10. Pointer Applications

[ECE10002/ITP10003] C Programming

Agenda

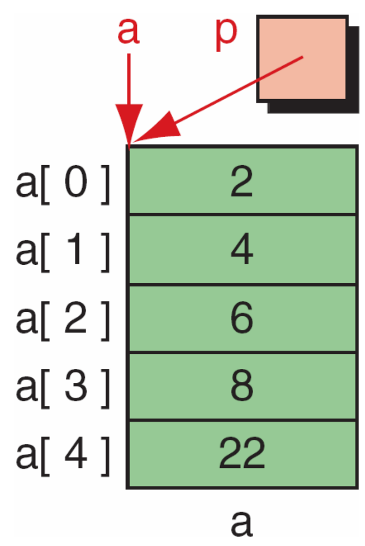
- Arrays and Pointers
- Pointer Arithmetic and Arrays
- Memory Allocation Functions

Arrays and Pointers

- The name of an array is a pointer constant to the first element.
- Index operator is also available for pointers.
  - p[n]: n^{th} element starting from p

Ex)

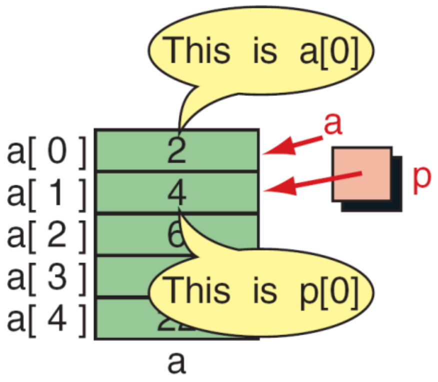
```
int a[5];
int *p = a;          // Then, a[i] ≡ p[i], for all i's
for(i = 0; i < 5; i++)
   printf("a[%d] = %d, p[%d] = %d₩n",
                                 i, a[i], i, p[i]);
```



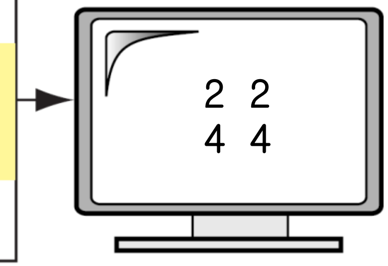
Note! p is not a duplication of a, but just an alias of the same memory space

Array and Pointers

- Multiple names for an array to reference different location




```c
#include <stdio.h>
int main (void)
{
  int  a[5] = {2, 4, 6, 8, 22};
  int* p;
  …
  p = &a[1];
  printf("%d %d", a[0], p[-1]);
  printf("\n");
  printf("%d %d", a[1], p[0]);
  …
} // main
```



[Advanced Issue] What if we run the following statements?

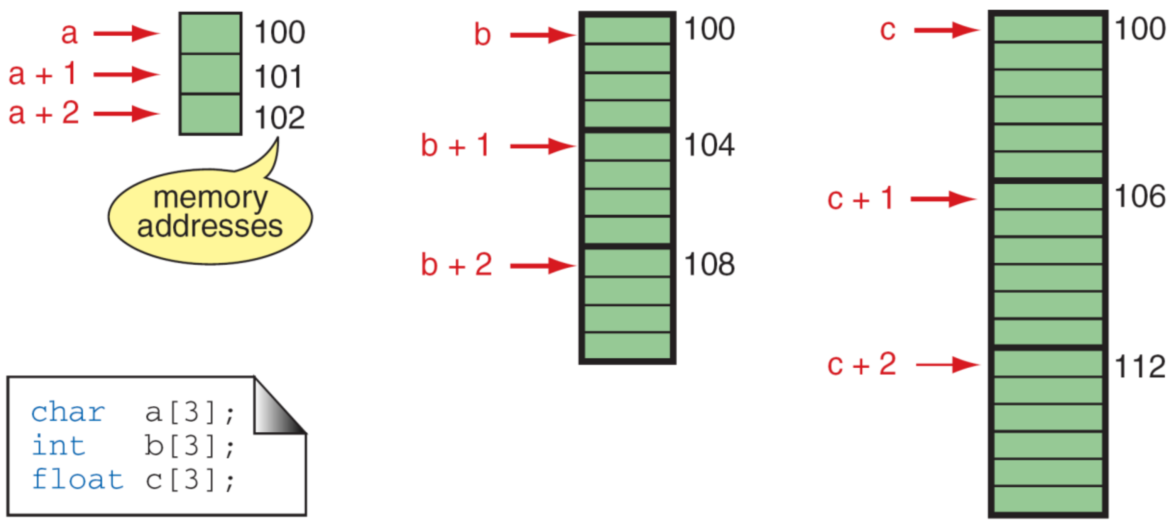
- p = p + 1;   // or “p++;”
- a = a + 1;   // or “a++;”

Agenda

- Arrays and Pointers
- <u>Pointer Arithmetic and Arrays</u>
- Memory Allocation Functions

Pointer Arithmetic and Arrays

- For a pointer p, p ± n is a pointer to the value n elements away.
  - n is called offset
  - address = pointer + (offset * size_of_element)
  - p + n == &p[n], *(p+n) ≡ p[n]

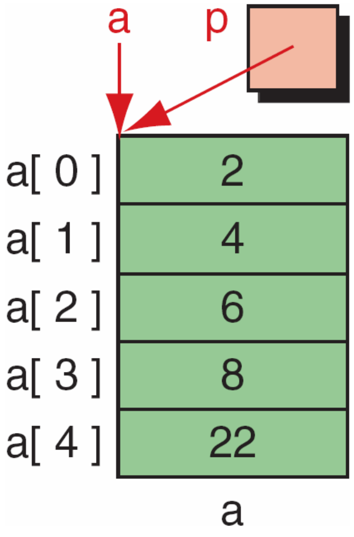


Pointer Arithmetic and Arrays

- Pointer constant cannot be assigned, but pointer variable can be.

```
int a[10];
int *p = a;                    // *p ≡ a[0]

a = a + 1;   // invalid
p = p + 1;   // valid          // *p ≡ a[1]
p++;         // valid          // *p ≡ a[2]
```



Pointer Arithmetic and Arrays

- Printing array using pointer

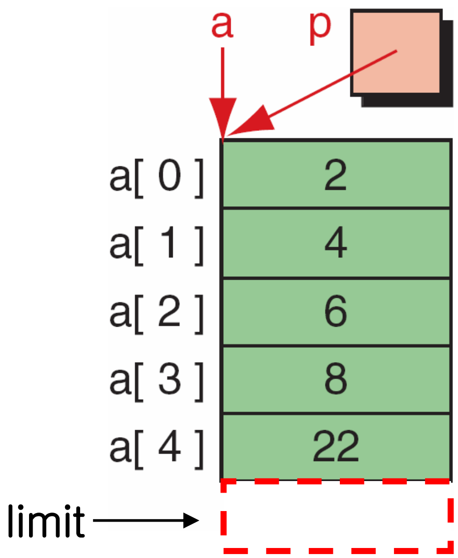
  int a[5];

  - Using counter variable

    ```
    int i = 0;
    for(i =0; i < 5; i++)
       printf("%d₩n", a[i]);
    ```

  - Using pointers

    ```
    int *p = NULL, *limit = a + 5;
    for(p = a; p < limit; p++)
       printf("%d₩n", *p);
    ```



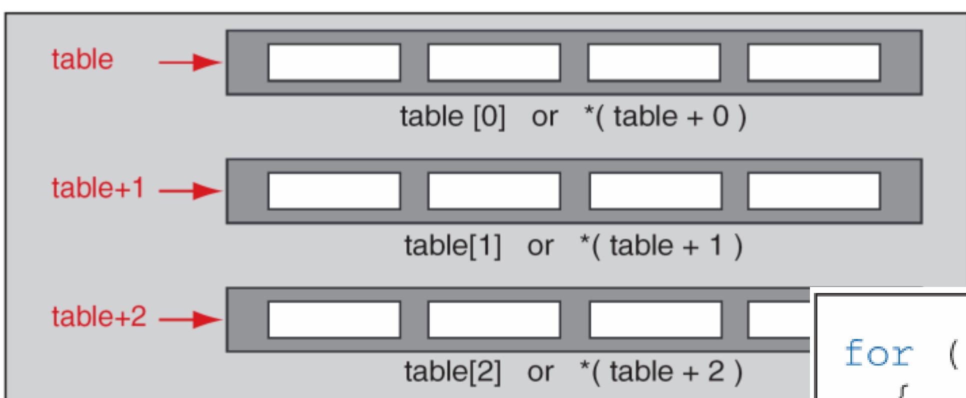
Pointers and Two Dimensional Arrays

- For a 2D array table, table[idx] is a 1D array

  Ex) int table[3][4];

  - table[i]'s are rows(1D array) composing table
  - table[i] = *(table+i) is also true for high dimensional arrays

  Ex) table[i][j] = (*(table+i))[j] = *(*(table+i)+j) = *(table[i]+j)




```
for (i = 0; i < 3; i++)
  {
   for (j = 0; j < 4; j++)
      printf("%6d", *(*(table + i) + j));
   printf( "\n" );
  } // for i
```

Agenda

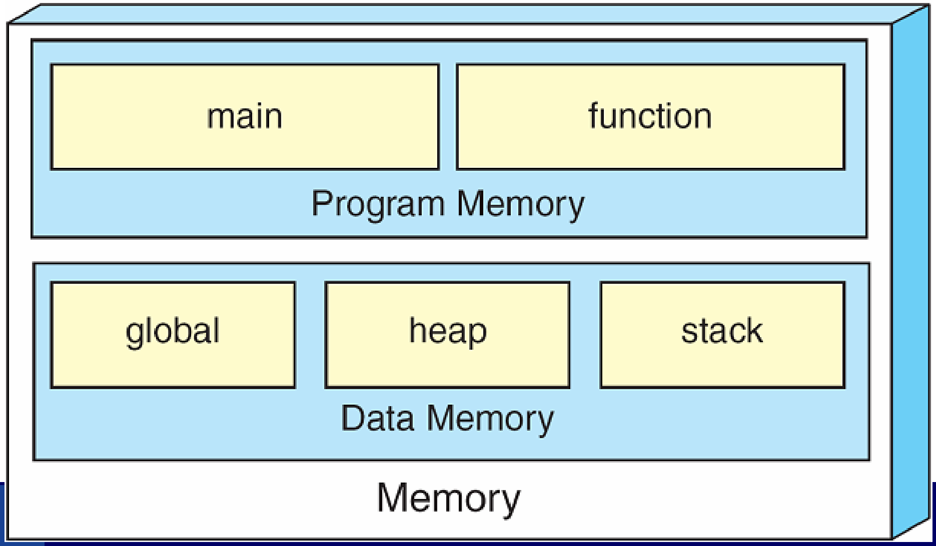
- Arrays and Pointers
- Pointer Arithmetic and Arrays
- Memory Allocation Functions

Memory Allocation Functions

- Memory allocation: allocation (reservation) of memory storage for use in a computer program during execution
  - Static allocation
    - Variable declaration
  - Dynamic allocation

Conceptual View of Memory

- Memory is divided into program memory and data memory
  - Program memory: program codes (instructions)
  - Data memory: data storage (variable, dynamic memory)
    - Global memory: global variables
    - Stack: local variables
    - Heap: dynamically allocated memory

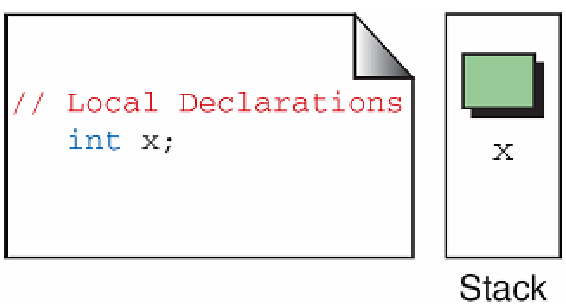


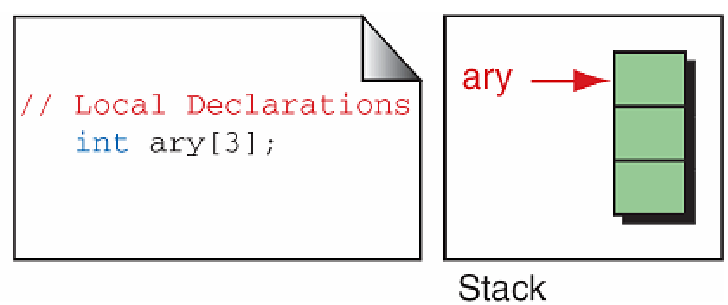
Static Memory Allocation

- Static memory allocation
  - Memory allocation through declarations in source code

    Ex) variables, array, pointers, streams, …
    - Allocated from stack (local variables) or global data memory (global variables)
- Limitations
  - Local variables: limited scope and lifetime

    Note! In general, stack size is limited.
  - Global variables: fixed size.





Example

- Goal: read a series of numeral data and store it in memory
  - # of data is decided by user
- Problems of solution using static allocation
  - If n < 100, storage is wasted.
  - If n > 100, program can crash.

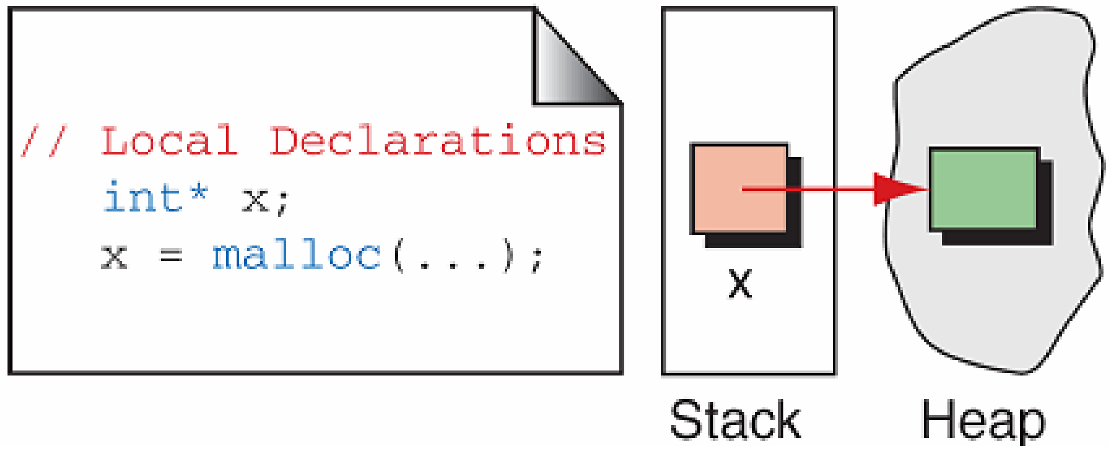
```
int main()
{
    int n = 0, i = 0;
    int data[100];

    printf("Input # of data:");
    scanf("%d", &n);

    for(i = 0; i< n; i++)
        scanf("%d", &data[i]);
    ...
    return 0;
}
```

Dynamic Memory Allocation

- Dynamic memory allocation
  - Memory allocation using predefined allocation functions
    - Size is dynamically determined
    - Allocated from heap



Bank vs. Heap

- Bank
  - Getting a loan
    - Loan application form
      - Amount of money
  - Using money
    - Account number
    - cash card, debit card
  - Redeeming the loan
    - Repayment application form
      - Account number, money

- Heap
  - Allocating memory
    - malloc() function
      - Size of memory block
  - Using memory
    - Address (pointer)
    - * or [] operator
  - Releasing memory
    - free() function
      - Address of the memory block

Example

- Static allocation

```
int main()
{
   int n = 0, i = 0;
   int array[100];
   int *data = array;

   printf("Input # of data:");
   scanf("%d", &n);

   for(i = 0; i< n; i++)
      scanf("%d", &data[i]);
   …
   return 0;
}
```

- Dynamic allocation

```
int main()
{
   int n = 0, i = 0;
   int *data = NULL;

   printf("Input # of data:");
   scanf("%d", &n);
   data = (int*)malloc(n*sizeof(int))

   for(i = 0; i< n; i++)
      scanf("%d", &data[i]);
   …
   free(data);
   return 0;
}
```

Memory Allocation Functions

- Allocation
  - void *malloc(size_t size);
    - Size: size of memory in bytes
      - size_t is defined in stdio.h (usually, unsigned int)
    - Returns value: pointer to allocated memory
      - If it fails, return NULL.
    - Allocated memory is not initialized

- Deallocation
  - void free(void *ptr);
    - Releases a memory block pointed by ptr, which was allocated by malloc, calloc, or realloc
    - The released memory block can be used for other purpose

Example

- Allocating a variable

```
int *p = (int*)malloc(sizeof(int));
*p = 10;
printf("p = %p, *p = %d₩n",
                          p, *p);
...
free(p);
```

cf.

```
int a;
Int *p = &a;
*p = 10;  // same with a == 10;
printf("p = %p, *p = %d₩n",
                          p, *p);
```

- Allocating an array

```
int n = 0;
int *a = NULL;

scanf("%d", n);
a = (int*)malloc(n * sizeof(int));
for(i = 0; i < 10; i++)
    a[i] = i;
...
free(a);
```

size can be determined dynamically

"int *a = (int*)malloc(10*sizeof(int));" is similar to "int a[10]"

Using Dynamic Memory Allocation

- Memory allocation/free functions are declared in stdlib.h

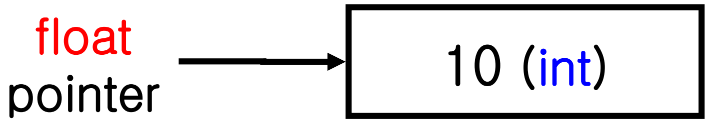
  Ex) #include <stdlib.h>

- All dynamically allocated memory blocks should be released.
  - Otherwise, the memory block is not available for other purpose. (memory leak)

Invalid Use of Pointer

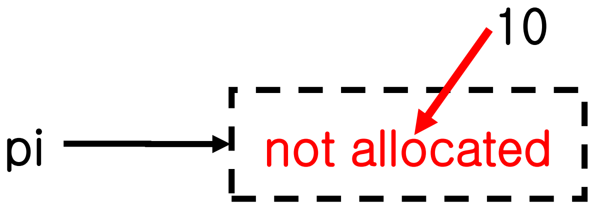
- Invalid type casting

```
Ex) int i = 10;
    int *pi = &i;
    float *pf = (float*) pi;      // can be dangerous
```



- Unassigned pointer

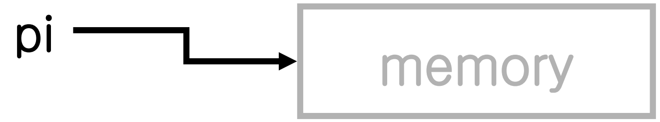
```
int *pi;
// pi = (int*) malloc(sizeof(int));    // forgot
*pi = 10;                               // error
```



Invalid Use of Pointer

- Dangling pointer

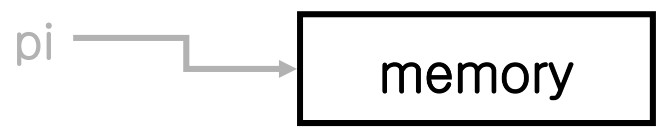
```
int *pi = malloc(sizeof(int));
*pi = 10;       // valid use
…
free(pi);
…
free(pi);       // error: pi is already deallocated
```



- Memory leak

```
{
   int *pi = NULL;
   pi = func(10);
   pi[0] = 10;
   …
   // free(pi);// forgot
}
```

```
int *func(int len)
{
   int *a = malloc(len*sizeof(int));
   …
   return a;
}
```



Recommendation

- Initialize every pointer at declaration

  Ex)

```
int *pi;            // bad
int *pi = NULL;     // good
```

- All memory allocated in a function should be deallocated before leaving that function.
  - Possible exceptions: Creator (constructor) / Destructor

- Set deallocated pointer variable by NULL

```
free(pi);
pi = NULL;          // free(NULL) is safe
```

Recommendation

- Always check the result of memory allocation

```
int *p = (int *)malloc(size_in_bytes);
if(p == NULL){
    printf("Failed to allocate memory in line %d.₩n", __LINE__);
    exit(-1);            // optional
}
```

  - Preprocessor macros for debugging
    - __LINE__ is a preprocessor macro that expands to current line number in the source file, as an integer
    - __FILE__ is a preprocessor macro that expands to full path to the current file.