CS 211
Computer Architecture

Instructor: Prof. Brian Russell

TAs

Bhrushundi, Abhishek	abhishek.bhr@rutgers.edu
Bronner, Eric	eric.bronner@rutgers.edu
Soltaniyeh, Mohammadreza	ms2344@rutgers.edu
Gang Qiao	gq19@cs.rutgers.edu

Syllabus

1. Von Neumann Architecture, Hardware trends, Importance of Speed, Cost, Energy
2. Intro to C programming
3. Data Representation, Computer Arithmetic
4. Assembly language techniques, including macro-instruction definition
5. Digital logic, registers, instruction counter
6. Processor Architecture
7. Pipelining
8. Memory hierarchies, Caching (L0, L1, L2 caches)
9. Virtual Memory
10. Interrupts
11. Input and Output, buses

Textbook

Computer Systems, A Programmer's Perspective
by Bryant and O'Hallaron

C brief review

[1] Data types and Constant

(1.1) Numeric integers:

Decimal: 123, 123L (‘L’ denotes Long)

Hexadecimal: {0~9, A, B, C, D, E, F}   int x=0x18; // Hex.   It’s decimal value is 24

Octal: {0-7}   int x=016;   // Octal. It’s decimal value is 14

Binary: {0, 1}   10011101

(1.2) Char and String

```
char c1;
c1 = ‘a’      //c1 is a char


char* c2;  //you don’t need to assign the length when declaring a char*
c2 = “a”;  //c2 is a string
           //in memory, it takes 2 bytes:
```

a	\0

```
           //But its length is 1 using strlen()
```

[1] Data types and Constant

(1.3) size_t

Question: What is Type size_t in C?
The size_t type is the **unsigned integer type** that is the **result of the sizeof operator**.

It is the alias of one of the fundamental unsigned integer types while its actual type is platform dependent for maximizing performance because it's typedef'd to be a specific unsigned integer type that's **big enough -- but not too big -- to represent the size of the largest possible object on the target platform.**

This also **allows portability** among different platforms.

[1] Data types and Constant

(1.4) const qualifier

Case1: *p is constant, p could change

```
const char* p
char  const* p
```

Case2: p is constant, *p could change

```
char* const  p
(char*) const p
const (char*) p
```

Case3: both p and *p are constant

```
const char* const p
char const* const p
```

main purpose: data isolation

Declaration:

```
bool func(const char* p)
{
  char a = *p;     //read content only
  if (a> 10)
    return True;
   else
    return False;
}
```

Calling:

```
char* p0 = malloc(1);
bool   b0 = func(p0);
```

[1] Data types and Constant

(1.5) structure **Purpose: express abstract data object via encapsulation**

```
struct TokenizerT_
{
  member list      //members are attributes of the structure object
};                 //members could be integer, float, pointer or even a structure
```

```
typedef struct TokenizerT_ Tokenizer;
Tokenizer token;
Token.member = 5;
```

```
Tokenizer* pTokenizerT;
pTokenizerT->member = 5;
```

⇒

```
Tokenizer* pTokenizerT;
pTokenizerT = &token;
pTokenizerT->member = 5;
```

[1] Data types and Constant

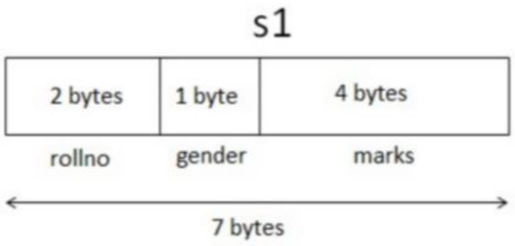
(1.6) Union: similar to structure **What is the difference between structure and union?**

Struct

The memory a struct var occupies is the sum of memory that all its members occupy;

After the assginments, all members are stored;

```
struct student
{
        int rollno;
        char gender;
        float marks;
}s1;

struct student* p = &s1;

p-> rollno = 1;
p-> gender = 'F';
p-> marks = 46;

sizeof(student) = sizeof(int)+sizeof(char)+sizeof(float)
```



Memory Allocation in Structure

Union

The memory a union var occupies is the memory that its longest member occupies;

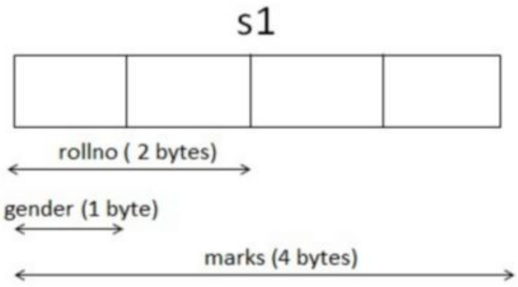
After assginments, only the last member is stored. All previous members are overridden;

```
union student
{
        int rollno;
        char gender;
        float marks;
}s1;
```

union

~~struct~~ student* p = &s1;

```
p-> rollno = 2;      (the value of rollno will be overridden)
p-> gender = 'M';    (the value of gender will be overridden)
p-> marks = 20;      (the value of marks is stored)

sizeof(student) = max{ sizeof(int), sizeof(char), sizeof(float) }
```



Memory Allocation in Union

[1] Data types and Constant

(1.7) Enumeration type

Purpose: refer to a countable set (e.g., the set of events, which might be complex and abstract) by 0, 1, 2 (0, 1, 2 are just codes for members of the set, without the meaning of numeric values)

```
enum
{
    CV_EVENT_MOUSEMOVE       =0,
    CV_EVENT_LBUTTONDOWN     =1,
    CV_EVENT_RBUTTONDOWN     =2,
    CV_EVENT_MBUTTONDOWN     =3,
    CV_EVENT_LBUTTONUP       =4,
    CV_EVENT_RBUTTONUP       =5,
    CV_EVENT_MBUTTONUP       =6,
    CV_EVENT_LBUTTONDBLCLK   =7,
    CV_EVENT_RBUTTONDBLCLK   =8,
    CV_EVENT_MBUTTONDBLCLK   =9
};
```

//callback function is triggered once some event occurs
//argument list of callback function is fixed and parameters are passed by system/library function, though the function name and its implementation is user-defined

```
//define callback function
void my_MouseRectangle_callback( int event, int x, int y, int flags, void* param )
{
    IplImage* pimage = ( IplImage* ) param;
    switch( event )
    {
        case CV_EVENT_MOUSEMOVE:
        {
            if( drawing_box )
            {
                /* resize the box using current mouse position */
                box.width  = x - box.x;
                box.height = y - box.y;
            }
        }
        break;

        case CV_EVENT_LBUTTONDOWN:
        {
            drawing_box = true;
            box = cvRect(x, y, 0, 0);
        }
        break;
```

```
        case CV_EVENT_LBUTTONUP:
        {
            drawing_box = false;

            //dealing with the th
            if( box.width < 0 )
            {
                box.x += box.widt
                box.width *= -1;
            }

            //dealing with the th
```

[1] Data types and Constant

(1.8) Functions (**functions are also data!**)

```
int func (int A, int B)
{
  if (B ==0)
       return A;
  else
       return func(B, A%B);
 }
```

```
int main(int argc, char** argv)
{
   int i;
   for(i = 0; i < argc; i++)
   {
     printf( “command line argv[%d] is %s length %d\n”, i, argv[i], strlen(argv[i]) );
   }
   return 0;
 }
```

(1) All functions are called by name

(2) Functions can be recursive

(3) How to change an internal value? -- Pass a pointer

(4) argc value = 1+ number of command-line arguments

(5) argv is a vector of command-line arguments: argv[0] is the name of the executable, argv[1] is the first argument and so on

[1] Data types and Constant

(1.8) Functions (**functions are also data!**)

function recursion example:

```c
#include <stdio.h>

int factorial(unsigned int i) {

   if(i <= 1) {
      return 1;
   }
   return i * factorial(i - 1);
}

int  main() {
   int i = 15;
   printf("Factorial of %d is %d\n", i, factorial(i));
   return 0;
}
```

condition of terminating
the recursive calls

[1] Data types and Constant

(1.8) Functions (**functions are also data!**)

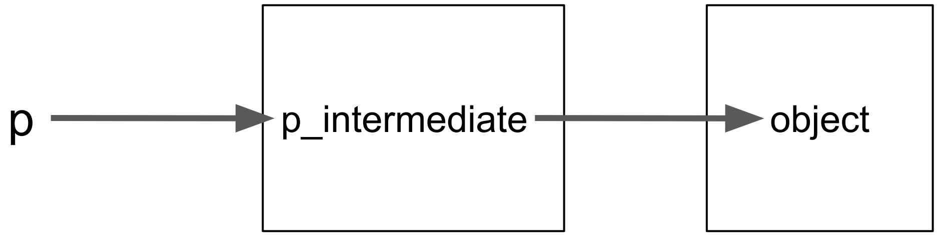
How to provide multiple outputs using one function: **as a rule of thumb, using **p to indicate output**

```
void func(/*input*/   float* p1, unsigned char* p2,

          /*output*/ float** p3, float** p4)
```

How to interpret **p?

p still points to the object with an intermediate pointer

Same idea applies for ***p



[1] Data types and Constant

(1.9) typedef **Purpose: create an alias name for data types**

```
typedef int integer;
```

```
struct TokenizerT_
{
  member list
};
```

```
typedef struct TokenizerT_ Tokenizer;
```

[1] Data types and Constant

(1.9) typedef

typedef vs #define

#define is a C-directive which is also used to define the aliases for various data types similar to **typedef** but with the following differences −

- **typedef** is limited to giving symbolic names to types only where as **#define** can be used to define alias for values as well, q., you can define 1 as ONE etc.
- **typedef** interpretation is performed by the compiler whereas **#define** statements are processed by the pre-processor.

The following example shows how to use #define in a program −

```
#include <stdio.h>

#define TRUE  1
#define FALSE 0

int main( ) {
   printf( "Value of TRUE : %d\n", TRUE);
   printf( "Value of FALSE : %d\n", FALSE);

   return 0;
}
```

Try it

```
Value of TRUE : 1
Value of FALSE : 0
```

[1] Data types and Constant

(1.10) where is array?

You actually don’t need it.

double balance[10] ==> Use the pointer double* pbalance
and int num_elements to mimic it

[2] Pointers

(2.1) pointer to function (**function is also data object!**)

Purpose: provide a trigger mechanism (software interrupt) via callback

Syntax:

A useful technique is the ability to have pointers to functions. Their declaration is easy: write the declaration as it would be for the function, say

```
int func(int a, float b);
```

and simply put brackets around the name and a * in front of it: that declares the pointer. Because of precedence, if you don't parenthesize the name, you declare a function returning a pointer:

```
/* function returning pointer to int */
int *func(int a, float b);

/* pointer to function returning int */
int (*func)(int a, float b);
```

[2] Pointers

(2.1) pointer to function (**function is also data object!**)

E.g., the usage of signal function

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>

void sighandler(int);

int main()
{
   signal(SIGINT, sighandler);

   while(1)
   {
      printf("Going to sleep for a second...\n");
      sleep(1);
   }
   return(0);
}

void sighandler(int signum)
{
   printf("Caught signal %d, coming out...\n", signum);
   exit(1);
}
```

Registration. In C, there is no difference between &function and function when passing as argument

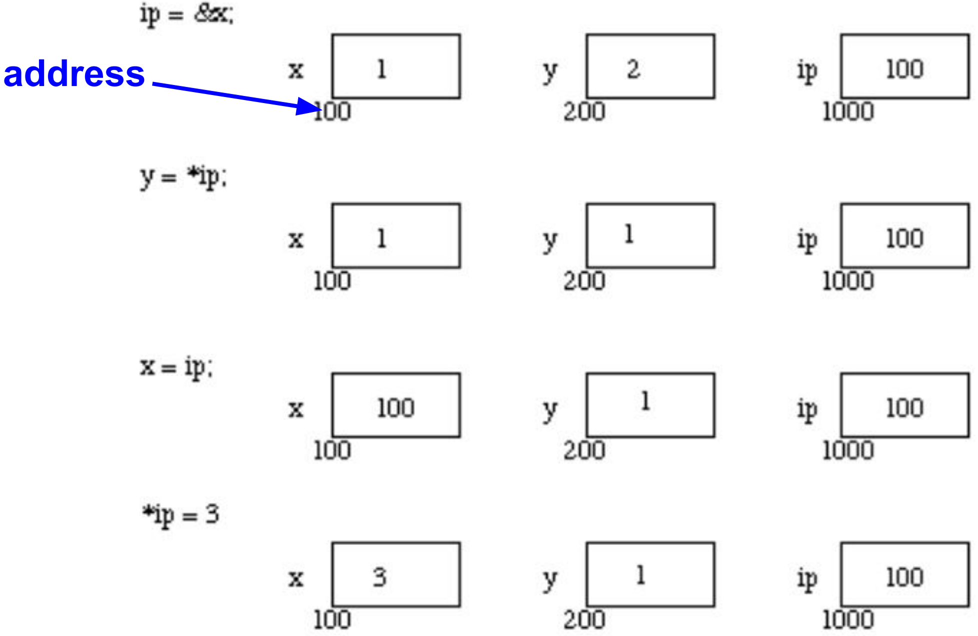
//callback function is triggered once some event occurs
//argument list of callback function is fixed and parameters are passed by system/library function, though the function name and its implementation is user-defined

[2] Pointers

(2.2) regular pointer

```
int x = 1, y =2;
int *ip;
```

The only difference btw a variable and a pointer is that the value (content) of a pointer (e.g., ip here) represents an address!



& is the reference operator: get address
*** is the dereference operator: get the target pointed by a pointer**

[2] Pointers

(2.3) void* pointer **Purpose: generic programming**

```
// ok: void* can hold the address value of any data pointer type
void *pv = &obj;        // obj can be an object of any type
pv = pd;                // pd can be a pointer to any type
```

Following is the declaration for memcpy() function.

```
void *memcpy(void *str1, const void *str2, size_t n)
```

Parameters

- **str1** -- This is pointer to the destination array where the content is to be copied, type-casted to a pointer of type void*.
- **str2** -- This is pointer to the source of data to be copied, type-casted to a pointer of type void*.
- **n** -- This is the number of bytes to be copied.

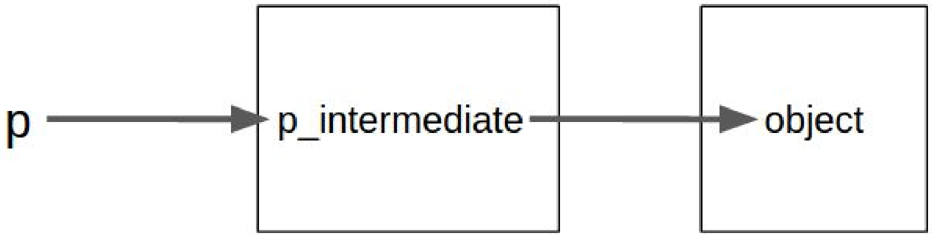
Return Value

This function returns a pointer to destination, which is str1.

[2] Pointers

(2.4) How to interpret **p?

p still points to the object with an intermediate pointer
Same idea applies for ***p



[4] Dynamic memory

(1) Allocation and free

void* malloc (size_t size); If this function fails, it returns NULL

void * calloc(size_t num, size_t size);

calloc() allocates the memory and also initializes the allocated memory to zero.
calloc(...) is basically malloc + memset(if you want to 0 initialise the memory):
calloc() takes two arguments: 1) number of blocks to be allocated 2) size of each block.

void* realloc (void* ptr, size_t size); **//size_t size specifies the size of the new memory!!**

It reallocates memory block which might change the size of the memory block pointed to by ptr.

The function may move the memory block to a new location (whose address is returned by the function).

The content of the memory block is preserved up to the lesser of the new and old sizes, even if the block is moved to a new location. If the new size is larger, the value of the newly allocated portion is indeterminate.

In case that ptr is a null pointer, the function behaves like malloc, assigning a new block of size bytes and returning a pointer to its beginning.

void free(void *ptr); no return value; If a null pointer is passed as argument, no action occurs.

[4] Dynamic memory

(1) Allocation and free

What can go wrong with the stack and the heap?

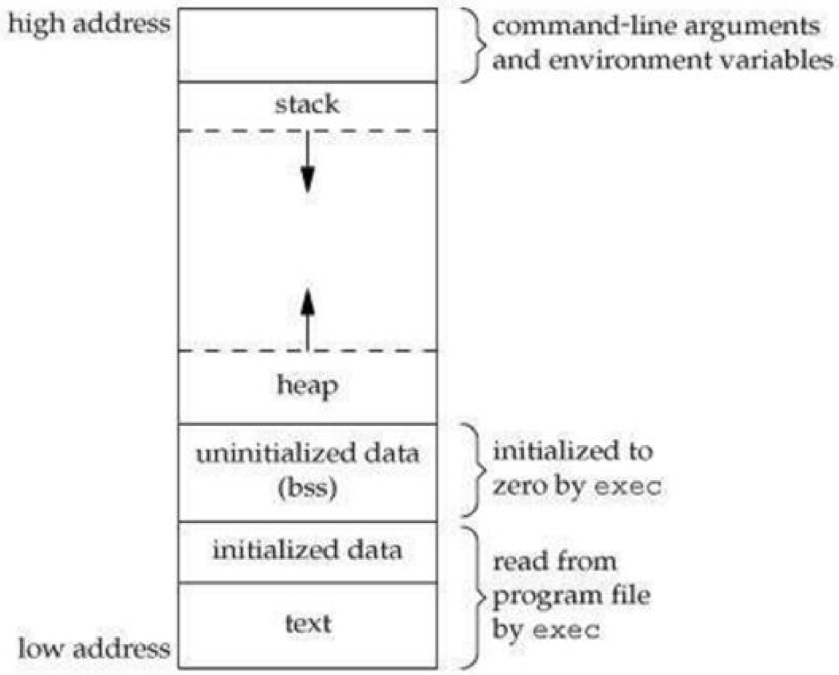
If the stack runs out of memory, then this is called a *stack overflow* – and could cause the program to crash. The heap could have the problem of *fragmentation*, which occurs when the available memory on the heap is being stored as noncontiguous (or disconnected) blocks – because *used* blocks of memory are in between the *unused* memory blocks. When excessive fragmentation occurs, allocating new memory may be impossible because of the fact that even though there is enough memory for the desired allocation, there may not be enough memory in one big block for the desired amount of memory.

The number of malloc() should equal to the number of free() to avoid memory leak

Important Principle: Who calls malloc(), who is responsible to free() it:
E.g., if malloc( ) is called in a function, this function is also responsible to free( ) it.

[4] Dynamic memory

(2) Memory organization



(1) Text segment contains machine code of the compiled program.

(2) Initialized data segment stores all **global, static, constant, and external variables** (declared with extern keyword) that are initialized beforehand

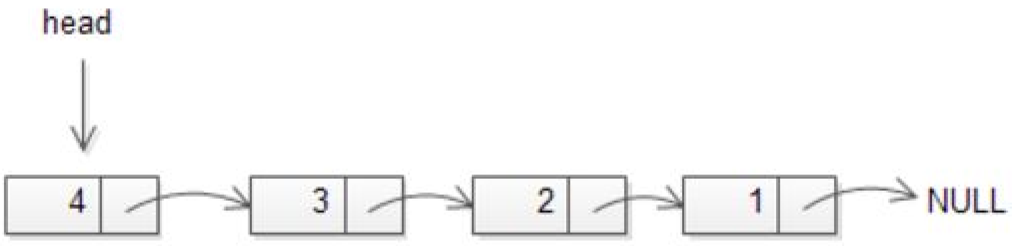
(3) Uninitialized data segment stores all **global** and **static** variables that are initialized to 0 or do not have explicit initialization

```c
#include <stdio.h>

char c;                 /* Uninitialized variable stored in bss*/

int main()
{
    static int i;      /* Uninitialized static variable stored in bss */
    return 0;
}
```

(4) Stack segment stores all **local** variables and is used for passing **arguments to the functions** along with the **return address of the instruction** which is to be executed after the function call is over.

[4] Dynamic memory

(3) memory management via linked list **Purpose: provide a generic dynamic data container**



Node structure:

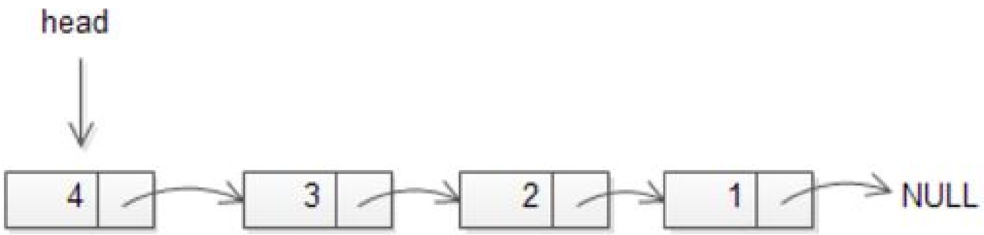
```
typedef struct node
{
    int val; // can be arbitrary type
    struct node * next;
} node_t;
```

Head:

```
node_t * head = malloc(sizeof(node_t));
head->val = 1;
head->next = malloc(sizeof(node_t));

head->next->val = 2;
head->next->next = NULL;
```

[4] Dynamic memory

(3) memory management via linked list **Purpose: provide a generic dynamic data container**



Iterating over a list:

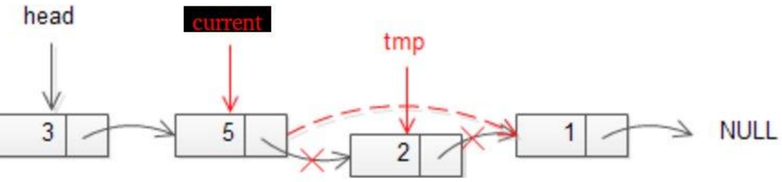
```
void print_list(node_t * head)
{
    node_t * current = head;

    while (current != NULL)
    {
        printf("%d\n", current->val);
        current = current->next;
    }
}
```

Add an item to the end of the list:

```
void push_back(node_t * head, int val)
{
    node_t * current = head;
    while (current->next != NULL)  //'current' points to the last element
    {
        current = current->next;
    }

    /* now we can add a new variable */
    current->next = malloc(sizeof(node_t)); //access last link pointer via 'current'
    current->next->val = val;
    current->next->next = NULL;
}
```

[4] Dynamic memory

(3) memory management via linked list



Remove the first item

```
int pop_front(node_t ** head) {
    int retval = -1;
    node_t * pnext_node = NULL;

    if (*head == NULL) {
        return -1;
    }

    p_next_node = (*head)->next;
    retval = (*head)->val;
    free(*head);
    *head = p_next_node;

    return retval;
}
```

Remove a specific item

```
int remove_by_index(node_t * head, int n) {
    int i = 0;
    int retval = -1;
    node_t * current = head;
    node_t * temp_node = NULL;

    if (n == 0) {  // n is the index of the node to be removed
        return pop_front(head);
    }

    for (int i = 0; i < n-1; i++) {
        if (current->next == NULL) {
            return -1;
        }
        current = current->next; //move 'current' to (n-1)-th item
    }

    temp_node = current->next; //move 'temp_node' to n-th item
    retval = temp_node->val;  //return content of the removed node
    current->next = temp_node->next; //move 'current->next' to (n+1)
    free(temp_node);
    return retval;
}
```

[5] Preprocessor

C language translation actually has two phases: preprocessing and compilation.

C preprocessor supports:

- --- macro substitution
- --- inclusion of named files
- --- conditional compilation

Lines in source or header files that **start with # are preprocessor directives**

```
Macro:
  #define PI_PLUS_ONE (3.14 + 1)
  x = PI_PLUS_ONE * 5;

Conditional compilation:
  #if condition
      Code1
  #else
      Code2
  #endif
```

[5] Preprocessor

E.g., use macro and conditional compilation together for debugging

config.h

```
#define DEBUG 1
```

file1.c

```
#include “config.h”

#if DEBUG
    printf(“a = %d\n”, a);
#endif
```

file2.c

```
#include “config.h”

#if DEBUG
    printf(“b = %f\n”, b);
#endif
```

[6] Program structure

(6.1) general principle

Source files (.c files) contain function definitions, global variables, static variables.
A source file is a scope.

Source files can #include header files (.h files). **The function declarations and macros that need to be exposed can be included in header**.

As a rule of thumb:
Each .c source file (except for main.c) should have a corresponding header to expose interfaces of this .c source

Do not define global variables in header files. You can declare them as extern in header file and define them in a .c source file.

E.g., file1.h

```
extern int i;
```

file1.c

```
int i = 0;
```

[6] Program structure

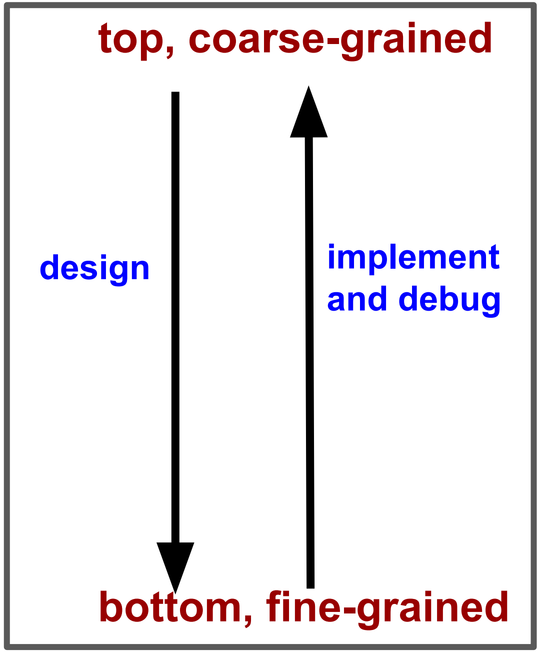
(6.1) general principle

When designing the program:

From top to bottom,
From coarse to fine,
From header to source

When implementing and debugging:

From bottom to top



Tips:

Always define the abstract object first (e.g., structure, linked-list, container, shared global variable or shared memory);

Treat each function as a module: with one and only one specific functionality;

Group several related functions into a source-header pair to provide a complex service;

Simplify the service: enable sequential callings of this group of functions;

[6] Program structure

(6.2) multiple inclusion protection

Multiple inclusion protection: preventing multiple declarations

```
#ifndef MYHEADER_H
#define MYHEADER_H

    // header file contents go here...
#endif // MYHEADER_H
```

This part will be taken into account only if MYHEADER_H is not defined

Here, the first inclusion causes the macro `MYHEADER_H` to be defined. Then, when the header is included for the second time, the `#ifndef` test returns false, and the preprocessor skips down to the `#endif`

[6] Program structure

(6.3) ‘extern’ and ‘static’

Example1:

file1.c
int A;

file2.h
extern int A;

Purpose of ‘extern’: share variable among source files

global variable
share same memory

Example2:

file1.c

```
static int A;  //Purpose of 'static': static for external variable restricts that the global variable A is only available within the current file

void main()
{            }
```

file2.c

```
extern int A; //this A is not from file1.c. It could be from other files, say file3.c
```

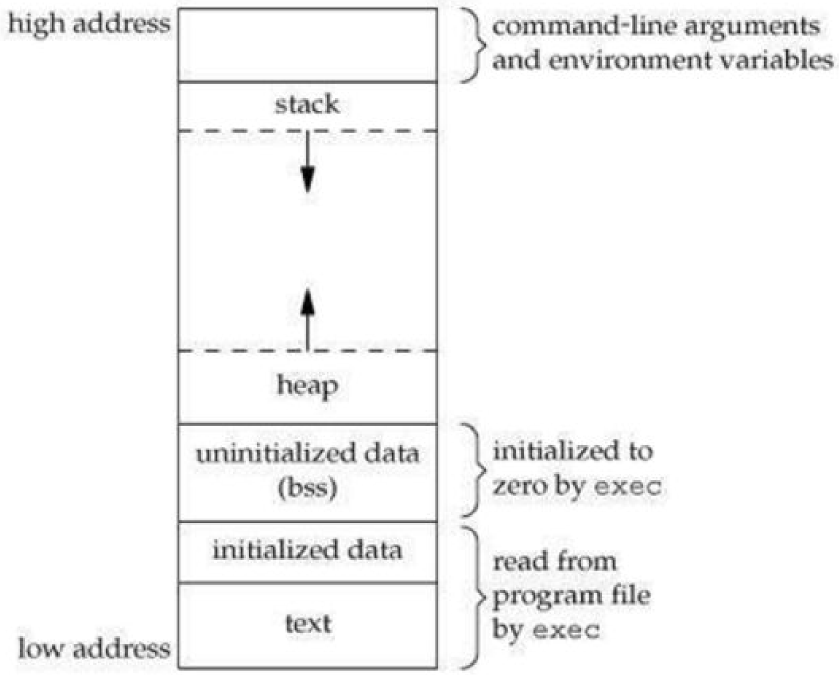
[6] Program structure

(6.3) ‘extern’ and ‘static’

Example3: static for internal variable

```
int f( )
{
   static int c = 3;  //when f() is first called, c is initialized only this time (only once);
                      //when f() is called again, the initialization directive is omitted;
                      //while the latest c value is always kept for use;

   c = c+1;

   return c;
}
```

[7] Memory organization



(1) Text segment contains machine code of the compiled program.

(2) Initialized data segment stores all **global, static, constant, and external variables** (declared with extern keyword) that are initialized beforehand

(3) Uninitialized data segment stores all **global** and **static** variables that are initialized to 0 or do not have explicit initialization

```c
#include <stdio.h>

char c;                /* Uninitialized variable stored in bss*/

int main()
{
    static int i;      /* Uninitialized static variable stored in bss */
    return 0;
}
```

(4) Stack segment stores all **local** variables and is used for passing **arguments to the functions** along with the **return address of the instruction** which is to be executed after the function call is over.

[8] build process and makefile

-c (output an object file (.o))
-o filename (specify the name of the object file or executable file)

[Step A] Compile each .c file into individual object file

```
gcc -c -o factorial.o factorial.cpp
gcc -c -o hello.o     hello.cpp
gcc -c -o main.o      main.cpp
```

[Step B] link all object files to generate an executable file

```
gcc -o exe factorial.o hello.o main.o
```

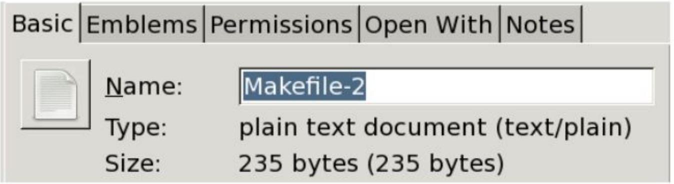
[Step C] Execute `./exe`

[8] build process and makefile

Purpose of using make file:

use one command to generate the executable, even though source files change

Makefile is simply a way of associating short names, called targets, with a series of commands to execute when the action is requested.



The basic form of makefile

```
target: dependencies
[tab] system command
```

Note:
Each command must be proceeded by a **tab** (yes, a tab, not four, or eight, spaces)!

A target could be an executable, an object file or a specific target (clean).

System command describes how target depends on its dependencies.

The basic form of makefile

```
target: dependencies
[tab] system command
```

A target could be an executable or an object file or a specific target (clean).
System command describes how target depends on its dependencies.

e.g.

Makefile-2

```
all: hello2

hello2: main.o factorial.o hello.o
	g++ main.o factorial.o hello.o -o hello2

main.o: main.cpp
	g++ -c main.cpp

factorial.o: factorial.cpp
	g++ -c factorial.cpp

hello.o: hello.cpp
	g++ -c hello.cpp

clean:
	rm *o hello
```

The executable

Get rid of all object files and the executable

[8] build process and makefile

Using Macros CC and CFLAGS in makefile

Macros

Makefile-3

```
# I am a comment, and I want to say that the variable CC
CC = g++

# Hey! I am comment number 2. I want to say that CFLAGS
CFLAGS = -c -Wall


all: hello3

hello3: main.o factorial.o hello.o
	$(CC) main.o factorial.o hello.o -o hello3


main.o: main.cpp
	$(CC) $(CFLAGS) main.cpp

factorial.o: factorial.cpp
	$(CC) $(CFLAGS) factorial.cpp

hello.o: hello.cpp
	$(CC) $(CFLAGS) hello.cpp

clean:
	rm *o hello
```

[8] build process and makefile

Run a makefile to generate your new executable

```
bash-4.1$ make -f Makefile-2
```

The name of your makefile

Make option '-f Makefile-2': Read the file named *Makefile-2* as a makefile

```
bash-4.1$ make
```

(If your makefile has the default name "Makefile" or "makefile")

Execute the executable generated via make

./hello3

[9] error handling

(9.1) errno

The <errno.h> header file defines the integer variable errno, whose value describes the error type produced by a system call or a call to a library function (any function of the C standard library may set a value for errno, even if not explicitly) in the event of an error to indicate what went wrong.

Valid errno are all nonzero (-1 from most system calls; -1 or NULL from most library functions); errno is never set to zero by any system call or library function.

(9.2) perror and strerror **same purpose: display the text message associated with errno**

- The **perror()** function displays the string you pass to it, followed by a colon, a space, and then the textual representation of the current errno value.

  perror should be called right after the error was produced, otherwise it can be overwritten by calls to other functions.

- The **strerror()** function, which returns a pointer to the textual representation of the current errno value.

[9] error handling

(9.3) example

```c
#include <stdio.h>
#include <errno.h>
#include <string.h>

extern int errno ;

int main () {

   FILE * pf;
   pf = fopen ("unexist.txt", "rb");

   if (pf == NULL) {
      fprintf(stderr, "Value of errno: %d\n", errno);

      perror("Error printed by perror");

      fprintf(stderr, "Error opening file: %s\n", strerror( errno ));
   }
   else {
      fclose (pf);
   }

   return 0;
}
```

Opens an existing binary file for reading purpose

fopen fails and it sets errno according to the failure type

```
Value of errno: 2
Error printed by perror: No such file or directory
Error opening file: No such file or directory
```

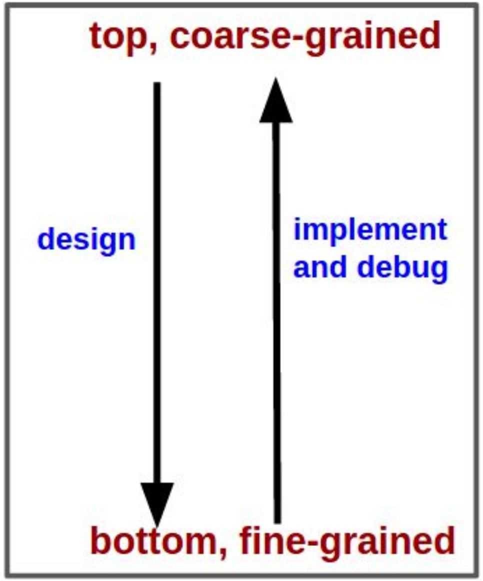
[10] take-home message

When designing the program:

From top to bottom,
From coarse to fine,
From header to source

When implementing and debugging:

From bottom to top



Tips:

Always define the abstract object first (e.g., structure, linked-list, container, shared global variable or shared memory);

Treat each function as a module: with one and only one specific functionality;

Group several related functions into a source-header pair to provide a complex service;

Simplify the service: enable sequential callings of this group of functions;

Thank you for your attention