NETWORK PROGRAMING

Tong Van Van, SoICT, HUST

Lecturer Information

Dr. Tong Van Van

School of Information and Communication Technology,

Hanoi University of Science and Technology

- Office : 405-B1
- Mobile : 0964262425
- E-mail : vantv@soict.hust.edu.vn
- Google Scholar: https://scholar.google.com/citations?user=XKJm_fQAAAAJ&hl=en

Course information

- IT4062: Network programming
- What we study in this course
  - How to build network applications using socket programming paradigm.
  - Socket programming using C (in details)

Course information

- References
  - Documents of professional group related to subject: *Network Programming*.
  - *Unix Network Programming Vol.1*, 3rd edition, W.Richard Stevens, Prentice-Hall
  - *Internetworking with TCP/IP vol.3, Client-Server Programming and Applications (BSD version)*, Douglas E. Comer, David L. Stevens, Prentice-Hall
  - *TCP/IP Sockets in C: Practical Guide for Programmers*, Michael Donahoo, Kenneth Calvert, Elsevier Science

Course information

- **Exercises** in class
  - After each lecture
  - Used for mid-term evaluation
- **Final project**
  - Development of network applications in groups
  - 2-3 members/ group
  - Used for final evaluation
- Grading
  - **Exercises** (50%)
  - **Final project** (50%)

Course information

- **Email** for submitting exercises: *hedspi.hw.it4062@gmail.com*
- **Subject**: *Homework **X***
- You must store the files containing source code in a directory. The directory is compressed and named by Name_*StudentID_HW**x***. Besides, **x** is the sequence number of the homework.
- **Dateline** for homework: Before subsequent session.
- **Cheating** and plagiarism: You will FAIL this course.
- Testing environment: Ubuntu 16.04, GCC compiler
- HW Grading
  - **Functionality**: *30%*
  - **Clean code** (comment, naming, modular design, v.v): *70%*

Course contents

- Lecture contents
  - Review of C programming language
  - Review of related concept in Computer Networks
  - Introduction to Socket API
  - Basic TCP socket: server side, client side
  - UDP socket
  - Multi-process TCP server
  - Application protocol

REVIEW C PROGRAMING

Tong Van Van, SoICT, HUST

Content

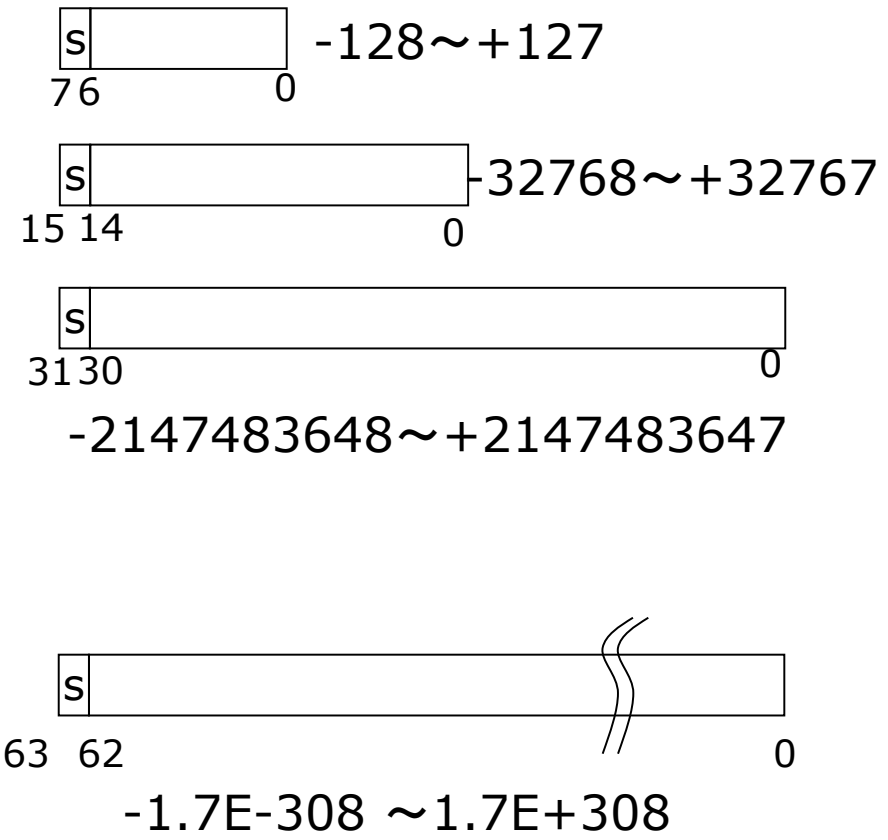
- Data type
- Condition and Iteration
- Function
- Command line argument
- Pointer
- Structure
- Link listed
- I/O function

Data type

- Integer
  - int, char, short, long
- Floating
  - double, float
- Array
  - Collection of A data type
  - Declare : int a[10];

Size of Type

- size of char: 1 bytes
- size of short: 2 bytes
- size of int: 4 bytes
- size of long: 4 bytes
- size of float: 4 bytes
- size of double: 8 bytes



Condition and Loop Structure

- if … else
- switch
- for
- while, do … while

Condition

- a == b
  - b equals to a
- a != b
  - b is different to a
- a > b
  - b is smaller than a
- a >= b
  - b isn’t greater than a
- a < b
  - b is greater than a
- a <= b
  - b isn’t smaller than a

if … else

```
if(condition){
        statement1;
        …
}
else{
        statement2;
        …
}

Example :
if( x == 1){
        y = 3;
        z = 2;
}
else{
        y = 5;
        z = 4;
}
```

```
if (condition)
        task1;
else task2;
```

is equivalent?

```
if (condition)
task1;
if (!condition)
task2;
```

switch

```
switch(condition){
        case value1: statement1;…; break;
        case value2: statement2;…; break;
        …
        default: statement;…; break;
}

Example :
int monthday( int month ){
        switch(month){
        case 1: return 31;
        case 2: return 28;
        …
        case 12: return 31;
        }
}
```

for

- `expr1, expr3`: assignments or function calls
- `expr2`: relational expression

  Any of the three expression can be omitted
  - the semicolons must remain

```
for(expr1 ; expr2 ; expr3){
        statements ;
        …
}

Example :
for( x = 0; x < 10; x++){
        printf("%d\n", x);
}
```

while, do…while

- If there is no initialization or re-initialization, the `while` is most natural

```
while ((c = getchar()) == ' ' || c == '\n' || c = '\t')
        /* skip white space characters *
```

```
while(condition){
        statement;
        …
}

Example:
x = 0;
while(x < 10){
        printf("%d\n",x);
        x = x + 1;
}
```

```
do{
        statement;
        …
} while(condition)

Example:
x = 0;
do{
        printf("%d\n",x);
        x = x + 1;
} while(x < 10)
```

break

- `break`
  - Terminates the execution of the nearest enclosing loop or conditional statement in which it appears.
- `continue`
  - Pass to next iteration of nearest enclosing do, for, while statement in which it appears
- Example

```
/* trim: remove trailing blanks, tabs, newlines */
char s[MAX]
int n;
for (n = strlen(s)-1; n >= 0; n--)
   if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
        break;
s[n+1] = '\0';
```

```
for (i = 0; i < n; i++)
   if (a[i] < 0) /* skip negative elements */
        continue;
... /* do positive elements */
```

Function

- A function is a group of statements that is executed when it is called from some point of the program. The following is its format:

```
type name ( parameter1, parameter2, ...) {
      statements;
}
```

- where:
  - `type` is the data type specifier of the data returned by the function.
  - `name` is the identifier by which it will be possible to call the function.
  - `parameters` (as many as needed): Each parameter consists of a data type specifier followed by an identifier
  - `statements` is the function's body. It is a block of statements surrounded by braces { }.

Example of function

```
#include <stdio.h>
int squaresub(int a)
{
    return a*a;
}

int main()
{
    int  b = 10;
    printf("%d\n", squaresub(5));
    return 0;
}
```

Data type of function

Return value statement

Use function

Usage of command line arguments

- `main( int argc, char **argv)`
- `main( int argc, char *argv[])`

- Argc : number of arguments
- argv[0] : argument 0
- argv[1] : argument 1
- argv[2] : argument 2

Example :

%./a.out 123 456 789

arg[0]: ./a.out

arg[1]: 123

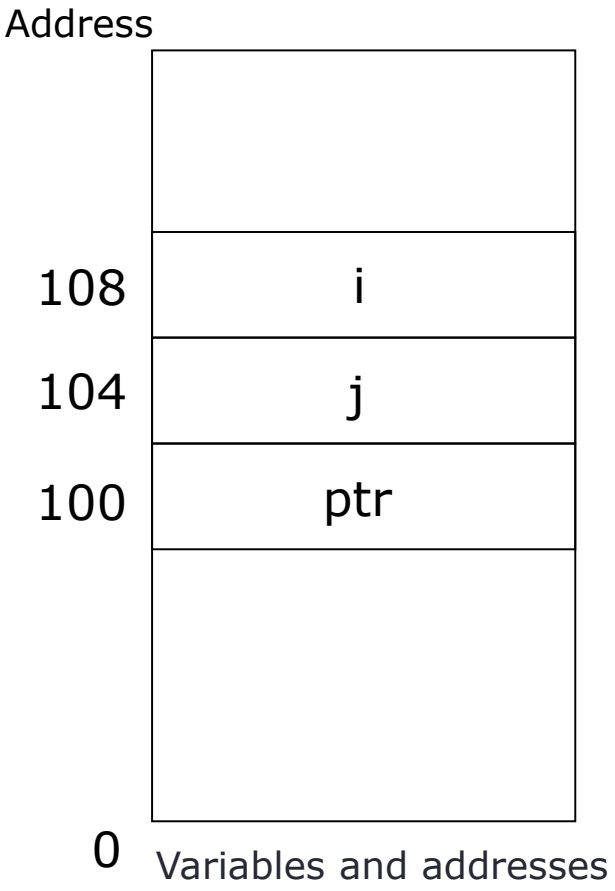
arg[2]: 456

arg[3]: 789

Pointer

- Pointer variable
  - "Variable" refers to variable
- int i = 10;
- int j = 20;
- int *ptr;
- Pointer to pointer:

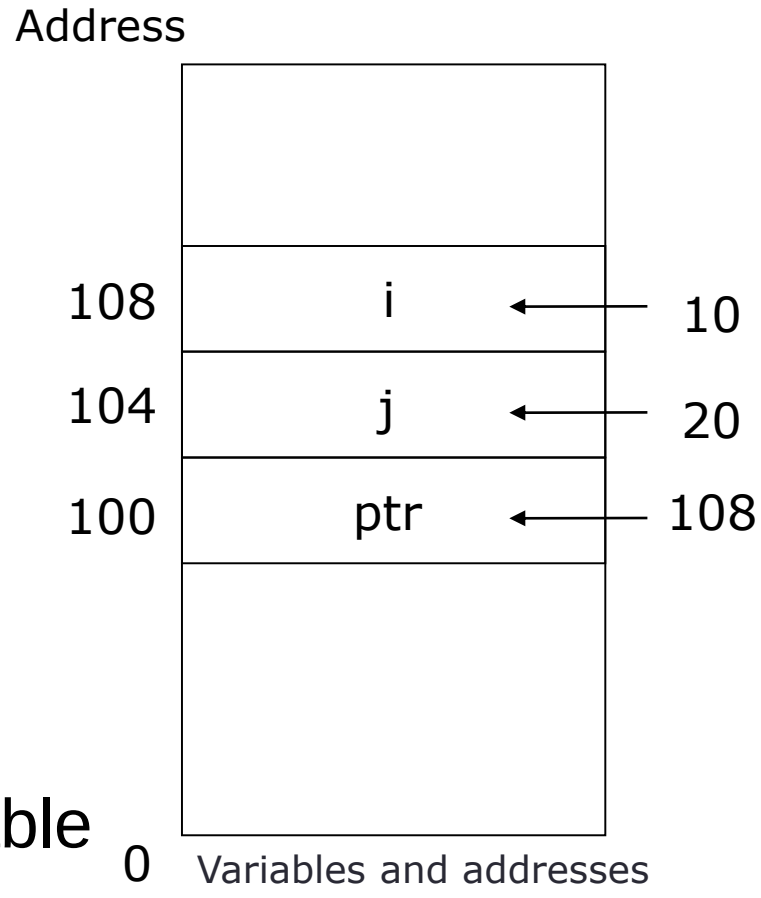
  int **p;



Pointer (cont)

- int i = 10;
- int j = 20;
- int *ptr = &i;

- printf("i=%d\n", &i)
- printf("ptr=%d\n", ptr)
- printf("i=%d\n", i)
- printf("*ptr=%d\n",*ptr)
- Ptr refers to the pointer variable



Pointer (cont)

- int x=1, y=5;
- int z[10];
- int *p;
- p=&x; /* p refers to x   */
- y=*p;   /*y is assigned the value of x*/
- *p = 0; /* x = 0   */
- p=&z[2]; /* p refer to z[2] */

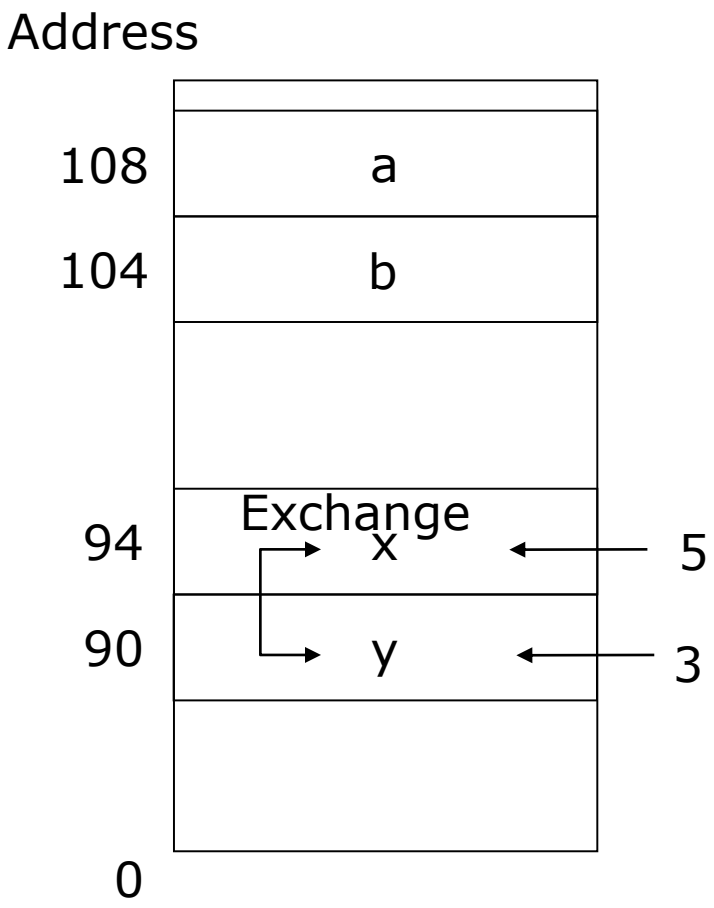
Pointer and function

```
#include <stdio.h>
void swap(int x, int y)
{
        int temp;
        temp = x;
        x = y;
        y = temp;
}

int main(){
        int a = 5;
        int b = 3;
        swap (a, b);
        printf("a=%d\n", a);
        printf("b=%d\n", b);
        return 0;
}
```

Result ?

Pointer and function (cont)

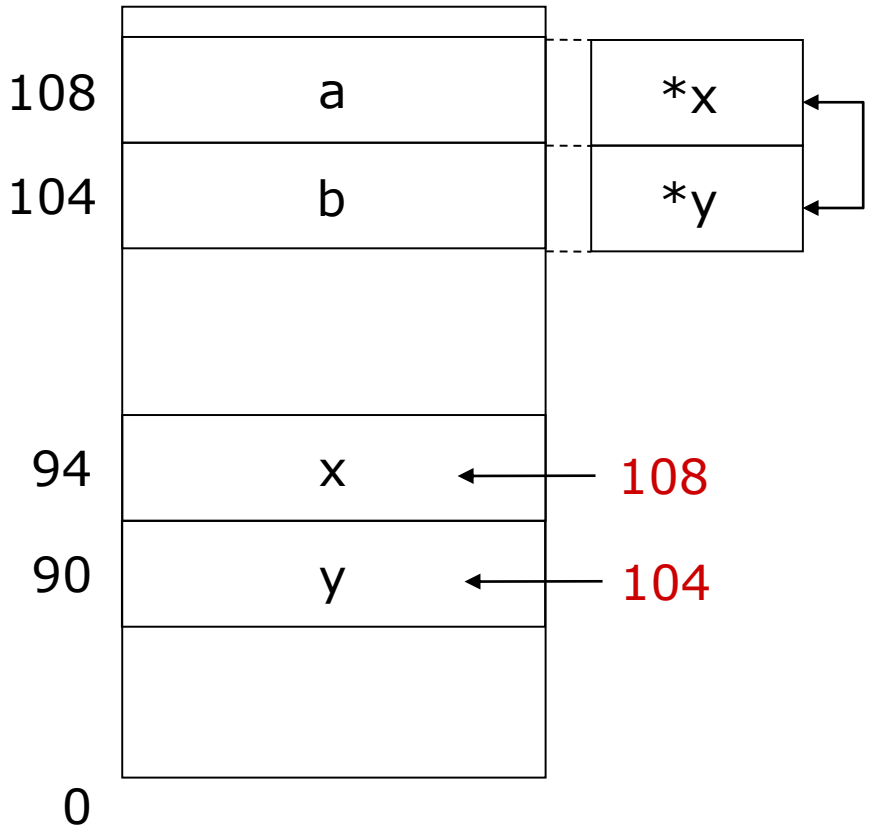
```
#include <stdio.h>
void swap(int x, int y)
{
        int temp;
        temp = x;
        x = y;
        y = temp;
}

int main(){
        int a = 5;
        int b = 3;
        swap (a, b);
        printf("a=%d\n", a);
        printf("b=%d\n", b);
        return 0;
}
```



Pointer and function (cont)

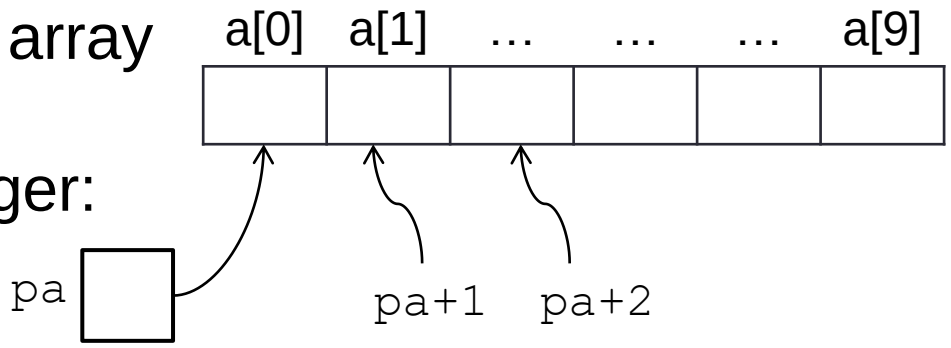
```
#include <stdio.h>
void swap(int *x, int *y)
{
        int temp;
        temp = *x;
        *x = *y;
        *y = temp;
}

int main(){
        int a = 5;
        int b = 3;
        swap (&a,&b);
        printf("a=%d\n", a);
        printf("b=%d\n",b);
        return 0;
}
```

Program to exchange 2 value of variables



Pointer and Array

- The declaration an integer array
  `int a[10];`
- If `pa` is a pointer to an integer:
  `int *pa;`
  `pa = &a[0];`
- Similarity: `pa` and `a` are pointers
- Difference: `pa` is a variable but `a` is not
  - legal: `pa ++; pa = a;`
  - Illegal: `a++; a = pa;`
- `a`: constant pointer



Constant pointer vs Pointer to constant

- Constant pointer: a pointer that cannot change the address its holding.
  - Declaration: `<type> *const <name of pointer>`
- Pointer to constant: a pointer through which one cannot change the value of variable it points
  - Declaration: `const <type>* <name of pointer>`
- Constant Pointer to a Constant: mixture of the above two types of pointers
  - Declaration:

    `const <type of pointer>* const <name of pointer>`

Constant pointer

```c
#include <stdio.h>
int main(void)
{
    int var1 = 0, var2 = 0;
    int *const ptr = &var1;
    ptr = &var2;
    printf("%d\n", *ptr);

    return 0;
}
```

```
$ gcc -Wall constptr.c -o constptr
constptr.c: In function ‘main’:
constptr.c:7: error: assignment of read-only variable
‘ptr’
```

Pointer to constant

```c
#include <stdio.h>
int main(void)
{
    int var1 = 0;
    const int* ptr = &var1;
    *ptr = 1;
    printf("%d\n", *ptr);

    return 0;
}
```

```
$ gcc -Wall constptr.c -o constptr
constptr.c: In function ‘main’:
constptr.c:7: error: assignment of read-only location
‘*ptr’
```

Constant Pointer to a Constant

```c
#include <stdio.h>
int main(void)
{
    int var1 = 0,var2 = 0;
    const int* const ptr = &var1;
    *ptr = 1;
    ptr = &var2;
    printf("%d\n", *ptr);

    return 0;
}
```

```
$ gcc -Wall constptr.c -o constptr
constptr.c: In function ‘main’:
constptr.c:7: error: assignment of read-only location
‘*ptr’
constptr.c:8: error: assignment of read-only variable
‘ptr’
```

Return pointer from functions vs Function pointer

- Return pointer from functions:

```
<type>* <name of function> (<types of parameter>)
```

- Function pointer: pointers to functions

```
<type> (*<name of function>) (<types of parameter>)
```

```c
int func (int a, int b)
{
   printf("\n a = %d\n",a);
   printf("\n b = %d\n",b);

   return 0;
}
```

```c
int main(void)
{
   // Function pointer
   int(*fptr)(int,int);
   // Assign address to
   // function pointer
   fptr = func;
   func(2,3);
   fptr(2,3);

   return 0;
}
```

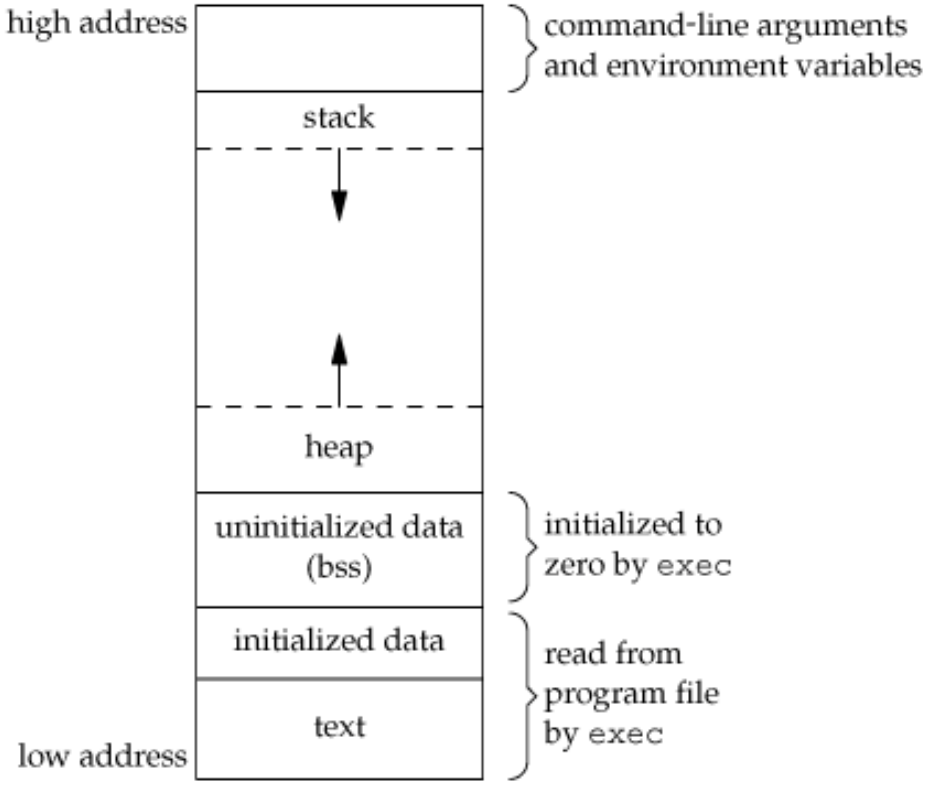
`void` pointer

- `void` pointer: a special a pointer that has no associated data type with it
  - Can hold address of any type and can be typcasted to any type.
  - Generic programming
- Declaration: `void *<name of pointer>;`
- The void pointer cannot be dereferenced directly
  - The void pointer must first be explicitly cast to another pointer type before it is dereferenced.

```
#include <stdio.h>
int main()
{
    int a = 10;
    void *ptr = &a;
    printf("%d", *(int *)ptr);
    return 0;
}
```

Dynamic Memory Allocation

- A typical memory representation of C program consists of following sections.
  1. Text segment: code segment
  2. Initialized data segment
  3. Uninitialized data segment
  4. Stack
  5. Heap: the segment where dynamic memory allocation usually takes place



Dynamic Memory Allocation

- `void * malloc( size_t size );`
  - Allocates requested size of bytes and returns a pointer to first byte of allocated space
  - Doesn't initialize the allocated memory
  - Asigment: `ptr = (cast-type*) malloc(byte-size)`
- `void * calloc( size_t num, size_t size );`
  - Allocates space for an array elements, initializes to zero and then returns a pointer to memory
  - Initializes the allocates memory block to zero
  - Asigment: `ptr = (cast-type*)calloc(n, element-size);`
  - Equivalent:

```
ptr = malloc(size);
memset(ptr, 0, size);
```

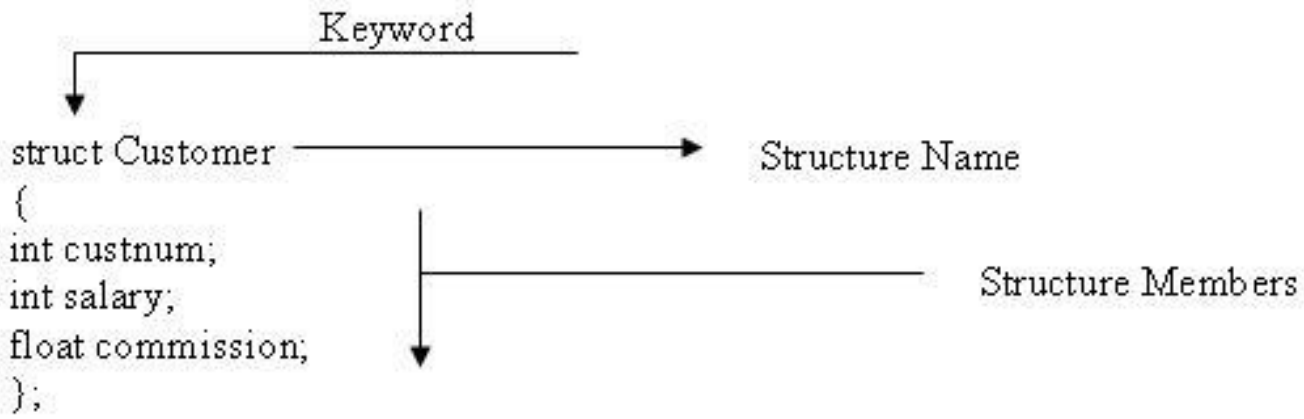
Dynamic Memory Allocation

- `void *realloc(void *ptr, size_t size);`
  - Deallocates the old object pointed to by ptr and returns a pointer to a new object that has the size specified by size
  - `ptr = realloc(ptr, newsize);`
- `void free(void *ptr);`
  - Deallocate the previously allocated space
- Memory Leak
  - Create a memory in heap and forget to delete it
  - To avoid memory leaks, memory allocated on heap should always be freed when no longer needed
- valgrind: suite of tools for debugging and profiling programs.

```
$ valgrind –leak-check=full <program>
```

Structure

- Structure is a collection of variables under a single name. Variables can be of any type: int, float, char etc.
- ***Declaring a Structure:***



Using variable structure

- Declare structure variable?
  - This is similar to variable declaration.
  - Example

```
int a;
struct Customer John;
```

- Access structure members: use the dot operator

```
<structure variable name>.<member name>
```

- Access to members of a pointer to the variable structure: using operators →

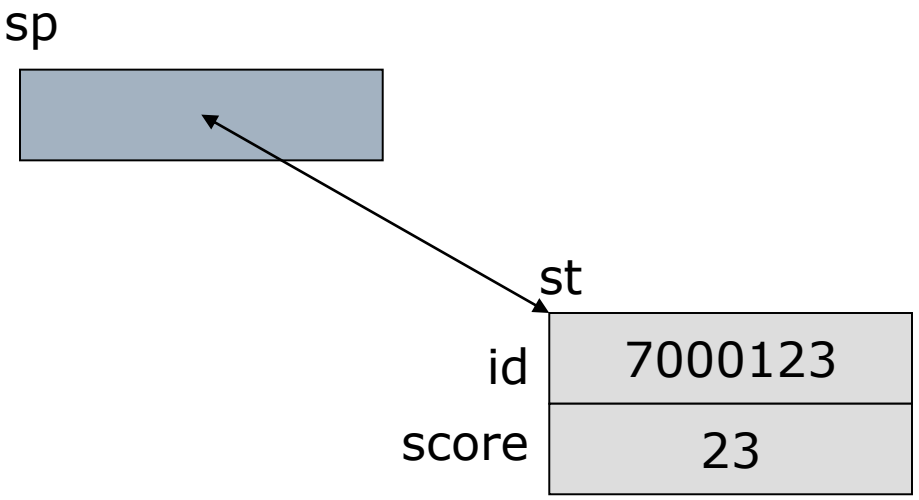
```
<structure variable name> -> <member name>
```

Example

```
struct student{
    int id;
    int score;
};

int main()
{
    int i;
    struct student students[5];
    for(i=0; i<5; i++){
        students[i].id = i;
        students[i].score = i;
    }
    for(i=0;i<5;i++){
        printf("student id:%d, score:%d\n",
             students[i].id, students[i].score);
    }
    return 0;
}
```

Structure and Pointer

```
struct student st;
struct student *sp;
sp = &st;
sp->id = 7000123;
(*sp).score = 23;
```



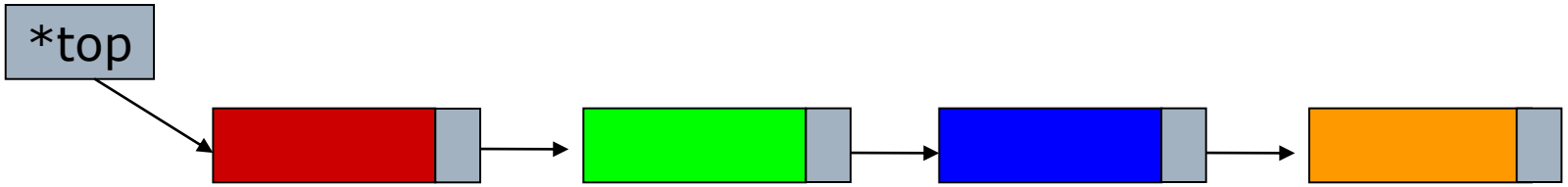
printf("%d\n", sp->score);

Link list

- Store a pointer to the next structure in the structure

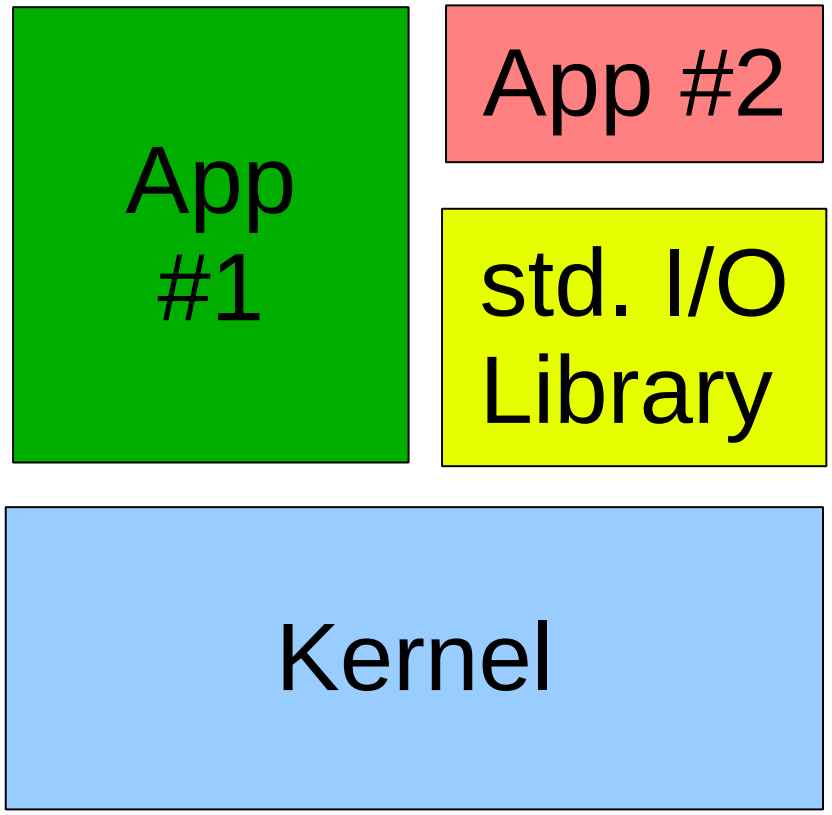
```
struct student {
   int id;
   int score;
   struct student *next;
}
```

- *Warning : allocate memory before use and release memory after use*



I/O function

- All I/O calls ultimately go to the kernel
- I/O library helps with buffering, formatting, interpreting (esp. text strings & conversions)



Input function (include in stdio.h)

- Functions
  - printf( )
    - Print formatted data to stdout
  - fprintf( )
    - Write formatted output to stream
  - gets( )
    - Read one line from standard input
    - **NEVER EVER USE THIS!**
  - fgets( )
    - Get string from stream, a newline character makes fgets stop reading
    - **USE THIS INSTEAD**
  - getc( )
    - Character read from standard input
  - putc( )
    - Export one character to standard output
- Deprecated functions
  - scanf( )
    - Read formatted data from stdin
  - fscanf( )
    - Read formatted data from stream

Input function (include in unistd.h)

- Function
  - read()
    - Argument : number of bytes read and target
  - write()
    - Argument : the number of bytes to write to output
  - open()
  - close()

open()/read()/write()/close()

```
#include <fcntl.h>
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
#define BUFSIZE 1024

int main()
{
    char buf[BUFSIZE];
    int fd;
    int nbyte;
    fd = open("test.txt", O_RDONLY, 0);
    while((nbyte = read(fd, buf, BUFSIZE)) > 0) {
        write(1, buf, nbyte);
    }
    close(fd);

    return 0;
}
```

File handling functions

- `fopen(char *filename, char *mode)`
  - `r,w,a,r+,w+,a+`
- `fgets(char *s,int length,FILE *fd)`
- `fgetc(FILE *fd)`
- `fclose(FILE *fd)`

Example

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  FILE *fp;
  char buf[1024];
  int c;
  fp = fopen(argv[1],"r");
       while((fgets(buf, sizeof(buf),fp)) != NULL){
                    fputs(buf,stdout);
      }
  fclose(fp);

  return 0;
}
```