ECE 264 Advanced C Programming

Contents

1 Deep Copy

The previous lecture ended when we encountered a problem:

```
   p1 = p2;
```

made `p1` and `p2` share the same **address** for their names. As a result, changing `p1`'s name also changes `p2`'s name

There are several solutions and we will explain only one called *deep copy*. Another solution uses *reference counts* and *copy-on-write*; this is beyond the scope of ECE 264. Instead of using "=", we have to provide a function to copy the objects. In this example, we also use pointers for objects. From now on, we use pointers for objects in most cases because of their flexibility. When we use structures in C, we often need to dynamically allocate and release memory. Pointers are much easier to handle memory allocation and release.

```
/* person.h */
#ifndef PERSON_H
#define PERSON_H
typedef struct
{
  int p_age;
  char * p_name;
} Person;

Person * Person_construct(int a, char * n); /* return pointer */
Person * Person_copy(Person * p);
void Person_assign(Person * * p1, Person * p2);
/* notice * * for the first object */
void Person_destruct(Person * p);
void Person_print(Person * p);
int Person_getAge(Person * p);
char * Person_getName(Person * p);
#endif
```

We change the constructor to return a pointer of a `Person` object. The function first creates an object and then creates the array to hold the name. There are two new functions: `Person_assign` and `Person_copy`. The former **replaces** the assignment ("=") that caused problems earlier when only shallow copy was used. `Person_assign` uses *deep copy*. What does this mean? Instead of copying the address of the array for the name, the function will **allocate additional memory** so that the source and the destination do **not** share the memory address. We expect the following statement does nothing.

```
    x = x;
```

Thus, we have to check whether the source and the destination are the same. If they are the same, the function does nothing and returns immediately. Why is this necessary? Because the next step **releases the memory of the destination**. If we do not check, the destination and the source are both released.

```
/* person.c */
#include "person.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
Person * Person_construct(int a, char * n)
{
  Person * p = malloc(sizeof(Person));
  p -> p_age = a;
  p -> p_name = malloc((strlen(n) + 1) * sizeof(char));
  strcpy(p -> p_name, n);
  return p;
}

Person * Person_copy(Person * p)
{
  return Person_construct(p -> p_age, p -> p_name);
}

void Person_assign(Person * * p1, Person * p2)
/* replace = because it does shallow copy */
/* notice * * for the first object */
{
  if ((*p1) == p2) { return; } /* must check first */
  Person_destruct (* p1);
  * p1 = Person_copy(p2);
}

void Person_destruct(Person * p)
{
  free (p -> p_name);
  free (p);
}

void Person_print(Person * p)
{
  printf("age= %d, name= %s\n", p -> p_age, p -> p_name);
}

int Person_getAge(Person * p)
{
  return p -> p_age;
}

char * Person_getName(Person * p)
{
  return p -> p_name;
}
```

Why do we need to destroy `p1`? Because it holds memory for the name. Without calling the destructor, the memory is leaked. Then, we create a new `Person` object by calling `Person_copy`. This function is called to create a **new object from an existing object** and is implemented by calling `Person_construct`. The output of `main` is what we expect: changing `p1`'s name does not change `p2`'s nor `p3`'s names. Each object has its own memory to hold the name.

Do **not** use the word `new` in your C program because it is reserved in C++. Due to the similarity between C and C++, it is common to "migrate" C code to C++. If a C program contains `new`, the C++ program will not compile.

In main, we have to pass the address of p1 when calling Person_assign because we will change where p1 points to. When p3 is created, Person_copy is called because this is the **first time** p3 occupies memory. If we want to change a Person object latter, we have to call Person_assign. If we call Person_copy, the memory is leaked.

```
/* main.c */
#include <stdio.h>
#include <string.h>
#include "person.h"
int main(int argc, char * argv[])
{
  Person * p1 = Person_construct(19, "Tom Johnson");
  Person * p2 = Person_construct(21, "Mary Smith");
  Person * p3;
  p3 = Person_copy(p1);
  Person_print(p1);
  Person_print(p2);
  Person_print(p3);
  Person_assign(& p1, p2); /* notice & */
  Person_print(p1);
  Person_print(p2);
  Person_assign(& p1, p1); /* source = destination */
  Person_print(p1);
  strcpy(p1 -> p_name, "Edward");
  Person_print(p1);
  Person_print(p2);
  Person_print(p3);
  Person_destruct(p1);
  Person_destruct(p2);
  Person_destruct(p3);
  return 0;
}

/*
  output:
  age= 19, name= Tom Johnson
  age= 21, name= Mary Smith
  age= 19, name= Tom Johnson
  age= 21, name= Mary Smith
  age= 21, name= Mary Smith
  age= 21, name= Mary Smith
  age= 21, name= Edward
  age= 21, name= Mary Smith
  age= 19, name= Tom Johnson
*/
```

General Rules: If the constructor allocates memory, we also need to provide the copy and the assignment functions. The copy function is called to create a new object using an existing object. It has only one input argument, as a pointer to an existing object. The assignment function has two arguments, the first as the destination and the second as the source. At the beginning of the function, it checks whether the destination and the source are the same. If they are the same, do nothing and return. If they are different, destroy the destination and create an object by calling the copy function. This rule can help you prevent memory errors in larger programs, particularly when the programs contain structures of structures.

2 Complexity of Selection Sort

How much time does it take for selection sort to order a array of n elements?

```
#include <stdio.h>
void swap(int * a, int * b)
{
  int temp = *a;
  (*a) = (*b);
  (*b) = temp;
}
void printArray(int * x, int n)
{
  int i;
  for (i = 0; i < n; i ++)
    { printf("%8d", x[i]);  }
  printf("\n");
}
int main(int argc, char * argv[])
{
  int x[] = {6, 7, 3, 2, 0, 9, -4, 1};
  int n = sizeof(x) / sizeof(int);
  printArray(x, n);
  int i1, i2, mInd;
  for (i1 = 0; i1 < n - 1; i1 ++)
    {
      mInd = i1;
      for (i2 = i1 + 1; i2 < n; i2 ++)
        {
          if (x[mInd] > x[i2])
            { mInd = i2; }
        }
      if (mInd != i1)
        {
          printf("\ni1 = %d, mInd = %d, x[i1] = %d, x[mInd] = %d\n",
                 i1, mInd, x[i1], x[mInd]);
          swap(&x[i1], &x[mInd]);
          printArray(x, n);
        }
    }
  printArray(x, n);
  return 0;
}
```

The first iteration goes through `0`, `1`, `2`, ..., until `n - 2`. The second iteration goes through `i + 1`, `i + 2`, ..., until `n - 1`, namely `n - 1 - i1` times. Totally, the `if` condition is tested

$$\sum_{i=0}^{n-2}(n-1-i) = (n-1)\cdot(n-1) + \sum_{i=0}^{n-2}(-i) = (n-1)^2 - \sum_{i=0}^{n-2} i. \tag{1}$$

$\boxed{\text{How to compute } \sum_{i=1}^{n} i?}$

Let $f(n)$ be $\sum_{i=1}^{n} i$.

$$\begin{array}{lccccccccc}
f(n) = & 1 & + & 2 & + & 3 & + & ... & + & n \\
f(n) = & n & + & (n-1) & + & (n-2) & + & ... & + & 1 \\
\Rightarrow \\
2f(n) = & (n+1) & + & (n+1) & + & (n+1) & + & ... & + & (n+1) \\
2f(n) = & (n+1)\times n \\
\Rightarrow \\
f(n) = & \frac{n(n+1)}{2}
\end{array} \tag{2}$$

$$(n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} = \frac{n^2-n}{2} \approx \frac{1}{2}n^2 \approx n^2. \tag{3}$$

When we analyze a computer program, we are often interested in the *growth rate* of the execution time related to the size of the problem. If we have an array of n elements, how long does it take? Why do we care about the growth rate? Because computers are designed to handle large amounts of data. As references, *Yahoo.com* has 500 million users; *facebook.com* has 150 million users; Southwest Airlines fly more than 100 million passengers per year; *eBay.com* has 84 million active users. We want to design programs whose execution times grow slowly relative to the size of the problems. This is called the *scalability* of programs. Here is a basic rule you need to know

$$\log n < n < n^2 < n^3 < 2^n \tag{4}$$

when n **is sufficiently large (**$n > 10$**)**. The following program can show their values for n between 1 and 20.

```
/* growrate.c */
/* to compile and link, use
   gcc growrate.c -lm because we need to link the math library
 */
#include <math.h>
#include <stdio.h>
int main(int argc, char * argv[])
{
  int valN;
  printf(" n      exp      n^2      n^3        2^n\n");
  for (valN = 1; valN <= 20; valN ++)
    {
      printf("%2d %8d %8d %8d %10d\n",
             valN, (int) log(valN), valN * valN,
             valN * valN * valN, (int) pow(2, valN));
    }
  return 0;
}
/*
 n      exp      n^2      n^3        2^n
 1        0        1        1          2
 2        0        4        8          4
 3        1        9       27          8
 4        1       16       64         16
 5        1       25      125         32
 6        1       36      216         64
 7        1       49      343        128
 8        2       64      512        256
 9         2        81       729         512
10         2       100      1000        1024
11         2       121      1331        2048
12         2       144      1728        4096
13         2       169      2197        8192
14         2       196      2744       16384
15         2       225      3375       32768
16         2       256      4096       65536
17         2       289      4913      131072
18         2       324      5832      262144
19         2       361      6859      524288
20         2       400      8000     1048576
*/
```

This program also shows a few important concepts in C. First, we are using the math library. Since this is not a standard C library (such as `printf`), we have to add `-lm` to link the program with the math library. Second, we can format the output by adding a number after `%`. This `printf` gives 2 columns for the first number, 8 columns for the second number, and 10 columns for the last number. Third, we use *type cast* to convert a floating-point number to an integer. The outputs of `log` and `pow` are floating-point numbers (double precision). We are interested in only the integer values. The first function uses *natural logarithm* as the base. The second function uses 2 as the base for the power function.

We are more interested in the **approximation** of the growth rate. If $f(n) \gg g(n)$ when n is large, we ignore $g(n)$ in $f(n) + g(n)$ and use $f(n)$ only. We also ignore the constant coefficient $\frac{1}{2}$. Hence,

the complexity of selection sort is n^2.

Exercise: Suppose X, Y, and Z are three square matrices:

$$X = \begin{bmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,n} \\ x_{2,1} & x_{2,2} & \dots & x_{2,n} \\ \dots & \dots & \dots & \dots \\ x_{n,1} & x_{n,2} & \dots & x_{n,n} \end{bmatrix}_{n\times n.} \quad Y = \begin{bmatrix} y_{1,1} & y_{1,2} & \dots & y_{1,n} \\ y_{2,1} & y_{2,2} & \dots & y_{2,n} \\ \dots & \dots & \dots & \dots \\ y_{n,1} & y_{n,2} & \dots & y_{n,n} \end{bmatrix}_{n\times n.}$$
$$Z = \begin{bmatrix} z_{1,1} & z_{1,2} & \dots & z_{1,n} \\ z_{2,1} & z_{2,2} & \dots & z_{2,n} \\ \dots & \dots & \dots & \dots \\ z_{n,1} & z_{n,2} & \dots & z_{n,n} \end{bmatrix}_{n\times n.} \tag{5}$$

Matrix multiplication Z = XY is defined as

$$z_{i,j} = \sum_{k=1}^{n} x_{i,k} \times y_{k,j}, \qquad 1 \le i, j \le n. \tag{6}$$

Write C code to implement matrix multiplication. What is the complexity of the code?

◁

3 Pointer Arithmetics

C provides “pointer arithmetics”. Suppose `ptr` is a pointer:

```
int array[100];
int * ptr = array;
```

The increment operation `ptr ++;` makes `ptr` point to next element.

```
#include <stdio.h>
void printInt(int * intArray, int numElem)
{
  int index;
  int * intPtr;
  intPtr = intArray;
  for (index = 0; index < numElem; index ++)
    {
      printf("%d %d\n", * intPtr, intArray[index]);
      /* notice * before intPtr */
      /* *intPtr same as intArray[index] */
      intPtr ++;
    }
  printf("\n");
}
void printDouble(double * doubleArray, int numElem)
{
  int index;
  double * doublePtr;
  doublePtr = doubleArray;
  for (index = 0; index < numElem; index ++)
```

```
    {
      printf("%.2f %.2f\n", * doublePtr, doubleArray[index]);
      /* .2f means two digits after the decimal point */
      doublePtr ++;
    }
  printf("\n");
}
int main(int argc, char * argv[])
{
  int    intArray[] = {9, 7, -11, 5, 6, 8};
  double doubleArray[] = {9.8, 3.4, -11.893, 6.1, 7.5, 0.24};
  printInt(intArray, sizeof(intArray)/ sizeof(int));
  printDouble(doubleArray, sizeof(doubleArray)/ sizeof(double));
  return 0;
}
/*
  output:
  9 9
  7 7
  -11 -11
  5 5
  6 6
  8 8

  9.80 9.80
  3.40 3.40
  -11.89 -11.89
  6.10 6.10
  7.50 7.50
  0.24 0.24
*/
```

C knows the distance between elements (from `sizeof`). We can use this property to write the `printInt` and the `printDouble` function. Hence, `intPtr ++;` and `doublePtr ++;` mean the next elements in the two arrays.