به نام خدا

زبان‌های برنامه‌نویسی

آرش شفیعی

متغیرها و نوع‌های داده‌ای

نام‌ها

- 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.

نام‌ها

- یک متغیر در واقع یک مفهوم انتزاعی [1] برای تعدادی از خانه‌های حافظه است. در برخی مواقع متغیر دقیقا همان مقداری را دارد که در خانهٔ 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 [2] و طول عمر [3] آن اشاره کرد.

[1] abstraction
[2] scope
[3] lifetime

نام‌ها

- یک نام [1] یا شناسه [2] 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 [3] به بزرگ و کوچک‌اند.
- معمولا کلمات کلیدی زبان را نمی‌توان به عنوان نام انتخاب کرد. زبان کوبول تعداد کلمات کلیدی زیادی دارد مانند LENGTH و COUNT که منجر به محدودیت برای برنامه نویس می‌شود.
- در زبان سی ++، برای نام‌ها می‌توان فضای نام [4] تعریف کرد که باعث می‌شود یک نام را بتوان در چند مکان متفاوت به کار برد.

---

[1] name
[2] identifier
[3] case sensitive
[4] name space

متغیرها

- یک متغیر را می‌توان با چند ویژگی مشخص کرد : نام، آدرس، مقدار، نوع، طول عمر و حوزهٔ 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. مثلا متغیر محلی در یک تابع در هر بار فراخوانی یک آدرس جدید می‌گیرد.

متغیرها

- آدرس متغیر گاهی مقدار چپ [1] 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 [2] گفته می‌شود. در زبان سی++، با استفاده از اشاره‌گرها و متغیرهای مرجع می‌توان نام مستعار تعریف کرد.
- یک متغیر همچنین دارای یک نوع [3] 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 [4] می‌نامیم زیرا هر گاه متغیر در سمت راست قرار گیرد به مقدار آن نیاز داریم.

[1] l-value
[2] alias
[3] type
[4] r-value

انقیاد

- انقیاد [1] 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 [2] می‌گویند.
- برای مثال نماد ستاره (*) در زمان طراحی زبان [3] به عمل ضرب مقید شده است.
  نوع یک متغیر به مقادیر ممکن آن زمان پیاده سازی زبان [4] مقید شده است.
  یک متغیر به نوع آن در زبان جاوا در زمان کامپایل [5] مقید شده است.
  و در نهایت یک متغیر به سلول حافظه در زبان جاوا در زمان اجرا [6] مقید می‌شود.

---

[1] binding
[2] binding time
[3] language design time
[4] language implementation time
[5] compile time
[6] run time or execution time

انقیاد

- یک انقیاد ایستا [1] 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 [2] می‌گوییم.

[1] static binding
[2] dynamic binding

انقیاد

- 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 [1] و ضمنی [2].
- 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.

[1] explicit declaration
[2] implicit declaration

انقیاد

- در برخی از زبان‌ها نام متغییر تعیین کننده نوع آن نیز هست. مثلا در زبان پرل متغیرهای عددی [1] با علامت $ و آرایه‌ها [2] 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.

[1] scalar
[2] arrays

انقیاد

- 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.

انقیاد

- 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، همهٔ متغیرها ارجاعی و از نوع شیء عمومی هستند و به کلاس‌های مختلف ارجاع می‌دهند.

انقیاد

- 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 هزینهٔ 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.

انقیاد

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

[1] memory allocation
[2] memory deallocation
[3] lifetime

انقیاد

- متغیرها را از لحاظ طول عمر به چهار دسته می‌توان تقسیم کرد : (۱) ایستا [1] ، (۲) پویا در پشته [2] ، (۳) پویا در هیپ به طور صریح [3] ، (۴) پویا در هیپ به طور ضمنی [4]

[1] static
[2] stack-dynamic
[3] explicit heap-dynamic
[4] implicit heap-dynamic

انقیاد

- یک متغیر ایستا [1] 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 [2] در حافظه تخصیص داده می‌شوند.
- یکی از مزیت‌های استفاده از متغیر ایستا سرعت دسترسی به آن است. در زبان سی++ برای متغیرهای ایستا از کلمهٔ کلیدی `static` استفاده می‌کنیم.
- وقتی متغیر ایستا در یک کلاس جاوا یا سی++ تعریف شود، آن متغیر متعلق به کلاس است نه اشیای آن کلاس.

[1] static variable
[2] data segment

انقیاد

- یک متغیر پویا در پشته [1] متغیری است که انقیاد حافظه آن در زمان اجرا در لحظه تعریف متغیر رخ می‌دهد.
- این متغیرها در فضای پشتهٔ [2] 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، همهٔ 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++ همهٔ متغیرها به طور پیش فرض متغیر پویا هستند که در پشته تعریف می‌شوند.

[1] stack-dynamic variable
[2] stack

انقیاد

- یک متغیر صریح پویا در هیپ [1] خانهٔ 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 [2] قرار می‌گیرند، که تنها توسط اشاره‌گر و مرجع قابل دسترسی هستند.
- فضای هیپ فضایی در حافظه است که ساختار آن کاملا 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.

[1] explicit heap-dynamic variable
[2] heap

انقیاد

- وقتی فضای هیپ تخصیص 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. همهٔ اشیا در جاوا متغیر پویا هستند که در هیپ تخصیص داده می‌شوند و توسط متغیر مرجع 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 مرجع‌هاست.

انقیاد

- یک متغیر ضمنی پویا در هیپ [1] 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 بالای 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.

[1] implicit heap-dynamic variable

حوزهٔ تعریف

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

---

[1] scope
[2] visible
[3] block
[4] local
[5] nonlocal

حوزهٔ تعریف

- حوزهٔ تعریف به دو دستهٔ ایستا و پویا تقسیم می‌شود.
- برای پیدا کردن مقدار متغیری که حوزهٔ تعریف آن ایستا است، کامپایلر ابتدا در بلوک فعلی به دنبال متغیر می‌گردد. اگر متغیر یافت نشد، بلوک‌های پدر را برای پیدا کردن تعریف متغیر و مقدار آن جستجو می کند.
- اگر حوزهٔ تعریف متغیری پویا باشد، مقدار متغیر بستگی به اجرا و پشته فراخوانی پیدا می‌کند. آخرین مقداری که در پشتهٔ فراخوانی به آن متغیر داده شده است، مقداری است که برای آن متغیر در نظر گرفته می‌شود.

حوزهٔ تعریف

- در مثال زیر اگر حوزهٔ تعریف x ایستا باشد، آنگاه خروجی برنامه ۱۰ است، اما اگر حوزهٔ 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 }
```

حوزهٔ تعریف

- حوزهٔ تعریف ایستا [1] اولین بار در زبان الگول معرفی شد و بیشتر زبان‌های دستوری [2] بعد از الگول این مفهوم را از الگول گرفته‌اند.
- دو دسته از زبان‌ها وجود دارند که در آنها حوزهٔ 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.

[1] static scope
[2] imperative languages

حوزهٔ تعریف

- در زبان‌هایی با حوزهٔ 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 [1] 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.

[1] parent block

حوزهٔ تعریف

- برنامه زیر را در زبان پایتون در نظر بگیرید :

```
1  def func() :
2      def sub1() :
3          x = 7
4          sub2()
5      def sub2() :
6          y = x
7          print("y = ",y)
8      x = 3
9      sub1()
10
11 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.

حوزهٔ تعریف

- 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.

حوزهٔ تعریف

- 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 همنام در حلقه تعریف شده است.

حوزهٔ تعریف

- 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 حوزهٔ تعریف ایستا در توابع تو در تو استفاده می‌شود ولی در بلوک‌های تو در تو حوزهٔ تعریف ایستا استفاده نمی‌شود.

حوزهٔ تعریف

- کد زیر در زبان پایتون را در نظر بگیرید :

```
for i in range (2) :
     x = 5
     print(x)
print(x)
```

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

حوزهٔ تعریف

- در زبان‌های تابعی حوزهٔ 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 :

```
let
    val   top = a + b
    val   bottom = c - d
in
    top / bottom
end;
```

حوزهٔ تعریف

- 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 :

```
1 console.log(x)
2 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.

حوزهٔ تعریف

- 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 [1] می‌نامیم که توسط همهٔ توابع قابل مشاهده‌اند.
- در زبان سی و سی++، می‌توان علاوه بر تعریف [2] متغیرها، آن‌ها را اعلام [3] 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 کلمهٔ کلیدی `extern` اعلام کرد.

[1] global variable
[2] define
[3] declare

حوزهٔ تعریف

- 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 حوزهٔ تعریف [1] (::) 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 می شود. برای اعلام متغیر به عنوان یک متغیر عمومی در یک تابع از کلمهٔ `global` استفاده می‌کنیم.

[1] scope operator

حوزهٔ تعریف

- برنامه زیر را در نظر بگیرید :

```
day = " Monday "
def  today() :
       print (" Today is ", day)
today()
```

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

حوزهٔ تعریف

- 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.

حوزهٔ تعریف

- 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()
```

حوزهٔ تعریف

- 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 خطرساز 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.

حوزهٔ تعریف

- در برخی از زبان‌ها مانند لیسپ، حوزهٔ 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 حوزهٔ تعریف ایستا انتظار داریم.

حوزهٔ تعریف

- حال برنامه زیر را در نظر بگیرید :

```
(defparameter    x    100)
(defun    fun1(x)    (print  x)    (fun2))
(defun    fun2()    (print  x))
```

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

حوزهٔ تعریف

- 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 حوزهٔ تعریف پویا در بسیاری از زبان‌های برنامه نویسی استفاده نمی‌شود.

نوع‌های داده‌ای

- در این فصل در مورد نوع‌های داده‌ای از جمله نوع‌های داده‌ای اصلی، آرایه‌ها، لیست‌ها و غیره صحبت خواهیم کرد.

- یک نوع داده‌ای [1] 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.

[1] data type

نوع‌های داده‌ای

- زبان پی‌ال [1] 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 [2] 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 [3] انجام دهد.
- سیستم نوع [4] در یک زبان برنامه نویسی تعریف می‌کند که نوع‌های مختلف را چگونه می‌توان مقدار دهی و به یکدیگر منتسب کرد.

[1] PL/I
[2] User-defined data types
[3] type checking
[4] 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 [1] و رکوردها [2] 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.

[1] array
[2] record

نوع داده‌ای اصلی

- نوع داده‌هایی که خود از نوع داده‌های ساده‌تر تشکیل نشده‌اند را نوع داده‌های اصلی [1] می‌نامیم.
- برخی از زبان‌های برنامه نویسی ابتدا تنها نوع داده‌ای عددی داشتند. مهم‌ترین نوع دادهٔ 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.

[1] primitive data types

نوع داده‌ای اصلی

- نوع داده‌ای ممیز شناور [1] 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 [2] است، یک بیت برای علامت، ۱۱ بیت برای توان و ۵۲ بیت برای قسمت اعشاری به کار می‌رود.

[1] floating-point
[2] double-precision floating-point

نوع داده‌ای اصلی

- برخی از زبان‌های برنامه نویسی مانند پایتون، نوع داده‌ای مختلط [1] 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 دهدهی 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 دهدهی دقتی را از دست بدهند. برای ذخیره سازی دقیق این نوع داده‌ای اعداد را مانند رشته ذخیره می‌کند.

[1] complex

نوع داده‌ای اصلی

- نوع داده‌ای بولی [1] برای ذخیرهٔ مقادیر درست [2] و نادرست [3] 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.

[1] boolean
[2] true
[3] false

نوع داده‌ای اصلی

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

[1] character
[2] ASCII (American Standard Code for Information Interchange)
[3] Unicode Consorsium

نوع داده‌ای رشته‌ای

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

---

[1] character string type
[2] concatenation
[3] assigniment
[4] substring
[5] 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 طولانی‌است. 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

- نوع داده‌ای شمارشی [1] 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 [2] را تعریف می‌کند.

[1] enumeration type
[2] 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 دادهٔ 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 داده آرایه

- نوع دادهٔ آرایه [1] 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 [2] یا اندیس [3] 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 [4] ندارند، اما جاوا بازهٔ دسترسی را بررسی می‌کند.

[1] array
[2] subscript operator
[3] index
[4] array range check

نوع داده آرایه

- چهار نوع مختلف از آرایه‌ها وجود دارند : (۱) آرایه‌های ایستا [1] ، (۲) آرایه‌های پویای ثابت روی پشته [2] ، (۳) آرایه‌های پویای ثابت بر روی هیپ [3] و (۴) آرایه‌های پویای بر روی هیپ [4].
- آرایه‌های ایستا قبل از اجرای برنامه اندازه معین دارند و قبل از شروع برنامه بر روی قسمت داده [5] 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.

[1] static arrays
[2] fixed stack-dynamic arrays
[3] fixed heap-dynamic arrays
[4] heap-dynamic arrays
[5] data segment

نوع داده آرایه

- آرایه‌های پویای ثابت بر روی پشته قبل از اجرای برنامه اندازه معین دارند و در هنگام تعریف بر روی پشتهٔ 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 اندازهٔ 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.

نوع داده آرایه

- 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.

نوع داده آرایه

- مثلا 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")
```

نوع داده آرایه

- در برخی از زبان‌ها مانند روبی، مفهومی به نام برش آرایه [1] 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]]
```

[1] slice of array

نوع داده آرایه

- به طور کل قوانین برش در پایتون به صورت زیر می‌باشند.

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

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

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

- به طور مثال :

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

نوع داده آرایه

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

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

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

```
vector [:: -1] # [16,14,12,10,8,6,4,2]
vector [1 :: -1] # [4,2]
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 می‌توانیم با استفاده از رابطه زیر به دست آوریم :

address (array[k]) = address (array[0]) + k * element-size

- در زبان‌هایی که محدوده دسترسی اندیس را بررسی می‌کنند، کامپایلر نیاز به نگهداری اطلاعات مربوط به آدرس آرایه و نوع آرایه و اندازهٔ آرایه دارد ولی در صورتی که کامپایلر محدوده دسترسی را بررسی نکند، نیازی به نگهداری اندازه آرایه نیست.

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

address (m[i][j]) = address (m[0][0]) + (i * n + j) * element-size

آرایه‌های انجمنی

- یک آرایه انجمنی [1] 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.

[1] associative array

آرایه‌های انجمنی

- در زبان پایتون به آرایه‌های انجمنی، دیکشنری [1] 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 [2] 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
```

[1] dictionary
[2] hash value

نوع داده‌ای رکورد

- یک رکورد [1] تعدادی متغیر که هر کدام می‌توانند از یک نوع متفاوت باشند را تجمیع 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 اندازهٔ عناصر قبلی آن قابل محاسبه است.
- در سی و سی++، برای تعریف یک رکورد از نوع داده‌ای ساختمان یا استراکت [2] 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.

[1] record
[2] struct

نوع داده‌ای چندتایی

- یک نوع داده‌ای چندتایی [1] 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 [2] هستند، ولی عناصر چندتایی غیر قابل تغییر [3] 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.

[1] tuple
[2] mutable
[3] 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 [1] نامیده می‌شود.

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

[1] list comprehension

نوع داده‌ای اجتماع

- نوع داده‌ای اجتماع [1] ، 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 بازهٔ 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.

[1] union

نوع داده‌ای اجتماع

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

```
union Number {
      int i ;
      float f ;
};
union Number n;
n.i = 2
n.f = 3.1 // n.i is erased
```

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

نوع‌های اشاره‌گر و مرجع

- یک اشاره‌گر [1] 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 حافظهٔ 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.

[1] pointer

نوع‌های اشاره‌گر و مرجع

- زبان‌هایی که نوع اشاره‌گر را پیاده سازی می‌کنند به یک عملگر یا تابع برای تخصیص حافظه پویا و انتساب آن به اشاره‌گر نیاز دارند که این عملگر در زبان سی++ توسط کلمهٔ new پیاده‌سازی شده است. همچنین با استفاده از عملگر delete می‌توان حافظهٔ تخصیص داده شده را آزاد کرد.
- عملگری که برای دریافت مقدار مکان حافظه‌ای که توسط اشاره‌گر قابل دسترسی است استفاده می‌شود عملگر رفع ارجاع [1] نام دارد. این عملگر در زبان سی++ توسط توسط ستاره (*) پیاده سازی 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 پیاده سازی کرد زبان پی‌ال۱ بود.

[1] dereference

نوع‌های اشاره‌گر و مرجع

- 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 [1] 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 [2] 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.

[1] dangling pointer
[2] memory leakage

نوع‌های اشاره‌گر و مرجع

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

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

[1] ampersand

نوع‌های اشاره‌گر و مرجع

- نوع داده‌ای مرجع 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 مرجع

- 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 مرجع در اینجا نسبت به اشاره‌گر این است که عملیات توسط آن ساده‌تر انجام می‌شود.

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

- در زبان جاوا همه اشیای کلاس‌ها متغیر مرجع هستند.

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

نوع‌های اشاره‌گر و مرجع

- یکی از روش‌های پیاده سازی اشاره‌گرها، روش قفل و کلید [1] 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 همهٔ 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.

[1] locks and keys approach

نوع‌های اشاره‌گر و مرجع

- در زبان‌هایی مانند جاوا که جمع‌آوری فضاهای آزاد شده یا بازیافت حافظه (زباله‌روبی) [1] 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 [2] و روش دوم علامت گذاری و جاروب [3] نامیده می‌شوند.
- در روش شمارندهٔ 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 مرجع 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.

[1] garbage collection
[2] reference counter
[3] mark and sweep

نوع‌های اشاره‌گر و مرجع

- در روش نشانه گذاری و جاروب، ابتدا سلول‌های حافظه تخصیص داده می‌شوند تا هنگامی که حافظه پر شود. در این لحظه بازیافت کنندهٔ حافظه همهٔ مرجع‌ها و اشاره‌گرها در برنامه را در حافظه دنبال می‌کند و همهٔ فضاهایی که توسط آن اشاره‌گرها استفاده می‌شوند را علامت گذاری می‌کند. سپس همهٔ سلول‌های حافظه که علامت گذاری نشده‌اند جاروب می‌شوند یا به عبارتی در مجموعهٔ سلول‌های قابل تخصیص قرار می‌گیرند.

بررسی نوع

- بررسی نوع [1] فعالیتی است که به موجب آن اطمینان حاصل می‌شود که همهٔ 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 [2] 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.

[1] type checking
[2] compatible

بررسی نوع

- خطای نوع [1] 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 [2] 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.
- یک زبان برنامه نویسی در دسته زبان‌های نوع دهی قوی [3] است اگر همهٔ خطاهای نوع قبل از اجرا تشخیص داده شوند.

[1] type error
[2] dynamic type checking
[3] strongly typed

راست

- راست [1] یک زبان برنامه نویسی است که پارادایم‌های برنامه نویسی رویه‌ای، تابعی و همروند را پشتیبانی می‌کند.
- بیشترین تمرکز زبان برنامه نویسی راست بر روی راندمان [2] ، قابلیت اطمینان [3] و همروندی [4] است.

[1] rust
[2] efficiency
[3] reliability
[4] concurrency

راست

- زبان راست یک زبان کامپایل شونده است و کامپایلر آن به نحوی طراحی شده است که برنامه‌های آن از راندمان بالایی برخوردارند. علاوه بر این کامپایلر اطمینان حاصل می‌کند که برنامهٔ نوشته شده در زمان اجرا دسترسی غیر مجاز به حافظه ندارد. در زبان راست، بازیافت کنندهٔ حافظه [1] وجود ندارد اما ساختارهایی وجود دارد که به کامپایلر کمک می‌کند بتواند در زمان کامپایل از نشتی‌های 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 توسعهٔ هستهٔ لینوکس مورد استفاده قرار گرفت.

[1] garbage collector

راست : نمادها و متغیرها

- نماد [1] نامی است که یک مقدار را نشان می‌دهد. نمادها در زبان راست غیر قابل تغییر [2] هستند.
- یک نماد با کلمهٔ `let` تعریف می‌شود. اگر مقدار یک نماد 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 }
```

[1] symbol
[2] immutable

راست : نمادها و متغیرها

- با استفاده از کلمه mut مخفف کلمه قابل تغییر [1] می‌توان یک نماد قابل تغییر تعریف کرد. نماد قابل تغییر در واقع یک متغیر است.

```
let mut x = 5 ;
x = 6
```

- با استفاده از کلمه const می‌توان یک ثابت تغییر کرد. مقدار یک ثابت قابل تغییر است و تفاوت آن با نماد این است که نمی‌توان آن را مجدداً تعریف کرد.

```
const PI = 3.141592 ;
PI = 3.1415 // error
const PI : f16 = 3.1415 // error
let pi = 3.141592
let pi = 3.1415 // OK
```

[1] mutable

راست : نمادها و متغیرها

- یک نماد را می‌توان در یک بلوک تعریف کرد و حوزهٔ تعریف آن نماد 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}")
۷ // x = 6
```

راست : نمادها و متغیرها

- وقتی یک نماد بازتعریف می‌شود، نوع آن می‌تواند متفاوت باشد.

```
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 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 نماد 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.

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

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

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

راست : نوع‌های داده‌ای

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

```
1 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.

```
let a = [1, 2, 3, 4, 5]
```

راست : نوع‌های داده‌ای

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

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

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

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

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

راست : توابع

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

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

- عبارتی که در یک بلوک بدون نقطه ویرگول یا سمی‌کالن [1] اعلام شده است، از بلوک بازگردانده می‌شود.
- برای مثال می‌توانیم بنویسیم :

```
let y = {
            let x = 3 ;
            x + 1
} ;
// y = 4
```

[1] 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 کلمهٔ 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
```

راست : ساختار کنترلی

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

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

راست : ساختار کنترلی

- همچنین با استفاده از دستور for می‌توان به صورت پیمایش در یک آرایه و یا پیمایش در 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 }
```

راست : مالکیت

- مالکیت [1] 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 [2] را تضمین کند، بدون اینکه نیازی به بازیافت کنندهٔ حافظه [3] داشته باشد.

[1] ownership
[2] memory safety guarantee
[3] garbage collector

راست : مالکیت

- مالکیت در زبان راست به مجموعهٔ قوانینی گفته می‌شود که تعیین می‌کنند مدیریت حافظه چگونه انجام می‌شود.
- برخی از زبان‌ها مانند جاوا یک مکانیزم بازیافت حافظه دارند که به طور منظم بررسی می‌کند به کدام قسمت‌های حافظه دیگری نیازی نیست و آن مکان‌های حافظه بی استفاده را برای استفاده مجدد آزادسازی و بازیافت می‌کند. در برخی از زبان‌های دیگر مانند سی++ مدیریت حافظه باید توسط برنامه نویس به طور صریح انجام شود. مشکل زبان‌های دسته اول سرعت اجرای پایین آنهاست و مشکل زبان‌های دسته دوم پایین بودن قابلیت اطمینان برنامه‌های آنهاست چرا که برنامه نویس ممکن است به درستی حافظه را تخصیص و آزادسازی نکند.
- زبان راست راه‌حل سومی را پیشنهاد می‌کند. حافظه توسط سیستمی به نام سیستم مالکیت مدیریت می‌شود. این سیستم مالکیت قوانینی را تعریف می‌کند که در زمان کامپایل قابل بررسی هستند، پس کامپایلر می‌تواند اطمینان حاصل کند که این قوانین به درستی اعمال شده‌اند و در زمان اجرا به حافظه دسترسی ایمن وجود دارد. اگر یکی از قوانین مالکیت نقض شده باشد، برنامه راست کامپایل نمی‌شود. این مکانیزم سرعت اجرای برنامه را کاهش نمی‌دهد.

راست : مالکیت

- 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.

راست : مالکیت

- حافظهٔ 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 دهندهٔ حافظه [1] 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.

[1] memory allocator

راست : مالکیت

- قوانین مالکیت در راست به صورت زیر هستند :
۱. هر مقداری در زبان راست یک مالک [1] دارد.
۲. هر مقداری در هر لحظه فقط یک مالک دارد.
۳. وقتی از حوزهٔ تعریف مالک خارج می‌شویم، مقدار آن از بین می‌رود.

[1] owner

راست : مالکیت

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

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

راست : مالکیت

- 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 کد زیر رشته در قسمت کد برنامه قرار می‌گیرد.

```
1 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، اندازهٔ آن به مقدار دلخواه می‌تواند افزایش پیدا کند.
- از آنجایی که حافظه بر روی هیپ قرار دارد بنابراین باید در هنگام نیاز تخصیص داده شود و در هنگام عدم نیاز حافظهٔ آن 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 کنندهٔ حافظه وجود ندارد، این کار باید توسط برنامه نویس انجام شود. آزادسازی حافظه به طور دستی معمولا 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 خط مواجه می‌شوند. اگر آزادسازی حافظه دوبار انجام شود، با خطای دسترسی مواجه می‌شویم.

راست : مالکیت

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

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

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

راست : مالکیت

- کد زیر را در نظر بگیرید :

```
let x = 5 ;
let y = x ;
```

- نمادها و 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.

راست : مالکیت

- حال کد زیر را در نظر بگیرید :

```
let s1 = String :: from ("hello");
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.

راست : مالکیت

- گفتیم وقتی از حوزهٔ تعریف یک متغیر خارج می‌شویم آزادسازی حافظه انجام می‌شود، اما باید توجه داشت که اشاره‌گر متعلق به کدام متغیر است و آزادسازی حافظه برای کدام متغیر انجام می‌شود.
- وقتی می‌نویسیم `let s2 = s1` در واقع `s1` بی اعتبار می‌شود. بنابراین در کد زیر با پیام خطای کامپایلر مواجه می‌شویم :

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

راست : مالکیت

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

[1] shallow copy
[2] deep copy
[3] move

راست : مالکیت

- اما اگر بخواهیم مقدار یک رشته را کپی عمیق کنیم بعنی 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.

```
fn main() {
    let s = String::from("hello");
    takes_ownership(s);  // s's value moves into
                         // the function. and so is
                        // no longer valid.
}
fn takes_ownership(st: String) {
     println!("{}", st);
} // 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.

راست : مالکیت

- 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 } // frop 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 را 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 :

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

- اما اگر بخواهیم همیشه این کار را انجام دهیم برنامه نویسی بسیار سخت می‌شود.

- خوشبختانه در راست مفهوم دیگری به نام مرجع وجود دارد که توسط آن می‌توان از انتقال مالکیت جلوگیری کرد.

راست : مرجع‌ها

- یک متغیر مرجع در زبان راست، همانند متغیر مرجع در سی++ یک نام مستعار برای یک مکان حافظه است.
- وقتی یک متغیر با استفاده از یک متغیر مرجع 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 : مرجع‌ها

- یک متغیر مرجع 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 مرجع است. از آنجایی که یک متغیر مرجع 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 [1] گفته می‌شود، چرا که یک متغیر مرجع مالکیت حافظه را به دست نمی‌آورد، اما آن را برای استفاده برای مدت زمانی قرض می‌گیرد. در واقع در عملیات قرض گرفتن متغیر مرجع 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 مرجع نیز که به یک نماد غیر قابل تغییر اشاره می‌کند، قابل تغییر نیست.

[1] borrowing

راست : مرجع‌ها

- یک متغیر مرجع را می‌توان قابل تغییر [1] تعریف کرد. یک مرجع 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 }
```

[1] mutable reference

راست : مرجع‌ها

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

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

راست : مرجع‌ها

- در واقع در این کد خواسته‌ایم یک متغیر قابل تغییر را در یک زمان به دو مرجع قرض دهیم، که این عملیات در زبان راست ممنوع است.

- این محدودیت به کامپایلر کمک می‌کند که از وضعیت رقابت داده [1] در زمان کامپایل جلوگیری کند. اگر دو یا چند مرجع 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 مرجع قابل تغییر را در یک بلوک جداگانه تعریف کنیم

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

[1] data race

راست : مرجع‌ها

- چند مرجع غیر قابل تغییر به طور همزمان می‌توانند برای اشاره به یک داده تعریف شوند، ولی مرجع قابل تغییر حتی با مرجع 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!("{}, {}, and {}", r1, r2, r3);
```

راست : مرجع‌ها

- دلیل این امر این است که مرجع‌های غیر قابل تغییر انتظار ندارند مقداری که به آن اشاره می‌کنند ناگهان تغییر کندو اما چند مرجع غیر قابل تغییر به طور همزمان می‌توانند تعریف شوند چون هیچ‌کدام مقدار داده را تغییر نمی‌دهند.

راست : مرجع‌ها

- دقت کنید که حوزه تعریف یک متغیر مرجع از زمانی است که تعریف می‌شود تا زمانی که استفاده می‌شود.
  بنابراین اگر یک متغیر مرجع تعریف و سپس استفاده شود، یک متغیر مرجع 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!("{} and {}", 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 : مرجع‌ها

- 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 : مرجع‌ها

- در زبان‌هایی که اشاره‌گر در آنها وجود دارد، ممکن است به راحتی خطای اشاره‌گر معلق [1] 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 مرجع معلق به وجود نمی‌آید. اگر مرجعی به یک متغیر وجود داشته باشد، کامپایلر اطمینان حاصل می‌کند که متغیر مربوطه قبل از مرجع از حوزه تعریف خارج نمی‌شود و فضای حافظه آن آزاد نمی‌شود.

[1] dangling pointer

راست : مرجع‌ها

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

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

راست : مرجع‌ها

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

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

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

راست : نوع برش

- برش [1] کمک می‌کند مرجعی به دنباله‌ای از عناصر در یک مجموعه بسازیم. از آنجایی که برش یک اشاره‌گر است، مالکیت داده ندارد.
- فرض کنید می‌خواهیم یک برش یا یک قسمت از یک رشته را توسط یک تابع بازگردانیم.
- برای مثال اگر بخواهیم اولین کلمه از یک رشته را که با خط فاصله از کلمهٔ دوم جدا شده است به دست آوریم، می‌توانیم تابع زیر را بنویسیم :

```
۱ fn first_word(s: &String) -> usize {
۲     let bytes = s.as_bytes();
۳     for (i, &item) in bytes.iter().enumerate() {
۴         if item == b' ' {
۵             return i;
۶         }
۷     }
۸     s.len()
۹ }
```

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 رشتهٔ تهی تبدیل کنیم. متغیری که به اندیس اولین خط فاصله اشاره می‌کند، اکنون غیر معتبر است.

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

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[..];
```

راست : نوع برش

- حال تابع `first_world` را با استفاده از برش به صورت زیر تعریف می‌کنیم.

```
۱ fn first_word(s: &String) -> &str {
۲     let bytes = s.as_bytes();
۳     for (i, &item) in bytes.iter().enumerate() {
۴         if item == b' ' {
۵             return &s[0..i];
۶         }
۷     }
۸     &s[..]
۹ }
```

راست : نوع برش

- کامپایلر اطمینان حاصل می‌کند که همه مرجع‌ها به یک متغیر معتبر می‌مانند. پس اگر سعی کنیم این بار رشته را به یک رشتهٔ تهی تبدیل کنیم، از آنجایی که متغیر مرجعی داریم که به یک برش از رشته اشاره می‌کند، کامپایل 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 می‌آید. به یاد داریم وقتی که یک مرجع به یک متغیر داریم نمی‌توانیم یک مرجع قابل تغییر تعریف کنیم. از آنجایی که تابع `clear()` می‌خواهد رشته را تهی کند باید یک مرجع قابل تغییر از آن بگیرد و این کار امکان پذیر نیست چرا که متغیر `word` به یک مرجع به متغیر `s` است.
- بنابراین کامپایل 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 مرجع 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.

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

راست : نوع برش

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

```
let a = [1,2,3,4,5];
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 مواجه نمی‌شود.

راست : ساختمان

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

```
struct User {
    active: bool,
    username: String,
    email: String,
    sign_in_count: u64,
}
```

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

راست : ساختمان

- 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.

راست : ساختمان

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

```
fn build_user(email: String, username: String) -> User {
    User {
        active: true,
        username: username,
        email: email,
        sign_in_count: 1,
    }
}
```

راست : ساختمان

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

```
fn build_user(email: String, username: String) -> User {
    User {
        active: true,
        username,
        email,
        sign_in_count: 1,
    }
}
```

راست : ساختمان

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

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

راست : ساختمان

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

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

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 }
```

راست : ساختمان

- 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 [1] 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 }
```

[1] method

راست : ساختمان

```
fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };
    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
}
```

راست : ساختمان

- یک متود می‌تواند با یک فیلد ساختمان همنام باشد.

```
۱  impl Rectangle {
۲      fn width(&self) -> bool {
۳          self.width > 0
۴      }
۵  }
۶  fn main() {
۷      let rect1 = Rectangle {
۸          width: 30,
۹          height: 50,
۱۰     };
۱۱     if rect1.width() {
۱۲         println!("The rectangle has a nonzero width; it is {}", rect1.w
۱۳     }
۱۴ }
```

راست : ساختمان

- یک متود می‌تواند پارامتر نیز داشته باشد.

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };
```

راست : ساختمان

```
    let rect2 = Rectangle {
        width: 10,
        height: 40,
    };
    let rect3 = Rectangle {
        width: 60,
        height: 45,
    };
    println!("Can rect1 hold rect2? {}", rect1.can_hold(&rect2));
    println!("Can rect1 hold rect3? {}", rect1.can_hold(&rect3));
}
```

راست : ساختمان

- توابع یک ساختمان می‌توانند به عنوان ورودی پارامتر `self` نداشته باشند. این توابع را توابع مرتبط [1] 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 کلمهٔ `Self` استفاده می‌کنیم.

```
impl Rectangle {
    fn square(size: u32) -> Self {
        Self {
            width: size,
            height: size,
        }
    }
}
```

[1] associated function

راست : ساختمان

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

راست : ساختمان

- 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 دادهٔ شمارشی برای نام‌گذاری مجموعه‌ای از مقادیر استفاده می‌شود.
- برای مثال :

```
enum IpAddrKind {
    V4,
    V6,
}
struct IpAddr {
    kind: IpAddrKind,
    address: String,
}
let home = IpAddr {
    kind: IpAddrKind::V4,
    address: String::from("127.0.0.1"),
};
```

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 دادهٔ شمارشی در یک ساختمان نیز تعریف کنیم و مقادیر مورد نیاز را در ساختمان قرار دهیم.
- در راست نوع دادهٔ 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 دادهٔ شمارشی زیر معادل تعریف چهار ساختمان متفاوت است، با این تفاوت که نوع دادهٔ شمارشی همهٔ ساختمان‌های همانند را در یک گروه قرار می‌دهد.

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
// it is equivalent to :
struct QuitMessage; // unit struct
struct MoveMessage {
    x: i32,
    y: i32,
}
struct WriteMessage(String); // tuple struct
struct ChangeColorMessage(i32, i32, i32); // tuple struct
```

راست : نوع شمارشی

- مشکل تعریف چند ساختمان در مثال قبل این است که اگر بخواهیم تابعی تعریف کنیم که با همهٔ 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 دادهٔ شمارشی که توسط کتابخانهٔ 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 همهٔ این حالات باید توسط برنامه نویس بررسی شوند ولی در زبان راست کامپایلر اطمینان حاصل می‌کند که همهٔ 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 طراحی‌های قبلی وظیفه برنامه نویس بود که بررسی کند آیا مقداری `null` است یا خیر. در طراحی زبان راست از نوع داده‌ای `option` استفاده می‌شود و بدین صورت در زمان کامپایل اطمینان حاصل می‌شود که همهٔ 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.

راست : نوع شمارشی

- نوع داده‌ای option به صورت زیر تعریف و به‌کار برده می‌شود.

```
enum Option<T> {
    None,
    Some(T),
}
let some_number = Some(5);
let some_char = Some('e');
let absent_number: Option<i32> = None;
```

راست : نوع شمارشی

- حال اگر سعی کنیم یک عدد صحیح را با عدد صحیح دیگری که می‌تواند تهی نیز باشد جمع کنیم با خطای کامپایل مواجه می‌شویم. در زبان‌های دیگر برنامه کامپایل می‌شود و با خطای زمان اجرا مواجه می‌شویم.
بنابراین در زبان راست برنامه نویس با مواجه شدن با خطای کامپایل مجبور می‌شود حالت‌های مختلف را در نظر بگیرید.

```
let x: i8 = 5;
let y: Option<i8> = Some(5);
let sum = x + y; //error
```

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 اسا الگوی تطبیق داده شده، دستورات مناسب را اجرا کند.
- کامپایلر اطمینان حاصل می‌کند که همهٔ 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 دادهٔ شمارشی داریم که همهٔ سکه‌های پولی موجود را شمارش می‌کند. حال می‌خواهیم تابعی بنویسیم که ارزش یک سکه را برگرداند. نیاز داریم که این تابع همهٔ حالات را بررسی کند، پس می‌توانیم به صورت از `match` استفاده کنیم.

راست : ساختار کنترلی تطابق

```rust
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

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 [1] 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.

[1] 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 می‌توانیم 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 }
```

راست : ساختار کنترلی تطابق

```rust
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => {
            println!("State quarter from {:?}!", state);
            25
        }
    }
}
```

راست : ساختار کنترلی تطابق

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

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        None => None,
        Some(i) => Some(i + 1),
    }
}
let five = Some(5);
let six = plus_one(five);
let none = plus_one(None);
```

راست : ساختار کنترلی تطابق

- همانطور که اشاره شد همهٔ شاخه‌های ممکن در یک الگو باید بررسی شوند، در غیر اینصورت 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 کلمهٔ `other` استفاده می‌کنیم.

```
let dice_roll = 9;
match dice_roll {
    3 => add_fancy_hat(),
    7 => remove_fancy_hat(),
    other => move_player(other),
}
fn add_fancy_hat() {}
fn remove_fancy_hat() {}
fn move_player(num_spaces: u8) {}
```

راست : ساختار کنترلی تطابق

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

```
let dice_roll = 9;
match dice_roll {
    3 => add_fancy_hat(),
    7 => remove_fancy_hat(),
    _ => reroll(),
}
fn add_fancy_hat() {}
fn remove_fancy_hat() {}
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(3u8);
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(3u8);
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 می‌توان استفاده کرد.

```
let mut count = 0;
match coin {
    Coin::Quarter(state) => println!("State quarter from {:?}!", state),
    _ => count += 1,
}
// it is equivalent to :
let mut count = 0;
if let Coin::Quarter(state) = coin {
    println!("State quarter from {:?}!", state);
} else {
    count += 1;
}
```