Introduction to C Programming in Unix Environment - II

- Memory Arrangement
- Structs & Arrays
- Pointers
- Memory Management in C
- Memory layout in Unix
Last week - Summary

- Unix environment
- Data types in C
- C – simple programs
- C Arrays – A Glance
- Makefiles
Definitions

- Bit – a binary digit – zero or one

- Byte – 8 bits

MSB (Most Significant Bit)  
LSB (Least Significant Bit)
Memory Arrangement

- Memory is arranged in a sequence of addressable units (usually bytes)
  - sizeof( <type> ) returns the number of units it takes to store a type:
    - sizeof(char) = 1
    - sizeof(int) = 4 (on most of our machines)
int main(){
    char c;
    int i,j;
    double x;
    ...
}

Not necessarily ordered this way.
Array

Defines a block of consecutive cells

```c
int main()
{
    int i;
    int a[4];
    ...
}
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Arrays - the [ ] operator

int arr[5] = {1, 5, 2, 1, 3}; /*arr begins at address 40*/

- Address Computation Examples
  - arr[0] 40+0*sizeof(int) = 40
  - arr[3] 40+3*sizeof(int) = 52
  - arr[i] 40+i*sizeof(int) = 40 + 4*i
  - arr[-1] 40+(-1)*sizeof(int) = 36 /* can be the code segment or other variables*/
In C, we can define new types

These types are a composite of other types – *structures*

*origin of classes*

```c
struct Complex {
    double _real;
    double _imag;
};
```

```c
struct Complex c;
c._real = 1;
c._imag = 0;
```

`sizeof(struct Complex) = ?`
Structs

- Contiguously-allocated region of memory
- Refer to members within structure by name
- Members may be of different types

```c
typedef struct rec {
    int i;
    int a[3];
    int *p;
} rec;
```
Pointers

• Declaration:
  <type> *p;
  p points to objects of type <type>

• Pointer → value
  *p = x;
  y = *p;
  *p is the value that p points to.

• value → pointer
  &x - the address of x
Pointers

```c
int main()
{
    int i, j;
    int *x; /* x points to an integer*/
    i = 1;
    x = &i;
    j = *x;
    x = &j;
    (*x) = 3;
}
```

```
  i   j   x
   1
```
Pointers

int main()
{
    int i,j;
    int *x; /* x points to an integer*/
    i = 1;
    x = &i;
    j = *x;
    x = &j;
    (*x) = 3;
}

i j x
0x0100

0x0100
int main()
{
    int i,j;
    int *x; // x points to an integer
    i = 1;
    x = &i;
    j = *x;
    x = &j;
    (*x) = 3;
}

1 1 0x0100

0x0100 ............................
Pointers

```c
int main()
{
    int i,j;
    int *x; // x points to an integer
    i = 1;
    x = &i;
    j = *x;
    x = &j;
    (*x) = 3;
}
```

Pointers

```c
int main()
{
    int i, j;
    int *x; // x points to an integer
    i = 1;
    x = &i;
    j = *x;
    x = &j;
    (*x) = 3;
}
```

```
+---+---+---+
i        x
+---+---+---+
  1     3  0x0104
  0x0100
```


Pointers – Motivation!

Does nothing

```c
void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

Works

```c
void swap(int *pa, int *pb)
{
    int temp = *pa;
    *pa = *pb;
    *pb = temp;
}
```

```c
int main()
{
    int x, y;
    x = 3; y = 7;
    swap(x, y);
    ...
Pointer Arithmetic

```c
int a[4];
int *p = a;
char *q = (char *)a; /* Explicit cast */
p++;   Where do p and q point ?
q++;   And now ?
```

Diagram:
```
      q
      p
```
Arrays are constant pointers
Can only change the value of their cells.

```c
int *p;
int a[4];
p = a; /* same as p = &a[0] = a*/
p[1] = 102; /* same as *(p+1)=102; */
*(a+1) = 102; /* same */
p++; /* p ++ p+1 == &a[1] */
a = p;    illegal
a++;    illegal
```
Pointers and Arrays

```c
int *p;
int a[4];

sizeof (p) = sizeof (void*) = ? (4 bytes)
sizeof (a) = ? (sizeof(int) * array length)
```

*special case where array is not considered as pointer*
Pointers and Arrays

\[
\text{int foo( int *p );}
\]

and

\[
\text{int foo( int a[] );}
\]

- Declare the same interface
- In both cases, a pointer to \textit{int} is being passed to the function \texttt{foo}
Pointers to pointers

(1) int i = 3;
(2) int j = 4;
(3) int k = 5;

(4) int *ip1 = &i;
(5) int *ip2 = &j;
(6) int **ipp = &ip1;
Pointers to pointers

(1) int i = 3;
(2) int j = 4;
(3) int k = 5;

(4) int *ip1 = &i;
(5) int *ip2 = &j;
(6) int **ipp = &ip1;
(7) ipp = &ip2;
(1) int i=3;
(2) int j=4;
(3) int k=5;

(4) int *ip1 = &i;
(5) int *ip2 = &k;
(6) int **ipp = &ip1;
(7) ipp = &ip2;
(8) *ipp = &k;
Memory Management

- During run time, variables can be stored in one of three places
  - Stack
  - Static heap
  - Dynamic heap
Memory layout in Unix

- Each program has its own logical memory
- OS maps logical memory to physical memory
- A program logical memory is not visible to other programs
Stack

• Maintains memory **during function calls**
  • Arguments of the function
  • Local variables
  • Call Frame

• Variables on the stack have limited “life time”
Stack - example

```c
int foo( int a, double f )
{
    int b;
    ...
    {
        int c;
        ...
    }
    ...
}
```
int foo( int a, double f )
{
    int b;
    ...
    {
        int c;
        ...
    }
    ...
}

Stack - example

```c
int foo( int a, double f )
{
    int b;
    ...
    {
        int c;
        ...
    }
    ...
}
```
int foo( int a, double f )
{
    int b;
    ...
    {
        int c;
        ...
    }
    ...
}
Stack

• How the stack looks like when a **recursive function** is invoked?

• what about **infinite loops**?
Static heap

• Memory for global/static variables (initialized, uninitialized)

• Static variables are defined throughout the execution of the program

```c
int add2(int a, int b) {
    int c = 0;
    static int total = 0;
    c = a+b; total+= c;
    return c;
}
```
Dynamic heap

• Memory that can be allocated and freed by the program during run time

• The program controls how much is allocated and when

• Limitations based on run-time situation
  • Available memory on the computer
Dynamic heap

```c
void *malloc( size_t Size );
```

- Returns a `void *` pointer to a **new** memory block of size `Size bytes`
- Returns `NULL` if it cannot allocate memory of the requested size.
Signature:
void *malloc( size_t Size );

int* iptr = (int*) malloc(sizeof(int));

struct Complex* complex_ptr = (struct Complex*)malloc(sizeof(struct Complex));
De-allocating/freeing memory

```c
void free( void *p );
```

- Returns the memory block pointed by `p` to the pool of unused memory

- No error checking!
  - If `p` was not allocated by malloc or was freed before, undefined behavior
Further Info
see: http://linux.die.net/man/3/calloc

Read manual page of
• malloc
• calloc: void* calloc(size_t nmemb, size_t size);
allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory. The memory is set to zero (takes more time).
• Realloc: void* realloc(void* ptr, size_t size);
changes the size of the memory block pointed to by ptr to size bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes.
• free
Memory layout in Unix

```c
#include <stdio.h>
#include <stdlib.h>

int global_variable;

int main(int argc, char **argv) {
    int local_variable;
    static int static_variable;
    int *dynamic_variable = (int*)malloc(sizeof(int));

    printf("addresses of:
    function main: 0x080483f4
    global variable: 0x0804a018
    local variable: 0xbf806598
    static variable: 0x0804a022
    dynamic memory: 0x08c19008
");
    (unsigned long)main,
    (unsigned long)&global_variable,
    (unsigned long)&local_variable,
    (unsigned long)&static_variable,
    (unsigned long)dynamic_variable);
    return 0;
}
```