



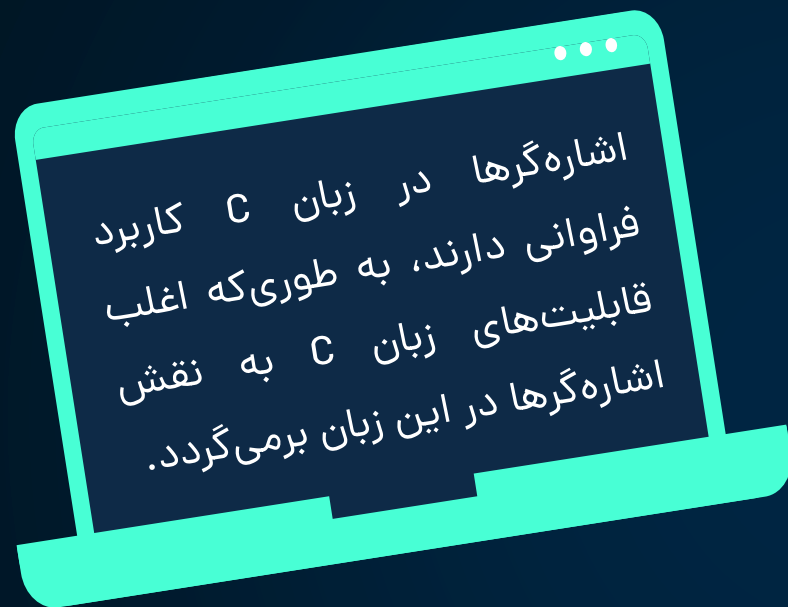
پوینتر

جلسه دهم

بسم الله الرحمن الرحيم



کارگاه مبانی برنامه نویسی - دانشکده مهندسی کامپیوتر دانشگاه هوایی شهید ستاری



تاکنون ما با داده‌های ساده‌ای مانند کاراکتر، عدد صحیح و عدد اعشاری برخورد داشته‌ایم. نوع دیگری از این متغیرها، اشاره‌گرها هستند. زمانی که ما یک متغیر را برای ذخیره‌ی اطلاعاتی تعریف می‌کنیم، در واقع این نام متغیر، یک نام مستعار برای آدرس آن اطلاعات در حافظه است. یعنی ما با کمک این متغیر می‌توانیم بدون درگیر شدن با آدرس‌های حافظه، داده‌ها و اطلاعات مورد نیاز خود را ذخیره یا استفاده کنیم.


اما گاهی در برنامه‌نویسی لازم است به جای نام متغیر، با خود آدرس آن متغیر کار کنیم. اشاره‌گرها می‌توانند آدرس یک متغیر را در حافظه در خود نگه دارند و در چنین مواقعی به کمک ما می‌آیند.

فهرست



سوال اول: پوینتر تو پوینتر

برای یادگیری عملکرد پوینترها لازم است که روی تحلیل برنامه‌های پوینتری مسلط باشیم و بتوانیم آن‌ها را دیباگ کنیم. بیایید برای شروع با قطعه کدهای کوتاه و نکته‌دار شروع کنیم تا دانش پوینتری‌مان را محک بزنیم و نکات را یکی یکی بررسی کنیم.

قطعه کد زیر را اجرا کنید. در خروجی چه چیزی مشاهده می‌کنید؟ برنامه را خط به خط تحلیل کنید و توضیح دهید که چطور این خروجی حاصل شده است؟ 

```
int main() {  
    int a[5] = {1, 2, 3, 4, 5};  
    printf("%lu\n", sizeof(a));  
    int *ptr = (int *)(&a + 1);  
    printf("%d %d\n", *(a + 1), *(ptr - 1));  
    return 0;  
}
```

سوال دوم: پوینتر به پوینتر

```
// Assume that the size of int is 4.
```

```
void f(char **);
```

```
int main() {
```

```
    char *argv[] = {"ab", "cd", "ef", "gh", "ij", "kl"};
```

```
    f(argv);
```

```
    return 0;
```

```
}
```

```
void f(char **p) {
```

```
    char *t;
```

```
    t = (p += sizeof(int))[-1];
```

```
    printf("%s", t);
```

```
}
```


حال سعی کنید بدون چاپ کردن خروجی نهایی



این قطعه کد، آن را تحلیل کنید و تشخیص دهید

که خروجی چه خواهد بود.

سوال سوم: دیباگ پونتری

حال سعی کنید کمی عمیقتر و دقیقتر کدها را بررسی کنید. خطای برنامه زیر را پیدا کرده و آن را اصلاح کنید. به نظر شما برنامه‌ی فعلی (دارای خطا) درست کار می‌کند؟ فکر می‌کنید علت این اتفاق چیست؟ 

```
#include<stdio.h>
```


```
int f(int* p) {  
    printf("a = %d\n", *p); // a = 10?  
}
```

```
int main() {  
    int a = 10;  
    f((int *) a);  
}
```


سوال چهارم: چالش



```
int *a[3];  
int (*b)[3];
```

تفاوت متغیرهای a و b در چیست؟ 

```
#include<stdio.h>
```

خروجی قطعه کد زیر چیست؟ 

```
int main() {  
    int a[][3] = {1, 2, 3, 4, 5, 6};  
    int (*ptr)[3] = a;  
    printf("%d %d ", (*ptr)[1], (*ptr)[2]);  
    ++ptr;  
    printf("%d %d\n", (*ptr)[1], (*ptr)[2]);  
    return 0;  
}
```

سوال پنجم: بی نام



فکر می کنید چرا کد زیر به درستی ۵ فاکتوریل را حساب نمی کند؟



```
void factorial (int *res, int num) {  
    *res = 1;  
    for (int i = 1; i <= num; i++) {  
        *res *= i;  
    }  
}
```

```
int main() {  
    int *res;  
    int num = 5;  
    factorial(*res, num);  
    printf("%d! = %d", num, *res);  
    return 0;  
}
```


سوال آخر: رمزنگاری

سلام رفقا امیدوارم که حالتون خوب باشه. البته تو کارگاه پوینتر مگه میشه کسی حالش بد باشه ☺ پس بی معطلی بریم سراغ سوال جذابمون...

یکی از شاخه‌های مهم کامپیوتر، **رمزنگاری**^۱ هست. هدف این شاخه اینه که اطلاعات حساس رواز دید بقیه‌ی کاربرها پنهان کنه. مثلا فرض کنین می‌خواین اطلاعات مهمی (مثل رمزهای عبور، اطلاعات کارت ملی...) رو با اینترنت برای کسی بفرستین. طبق ساختار اینترنت که بعدا تو درس شبکه‌های کامپیوتری حسابی باهاش آشنا می‌شین، این اطلاعات شما تا رسیدن به مقصد از چند تا کامپیوتر دیگه هم رد می‌شه و شما قاعدتا نمی‌خواین اطلاعات حساستون بین راه توسط کسایی که به این کامپیوترها دسترسی دارن، خونده بشه. پس این اطلاعات رو رمزنگاری می‌کنین تا این کامپیوترها متوجه جزییات اطلاعات ارسال‌شده‌ی شما نشن و اگر هم اونا رو نگاه کردن، چیز دیگه‌ای به جای اطلاعات اصلی ببینن. در رمزنگاری به داده‌ی خام **Plaintext** و داده‌ی رمزنگاری شده **Ciphertext** گفته می‌شود.



بیشتر بدانید

برای مطالعه



برای مثال، معروفترین اون‌ها (که در سیستم‌های امروزی خیلی پرکاربرده) الگوریتم RSA هست که اساس امنیتش روی این مساله‌س که شکستن یک عدد بزرگ به مقسوم‌علیه‌های اولش برای کامپیوترهای در دسترس امروزی خیلی زمان‌بره.

برای اینکه بخوام یه دیدی براتون ایجاد کنم، بهتره اینطور بگم که کامپیوترهای امروزی برای شکستن یک متن رمزگذاری شده با الگوریتم RSA، به 300 تریلیون سال احتیاج دارن!!

چندین و چند متد مختلف برای رمزنگاری وجود داره که بر اساس تئوری‌های ریاضیاتی و پیچیدگی محاسباتی (یعنی سخت بودن محاسبه برای کامپیوترها) طراحی‌شدن.



حالا می‌خوایم ما هم یه برنامه‌ی رمزنگاری به نسبت ساده‌ای رو با زبان C بنویسیم. اسم این الگوریتم Square Code هست. ورودی این الگوریتم یک متن انگلیسی ساده و خروجیش متن رمزنگاری‌شده‌ی همون متن می‌شه.



اول اول، باید ورودی رو نرمالایز کنیم!! یعنی چی؟ یعنی علایم سجاوندی (مثل نقطه، ویرگول، نقطه ویرگول و ...) و whitespace‌ها رو حذف کنیم و همه‌ی حروف رو هم به شکل lowercase تبدیل کنیم.

بعد این متن باید جوری مرتب بشه که به شکل یه مربع در بیاد.

بذارین اینجاشو با مثال بگم:

Input:

"If man was meant to stay on the ground, god would have given us roots."

Step 1: Normalization:

"ifmanwasmeanttostayonthegroundgodwouldhavegivenusroots"

Step 2: 54 characters => $c = 8$, $r = 7$, make rectangle:

"ifmanwas"

"meanttost"

"tayonthe"

"groundgo"

"dwouldha"

"vegivenu"

"sroots "

اندازهی این مربع ($r \times c$) باید به نسبت طول ورودی تنظیم بشه، به جوری که $c \geq r$ و $c - r \leq 1$ باشه. این جا c یعنی تعداد ستون ها (Columns) و r تعداد سطرها (Rows) هست.

متن رمزنگاری شده با خواندن از این مربع تولیدشدهی ما به صورت عمودی تولید می شه. (از بالا به پایین و از چپ به راست) مثلا خروجی اولیهی ورودی نمونهی ما به این شکل در می آید:

"imgdvsfearwermayoogoanouuiontnnlvtwttddesaohghnsseoau"





برای تولید یه خروجی نهایی که قابل بازگشت به شکل اصلیش باشه، باید توجه کنیم که برای عبارتهایی که n کاراکتر کوتاهتر از یک مربع کامل ($c * r$) دارن، باید در انتهای n سطر آخر مربع، یک space اضافی بذارین و آخر سر سطرهای Square Code تولید شده رو با یه separator یا جداکننده به انتخاب خودتون (مثلا یک ! یا ؟) جدا کنین، که در نهایت خروجی نهایی ما به این شکل در می‌آید:

```
"imtgdvs!fearwer!mayoogo!anouuio!ntnnlvt!wtdddes!aohghn !sseoau "
```



به عنوان یک تمرین امتیازی می‌تونین کدی رو بزنین که برعکس این پروسه رو انجام میده، یعنی سطرهای یک Square Code رو تو ورودی می‌گیره و متن اصلی اون رو خروجی میده. برای مثال، با روی هم چیدن قسمت‌های مختلف خروجی نهایی قسمت قبل، به شکل روبه‌رو می‌رسیم که می‌تونیم اون رو به ورودی اولیه تبدیل کنیم.

"imtgdvs"

"fearwer"

"mayoogo"

"anouuio"

"ntnnlvt"

"wtdddes"

"aohghn "

"sseoau "

;