REVIEW

Lecture notes of the course "*Programming Techniques*"

Lê Hồng Phương[1]

[1]Department of Mathematics, Mechanics and Informatics
VNU University of Science, Hanoi
<*phuonglh@gmail.com*>

09/2012

Content

1. Review

2. Pointers and memory addresses
   - Physical and virtual memory
   - Addressing and dereferencing

3. Arrays and pointer arithmetics

4. Simple algorithms for sorting an array
   - Bubble sort
   - Selection sort

Content

1. Review
2. Pointers and memory addresses
   - Physical and virtual memory
   - Addressing and dereferencing
3. Arrays and pointer arithmetics
4. Simple algorithms for sorting an array
   - Bubble sort
   - Selection sort

Content

1. Review
2. Pointers and memory addresses
   - Physical and virtual memory
   - Addressing and dereferencing
3. Arrays and pointer arithmetics
4. Simple algorithms for sorting an array
   - Bubble sort
   - Selection sort

Content

1. Review
2. Pointers and memory addresses
   - Physical and virtual memory
   - Addressing and dereferencing
3. Arrays and pointer arithmetics
4. Simple algorithms for sorting an array
   - Bubble sort
   - Selection sort

Review: I/O functions

- I/O provided by `stdio.h`, not language itself
- Character I/O: `putchar()`, `getchar()`, `getc()`, `putc()`, etc.
- String I/O: `puts()`, `gets()`, `fgets()`, `fputs()`, etc.
- Formatted I/O: `fprintf()`, `fscanf()`, etc.
- Open and close files: `fopen()`, `fclose()`
- File read/write position: `feof()`, `fseek()`, `ftell()`, etc.

Review: `printf()` and `scanf()`

- Formatted output:

  `int printf(char* format, arg1, arg2, ...)`

- Format specification:
  - `%[flags][width][.precision][length]<type>`
  - Types: `d, i (int); u, o, x, X (unsigned int); e, E, f, F, g, G (double); c (char); s (string)`
  - `flags, width, precision, length` – modify meaning and number of characters printed
- Formatted input: `scanf()` – similar form, takes pointers to arguments (except strings), ignores whitespace in input

Review: string and character arrays

- Strings are represented in C as an array of characters (`char[]`).
- Strings must be null-terminated (`'\0'` at end).
- Declaration:
  - `char str[] = “I am a string.”;`
  - `char str[20] = “I am a string.”;`
- `strcpy()` – function for copying one string to another.

Pointers and addresses

- Pointer: memory address of a variable
- Address can be used to access/modify a variable from anywhere.
- Extremely useful, especially for data structures
- Well-known for obfuscating code

Content

1. Review
2. Pointers and memory addresses
   - Physical and virtual memory
   - Addressing and dereferencing
3. Arrays and pointer arithmetics
4. Simple algorithms for sorting an array
   - Bubble sort
   - Selection sort

Physical and virtual memory

- Physical memory: physical resources where data can be stored and accessed by your computer
  - Cache
  - RAM
  - Hard disk
  - Removable storage
- Virtual memory: abstraction by OS, addressable space accessible by your code

Physical memory consideration

- Different sizes and access speeds
- Memory management – major function of OS
- Optimization – to ensure your code make the best use of physical memory available
- OS moves around data in physical memory during execution

Virtual memory

- *How much physical memory do I have?*

  2MB (cache) + 2GB (RAM) + 256GB (hard drive) + $\cdots$
- *How much virtual memory do I have?*
  - Less than 4GB on a 32-bit OS, typically 2GB for Windows, 3-4GB for Linux
  - Virtual memory maps to different parts of physical memory.
- Usable parts of virtual memory: stack and heap
  - **Stack**: where declared variables go
  - **Heap**: where dynamic memory goes

Content

1 Review

2 Pointers and memory addresses
- Physical and virtual memory
- Addressing and dereferencing

3 Arrays and pointer arithmetics

4 Simple algorithms for sorting an array
- Bubble sort
- Selection sort

Addressing variables

- Every variable residing in memory has an address.
- *What doesn't have an address?*
  - Register variables
  - Constants, literals, preprocessor defines
  - Expressions (unless result is a variable)
- *How to find an address of a variable?* The `&` operator.
- Address of a variable of type `t` has type `t*`.

```
int n = 4;
double pi = 3.14159;
int *pn = &n;       /* address of integer n */
double *ppi = &pi; /* address of double pi */
```

Dereferencing pointers

- I have a pointer, now what?
- Accessing/modifying addressed variable: dereferencing/indirection operator `*`:

```
printf("pi = %g\n", *ppi);
*ppi = *ppi + *pn;
```

- Dereferenced pointer is like any other variable.
- Null pointer (`0`, `NULL`): pointer that does not reference anything.

Casting pointers

- Can explicitly cast any pointer to any other pointer type

```
ppi = (double*)pn; /* pn originally of type (int*) */
```

- Implicit cast to/from `(void*)` is also possible (*more next weeks*)
- Dereferenced pointer has new type, regardless of real type of data
- Possible to cause segmentation faults, other difficult-to-identify errors
- What happens if we dereference `ppi` now?

Accessing caller variables

- Want to write function to swap two integers
- Need to modify variables in caller to swap them
- Pointers to variables as arguments

```
void swap(int *x, int *y) {
  int temp = *x;
  *x = *y;
  *y = temp;
}
```

- Calling `swap()` function:

```
int a = 5, b = 7;
swap(&a, &b);
/* now a = 7, b = 5 */
```

Variables passing out of scope

What is wrong with this code?

```
char *get_message() {
  char msg[] = "Aren't pointers fun?";
  return msg;
}

int main() {
  char *string = get_message();
  puts(string);
  return 0;
}
```

Arrays and pointers

- Primitive arrays implemented in C using pointer to block of contiguous memory.
- Consider array of 8 ints: `int arr[8];`
- Accessing `arr` using array entry operator: `int a = arr[0];`
- `arr` is like a pointer to element 0 of the array:

```
int *pa = arr;
int *pa = &arr[0];
```

- Not modifiable/reassignable like a pointer.

The `sizeof()` operator

- For primitive types/variables, size of type in bytes:

```
int s = sizeof(char);   /* == 1 */
double f;               /* sizeof(f) == 8 (64-bit OS) */
```

- For primitive arrays, size of arrays in bytes:

```
int arr [8];    /* sizeof(arr) == 32 (64-bit OS) */
long arr [5];   /* sizeof(arr) == 40 (64-bit OS) */
```

- Array length (need to be on one line when implemented):

```
#define array_length (arr) (sizeof (arr) == 0 ? 0 :
    sizeof(arr)/sizeof((arr)[0]))
```

The sizeof() operator

- For primitive types/variables, size of type in bytes:

```
int s = sizeof(char);   /* == 1 */
double f;               /* sizeof(f) == 8 (64-bit OS) */
```

- For primitive arrays, size of arrays in bytes:

```
int arr [8];    /* sizeof(arr) == 32 (64-bit OS) */
long arr [5];   /* sizeof(arr) == 40 (64-bit OS) */
```

- Array length (need to be on one line when implemented):

```
#define array_length (arr) (sizeof (arr) == 0 ? 0 :
        sizeof(arr)/sizeof((arr)[0]))
```

The `sizeof()` operator

- For primitive types/variables, size of type in bytes:

```
int s = sizeof(char);   /* == 1 */
double f;               /* sizeof(f) == 8 (64-bit OS) */
```

- For primitive arrays, size of arrays in bytes:

```
int arr [8];    /* sizeof(arr) == 32 (64-bit OS) */
long arr [5];   /* sizeof(arr) == 40 (64-bit OS) */
```

- Array length (need to be on one line when implemented):

```
#define array_length (arr) (sizeof (arr) == 0 ? 0 :
        sizeof(arr)/sizeof((arr)[0]))
```

Pointer arithmetic

- Suppose `int *pa = arr;`
- Pointer is not an `int`, but can add or subtract an `int` from a pointer:

  `pa + i` points to `arr[i]`
- Address value increments by `i` times size of data type:
  - Suppose `arr[0]` has address `100`, then `arr[3]` has address `112`.
- Suppose `char *pc = (char*)pa`, what value of `i` satisfies `(int*)(pc+i) == pa + 3`?

Sorting an array

- Sorting is one of the fundamental problems in computer science.
- A sorting algorithm is an algorithm that puts elements of a list in a certain order.
- The most-used orders are numerical order and lexicographical order.
- Efficient sorting is important for optimizing the use of other algorithms (such as searching and merging algorithms).
- The sorting problem has attracted a great deal of research due to the complexity of solving it efficiently.

Sorting an array

- There are many sorting algorithms, many of them provide a gentle introduction to a variety of core algorithm concepts.
- Although many people consider that is is a solved problem, useful new sorting algorithms are still being invented, for example *library sort* was first published in 2004.
- Common and well-known sorting algorithms: *bubble sort, selection sort, insertion sort, quicksort*, merge sort, heap sort.

Content

1 Review

2 Pointers and memory addresses
- Physical and virtual memory
- Addressing and dereferencing

3 Arrays and pointer arithmetics

4 Simple algorithms for sorting an array
- Bubble sort
- Selection sort

Bubble sort

- Bubble sort is a simple sorting algorithm.
- *How does it work?*
  - Start at the beginning of the array, compares the first two elements and if the first is greater than the second, it swaps them.
  - Continue doing this for each pair of adjacent elements to the end of the array.
  - Starts again with the first two elements, repeating until no swaps have occurred on the last pass.
- *In the worst case, how many swaps this algorithm needs to sort an array of n elements?*

Bubble sort

```
void bubbleSort(int a[], int n) {
  int i, j;

  for (i = (n - 1); i > 0; i--) {
    for (j = 1; j <= i; j++) {
      if (a[j - 1] > a[j]) {
        swap(&a[j-1], &a[j]);
      }
    }
  }
}
```

Examples: `a = {5, 1, 4, 2, 8}`; `a = {25, 17, 31, 13, 2}`

Bubble sort

- Bubble sort is rarely used to sort unordered large data sets because of its high time complexity ($O(n^2)$). It can be used to sort small data sets.
- It is also efficiently used on an array that is already sorted except for a very small number of elements.
  - If only one element is not in order, bubble sort will take only $2n$ time.
  - If two elements are not in order, it will take only $3n$ time.

Content

1 Review

2 Pointers and memory addresses
- Physical and virtual memory
- Addressing and dereferencing

3 Arrays and pointer arithmetics

4 Simple algorithms for sorting an array
- Bubble sort
- Selection sort

Selection sort

- Selection sort is an in-place comparison sort.
- It has the same complexity as bubble sort, making it inefficient on large data sets.
- *How does it work?*
  - Find the minimum value of the array
  - Swap it with the value in the first position
  - Repeat this process for the remainder of the array
- Selection sort does no more than n swaps and thus is useful when swapping is very expensive.

Selection sort – first implementation

```
void selectionSort(int a[], int n) {
  int i, j;

  for (i = 0; i < n - 1; i++) {
    1) find j: a[j] = min{a[i+1]...a[n-1]}
    2) swap(&a[i],&a[j]);
  }
}
```

Selection sort – second implementation

```
void selectionSort(int a[], int n) {
  int i, j;

  for (i = 0; i < n - 1; i++) {
    for (j = i+1; j < n; j++) {
      if (a[i] > a[j]) {
        swap(&a[i],&a[j]);
      }
    }
  }
}
```

What is the difference between the two implementations?

Exercises

- Set up Eclipse IDE for C/C++ programming
- Implement two sorting algorithms:
  - Bubble sort
  - Selection sort (two implementations)
- Techniques:
  - Using array
  - Using pointers

Summary

- Review of variables and pointers
- Arrays and pointers
- Two simple algorithms for sorting an array of numbers