ECE 264 Advanced C Programming

Contents

1 Pointer (Review)

Explain the meanings of the following statements:

```
int a[100];
int * b;
b = a;
int * c;
c = malloc(100 * sizeof(int));
b = c;

void f1(int * v)
{
   * v = 3;
}

int d = -9;
f1(& d);

void f2(int * * p)
{
   * p = malloc(200 * sizeof(int));
}
```

```
int * g;
f2(& g);

Person * h;
h = malloc(60 * sizeof(Person));
```

Rules:

- `a[100]` allocates 100 elements and its address **cannot** be changed.
- `int *b` allocates a pointer and it is **not** pointing to any meaningful location (yet).
- `b = a` is valid; `b` points to the starting address of the array `a`.
- `*b` is the value of the first element of the array.
- `b++;` moves `b` to point to the next element.
- `a = b` is invalid;
- `c = malloc ...` makes `c` point to a newly allocated memory. The memory should be released later by calling `free`.
- `b` can point to other locations, such as `b = c`.
- If `b` and `c` point to the same location, the memory can be released only **once**.
- C function calls use *pass by value*. The value is **copied** from the caller to the functions' arguments.
- If you want to change the value of an argument and make the change visible at the caller, you have to pass the **address** from the caller and uses a **pointer** at the function.

2 Structure

So far we have learned two types of data: *scalar* and *array*. A scalar means a single piece of data, such as an integer (`int`), a double-precision floating-point number (`double`), or a character (`char`). An array means a collection of data of the **same** type, such as an array of integers (`int array[10]`) or characters (`char str[20];`). In many cases, however, we want to **mix** data types and create a new data type. For example, a `Person` has a name (char []) and an age (int). A group of people will be an array of Person. If we create two arrays— one of char * and the other of int— there is no obvious way to connect a name with this Person's age. You can read the fifth's name from the first array and the second age from the second array. This causes confusion and leads to mistakes. What we need is a way to **create our own data type** for Person.

```
const int maxNameLength = 60;
struct Person
{
  int p_age;
  char p_name[maxNameLength];
};
```

We have created a *structure* using C's struct. Each person has an age and a name. We assume that a Person's name cannot exceed 59 characters (the last one is for the ending character '\0'). These two are called the *attributes* of Person. How do we use the structure? Here are two examples

```
struct Person p1;
struct Person p2;
```

We have created two Person variables: p1 and p2. To change an attribute, we can write

```
p1.p_age = 20;
strcpy(p2.p_name, "Amy Johnson");
```

Next is another example for a Date structure.

```
struct Date
{
  int d_date;
  int d_month;
  int d_year;
  char d_dayName[4]; /* Mon, Tue, Wed, ... */
  char d_monthName[4]; /* Jan, Feb, Mar, ... */
};
```

To create a Date variable, we can write

```
struct Date today;
```

You may notice the coding style I am using. The **name of a structure is a noun and starts an uppercase letter**. We are borrowing the convention adopted in object-oriented programming (OOP), where classes' names are usually nouns and starts with capital letters. I also invent my own coding convention by adding a **prefix** with an underscore before each attribute. The purpose is to make it easier to recognize what each identifier means. When we see `p_age`, we know that it is probably an attribute of a structure whose name starts with `P`. When we see `d_year`, we know that it is probably an attribute of a structure whose name starts with `D`. These conventions make the program easier to read. You can develop your own coding styles, as long as they are reasonable and consistent. You can also find the *GNU Coding Standards* .

Instead of typing `struct` every time, we can add `typedef` before `struct`:

```
const int maxNameLength = 60;
typedef struct /* typedef before struct */
{
  int p_age;
  char p_name[maxNameLength];
} Person;
Person p1;
Person p2;
```

I am going to borrow some terms and concepts from object-oriented programming. OOP can help us organize our programs in a more consistent way.

3 Objects and Operations

In these two examples, `Person` and `Date` are *types*; `p1`, `p2`, and `today` are variables. We also call them *objects* even though they are **not** strictly objects for C++ or Java. They have three properties as an "object": (1) identity (`p1` and `p2`), (2) states (names and ages), and (3) operations. (introduced later). Next example shows operations.

```
/* written in the style of object-oriented programming */
/* It is different from the example in 9.2 ABoC */
#include <stdio.h>
typedef struct
{
  double c_x;
  double c_y;
} Vector;

Vector Vector_construct(double x, double y)
```

```
{
  Vector c;
  c.c_x = x;
  c.c_y = y;
  return c;
}

Vector Vector_add(Vector c1, Vector c2)
{
  Vector c3;
  c3.c_x = c1.c_x + c2.c_x;
  c3.c_y = c1.c_y + c2.c_y;
  return c3;
}

Vector Vector_subtract(Vector c1, Vector c2)
{
  Vector c3;
  c3.c_x = c1.c_x - c2.c_x;
  c3.c_y = c1.c_y - c2.c_y;
  return c3;
}

void Vector_print(Vector c)
{
  printf("x = %f, y = %f\n", c.c_x, c.c_y);
}

int main(int argc, char * argv[])
{
  Vector c1 = Vector_construct(1.9, 2.4);
  Vector c2 = Vector_construct(3.4, 5.7);
  Vector c3 = Vector_add(c1, c2);
  Vector c4 = Vector_subtract(c1, c2);
  Vector_print(c1);
  Vector_print(c2);
  Vector_print(c3);
  Vector_print(c4);
  return 0;
}

/*
```

```
  output
  x = 1.900000, y = 2.400000
  x = 3.400000, y = 5.700000
  x = 5.300000, y = 8.100000
  x = -1.500000, y = -3.300000
*/
```

In this example, we create a structure for 2-dimensional `Vector`s and four operations. You may find that **each operation starts with the prefix** `Vector` so that we know the operations are for the `Vector` structure. This is another example making the program easier to understand. You may also notice that **each operation is a verb**.

The first function `Vector_construct` creates an object of `Vector` by taking two arguments, for the x and the y coordinates. The next two functions add and subtract `Vector` objects and return the results. Finally, the last function prints the coordinates.

The next example creates an array of structures. Each element is treated as a `Vector` object.

```
int main(int argc, char * argv[])
{
  Vector ca[4];
  ca[0] = Vector_construct(1.9, 2.4);
  ca[1] = Vector_construct(3.4, 5.7);
  ca[2] = Vector_add(ca[0], ca[1]);
  ca[3] = Vector_subtract(ca[0], ca[1]);
  Vector_print(ca[0]);
  Vector_print(ca[1]);
  Vector_print(ca[2]);
  Vector_print(ca[3]);
  return 0;
}
```

4 Structure with Memory Allocation

In `Vector`, the attributes are *scalars*: double-precision floating point numbers. What happens if an attribute needs to allocate memory? The next example shows how to do that. As our program becomes more and more complicated, we should organize it into multiple files. This program has three files. In addition, we create `Makefile` so that we do not have to compile and link the files by typing all commands every time. We just need to type `make`.

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

Person Person_construct(int a, char * n);
void Person_destruct(Person p);
int Person_getAge(Person p);
char * Person_getName(Person p);
void Person_print(Person p);
#endif
```

A *header file* (extension `.h`) has the following structure

```
#ifndef FILENAME_H
#define FILENAME_H
...
#endif
```

Right now, our program is not complex enough to explain the necessity of this structure. For the time being, just remember the format. If you use `Eclipse`, it automatically adds `#ifndef ...  #endif` to header files.

A header file provides *type* and *function declarations*. In this example, we declare a structure called `Person` and several functions related to the structure.

The next file is a *source file* and *implements* the functions declared in the header file. A typical C program has several header files (`.h`) and corresponding source files (`.c`).

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

void Person_destruct(Person p)
{
  free (p.p_name);
}

int Person_getAge(Person p)
{
  return p.p_age;
}

char * Person_getName(Person p)
{
  return p.p_name;
}

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

The *constructor* allocates memory for the name. Please remember that `strlen` does not include the ending character '\0' and we have to allocate one additional character. **When we allocate memory in the constructor, we should also create a destructor (called `destruct`) to release the memory.** Otherwise, the program will leak memory. We also have two functions to retrieve the age and the name of a `Person` object.

In addition to pairs of `.h` and `.c` files, a file contains the `main` function. The `main` function creates two `Person` objects, `p1` and `p2`. We can compare their ages to determine who is younger.

```
#include <stdio.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_print(p1);
  Person_print(p2);
  if (Person_getAge(p1) > Person_getAge(p2))
    {
      printf("%s is older than %s\n", Person_getName(p1),
             Person_getName(p2));
    }
  else
    {
      if (Person_getAge(p1) < Person_getAge(p2))
        {
          printf("%s is younger than %s\n", Person_getName(p1),
                 Person_getName(p2));
        }
      else
        {
          printf("%s and %s have the same age\n", Person_getName(p1),
                 Person_getName(p2));
        }
    }
  Person_destruct(p1);
  Person_destruct(p2);
  return 0;
}

/*
  output:
  age= 19, name= Tom Johnson
  age= 21, name= Mary Smith
  Tom Johnson is younger than Mary Smith
*/
```

Since we have several files now, compiling and linking them requires running `gcc` several times. We use `Makefile` for this purpose.

```
person: person.h person.c main.c
        gcc -Wall -c person.c
        gcc -Wall -c main.c
        gcc -Wall person.o main.o -o person
clean:
        rm -f *.o person
```

This program actually has a serious problem, as shown in the next example.

```
#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_print(p1);
  Person_print(p2);
  p1 = p2; /* assign p2 to p1, they should be the same */
  Person_print(p1);
  Person_print(p2);
  strcpy(p1.p_name, "Edward"); /* modify p1's name */
  Person_print(p2); /* p2's name is also changed to Edward */
  Person_destruct(p1);
  Person_destruct(p2);
  return 0;
}

/*
  output:
  age= 19, name= Tom Johnson
  age= 21, name= Mary Smith
  age= 21, name= Mary Smith
  age= 21, name= Mary Smith
  age= 21, name= Edward
  *** glibc detected *** double free or corruption (fasttop): 0x0804a018 **
  ======= Backtrace: =========
  /lib/tls/i686/cmov/libc.so.6[0x4009ca85]
  /lib/tls/i686/cmov/libc.so.6(cfree+0x90)[0x400a04f0]
  ./person4[0x80484eb]
  ./person4[0x8048612]

*/
```

The first four lines have nothing special. We create two `Person` objects and print them. The outputs are expected. Next, we assign `p2` to `p1`. We print these two objects again. Both of them show “Mary Smith.” The next line changes `p1`'s name to Edward. Since “Edward” is shorter than “Mary Smith”, there is enough space for the new name. The next line prints `p2` and it shows “Edward”.

We changed `p1`'s name and also `p2`'s name. Let's consider the following code for comparison.

```
   int x = 1;
```

```
int y = 5;
x = y; /* x is 5 now */
x = 18; /* x is 18, what is y? */
```

What is the value of `y`? It should be 5, right? We would be surprised if `y` were 18.

Remember that the name is stored as an array of characters and an array is a pointer. When we use

```
p1 = p2;
```

C does *shallow copy* by assigning the **address**, not the value, of `p2`'s name to `p1`'s name. As a result, `p1`'s name and `p2`'s name **share the same address** (`p2`'s address). When we modify `p1`'s name, we also modify `p2`'s name. The memory allocated to `p1`'s name is no longer accessible; this is an example of memory leak. This also explains why `Person_destruct` causes problems. When we try to release the memory allocated to `p2`'s name, it has already been released earlier, by `Person_destruct(p1)`.

We will solve this problem in the next lecture using a concept called *deep copy*.