

به نام خدا

## زبان‌های برنامه‌نویسی

آرش شفیعی



## متغیرها و نوع‌های داده‌ای

- در این فصل در مورد نام‌ها و متغیرها و کلمات کلیدی در زبان‌های برنامه نویسی صحبت خواهیم کرد.
- سپس در مورد انواع متغیرها و حوزه‌های تعریف صحبت می‌کنیم.

- یک متغیر در واقع یک مفهوم انتزاعی<sup>1</sup> برای تعدادی از خانه‌های حافظه است. در برخی مواقع متغیر دقیقاً همان مقداری را دارد که در خانه حافظه قرار می‌گیرد، مانند یک متغیر عدد صحیح و گاهی مقادیر ذخیره شده در متغیر باید به نحوی در حافظه نگاشت شوند چرا که مقدار متغیر با آنچه در حافظه ذخیره می‌شود متفاوت است، مانند یک آرایه از کاراکترها.
- از مهم‌ترین ویژگی‌های یک متغیر می‌توان به حوزه تعریف<sup>2</sup> و طول عمر<sup>3</sup> آن اشاره کرد.

---

<sup>1</sup> abstraction

<sup>2</sup> scope

<sup>3</sup> lifetime

- یک نام<sup>1</sup> یا شناسه<sup>2</sup> رشته ای است که برای تمیز دادن یک موجودیت (مانند تابع یا متغیر) در یک برنامه به کار می‌رود.
- هر زبان محدودیتی بر روی نام‌ها دارد. معمولاً در بیشتر زبان نام‌ها باید با حرف شروع شوند و نام‌ها حساس<sup>3</sup> به بزرگ و کوچک‌اند.
- معمولاً کلمات کلیدی زبان را نمی‌توان به عنوان نام انتخاب کرد. زبان کوپول تعداد کلمات کلیدی زیادی دارد مانند LENGTH و COUNT که منجر به محدودیت برای برنامه نویسی می‌شود.
- در زبان سی ++، برای نام‌ها می‌توان فضای نام<sup>4</sup> تعریف کرد که باعث می‌شود یک نام را بتوان در چند مکان متفاوت به کار برد.

---

<sup>1</sup> name

<sup>2</sup> identifier

<sup>3</sup> case sensitive

<sup>4</sup> name space

- یک متغیر را می‌توان با چند ویژگی مشخص کرد : نام، آدرس، مقدار، نوع، طول عمر و حوزه تعریف.
- آدرس یک متغیر در واقع آدرس حافظه‌ای در ماشین است که به آن متغیر اختصاص داده شده است. در طول اجرای یک برنامه یک متغیر ممکن است چندین بار تخصیص داده شده و آزاد شود و در هر بار تخصیص، آدرس آن می‌تواند متفاوت باشد. مثلاً متغیر محلی در یک تابع در هر بار فراخوانی یک آدرس جدید می‌گیرد.

- آدرس متغیر گاهی مقدار چپ<sup>1</sup> نامیده می‌شود چون وقتی متغیر در سمت چپ یک عبارت باشد به آدرس آن نیاز داریم.
- ممکن است چند متغیر یک آدرس واحد داشته باشند که در اینصورت به اسامی دیگری که به یک آدرس واحد اشاره می‌کنند نام مستعار<sup>2</sup> گفته می‌شود. در زبان سی++، با استفاده از اشاره‌گرها و متغیرهای ارجاعی می‌توان نام مستعار تعریف کرد.
- یک متغیر همچنین دارای یک نوع<sup>3</sup> است که محدوده مقادیری که در آن می‌توانند قرار بگیرند و عملگرهایی که بر روی آن متغیر می‌توانند اعمال شوند را تعیین می‌کند.
- مقدار یک متغیر محتوایی است که در آن آدرس حافظه متناظر با آن ذخیره شده است. مقدار یک متغیر را گاهی مقدار راست<sup>4</sup> می‌نامیم زیرا هرگاه متغیر در سمت راست قرار گیرد به مقدار آن نیاز داریم.

<sup>1</sup> l-value

<sup>2</sup> alias

<sup>3</sup> type

<sup>4</sup> r-value

- انقیاد<sup>1</sup> به معنی پیوند دادن یک موجودیت با ویژگی آن است. برای مثال پیوند دادن نوع یک متغیر با آن متغیر انقیاد نوع، و انتساب مقدار به یک متغیر انقیاد مقدار نامیده می‌شود.
- زمانی را که انقیاد صورت می‌گیرد، زمان انقیاد<sup>2</sup> می‌گویند.
- برای مثال نماد ستاره (\*) در زمان طراحی زبان<sup>3</sup> به عمل ضرب مقید شده است. نوع یک متغیر به مقادیر ممکن آن زمان پیاده سازی زبان<sup>4</sup> مقید شده است. یک متغیر به نوع آن در زبان جاوا در زمان کامپایل<sup>5</sup> مقید شده است. و در نهایت یک متغیر به سلول حافظه در زبان جاوا در زمان اجرا<sup>6</sup> مقید می‌شود.

---

<sup>1</sup> binding

<sup>2</sup> binding time

<sup>3</sup> language design time

<sup>4</sup> language implementation time

<sup>5</sup> compile time

<sup>6</sup> run time or execution time



– یک انقیاد ایستا<sup>1</sup> نامیده می‌شود اگر قبل از اجرای برنامه رخ دهد و در زمان اجرای برنامه بدون تغییر باقی بماند. اما اگر انقیاد در زمان اجرا صورت بگیرد و قابل تغییر باشد، به آن انقیاد پویا<sup>2</sup> می‌گوییم.

---

<sup>1</sup> static binding

<sup>2</sup> dynamic binding

- انقیاد نوع نیز می‌تواند ایستا یا پویا باشد. انقیاد نوع در جاوا ایستا و در پایتون پویا است.
- متغیرها همچنین می‌توانند به دو صورت تعریف شوند: صریح<sup>1</sup> و ضمنی<sup>2</sup>.
- در تعریف صریح، نوع متغیر تعیین می‌شود، اما در تعریف ضمنی نوع متغیر در اولین مقداردهی به متغیر تعیین می‌شود.
- در بیشتر زبان‌های قدیمی مانند جاوا انقیاد نوع ایستا و تعریف متغیر به صورت صریح است.

---

<sup>1</sup> explicit declaration

<sup>2</sup> implicit declaration

- در برخی از زبان‌ها نام متغیر تعیین کننده نوع آن نیز هست. مثلا در زبان پرل متغیرهای عددی<sup>1</sup> با علامت \$ و آرایه‌ها<sup>2</sup> با @ آغاز می‌شوند.
  - تعریف ضمنی می‌تواند برای انقیاد ایستا نیز صورت بگیرد. مثلا در زبان سی++ با استفاده از کلمه auto می‌توان یک متغیر را به طور ضمنی تعریف کرد :
- ```
auto a = 12 ; auto b = " blue" ;
```
- در اینجا a از نوع int و b از نوع string در زمان کامپایل به صورت انقیاد ایستا تعیین می‌شود.

---

<sup>1</sup> scalar

<sup>2</sup> arrays

- در انقیاد نوع پویا، نوع متغیر در زمان اجرا با اولین انتساب مقدار به آن تعیین می‌شود. در چنین مواردی انقیاد متغیر به مکان حافظه نیز باید به صورت پویا باشد چون هر نوع مقدار حافظه متفاوتی اشغال می‌کند. همچنین در انقیاد پویا، نوع متغیر نیز در زمان اجرا می‌تواند تغییر کند.
- انقیاد پویا باعث می‌شود برنامه انعطاف بیشتری داشته باشد. برای مثال فرض کنید برنامه‌ای توسط پایتون نوشته ایم که مقادیری را از ورودی می‌خواند و محاسباتی را انجام می‌دهد. بسته به نوع مقادیر ورودی نوع متغیرها در این برنامه می‌توانند تغییر کنند، اما در یک برنامه سی اگر نوع متغیرها صحیح تعریف شده باشند نمی‌توان ورودی اعشاری به برنامه داد.

- در زبان لیسپ که یک برنامه تابعی قدیمی است انقیاد پویا است اما در بیشتر زبان‌های قدیمی انقیاد به صورت ایستا بوده است. در زبان‌های پایتون، روبی، جاوا اسکریپت و پی‌اچ‌پی انقیاد به صورت پویا است. برای مثال در پایتون می‌توانیم بنویسیم :

---

```
۱ a = [12.2 , 19.5]
```

```
۲ a = "hello"
```

---

- بدین صورت a ابتدا از نوع لیست تعریف می‌شود و سپس نوع آن به رشته تغییر پیدا می‌کند.
- در زبان روبی که یک زبان شیء‌گرای خالص است، همه متغیرها ارجاعی و از نوع شیء عمومی هستند و به کلاس‌های مختلف ارجاع می‌دهند.

- سه نقطه ضعف برای انقیاد پویا وجود دارد :

۱. خطاهای نوع نمی‌توانند در زمان کامپایل تعیین شوند بنابراین برنامه‌ها در زبان‌ها با انقیاد پویا کمتر قابل اعتماد هستند.

۲. ممکن است به خاطر خطای برنامه نویس نوع متغیر تغییر کند در حالی که برنامه نویس انتظار نداشته است نوع تغییر کند.

۳. انقیاد پویا هزینه زمانی دارد، زیرا بررسی نوع در زمان اجرا باید صورت بگیرد که باعث کندی برنامه می‌شود.

- معمولا زبان‌های با انقیاد پویا توسط مفسر یا به صورت ترکیبی پیاده سازی می‌شوند. یک کامپایلر نمی‌تواند کدی را ترجمه کند که در آن نوع‌ها نامشخص هستند.

- فضایی در حافظه که به یک متغیر مقید می‌شود از مخزنی از حافظه‌های موجود گرفته می‌شود. این فرایند را تخصیص حافظه <sup>1</sup> می‌نامیم و فرایند آزاد سازی حافظه <sup>2</sup> فرایند گرفتن سلول حافظه از متغیر و بازگرداندن آن به مخزن فضاها می‌شود.
- طول عمر <sup>3</sup> یک متغیر مدت زمانی است که در آن متغیر به فضای حافظه مقید شده است.

---

<sup>1</sup> memory allocation

<sup>2</sup> memory deallocation

<sup>3</sup> lifetime

– متغیرها را از لحاظ طول عمر به چهار دسته می‌توان تقسیم کرد: (۱) ایستا<sup>۱</sup>، (۲) پویا در پشته<sup>۲</sup>، (۳) پویا در هیپ به طور صریح<sup>۳</sup>، (۴) پویا در هیپ به طور ضمنی<sup>۴</sup>

---

<sup>۱</sup> static

<sup>۲</sup> stack-dynamic

<sup>۳</sup> explicit heap-dynamic

<sup>۴</sup> implicit heap-dynamic



- یک متغیر ایستا<sup>1</sup> متغیری است که به سلول حافظه قبل از اجرای برنامه مقید شده باشد و تا وقتی که برنامه خاتمه یابد به همان سلول حافظه مقید بماند.
- یکی از کاربردهای متغیر ایستا زمانی است که بخواهیم متغیری در یک تابع یا کلاس مقدار خود را پس از خروج از تابع یا پس از تخریب اشیا از دست ندهد. این متغیرها در قسمت داده‌ها<sup>2</sup> در حافظه تخصیص داده می‌شوند.
- یکی از مزیت‌های استفاده از متغیر ایستا سرعت دسترسی به آن است. در زبان سی++ برای متغیرهای ایستا از کلمه کلیدی static استفاده می‌کنیم.
- وقتی متغیر ایستا در یک کلاس جاوا یا سی++ تعریف شود، آن متغیر متعلق به کلاس است نه اشیا آن کلاس.

---

<sup>1</sup> static variable

<sup>2</sup> data segment

- یک متغیر پویا در پشته<sup>1</sup> متغیری است که انقیاد حافظه آن در زمان اجرا در لحظه تعریف متغیر رخ می‌دهد.
- این متغیرها در فضای پشته<sup>2</sup> برنامه تخصیص داده می‌شوند.
- یکی از مزیت‌های استفاده از متغیرهای پویا در پشته، استفاده از آنها در توابع بازگشتی است. در هر فراخوانی یک تابع بازگشتی، همه متغیرهای تابع در فضای پشته کپی می‌شوند. مزیت دیگر این متغیرها این است که هر تابع یا کلاس، متغیرهای خود را در فضای پشته خود تعریف می‌کند که باعث امنیت بیشتر برنامه می‌شود.
- سرعت تخصیص فضا برای متغیرهای پویا نسبت به متغیرهای ایستا کمتر است.
- در زبان جاوا متغیرهای نوع اصلی و در سی++ همه متغیرها به طور پیش فرض متغیر پویا هستند که در پشته تعریف می‌شوند.

---

<sup>1</sup> stack-dynamic variable

<sup>2</sup> stack

- یک متغیر صریح پویا در هیپ<sup>1</sup> خانه حافظه بدون نام است که فضای حافظه در آن به طور صریح توسط برنامه نویس تخصیص داده شده و آزاد می‌شود. این متغیر در فضایی در حافظه به نام هیپ<sup>2</sup> قرار می‌گیرند، که تنها توسط اشاره‌گر و متغیر ارجاعی قابل دسترسی هستند.
- فضای هیپ فضایی در حافظه است که ساختار آن کاملاً نامنظم است به دلیل اینکه فضای بیشتری را در حافظه اشغال می‌کند.
- در زبان سی++، این متغیرها توسط عملگر new تخصیص و توسط عملگر delete آزاد می‌شوند.

---

<sup>1</sup> explicit heap-dynamic variable

<sup>2</sup> heap

- وقتی فضای هیپ اختصاص داده می‌شود، آدرس آن بازگردانده می‌شود. زمان انقیاد حافظه این متغیرها در هنگام اجرا و در زمان تخصیص فضای حافظه است.
- در زبان جاوا همه متغیرها به جز متغیرهای اصلی، شیء هستند. همه اشیا در جاوا متغیر پویا هستند که در هیپ تخصیص داده می‌شوند و توسط متغیر ارجاعی آنها قابل دسترسی هستند. در جاوا راهی برای آزاد سازی حافظه توسط برنامه نویس وجود ندارد، بلکه فضاها به طور خودکار آزادسازی می‌شوند.
- معمولا ساختارهای داده مثل لیست‌های پیوندی که به فضای زیادی نیاز دارند و مقدار حافظه مورد نیاز آنها از ابتدا نامعلوم است از متغیرهای پویا در هیپ استفاده می‌کنند.
- نقطه ضعف این متغیرها هزینه زمانی برای تخصیص حافظه و همچنین سختی آنها در مدیریت اشاره‌گرها و متغیرهای ارجاعی است.

- یک متغیر ضمنی پویا در هیپ<sup>1</sup> متغیری است که به طور ضمنی بدون دخالت برنامه نویس در زمان مقدار دهی، به حافظه هیپ مقید می‌شود.
- اگر این متغیر قبلاً نیز در برنامه استفاده شده باشد، در هر مقدار دهی جدید مجدداً به یک فضا در حافظه هیپ مقید می‌شود.
- به طور مثال در زبان پایتون با تعریف `Var = [2,3]` فضایی در حافظه هیپ تخصیص داده می‌شود و چنانچه در همان برنامه مجدداً با دستور `Var = 'hello'` یا `Var = [1,2]` مواجه شویم، فضای جدیدی در حافظه تخصیص داده می‌شود.
- مزیت این متغیر انعطاف پذیری آن است. میزان حافظه ای که اشغال می‌کند می‌تواند به صورت پویا و ضمنی تغییر کند. همچنین برنامه‌نویس نیازی به تخصیص آن ندارد. نقطه ضعف این روش هزینه بالایی اجرا است.
- یک متغیر همچنین می‌تواند ثابت تعریف شود بدین معنی که مقدار آن در طول برنامه غیر قابل تغییر است. متغیرهای ثابت در سی++ با کلمه کلیدی `const` و در جاوا با `final` مشخص می‌شوند.

---

<sup>1</sup> implicit heap-dynamic variable

- حوزه تعریف<sup>1</sup> یک متغیر محدوده‌ای از دستورات است که برای آنها آن متغیر قابل مشاهده<sup>2</sup> است.
- یک متغیر برای یک دستور قابل مشاهده است اگر آن دستور بتواند متغیر را مقداردهی یا مقدارگیری کند.
- قسمتی از یک برنامه که با یک علامت شروع و پایان مشخص شده است را یک بلوک<sup>3</sup> می‌گوییم.
- یک متغیر را برای یک بلوک محلی<sup>4</sup> می‌نامیم، اگر در آن بلوک از کد تعریف شده باشد. یک متغیر را برای یک بلوک غیر محلی<sup>5</sup> می‌نامیم اگر متغیر در آن بلوک از برنامه تعریف نشده، ولی قابل مشاهده باشد.

---

<sup>1</sup> scope

<sup>2</sup> visible

<sup>3</sup> block

<sup>4</sup> local

<sup>5</sup> nonlocal

- حوزه تعریف به دو دسته ایستا و پویا تقسیم می شود.
- برای پیدا کردن مقدار متغیری که حوزه تعریف آن ایستا است، کامپایلر ابتدا در بلوک فعلی به دنبال متغیر می گردد. اگر متغیر یافت نشد، بلوک های پدر را برای پیدا کردن تعریف متغیر و مقدار آن جستجو می کند.
- اگر حوزه تعریف متغیری پویا باشد، مقدار متغیر بستگی به اجرا و پشته فراخوانی پیدا می کند. آخرین مقداری که در پشته فراخوانی به آن متغیر داده شده است، مقداری است که برای آن متغیر در نظر گرفته می شود.

- در مثال زیر اگر حوزه تعریف  $x$  ایستا باشد، آنگاه خروجی برنامه ۱۰ است، اما اگر حوزه تعریف پویا باشد، خروجی برنامه ۲۰ خواهد بود.

---

```
۱  int x = 10;
۲  int f() {
۳      return x;
۴  }
۵  int g() {
۶      int x = 20;
۷      return f();
۸  }
۹  int main() {
۱۰     printf("%d \n", g());
۱۱     return 0;
۱۲ }
```

---



- حوزه تعریف ایستا<sup>1</sup> اولین بار در زبان الگول معرفی شد و بیشتر زبان‌های دستوری<sup>2</sup> بعد از الگول این مفهوم را از الگول گرفته‌اند.
- دو دسته از زبان‌ها وجود دارند که در آنها حوزه تعریف ایستا به کار می‌رود: زبان‌هایی که در آنها می‌توان زیر برنامه‌های تو در تو نوشت و زبان‌هایی که زیر برنامه‌ها نمی‌توانند در آن تو در تو باشند.
- در دسته اول زبان‌های جاوا اسکریپت و لیسپ معمولی و آدا و پایتون قرار دارند و در دسته دوم زبان‌هایی به سبک سی.

---

<sup>1</sup> static scope

<sup>2</sup> imperative languages

- در زبان‌هایی با حوزه تعریف ایستا، وقتی با متغیری برخورد می‌کنیم ابتدا در همان بلوکی که متغیر استفاده شده به دنبال تعریف آن می‌گردیم. اگر متغیر در آن بلوک تعریف نشده بود یا به عبارت دیگر یک متغیر غیر محلی بوده آنگاه در بلوک پدر<sup>1</sup> به دنبال تعریف آن متغیر می‌گردیم و اگر در بلوک پدر متغیر تعریف نشده بود، در بلوک پدرپدر به دنبال آن می‌گردیم و این روند را ادامه می‌دهیم تا یا تعریف متغیر را بیابیم و یا خطای متغیر تعریف نشده را گزارش کنیم.

---

<sup>1</sup> parent block

- برنامه زیر را در زبان پایتون در نظر بگیرید :

```

۱ def func() :
۲     def sub1() :
۳         x = 7
۴         sub2()
۵     def sub2() :
۶         y = x
۷         print("y = ",y)
۸     x = 3
۹     sub1()
۱۰
۱۱ func()

```

- متغیر x در تابع sub2 تعریف نشده است بنابراین باید در بلوک پدر یعنی در تابع func به دنبال آن بگردیم. در این بلوک x برابر با ۳ قرار گرفته است. توجه کنید که تابع sub2 از متغیر x در تابع sub1 استفاده نمی‌کند زیرا این تابع پدر تابع sub2 نیست.

– برنامه زیر را در نظر بگیرید :

```
۱ void sub() {  
۲     int count;  
۳     ...  
۴     while (...) {  
۵         count++;  
۶         ...  
۷     }  
۸ }
```

– متغیر count در بلوک تابع sub تعریف شده است و بلوک حلقه از این متغیر استفاده می‌کند. در واقع بلوک حلقه از متغیری استفاده می‌کند که در بلوک پدر تعریف شده و غیرمحلی است.

- حال برنامه زیر را در نظر بگیرید :

```
۱ void sub() {  
۲     int count;  
۳     ...  
۴     while (...) {  
۵         int count;  
۶         count++;  
۷         ...  
۸     }  
۹ }
```

- متغیر count که در حلقه while تعریف شده است، در بلوک حلقه استفاده می‌شود، اما متغیر count که در بلوک تابع sub تعریف شده در حلقه قابل مشاهده نیست، زیرا متغیری همانام در حلقه تعریف شده است.

- زبان جاوا تعریف نام‌های تکراری در بلوک‌های تو در تو را ممنوع کرده است، چرا که قابلیت اطمینان برنامه با مجاز کردن آنها پایین می‌آید.
- در زبان سی می‌توان با بازکردن آکولاد یک بلوک فرزند ایجاد کرد ولی در زبان پایتون این قابلیت وجود ندارد. همچنین در زبان پایتون گرچه حوزه تعریف ایستا در توابع تو در تو استفاده می‌شود ولی در بلوک‌های تو در تو حوزه تعریف ایستا استفاده نمی‌شود.

- کد زیر در زبان پایتون را در نظر بگیرید :

```
۱ for i in range (2) :  
۲     x = 5  
۳     print(x)  
۴ print(x)
```

- گرچه متغیر x در بلوک for تعریف شده اما بیرون از بلوک نیز قابل مشاهده است.

- در زبان‌های تابعی حوزه تعریف یک متغیر با استفاده از کلمه `let` تعیین می‌شود. برای مثال در زبان ام‌ال می‌توان به صورت زیر چند متغیر تعریف کرده و از آن متغیرها در یک عبارت استفاده نمود :

---

```
۱ let
۲     val    top = a + b
۳     val    bottom = c - d
۴ in
۵     top / bottom
۶ end;
```

---



- در برخی از زبان‌ها وقتی متغیری در بلوکی تعریف می‌شود آن متغیر در همه بلوک قابل مشاهده است اما مقدار آن اگر بعد از استفاده تعریف شده باشد undefined است.
- برای مثال برنامه زیر در جاوا اسکریپت را در نظر بگیرید :

---

```
۱ console.log(x)
۲ var x = 10
```

---

- مقدار x برابر با undefined چاپ خواهد شد، ولی اگر x تعریف نشده باشد با پیام خطای مفسر رو به رو می‌شویم.

- برخی از زبان‌ها مانند سی و سی++ و پایتون اجازه می‌دهند که متغیرها در خارج از توابع و کلاس‌ها نیز تعریف شوند. این متغیرها را متغیرهای عمومی<sup>1</sup> می‌نامیم که توسط همه توابع قابل مشاهده‌اند.
- در زبان سی و سی++، می‌توان علاوه بر تعریف<sup>2</sup> متغیرها، آن‌ها را اعلام<sup>3</sup> نمود.
- در زمان کامپایل، اعلام متغیر توسط برنامه نویس به کامپایلر اعلام میکند که متغیری از نوع داده‌ای اعلام شده در کد وجود دارد، اما این متغیر ممکن است هنوز تعریف نشده باشد. اعلام متغیرها معمولاً وقتی به کار می‌رود که یک فایل دیگر تعریف شده باشد و بخواهیم از آن در یک فایل دیگر استفاده کنیم. همچنین اگر متغیری بعد از استفاده از آن تعریف شده باشد، باید قبل از استفاده آن را اعلام کنیم. در زمان اجرا، با تعریف متغیر فضای حافظه به آن تخصیص داده می‌شود، اما با اعلام متغیر تنها انقیاد نوع صورت می‌گیرد.
- در زبان سی، یک متغیر را می‌توان توسط کلمه کلیدی extern اعلام کرد.

---

<sup>1</sup> global variable

<sup>2</sup> define

<sup>3</sup> declare

- در زبان سی++ اگر یک متغیر عمومی و یک متغیر محلی هم نام باشیم، متغیر عمومی غیر قابل مشاهده است، اما می‌توان با استفاده از عملگر حوزه تعریف<sup>1</sup> (::) به آن دسترسی پیدا کرد.
- برای مثال برای دسترسی به متغیر عمومی X در تابعی که متغیر X را تعریف کرده از عبارت X:: استفاده می‌کنیم.
- در زبان پایتون، می‌توانیم از یک متغیر عمومی در یک تابع استفاده کنیم، اما اگر متغیر در تابع دوباره تعریف شود، آن متغیر تبدیل به یک متغیر محلی می‌شود و متغیر عمومی غیر قابل مشاهده می‌شود. برای اعلام متغیر به عنوان یک متغیر عمومی در یک تابع از کلمه global استفاده می‌کنیم.

---

<sup>1</sup> scope operator

- برنامه زیر را در نظر بگیرید :

```
۱ day = " Monday "  
۲ def today() :  
۳     print (" Today is ", day)  
۴ today()
```

- این برنامه بدون خطا اجرا می شود و خروجی آن برابر است با " Today is Monday ".

- حال برنامه زیر را در نظر بگیرید :

---

```
۱ day = " Monday "  
۲ def today() :  
۳     print (" Today is ", day)  
۴     day = " Tuesday"  
۵     print (" Tomorrow is ", day)  
۶ today()
```

---

- در اینجا با پیام خطا رو به رو می‌شویم چرا که day به یک متغیر محلی در تابع today() تبدیل شده است و اولین دسترسی به آن بدون مقدار است.

- برای حل این مشکل باید متغیر day را درون تابع به صورت عمومی تعریف کنیم.

---

```
۱ day = " Monday "  
۲ def today() :  
۳     global day  
۴     print (" Today is ", day)  
۵     day = " Tuesday"  
۶     print (" Tomorrow is ", day)  
۷ today()
```

---

- متغیرها می‌توانند در توابع تو در تو نیز در پایتون تعریف شوند. اگر یک متغیر در یک تابع فرزند همنام یک متغیر در تابع پدر تعریف شده باشد و بخواهیم از متغیر تابع پدر استفاده کنیم، از کلمه کلیدی `nonlocal` استفاده می‌کنیم.
- متغیرهای عمومی می‌توانند خطر ساز باشند چرا که توابع گوناگون آنها را تغییر می‌دهند و ممکن است طراح برنامه نتواند همهٔ حالت‌هایی که متغیر عمومی ممکن است در آنها تغییر کنند را در نظر بگیرد و این موجب خروجی نادرست در برنامه شود.

- در برخی از زبان‌ها مانند لیسپ، حوزه تعریف می‌تواند پویا باشد بدین معنی که مقدار متغیر و حوزه تعریف آن بستگی به نحوه اجرا و ترتیب اجرای توابع پیدا می‌کند.
- برنامه زیر را در زبان لیسپ در نظر بگیرید :

---

```
۱ (setf      r      100)
۲ (defun    fun1(r)  (print r)    (fun2))
۳ (defun    fun2()   (print r))
```

---

- حال با اجرای (fun1 5) ابتدا مقدار ۵ و سپس مقدار ۱۰۰ چاپ می‌شود. این همان چیزی است که از حوزه تعریف ایستا انتظار داریم.



- حال برنامه زیر را در نظر بگیرید :

---

```
۱ (defparameter x 100)
۲ (defun fun1(x) (print x) (fun2))
۳ (defun fun2() (print x))
```

---

- توسط کلمه کلیدی defparameter یک متغیر با حوزه تعریف پویا تعریف می شود.
- این بار با اجرای (fun1 5) مقدار ۵ دو بار چاپ می شود.
- با فراخوانی تابع fun2 درون تابع fun1 متغیر x تبدیل به یک متغیر محلی می شود.

- نقطه قوت حوزه تعریف پویا انعطاف پذیری آن است. توابع می‌توانند بدون ارسال صریح متغیرها به توابع دیگر به عنوان پارامتر، مقدار متغیرها را انتقال دهند.
- یکی از معایب حوزه تعریف پویا این است که قابلیت اطمینان و خوانایی برنامه‌ها توسط آن پایین می‌آید، زیرا متغیرهای محلی در توابع ممکن است در توابع دیگر قابل مشاهده شوند.
- به همین دلیل حوزه تعریف پویا در بسیاری از زبان‌های برنامه نویسی استفاده نمی‌شود.

- در این فصل در مورد نوع‌های داده‌ای از جمله نوع‌های داده‌ای اصلی، آرایه‌ها، لیست‌ها و غیره صحبت خواهیم کرد.
- یک نوع داده‌ای<sup>1</sup> مجموعه‌ای از داده‌ها از جنس یکسان و عملگرهای ممکن برای انجام محاسبات بر روی آن داده‌ها را تعیین می‌کند. برنامه‌های کامپیوتری برای حل مسائل دنیای واقعی، باید بتوانند عناصر و اشیای دنیای واقعی را مدلسازی کنند و برای این مدلسازی نیاز به ساختارهای داده‌ای دارند. زبان‌های مختلف ساختارها و نوع‌های داده‌ای متفاوتی را ارائه می‌دهند. هرچه نوع‌های داده‌ای فراهم شده توسط یک زبان بیشتر باشند، مسائل را می‌توان ساده‌تر با استفاده از آن حل نمود.
- زبان‌های ابتدایی مانند کوبول و فورترن نوع‌های داده‌ای بسیار محدودی داشتند.

---

<sup>1</sup> data type

- زبان پی‌ال<sup>1</sup> زبانی بود که نوع‌های داده‌ای بسیار متنوعی را ارائه کرد. از طرف دیگر سازندگان زبان الگول تصمیم گرفتند نوع داده‌ای را محدود کرده و عملگرهایی را برای ایجاد امکان ساخت نوع‌های داده‌ای متنوع ارائه کنند. بنابراین اولین بار در این زبان نوع‌های داده‌ای تعریف شده توسط کاربر<sup>2</sup> به وجود آمدند. به این ترتیب خوانایی برنامه بسیار بالا می‌رفت. همچنین با استفاده از این داده‌ها، تغییر دادن برنامه‌های بزرگ و پیچیده نیز آسان‌تر می‌شد و کامپایلر نیز می‌توانست برای نوع داده‌ای جدید بررسی نوع<sup>3</sup> انجام دهد.
- سیستم نوع<sup>4</sup> در یک زبان برنامه نویسی تعریف می‌کند که نوع‌های مختلف را چگونه می‌توان مقدار دهی و به یکدیگر منتسب کرد.

---

<sup>1</sup> PL/I

<sup>2</sup> User-defined data types

<sup>3</sup> type checking

<sup>4</sup> type system

## نوع‌های داده‌ای

- در کنار نوع‌های داده‌ای اصلی مانند اعداد صحیح و اعشاری و کاراکتر و غیره، دو نوع داده‌ای مهم که در بیشتر زبان‌ها وجود دارند، آرایه‌ها<sup>1</sup> و رکوردها<sup>2</sup> هستند. یک آرایه مجموعه‌ای از مقادیر هم‌جنس است، در حالی که یک رکورد مجموعه‌ای از نوع‌های داده‌ای غیرهم‌جنس است. در چنین زبان‌هایی می‌توان آرایه‌ای تعریف کرد از نمونه‌هایی از رکوردهای هم‌جنس و یا می‌توان رکوردهایی را تعریف کرد که یک یا چند عنصر از آنها آرایه باشند.
- لیست‌ها نوع داده‌ای دیگری هستند که می‌توانند شامل مقادیر غیر هم‌جنس باشند. در زبان‌های تابعی مانند لیسپ، لیست‌ها بسیار مورد استفاده بودند. در سال‌های اخیر لیست‌ها در زبان‌هایی مانند پایتون نیز پیاده سازی شده‌اند.
- عملگرهای تبدیل نوع، به عملگرهایی در یک زبان گفته می‌شود که یک نوع را به نوع دیگر تبدیل می‌کنند مثلاً در زبان سی عملگر ستاره نوع داده‌ای اصلی را به اشاره‌گر و عملگر براکت یک نوع داده‌ای را به آرایه تبدیل می‌کند.

---

<sup>1</sup> array

<sup>2</sup> record

# نوع داده‌ای اصلی

- نوع داده‌هایی که خود از نوع داده‌های ساده‌تر تشکیل نشده‌اند را نوع داده‌های اصلی<sup>1</sup> می‌نامیم.
- برخی از زبان‌های برنامه نویسی ابتدا تنها نوع داده‌ای عددی داشتند. مهم‌ترین نوع داده‌ی عددی نوع عدد صحیح یا integer است.
- در زبان جاوا بسته به اندازه عدد صحیح مورد نیاز می‌توان از نوع‌های `byte`, `short`, `int`, `long` استفاده کرد.
- در برخی زبان‌ها مانند پایتون می‌توان اعداد صحیح با طول نامحدود تعریف کرد. چنین اعدادی توسط سخت افزار پشتیبانی نمی‌شوند بلکه نیاز به طراحی در سطح زبان برنامه نویسی می‌باشد. یک عدد صحیح طولانی بدون محدودیت بر روی تعداد ارقام عدد را می‌توان در زبان پایتون تعریف کرد.
- برخی از زبان‌ها مانند سی++ برای اعداد مثبت و منفی عملگر نوع تعریف کرده‌اند، بدین ترتیب می‌توان یک نوع داده‌ای شامل اعداد مثبت با استفاده از کلمه کلیدی `unsigned` تعریف کرد.

---

<sup>1</sup> primitive data types

# نوع داده‌ای اصلی

- نوع داده‌ای ممیز شناور<sup>1</sup> برای نمایش اعداد گویا و حقیقی به کار می‌رود، با این تفاوت که به دلیل محدودیت حافظه این اعداد را می‌توان تنها با تقریب در حافظه ذخیره کرد. برای ذخیره این اعداد باید قسمت اعشاری عدد که بعد از ممیز مشخص می‌شود و همچنین مرتبه بزرگی عدد را که توسط مقدار توانی مشخص می‌شود را ذخیره سازی کرد.
- بیشتر زبان‌های برنامه نویسی دو نوع داده اعشاری با دقت متفاوت به نام float و double را پشتیبانی می‌کنند که اولی در چهار بایت و دومی در هشت بایت ذخیره می‌شود.
- در نوع داده‌ای float، یک بیت برای علامت، ۸ بیت برای توان و ۲۳ بیت برای قسمت اعشاری به کار می‌رود. در نوع داده‌ای double که به معنای ممیز شناور با دقت دو برابر<sup>2</sup> است، یک بیت برای علامت، ۱۱ بیت برای توان و ۵۲ بیت برای قسمت اعشاری به کار می‌رود.

---

<sup>1</sup> floating-point

<sup>2</sup> double-precision floating-point

- برخی از زبان‌های برنامه نویسی مانند پایتون، نوع داده‌ای مختلط<sup>1</sup> را نیز پشتیبانی می‌کنند. برای مثال در پایتون می‌توان یک عدد مختلط را به صورت  $(1 + 2j)$  تعریف کرد.
- برخی از زبان‌ها مانند زبان کوبول که برای استفاده‌های تجاری به وجود آمده است و نیاز به نگهداری دقیق اعداد دهمی اعشاری دارند، یک نوع داده‌ای ویژه به نام decimal پشتیبانی می‌کنند. مزیت این نوع داده‌ای این است که اعداد اعشاری را دقیق به همان صورتی که هستند ذخیره می‌کند چرا که اعداد ممیز شناور ممکن است در تبدیل دودویی به دهمی دقتی را از دست بدهند. برای ذخیره سازی دقیق این نوع داده‌ای اعداد را مانند رشته ذخیره می‌کند.

---

<sup>1</sup> complex



# نوع داده‌ای اصلی

- نوع داده‌ای بولی <sup>1</sup> برای ذخیرهٔ مقادیر درست <sup>2</sup> و نادرست <sup>3</sup> به کار می‌رود.
- در برخی زبان‌ها مانند سی که نوع داده‌ای بولی را پشتیبانی نمی‌کنند، عدد صفر معادل مقدار نادرست و اعداد غیر صفر معادل درست به کار می‌روند.
- در زبان سی++ نوع داده‌ای بولی با کلمه bool تعریف می‌شود، برای نوع داده‌ای بولی تنها به یک بیت نیاز است اما به دلیل اینکه دسترسی به یک بیت راندمان پایینی دارد، برای ذخیره آنها از یک بایت استفاده می‌شود.

---

<sup>1</sup> boolean

<sup>2</sup> true

<sup>3</sup> false

- نوع داده‌ای کاراکتر یا حرف<sup>1</sup> برای ذخیره حروف الفبا با یک کدگذاری مشخص به کار می‌رود. حروف استاندارد اسکی<sup>2</sup> به یک بایت برای ذخیره سازی نیاز دارند. برای حروف الفبا از زبان‌های مختلف می‌توان از استانداردهای یونیکد<sup>3</sup> از جمله UTF-۳۲ استفاده کرد که به چهار بایت فضا نیاز دارد.

---

<sup>1</sup> character

<sup>2</sup> ASCII (American Standard Code for Information Interchange)

<sup>3</sup> Unicode Consortium

# نوع داده‌ای رشته‌ای

- نوع داده‌ای رشته کاراکتری<sup>1</sup> یا رشته برای ذخیره دنباله‌ای از حروف به کار می‌رود. کاربرد این نوع داده‌ای ذخیره سازی کلمات، جملات و متون است.
- برخی از زبان‌ها مانند سی و سی++، رشته‌ها به صورت آرایه‌ای از حروف تعریف می‌شوند و توابعی در کتابخانه‌های جانبی برای اعمال عملگرهایی مانند الحاق<sup>2</sup>، انتساب<sup>3</sup> و کپی زیر رشته<sup>4</sup> تعریف شده است.
- در برخی زبان‌ها مانند جاوا یا کتابخانه استاندارد سی++، رشته‌ها به صورت کلاس تعریف می‌شوند و عملگرها برای این کلاس‌ها سربارگذاری<sup>5</sup> شده‌اند.

---

<sup>1</sup> character string type

<sup>2</sup> concatenation

<sup>3</sup> assignment

<sup>4</sup> substring

<sup>5</sup> overload

## نوع داده‌ای رشته‌ای

- در برخی زبان‌ها مانند پایتون رشته‌ها به عنوان یک داده اصلی تعریف می‌شوند که همه عملگرهای مورد نیاز برای آنها تعریف شده است.
- در زبان‌هایی که طول رشته در آنها ثابت است، برای ذخیره سازی رشته تنها نیاز به آدرس شروع رشته در حافظه و طول رشته داریم. در زبان‌هایی که در طول رشته می‌تواند متغیر باشد، باید طول فعلی و طول ماکزیمم مشخص باشند.
- رشته‌هایی با طول متغیر را می‌توان توسط لیست‌های پیوندی ذخیره کرد. نقطه ضعف این روش پیچیدگی آن برای رشته‌های طولانی و در نتیجه راندمان پایین آن است. روش دیگر استفاده از آرایه‌ای از کاراکترهاست که نقطه ضعف آن محدودیت حافظه برای رشته‌های طولانی است. روشی که معمولاً استفاده می‌شود این است که رشته در یک آرایه نگهداری می‌شود و هنگامی که طول آرایه نیاز به افزایش داشت فضای جدیدی در هیپ با حافظه مورد نیاز تخصیص داده شده و رشته از مکان قبلی به مکان فعلی منتقل می‌شود.

- نوع داده‌ای شمارشی<sup>1</sup> نوعی است که توسط آن می‌توان یک مقدار از بین چند مقدار نامگذاری شده را انتخاب کرد. به عبارت دیگر این نوع داده‌ای مقادیر ثابت نامگذاری شده<sup>2</sup> را تعریف می‌کند.

---

<sup>1</sup> enumeration type

<sup>2</sup> named constant

## نوع داده‌ای شمارشی

- برای مثال در زبان سی می‌توان نوع داده‌ای day را به صورت `enum day { Mon, Tue, Wed, Thu, Fri, Sat, Sun }` تعریف کرد. مقادیر نوع داده‌ای شمارشی معمولاً به صورت اعداد صحیح ذخیره می‌شوند. در زبان سی++ می‌توان بر روی داده‌های شمارشی عملگر نیز تعریف کرد و عملیات بر روی متغیرهای از نوع شمارشی اعمال کرد. برای این کار باید از `enum class` استفاده کرد. همچنین با استفاده از `enum class` در زمان کامپایل می‌توان نوع داده شمارشی را بررسی کرد. اگر داشته باشیم :
- `enum class color = { red, blue, yellow }` و `enum class rgb = { red, green, blue }`
- آنگاه کامپایلر برای `rgb c = blue` پیام خطا صادر می‌کند و باید مشخص کرد این مقدار از چه نوعی است.

- نوع دادهٔ آرایه <sup>1</sup> برای نگهداری مقادیر داده‌ای هم نوع استفاده می‌شود، به طوری که هر مقدار در آرایه توسط مکان آن در آرایه نسبت به اول آرایه قابل دسترسی است.
- معمولا مقادیر آرایه توسط عملگر زیرنویس <sup>2</sup> یا اندیس <sup>3</sup> می‌توان دسترسی پیدا کرد.
- اندیس یک آرایه در بیشتر زبان‌ها از جمله سی++، جاوا و پایتون با براکت مشخص می‌شود.
- بسیاری از زبان‌ها بررسی دسترسی در آرایه <sup>4</sup> ندارند، اما جاوا بازهٔ دسترسی را بررسی می‌کند.

---

<sup>1</sup> array

<sup>2</sup> subscript operator

<sup>3</sup> index

<sup>4</sup> array range check

- چهار نوع مختلف از آرایه‌ها وجود دارند : (۱) آرایه‌های ایستا<sup>۱</sup> ، (۲) آرایه‌های پویای ثابت روی پشته<sup>۲</sup> ، (۳) آرایه‌های پویای ثابت بر روی هیپ<sup>۳</sup> و (۴) آرایه‌های پویای بر روی هیپ<sup>۴</sup>.
- آرایه‌های ایستا قبل از اجرای برنامه اندازه معین دارند و قبل از شروع برنامه بر روی قسمت داده<sup>۵</sup> حافظه مقید می‌شوند. در زبان سی++، این دسته از آرایه‌ها به صورت `static type name[N]` تعریف می‌شوند. به طوری که N یک عدد صحیح است.

---

<sup>۱</sup> static arrays

<sup>۲</sup> fixed stack-dynamic arrays

<sup>۳</sup> fixed heap-dynamic arrays

<sup>۴</sup> heap-dynamic arrays

<sup>۵</sup> data segment



- آرایه‌های پویای ثابت بر روی پشته قبل از اجرای برنامه اندازه معین دارند و در هنگام تعریف بر روی پشته حافظه مقید می‌شوند. این آرایه‌ها در سی++ به صورت `type name[N]` تعریف می‌شوند.
- آرایه‌های پویای ثابت بر روی هیپ، قبل از اجرای برنامه اندازه معین ندارند و در زمان اجرا اندازه آنها تعیین و بر روی پشته مقید می‌شوند. وقتی این آرایه‌ها به حافظه مقید شدند اندازه آنها غیر قابل تغییر است. این آرایه‌ها در زبان سی++ به صورت `type[] name = new type[n]` تعریف می‌شوند، به طوری که `n` یک متغیر یا ثابت است.
- آرایه‌های پویای بر روی هیپ، اندازه آنها در زمان اجرا تعیین می‌شود و همچنین انقیاد حافظه آنها در زمان اجرا صورت می‌گیرد. همچنین می‌توان چندین بار در زمان اجرا فضای حافظه آنها را آزاد و دوباره تخصیص داد. وکتورها در زبان سی++ در این دسته از آرایه‌ها قرار می‌گیرند. در زبان سی تخصیص حافظه بر روی هیپ در زمان اجرا با استفاده از دستور `malloc` و آزادسازی حافظه با استفاده از `free` انجام می‌شود. در سی++ نیز تخصیص با `new` و آزادسازی با `delete` صورت می‌گیرد.

- در زبان پایتون از نوع داده‌ای لیست می‌توان به عنوان آرایه استفاده کرد. می‌توان آرایه را به صورت زیر تعریف کرد و به عناصر آن مقدار افزود.

---

```
۱ array = [1,2,3]
۲ array.append(4)
```

---

در زبان سی و سی++ و جاوا، به طور پیش فرض بر روی آرایه‌ها عملگر تعریف نشده است، اما برای لیست‌ها در پایتون عملگرهایی تعریف شده است.

- مثلاً عملگر + دو لیست را به یکدیگر الحاق می‌کند و توسط عملگر in می‌توان بررسی کرد آیا مقداری در آرایه وجود دارد یا خیر. عملگر == بررسی می‌کند آیا دو لیست از نظر اندازه و مقدار با یکدیگر برابرند یا خیر.

---

```
۱ a = [1,2]
۲ b = [3,4]
۳ c = a + b
۴ if 4 in c or a==b :
۵     print ("ok")
```

---

- در برخی از زبان‌ها مانند روبی، مفهومی به نام برش آرایه<sup>1</sup> وجود دارد که توسط آن می‌توان قسمتی از یک آرایه را به عنوان یک آرایه دیگر استفاده کرد.
- در برش آرایه، توسط عملگر : در داخل براکت می‌توان شروع و پایان برش را تعیین کرد. قطعه کد زیر چند مثال از برش آرایه‌ها آورده شده است.

---

```
۱ vector = [2,4,6,8,10,12,14,16]
۲ mat = [[1,2,3],[4,5,6],[7,8,9]]
۳ v1 = vector[3:6] # v1=[8,10,12]
۴ m1 = mat[0][0:2] # m1=[1,2]
۵ m2 = mat[:1] # m2=[[1,2,3]]
```

---

---

<sup>1</sup> slice of array

## نوع داده آرایه

- به طور کل قوانین برش در پایتون به صورت زیر می باشند.

```
۱ a[start : stop] # [a[start], ... , a[stop-1]]  
۲ a[start :] # [a[start], ... , a[len(a)-1]]  
۳ a[: stop] # [a[0], ... , a[stop-1]]  
۴ a[:] # [a[0], ... , a[len(a)-1]]
```

- همچنین می توان علاوه بر شروع و پایان اندیس برش، مقدار افزایش اندیس در هرگام برش را نیز به صورت زیر تعیین کرد.

```
۱ a[start : stop : step] # items from start index incremented  
۲                        # by step not after stop -1
```

- به طور مثال :

```
۱ v2 = vector [0:4:2] # v2 = [2,6]
```

- مقدار منفی در اندیس‌های آرایه و همچنین برش، به معنی شمارش از آخر است.

---

```
۱ vector [-2] # 14
۲ vector [-2 :] # [14,16]
۳ vector [: -2] # [2,4,6,8,10,12]
```

---

- مقدار منفی در پارامتر سوم برش به معنی شمارش معکوس است.

---

```
۱ vector [:: -1] # [16,14,12,10,8,6,4,2]
۲ vector [1 :: -1] # [4,2]
۳ vector [: -3 : -1] # [16,14]
```

---

## نوع داده آرایه

- برای پیاده سازی آرایه در یک زبان برنامه نویسی نیاز به دسترسی به آدرس حافظه هر یک از عناصر آن داریم. آدرس یک سلول از حافظه را می توانیم با استفاده از رابطه زیر به دست آوریم :

$$\text{address}(\text{array}[k]) = \text{address}(\text{array}[0]) + k * \text{element-size}$$

- در زبان هایی که محدوده دسترسی اندیس را بررسی می کنند، کامپایلر نیاز به نگهداری اطلاعات مربوط به آدرس آرایه و نوع آرایه و اندازه آرایه دارد ولی در صورتی که کامپایلر محدوده دسترسی را بررسی نکند، نیازی به نگهداری اندازه آرایه نیست.

- برای پیاده سازی آرایه های چند بعدی نیاز به محاسبه آدرس حافظه یک درایه برای دسترسی به آن را داریم، زیرا حافظه یک بعدی است. برای مثال در یک آرایه دو بعدی که همه سطرهای آن طول یکسان دارند، داریم :

$$\text{address}(\text{m}[i][j]) = \text{address}(\text{m}[0][0]) + (i * n + j) * \text{element-size}$$

- یک آرایه انجمنی<sup>1</sup> مجموعه‌ای است از مقادیر که برای دسترسی به آنها از مقادیری به نام کلید استفاده می‌کنیم. به عبارت دیگر هر یک از عناصر یک رابطه انجمنی جفتی است که قسمت اول آن کلید و قسمت دوم آن مقدار نامیده می‌شود. برای دسترسی به یک مقدار باید از کلید مربوط به آن استفاده کرد.
- در آرایه‌های غیر انجمنی در واقع کلیدها، اندیس‌هایی هستند که مکان یک مقدار را در آرایه تعیین می‌کنند.
- آرایه‌های انجمنی در زبان‌ها برل، پایتون و روبی پیاده سازی شده‌اند.

---

<sup>1</sup> associative array



# آرایه‌های انجمنی

- در زبان پایتون به آرایه‌های انجمنی، دیکشنری<sup>1</sup> گفته می‌شود.
- در زبان پایتون کلیدهای یک دیکشنری می‌توانند تنها رشته‌ها و اعداد باشند در حالی که در روبی کلیدها می‌توانند از هر نوع کلاسی باشند.
- برای پیاده سازی آرایه انجمنی در پرل، به ازای هر کلید یک مقدار هش<sup>2</sup> ۳۲ بیتی محاسبه می‌شود.
- در زبان پایتون یک متغیر از نوع دیکشنری به صورت زیر تعریف می‌شود.

---

```
۱ d = {1: 'one', 'two': 2}
۲ d[1] # 'one'
۳ d['two'] # 2
```

---

---

<sup>1</sup> dictionary

<sup>2</sup> hash value

- یک رکورد<sup>1</sup> تعدادی متغیر که هر کدام می‌توانند از یک نوع متفاوت باشند را تجميع می‌کند. هر عنصر از یک رکورد با یک نام و یک نوع مشخص می‌شود و مکان آن در حافظه نسبت به ابتدای رکورد با محاسبه اندازه عناصر قبلی آن قابل محاسبه است.
- در سی و سی++، برای تعریف یک رکورد از نوع داده‌ای ساختمان یا استراکت<sup>2</sup> استفاده می‌شود.
- عناصر یک رکورد برخلاف آرایه که با اندیس مشخص می‌شوند با نام و نوع عنصر مشخص می‌شوند. هر عنصر یک رکورد، فیلد نامیده می‌شود که در بیشتر زبان‌ها با عملگر نقطه (.) قابل دسترسی هستند.

---

<sup>1</sup> record

<sup>2</sup> struct

# نوع داده‌ای چندتایی

- یک نوع داده‌ای چندتایی <sup>1</sup> نوع داده‌ای است برای نگهداری مقادیر از انواع متفاوت. به عناصر چندتایی با اندیس یا شماره آنها در چندتایی می‌توان دسترسی پیدا کرد.
- بنابراین یک چندتایی از لحاظ این که به عناصرش با اندیس می‌توان دسترسی پیدا کرد شبیه آرایه است و تفاوت آن با آرایه این است که عناصر آن می‌توانند از نوع‌های متفاوت باشند.
- در زبان پایتون، عناصر لیست قابل تغییر <sup>2</sup> هستند، ولی عناصر چندتایی غیر قابل تغییر <sup>3</sup> اند.
- یک مورد استفاده از چندتایی وقتی است که می‌خواهیم تعدادی مقدار به تابعی دیگر ارسال کنیم ولی نمی‌خواهیم تابع بتواند مقادیر متغیر ارسال شده را تغییر دهد.

---

<sup>1</sup> tuple

<sup>2</sup> mutable

<sup>3</sup> immutable

## نوع داده‌ای چندتایی

- در زبان پایتون چندتایی را با استفاده از پرانتز تعریف می‌کنیم.

---

```
۱ t = (3 , 1.2 , 'hello')  
۲ t [1] # 1.2  
۳ t [0] = 2 # error
```

---

- عملگر + بر روی چندتایی تعریف شده است که دو چندتایی را با هم الحاق می‌کند.

## نوع داده‌ای لیست

- لیست در اولین زبان تابعی یعنی لیسپ به وجود آمد و از اهمیت ویژه‌ای در همهٔ زبان‌های تابعی برخوردار است.
- لیست مجموعه‌ای است از عناصر با نوع‌های متفاوت به طوری که مقدار عناصر آن قابل تغییر هستند.  
بنابراین لیست شبیه چندتایی است با این تفاوت که مقادیر عناصر آن را می‌توان تغییر داد و شبیه آرایه است با این تفاوت که نوع عناصر آن ممکن است یکسان نباشد.
- در زبان پایتون یک لیست به صورت زیر تعریف می‌شود :

---

```
۱ l = [1 , 2.3 , 'apple' ]  
۲ l [1] = 'grape'
```

---

- یک عنصر از لیست را می‌توان توسط عملگر `del` حذف کرد.

# نوع داده‌ای لیست

- یک متغیر از نوع چندتایی را می‌توان توسط تابع `list` به لیست تبدیل کرد. همچنین یک متغیر از نوع لیست را می‌توان توسط تابع `tuple` به چندتایی تبدیل نمود.
- پایتون روشی مختصر برای توصیف لیست ارائه می‌کند که روش شمول کامل<sup>1</sup> نامیده می‌شود.

---

```
۱ # [expression for var in list if condition]
۲ a = [x * x for x in range(12) if x % 3 == 0]
۳ # a = [0 , 9 , 36 , 81]
```

---

---

<sup>1</sup> list comprehension

- نوع داده‌ای اجتماع<sup>1</sup>، نوعی است که متغیر آن در زمان‌های متفاوت می‌تواند نوع‌های متفاوت داشته باشد. برای مثال فرض کنید به متغیری نیاز داشته باشیم که گاهی در آن عدد صحیح قرار می‌گیرد و گاهی عدد اعشاری. اگر بخواهیم دو متغیر تعریف کنیم در هر بازه زمانی یکی از آنها بدون استفاده می‌ماند. اگر این دو متغیر را در یک اجتماع قرار دهیم برای هر دوی آنها یک فضای حافظه برابر با حافظه مورد نیاز برای متغیر بزرگ‌تر تخصیص داده می‌شود.

---

<sup>1</sup> union

# نوع داده‌ای اجتماع

- در سی و سی++ این نوع با کلمه union تعریف می‌شود.

```
۱ union Number {  
۲     int i ;  
۳     float f ;  
۴ };  
۵ union Number n;  
۶ n.i = 2  
۷ n.f = 3.1 // n.i is erased
```

- در پیاده سازی اجتماع همه عناصر آن یک آدرس حافظه می‌گیرند.



## نوع‌های اشاره‌گر و ارجاعی

- یک اشاره‌گر<sup>1</sup> نوعی است که متغیر آن آدرس یک سلول از حافظه را نگهداری می‌کند. همچنین مقدار آن می‌تواند تهی باشد که در این صورت به هیچ مکانی در حافظه اشاره نمی‌کند.
- اشاره‌گرها استفاده‌های متعددی دارند. یکی از موارد استفاده آنها در فراخوانی توابع است. ارسال آدرس متغیرها به توابع به جای ارسال مقدار آنها موجب بهبود سرعت اجرای برنامه می‌شود. مورد استفاده دیگر اشاره‌گرها تخصیص حافظه پویا در فضای هیپ است. به فضای حافظه تخصیص داده شده توسط یک اشاره‌گر می‌توان دسترسی پیدا کرد.
- برای مثال فرض کنید بخواهیم یک لیست پیوندی بسازیم که تعداد عناصر آن مشخص نباشد. به ازای هر عنصر در لیست پیوندی باید یک فضای جدید در حافظه هیپ تخصیص دهیم به طوری که هر عنصر لیست به عنصر بعدی خود اشاره می‌کند.
- یک متغیر از نوع اشاره‌گر یک آدرس را نگهداری می‌کند و در زبان سی و سی++ می‌توان توسط عملگر ستاره (\*) به مقدار یک مکان حافظه دسترسی پیدا کرد.

---

<sup>1</sup> pointer

## نوع‌های اشاره‌گر و ارجاعی

- زبان‌هایی که نوع اشاره‌گر را پیاده‌سازی می‌کنند به یک عملگر یا تابع برای تخصیص حافظه پویا و انتساب آن به اشاره‌گر نیاز دارند که این عملگر در زبان سی++ توسط کلمه `new` پیاده‌سازی شده است. همچنین با استفاده از عملگر `delete` می‌توان حافظه تخصیص داده شده را آزاد کرد.
- عملگری که برای دریافت مقدار مکان حافظه‌ای که توسط اشاره‌گر قابل دسترسی است استفاده می‌شود عملگر رفع ارجاع<sup>1</sup> نام دارد. این عملگر در زبان سی++ توسط `&` ستاره (\*) پیاده‌سازی شده است.
- عملگر مورد نیاز دیگر، جهت دریافت آدرس یک متغیر برای ذخیره‌سازی آدرس در اشاره‌گر است. در زبان سی++ با استفاده از عملگر امپرسند (&) می‌توان آدرس یک متغیر را به دست آورد.
- اولین زبانی که نوع اشاره‌گر را پیاده‌سازی کرد زبان پی‌ال‌ای<sup>۱</sup> بود.

---

<sup>1</sup> dereference

## نوع‌های اشاره‌گر و ارجاعی

- اشاره‌گرها می‌توانند خطراتی را نیز به همراه داشته باشند که قابلیت اطمینان برنامه را پایین می‌آورند.
- برای مثال اگر اشاره‌گری به یک فضای حافظه در هیپ اشاره کند و فضا توسط دستوری آزاد شود ولی اشاره‌گر همچنان به آن فضای حافظه اشاره کند، اشاره‌گری داریم که مقدار ناصحیح دارد و در برخی زبان‌ها ممکن است دسترسی به این اشاره‌گر معلق<sup>1</sup> موجب توقف برنامه شود. همچنین اگر در همان فضای حافظه مجدداً حافظه تخصیص داده شود، اشاره‌گر مذکور ممکن است مقدار ناصحیح از حافظه بخواند.
- مشکل دیگر اشاره‌گرها این است که ممکن است برنامه نویس بدون آزاد سازی فضای حافظه‌ای که یک اشاره‌گر به آن اشاره می‌کند، اشاره‌گر را به مکان دیگری اشاره دهد. در این صورت مکان حافظه در هیپ غیر قابل دسترسی می‌شود. به این پدیده نشست حافظه<sup>2</sup> گفته می‌شود که ممکن است پس از انباشته شده زیاد مکان‌های تخصیص داده شده موجب پر شدن حافظه و توقف برنامه شود.
- همچنین اگر چند اشاره‌گر به یک فضای حافظه اشاره کنند، ممکن است به اشتباه بخواهیم یک فضای حافظه را چند بار آزاد کنیم.

---

<sup>1</sup> dangling pointer

<sup>2</sup> memory leakage

# نوع‌های اشاره‌گر و ارجاعی

- نوع داده‌ای ارجاعی شبیه اشاره‌گر است با این تفاوت که یک متغیر ارجاعی آدرس نگهداری نمی‌کند بلکه نام مستعاری است برای یکی از خانه‌های حافظه.
- بنابراین متغیر ارجاعی وقتی برای اولین بار به خانه‌ای در حافظه اشاره کرد، در طول برنامه فقط به همان خانه حافظه می‌تواند اشاره کند.
- در زبان سی++ می‌توان یک متغیر ارجاعی را با استفاده از عملگر امپرسند<sup>1</sup> (&) به صورت زیر تعریف کرد.

---

```
۱ int x = 2;  
۲ int &r = x;  
۳ r++; // r = 3 and x = 3
```

---

---

<sup>1</sup> ampersand

# نوع‌های اشاره‌گر و ارجاعی

- نوع داده‌ای ارجاعی برای ارسال متغیر به یک تابع استفاده می‌شود هنگامی که بخواهیم مقدار آن متغیر توسط تابع تغییر کند.

---

```
۱ void swap (int& x, int& y) {  
۲     int tmp = x;  
۳     x = y;  
۴     y = tmp;  
۵ }
```

---

## نوع‌های اشاره‌گر و ارجاعی

- مورد استفاده دیگر وقتی است که می‌خواهیم متغیری که فضای زیادی در حافظه اشغال می‌کند (برای مثال یک نوع داده‌ای تعریف شده توسط کاربر) را به تابعی ارسال کنیم و نمی‌خواهیم تابع از آن متغیر کپی بگیرد بلکه می‌خواهیم تنها آدرس مکان حافظه را به تابع ارسال شود. به این فرایند فراخوانی تابع فراخوانی با ارجاع می‌گوییم. مزیت استفاده از متغیر ارجاعی در اینجا نسبت به اشاره‌گر این است که عملیات توسط آن ساده‌تر انجام می‌شود.

---

```
۱ *zptr = *xptr + *yptr;  
۲ zref = xref + yref;
```

---

- در زبان جاوا همه اشیای کلاس‌ها متغیر ارجاعی هستند.

---

```
۱ A a1 = new A();  
۲ A a2 = a1;  
۳ a1.x = 1; // a2.x = 1
```

---

# نوع‌های اشاره‌گر و ارجاعی

- یکی از روش‌های پیاده سازی اشاره‌گرها، روش قفل و کلید<sup>1</sup> نامیده می‌شود.
- در این روش، وقتی اشاره‌گر به یک مکان در حافظه هیپ اشاره می‌کند، اشاره‌گر علاوه بر آدرس حافظه یک کلید را ذخیره می‌کند که یک عدد صحیح است. در مکان حافظه‌ای که آن اشاره‌گر به آن اشاره می‌کند نیز همان مقدار ذخیره می‌شود که به آن قفل گفته می‌شود. در هنگام دسترسی یک اشاره‌گر به مکان حافظه، اگر قفل و کلید همخوانی داشته باشند دسترسی مجاز است. هنگامی که فضایی در حافظه آزاد می‌شود، مقدار قفل آن فضا تغییر می‌کند، بنابراین همه اشاره‌گرهایی که به آن مکان اشاره می‌کنند هنگام دسترسی با پیام خطا مواجه می‌شوند چرا که قفل و کلید دیگر همخوانی ندارند.
- با استفاده از این روش مشکل اشاره‌گر معلق رفع می‌شود.
- در برخی زبان‌ها مانند جاوا اجازه آزادسازی حافظه به برنامه نویس داده نمی‌شود و بنابراین مشکل اشاره‌گر معلق وجود نخواهد داشت.

---

<sup>1</sup> locks and keys approach

# نوع‌های اشاره‌گر و ارجاعی

- در زبان‌هایی مانند جاوا که جمع‌آوری فضا‌های آزاد شده یا بازیافت حافظه (زباله‌روبی)<sup>1</sup> به طور خودکار انجام می‌شود، باید الگوریتم بهینه برای این کار نیز پیاده سازی شود.
- دو روش برای بازیافت حافظه وجود دارد که روش اول شمارنده ارجاع<sup>2</sup> و روش دوم علامت گذاری و جاروب<sup>3</sup> نامیده می‌شوند.
- در روش شمارنده ارجاع به ازای هر سلول حافظه شمارنده‌ای در نظر گرفته می‌شود که تعداد اشاره‌گرهایی که به هر مکان از حافظه اشاره می‌کنند را می‌شمارد. هرگاه تعداد اشاره‌گرها یا متغیرهای ارجاعی که به یک سلول حافظه اشاره می‌کنند به صفر رسید، سلول حافظه به مجموعه سلول‌های قابل استفاده باز می‌گردد. از آنجایی که تعداد سلول‌های حافظه زیاد است تعداد شمارنده‌هایی که باید نگهداری شوند سربار زیادی به سیستم تحمیل می‌کند. در زمان اجرا نیز این شمارش سربار زمانی ایجاد می‌کند و باعث کاهش سرعت می‌شود.

---

<sup>1</sup> garbage collection

<sup>2</sup> reference counter

<sup>3</sup> mark and sweep



- در روش نشانه‌گذاری و جاروب، ابتدا سلول‌های حافظه تخصیص داده می‌شوند تا هنگامی که حافظه پر شود. در این لحظه بازیافت‌کننده حافظه همه متغیرهای ارجاعی و اشاره‌گرها در برنامه را در حافظه دنبال می‌کند و همه فضاهایی که توسط آن اشاره‌گرها استفاده می‌شوند را علامت‌گذاری می‌کند. سپس همه سلول‌های حافظه که علامت‌گذاری نشده‌اند جاروب می‌شوند یا به عبارتی در مجموعه سلول‌های قابل تخصیص قرار می‌گیرند.

- بررسی نوع<sup>1</sup> فعالیتی است که به موجب آن اطمینان حاصل می‌شود که همه عملگرها با عملوندهای آنها همخوانی دارند. در اینجا توابع را نیز عملگر در نظر می‌گیریم و ورودی و خروجی توابع را نیز عملوند برای توابع.
- یک نوع سازگار<sup>2</sup> نوعی است که برای یک عملگر تحت قوانین حاکم بر آن زبان معتبر باشد. برای مثال وقتی یک عدد صحیح و اعشاری در زبان جاوا جمع می‌شوند، عدد صحیح به اعشاری تبدیل می‌شود بنابراین این دو نوع عملوند تحت قوانین جاوا با عملگر سازگارند.

---

<sup>1</sup> type checking

<sup>2</sup> compatible

- خطای نوع<sup>1</sup> هنگامی رخ می‌دهد که عملگر با عملوندهایش سازگاری نداشته باشد.
- اگر انقیاد نوع ایستا باشد آنگاه بررسی نوع می‌تواند در زمان کامپایل انجام شود اما اگر انقیاد نوع پویا باشد بررسی نوع در زمان اجرا خواهد بود که بررسی نوع پویا<sup>2</sup> نامیده می‌شود
- بررسی نوع در زبان پایتون پویاست که باعث می‌شود خطاهای نوع قبل از اجرا مشخص نشوند اما از طرفی در این زبان انعطاف پذیری برنامه افزایش یافته است.

---

<sup>1</sup> type error

<sup>2</sup> dynamic type checking

- راست<sup>1</sup> یک زبان برنامه نویسی است که پارادایم‌های برنامه نویسی رویه‌ای، تابعی و همروند را پشتیبانی می‌کند.
- بیشترین تمرکز زبان برنامه نویسی راست بر روی کارایی<sup>2</sup>، قابلیت اطمینان<sup>3</sup> و همروندی<sup>4</sup> است.

---

<sup>1</sup> rust

<sup>2</sup> efficiency

<sup>3</sup> reliability

<sup>4</sup> concurrency

- زبان راست یک زبان کامپایل شونده است و کامپایلر آن به نحوی طراحی شده است که برنامه‌های آن از کارایی (سرعت اجرا) بالایی برخوردارند. علاوه بر این کامپایلر اطمینان حاصل می‌کند که برنامه نوشته شده در زمان اجرا دسترسی غیر مجاز به حافظه ندارد. در زبان راست، بازیافت کننده حافظه<sup>1</sup> وجود ندارد اما ساختارهایی وجود دارد که به کامپایلر کمک می‌کند بتواند در زمان کامپایل از نشستی‌های احتمالی حافظه مطلع شده و پیام خطا صادر کند. بنابراین برنامه‌های نوشته شده در زبان راست از قابلیت اطمینان بالایی برخوردارند. همچنین ساختارهایی برای برنامه نویسی همروند به این زبان افزوده شده است.
- در دسامبر ۲۰۲۲، زبان راست به عنوان اولین زبان جدید در کنار زبان‌های قدیمی سی و اسمبلی در توسعه هسته لینوکس مورد استفاده قرار گرفت.

---

<sup>1</sup> garbage collector

- متغیرها در زبان راست غیر قابل تغییر<sup>2</sup> هستند. این ویژگی باعث می‌شود بتوانید برنامه‌هایی بنویسید که از قابلیت اطمینان بالاتری برخوردار هستند. به خصوص هنگامی که در برنامه نویسی همروند می‌خواهیم دسترسی به متغیرها را کنترل کنیم، غیر قابل تغییر بودن متغیرها به امنیت برنامه کمک می‌کند.
- با این وجود می‌توانیم متغیرها را به گونه‌ای تعریف کنیم که مقدار آنها قابل تغییر باشد.

---

<sup>2</sup> immutable

## راست : متغیرها

- اگر یک متغیر غیرقابل تغییر باشد، پس از اینکه مقداری به نام آن مقید می‌شود، نمی‌توانیم مقدار آن را تغییر دهیم.
- یک متغیر (غیر قابل تغییر) با کلمه `let` تعریف می‌شود. اگر مقدار یک متغیر را تغییر دهیم، کامپایلر پیام خطا صادر می‌کند.

```
۱ fn main() {  
۲     let x = 5 ;  
۳     println! ("The value of x is : {x}") ;  
۴     x = 6 ; // error  
۵ }
```

- اگر یک قسمت از کد متغیری را با فرض اینکه غیر قابل تغییر است تعریف کند و یک قسمت دیگر از کد بخواهد مقدار آن را تغییر دهد، با خطای کامپایلر مواجه می‌شویم.

– اما گاهی نیاز داریم مقدار متغیر را تغییر دهیم که در اینصورت با استفاده از کلمه `mut` مخفف کلمه قابل تغییر<sup>1</sup> می‌توان یک متغیر قابل تغییر تعریف کرد.

---

```
۱ let mut x = 5 ;  
۲ x = 6 ;
```

---

---

<sup>1</sup> mutable



## راست : متغیرها

- همانند متغیرهای غیرقابل تغییر، ثابت‌ها<sup>1</sup> نیز مقادیری هستند که به یک نام مقید می‌شوند و قابل تغییر نیستند.
- با این حال تفاوت‌هایی بین ثابت‌ها و متغیرهای غیرقابل تغییر وجود دارد.
- اول این که ثابت‌ها نمی‌توانند هیچ گاه قابل تغییر باشند، اما متغیرها را می‌توان به صورت قابل تغییر نیز تعریف کرد. همچنین ثابت‌ها را نمی‌توان مجدداً تعریف کرد، اما متغیرهای غیرقابل تغییر قابل تعریف مجدد هستند.
- دومین تفاوت این است که متغیرها را نمی‌توان در حوزه تعریف عمومی تعریف کرد، زیرا متغیرهای عمومی معمولاً قابلیت اطمینان برنامه را کاهش می‌دهند.
- سومین تفاوت این است که ثابت‌ها را تنها با مقادیر ثابت می‌توان مقداردهی کرد و نمی‌توان مقدار یک متغیر را به یک ثابت تخصیص داد، در صورتی که یک متغیر را می‌توان با استفاده از مقادیر دیگر متغیرها مقداردهی کرد.

---

<sup>1</sup> constants

- با استفاده از کلمه `const` می‌توان یک ثابت تغییر کرد. مقدار یک ثابت قابل تغییر نیست و نمی‌توان آن را مجدداً تعریف کرد، در صورتی که یک متغیر غیرقابل تغییر را می‌توان مجدداً تعریف کرد.

---

```
۱  const PI = 3.141592 ;  
۲  PI = 3.1415 ; // error  
۳  const PI : f16 = 3.1415 ; // error  
۴  let pi = 3.141592 ;  
۵  let pi = 3.1415 ; // OK  
۶  const C : f16 = pi ; // error  
۷  let var : f16 = pi ; // OK
```

---

- یک متغیر را می‌توان در یک بلوک تعریف کرد و حوزه تعریف آن متغیر فقط مختص بلوک مورد نظر است.

---

```
۱ let x = 5 ;  
۲ let x = x + 1 ;  
۳ {  
۴     let x = x * 2 ; // x = 12  
۵ }  
۶ println! ("The value of x is : {x}") ; // x = 6
```

---

- وقتی یک متغیر بازتعریف می‌شود، نوع آن می‌تواند متفاوت باشد. اما وقتی یک متغیر قابل تغییر را مقداردهی مجدد می‌کنیم، نوع آن باید الزاما همان نوعی باشد که در ابتدا تعریف شده است.

---

```
۱ let s = "021" ;  
۲ let s = s.len() ; // s = 3  
۳ let mut m = "012" ;  
۴ m = m.len() ; //error
```

---

## راست : نوع‌های داده‌ای

- نوع‌های داده‌ای عددی در زبان راست می‌توانند صحیح علامت دار، صحیح بدون علامت و یا اعشاری باشند. عدد صحیح علامت دار با `i` ، عدد صحیح بدون علامت با `u` و عدد اعشاری با `f` نشان داده می‌شوند.
- اگر نوع یک متغیر توسط برنامه نویس تعریف نشود، کامپایلر توسط اولین مقداردهی نماد، نوع آن را مشخص می‌کند.

---

```
۱ let x = 2.0 ; // f64
۲ let y : f32 = 3.0 ; // f32
۳ let z = 4 ; // i32
۴ let w : i128 = 5 ; // i128 (128-bit integer)
۵ let m : u64 = 6 ; // u64 (64-bit unsigned integer)
```

---

## راست : نوع‌های داده‌ای

- نوع داده‌ای منطقی توسط کلمهٔ `bool` تعریف می‌شود.

---

```
۱ let t = true ;  
۲ let f : bool = false ;
```

---

- نوع داده‌ای کاراکتر توسط کلمهٔ `char` تعریف می‌شود.

---

```
۱ let c = 'z' ;  
۲ let z : char = 'Z' ;
```

---

## راست : نوع‌های داده‌ای

- نوع داده‌ای چندتایی مجموعه‌ای است از چند مقدار از نوع‌های دلخواه.

---

```
\ let x : (i32 , f64 , char) = (500 , 604 , 'y') ;
```

---

## راست : نوع‌های داده‌ای

- نوع داده‌ای آرایه نوعی است که مجموعه‌ای از مقادیر از یک نوع یکسان را نگهداری می‌کند.

---

```
۱ let a = [1, 2, 3, 4, 5] ;
```

---



## راست : نوع‌های داده‌ای

- آرایه را می‌توان با نوع عناصر و تعداد عناصر آن نیز تعریف کرد.

```
۱ let a : [i32 ; 5] = [1, 2, 3, 4, 5] ;  
۲ let a0 = a[0] ; // a0 = 1
```

- همچنین می‌توان مقادیر یک آرایه را با استفاده از یک روش میانبر تعریف کرد.

```
۱ let a = [3 ; 5] ; // [3, 3, 3, 3, 3]
```

- دسترسی به عناصر یک آرایه در بیرون از محدوده منجر به خطای زمان کامپایل می‌شود.

- همچنین استفاده از مقادیر آرایه بدون اینکه مقداردهی شده باشند، باعث صدور خطای کامپایل می‌شود.

## راست : توابع

- یک تابع را می‌توان یا پارامترهای ورودی آن و نوع خروجی آن تعریف کرد.

```
۱ fn plus_one (x : i32) -> i32 {  
۲     x+1  
۳ }
```

- عبارتی که در یک بلوک بدون نقطه ویرگول یا سمی‌کالن<sup>1</sup> اعلام شده است، از بلوک بازگردانده می‌شود.

- برای مثال می‌توانیم بنویسیم :

```
۱ let y = {  
۲     let x = 3 ;  
۳     x + 1  
۴ } ;  
۵ // y = 4
```

---

<sup>1</sup> semicolon

## راست : ساختار کنترلی

- ساختار کنترلی شرطی به صورت زیر استفاده می‌شود.

```
۱ if n % 2 == 0 {  
۲     // even  
۳ } else {  
۴     // odd  
۵ }  
۶ let number = if n % 2 ==0 {6} else {5} ;
```

## راست : ساختار کنترلی

- از کلمه `loop` می‌توان برای ایجاد یک حلقه تکرار بدون شرط استفاده کرد.

```
۱ loop {  
۲     println! ("again!");  
۳ }
```

- یک حلقه می‌تواند یک مقدار را نیز بازگرداند.

```
۱ let mut counter = 0 ;  
۲ let result = loop {  
۳     counter += 1;  
۴     if counter == 10 {  
۵         break counter * 2 ;  
۶     }  
۷ };  
۸ println! ("The result is {result}") ; // 20
```

- همچنین با استفاده از ساختار while می توان یک حلقه ایجاد کرد.

---

```
۱ let a = [10, 20, 30, 40, 50] ;  
۲ let mut index = 0 ;  
۳ while index < 5 {  
۴     println! ("The value is : {}", a[index]);  
۵     index += 1 ;  
۶ }
```

---

- همچنین با استفاده از دستور for می‌توان به صورت زیر جهت پیمایش در یک آرایه و یا یک بازهٔ عددی حلقه ایجاد کرد.

---

```
۱ let a = [10, 20, 30, 40, 50] ;
۲ for element in a {
۳     println! ("The value is : {element}");
۴ }
۵ for index in (0 .. 4) {
۶     println! ("The value is : {}", a[index]);
۷ }
```

---

- مالکیت<sup>1</sup> یکی از مفاهیم اصلی و ویژهٔ زبان راست است که پیامدهای مهمی در استفاده از آن دارد. مفهوم مالکیت به زبان راست کمک می‌کند ایمنی استفاده از حافظه<sup>2</sup> را تضمین کند، بدون اینکه نیازی به بازیافت کنندهٔ حافظه<sup>3</sup> داشته باشد.

---

<sup>1</sup> ownership

<sup>2</sup> memory safety guarantee

<sup>3</sup> garbage collector

## راست : مالکیت

- مالکیت در زبان راست به مجموعه قوانینی گفته می شود که تعیین می کنند مدیریت حافظه چگونه انجام می شود.
- برخی از زبان ها مانند جاوا یک مکانیزم بازیافت حافظه دارند که به طور منظم بررسی می کند به کدام قسمت های حافظه دیگر نیازی نیست و مکان های حافظه بی استفاده را برای استفاده مجدد آزادسازی و بازیافت می کند. در برخی از زبان های دیگر مانند سی ++ مدیریت حافظه باید توسط برنامه نویس به طور صریح انجام شود. مشکل زبان های دسته اول سرعت اجرای پایین آنهاست و مشکل زبان های دسته دوم پایین بودن قابلیت اطمینان برنامه های آنهاست چرا که برنامه نویس ممکن است به درستی حافظه را تخصیص و آزادسازی نکند.
- زبان راست راه حل سومی را پیشنهاد می کند. حافظه توسط سیستمی به نام سیستم مالکیت مدیریت می شود. این سیستم مالکیت قوانینی را تعریف می کند که در زمان کامپایل قابل بررسی هستند، پس کامپایلر می تواند اطمینان حاصل کند که این قوانین به درستی اعمال شده اند و در زمان اجرا به حافظه دسترسی ایمن وجود دارد. اگر یکی از قوانین مالکیت نقض شده باشد، برنامه راست کامپایل نمی شود. این مکانیزم سرعت اجرای برنامه را کاهش نمی دهد.



- مفهوم مالکیت، یک مفهوم جدید در زبان‌های برنامه نویسی است که توسط زبان راست ابداع شده است.
- در بسیاری از زبان‌های برنامه نویسی نیازی به فکر کردن به حافظه پشته و هیپ نیست، زیرا حافظه توسط زبان برنامه نویسی مدیریت می‌شود. در قوانین مالکیت زبان راست، برنامه نویس آگاهانه از پشته و هیپ استفاده می‌کند.

- حافظه پشته مناسب برای فراخوانی توابع است، زیرا وقتی یک تابع فراخوانی می‌شود، فقط به متغیرهای آن تابع می‌توان دسترسی پیدا کرد و به محض اتمام اجرای تابع، متغیرهای آن از روی برداشته می‌شوند و دیگر قابل دسترسی نخواهند بود.
- حافظه هیپ نظم خاصی ندارد. وقتی برنامه به حافظه پویا نیاز دارد، مقدار حافظه مورد نیاز را درخواست می‌کند و تخصیص دهنده حافظه<sup>1</sup> فضایی را در حافظه پیدا کرده و به اشاره‌گری به فضای تخصیص داده شده به حافظه می‌دهد.
- دسترسی به حافظه پشته سریع‌تر از دسترسی به هیپ است زیرا نیاز به جستجوی فضای خالی وجود ندارد.

---

<sup>1</sup> memory allocator

- قوانین مالکیت در راست به صورت زیر هستند :
  ۱. هر مقداری در زبان راست یک مالک<sup>1</sup> دارد.
  ۲. هر مقداری در هر لحظه فقط یک مالک دارد.
  ۳. وقتی از حوزه تعریف مالک خارج می شویم، مقدار آن از بین می رود.

---

<sup>1</sup> owner

- برای توضیح مفهوم مالکیت از نوع داده‌ای رشته استفاده می‌کنیم.
- یک رشته می‌تواند به صورت زیر ساخته و مورد استفاده قرار بگیرد.

---

```
۱ let mut s = String :: from ("hello") ;  
۲ s.push_str (" , world!"); //append  
۳ println! ("{}", s);
```

---

- اگر یک رشته را بدون استفاده از نوع String تعریف کنیم، در واقع رشته مورد نظر بر روی حافظه قرار نمی‌گیرد و درون فایل اجرایی (قسمت کد) قرار می‌گیرد، زیرا چنین رشته‌هایی یک بار مصرف هستند. برای مثال در قطعه برنامه زیر رشته در قسمت کد برنامه قرار می‌گیرد.

---

```
۱ let s = "hello" ;
```

---

## راست : مالکیت

- با استفاده از تابع `from` از نوع `String` رشته مورد نظر بر روی هیپ ساخته می‌شود. از آنجایی که رشته بر روی هیپ قرار می‌گیرد، اندازه آن به مقدار دلخواه می‌تواند افزایش پیدا کند.
- حافظه مورد نیاز رشته در هنگام نیاز در هیپ تخصیص داده شود و در هنگام عدم نیاز به فضاهای آزاد هیپ بازگردانده شود.
- تخصیص حافظه در تابع `from` برای نوع `String` انجام می‌شود، اما چگونه حافظه آزادسازی می‌شود؟
- در زبان‌هایی که از سازوکار بازیافت حافظه استفاده می‌کنند، بازیافت کننده به طور خودکار وقتی به حافظه نیاز نیست آن را آزاد می‌کند، اما در زبان‌هایی که بازیافت کننده حافظه وجود ندارد، این کار باید توسط برنامه نویس انجام شود. آزادسازی حافظه به طور دستی معمولاً کار سختی است. اگر آزادسازی حافظه فراموش شود با نشت حافظه مواجه می‌شویم که در بلند مدت منجر به پر شدن حافظه می‌شود. اگر آزادسازی حافظه زود انجام شود دسترسی‌های بعدی آن با خطای دسترسی (اشاره‌گر معلق) مواجه می‌شوند. اگر آزادسازی حافظه دوبار انجام شود، با خطای دسترسی (آزادسازی مجدد) مواجه می‌شویم.

## راست : مالکیت

- در زبان راست هرگاه از حوزه تعریف متغیری خارج شویم فضای حافظه آن آزاد می‌شود.
- بنابراین در خارج از آکولاد در کد زیر s غیر قابل دسترس است و فضای حافظه آن آزاد می‌شود.

```
۱ {  
۲     let s = String :: from ("hello") ;  
۳     // do stuff with s  
۴ } // the scope is over, and s and  
۵ // its memory are no longer valid
```

- در پایان حوزه تعریف، راست به طور خودکار تابع drop را فراخوانی و حافظه را به تخصیص دهنده حافظه پس می‌دهد.

– کد زیر را در نظر بگیرید :

---

```
۱ let x = 5 ;  
۲ let y = x ;
```

---

– متغیرهای نوع صحیح بر روی حافظه پشته ساخته می‌شود، بنابراین در واقع مقدار ۵ در متغیر x کپی و سپس مقدار x که ۵ است در y کپی می‌شود.



- حال کد زیر را در نظر بگیرید :

```
۱ let s1 = String :: from ("hello");  
۲ let s2 = s1;
```

- ممکن است انتظار داشته باشیم که به طور مشابه به مقدار s1 در s2 کپی شود، اما چون s1 به یک فضا در هیپ اشاره می‌کند این اتفاق نمی‌افتد.
- یک رشته از نوع String از سه بخش تشکیل شده است. اشاره‌گری به مکان حافظه در هیپ، اندازه رشته و ظرفیت رشته که اعداد صحیح هستند. این ساختار که شامل یک اشاره‌گر و دو مقدار است بر روی پشته قرار می‌گیرد، اما محتوای رشته بر روی هیپ قرار می‌گیرد.
- وقتی رشته s2 برابر با رشته s1 قرار می‌گیرد، در واقع آدرس اشاره‌گر و اندازه رشته متعلق به s1 در s2 کپی می‌شود. پس s2 به همان فضای هیپ اشاره می‌کند که s1 اشاره می‌کند.

- گفتیم وقتی از حوزه تعریف یک متغیر خارج می شویم آزادسازی حافظه انجام می شود، اما باید توجه داشت که اشاره گر متعلق به کدام متغیر است و آزادسازی حافظه برای کدام متغیر انجام می شود.
- وقتی می نویسیم `let s2 = s1` در واقع `s2` مالک فضای تخصیص داده شده در هیپ می شود و `s1` بی اعتبار می شود. بنابراین در کد زیر با پیام خطای کامپایلر مواجه می شویم:

---

```
۱ let s1 = String :: from ("hello");  
۲ let s2 = s1;  
۳ println! ("{} , world!", s1); // error
```

---

- در زبان‌های برنامه نویسی دیگر، وقتی مقدار یک اشاره‌گر کپی می‌شود می‌گوییم کپی سطحی<sup>1</sup> انجام شده است و هنگامی که مکان حافظه در هیپ برای اشاره‌گری کپی می‌شود می‌گوییم کپی عمیق<sup>2</sup> انجام شده است از آنجایی که در کپی سطحی در راست مقدار اول بی اعتبار می‌شود به آن عمل جابجایی<sup>3</sup> گفته می‌شود.
- پس در کد قبلی تنها s2 معتبر است و هرگاه از حوزه تعریف s2 خارج شویم، حافظه آزاد می‌شود.

---

<sup>1</sup> shallow copy

<sup>2</sup> deep copy

<sup>3</sup> move

- اما اگر بخواهیم مقدار یک رشته را کپی عمیق کنیم یعنی فضای جدیدی در حافظه هیپ تخصیص دهیم از تابع clone استفاده می‌کنیم.

---

```
۱ let s1 = String :: from ("hello");  
۲ let s2 = s1.clone ();  
۳ println! ("s1 = {} , s2 = {}", s1, s2);
```

---

## راست : مالکیت

- در فراخوانی تابع نیز وقتی یک اشاره‌گر به عنوان آرگومان به تابعی ارسال شود، تابع مالکیت متغیر را به دست می‌آورد، بنابراین با خارج شدن از تابع باید فضای آن آزاد شود.

```
1 fn main() {  
2     let s = String::from("hello");  
3     takes_ownership(s); // s's value moves into  
4                          // the function. and so is  
5                          // no longer valid.  
6 }  
7 fn takes_ownership(st: String) {  
8     println!("{}", st);  
9 } // st goes out of scope and drop is called.
```

- اگر بخواهیم به متغیر s بعد از فراخوانی تابع دسترسی پیدا کنیم، با خطای کامپایل مواجه می‌شویم. چنین پیام‌های خطا قابلیت اطمینان برنامه‌های راست را افزایش می‌دهد.

- همچنین بازگرداندن یک مقدار از یک تابع مالکیت را منتقل می‌کند.

---

```
۱ fn main() {  
۲     let s1 = String::from("hello");  
۳     let s2 = takes_given_ownership(s1);  
۴ } // drop is called for s2  
۵ fn takes_given_ownership(s: String) -> String {  
۶     s  
۷ } // s moves out of function
```

---

## راست : مالکیت

- حال ممکن است بخواهیم یک تابع را فراخوانی کنیم اما نخواهیم مالکیت متغیرهایی که به عنوان آرگومان ارسال می‌شوند را از دست بدهیم. راه حل اول این است که مالکیت را بعد از دست دادن، پس بگیریم:

```
۱ fn main() {  
۲     let s1 = String::from("hello");  
۳     let (s2, len) = calculate_length(s1);  
۴     println!("The length of '{}' is {}.", s2, len);  
۵ }  
۶ fn calculate_length(s: String) -> (String, usize) {  
۷     let length = s.len(); // len() returns the length of a String  
۸     (s, length)  
۹ }
```

- اما اگر بخواهیم همیشه این کار را انجام دهیم برنامه نویسی بسیار سخت می‌شود.
- خوشبختانه در راست مفهوم دیگری به نام متغیر ارجاعی وجود دارد که توسط آن می‌توان از انتقال مالکیت جلوگیری کرد.

## راست : متغیرهای ارجاعی

- یک متغیر ارجاعی در زبان راست، همانند متغیر ارجاعی در سی++ یک نام مستعار برای یک مکان حافظه است.
- وقتی یک متغیر ارجاعی به عنوان پارامتر به یک تابع ارسال شود، مالکیت منتقل نمی‌شود.
- در برنامه زیر طول یک رشته محاسبه می‌شود بدون اینکه مالکیت متغیر s1 منتقل شود.

```
۱ fn main() {  
۲     let s1 = String::from("hello");  
۳     let len = calculate_length(&s1);  
۴     println!("The length of '{}' is {}.", s1, len);  
۵ }  
۶ fn calculate_length(s: &String) -> usize {  
۷     s.len()  
۸ }
```



## راست : متغیرهای ارجاعی

- یک متغیر ارجاعی همانند سی++ با علامت امپرسند (&) تعریف می‌شود و همچنین آدرس یک متغیر با عملگر (&) به دست می‌آید.
- بنابراین &s1 به مکان حافظه s1 اشاره می‌کند و s در پارامتر تابع یک متغیر از نوع ارجاعی است. از آنجایی که یک متغیر ارجاعی هیچ مالکیتی بر داده‌ای که به آن اشاره می‌کند ندارد بنابراین وقتی از تابع خارج شویم، فضای حافظه‌ای که s به آن اشاره می‌کند آزاد نمی‌شود.
- به این عملیات قرض گرفتن<sup>1</sup> گفته می‌شود، چرا که یک متغیر ارجاعی مالکیت حافظه را به دست نمی‌آورد، اما آن را برای استفاده برای مدت زمانی قرض می‌گیرد. در واقع در عملیات قرض گرفتن متغیر ارجاعی مالکیت موقت به دست می‌آورد و پس از اتمام کار خود مالکیت را پس می‌دهد.
- همانطور که مقدار یک متغیر غیر قابل تغییر را نمی‌توان تغییر داد، متغیر ارجاعی نیز که به یک متغیر غیر قابل تغییر اشاره می‌کند، قابل تغییر نیست.

---

<sup>1</sup> borrowing

## راست : متغیرهای ارجاعی

- یک متغیر ارجاعی را می‌توان قابل تغییر<sup>1</sup> تعریف کرد. یک متغیر ارجاعی قابل تغییر به یک متغیر قابل تغییر اشاره می‌کند و مقدار را می‌توان تغییر داد.
- برای مثال :

```
۱ fn main() {  
۲     let mut s = String::from("hello");  
۳     change(&mut s);  
۴ }  
۵ fn change(some_string: &mut String) {  
۶     some_string.push_str(", world");  
۷ }
```

---

<sup>1</sup> mutable reference

## راست : متغیرهای ارجاعی

- یک متغیر قابل تغییر را در یک زمان فقط به یک متغیر ارجاعی قابل تغییر می توان قرض داد. اگر یک متغیر به بیش از دو متغیر ارجاعی قابل تغییر قرض داده شود، کامپایلر پیام خطا صادر می کند.
- بنابراین کامپایل برنامه زیر پیام خطا صادر می کند.

```
۱ let mut s = String::from("hello");  
۲ let r1 = &mut s;  
۳ let r2 = &mut s; // error  
۴ println!("{}", {}, r1, r2);
```

## راست : متغیرهای ارجاعی

- در واقع در این کد خواسته‌ایم یک متغیر قابل تغییر را در یک زمان به دو متغیر ارجاعی قرض دهیم، که این عملیات در زبان راست ممنوع است.
- این محدودیت به کامپایلر کمک می‌کند که از وضعیت رقابت داده<sup>1</sup> در زمان کامپایل جلوگیری کند. اگر دو یا چند متغیر ارجاعی بتوانند به طور همزمان به یک داده دسترسی داشته باشند ممکن است هر دو به طور همزمان داده را تغییر دهند و رفتار سیستم غیر قابل پیش بینی می‌شود و پیدا کردن خطای خروجی بسیار مشکل می‌شود. راست از این خطاهای احتمالی در زمان کامپایل جلوگیری می‌کند.
- البته اجازه داریم یک متغیر ارجاعی قابل تغییر را در یک بلوک جداگانه تعریف کنیم

```
۱ let mut s = String::from("hello");  
۲ {  
۳     let r1 = &mut s;  
۴ } // r1 goes out of scope here, so we can make a new reference  
۵     //with no problems.  
۶ let r2 = &mut s;
```

---

<sup>1</sup> data race

## راست : متغیرهای ارجاعی

- چند متغیر ارجاعی غیر قابل تغییر به طور همزمان می‌توانند برای اشاره به یک داده تعریف شوند، ولی متغیر ارجاعی قابل تغییر حتی با متغیر ارجاعی غیر قابل تغییر هم نمی‌تواند به طور همزمان تعریف شود.

```
۱ let mut s = String::from("hello");  
۲ let r1 = &s; // no problem  
۳ let r2 = &s; // no problem  
۴ let r3 = &mut s; // error  
۵ println!("{}", r1, r2, r3);
```

## راست : متغیرهای ارجاعی

- دلیل این امر این است که متغیرهای ارجاعی غیر قابل تغییر انتظار ندارند مقداری که به آن اشاره می‌کنند ناگهان تغییر کند، اما چند متغیر ارجاعی غیر قابل تغییر به طور همزمان می‌توانند تعریف شوند چون هیچ‌کدام مقدار داده را تغییر نمی‌دهند.

## راست : متغیرهای ارجاعی

- دقت کنید که حوزه تعریف یک متغیر ارجاعی از زمانی است که تعریف می‌شود تا آخرین زمانی که از آن استفاده می‌شود. بنابراین اگر یک متغیر ارجاعی تعریف و سپس استفاده شود، یک متغیر ارجاعی قابل تغییر بعد از آن می‌تواند تعریف شود و به همان داده متغیر قبلی اشاره کند.
- بنابراین کد زیر بدون خطا کامپایل می‌شود.

```
۱ let mut s = String::from("hello");
۲ let r1 = &s; // no problem
۳ let r2 = &s; // no problem
۴ println!("{}", r1, r2);
۵ // variables r1 and r2 will not be used after this point
۶ let r3 = &mut s; // no problem
۷ println!("{}", r3);
```

## راست : متغیرهای ارجاعی

- قوانین مالکیت و قرض گرفتن ممکن است کمی پیچیده به نظر برسند، اما این قوانین کمک می‌کنند که کامپایلر خطاهای احتمالی را در زمان کامپایل پیدا کرده و از بروز آنها جلوگیری کند. بدون این قوانین ممکن است برنامه به راحتی کامپایل شود ولی پیدا کردن خطا به طور دستی توسط برنامه نویس می‌تواند بسیار پیچیده و دشوار شود.



## راست : متغیرهای ارجاعی

- در زبان‌هایی که اشاره‌گر در آنها وجود دارد، ممکن است به راحتی خطای اشاره‌گر معلق<sup>1</sup> به وجود بیاید، بدین معنی که یک اشاره‌گر به مکانی در حافظه اشاره کند که توسط یک متغیر دیگر آزاد شده باشد.
- اما در زبان راست کامپایلر اطمینان حاصل می‌کند که هیچ‌گاه متغیر ارجاعی معلق به وجود نمی‌آید. اگر متغیر ارجاعی به یک متغیر وجود داشته باشد، کامپایلر اطمینان حاصل می‌کند که متغیر مربوطه قبل از متغیر ارجاعی از حوزه تعریف خارج نمی‌شود و فضای حافظه آن آزاد نمی‌شود.

---

<sup>1</sup> dangling pointer

## راست : متغیرهای ارجاعی

- کد زیر را در نظر بگیرید. وقتی از تابع خارج می‌شویم متغیر s از بین می‌رود، اما برنامه نویس متغیر ارجاعی از آن بازگردانده است که کامپایلر پیام خطا صادر می‌کند.

---

```
۱ fn main() {  
۲     let reference_to_nothing = dangle();  
۳ }  
۴ fn dangle() -> &String { // dangle returns a reference to a String  
۵     let s = String::from("hello"); // s is a new String  
۶     &s // error: we return a reference to the String, s  
۷ } // Here, s goes out of scope, and is dropped. Its memory goes away.  
۸     // Danger!
```

---

## راست : متغیرهای ارجاعی

- برنامه را می‌توانیم به صورت زیر تصحیح کنیم. مالکیت s از درون تابع به بیرون انتقال پیدا می‌کند.

```
۱ fn no_dangle() -> String {  
۲     let s = String::from("hello");  
۳     s  
۴ }
```

- بنابراین در هر زمان، یا فقط یک متغیر ارجاعی قابل تغییر می‌تواند به یک داده اشاره کند و یا تعدادی متغیر ارجاعی غیر قابل تغییر.

## راست : نوع برش

- برش<sup>1</sup> کمک می‌کند یک متغیر ارجاعی به دنباله‌ای از عناصر در یک مجموعه بسازیم. از آنجایی که برش یک اشاره‌گر است، مالکیت به داده ندارد. برای مثال با استفاده از برش می‌توانیم یک قسمت از یک رشته را توسط یک تابع بازگردانیم.
- فرض کنید می‌خواهیم اولین کلمه از یک رشته را که با خط فاصله از کلمه دوم جدا شده است به دست آوریم، می‌توانیم تابع زیر را بنویسیم :

```
۱ fn first_word(s: &String) -> usize {  
۲     for (i, item) in s.chars().enumerate() {  
۳         if item == ' ' {  
۴             return i;  
۵         }  
۶     }  
۷     s.len()  
۸ }
```

---

<sup>1</sup> slice

## راست : نوع برش

- خروجی تابع، اندیس اولین خط فاصله در رشته ورودی است. حال فرض کنید پس از یافتن اولین خط فاصله، رشته را به یک رشته تهی تبدیل کنیم. متغیری که به اندیس اولین خط فاصله اشاره می‌کند، اکنون غیر معتبر است.

```
۱ fn main() {  
۲     let mut s = String::from("hello world");  
۳     let word = first_word(&s); // word will get the value 5  
۴     s.clear(); // this empties the String, making it equal to ""  
۵     // word still has the value 5 here,  
۶     // but there's no more string that  
۷     // we could meaningfully use the value 5 with.  
۸     // word is now totally invalid!  
۹ }
```

## راست : نوع برش

- می‌خواهیم برنامه را به گونه‌ای بنویسیم که در زمان کامپایل این خطا تشخیص داده شود.
- برای این کار از نوع برش استفاده می‌کنیم.

## راست : نوع برش

- یک برش از یک آرایه یا رشته به صورت زیر تعریف می‌شود.

---

```
۱ let s = String::from("hello world");  
۲ let hello = &s[0..5];  
۳ let world = &s[6..11];  
۴ let slice = &s[0..2];  
۵ let slice = &s[..2];  
۶ let len = s.len();  
۷ let slice = &s[3..len];  
۸ let slice = &s[3..];  
۹ let slice = &s[0..len];  
۱۰ let slice = &s[..];
```

---

- حال تابع first\_word را با استفاده از برش به صورت زیر تعریف می‌کنیم.

---

```
۱ fn first_word(s: &String) -> &str {  
۲     for (i, item) in s.chars().enumerate() {  
۳         if item == ' ' {  
۴             return &s[0..i];  
۵         }  
۶     }  
۷     &s[..]  
۸ }
```

---



## راست : نوع برش

- کامپایلر اطمینان حاصل می‌کند که همه متغیرهای ارجاعی به یک متغیر معتبر می‌مانند. پس اگر سعی کنیم این بار رشته را به یک رشته تهی تبدیل کنیم، از آنجایی که یک متغیر ارجاعی داریم که به یک برش از رشته اشاره می‌کند، کامپایلر پیام خطا صادر می‌کند.

```
۱ fn main() {  
۲     let mut s = String::from("hello world");  
۳     let word = first_word(&s);  
۴     s.clear(); // error!  
۵     println!("the first word is: {}", word);  
۶ }
```

## راست : نوع برش

- در واقع در اینجا یکی از قوانین قرض گرفتن به کمک ما می آید. به یاد داریم وقتی که یک متغیر ارجاعی به یک متغیر داریم نمی توانیم یک متغیر ارجاعی قابل تغییر تعریف کنیم. از آنجایی که تابع `clear()` می خواهد رشته را تهی کند باید یک متغیر ارجاعی قابل تغییر از آن بگیرد و این کار امکان پذیر نیست چرا که متغیر `word` یک متغیر ارجاعی به متغیر `s` است. در واقع پیاده سازی تابع `clear()` به صورت `fn clear(&mut self)` است.
- بنابراین کامپایلر راست به ما کمک کرد که از یک خطای احتمالی در زمان اجرا جلوگیری کنیم.

## راست : نوع برش

- وقتی یک رشته را به صورت خام تعریف می‌کنیم، رشته در داخل کد قرار می‌گیرد و بنابراین نوع آن از نوع برش است، زیرا رشته تعریف شده یک برش از کد است.

---

```
\ let s = "Hello, world!";
```

---

- در اینجا متغیر s از نوع &str است که یک برش از یک رشته است. در واقع &str یک متغیر ارجاعی غیر قابل تغییر است.

## راست : نوع برش

- می‌توانستیم ورودی تابع `first_word` را از نوع `&str` در نظر بگیریم تا بتوانیم از برش‌ها نیز برش به دست آوریم.

---

```
\ fn first_word(s:&str) -> &str {
```

---

## راست : نوع برش

- از آرایه‌ها نیز می‌توانیم به صورت زیر برش تهیه کنیم.

---

```
۱ let a = [1,2,3,4,5];  
۲ let slice = &a[1..3];
```

---

## راست : نوع برش

- به طور خلاصه، با استفاده از مفاهیم مالکیت، قرض دادن و برش می‌توان در زمان کامپایل اطمینان حاصل کرد که برنامه‌های راست به طور امن از حافظه استفاده می‌کنند و برنامه در زمان اجرا با دسترسی غیر مجاز مواجهه نمی‌شود.

- یک ساختمان را در زبان راست می‌توان به صورت زیر تعریف کرد.

---

```
۱ struct User {  
۲     active: bool,  
۳     username: String,  
۴     email: String,  
۵     sign_in_count: u64,  
۶ }
```

---

- تفاوت نوع چندتایی و نوع ساختمان در این است که به اعضای ساختمان می‌توان با نام دسترسی پیدا کرد.

- همچنین برای تعریف یک ساختمان می‌توان با استفاده از نام اعضای ساختمان آن را مقداردهی اولیه کرد.

---

```
۱ fn main() {  
۲     let mut user1 = User {  
۳         active: true,  
۴         username: String::from("someusername123"),  
۵         email: String::from("someone@example.com"),  
۶         sign_in_count: 1,  
۷     };  
۸     user1.email = String::from("anotheremail@example.com");  
۹ }
```

---

- یک نمونه از یک ساختمان می‌تواند قابل تغییر یا غیر قابل تغییر باشد.



- یک نمونه از یک ساختمان در یک تابع توسط پارامترهای تابع می‌تواند به صورت زیر ساخته شود.

```
۱ fn build_user(email: String, username: String) -> User {  
۲     User {  
۳         active: true,  
۴         username: username,  
۵         email: email,  
۶         sign_in_count: 1,  
۷     }  
۸ }
```

- یک میانبر نیز برای مقداردهی اولیه نمونه ساختمان به صورت زیر وجود دارد.

```
۱ fn build_user(email: String, username: String) -> User {  
۲     User {  
۳         active: true,  
۴         username,  
۵         email,  
۶         sign_in_count: 1,  
۷     }  
۸ }
```

- یک نمونه از ساختمان را می‌توان با استفاده از مقادیر یک نمونه دیگر از ساختمان به صورت زیر ساخت :

---

```
۱ let user2 = User {  
۲     active: user1.active,  
۳     username: user1.username,  
۴     email: String::from("another@example.com"),  
۵     sign_in_count: user1.sign_in_count,  
۶ };
```

---

- یک میانبر نیز برای ساختن یک نمونه با استفاده از مقادیر یک نمونه دیگر به صورت زیر وجود دارد :

---

```
۱ let user2 = User {  
۲   email: String::from("another@example.com"),  
۳   ..user1  
۴ };
```

---

## راست : ساختمان

- در راست می‌توان ساختمان‌ها را بدون ذکر نام عناصر نیز ایجاد کرد. این نوع ساختمان‌ها وقتی استفاده می‌شوند که می‌خواهیم یک چندتایی تعریف کنیم که دارای یک نام معین باشد.

```
۱ struct Color(i32, i32, i32);  
۲ struct Point(i32, i32, i32);  
۳ fn main() {  
۴     let black = Color(0, 0, 0);  
۵     let origin = Point(0, 0, 0);  
۶     let p = Point(10,20,30);  
۷     let x = p.0; // x=10  
۸     let y = p.1; // y=20  
۹     let z = p.2; // z=30  
۱۰ }
```

- یک ساختمان را می‌توان بدون عضو نیز تعریف کرد. در آینده خواهیم دید چگونه برای یک ساختمان رفتار تعریف می‌کنیم.

---

```
۱ struct AlwaysEqual;  
۲ fn main() {  
۳     let subject = AlwaysEqual;  
۴ }
```

---

## راست : ساختمان

- یک ساختمان می‌تواند علاوه بر اعضای داده‌ای تعدادی تابع عضو نیز داشته باشد که به آنها متود<sup>1</sup> گفته می‌شود. متودهای یک ساختمان به صورت زیر تعریف می‌شوند.

```
۱  #[derive(Debug)]
۲  struct Rectangle {
۳      width: u32,
۴      height: u32,
۵  }
۶  impl Rectangle {
۷      fn area(&self) -> u32 {
۸          self.width * self.height
۹      }
۱۰ }
```

---

<sup>1</sup> method

```
۱ fn main() {  
۲     let rect1 = Rectangle {  
۳         width: 30,  
۴         height: 50,  
۵     };  
۶     println!(  
۷         "The area of the rectangle is {} square pixels.",  
۸         rect1.area()  
۹     );  
۱۰ }
```



## راست : ساختمان

- یک متود می‌تواند با یک فیلد ساختمان همنام باشد.

```
1  impl Rectangle {
2      fn width(&self) -> bool {
3          self.width > 0
4      }
5  }
6  fn main() {
7      let rect1 = Rectangle {
8          width: 30,
9          height: 50,
10     };
11     if rect1.width() {
12         println!("The rectangle has a nonzero width; it is {}",
13                 rect1.width);
14     }
15 }
```

- یک متود می‌تواند پارامتر نیز داشته باشد.

```
۱ impl Rectangle {  
۲     fn area(&self) -> u32 {  
۳         self.width * self.height  
۴     }  
۵     fn can_hold(&self, other: &Rectangle) -> bool {  
۶         self.width > other.width && self.height > other.height  
۷     }  
۸ }
```

```
۱ fn main() {  
۲     let rect1 = Rectangle {  
۳         width: 30,  
۴         height: 50,  
۵     };  
۶     let rect2 = Rectangle {  
۷         width: 10,  
۸         height: 40,  
۹     };  
۱۰    let rect3 = Rectangle {  
۱۱        width: 60,  
۱۲        height: 45,  
۱۳    };  
۱۴    println!("Can rect1 hold rect2? {}", rect1.can_hold(&rect2));  
۱۵    println!("Can rect1 hold rect3? {}", rect1.can_hold(&rect3));  
۱۶ }
```

## راست : ساختمان

- توابع یک ساختمان می‌توانند به عنوان ورودی پارامتر `self` نداشته باشند. این توابع را توابع مرتبط<sup>1</sup> با ساختمان می‌نامیم. این توابع متعلق به یک ساختمان هستند، برخلاف متودها که متعلق به نمونه‌های ساختمان هستند. اگر بخواهیم در یک تابع مرتبط با یک ساختمان، نوع ساختمان را بازگردانیم از کلمه `Self` استفاده می‌کنیم.

```
۱ impl Rectangle {  
۲     fn square(size: u32) -> Self {  
۳         Self {  
۴             width: size,  
۵             height: size,  
۶         }  
۷     }  
۸ }
```

---

<sup>1</sup> associated function

- به توابع مرتبط می‌توان توسط عملگر (::) دسترسی پیدا کرد، به طور مثال `square(3) :: Rectangle` یک نمونه از ساختمان مربع به طول ضلع ۳ باز می‌گرداند.

- متودها و توابع مرتبط با یک ساختمان را می‌توان در چند قطعه جدا نیز تعریف کرد.

---

```
۱ impl Rectangle {  
۲     fn area(&self) -> u32 {  
۳         self.width * self.height  
۴     }  
۵ }  
۶ impl Rectangle {  
۷     fn can_hold(&self, other: &Rectangle) -> bool {  
۸         self.width > other.width && self.height > other.height  
۹     }  
۱۰ }
```

---

## راست : نوع شمارشی

- نوع داده شمارشی در راست با کلمه `enum` تعریف می‌شود. از نوع داده شمارشی برای نام‌گذاری مجموعه‌ای از مقادیر استفاده می‌شود.

- برای مثال :

```
۱ enum IpAddrKind {  
۲     V4,  
۳     V6,  
۴ }  
۵ struct IpAddr {  
۶     kind: IpAddrKind,  
۷     address: String,  
۸ }  
۹ let home = IpAddr {  
۱۰     kind: IpAddrKind::V4,  
۱۱     address: String::from("127.0.0.1"),  
۱۲ };
```

## راست : نوع شمارشی

- اما گاهی نیاز داریم در عناصر نوع داده شمارشی، مقداری نیز قرار دهیم. بدین ترتیب نیاز نداریم هر بار در کنار نوع داده شمارشی در یک ساختمان نوع دیگری نیز تعریف کنیم و مقادیر مورد نیاز را در ساختمان قرار دهیم.
- در راست نوع داده شمارشی می‌تواند مقداری نیز در خود ذخیره کند. برای مثال :

```
۱ enum IpAddr {  
۲     V4(String),  
۳     V6(String),  
۴ }  
۵ let home = IpAddr::V4(String::from("127.0.0.1"));
```



## راست : نوع شمارشی

- نوع داده شمارشی زیر معادل تعریف چهار ساختمان متفاوت است، با این تفاوت که نوع داده شمارشی همه ساختمان‌های همانند را در یک گروه قرار می‌دهد.

---

```
۱ enum Message {
۲     Quit,
۳     Move { x: i32, y: i32 },
۴     Write(String),
۵     ChangeColor(i32, i32, i32),
۶ }
۷ // it is equivalent to :
۸ struct QuitMessage; // unit struct
۹ struct MoveMessage {
۱۰     x: i32,
۱۱     y: i32,
۱۲ }
۱۳ struct WriteMessage(String); // tuple struct
۱۴ struct ChangeColorMessage(i32, i32, i32); // tuple struct
```

---

- مشکل تعریف چند ساختمان در مثال قبل این است که اگر بخواهیم تابعی تعریف کنیم که با همه این ساختمان‌ها رفتار مشابهی انجام دهد، امکان آن وجود ندارد و باید یک تابع به ازای هر یک ساختمان‌ها تعریف کنیم.

## راست : نوع شمارشی

- برای داده‌های شمارشی همانند ساختمان‌ها می‌توانیم متود تعریف کنیم.

```
۱ enum Message {  
۲     Quit,  
۳     Move { x: i32, y: i32 },  
۴     Write(String),  
۵     ChangeColor(i32, i32, i32),  
۶ }  
۷ impl Message {  
۸     fn call(&self) {  
۹         // method body would be defined here  
۱۰    }  
۱۱ }  
۱۲ let m = Message::Write(String::from("hello"));  
۱۳ m.call();
```

## راست : نوع شمارشی

- یک نوع داده شمارشی که توسط کتابخانه استاندارد تعریف شده است، نوع Option یا انتخاب است. این نوع داده یک سناریوی بسیار پر کاربرد دارد. یک متغیر در بسیاری از مواقع یا مقداری دارد که متناسب با نوع متغیر است یا هیچ مقداری ندارد.
- برای مثال می‌خواهید عنصر اول یک لیست را در یک متغیر ذخیره کنید. این متغیر صحیح یا مقداری می‌گیرد و یا اگر لیست خالی باشد هیچ مقداری نمی‌گیرد.
- در زبان‌های دیگر همه این حالات باید توسط برنامه نویس بررسی شوند ولی در زبان راست کامپایلر اطمینان حاصل می‌کند که همه حالات بررسی شده‌اند و در غیر اینصورت پیام خطا صادر می‌کند.

## راست : نوع شمارشی

- در زبان راست همچون بسیاری زبان‌های دیگر مقدار `null` وجود ندارد، چرا که زبان‌های قبلی وظیفه برنامه نویس بود که بررسی کند آیا مقداری `null` است یا خیر. در زبان راست از نوع داده‌ای `Option` استفاده می‌شود و بدین صورت در زمان کامپایل اطمینان حاصل می‌شود که همه حالات بررسی شده‌اند.
- مشکل مقدار `null` این است که اگر مقدار آن را بدون بررسی به عنوان یک مقدار غیر تهی استفاده کنیم با خطای زمان اجرا مواجه می‌شویم.

- نوع داده‌ای Option به صورت زیر تعریف و به کار برده می‌شود.

```
۱ enum Option<T> {  
۲     None ,  
۳     Some(T) ,  
۴ }  
۵ let some_number = Some(5);  
۶ let some_char = Some('e');  
۷ let absent_number: Option<i32> = None;
```

## راست : نوع شمارشی

- در زبان‌هایی مانند سی و جاوا، معمولاً وقتی یک تابع نیاز به بازگرداندن مقدار تهی داشت، یک عدد قراردادی به عنوان مقدار تهی بازمی‌گردانیدیم.

```
۱ fn find(s : &String, c : char) -> i32 {  
۲     let mut i = 0;  
۳     for ch in s.chars() {  
۴         if ch == c {  
۵             return i;  
۶         }  
۷         i+=1;  
۸     }  
۹     -1  
۱۰ }
```

- مشکلی که این روش ایجاد می‌کند این است که برنامه‌نویس باید از تابع `find` آگاهانه استفاده کند، بدین معنی که همیشه بررسی کند مقدار بازگردانده شده برابر با `-1` نباشد و در غیر اینصورت پیام خطای مناسب تولید کند.

## راست : نوع شمارشی

- در زبان راست می‌توانیم از نوع Option استفاده کنیم، بدین معنی که مقدار بازگردانده شده می‌تواند تهی (None) نیز باشد.

```
۱ fn find(s : &String, c : char) -> Option<i32> {  
۲     let mut i = 0;  
۳     for ch in s.chars() {  
۴         if ch == c {  
۵             return Some(i);  
۶         }  
۷         i+=1;  
۸     }  
۹     None  
۱۰ }
```

- حال کامپایلر راست تنها در صورتی برنامه را کامپایل می‌کند که همه حالت‌های ممکن پس از دریافت مقدار تابع find بررسی شده باشند.



```
1 struct Point {
2     x : i32,
3     y : i32
4 }
5 impl Point {
6     fn move_point(&mut self, h : i32, v : i32) {
7         self.x+=h;
8         self.y+=v;
9     }
10 }
11 fn move_up(p : Point, y : i32) {
12     p.move_point(0,y); // p must have a value and cannot be null
13 }
14 fn move_up(p : Option<Point>, y : i32) {
15     p.move_point(0,y); // error : all cases must be handled
16 }
```

## راست : نوع شمارشی

- عبارت match یک ساختار کنترلی در زبان راست برای تطبیق مقدار یک نوع داده شمارشی است. با استفاده از این ساختار کنترلی می‌توانیم همهٔ حالت‌های یک نوع دادهٔ شمارشی را بررسی کنیم.

## راست : ساختار کنترلی تطابق

- در زبان راست یک ساختار کنترلی به نام تطابق یا match وجود دارد که به برنامه نویس کمک می‌کند یک مقدار را با چند الگوی متعدد مقایسه کند و سپس بر اساس الگوی تطبیق داده شده، دستورات مناسب را اجرا کند.
- کامپایلر اطمینان حاصل می‌کند که همه الگوهای ممکن بررسی شده‌اند و بنابراین اگر برنامه نویس فراموش کند تعدادی از حالات را بررسی کند، با خطای کامپایلر مواجه می‌شود.

## راست : ساختار کنترلی تطابق

- برای مثال فرض کنید یک نوع داده شمارشی داریم که همه سکه‌های پولی موجود را نگهداری می‌کند. حال می‌خواهیم تابعی بنویسیم که ارزش یک سکه را برگرداند. در این تابع نیاز داریم همه حالات را بررسی کنیم، پس می‌توانیم به صورت زیر از match استفاده کنیم.

```
۱ enum Coin {  
۲     Penny,  
۳     Nickel,  
۴     Dime,  
۵     Quarter,  
۶ }  
۷ fn value_in_cents(coin: Coin) -> u8 {  
۸     match coin {  
۹         Coin::Penny => 1,  
۱۰        Coin::Nickel => 5,  
۱۱        Coin::Dime => 10,  
۱۲        Coin::Quarter => 25,  
۱۳    }  
۱۴ }
```

## راست : ساختار کنترلی تطابق

- ساختار کنترلی match شباهت زیادی با if دارد، با این تفاوت که در if یک شرط منطقی بررسی می‌شود اما در اینجا تطابق یک متغیر با مقدار بررسی می‌شود.
- در یک بلوک تطابق چند شاخه<sup>1</sup> وجود دارد. هر شاخه از دو بخش تشکیل شده است. بخش اول الگو و بخش دوم کد عملیاتی است و این دو بخش با علامت => از یکدیگر جدا می‌شوند.

---

<sup>1</sup> arm

## راست : ساختار کنترلی تطابق

- در قسمت کد عملیاتی اگر چندین دستور وجود داشته باشند، از آکولاد استفاده می‌کنیم.

```
۱ fn value_in_cents(coin: Coin) -> u8 {  
۲     match coin {  
۳         Coin::Penny => {  
۴             println!("Lucky penny!");  
۵             1  
۶         }  
۷         Coin::Nickel => 5,  
۸         Coin::Dime => 10,  
۹         Coin::Quarter => 25,  
۱۰     }  
۱۱ }
```

## راست : ساختار کنترلی تطابق

- همچنین چنانکه گفتیم هر یک از اعضای یک نوع داده شمارشی می توانند مقدار نیز داشته باشند، پس می توانیم ساختار تطابق را به صورت زیر نیز بنویسیم.

---

```
۱  #[derive(Debug)] // so we can inspect the state in a minute
۲  enum UsState {
۳      Alabama,
۴      Alaska,
۵      // --snip--
۶  }
۷  enum Coin {
۸      Penny,
۹      Nickel,
۱۰     Dime,
۱۱     Quarter(UsState),
۱۲ }
```

---



```
1 fn value_in_cents(coin: Coin) -> u8 {  
2     match coin {  
3         Coin::Penny => 1,  
4         Coin::Nickel => 5,  
5         Coin::Dime => 10,  
6         Coin::Quarter(state) => {  
7             println!("State quarter from {:?}!", state);  
8             25  
9         }  
10     }  
11 }
```

## راست : ساختار کنترلی تطابق

- پس با استفاده از نوع داده انتخاب یا Option و ساختار کنترلی تطابق یا match می‌توانیم حالات مختلف یک مقدار که می‌تواند تهی باشد را بررسی کنیم.

```
۱ fn plus_one(x: Option<i32>) -> Option<i32> {  
۲     match x {  
۳         None => None,  
۴         Some(i) => Some(i + 1),  
۵     }  
۶ }  
۷ let five = Some(5);  
۸ let six = plus_one(five);  
۹ let none = plus_one(None);
```

## راست : ساختار کنترلی تطابق

- همانطور که اشاره شد همه شاخه‌های ممکن در یک الگو باید بررسی شوند، در غیر این صورت عبارت تطابق با خطای کامپایل مواجه می‌شود.

```
۱ fn plus_one(x: Option<i32>) -> Option<i32> {  
۲     match x {  
۳         Some(i) => Some(i + 1),  
۴     } // error  
۵ }
```

## راست : ساختار کنترلی تطابق

- در برخی مواقع پس از این که چند الگو را بررسی کردیم می‌خواهیم با بقیه الگوها به طور مشابه رفتار کنیم. در این مواقع از کلمه `other` استفاده می‌کنیم.

---

```
۱ let dice_roll = 9;
۲ match dice_roll {
۳     3 => add_fancy_hat(),
۴     7 => remove_fancy_hat(),
۵     other => move_player(other),
۶ }
۷ fn add_fancy_hat() {}
۸ fn remove_fancy_hat() {}
۹ fn move_player(num_spaces: u8) {}
```

---

## راست : ساختار کنترلی تطابق

- همچنین در برخی مواقع با بقیه الگوها می‌خواهیم مشابه رفتار کنیم اما نیازی به مقدار آن الگوها نداریم. در چنین مواقعی از زیر خط (`_`) استفاده می‌کنیم.

---

```
۱ let dice_roll = 9;
۲ match dice_roll {
۳     3 => add_fancy_hat(),
۴     7 => remove_fancy_hat(),
۵     _ => reroll(),
۶ }
۷ fn add_fancy_hat() {}
۸ fn remove_fancy_hat() {}
۹ fn reroll() {}
```

---

## راست : ساختار کنترلی تطابق

- و در نهایت گاهی بر روی مابقی الگوها نمی‌خواهیم هیچ عملیاتی انجام دهیم.

---

```
۱ let dice_roll = 9;
۲ match dice_roll {
۳     3 => add_fancy_hat(),
۴     7 => remove_fancy_hat(),
۵     _ => (),
۶ }
۷ fn add_fancy_hat() {}
۸ fn remove_fancy_hat() {}
```

---

## راست : ساختار کنترلی تطابق

- در بسیاری مواقع بررسی کردن حالت‌های باقیمانده که عملیاتی نمی‌خواهیم بر روی آنها انجام دهیم، برنامه بسیار شلوغ می‌کند. یک ساختار میانبر به نام `if let` برای چنین مواقعی وجود دارد.

```
۱ let config_max = Some(3);
۲ match config_max {
۳     Some(max) => println!("The maximum is configured to be {}", max),
۴     _ => (),
۵ }
۶ // it is equivalent to :
۷ let config_max = Some(3);
۸ if let Some(max) = config_max {
۹     println!("The maximum is configured to be {}", max);
۱۰ }
```

## راست : ساختار کنترلی تطابق

- از ساختار `if let` به صورت زیر می توان استفاده کرد.

```
۱ let mut count = 0;
۲ match coin {
۳     Coin::Quarter(state) => println!("State quarter from {:?}!", state)
۴     _ => count += 1,
۵ }
۶ // it is equivalent to :
۷ let mut count = 0;
۸ if let Coin::Quarter(state) = coin {
۹     println!("State quarter from {:?}!", state);
۱۰ } else {
۱۱     count += 1;
۱۲ }
```



```
۱ fn find(s : &String, c : char) -> Option<i32> {
۲     let mut i = 0;
۳     for ch in s.chars() {
۴         if ch == c { return Some(i); }
۵         i+=1;
۶     }
۷     None
۸ }
۹ fn main() {
۱۰     let s = String::from("hello");
۱۱     let i = find(&s, 'l');
۱۲     match i {
۱۳         None => println!("not found"),
۱۴         Some(i) => println!("{}", i)
۱۵     }
۱۶ }
```

```
1 fn move_up(op : Option<Point>, y : i32) {
2     match op {
3         None => println!("input point is null"),
4         Some(mut p) => p.move_point(0,y)
5     }
6 }
7 fn main() {
8     let p = Some(Point {x:2, y:3});
9     move_up(p,10);
10    let pn = None;
11    move_up(pn,10);
12 }
```