CMPUT 201: Practical Programming Methodology

Guohui Lin
guohui@ualberta.ca
Department of Computing Science
University of Alberta
October 2019

Instructions (during all exams):

- Read these instructions and wait for the signal to turn this cover-sheet over.
- Use space below/beside the questions to write your solutions legibly.
- Closed book;
  - no electronic devices (make sure your cellphone is OFF),
  - no calculators,
  - no conversations.
- In general, no questions will be answered during the quiz;
  - if unsure, state your best assumptions clearly and proceed;
- We will provide an exam clock.

Problem 1 (5 marks)

- Consider the following C program that reads in a positive integer.

```
#include <stdio.h>

int main(void) {
    int n;
    printf("Enter a positive integer: ");
    scanf("%d", &n);
    while (n > 1) {
        printf("%d ", n);
        if (n % 2 == 0)
            n = n / 2;
        else
            n = n * 3 + 1;
    }
    printf("%d\n", n);
    return 0;
}
```

- Trace for $n = 7$ to obtain the output of the program.

- For an input n, suppose this `while`-loop terminates, then what is the last printed value? why?

Problem 2 (5 marks)

- Re-write the following function `pb` to remove the recursion:

```
void pb(int n) {
    if (n != 0) {
        pb(n / 2);
        putchar('0' + n % 2);
    }
}
```

Problem 1 (5 marks)

- Trace for $n = 7$ to obtain the output of the program:

  `7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1`

- For an input n, suppose this `while`-loop terminates, then what is the value in the last `printf()`? why?

  `1.`

  The reasons:

  If `(n % 2 == 0)` "`n = n / 2`" is at least 1, otherwise "`n = n * 3 + 1`" is at least 10.

Problem 2 (5 marks)

- Re-write the following function pb to remove the recursion:

```
void pb(int n) {
    if (n != 0) {
        pb(n / 2);
        putchar('0' + n % 2);
    }
}
```

- ```
#include <stdio.h>

void pb(int n) {
    int length = 0,   // length denotes the number of digits
        i = n;
    while (i) {
        length++;
        i /= 2;
    }
    char digit[length];   // variable-length array
    i = length - 1;
    while (n) {
        digit[i--] = '0' + n % 2;
        n /= 2;
    }
    for (i = 0; i < length; i++)
        putchar(digit[i]);
    return;   // always 'return'!
}
```
```

Agenda:

- One of the most important features
- Pointer variables
  - for holding a memory address (`unsigned long int`)
- The address (`&`) and indirection (`*`) operators
- Pointer assignment
- Pointers as arguments
- Pointers as return values

Reading:

- Textbook: Chapter 11

Machine-level representation:

- Main memory divided into **bytes**
  - a byte stores 8 bits of information

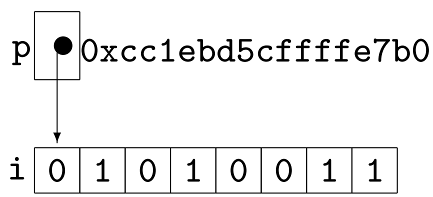
0	1	0	1	0	0	1	1

  - each byte has a unique address, e.g., `0xcc1ebd5cffffe7b0`
  - in hexadecimal
  - recall that every `int` uses 4 (consecutive) bytes: `0xcc1ebd5cffffe7b0`$\sim$`...3`
- An executable program: code + data
  - each variable occupies one or more bytes
  - the address of the first byte is the address of the variable

Pointer variables:

- The addresses are represented by (hexadecimal, usually) numbers
  - `unsigned long int` (max 4.3×10^9? no worries!)
  - its range is machine-dependent (not necessarily starting from 0)
  - not stored in int-type variables
- But stored by **pointer variables**
- e.g., the address of variable `i` is stored in a pointer variable `p`

  "`p` points to `i`"



- Declaration: "`p` is a pointer to an `int` object" (in brief, a type `int` pointer)

```
int *p;
double *q;
char *r;

int **p;
```

Operators for use with pointers:

- Basically **two**, getting the address vs. getting the object (or content, value)
- `&` — getting the address of a variable
- `*` — getting the object pointed to by a pointer
- **The address operator `&`:**

```
int i, *p;
...

p = &i;
scanf("%d", p);
```

1. assigning the address of `i` to the variable `p`
2. `p` points to `i`
3. store an integer into the memory address specified by `p`

Pointer variables vs. pointer constants:

- For example,

```
int i, *p;
...

p = &i;
scanf("%d", p);
```

- `p` is a variable, of which the value can be changed

```
p = &i;
```

- `&i` is a constant – illegal to do the following:

```
&i = p;
```

The indirection operator *:

- Access the content stored in the object pointed to by a pointer

```
int i, *p;
...

p = &i;
printf("%d", *p);
```

- Conclusion, mathematically,

```
int i, j, *p;

j = *&i; /* the same as j = i; */

*p = 1;  /* wrong */
```

- Cares:
  - a declared pointer is **not** automatically initialized (`p` does not have a value yet)
  - **cannot** access the content `*p` if `p` is not initialized

Pointer assignment:

- Be careful of the pointer types
- `p = q; /* copies the content/value of pointer q into pointer p */`

  Effects:

  - `*p` and `*q` are the **always** same (`p` and `q` point to the same address)
  - different from the assignment `*p = *q`,

    only copies the content of the object pointed by `q` into the object pointed by `p`

    (`p` and `q` not necessarily point to the same address)

- A common memory leak (not an error, but should be avoided):
  - by pointer assignment: `p = q;`
  - the old memory address of `p` might have no way to re-gain
  - neither has it been returned back to the OS

Pointers as arguments:

- Recall: arguments pass by value to parameters

```
void decompose(double x, long int_part, double frac_part) {

    int_part = (long) x;
    frac_part = x - int_part;
}
```

Question: How do we use the two parts **outside** of the function `decompose`?

  - by declaring the function to return a value?
  - by using an array?
  - by using two arrays?

- By using two arrays each of length 1 !
  - an array argument passes its address to an array parameter
  - so, it is a pointer!

```
void decompose(double x, long *int_part, double *frac_part) {

    *int_part = (long) x;
    *frac_part = x - int_part;
}
```

Pointer variables:

- Swapping `a` and `b`, if necessary, such that `a <= b`:

```
if (a > b) swapping(a, b);

void swapping(int a, int b) {

    int temp;

    temp = a;
    a = b;
    b = temp;
    return;
}
```

- A correct version is

```
if (a > b) swapping(&a, &b);

void swapping(int *a, int *b) {

    int temp;

    temp = *a;
    *a = *b;
    *b = temp;
    return;
}
```

Pointer as a return value:

- Use `const` to protect the object **(if you really really want !!!)**

```
void swapping(const int *a, const int *b) {

    int temp;

    temp = *a;
    *a = *b;    /* wrong */
    ...
}
```

- Swapping `a` and `b`, if necessary, such that `a <= b`; **and** returns a pointer to the larger one:

```
int *swapping(int *a, int *b) {

    int temp;

    temp = *a;
    *a = *b;
    *b = temp;
    return b;
}
```

- Functions returning a pointer is common
  - no pointer to an automatic local variable can be returned !

Lecture 17: Pointers and Arrays

Agenda:

- Pointer arithmetic
- Using pointers to process arrays
- Using array name as a pointer
- Pointers and multidimensional arrays
- Pointers and variable-length arrays

Reading:

- Textbook: Chapter 12

A close relationship:

- Pointers and arrays
- A critical element/feature of C
  - in the design
  - in many existing C programs
- Initially for **efficiency**
  - not that important any more due to
  - improvements in machine and compiler

Pointer arithmetic:

- A pointer can certainly points to an array element

```
int a[10], *p, *q, i, j;

p = &a[0];
*p = 5;
```

- Three supported arithmetic operations on pointers
  - adding an integer (what does “1” mean?)
  - subtracting an integer
  - subtract one pointer from another
  - **care**: must both point to elements of the same array

```
p = &a[i];
q = p + 3; /* q points to a[i+3] */
p += 6;    /* p points to a[i+6] */
p -= 3;    /* p points to a[i+3] */

p = &a[i];
q = &a[j];
printf("p - q = %d\n", p - q); /* the same as i - j */
```

Pointer arithmetic:

- A pointer can certainly points to an array element

```
int a[10], *p, *q, i, j;

p = &a[0];
*p = 5;
```

- Three supported arithmetic operations on pointers
  - adding an integer (what does "1" mean?)
  - subtracting an integer
  - subtract one pointer from another
  - **care**: must both point to elements of the same array
- Yes, you may surely do comparison too: `if (p <= q) {...}`

Using pointers for array processing:

- For example,

```
int a[10], *p, sum;

sum = 0;
for (p = &a[0]; p < &a[10]; p++)
    sum += *p;
```

- Did you notice?
  - what? Are we using `&a[10]` ?
  - `a[10]` does not exist
  - but no problem with `&a[10]`, the address of the memory unit after `a[9]`

    (Recall: C does not do out-of-range check.)
  - we **do not attempt** to get the object !

    (otherwise we might get "**segmentation fault**" or something meaningless)

The indirection operator *:

- Okay, what does `*p++` mean?

  ```
  int a[10], *p, i;

  p = &a[i];
  a[i++] = j; /* or equivalently the following ? */

  *p++ = j;
  ```

- `++` has a higher precedence over `*`

  ```
  *p++ = j;  /* is the same as *(p++) = j */
  ```

- That is,
  - `a[i]` is assigned value `j`
  - and afterwards, `p` points to `a[i+1]`

- Then, what is `(*p)++ = j;` ?

Combining operators * and ++:

- `int a[10], *p, i;`
  - `*p++` or `*(p++)`: value of expression is `*p` before increment
  - `(*p)++`: value of expression is `*p` before increment
  - `*++p` or `*(++p)`: value of expression is `*p` after increment
  - `++*p` or `++(*p)`: value of expression is `*p` after increment
  - Q: what is incremented in each case?
  - always use parentheses to avoid confusion ...
- For example,

```
int a[10], *p, sum;

sum = 0;
for (p = &a[0]; p < &a[10]; p++)
    sum += *p;

sum = 0;
p = &a[0];
while (p < &a[10])
    sum += *p++;
```

Using array name as a pointer:

- To further simplify the connection ...
- **The array name can be used as a pointer to the first element**

```
int a[10], *p, sum;

*a = 7;           /* stores 7 in a[0] */
*(a + 1) = 12;    /* stores 12 in a[1] */

sum = 0;
for (p = &a[0]; p < &a[10]; p++)
    sum += *p;

sum = 0;
for (p = a; p < a + 10; p++)
    sum += *p;
```

- But you **<u>cannot</u>** change the value of `a` — “protected” by OS (`const`)

  Imagine the value (address) is assigned by OS

Using a pointer as an array name:

- This inverse way is feasible too

```
int a[10], *p, sum, i;

p = a;
sum = 0;
for (i = 0; i < 10; i++)
    sum += p[i];
```

- That is,
  - `p[i]` and `*(p+i)` are the same thing

- **For the best programming practice: use the most precise variable(s)**

Array arguments:

- When passed to a function, an array name is **always** treated as a pointer
- Pass-by-value: the value is the address of the array
- Consequently, the values of the array elements can be changed

```
...
void quicksort(int a[], int left, int right);
int split(int a[], int left, int right);
...
```

- This is exactly what we have seen earlier

Pointers and multidimensional arrays:

- Recall that how a 2-dimensional array is stored — row-major order

```
int a[10][20], *p, *q;
int row, col;

    p = &a[0][0]; /* p points to the first element */
    q = p + 15;   /* q points to a[0][15] */
    q = p + 25;   /* q points to a[1][5] */
```

  – `a[10][20]` **is regarded as a 1-dimensional array of 10 elements**
  – each element is an array (1-dimensional, of length 20)

- Recall that the array name can be used as a pointer, pointing to the first element, i.e. `a` points to `a[0]`, or `a == &a[0]`

- It is valid to assign:

```
    p = &a[i][0]; /* p points to the first element */
```

  or

```
    p = a[i];     /* p points to the first element */
```

  but **invalid** to

```
    p = a;        /* try to let p points to the first row ? */
```

Pointers and multidimensional arrays:

- Using the array name as a pointer
  - in order to assign

    ```
    p = a;        /* try to let p points to the first row ? */
    ```

  - declare `p` as a pointer to a type int array of length 20: `int (*p)[20]`

    (`int *p[20]` declares `p` as an array of 20 type int pointers)

  - or declare `p` as a pointer to a type int pointer

```
int a[10][20], (*p)[20], **q;
int row, col;

p = a;            /* the same as p = &a[0] */
(*p)[i] = a[0][i];

q = a;
*q = a[0];        /* the same as *q = &a[0][0] */
**q = a[0][0];
...
```

Pointers and variable-length arrays (VLAs):

- Yes, pointers are allowed to point to elements of VLAs

```
void f(int n) {
    int a[n], *p;

    p = a;          /* the same as p = &a[0] */
    ...
}
```

- And even to more than one dimension:

```
void f(int m, int n) {
    int a[m][n], (*p)[k]; /* p is a pointer to a type int array of length k */

    p = a;          /* the same as p = &a[0] */
    ...
}
```

(Note: it is up to you to check whether `n == k`, which is **necessary** !)

Agenda:

- Some most important topics we have learnt
- Debugger `gdb` for detecting the first error
  - recursion `quicksort` as an example

Reading:

- Textbook: Chapters 1–10