Lesson 4: Pointers, Arrays and Memory Management
Pointers

- Pointer is variable whose value is address of some memory area of certain type
- Size of a pointer is same as that of int, but the size of memory area pointed by pointer is unknown (pointers do not have information about size)
- Declare pointers by * symbol:
  - int *pInt;
  - char *pChar;
  - struct Student *pSt;
- Access to value (indirection) also by *:
  - int aInt = *pInt; (*pInt is an int that pInt points to)
  - *pChar = 'A';
  - printf("Value: %d", *pInt);
Change address of pointers

- As value of a pointer is address, when changing its value, the pointer will point to another memory area
- Assign new address to a pointer by “=” operator as normal
  - int *pInt2;
  - pInt2 = pInt;
- Address-of operator &: produces a pointer by taking address of a variable
  - int a;
  - int* pA = &a;  /* pA points to a */
- & is reversion of *: for any variable a, *&a is equivalent to a; and for any pointer p, &*p is equivalent to p
Illustration

- `char c = 'A';`
- `int *pInt;`
- `short s = 50;`
- `int a = 10;`

```
    pInt = &a;
    *pInt = 100;
```

Addresses of variables in memory here is increasing only for illustration. In reality, variables are allocated in stack from higher memory to lower → first variable has highest address.

<table>
<thead>
<tr>
<th>Address</th>
<th>1500</th>
<th>1501</th>
<th>1502</th>
<th>1503</th>
<th>1504</th>
<th>1505</th>
<th>1506</th>
<th>1507</th>
<th>1508</th>
<th>1509</th>
<th>1510</th>
<th>1511</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable</td>
<td>char c</td>
<td>int* pInt</td>
<td>short s</td>
<td>int a</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value</td>
<td>'A'</td>
<td>1507</td>
<td>50</td>
<td>100</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

`pInt: 1507`

`*pInt: 100`

`&a: 1507`

`a: 100`
Void pointers (void*)

- Are pointers without type information
- Can be implicitly casted to any other pointer type, and vice versa (but not in C++)
  - void* pVoid; int *pInt; char *pChar;
    ```c
    pInt = pVoid; /* OK */
    pChar = pVoid; /* OK */
    
    pVoid = pInt; /* OK */
    pVoid = pChar; /* OK */
    
    pChar = pInt; /* error */
    pChar = (char*)pInt; /* OK */
    ```
- Indirection cannot be used with void* pointers
  - *pVoid /* error */
- void* pointers are used to access pure memory, or to manipulate variables of unknown type
  - memcpy(void* dest, const void* src, int size);
Null pointer (NULL)

- Is a pointer constant with address of 0, type of void*, which signifies a pointer that does not point to any memory
- Is macro declared as follows by nature:
  ```
  #define NULL   ((void*)0)
  ```
- Impossible to assign values to null pointers
  ```
  int *pInt = NULL
  *pInt = 100;    /* error */
  ```
- Distinguish null pointers with uninitialized ones (which point to random memory)
- Usually used to determine the validity of a pointer variable → to avoid errors, always assign NULL to pointer variables when unused
- As NULL is evaluated to 0, comparing a pointer to NULL can be ignored in logical expressions:
  ```
  if (p != NULL) ... → if (p) ...
  ```
# Pointer operations

- **Increment/decrement:** pointer points to the next/previous element (address increased/decreased by the size of the pointer type)

<table>
<thead>
<tr>
<th>Address</th>
<th>1500</th>
<th>1501</th>
<th>1502</th>
<th>1503</th>
<th>1504</th>
<th>1505</th>
<th>1506</th>
<th>1507</th>
<th>1508</th>
<th>1509</th>
<th>1510</th>
<th>1511</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p--</td>
<td></td>
<td></td>
<td>1502</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>short *p</td>
<td></td>
<td></td>
<td></td>
<td>1504</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p++</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1506</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Addition/subtraction:** similarly

<table>
<thead>
<tr>
<th>Address</th>
<th>1500</th>
<th>1501</th>
<th>1502</th>
<th>1503</th>
<th>1504</th>
<th>1505</th>
<th>1506</th>
<th>1507</th>
<th>1508</th>
<th>1509</th>
<th>1510</th>
<th>1511</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p-2</td>
<td></td>
<td></td>
<td>1502</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>short *p</td>
<td></td>
<td></td>
<td></td>
<td>1504</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p+3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1510</td>
<td></td>
</tr>
</tbody>
</table>

- **Comparison:** 2 pointers of same type can be compared using address like comparing two integers (greater, smaller, equal,...)

- A pointer can be subtracted to another of the same type
Pointers and arrays

- Array is static pointer (whose address is fixed), which points to (stores the address of) its first element
  - It is possible to manipulate arrays like pointers, except changing their address
  - `int arr[] = {1, 2, 3, 4, 5};
    int x;
    *arr = 10; /* equivalent to: arr[0] = 10; */
    printf("%d", *(arr+2)); /* equivalent to: arr[2] */
    arr = &x; /* error */`

- Pointer can be used as array and be manipulated as array
  - `int *p = arr;
    p = arr+2; /* same as: p = &arr[2]; */
    p[0] = 30; /* same as: arr[2] = 30; or: *p = 30; */`

- Conclusion: pointers and arrays can be used interchangeably, depending on programmer’s convenience
Pointers and arrays (cont.)

- Differences:
  - Impossible to assign new address to an array variable
  - Memory is allocated for array elements immediately at declaration (in stack)
  - `sizeof()` operator returns real size for arrays (total size of elements), while returns size of address for pointers
    - `float arr[5];` \(\rightarrow\) `sizeof(arr)` returns 20 (5*4)
    - `float* p = arr;` \(\rightarrow\) `sizeof(p)` returns 4 for 32-bit platforms
    - `sizeof(arr)/sizeof(arr[0])` \(\rightarrow\) number of elements
  - With pointer nature, it is possible to use negative index for arrays:
    - `int arr[] = {1, 2, 3, 4, 5};`
    - `int *p = arr + 2;`
    - `p[-1] = 10;` /* same as: `arr[1] = 10; */
Pointers to pointers

- It is possible to declare pointers to pointers in C
  ```c
  float x = 1.5;
  float *pX = &x;
  float **ppX = &pX;
  printf("%f", **ppX); /* output: 1.5 */
  **ppX = 2.3;
  ```

- Can be considered as 2-dimensional arrays, arrays of arrays, pointers to arrays, or arrays of pointers
Strings of characters

- Are arrays of characters, terminated by '\0'
  - `char name[10] = "Tung";`  (initialized by pointer)
  - `char name[10] = {'T', 'u', 'n', 'g', '\0'};`  (by array)
  - `char *name = "Tung";`  (pointer initialized by pointer)
  - `char *name = {'T', 'u', 'n', 'g', '\0'}; /* error */`

- Ex: calculate length of string:
  - `for (n=0; s[n]; n++) ; /* more preferable */`
  - `for (n=0; *s; n++, s++) ;`

- Some popular string functions:
  - `#include <string.h>`
  - `int strlen(s)`  → length of s
  - `char *strcpy(dst, src)`  → copy string src to dst
  - `char *strcat(dst, src)`  → concatenate src to the end of dst
  - `int strcmp(str1, str2)`  → compare 2 strings, result: 1, 0, -1
  - `char *strstr(s1, s2)`  → find position of s2 in s1
Command line processing

- Arguments (or parameters) can be passed to programs from command line
  - C:\>movefile abc.txt Documents

- Declaration of `main()` function
  - `int main(int argc, char* argv[]) { ... }`
    - `argc`: number of arguments (`argc ≥ 1`)
    - `argv`: array of arguments (as array of strings)

- Path and name of the program file is always the first argument
  - In above example:
    - `argc`: 3
    - `argv`: `[ "movefile", "abc.txt", "Documents" ]`
Dynamic memory allocation

- Memory is allocated for variables at declaration (in stack)
- When memory need to be allocated on demand (size unknown at writing time) → dynamic allocation (in heap)

Allocate memory:
- #include <stdlib.h>
- void* malloc(int size) /* size: number of bytes */
- int *p = (int*)malloc(10*sizeof(int)); /* alloc 10 int*/

- void* calloc(int num_elem, int elem_size)
- void* realloc(void* ptr, int size)
- If allocation fails, returns NULL → need to check

Release (deallocate) an allocated memory:
- void free(void* p);
- free(p);
Pointers to struct, union

- For a pointer to struct or union, possible to use “->” operator to access member variables instead of “*” and “.”
  - p->member is equivalent to (*p).member

Ex:

- typedef struct {
    int x, y;
} Point;
Point *pP = (Point*)malloc(sizeof(Point));
pP->x = 5; /* same as: (*pP).x = 5; */
(*pP).y = 7; /* same as: pP->y = 7; */
Common errors in using pointers

- Normally, programs are not allowed to access to outside of memory allocated to them → must examine address that pointers store

Consequences:

- Never use uninitialized pointers → assign NULL to pointers when not used, so that we can check if they have been initialized or not
- Only assign address of valid memory (static variables or allocated memory) to pointers to ensure that pointers always store valid address
- Must examine length of pointed memory so that program does not access to outside of allocated memory (buffer overflow error)
- When an allocated memory is not used any more, release it
Conclusion

- Pointer is a powerful notion of C compared to other languages, but is a double blade knife, as it is hard to debug pointer related errors → programmers must master and use pointers flexibly

- Pointers are usually used in following contexts:
  - Passing values from function to outside via parameters
  - Function pointers
  - Data structures: linked list, queue, dynamic array,…

→ Will come back on related chapters
Problems

Write programs to:

1. Enter an integer N, allocate memory for N integers and input data, then print the array in reverse order

2. Input an array of floating point numbers without knowing nor asking for the number of elements a priori (extend the array on demand)

3. Enter a string s1, then copy it to another string s2 (without using strcpy nor memcpy)

4. Enter two strings s1 and s2, then compare to see if they are equal (without using strcmp)

5. Read a string from program argument, then split it into an array of words

6. Declare two array of increasing float values, then merge them into another array also in increasing order