Scope and Class

- A **global variable** is valid everywhere, and is always permanent.
- A **local variable**
  - has a scope that is limited to the *block* in which it is declared and cannot be accessed outside that block
  - allocated from a section of memory called the *stack* at the beginning of the block.

Functions

- Functions allow us to **group commonly used code** into a compact unit that can be used repeatedly.
- Each function should begin with a comment block containing *name, description, parameters, and return value*.
- C uses a form of parameter passing called **"Call by value"**.
- A function can have any number of parameters, including none.
Bit Operations

- A bit is the smallest unit of information. Normally, it is represented by the values 1 and 0.

Advanced Types

- Structures, unions, and enumerated types
- typedef
Structure

- Data Types for storing a group of different data types
- In a structure, each element or **field** is named and has its own data type.
- To access the field of structure, use `variable.field`

```c
struct bin{
    char name[30]; /* name of the part */
    int quantity; /* how many are in the bin */
    int cost; /* The cost of a single part (in cents) */
}; /* Type declaration */

struct bin printer_cable_bin; /* Variable declaration */
printer_cable_bin.cost= 12 ;
```
**typedef**

- C allows the programmer to define her own variable types through the `typedef` statement.

- `typedef int count;`
  - defines a new type `count` that is the same as an integer.
  - “`count flag;`” is equivalent to “`int flag;`”

```c
typedef struct complex_struct {
    double real;
    double imag;
} complex_struct;
```
**enum Type**

- The enumerated data type is designed for variables that contain only a limited set of values.
- These values are referenced by name (tag).

```c
enum week_day {SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
                THURSDAY,
                FRIDAY, SATURDAY};
/* now use it */
enum week_day today = TUESDAY;
```
**Pointer Variable**

- A variable which can store memory address of the data variable.

<table>
<thead>
<tr>
<th>C Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>thing</td>
<td>Simple thing (variable)</td>
</tr>
<tr>
<td>&amp;thing</td>
<td>Pointer to variable thing</td>
</tr>
<tr>
<td>thing_ptr</td>
<td>Pointer to an integer (may or may not be specific integer thing)</td>
</tr>
<tr>
<td>*thing_ptr</td>
<td>Integer</td>
</tr>
</tbody>
</table>
\textbf{A} \texttt{thing\_ptr = \&thing;}  

Assigns \texttt{thing}'s address to \texttt{thing\_ptr}.  

\textbf{B} \texttt{other = \*thing\_ptr;}  

Assigns to \texttt{other} the value at the address \texttt{thing\_ptr} carries.  

\textbf{C} \texttt{\*thing\_ptr = 6;}  

Assigns to a value to what \texttt{thing\_ptr} points to.
Pointers as Function Arguments

- C passes parameters using "call by value".
- Pointers can be used to support a variable of return value – “call by reference”.

NULL

- There is a special pointer called NULL.
- It points to nothing. (The actual numeric value is 0.)

```c
void inc_count(int *count_ptr) {
    (*count)++;
}
```
```
int array[5] ;
Int * arr_ptr ;
arr_ptr=&array[0];
```
"array == &array[0]"

"&array[0] == arr_ptr"

"arr_ptr == array"

"type * == type []"

Variable Name of Array Type ==
Pointer Variable of That Type !!!
Re “man strlen”
- See the type of parameter of `strlen`
- Similarly, see `strcmp`, `strcpy`, `strcat`, and so forth.
- What is the meaning of “char *” and “const char *”? 
Problem

- Suppose we are given a string of the form "Last/First." We want to split this into two strings, one containing the first name and one containing the last name.
- We need a function to find the slash in the name. The standard function \texttt{strchr} performs this job for us. \textbf{However we use the pointer.}
Implement this function.

- `char *my_strchr(char * string_ptr, char find)`
  - It checks character `find` from `char * string_ptr`. 
  - If `string_ptr` contains the character in it, function returns the pointer for the substring started from the pointer.
  - Unless, function returns NULL.
int main()
{
    char line[80]; /* The input line */
    char *first_ptr; /* pointer to the first name */
    char *last_ptr; /* pointer to the last name */
    fgets(line, sizeof(line), stdin);
    /* Get rid of trailing newline */
    line[strlen(line)-1] = '\0';
    last_ptr = line; /* last name is at beginning of line */
    first_ptr = my_strchr(line, '/'); /* Find slash */
    /* Check for an error */
    if (first_ptr == NULL) {
        fprintf(stderr,"Error: Unable to find slash in %s\n", line);
        exit (8);
    }
    *first_ptr = '\0'; /* Zero out the slash */
    ++first_ptr; /* Move to first character of name */
    printf("First:%s Last:%s\n", first_ptr, last_ptr);
    return (0);
}
struct mailing {
    char name[60]; /* last name, first name */
    char address1[60]; /* two lines of street address */
    char address2[60];
    char city[40];
    char state[2]; /* two-character abbreviation */
    long int zip; /* numeric zip code */
} list [MAX_ENTRIES];

/* Pointer to the data */
struct mailing *list_ptrs[MAX_ENTRIES];
int current; /* current mailing list entry */
for (current = 0; current = number_of_entries; ++current)
    list_ptrs[current] = &list[current];
/* Sort list_ptrs by zip code */
Review

- The declaration for a two-dimensional array is:

```c
int matrix[2][4]; /* a typical matrix */
```

- To access an element of the matrix, we use the notation:

```c
matrix[1][2] = 10;
```
Pointers & Multi-Dimensional Array

Memory Layout of 2-D Array

```c
int array2d[2][4];
```

```
&array2d[1][3]  array2d[1][3]
&array2d[1][2]  array2d[1][2]
&array2d[1][1]  array2d[1][1]
&array2d[1][0]  array2d[1][0]
&array2d[0][3]  array2d[0][3]
&array2d[0][2]  array2d[0][2]
&array2d[0][1]  array2d[0][1]
&array2d[0][0]  array2d[0][0]
```

array2d[1][*]
array2d[0][*]
**Pointers Representation of 2-Dimensional Array**

```c
int * arr_ptr;
Int array2d[2][4];
arr_ptr = &array2d[0][0] = array2d;
```

<table>
<thead>
<tr>
<th>Index</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>arr_ptr+1*4+3</td>
<td><em>(arr_ptr+1</em>4+3)</td>
<td>&amp; array[i][j] ==</td>
</tr>
<tr>
<td>arr_ptr+1*4+2</td>
<td><em>(arr_ptr+1</em>4+2)</td>
<td>array + i*row_size+ j</td>
</tr>
<tr>
<td>arr_ptr+1*4+1</td>
<td><em>(arr_ptr+1</em>4+1)</td>
<td></td>
</tr>
<tr>
<td>arr_ptr+1*4+0</td>
<td><em>(arr_ptr+1</em>4+0)</td>
<td></td>
</tr>
<tr>
<td>arr_ptr+0*4+3</td>
<td><em>(arr_ptr+0</em>4+3)</td>
<td></td>
</tr>
<tr>
<td>arr_ptr+0*4+2</td>
<td><em>(arr_ptr+0</em>4+2)</td>
<td></td>
</tr>
<tr>
<td>arr_ptr+0*4+1</td>
<td><em>(arr_ptr+0</em>4+1)</td>
<td>array[i][j] =</td>
</tr>
<tr>
<td>arr_ptr+0*4+0</td>
<td><em>(arr_ptr+0</em>4+0)</td>
<td><em>(array + i</em>row_size+ j )</td>
</tr>
<tr>
<td></td>
<td>*arr_ptr</td>
<td></td>
</tr>
</tbody>
</table>
Exercise: 2-Dimensional Bitmap Array

- Make 2-dim bitmap array and print its contents using pointer.
  - Bitmap array elements only consists of “0(zero)” or “1(one)”.
  - int bitmap_array[5][5] ;
  - int * arr_ptr ;
  - man rand()
    - It needs to include the standard library <stdlib.h>.
    - What does the function rand() mean?
    - Using modular (%) operator.
  - Using ‘for’ loop about each column, and row
If we access a pointer variable which does not indicate specific memory location (null pointer), the system signals a segmentation fault and the program exits.

We have only used a pointer as a reference of a variable.
- Now, we use a pointer itself as a variable which is dynamically allocated.
- It mainly is used to make array variable dynamically.
In order to use a pointer as a dynamically allocated variable, we must allocate a memory using standard functions `<stdlib.h>`.

- \texttt{void* malloc(size\_t size)};
- \texttt{void* calloc(size\_t nmemb, size\_t size)};
- \texttt{void* realloc(void* ptr, size\_t size)};
- \texttt{man !!!}

When we allocate memory dynamically, it cannot be de-allocated and reused unless we free the memory explicitly.

- \texttt{void free (void* ptr)};

So C uses \texttt{void} for two purposes

- When used as a type in a function declaration, \texttt{void} indicates that the function returns no value.
- When used in a pointer declaration, \texttt{void} defines a generic pointer.
Allocation Memory for Pointer

- Stack
- Heap
- Data segment
- Code Segment

* function call
* Declare Local variable

malloc

Dynamic Memory
Static Memory
void *memset (void *s, int c, size_t n);
- Fill memory with a constant byte.
- We can use this function in order to initialize memory with “c = 0” (zero=null).
main()
{
    /* Pointer to a string that will be allocated from the heap */
    char *string_ptr;
    string_ptr = malloc(80);
    memset(string_ptr, 0, 80);

    struct person {
        char name[30]; /* name of the person */
        char address[30]; /* where he lives */
        char city_state_zip[30]; /* Part 2 of address */
        int age; /* his age */
        float height; /* his height in inches */
    }
    /* Pointer to a person structure to be allocated from the heap */
    struct person *new_item_ptr;
    new_item_ptr = malloc(sizeof(struct person));
    memset(new_item_ptr, 0, sizeof(struct person));
}
const int DATA_SIZE = 1024; /* Number of bytes in the buffer */
void copy(void)
{
    char *str_ptr; /* Pointer to large data buffer */
    str_ptr = malloc(DATA_SIZE); /* Get the buffer */
    memset(str_ptr, 0, DATA_SIZE);
    /* copy your information to the str_ptr and print them. */

    free(str_ptr);
    str_ptr = NULL; // to prevent “dangling pointer”
}
** Pointers & Multi-Dimensional Array

- type * == type []
- type ** == type * [] == type [][]
- type *** == type [][][

```c
int ** arr_ptr;

int arr2d[2][4];
arr_ptr=arr2d; (?)
```
Set up 2-Dimensional Array using Multi-pointer Variable.

```
int ** arr_ptr; == int * arr_ptr[]
```

- `arr_ptr` is a pointer to an array of pointers.
- `arr_ptr[0]` points to the first row of the array.
- `arr_ptr[1]` points to the second row of the array.

Example:

```
*(arr_ptr+2) = arr_ptr[2]  // Points to arr_ptr[0][2]
*(arr_ptr+1) = arr_ptr[1]  // Points to arr_ptr[0][1]
*arr_ptr = arr_ptr[0]     // Points to arr_ptr[0][0]
```

Type: (int *)

```
*(arr_ptr+3) = arr_ptr[0][3]
*(arr_ptr+2) = arr_ptr[0][2]
*(arr_ptr+1) = arr_ptr[0][1]
```

Type: (int)
** Pointers & Multi-Dimensional Array

```c
type ** arr_ptr;  == type array [][];
```

```c
*(arr_ptr+2)  
  = arr_ptr[2]
```

```c
*(arr_ptr+1)  
  = arr_ptr[1]
```

```c
*arr_ptr  
  = arr_ptr[0]
```

```c
// allocation for the array of type
arr_ptr[i] = (type*) malloc (sizeof(type)*row);
```

```c
// allocation for the array of type *
arr_ptr =  
(type**) malloc(sizeof(type)*column);
```

```c
(*arr_ptr+3)  
  = arr_ptr[0][3]
```

```c
(*arr_ptr+2)  
  = arr_ptr[0][2]
```

```c
(*arr_ptr+1)  
  = arr_ptr[0][1]
```

```c
**arr_ptr = 
arr_ptr[0][0]
```
Freeing memory of Multi-dimensional Array

- Reverse.
- Because of preventing “idle memory”.
  - Think over case in which we free `arr_ptr` first.

```
// free for the array of type 
free(arr_ptr[i]) ;
```

```
**arr_ptr = arr_ptr[0][0]
(*arr_ptr+1) = arr_ptr[0][1]
(*arr_ptr+2) = arr_ptr[0][2]
(*arr_ptr+3) = arr_ptr[0][3]

*(arr_ptr+1) = arr_ptr[1]
*(arr_ptr+2) = arr_ptr[2]
*(arr_ptr+3) = arr_ptr[3]
```

```
// free for the array of type *
free(arr_ptr) ;
```
```c
void main()
{
    int ** arr_ptr;
    int columns=3, rows=4;
    int i,j;

    /* memory allocation */
    arr_ptr=(int**)malloc(sizeof(int*)*columns);
    for(i=0; i<columns ; i++) {
        arr_ptr[i]=(int*)malloc(sizeof(int)*rows);
    }

    /* assign value into a variable */
    for(i=0;i<columns ; i++) {
        for(j=0;j<rows;j++) {
            arr_ptr[i][j] = i*columns+j;
            printf("arr_ptr[%d][%d]:%d\n", i, j, arr_ptr[i][j]);
        }
    }
    /* memory deallocation */
    for(i=0; i<columns ; i++ )
        free(arr_ptr[i]);
    free(arr_ptr);
    return;
}
```
Exercise

- **Edit & compile & run the program**
  - make a 2-dimensional array using pointer dynamically. (char** arr_ptr) and `malloc`.
  - Make random number using standard function `rand()` ( `rand` ) and modular (%) operator ranged 0 to 1.
  - Assign random value (0 or 1) into the array elements using for loop.
  - Print element values to the screen using for loop. ( `array[0][1]=1\n` )
  - De-allocate array memory space using `free` function.
The procedure main actually takes two arguments.

- They are called argc and argv

- `void(int) main (int argc, char *argv[])`
  - The parameter `argc` is the number of arguments on the command line (including the program name)
  - The array `argv` contains the actual arguments.

```
./hello this is a test
```

```
main (int argc, char*argv[]) ...
hello.c
```

```
argc = 5
argv[0] = "./hello"
argv[1] = "this"
argv[2] = "is"
argv[3] = "a"
argv[4] = "test"
```
Make a program

- Get the command line arguments which are name and two integer value.
- Print the name.
- Print the sum of two integer value using atoi().
  ```
  man atoi()
  ```
- Report an error and Exit unless the count of arguments is three.
  
  ```
  /arg_test HongGilDong 10  5
  - Name is **HongGilDong**
  - Sum is **15**.
  ```
How pointers may be used

Linked list

Array

Tree

LOGO
Structures can contain pointers, even a pointer to another instance of the same structure.

```c
struct node {
    struct node *next_ptr; /* Pointer to the next node */
    int value; /* Data for this node */
}
```

This structure contains two fields, one named value, shown here as the section containing the number 2. The other is a pointer to another structure. The field next_ptr is shown as an arrow.
We had to use the cumbersome notation to access the data field of the structure.

\[ (*\text{current\_ptr}).\text{data} \]

C provides a shorthand for this construct using the structure pointer (\(-\rightarrow\)) operator.

- The dot (\(\cdot\)) operator indicates the field of a structure.
- The \(-\rightarrow\) indicates the field of a structure pointer.

\[ (*\text{current\_ptr}).\text{data} = \text{value}; \]
\[ \text{current\_ptr}-\rightarrow\text{data} = \text{value}; \]