locations.
- The case of the preceding code.

VIRTUAL MEMORY LAYOUT

1. Segments `.text`, `.bss`, `.rodata`
   - From executable `a.out`
2. Segments `stack`, `heap`
   - Runtime info.: values of symbols during execution

- Each segment (or page) has its own permission.

VIRTUAL MEMORY LAYOUT



- Demo: print memory layout in Gdb
  - `info proc mappings #print mem layout`

EXAMINING STACK

- stack: frames, local variables, arguments, return
  - `backtrace`/`bt`/`bt full`,`where`
  - `up`,`down`,`frame #`
    - **context**

EXERCISE 1

- Use gdb to debug the following program. Identify the bug line.

```
int main() {
     void * p = "x";
     *p = 'y';}
```

EXERCISE 2

- Use gdb to debug the following program. Identify the bug line.

```
#include<stdio.h>
int main() {
    double x[10000000]; //vs x[1000];?
    x[13]=1; printf("%f\n",x[13]);}
```

ASSEMBLY LANGUAGE (APR.W2)

REFERENCES

- Hacking, the art of exploitation, 2nd edition, Jon Erickson, Chapter 0x250

ARCHITECTURE OVERVIEW

- x86 CPU registers
  - RAX,RBX,RCX,RDX: values
  - RSP,RBP,RSI,RDI: pointers
  - **RIP**
  - (EFLAG)
- demo in gdb:
  - `info registers #print register values`
  - `i r rip #print register rip`

- CPU execution model
  - like "a child pointing his finger at each word as he reads"
  - RIP, or PC, is the CPU's finger
  - words are instructions stored in section `.text` in virtual memory
    - machine instructions
    - assembly

ASSEMBLY PROGRAMS: WHERE TO FIND THEM

- program life cycle
  - preprocessor:
  - compiler: assembly code
  - assembler: machine instruction (PIC)
  - linker: machine instruction (executable)
  - loader: loaded in virtual addr space

```
   prep      gcc -S      gcc -c           ld/gcc        exec
.c------>.i-------->.s-------->.o(PIC)------->a.out------>VM
```

- Three places to examine assembly code
  - `gcc -S helloworld.c; vim helloworld.s`
  - `objdump -M intel -D a.out | grep -A20 main`
  - in `gdb`:
    `set disassembly intel; disassemble main`

ASSEMBLY LANGUAGE

- ISA:
  - ALU instruction: `add`
  - LD/ST instruction: `mov`
  - Control instruction: `jump`, `cmp; jle foo`
- Format: AT&T and Intel
  - AT&T: `89 e5    mov %rsp,%rbp`
  - Intel:`89 e5    mov rbp rsp`

DEMO: CPU EXECUTION IN ACTION

```c
#include <stdio.h>
int j = 3;
int main() {
  int i = 0;
  i = i + 2;
  if (i == 2)
    puts("Hello, world!\n"); // put the string to the output.
}
```

ASSEMBLY IN GDB

- `nexti`: step per instruction
- `x`: gdb command to examine memory
  - `x/3xb $rip`
  - `x/x`: `o/x/u/t`
    - `t` binary, `u` unsigned, `o` octal
  - `x/3b`: `b/h/w/g`
    - `b` byte, `h` halfword, `w` word, `g` giant
  - `x/4i`
    - `i` instruction

C VARIABLES, FUNCTIONS IN ASSEMBLY (APR.W3)

REFERENCES

- Hacking, the art of exploitation, 2nd edition, Jon Erickson, Chapter 0x260,270,280
- Smashing The Stack For Fun And Profit, [link]

DATA TYPE

- C language is typed, but assembly/machine instructions are not
- C data type determines:
  - how much space to allocate to store variable in memory
  - how to interpret the bit-string stored in the memory
  - how to calculate "primitive" arithmetic on the variable

- signed, unsigned, long long, float, char
  - unsigned:
    - a 32-bit unsigned integer, value from 0 to $2^{32} - 1$.
  - signed:
    - a 32-bit unsigned integer, value from -2^{31} to $2^{32} - 1$.
    - negative numbers are represented by two's complement, (which is suited for binary adders).

type	signed	unsigned	short	long long	float	char
`sizeof()`	4	4	2	8	4	1

TYPECASTING

```c
#include<stdio.h>
int main(){
  int i = 5;
  float f = (float) i;
  float d = f/3; // float d2 = i/3;
  printf("%f\n",d);
}
```

POINTER TYPES

- Array
  - A `C` array is a list of `n` elements of a specific data type, and allocated in `n` adjacent memory locations.
  - A null byte in the end is a delimiter character
- Array and pointer
  - `int a[];` vs `int *b = a;`

```c
#include<stdio.h>
int main(){
    int a[] = {2,1,0};
    int *b = a;
    unsigned long c = (unsigned long)a;//long
    for (int i=0; i<3; i++){
        printf("%d,%d,%d,%d,%d\n",a[i],*(b+i),*(a+i),b[i],*((int *)(c
    }
}
```

- Code-pointer: function pointer
- Data-pointer: pointer to variable, array
  - Pointer data type: `char *`, `int *`,
  - Pointer arithmetic: equiv. code
    - `int * p = array; p += 1;`
    - `int pp = array; pp += sizeof(int);`

FUNCTION EXECUTION AND STACK (APR.W3)

STACK

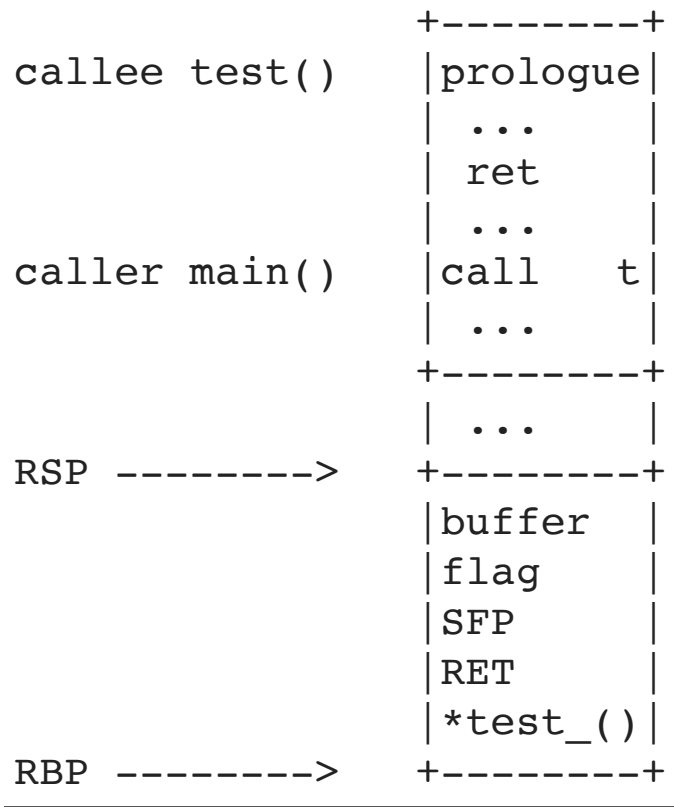
- Stack:
  - store **context** information
  - a stack of frames, with the top frame pushed (popped) by entering (leaving) a function
- Stack pointer: `RSP`, pointing to the stack end
- Frame pointer: `RBP`, pointing to the start of top frame.

ENTERING FUNCTION IN ASSEMBLY

- Calling convention, function prologue and `call` instruction.
- `SFP` is for restoring `RBP` to previous value, and the return address is used to restore `RIP`

- in gdb, `b test`, `x/16xw $RSP`

```c
void test(int a, int b, int c, int d) {
  int flag;
  char buffer[10];
  flag = 31337;
  buffer[0] = 'A';
}
int main() {
  test(1, 2, 3, 4);
}
```



COMMANDLINE ARGUMENT

```c
#include<stdio.h>
int main(int argc, char *argv[]) {
  if(argc < 2) return 1;
  printf("arg is %s\n", argv[1]); //%s for char array
}
```

```
./a.out helloworld
```

EXERCISE

- Debug the following program using `gdb`:
  - in Ubuntu, compile the program by `gcc -g auth.c -fno-stack-protector`
  - `strcpy`/`strcmp` copies/compares the two string arguments
  - find a commandline argument that is not password but passes the authentication.
  - find the bug and describe what the bug is in BB.
  - `gdb --args a.out ARGS` launches `a.out` with `ARGS`.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int check_authentication(char *password) {
  int auth_flag = 0;
```

```
  char password_buffer[16];

  strcpy(password_buffer, password);
  if(strcmp(password_buffer,"12345")==0)auth_flag=1;
  if(strcmp(password_buffer,"54321")==0)auth_flag=1;
  return auth_flag;
}
int main(int argc, char *argv[]) {
  if(argc < 2) exit(0);
  if(check_authentication(argv[1])) {
    printf("\nAccess Granted.\n");
  } else {
```

C++: OBJECT-ORIENTED PROGRAMMING (APR.W4)

REFERENCES

- The C++ Language Tutorial [link], Chapter "Object oriented programming, Classes (I)"

COMPOUND DATA TYPE: STRUCT

- `struct` is compound data type
- `struct` is custom data type
  - define your own `struct`

```c
#include<stdio.h>

struct movies_t {
  char * title;
  int year;
};
int main (){
  struct movies_t amovie;
  struct movies_t * pmovie;
  pmovie = &amovie;
  amovie.title = "the usual suspects";
  pmovie->year = 1995;
  printf("Movie %s is released in year %d\n", pmovie->title, pmovie->
  return 0;
}
```

CLASS

	compound type	class
declare	`struct sname {mvar} iname;`	`class cna`
define	N/A	`cname::mf`

- class is an expanded, compound data type, in that:
  1. "class holds both data and function", while compound data type holds data
  2. class defines access policy

```
 class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area (void);
} rect;
```

OBJECT

- An object is an instance of class.

life span	var	object
declare	`extern int x;`	`class cname {} objnam`
define (pointer)	`int x;`	`cname * objname`
initialize	`int x=5;`	`cname * objname = new`
reference	`x=3;`	`objname->foo();`
destroy		`delete objname;`

- Private members of a class are referenced only from within other members of the same class.
  - reference is between a pointer and pointee.
  - access policy (`public` or `private`) is about the permission of referencing pointee given the context of the pointer.

DEMO

```
// classes example
#include <stdio.h>
class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area () {return (x*y);}
};
void CRectangle::set_values (int a, int b) { x = a;
y = b; }
int main () {
  CRectangle rect;
  rect.set_values (3,4);
  printf("area: %d\n", rect.area());
  return 0;
}
```

```
g++ demo2.cpp
```

SYSTEM CALL (APR.W4)

REVISIT HELLOWORLD

```c
#include<stdio.h>
int main(){
   printf("hello world\n");
}
```

HOMEBREW **PRINTF**

```
void print() {
  char * message = "hello world";
  asm ( "mov $12, %%rdx\n\t"
      "mov %0, %%rcx\n\t"
      "mov $0, %%rbx\n\t"
      "mov $4, %%rax\n\t"
      "int $0x80 \n\t"
      ::"r"(message):"rdx","rcx","rbx");
}
int main(){
   print("hello world\n");
}
```

SYSTEM CALL

- `int` represents interrupt to CPU
- Syscall is one kind of interrupt
  - syscall interrupts CPU's normal execution and forces it to jump to a kernel routine and run it under kernel mode.
  - CPU will save the context information to the kernel-space memory.