Pointer tutorial

Quick Links: Home (https://q.utoronto.ca/courses/419441/assignments/syllabus) Lecture material (https://q.utoronto.ca/courses/419441/pages/lecture-material) Lab assignments (https://q.utoronto.ca/courses/419441/pages/lab-assignments) Piazza Discussion (https://piazza.com/utoronto.ca/winter2026/ece353/home)

Pointers

Introduction

- pointers allow dynamic allocation of memory
- explain need for dynamic allocation with an variable-sized array
- outline
  - pointers
  - pointer examples
  - dynamic memory
  - pointer rules
  - questions

Pointers

- a variable is a symbolic name for a memory location or address
- a pointer is a variable that contains the memory address of some data
  - analogy with house and address of house
  - e.g.

```
int *p;   // creates a variable called p. its type is "pointer to int".
   // read right to left.
int x;   // creates a variable of type int
x = 4;
```

- address operator

```
p = &x;  // & (address operator)
  // gets the location of x and stores it in p
```

- dereference operator

```
y = *p;  // * (dereference operator)
  // y gets the value 4 (*p can be used wherever x can be used)

                Address x 4
```

```
   ____________
  |____________|  4000000
  |____________|  3999999
  |____________|      ...
p |_ 300000 __|   400000
  |____________|      ...
x |______ 4 __|   300000
  |____________|      ...
  |____________|        5
  |____________|        4
  |____________|        3
  |____________|        2
  |____________|        1
  |____________|        0


   ____________                ____________
p |____________|------> x |______ 4 __|
  (pointer to int)          (int)
```

- is there a difference between the following?

```
p = &x    // assigns to the pointer (p points to x)
*p = x    // assigns to the object  (x is assigned to x)
```

- p can be assigned to another location

```
p = &y

   ____________                ____________
p |____________|          x |______ 4 __|
  (pointer to int)          (int)
      |
      |                      ____________
      -------------------> y |______ 4 __|
                               (int)

Is (*p == x)
Is (p == &x)
Is (p == &y)

x = 5
what is *p?
```

- NULL value
  - indicates that a pointer does not point to a valid region
  - e.g.

```
int *p;
p = NULL;   // NULL is actually 0. memory address 0 is never used
```

Pointer Examples

1.

```
int x = 5;
int *p = &x;
int *q;
q = p;
```

```
- explain what a pointer copy means
- show a diagram
```

2.

```
int **p_p;
int *p2;

p_p = &p;


 ___________                ___________                ___________
p_p  |___________|------> p |___________|------>   |_____ 10 __|
     (p. to p. to int)      (pointer to int)       (int)
                                                        ^
p2 = *p_p;    // same as p2 = p;                        |
                                                        |
                                                        |
                          ___________                   |
                     p2 |___________|-----------------|
                        (pointer to int)
```

3. Dynamic Memory

- Pointers allow managing dynamic variables

1) Memory Allocation

- space allocated dynamically when malloc is called

```
{
    int *p; // creates pointer to integer, not an integer

    // allocates an integer, also called dynamic memory
    // malloc returns a pointer to integer
    p = malloc(sizeof(int));

}
```

- p is allocated on the stack
- it goes out of scope at the end of curly brace
- however the dynamically allocated integer is not deallocated

2) Memory Deallocation

- dynamically allocated integer must be explicitly deallocated
- every malloc should eventually be followed with a free

```
{
    int *p;

    p = malloc(sizeof(int));
    ...
    free(p);  // frees the object pointed to by p
              // doesn't free p itself
}
```

- another e.g.

```
int f()
{
    int *p;

    p = malloc(sizeof(int));
    *p = 10;
    return p;
}

int g()
{
    int *q;
    q = f();
    ...
    free(q);
}
```

Pointer Rules

1) Dereference valid pointers

```
int *p;
int y;
y = *p; // bad, y has undefined value since p is an invalid pointer
```

2) Keep track of dynamic memory until it is deleted

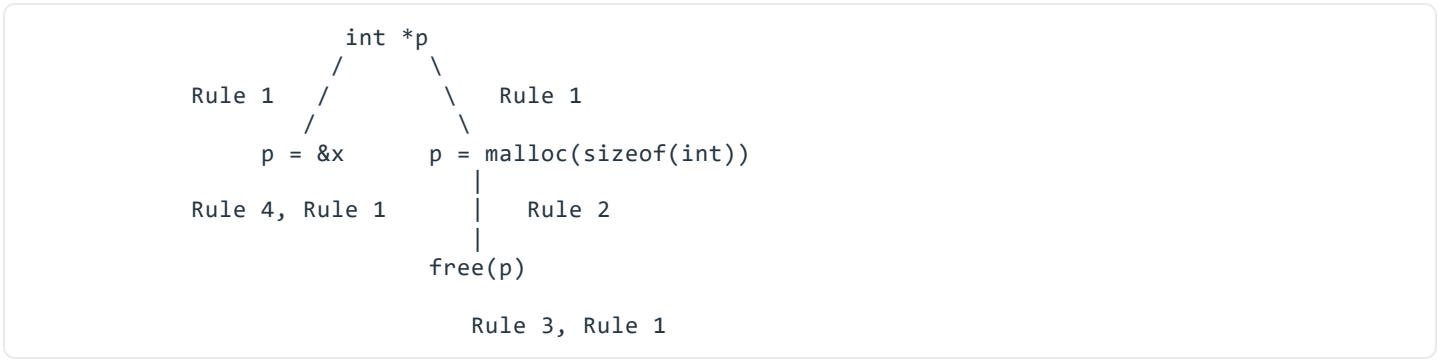
```
p = malloc(sizeof(int));
p = &x;       // dynamic memory is lost - memory leak
// show diagram
```

3) Free dynamic memory once
- after running "free(p)", p still points to dynamic memory
- a second "free(p)" is an error
- use p = NULL for safety since "free(NULL)" is a noop
- also, don't dereference a pointer after calling free on it
- see rule 1!

4) Don't free stack memory
- need to know what type of memory p points to

```
int x;
p = &x;
free(p); // error (since x is not allocated using malloc)
```

- also don't dereference p if x goes out of scope
- p becomes an invalid pointer (see rule 1)



Questions

1) Find out errors

```
int *p;
int **pp;
int *p2;

pp = &p;

pp = p;             // type mismatch
pp = *p;            // type mismatch
p2 = pp;            // type mismatch
*pp = p;            // allocation bug, unless pp = &p has been done
**pp = *p;          // allocation bug, unless pp = &p has been done
&p = pp;            // non lvalue in assignment
p = malloc(sizeof(int)); p2 = p; *p = 10; // what is the value of p2? - 10
p2 = p; p = malloc(sizeof(int)); *p = 10; // what is the value of p2? - undefined
```

2) Is there a problem with this code?

```
for (int *p = ptr; p != NULL; p = p->next) {
    free(p);
}
```

If there is a problem, how would you fix it?

3) Is there a problem with this code?

```
int *m = malloc(sizeof(int));
int *n = m;

free(m);
m = NULL;
free(n);
```

4) Find errors

```
{
    int y = 10;
    int *p, *q;

    if (y > 0) {
        int x;
```

```
            p = &x;
        }
        q = p;
        *p = y;
        y = *p;
    }
```

5) Find memory leaks or other memory errors if they exist

```
int *
simple_func()
{
    int i;
    int *p;

    p = malloc(sizeof(int));
    return p;

}

int *
error_func()
{
    int i;
    int *p;

    i = *p;
    p = &i;

    free(p);
    return &i;
}

int *
error2_func()
{
    int *p = malloc(sizeof(int));

    p = malloc(sizeof(int));
    free(p);
    free(p);
}
```