کدنامه

مبانی برنامه‌نویسی مهندسی‌کامپیوتر دانشگاه صنعتی شریف

WHAT IS CMAKE

سید محمد حسین حسینی

احتمالا تا اینجای کد زدن با C خیلی با چالش عجیب و غریبی هنگام build کردن و خروجی گرفتن از برنامه‌تان (به جز خطاهایی که گاهی اوقات با آن‌ها مواجه می‌شوید) روبرو نشده‌اید. اما فرض کنید که خروجی برنامه‌ای که می‌نویسید قرار است روی یک سیستم نصب شود، یا فرض کنید برنامه‌ای با فایل‌های فراوان نوشتید که از کتابخانه‌های غیرپیشفرض C استفاده می‌کند و این برنامه روی هر سیستمی که قرار است اجرا شود باید چک کنیم آن کتابخانه نصب شده است یا نه و... . یا در یک سطح بالاتر می‌خواهید یک قابلیت اختیاری در برنامه‌تان داشته‌باشید که فرضا در صورتی که یک کتابخانه خاص در دسترس باشد آن قابلیت فعال باشد. یا مثلا احتمالا در گلابی با این مسئله مواجه شده‌اید که بعضی کتابخانه‌ها یا بعضی دستورات فقط در یک سیستم عامل خاص یا با شرایط خاصی کار میکنند و این باعث می‌شود کد نتواند cross-platform باشد و اگر سیستم‌عامل متفاوت باشد باید از یک کتابخانه دیگر استفاده کرد و ... . اینها بخشی از چالش‌هایی‌ست که ممکن است در مورد build کردن یک نرم‌افزار با آن مواجه شد.

Cmake یک نرم‌افزار متن‌باز جهت مدیریت فرآیند build است که این قابلیت را در اختیار توسعه‌دهنده قرار می‌دهد تا پارامترهای مورد نیاز برای build را درون یک فایل متنی ذخیره کند. این فایل بعدتر توسط CMake استفاده می‌شود تا فایل‌های مورد استفاده ابزار build را تولید کند.

کار با CMake به این صورت است که داخل هر کدام از دایرکتوری‌های پروژه یک فایل CMakeLists.txt قرار می‌گیرد. هر کدام از این فایل‌ها شامل تعدادی دستور به زبان CMake هستند. هر دستور به این شکل است:

```
command (args…)
```

که به جای command نام دستور می‌آید و args نیز نمایانگر ورودی‌های آن است که با اسپیس یا سایر white-spaceها جدا می‌شوند. CMake همچنین از متغیرها هم پشتیبانی می‌کند که هر متغیر می‌تواند رشته یا لیستی از رشته‌ها باشد. متغیرهای محیطی سیستم و مقادیر registry ویندوز قابل استفاده مستقیم در CMake هستند.

برای مثال یک پروژه Hello World را در نظر بگیرید.

```
project (Hello)
add_executable (Hello Hello.c)
```

دستور project نام workspace نهایی را مشخص می‌کند و دستور add_executable به پروژه Hello یک executable target می‌افزاید.

دستور add_library یک کتابخانه را مشخص می‌کند. مثلا

```
add_library (foo a.c b.c)
```

کتابخانه‌ای به نام foo تعریف می‌کند که فایل‌های آن a.c و b.c هستند. به طور کلی برای استفاده از CMake و نوشتن فایل CMakeLists، دیدن فایل‌های نمونه پیشنهاد می‌شود تا درک ساختار کلی این فایل‌ها ساده‌تر باشد. می‌توانید با سرچ کردن فایل‌های نمونه بسیاری ببینید یا برای دیدن فایل‌های ساده‌تر از تی‌ای‌ها کمک بگیرید. در این لینک یک دموی خیلی ساده از محتویات فایل‌های CMakeLists می‌توانید ببینید.

برای به کارگیری CMake راه‌های مختلفی وجود دارد. یکیاز این راه‌ها راه که اندکی به آن اشاره می‌کنیم، استفاده از دستور cmake در کامندلاین (نیازمند نصب) است. شیوه به کارگیری این دستور به طور کلی به این شکل است.

```
Usage

  cmake [options] <path-to-source>
  cmake [options] <path-to-existing-build>
  cmake [options] -S <path-to-source> -B <path-to-build>

Specify a source directory to (re-)generate a build system for it in the
current working directory.  Specify an existing build directory to
re-generate its build system.
```

پس از اجرای این دستور، با توجه به generator انتخاب شده فایل‌های مدنظر برای build را تولید میکند. (generatorهای CMake در واقع آن بخشی از CMake هستند که وظیفه تولید فایل‌های ورودی برای یک ابزار build را دارند. ابزارهای مختلف build مثل IDEهای مختلف ممکن است فایل‌های ورودی متفاوتی بخواهند که در این فایل‌ها تنظیمات و کانفیگوریشن‌های مربوط به فرایند build هم مشخص شده‌است. می‌توان generator را از بین generatorهایی که قابل استفاده است پس از اجرای دستور cmake -help دید و با استفاده از آپشن -G انتخاب نمود یا از generator پیشفرض استفاده کرد.) فرضا اگر از لینوکس استفاده می‌کنید و generator انتخابی Unix Makefiles (که معمولا generator پیش‌فرض است) باشد دستور cmake به اصطلاح makefileها را برای پروژه تولید می‌کند که با استفاده از دستور make (در دایرکتوری مقصد) به سادگی build انجام می‌شود. (دستور make install هم در صورت وجود targetهایی برای نصب که مثلا کتابخانه مدنظر نصب شود، آن‌ها را نصب می‌کند.) فایل‌ها را می‌توان برای بعضی IDEها هم generate کرد (به جای make) که در این صورت با باز کردن این فایل‌ها در آن IDE می‌توان به صورت عادی build کرد. ابزارهای دیگری غیر از ابزار کامندلاین نیز برای استفاده از CMake وجود دارد مثل cmake-gui یا ccmake.

generatorهای قابل استفاده بستگی به IDEها و ابزارهای build نصب شده روی سیستم دارد. CMake در واقع فایل‌های مورد نیاز برای build را برای آن ابزار تولید میکند. (مثلا فرض کنید تنظیمات مورد نیاز برای build را که مشخص کردیم به فرمت همان ابزار ذخیره می‌کند.)

پس به طور کلی اگر بخواهیم جمع‌بندی کنیم ما تعدادی ابزار build داریم (مثلا همین make) که هر کدام فرم خاص خودشان را دارند و تنظیمات توسط فایل‌هایی به همان فرم و زبان آن‌ها در اختیارشان قرار می‌گیرد و آن‌ها عمل build را انجام می‌دهند. CMake برحسب تنظیمات و کانفیگوریشن‌های موجود در فایل‌های CMakeLists و وضعیت سیستم و اینکه چه کتابخانه‌ای‌هایی نصب هستند و چه سیستم عاملی مورد استفاده است و ... فایل‌های لازم برای ابزار انتخابی را تولید می‌کند.

اگر ساختار کلی CMake و شیوه استفاده از آن و رسالت آن را درک کرده باشید می‌توانید از documentation آن استفاده کنید و کار خود را راه بندازید. کتاب Mastering CMake نوشته Ken Martin و Bill Hoffman کتاب خوبی برای مطالعه در این زمینه است و بر حسب نیاز با توجه به فهرست آن می‌توان به بخش مورد نیاز مراجعه کرد.

```
Generators

The following generators are available on this platform (* marks default):
  Green Hills MULTI           = Generates Green Hills MULTI files
                                (experimental, work-in-progress).
* Unix Makefiles              = Generates standard UNIX makefiles.
  Ninja                       = Generates build.ninja files.
  Ninja Multi-Config          = Generates build-<Config>.ninja files.
  Watcom WMake                = Generates Watcom WMake makefiles.
  CodeBlocks - Ninja          = Generates CodeBlocks project files.
  CodeBlocks - Unix Makefiles = Generates CodeBlocks project files.
  CodeLite - Ninja            = Generates CodeLite project files.
  CodeLite - Unix Makefiles   = Generates CodeLite project files.
  Eclipse CDT4 - Ninja        = Generates Eclipse CDT 4.0 project files.
  Eclipse CDT4 - Unix Makefiles= Generates Eclipse CDT 4.0 project files.
  Kate - Ninja                = Generates Kate project files.
  Kate - Unix Makefiles       = Generates Kate project files.
  Sublime Text 2 - Ninja      = Generates Sublime Text 2 project files.
  Sublime Text 2 - Unix Makefiles
                              = Generates Sublime Text 2 project files.
```

WHAT IS UNDEFINED BEHAVIOR

امیرعلی سلیمی

تا حالا به این موضوع فکر کردین که در زبان C، آرایه‌ها می‌تونن اندیس منفی بپذیرن یا نه؟ اگر از یک آرایه، به عضوی با اندیس منفی دسترسی پیدا کنیم، چه اتفاقی میفته؟ به طور مثال، فرض کنین تکه کد زیر را اجرا کنیم:

```
int a[4];
printf("%d\n", a[-10]);
```

احتمالا اگر این کد را اجرا کنین، با جواب‌هایی عجیب و غریب و متغیر مواجه می‌شین (مثلا من با سه بار اجرای این کد، عددهای 1344334450، 1194977906 و 1478694286 خروجی گرفتم).

به مواقعی که توی برنامه‌مون تکه کدی داشته باشیم که رفتار اون تکه کد توسط زبان به صورت دقیق تعریف نشده و غیر قابل پیش‌بینی باشه، Undefined Behavior یا به اختصار UB می‌گن. این مفهوم در زبان‌های C و C++ از اهمیت خاصی برخورداره و خیلی جاها دیده میشه، پس خوبه که با مزیت‌ها و مشکلات اون آشنا بشیم.

در زبان C، طبق تعریف، مقدار a[i] برابر با *(a + i) هست. در اینجا اگر i منفی باشه هیچ مشکلی وجود نداره، مثلا در تکه کد زیر:

```
int arr[10];
int* p = &arr[2];
int x = p[-2];
// valid: accesses arr[0]
```

تکه کد بالا به درستی اجرا میشه و مقدار arr[0] رو در x می‌ریزه. اما در تکه کد اول، به اندیسی از آرایه دسترسی پیدا می‌کنیم که خارج از حدود آرایه هست. در بعضی زبان‌ها در این مواقع با خطای Index Out Of Bound مواجه می‌شیم، اما در زبان C این کار الزاما باعث ایجاد خطا نمیشه، بلکه رفتار غیر قابل پیش‌بینی و اصطلاحا UB رو به همراه داره. یعنی ممکنه هر خروجی‌ای رو از برنامه دریافت کنیم، یا حتی ممکنه با خطا روبرو بشیم! حالا یک سوال پیش میاد:

علت اینکه زبان C روی UB سخت‌گیری نداره و خطا نمیده چیه؟ به عنوان یک مثال دیگه، اگر در یک تابع (مثلا در خود main) متغیری رو تعریف کنین و مقداردهی اولیه نکنین، اینجا هم مقدار متغیر از مواردی هست که رفتار غیر قابل پیش‌بینی داره. چرا زبان C هم مثل زبان جاوا، هر متغیری رو به مقدار صفر مقداردهی اولیه نمی‌کنه و اجازه میده که UB داشته‌باشیم؟

علت این موضوع در اولویت‌هایی که طراح زبان مدنظر خودش قرار داده نهفته هست! طراح زبان جاوا، ایمنی و قابل پیش‌بینی بودن رو در اولویت بالایی قرار داده، و با توجه به این موضوع زبان رو طراحی کرده. اما در طراحی زبان C، کارایی و سرعت اجرا در اولویت بالاتری قرار گرفته. اینجاست که طراح زبان در مواردی باید رفتارهای غیر قابل پیش‌بینی رو بپذیره و تشخیص اون‌ها رو به برنامه‌نویس بسپره، تا بتونه کارایی زبان رو تضمین کنه.

علاوه بر دو مثالی که در بالا دیدیم (دسترسی به اندیس خارج از محدوده و عدم مقداردهی اولیه)، موارد دیگری هم وجود داره که UB رخ میده، مثلا دسترسی به اشاره‌گری که مقدارش NULL باشه، یا به خانه‌ای از حافظه اشاره کنه که در اختیار برنامه‌نویس نیست. در این نوع دسترسی‌ها، شما الزاما با خطا مواجه نمی‌شید و امکان داره که با خروجی‌های غیرمنتظره‌ای مواجه بشید. البته امکان اینکه خطای Segmentation Fault دریافت کنید هم وجود داره.

```
int *p = (int*)malloc(sizeof(int));
int *q = (int*)
realloc(p, sizeof(int));
*p = 1;
// UB access to a pointer that was
//passed to realloc
*q = 2;
if (p == q)
// UB access to a pointer
//that was passed to realloc
printf("%d%d\n", *p, *q);
```

شاید در اجرای تکه کد بالا، با خودتون بگین: خب این کد که مشکلی نداره، realloc میاد قسمتی از حافظه رو که قبلا توسط malloc به p اختصاص داده‌شده رو دوباره allocate می‌کنه و به q اختصاص می‌ده. اما اگر به تعریف realloc نگاه کنیم، می‌بینیم که این تابع در مواردی ممکن هست که قسمتی از حافظه که به p اختصاص داده‌شده رو به سیستم عامل برگردونه، و قسمت جدیدی از حافظه رو اشغال کنه و به q اختصاص بده. در این صورت تکه کد بالا منجر به UB می‌شه، چون اشاره‌گر p به قســـــمتی از حافظه اشاره می‌کنه که دیگه به برنامه ما اختصاص نداره. به خاطر همین هروقت از realloc استفاده کردین، نباید از اشاره‌گر قبلی استفاده کنین.

تکه کدهایی که منجر به Undefined Behavior می‌شن، می‌تونن ساعت‌ها برنامه‌نویس رو با خروجی‌های غیرمنتظره درگیر دیباگ کردن برنامه کنن. اینجاست که شناختن UB-ها اهمیت زیادی پیدا می‌کنه، در این لینک می‌تونید اطلاعات بیشتری درباره اون‌ها پیدا کنین.

With Undefined Behavior, Anything is Possible

HEADER FILES IN C

طاها اکبری

فرض کنید برنامه‌ای به زبان c دارید که می‌خواهید به عنوان زیر برنامه در برنامه‌ای دیگر استفاده کنید یا آن را به عنوان کتابخانه منتشر کنید با صرف نظر از فایل هدر دو گزینه دارید:

۱. به اشتراک گذاشتن فایل سورس: در این صورت به دو مشکل بزرگ برخواهیم خورد:

اول این که کدی زده‌اید برای عموم قابل مشاهده خواهد بود و اگر از دید تجاری به آن نگاه کنیم نمی‌توانید از این کد بهره برداری کنید.

از طرف دیگر احتمالا فردی که می‌خواهد برمبنای این کد کار کند، در پیاده‌سازیش مبتنی بر پیاده سازی کد شما کد زده باشد و با تعویض بخش هایی از کد شما بدون تغییر در عملکرد توابع ممکن است کدش به مشکل برخورد کند.

۲. به اشتراک گذاشتن کد کامپایل شده: در این صورت نیز برنامه نویسی که می‌خواهد از کد استفاده کند اطلاعی درمورد توابعی که کد شما در اختیارش قرار میدهد ندارد و باید یک داکیومنتیشن برای این توابع و آرگومان ها و... آن ها بنویسیم

راه حل پیشنهادی برای حل این مشکلات فایل‌های هدر است. به این شکل که کد c که حاوی پیاده سازی توابع مدنظر است را کامپایل کرده و همراه با آن یک فایل هدر می‌دهیم که شامل prototype توابع پیاده سازی شده می‌باشد.

نکاتی چند در مورد فایل هدر:

۱. فایل هدر معمولا شامل پیاده سازی نمی‌شود اما داشتن پیاده سازی در آن منعی ندارد و میتوان توابعی را که بدنه کوتاه یا غیرمهمی دارند را آنجا پیاده کرد.

۲. اگر یک فایل هدر در فایلی دو بار include شود به ارور برخورد میکنیم. برای جلوگیری از این اتفاق از "conditional preprocessor directive" استفاده می‌گردد و اگر فایل هدر قبلا include شده بود دیگر include نمیگردد.

```
#ifndef HEADER
#define HEADER
// Content of header
#endif
```

به عنوان مثال فرض کنید تابعی نوشته‌اید که دو عدد را یکدیگر جمع می‌کند و میخواهید آن تابع را درون فایل دیگری استفاده کنید. فایلی که پیاده‌سازی تابع جمع در آن نوشته شده به شکل زیر خواهد بود مثلا add.c

```
#include "add.h"
int add(int a, int b) {
return a + b;
}
```

در تعریف فایل هدر نیز صرفا پروتوتایپ تابع را قرار خواهیم داد:

```
#ifndef CTEST_ADD_H
#define CTEST_ADD_H
int add(int, int);
#endif //CTEST_ADD_H
```

و در نهایت میتوانیم از فایل add.h در پروژه اصلیمان استفاده می‌کنیم.

```
#include <stdio.h>
#include "add.h"
int main() {
int a, b;
scanf("%d%d", &a, &b);
printf("%d", add(a, b));
}
```

SAY HELLO TO TESTING ...!

آرش یادگاری

در تولید یک محصول لازمه تا جایی که امکان داره تست‌های مختلفی طراحی بشن تا از درستی برنامه اطمینان حاصل کنیم. فرض کنید شما نویسنده این تست‌ها بودین. چه راهی برای اجرا کردن برنامه انتخاب می‌کردین؟ در این شرایط معمولا روش تست کردن دستی(ران کردن برنامه و چک کردن نحوه اجرا برنامه) بسیار وقت‌گیر و هزینه‌برن. علاوه بر این، یک تغییر در قسمت کوچکی از برنامه باعث تکرار این فرآیند میشه.

برای رفع این مشکل می‌تونیم از روش‌های تست کردن اتوماتیک استفاده کنیم. در این قسمت سعی داریم تا یکی از روش‌های تست کردن یعنی unit test رو معرفی کنیم و همینطور با یک فریم‌ورک برای نوشتن unit test در C آشنا بشیم.

Unit test چیست؟

در تعریف کلی، unit test یعنی تست کردن کوچکترین واحدهای برنامه. در کدنامه‌های ویژه برنامه‌نویسی پیشرفته، در مورد اینکه کوچکترین واحد برنامه دقیقا چه معنی میده بیشتر صحبت می‌کنیم ولی فعلا فرض کنید که در C منظور همون توابع هستند.

برای رفع این مشکل می‌تونیم از روش‌های تست کردن اتوماتیک استفاده کنیم. در این قسمت سعی داریم تا یکی از روش‌های تست کردن یعنی unit test رو معرفی کنیم و همینطور با یک فریم‌ورک برای نوشتن unit test در C آشنا بشیم.

ابزارهای مورد نیاز

برای نوشتن تست‌هامون از کتابخانه Cmocka استفاده می‌کنیم که از کتابخانه cmockery توسط google نوشته و توسعه داده شده است.

- **راه اندازی در لینوکس**

برای نصب کتابخانه کافی است از دستور زیر استفاده کنید:

```
sudo apt install libcmocka-dev
```

- **راه اندازی در ویندوز**

برای نصب در ویندوز بهتر است مخزن Cmocka را clone کنید. مراحل تکمیلی رو می‌تونید به کمک این لینک انجام بدین.(توصیه می‌کنیم ولی رو لینوکس اینکارو انجام بدین و اگه نصب نکردید هرچه سریع‌تر اینکارو بکنید.)

نوشتن اولین تست

همونطور که گفتیم، کوچک‌ترین واحدهای برنامه ما، تابع‌ها هستند. اگر این قسمت‌های کوچک به درستی وظیفه خودشون رو انجام بدن در نهایت برنامه ما به درستی کار میکنه، به همین سادگی. برای ادامه بیاید باهم یادبگیریم که چطور یک تست بنویسیم.

- **پیاده‌سازی فاکتوریل**

در تکه کد زیر یک پیاده‌سازی ساده از تابع فاکتوریل رو می‌بینید:

```
int factorial(int n) {
  int result = 1;
  for (int i = 1; i <= n; i++) {
    result *= i;
  }
  return result;
}
```

این تابع رو در فایلی به اسم factorial.c ذخیره می‌کنیم. دقت کنید که یک فایل header هم برای توابعی که می‌خواین از اون فایل تست کنید بسازید. به طور مثال فایل هدر factorial.h به صورت زیر خواهد بود:

```
#ifndef TESTING_FACTORIAL_H
#define TESTING_FACTORIAL_H
int factorial(int n);
#endif //TESTING_FACTORIAL_H
```

- **include کردن فایل‌های لازم**

برای نوشتن فایل تست برای factorial یک فایل به نام test_factorial می‌سازیم. فایل‌های تست باید موارد زیر رو include کرده باشن.

```
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmocka.h>
#include "TO_BE_TEST.h"
 /* the header file
 of unit under test */
```

از اونجایی که قراره factorial رو تست کنیم بجای TO_BE_TEST از factorial استفاده می‌کنیم.

- **assertions**

وقتی می‌خواهیم درستی یک چیزی رو چک بکنیم باید یک ادعا در مورد عملکرد اون داشته باشیم و درصورتی که این ادعا با خروجی تابعی که در حال تست کردنش هستیم سازگار بود، تست تایید یا به اصطلاح pass می‌شود. همه‌ی کتابخونه‌های نوشته شده برای تست شامل تعداد زیادی از این assertionها هستن. در ادامه تعدادی از assertionهارو بررسی می‌کنیم.

- **assert_true(scalar expression) :**

درستی عبارت داخل پرانتز چک میشه و در صورتی که غلط باشه تست fail میشه.

- **asssert_ptr_equal(void *a, void *b):**

اگر دو اشاره‌گر به یک خانه از حافظه اشاره نکنن، تست fail میشه.

- **assert_in_set(**
LargetsIntegerType value,
LargestIntegrType values[ ], size_t count
):

اگر مقدار value داخل آرایه values با سایز count نباشه، تست fail میشه. (دقت کنید که به طور پیشفرض مقادیر عددی بزرگترین مقدار ممکن در نظر گرفته شدن تا نیاز به پیاده‌سازی‌های مختلف نباشه)
برای مشاهده لیست کامل assertionsها می‌تونید به این لینک سر بزنید.

- **توابع تست**

حالا که با assertionsها آشنا شدیم کافیه که تابعی بنویسیم که با استفاده از assertionها درستی حدس ما در مورد خروجی رو بررسی کنه.
بدنه توابع تست به صورت زیر تعریف می‌شن:

```
static void test_name() {
//some assertions
}
```

استفاده از static به این معنی هست که این تابع فقط برای این فایل قابل استفاده است و نمی‌تونیم داخل فایل‌های دیگه ازش استفاده کنیم. همینطور دقت کنید که یک تابع تست هیچ خروجی نداره و فقط درستی یه سری ادعا رو بررسی می‌کنه.

به مثال خودمون برگردیم. با توجه به چیزایی که تا الان یادگرفتیم، اولین تست رو برای بررسی !۰ می‌نویسیم.

```
static void when_0_expect_1() {
assert_int_equal(factorial(0), 1);
/*OR*/
assert_true(factorial(0) == 1);
}
```

برای انتخاب نام تابع تست می‌تونید از روش‌های مختلفی استفاده کنین. برای آشنایی با شیوه‌های نامگذاری تابع‌های تست یه سر به این لینک بزنید.
دو تابع تست دیگه هم در ادامه تعریف می‌کنیم.

```
static void when_5_expect_120() {
 assert_int_equal(factorial(5), 120);
}
static void when_10_expect_5() {
 assert_int_equal(factorial(10), 5);
}
```

- **اجرای تست‌ها**

حالا کافیه این فایل هم ذخیره کنید و در ادامه دستورات زیر رو برای کامپایل کردن و اجرای تست‌ها انجام بدین. (سه تا فایلی که نوشتیم رو تو یک پوشه ذخیره کنید.)

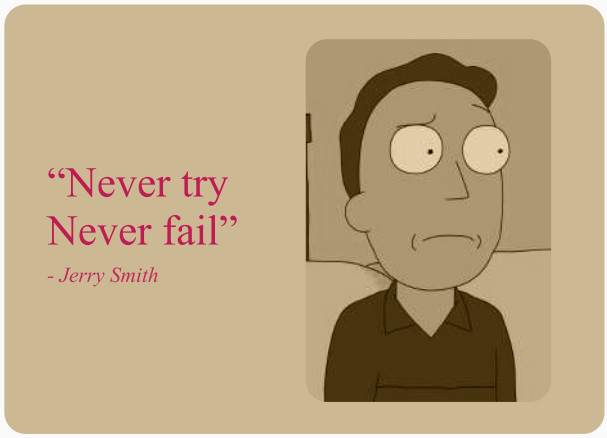
```
gcc -c factorial.c
gcc -c test_factorial.c
gcc -o "testmain" ./test_factorial.o ./
factorial.o -lcmocka
./testmain
```

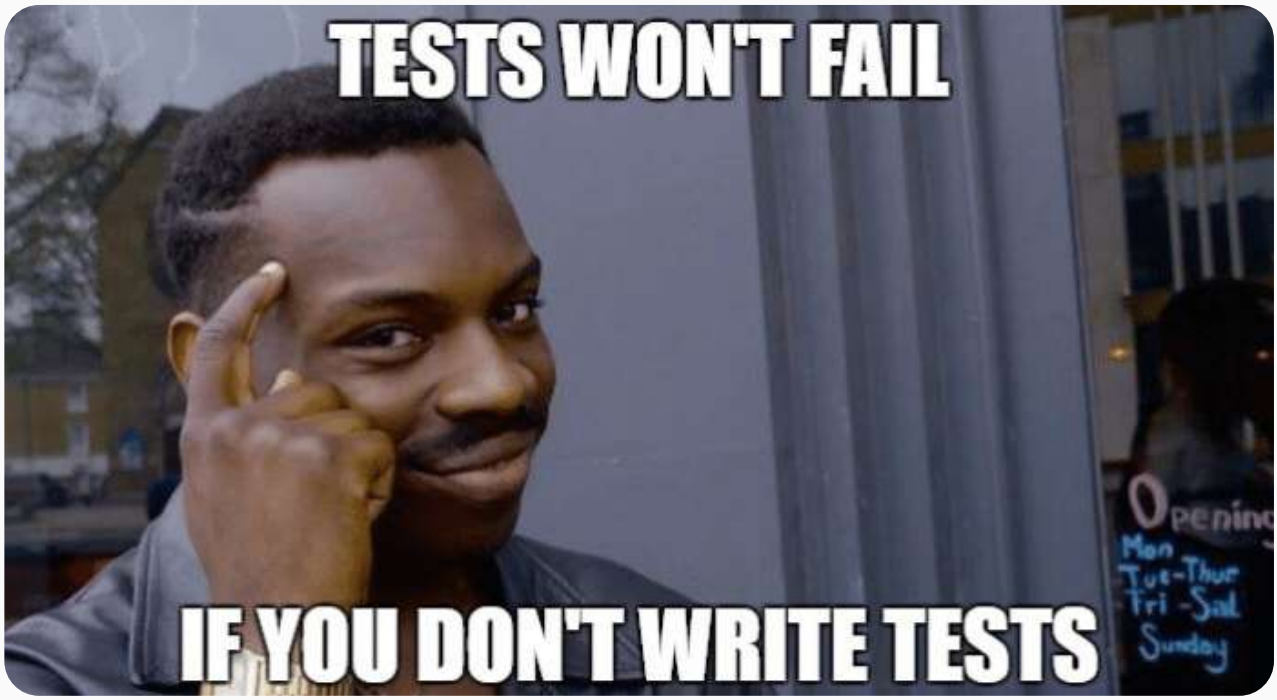
برای دانلود فایل‌های تست و همینطور یک فایل bash برای اینکه بتونین راحت‌تر فایل‌هاتون رو کامپایل و تست کنید به گیتهاب کدنامه برید.
برای نوشتن تست‌های پیچیده‌تر و جالب‌تر کدنامه‌های بعدی رو از دست ندین

در صورتی که همه چی درست پیش رفته باشه خروجی تست به این صورت گزارش میشه:

```
==========] Running 3 test(s).
[ RUN      ] when_0_expect_1
[       OK ] when_0_expect_1
[ RUN      ] when_5_expect_120
[       OK ] when_5_expect_120
[ RUN      ] when_10_expect_5
[  ERROR   ] --- 0x375f00 != 0x5
[   LINE   ] --- test_factorial.c:19: error: Failure!
[  FAILED  ] when_10_expect_5
==========] 3 test(s) run.
[  PASSED  ] 2 test(s).
[  FAILED  ] 1 test(s), listed below:
[  FAILED  ] when_10_expect_5

1 FAILED TEST(S)
```





PROGRAMMING ... IN STRUCTURED WAY!

محمد بروغنی

در بین دهه‌های ۵۰ تا ۶۰ میلادی، کم کم زبان‌های برنامه‌نویسی شروع به شکل گیری کردن و خانواده زبان‌های اسمبلی و زبان‌های سطح بالاتری مثل Algol و FORTRAN در این بازه متولد شدن. یکی از اصلی‌ترین اجزای زبان‌های برنامه‌نویسی (چه زبان‌های سطح بالاتر و چه زبان‌های سطح پایین)، دستور goto بود.

جهت یادآوری و آشنایی، یک مثال از دستور goto در زبان C رو بررسی می‌کنیم:

```
#include <stdio.h>

int main()
 {
 /* initialize variable whose
absolute value is to be printed*/
 int n;
 scanf("%d", &n);

 /* if the number is already
positive, print it directly */

 if(n>=0)
 {
 // goto statement
 goto positive;
 }

 /* to make the number positive
multiply by -1 */
 n = n*(-1);

 // declare positive label
 positive :
 printf("The absolute value is
%d", n);

 return 0;
}
```

توی خط ۷ از برنامه، عدد n از کاربر ورودی گرفته میشه. در خط ۱۰ چک می‌کنیم اگر که ورودی، بیشتر مساوی ۰ باشه، خط ۱۳ اجرا بشه. اگر ورودی بیشتر مساوی ۰ باشه، خط ۱۳ اجرا میشه و ادامه اجرای کد از خطی که لیبل positive تعریف شده، ادامه پیدا میکنه، یعنی خط بعدی که اجرا میشه، خط ۲۱ خواهد بود. ولی اگر ورودی منفی باشه، روال عادی اجرا برنامه ادامه پیدا میکنه و خط‌های ۱۷ و ۲۱ هر دو اجرا میشن.

در همین بازه زمانی که ذکر شد، بعضی از برنامه‌نویس‌ها حس کردن که وجود دستور goto در برنامه‌ها، خیلی وقتها باعث میشه که کد، خوانایی خودشو از دست بده و سخت‌تر بشه روند اجرایی اون رو دنبال کرد. با این حال یکسری حالت‌های استفاده از دستور goto هم هستند که برخلاف باقی استفاده‌هایی که از goto میشه، فهمشون ساده است و مغایرت چندانی با خوانایی کد ندارند و پر استفاده هم هستن. مثلا موقعی که از دستور goto برای ایجاد یک loop توی برنامه استفاده می‌کنیم(خیلی از زبان‌های برنامه‌نویسی در اون ایام، دستوراتی مثل while و for رو که برای ایجاد loop تو برنامه استفاده میشن رو نداشتن و برای پیاده‌سازی این ساختارها هم باید از goto استفاده میکردن).

جناب Edsger Dijkstra، که از اولین برنامه‌نویس های کشورش(هلند) هم بود، در سال ۱۹۶۶، دنباله‌ای از مقالات از جمله Go To Statement Considered Harmful رو منتشر کرد و نظر خودش رو در مورد این ساختار و عدم نیاز به استفاده از اون در زبان‌های برنامه‌نویسی سطح بالا بیان کرد. تو این مقاله‌ها در

در نهایت، این الگو جدید در برنامه‌نویسی پذیرفته شد و دستوراتی مثل while و for که شکل‌های سطح بالاتری از goto بودن به زبان های برنامه‌نویسی اضافه شدن. کم‌کم هم دستور goto از خیلی از زبان‌های برنامه‌نویسی(سطح بالا) جدیدی که ساخته می‌شدن، حذف شد. (البته در خیلی از زبان‌ها، با وجود اینکه دستور goto وجود ندارد، ولی ساختارهایی مثل continue و break وجود دارد که عملا همان کار دستور goto را انجام میده. این دستورها به این علت که در خیلی از شرایط به بالا رفتن خوانایی کد کمک می‌کنن، در این زبان‌ها وجود دارن.)

پس این شکل جدیدتر از برنامه‌ها، از ۳ ساختار (structure) اصلی برای تغییر روند اجرایی کد تشکیل شده:

Sequence

عادی، اجرای خط به خط کد در برنامه‌ها، به عنوان مثال:

```
doStepOne();
doStepTwo();
```

Selection

ساختاری است که با استفاده از دستور if در برنامه ها از آن استفاده میکنیم، یعنی:

```
if (someBooleanValue())
doThisStep();
else
doOtherStep();
```

Iteration

که همان دستور while آشنای خودمون در زبان‌های برنامه‌نویسی است:

```
while(someBooleanValue())
doThisStep();
```

که به همین علت به این شکل از برنامه‌نویسی، structured programming میگیم. در نتیجه، رعایت کردن این شکل از برنامه نویسی، به ما این امکان رو میده که با محدود کردن خودمون در استفاده از یک دستور پر قدرتی مثل goto، به خوانایی در کد و کاهش پیچیدگی دست پیدا کنیم!

“Computer scientists main challenge is Not to get confused by the complexities of his/her making”

- Edsger Dijkstra

POINTERS
WHY SHOULD WE USE THEM

سید پارسا نشایی

یکی از مهم‌ترین مباحث درسی مبانی برنامه‌سازی که در این نیم‌سال با آن‌ها آشنا شده‌ایم، «pointer» است. همه می‌دانیم که این ویژگی زبان C، منبع انواع و اقسام اشتباهات در کدنویسی است. به عنوان مثال، فراموش کردن نیاز به free کردن یک pointer که فضای آن توسط توابعی مانند malloc گرفته شده است، یکی از متداول‌ترین خطاهای مرتکب شده توسط برنامه‌نویس‌ها (چه تازه‌کار و چه کارکشته!) است. (اگر هنوز شک دارید که استفاده از pointer ها به چه اندازه می‌تواند سبب پیچیدگی کد شود، کافی است به سوالات پایان‌ترمی که در آن شرکت کردید، نگاهی بیندازید!) به همین جهت، ممکن است برای شما این سوال پیش آمده باشد که چرا این ویژگی در زبان C وجود دارد و TA های درس مبانی برنامه‌سازی به این اندازه روی یادگیری درست آن تاکید دارند؟

پشت پرده‌ی pointer ها

از جلسات درس و حل تمرین، می‌دانیم که pointer ها در حقیقت و ماهیت خود، متغیرهایی مانند سایر متغیرها هستند، با این تفاوت که به جای یک مقدار عددی عادی (مانند ۱۰ یا ۵ یا ۲- و نظایر آن)، «آدرس یک خانه‌ی دیگر حافظه» در آن‌ها نوشته شده است. برای درک چرایی نیاز به چنین مفهومی، لازم است در بررسی روند اجرای یک برنامه عمیق‌تر شویم.

می‌دانیم برنامه‌های C ما همگی به زبان اسمبلی ترجمه شده و سپس این زبان به زبان ماشین (رشته‌ای از صفرها و یک‌ها) تبدیل شده و روی رایانه اجرا می‌شوند. در این مقاله، زبان اسمبلی و زبان ماشین را معادل در نظر می‌گیریم، زیرا در اکثر مواقع (و در حد مورد نیاز برای این نوشته)، این دو صرفا دو نمایش متفاوت از یک زبان هستند. رایانه‌ها به طور مستقیم، صرفا قادر به درک زبان اسمبلی هستند و این زبان نیز تنها شامل دستوراتی بسیار ساده است. به عنوان مثال، چند نمونه از این دستورات عبارت‌اند از:

- **دستور mov:** جابه‌جایی محتویات یک خانه از حافظه به خانه‌ای دیگر.
- **دستور neg:** قرینه کردن محتویات یک خانه از حافظه
- **دستورات add و sub و mul و div:** انجام چهار عمل اصلی روی دو خانه از حافظه و ذخیره‌ی نتیجه در خانه‌ای دیگر در حافظه.
- **دستور jmp:** پریدن به خط دیگری از کد (این دستور در اصل مشابه goto در زبان C عمل می‌کند)
- **دستورات jz و jnz:** این دو دستور، پرش «شرطی» را انجام می‌دهند، به این شکل که اگر نتیجه‌ی عملیات قبلی انجام شده، به ترتیب «صفر» و «ناصفر» باشد، به خط مشخص شده پرش کرده و در غیر این صورت به اجرا به روند سابق ادامه می‌دهند.

علاوه بر دستورات فوق، تعدادی دستور دیگر نیز وجود دارند که با لیست کامل‌تر آن‌ها در درس ساختار و زبان کامپیوتر آشنا خواهید شد؛ اما آن‌چه قصد بیان آن را داشتیم، آن است که این دستورات، بسیار ساده و ابتدایی هستند و حتی if های ساده در زبان C نیز باید به چندین دستور زبان ماشین ترجمه شوند. لازم است بدانید که دلیل ابتدایی بودن این دستورات، تنبلی طراحان این زبان نبوده، بلکه محدودیت‌های اجرای کد در سخت‌افزار رایانه‌ها که در دروس آینده با آن‌ها آشنا خواهید شد، سبب این سادگی دستورات شده است. به عنوان مثال، کد ساده‌ی زیر را در C در نظر بگیرید:

```
if (x != y) goto X;
else goto Y;
```

تبدیل این کد به اسمبلی، نتیجه‌ای تقریبا شبیه به کد زیر خواهد داشت:

```
sub z, x, y
jnz X
jmp Y
```

این کد، ابتدا x را منهای y کرده و اگر نتیجه صفر نبود (یعنی نابرابر بودند)، به X وگرنه به Y می‌پرد.

قصد از بیان مقدمه‌ی کوتاه فوق بر اسمبلی آن بود که بدانیم هر دستوری که به سادگی در زبان C می‌نویسیم، ممکن است ساختاری بسیار پیچیده‌تر از انتظار در زبان ماشین داشته باشد. حتی یک حلقه‌ی ساده در زبان C نیز باید به مجموعه‌ای از کنترل‌های شرطی و پرش‌ها (مانند روشی که حلقه‌ها را در فلوچارت‌ها رسم می‌کردیم) تبدیل شود تا قابل اجرا توسط سخت‌افزار رایانه باشد.

یکی از ویژگی‌های مهم این زبان آن است که مفهومی به نام data type در آن وجود ندارد و به همه‌ی «متغیر»ها، به یک چشم نگاه می‌شود و این برنامه‌نویس است که با خود قرارداد می‌کند آیا در یک خانه‌ی حافظه، عدد صحیح نوشته شده، عدد اعشاری نوشته شده، و یا کاراکتر نوشته شده است. زبان اسمبلی و سخت‌افزار رایانه هیچ‌گونه کنترل روی نوع یک خانه‌ی حافظه انجام نمی‌دهند. هم‌چنین در چنین زبانی، مفاهیم پیچیده‌تری مانند آرایه بسیار دورتر از انتظار خواهند بود.

حال شرایطی را تصور کنید که حافظه‌ای از سیستم درخواست می‌شود (در دستوراتی در اسمبلی که معادل malloc در زبان C هستند). رایانه بخشی از حافظه RAM را رزرو می‌کند و باید به نحوی آن را به برنامه اطلاع‌رسانی کند. در روشی که رایانه این کار را انجام می‌دهد، «آدرس شروع» قسمتی از حافظه که رزرو شده است، به برنامه اطلاع‌رسانی می‌شود. طبیعتا برنامه‌ی اسمبلی باید این آدرس را در جایی (که خود آدرس دیگری از حافظه است) ذخیره کند. در این مکان است که به شکل اجباری و بدون آن که بدانیم، مفهوم pointer پدید می‌آید؛ **خانه‌ای از حافظه که در آن، آدرس خانه‌ی دیگری از حافظه نوشته شده است.** بدون وجود این مفهوم، طبیعتا امـــــکان استفاده از امکاناتی مانند حافظه‌ی پویا (dynamic) که با دستورات malloc و مشابه آن دریافت می‌شود، وجود ندارد؛ اما برنامه‌نویس اسمبلی، این خانه از حافظه که آدرس خانه‌ی دیگری در آن واقع است را به شکل ویژه‌ای متفاوت با بقیه‌ی خانه‌ها نمی‌بیند؛ این تنها قرارداد برنامه‌نویس است که خانه‌ای شامل «محتوای مستقیم» است، و یا «آدرس خانه‌ای دیگر» در آن است.

ممکن است به ظاهر این مفهوم بسیار ساده‌تر از آن‌چه در زبان C پیاده‌سازی شده است به نظر برسد، زیرا کنترل آن‌که در هر خانه از حافظه چه نوشته شود، به طور کامل در دست برنامه‌نویس خواهد بود و به نظر آزادی عمل کامل در برنامه‌نویسی وجود دارد. اما این آزادی عمل فراوان در زبان اسمبلی، عیب بزرگی نیز به همراه دارد؛ وقتی کنترل و اجبار از دست رایانه خارج باشد و به انسان سپرده شود، همواره باید انتظار اشتباه شدن و بی‌دقتی را نیز داشته باشیم. ممکن است برنامه‌نویس در اثر بی‌دقتی فراموش کند که خانه‌ی خاصی از حافظه شامل چه نوع داده‌ای است، یا در پروژه‌های تیمی به غیر از عضوی از تیم که بخش خاصی از کد را نوشته، باقی اعضای تیم ندانند که هر متغیر در کد نوشته شده توسط عضو اول تیم، از چه نوعی است، و بعدا در استفاده از آن‌ها دچار اشتباه شوند. به دلیل ساختار بسیار ساده‌ی زبان اسمبلی و یکسان دیده شدن تمامی خانه‌های حافظه در آن، خطایابی (debug) در این زبان بسیار دشوارتر از زبان‌هایی مانند C است و از این رو، این خطاها تا مدت زیادی پنهان باقی مانده و ممکن است در مواقعی خاص و بحرانی در آینده ظاهر شوند. این در حالی است که با توجه به اهمیت پایداری (reliability) بسیار بالای انواعی از سیستم‌ها (مانند سیستم‌های پزشکی یا شاتل‌های فضایی)، لازم است احتمال داشتن خطا در کد، بسیار ناچیز باشد.

زبان C با علم به این حقیقت که نمی‌توان نیاز به pointer ها را انکار کرد، سعی کرده تا با ارائه‌ی data type ای جدا برای آن‌ها، امکان اشتباه برنامه‌نویس را کاهش دهد. به همین دلیل، اگر به اشتباه از خانه‌ای که برای نگه‌داری آدرس خانه‌ای دیگر از حافظه در نظر گرفته شده، برای نگه‌داری مقادیر عادی (و یا برعکس)

استفاده کنید، کد به دلیل بروز syntax error کامپایل نخواهد شد. با آن که ممکن است بروز syntax error های متعدد، اعصاب خردکن به نظر برسد، در حقیقت این خطاها جلوی بروز runtime error ها در آینده را می‌گیرند و به نوعی نقش پیش‌گیرانه در هشدار به برنامه‌نویس را ایفا می‌کنند.

اگر با زبان‌های برنامه‌نویسی دیگر (مانند Python) کار کرده باشید، ممکن است این سوال برایتان پیش آمده باشد که اگر وجود «آدرس به خانه‌ای از حافظه» ضروری است، چرا این زبان‌ها مفهومی به نام pointer ندارند؟ در پاسخ باید گفت که در اصل همه‌ی زبان‌ها در پشت صحنه از آدرس به خانه‌های حافظه استفاده می‌کنند، اما برای سهولت کار برنامه‌نویسان، اصلا این مفهوم را در اختیار برنامه‌نویس به طور مستقیم قرار نمی‌دهند. در اصل، ویژگی‌های پیاده‌سازی شده به شکل داخلی در این زبان‌ها، برای کارکرد خود در پشت صحنه، به pointer ها نیازمندند، اما به عنوان یک برنامه‌نویس، آن‌ها را به شکل مستقیم حس نمی‌کنید. به دلیل استفاده شدن زبان C برای کاربردهای بسیار سطح پایین‌تر که اصلا پیاده‌سازی آن‌ها با سایر زبان‌ها ممکن نیست و یا بسیار دشوار است (مانند ساخت سیستم‌عامل)، نیاز است که مفهوم pointer در این زبان وجود داشته باشد و نمی‌توان آن را به طور کلی از دید برنامه‌نویس پنهان کرد، اما در اکثر زبان‌های دیگر، به دلیل کاربرد متفاوت‌شان، این مفهوم پیچیده از دید برنامه‌نویسان پنهان شده است.

به طور خلاصه می‌توان گفت نیاز به نگه‌داری آدرس‌های سایر متغیرها در حافظه، نیازی ضروری در نوشتن برنامه‌های سیستمی و سطح پایین است، اما در صورت استفاده مستقیم از اسمبلی برای کار با آدرس‌های حافظه ممکن است دچار خطاهای گوناگونی شویم که به سادگی نیز نتوان منشا آن‌ها را پیدا کرد. برای کاستن از شدت این مشکل، زبان C نوع داده‌ای به نام pointer تعریف کرده و در صورت استفاده از یک خانه‌ی درنظر گرفته شده برای آدرس (pointer) به عنوان خانه‌ی حاوی مقدار عادی (و برعکس)، در هنگام کامپایل خطا صادر کرده و جلوی کامپایل شدن کد را می‌گیرد تا از خطاهای runtime احتمالی در آینده و در موقعیت‌های حساس، جلوگیری شود.

Pointers be like:

LEARNING TO SEE!

امیرحسین رازلیقی

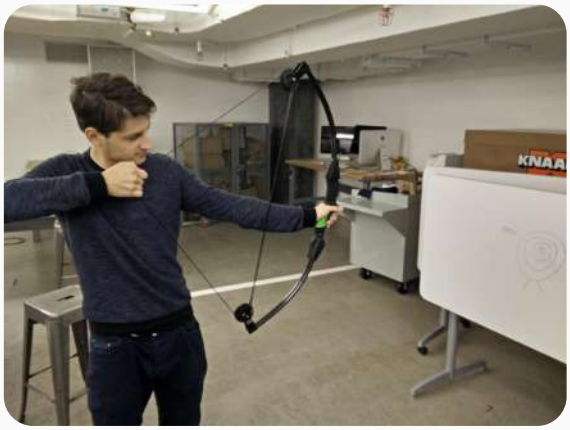
فرض کنید چشمانتان بسته است. پس از مدتی، چشم‌هایتان را باز می‌کنید و می‌بینید در یک محیط ناشناخته و جدید هستید (مثلا در یک جنگل بکرا!). ذهن شما بلافاصله مشغول انجام کارهایی می‌شود که شاید تا به حال اصلا به آنها توجه نکرده‌اید! در لحظه، محیطی که توسط چشمان شما قابل رویت است یا همان زاویه‌ی دید شما (Field of view)، تصویرهایی را که دریافت می‌کند، فریم به فریم (Frame by Frame) برای تحلیل به مغز ارسال می‌کند. اینجاست که پردازش تصویر در لحظه (Real time) اتفاق می‌افتد! شما از همان چند فریم ابتدایی (پس از اینکه چشمانتان را باز کردید) که دیده‌اید، داده‌های بسیار پیچیده‌ای کسب می‌کنید. ابتدایی ترین آنها رنگ و شکل اجسامی است که در نزدیکی شما هستند. در عین حال، تخمینی از دوری و نزدیکی درختان و سنگ‌ها و اجسام مختلف نسبت به خودتان دارید. شاید نه خیلی دقیق (که بگویید فلان مانع یا درخت دقیقا در فاصله‌ی ۵متری شما قرار دارد) اما تخمین خوبی از اوضاع می‌زنید (مثلا تشخیص می‌دهید یک جسم دیگر در پشت درخت قرار دارد و بخشی از آن معلوم است. یا مثلا یک سنگ در این جنگل نسبت به درختی که روبروی شماست، از شما دورتر است). نکته‌ی جالب‌تر اینکه اگر حالا چشمانتان را ببندید، همه چیز متفاوت است. شما با همان چند فریم کوتاهی که از جنگل مقابلتان دیدید، در ذهنتان یک تجسم از کل آن فضا و موقعیت خودتان در آن دارید و می‌توانید (با احتیاط و آهسته) در این فضا با چشمان بسته حرکت کنید. تمام این‌ها با چند فریم ساده از یک زاویه‌ی بسته و مشخص اتفاق افتاد. شما حتی در این جنگل قدم هم نزدید که اجسام را از زوایای مختلف ببینید. قدرت مغز انسان در تحلیل تصاویر و پیش‌بینی اطلاعات از روی آنها، به همین میزان حیرت انگیز است!

حالا فرض کنید می‌خواهیم taskهای این‌چنینی را به کامپیوتر یاد بدهیم. به‌طور خلاصه می‌خواهیم به کامپیوتر یاد بدهیم که "چطور ببیند"! این فرآیند یادگیری، روش‌های متنوع و جذابی دارد که از سال‌ها پیش بنیان گذاشته شدند و با اوج گرفتن دانش‌های دیگر (مثل Machine learning) به گستردگی آنها افزوده شد!

بیایید با یکی از معروف‌ترین محصولاتی که هر کدام از ما اگر با آن کار نکرده باشیم، لااقل اسم آن را شنیده‌ایم، شروع کنیم. Xbox Kinect :

همانطور که می‌دانید، Kinect یکی از محبوب‌ترین محصولات انقلابی مایکروسافت در ابتدای معرفی سری xbox ها بود. این دستگاه کوچک، نوع جدیدی از Gaming را برای کاربران به ارمغان می‌آورد که شما می توانستید جلوی دستگاهتان بایستید و با بدن خودتان، شروع به بازی کنید (اسکی کنید، تیراندازی کنید و ...). این دستگاه یکی از محصولات انقلابی‌ای بود که Computer vision را به نوعی در دسترس عموم قرار داد. این دستگاه کوچک با استفاده از دوربینی که داشت و سنسورهای مختلف خود، برای ساختن یک تجربه‌ی بازی بی‌نقص، باید حرکات شما را به صورت Real Time تشخیص می‌داد. همچنین باید تشخیص می‌داد که چه اعضایی از بدن شما در حال حرکت و انجام بازی هستند. مثلا در بازی‌ای مثل تیراندازی با کمان، باید به‌خوبی کشیده شدن عضلات بازوی شما و میزان آن را شناسایی می‌کرد تا بفهمد با

چه قدرتی در حال پرتاب تیر هستید. یا زاویه‌ای که کمان را با آن نگه‌داشته اید چگونه است و چند درجه است و آیا تیر شما به هدف می‌خورد یا خیر. البته که این دستگاه برای تسهیل فهمیدن این امور (مثل فهمیدن انواع action های مختلف کابر) از سنسورهایی هم در کنار دوربین خود استفاده می‌کرد.



وجود این دستگاه و کاربرد بسیار آن در صنعت بازی، سیر تحول عظیمی در computer vision و تحقیقات مرتبط به آن داشت. مایکروسافت همواره یکی از پیشتازان این عرصه بوده و هست اما kinect باعث شد تا کاربردهایی از vision که قبل‌تر خیلی به دید صنعتی به آنها نگاه نمی‌شد، تغییر کنند. به طور مثال تحقیقات زیادی روی تشخیص حرکت(motion detection) افراد در تصویر ورودی از یک وبکم انجام شد. یا تحقیقات جدی‌ای شکل گرفت که چطور بدون استفاده از سنسورها و فقط با استفاده از یک دوربین ساده، بتوانیم عمق اجسام را تخمین بزنیم (Depth Estimation) تا بفهمیم یک شی نسبت به شی دیگر به دوربین دورتر است یا نزدیک تر است.

در ادامه‌ی این سفر پر فراز و نشیب‌مان، اولین ایده‌ی جدی و صنعتی خودروی خودران را بررسی می‌کنیم که گوگل پیشگام آن شد و Uber هم به رقابت با آن پرداخت!

حوزه‌ی خودروهای خودران، رویایی است که مهندسین سال‌های سال است آن را دنبال می‌کنند و هنوز هم به موفقیت ۱۰۰ درصدی در آن نرسیده‌اند (Fully self driving cars)؛ اما به لطف شرکت‌هایی مثل Tesla، به‌شدت به آن نزدیک شده‌اند! تصور کنید شما به‌عنوان یک Research Engineer در تیم Self Driving car گوگل استخدام شده‌اید و قرار است لیستی از نیازمندی‌ها را تهیه کنید. برای اینکه یک ماشین به تنهایی و بدون هیچ کمکی از انسان، بتواند رانندگی کند، نیاز است چه چیزهایی را بداند؟ نحوه‌ی رانندگی و عوض کردن دنده و ... نیازهای بسیار ابتدایی و قابل دسترسی هستند. عمیق‌تر فکر کنید! فرض کنید سر یک چهارراه ایستاده‌اید و چراغ سبز می‌شود . ماشین باید راه بیفتد. ناگهان یک ماشین که خلاف می‌آید وارد چهارراه می‌شود. ماشین ما باید در لحظه، تصاویری که می‌بیند (ماشینی که در حال خلاف کردن است) را دریافت کند و خیلی زود آن را تحلیل کند و تصمیم درست را بگیرد. باید ترمز کند؟ یا باید لاینش را عوض کند؟ یا در همین لاین به مسیرش ادامه دهد؟ یا ...

پس یکی از مهمترین مسائل در این محصول، تصمیم‌گیری سریع بر اساس داده‌هایی است که در لحظه داریم. مثلا فرض کنید در یک جاده در حال حرکت هستید. ناگهان یک مانع (مثلا یک حیوان) در جاده دیده می‌شود. ماشین ما باید در لحظه چندین و چند کار را انجام دهد. فاصله‌ی ماشین را تا آن مانع تخمین بزند. بررسی کند در لاین مخالف ماشینی می‌آید یا نه. بررسی کند که با توجه به فاصله تا مانع، اگر الان ترمز کند موثر است یا قطعا به تصادف ختم می‌شود؟ اگر به تصادف ختم می‌شود، آیا می‌تواند به جای ترمز کردن، برای لحظه‌ای کوتاه به لاین مخالف برود و آنجا سرعت خود را کم کند تا با مانع برخورد نکند و پس از رد کردن مانع به لاین خود برگردد؟

همه‌ی این موارد، که اکثر آنها از جنس "تصمیم‌گیری سریع بنابر شواهدی که در لحظه دیده می‌شود" است، چالش‌هایی است که باعث شده حل آنها و رسیدن به یک Fully self driving car اینقدر

طولانی شود! البته که شرکت Tesla بسیار به این امر نزدیک شده و می‌تواند تا حد خوبی ماشین را بدون نیاز به کمک شما هدایت کند. اما همچنان نیاز دارد تا یک نفر با گواهینامه‌ی مورد تایید، پشت فرمان باشد و مراقب باشد که در صورتیکه اتفاق غیر منتظره‌ای افتاد یا جایی سیستم اشتباه کرد، فرمان را بدست بگیرد. یکی از جذاب‌ترین ویژگی‌های ماشین‌های تسلا این است که موقعیت ماشین را به نسبت بقیه‌ی ماشین‌ها و اجسام، visualize می‌کند و به شما در صفحه‌نمایش ماشین نشان می‌دهد!

درنهایت این سفر پرفراز و نشیب، سری به محصول انقلابی دیگر مایکروسافت می‌زنیم که بار دیگر، یک کاربرد خلاقانه و جذاب از computer vision را به نمایش گذاشت. محصولی به نام HoloLens: این محصول (که گوگل با محصول Google Glass خود تلاش کرد با آن رقابت کند) یک هدست واقعیت مجازی/ واقعیت افزوده است (AR/VR). با زدن این عینک، به دنیای نام‌آشنای این روزها سفر می‌کنید. دنیای MetaVerse! شما با زدن این عینک در حالت واقعیت مجازی، به یک دنیای کاملا گرافیکی (مثل بازی‌هایی که تاکنون تجربه کرده‌اید) سفر می‌کنید و با یک آواتار شخصی می‌توانید در این دنیا گشت و گذار کنید و حرکت کنید. تا به اینجای کار که تماما computer graphics و انواع سنسورهای مختلف کار را پیش می بردند و خبری از vision نبود! اما vision لحظه‌ای به این دنیا می‌آید که شما به‌جای virtual reality، از این هدست در حالت augmented reality استفاده

می‌کنید! یعنی مثل یک عینک، شما دارید دنیای عادی و روزمره‌ی اطراف خودتان را می‌بینید (نه یک دنیای گرافیکی جدید) اما در دنیایتان تغییرات زیادی ایجاد شده! مثلا می‌توانید sms های خودتان را روی میز صبحانه‌تان ببینید و با حرکات دست، آنها را scroll کنید. یا مثلا با google map می‌خواهید به سمت محل کارتان بروید. این بار به‌جای نگاه به صفحه‌ی گوشی و فهمیدن مسیر، مسیر جلوی شما با علایمی مشخص می‌شود و به شما راهنمایی می‌کند که باید از کدام سمت بروید. در واقع با این هدست (یا عینک)، هر کس یک نمونه‌ی خاص از دنیای کنونی را تجربه می‌کند که برای او شخصی سازی شده است!

حال به این فکر کنید که این عینک‌ها، چه چالش‌هایی دارند و computer vision چگونه به حل آنها کمک می‌کند؟ این بار خودتان را جای مهندسی در مایکروسافت بگذارید و سعی کنید لیستی از نیازمندی‌ها در بیاورید!

سفر ما اینجا به پایان می‌رسد اما هنوز یک دنیا محصول و اتفاق جذاب در حوزه‌ی computer vision و انواع کاربردهای تحقیقاتی و صنعتی آن انتظار شما را می‌کشد! هیچ چیز مانع کنجکاوی شما نیست که در این موارد جستجو کنید و اطلاعات بیشتری کسب کنید. از کجا معلوم؟ شاید در محصول بعدی انقلابی این حوزه، شما واقعا جای یکی از مهندسین و محققین خبره‌ی آن تیم باشید!

لیستی از لینک‌های پیشنهادی برای کنجکاوی بیشتر:

https://www.v7labs.com/blog/computer-vision-applications

https://medium.com/trapica/top-10-examples-of-computer-vision-and-augmented-reality-f2d47e18c707

https://www.aimprosoft.com/blog/computer-vision-in-healthcare/

https://www.superannotate.com/blog/computer-vision-robotics

“The journey is what bring us happiness not destination”

-Dan Millman

کدنامه

شماره: سوم

تاریخ انتشار: ۷ بهمن ۱۴۰۱

نویسندگان: آرش یادگاری، امیرحسین رازلیقی، امیرعلی سلیمی، طاها اکبری، محمد بروغنی، سید پارسا نشایی، سید محمد حسین حسینی

طراحی: محمد مصیبی

دستیاران آموزشی دروس مبانی برنامه‌سازی و برنامه‌سازی پیشرفته دانشکده مهندسی کامپیوتر،مجموعه مطالب آموزشی‌ای برای درک بهتر درس و همین‌طور علاقه‌مند شدن بیشتر به مباحث، در اختیار دانشجویان قرار می‌دادند. پس از مدتی، تصمیم بر آن شد که این مجموعه مطالب، در قالب یک نشریه اختصاصی، در قالبی مشخص در اختیار دانشجویان قرار بگیرد تا هم منبعی برای تکمیل دانسته‌های دانشجویان از این دو درس باشد و هم راهی برای شناخت هر چه بیشتر دنیای وسیع مهندسی کامپیوتر!