CS1010E: Programming Methodology

Tutorial 11: Struct

10 Apr 2017 - 15 Apr 2017

1. Discussion Questions

(a) [Basic of Structure] There are several ways of declaring **struct**. Can you tell what are the difference between them (*especially in terms of declaring and initializing them*)?

i.
```
struct Books {
  /* fields of structure */
};
```
i. ____________________

ii.
```
typedef struct {
  /* fields of structure */
} Books;
```
ii. ____________________

iii.
```
struct Books {
  /* fields of structure */
} book;
```
iii. ____________________

Convention: *I will use the second type for "template" in the tutorial and lab, while exam uses the third type. You may change the definition as you see fit. The first type of declaration is generally used for recursive data structure since the identifier has been declared and can be used within itself.*

(b) [Simple Structure Reasoning] What is/are the output of code fragments below?

i.
```
typedef struct { int x; } Obj;
int main() {
  Obj o = {3}; printf("%d", o.x);
  return 0;
}
```
i. ____________________

ii.
```
typedef struct { int x, y; } Obj;

int main() {
  Obj o = {3, 4}; printf("%d", o.x - o.y);
  return 0;
}
```
ii. ____________________

iii.
```
typedef struct { int x;          } Obj1;
typedef struct { Obj1 o; int  x; } Obj2;
int main() {
  Obj2 o = {{2}, 3}; printf("%d %d", o.x, o.o.x);
  return 0;
}
```
iii. ____________________

2. Program Analysis

(a) [Complex Structure Reasoning] Given the **struct** for a Person below:

```
typedef struct {
  char* first;
  char  last[SIZE+1];
  int   age;
} Person;
```

What is/are the output of code fragments below?

i.
```
void update(Person p, char first[], char last[], int age) {
  strcpy(p.first, first);
  strcpy(p.last , last );
  p.age = age;
}
int main() {
  char first[] = "John";
  Person p = {first, "Doe", 27};
  update(p, "GI", "Joe", 35);
  printf("%s %s %d", p.first, p.last, p.age);
}
```

i. ________________________________

ii.
```
void update(char name[], char newName[]) {
  strcpy(name, newName);
}
int main() {
  char first[] = "John";
  Person p = {first, "Doe", 27};
  update(p.first, "GI" );
  update(p.last , "Joe");
  printf("%s %s %d", p.first, p.last, p.age);
}
```

ii. ________________________________

iii.
```
void update(Person *p, char first[], char last[], int age) {
  strcpy(p->first, first); strcpy(p->last , last ); p->age = age;
}
int main() {
  char first[] = "John";
  Person p = {first, "Doe", 27};
  update(&p, "GI", "Joe", 35);
  printf("%s %s %d", p.first, p.last, p.age);
}
```

iii. ________________________________

(b) [Complex Structure Reasoning] What is/are the output of code fragments below?

i.
```
struct Obj { int v; struct Obj* n; }; // recursively defined struct
int main() {
  struct Obj a[7] = {0}, *c; int i;
  for(i=0; i<7; i++) {
    a[(i+3)%7].v = i;
    a[(i+3)%7].n = &a[i];
  } a[6].n = NULL;
  c = a;
  while(c != NULL) {
    printf("%d ", c->v);
    c = c->n;
  }
  return 0;
}
```
i. ______________________________

ii.
```
struct Obj { int v; struct Obj *l, *r; }; // recursively defined struct
void foo(struct Obj *c) {
  if(c == NULL) return;
  printf("%d ", c->v);
  foo(c->l); foo(c->r);
}
int main() {
  struct Obj a[7] = {0}; int i;
  for(i=0; i<7; i++)
    a[i].v = i+1;
  for(i=0; i<3; i++) {
    a[i].l = &a[2*i+1];
    a[i].r = &a[2*i+2];
  }
  foo(a);
  return 0;
}
```
ii. ______________________________

3. Designing a Solution

(a) [Structure] An ice cream store has 4 different ice cream flavors. The ice cream is sold with 2 different sizes: single (*1 scoop*) and double (*2 scoops*). Each scoop of the ice cream cost $0.75.

Additionally, the customer may get the ice cream served in cones (*additional $0.50*) or waffle (*additional $1.00*). To be consistent, we can also say that there are 3 different types to the ice cream, namely: plain, cone, and waffle. Lastly, the customer may add additional toppings at $0.25 per topping.

i. Create a `struct` for an ice cream. Use the template below:

```
typedef struct {
  /* Ice Cream fields here... */
} IceCream;
```

ii. Write a function to make a single scoop ice cream. Use the template below:

```
IceCream make() {
  /* Create a plain single scoop ice cream with no topping */
}
```

iii. Write a function to *upsize* the order to double scoop. Use the template below:

```
void addScoop(IceCream *ice) {
  /* ice: ice cream to be made double scoop */
}
```

iv. Write three functions to change the type based on the description below:

```
void makePlain(IceCream *ice) {
  /* Make ice to be plain ice cream */
}
```

```
void makeCone(IceCream *ice) {
  /* Make ice to be cone ice cream */







}
void makeWaffle(IceCream *ice) {
  /* Make ice to be waffle ice cream */







}
```

v. Write a function to add toppings to the order. Use the template below:

```
void addTopping(IceCream *ice) {
  /* ice: ice cream to be added the topping */







}
```

vi. Write a function to compute the price of the ice cream. Use the template below:

```
double computePrice(IceCream *ice) {
  /* ice: ice cream which price is to be computed */














}
```

4. Challenge

(a) [Abstraction; Approximation; Mathematics] The Challenge question in Tutorial 04 shows a method of approximating the value of e^x for any value of x with very high precision. In this problem, we will solve a generic method of approximating the *solution* to the equation $f(x) = 0$ for any polynomial $f(x)$. This method is called Newton-Rhapson method. (NRM).

NRM starts with an initial guess of x_0. The first approximation to $f(x)$ can be obtained using the formula shown in Equation 1 where $f^{'}(x)$ is the *first derivative* of $f(x)$. This first derivative is done via *differentiation*. We will discuss *differentiation* later on.

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \tag{1}$$

Now that we have a better approximation x_1, we can repeat the process and obtain a better and better approximation as shown in Equation 2.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{2}$$

Your main task is to find a *good enough* approximation of any polynomial $f(x)$ via NRM.

i. A polynomial is a Mathematical expression of the form $f(x) = c_n x^n + c_{n-1} x^{n-1} + ... + c_2 x^2 + c_1 x + c_0$. That is, every term (i.e. x^i *for all values of* i) has an attached coefficient (i.e. c_i). The *degree* of the polynomial is the largest value of i such that the coefficient c_i is non-zero. We then say that the degree of this polynomial is i. For instance, an example of a polynomial of degree 2 is $f(x) = 6x^2 + 3$ or $f(x) = x^2 + 2x + 1$. Create a `struct` for a polynomial assuming that all the coefficients are `integer` and the maximum degree of the polynomial is 100. Use the template below:

```
typedef struct {
  /* Polynomial fields here... */




} Polynomial;
```

ii. Differentiation is a calculus invented by Sir Isaac Newton. The process of differentiation for polynomial can be easily described by the formula below:

$$\text{let } f(x) = c_n x^n + c_{n-1} x^{n-1} + ... + c_2 x^2 + c_1 x + c_0,$$
$$f^{'}(x) = (n \times c_n) x^{n-1} + ((n-1) \times c_{n-1}) x^{n-2} + ... + (2 \times c_2) x^1 + c_1 \tag{3}$$

Note how the result of a differentiation of a polynomial yields another polynomial. For instance, the differentiation of $f(x) = x^2 + 2x + 1$ is $f^{'}(x) = 2x + 2$. Write the code to differentiate a polynomial. Use the template below:

```
Polynomial differentiate(Polynomial f) {
  Polynomial g; // g(x) = f'(x)
   /* f: polynomial function, g: differentiation of f */









  return g;
}
```

iii. An *evaluation* of a polynomial for a given value of x is the simplification of the expression obtained by *substituting* all occurrences of x in $f(x)$. For instance, the evaluation of $f(x) = x^2 + 2x + 1$ for $x = 2$ is $f(2) = 2^2 + (2 \times 2) + 1 = 4 + 4 + 1 = 9$. Write a function to evaluate an arbitrary polynomial. Use the template below (hint: *use* `pow` *using* `#include <math.h>` *library*):

```
double evaluate(Polynomial f, double x) {
  double res = 0;
   /* f: polynomial function, x: value for f(x) */









  return res;
}
```

iv. Write a function to perform NRM on a given polynomial. Use the template below (hint: *when should you stop the approximation?*):

```
double NewtonRhapson(Polynomial f, double x0) {
   /* f: polynomial function, x0: first approximation */















}
```