به نام خدا

مبانی برنامه‌نویسی

آرش شفیعی

فهرست مطالب

کتاب‌های مرجع

- زبان برنامه‌نویسی سی، از کرنیگان و ریچی [1]
- چگونه توسط سی برنامه بنویسیم، از دایتل و دایتل [2]
- هنر برنامه نویسی، از دونالد کنوث [3]

[1] The C Programming Language، by Brian Kernighan and Dennis Ritchie
[2] C How To Program، by Paul Deitel and Harvey Deitel
[3] The Art of Computer Programming، by Donald Knuth

معرفی سیستم‌های کامپیوتری

معرفی سیستم‌های کامپیوتری

- کامپیوتر وسیله‌ای است که با دریافت تعدادی ورودی و دریافت دنباله‌ای از عملیات منطقی و حسابی می‌تواند برروی ورودی‌ها محاسباتی انجام داده و نتیجه عملیات را به عنوان خروجی بازگرداند.
- سریع‌ترین کامپیوترها امروزه می‌توانند صدها کوادریلیون [1] یعنی در حدود $۱۰^{۱۷}$ عملیات محاسباتی در ثانیه انجام دهند. به عبارت دیگر اگر جمعیت کرهٔ زمین را ۸ میلیاد نفر در نظر بگیریم، سریع‌ترین کامپیوترهای امروزی قادرند در یک ثانیه ده‌ها میلیون عملیات محاسباتی به ازای هر یک از افراد کرهٔ زمین انجام دهند.

[1] quadrillion

معرفی سیستم‌های کامپیوتری

- دنبالهٔ عملیات منطقی و حسابی که توسط یک کامپیوتر دریافت می‌شود را یک برنامهٔ کامپیوتری [1] یا یک برنامه می‌نامیم. یک برنامه که توسط یک برنامه‌نویس [2] به کامپیوتر داده می‌شود، درواقع تعیین می‌کند چگونه ورودی‌ها پردازش شوند.

[1] computer program
[2] programmer

معرفی سیستم‌های کامپیوتری

- یک کامپیوتر در واقع یک ماشین انتزاعی است. این ماشین انتزاعی را می‌توان توسط قطعات فیزیکی–الکترونیکی پیاده‌سازی کرد.
- موتور تحلیلی چالرز بابیج [1] و ماشین تورینگ جهانی [2] دو نمونهٔ ابتدایی یک کامپیوتر انتزاعی هستند. ماشین تورینگ از یک واحد کنترل کننده و یک نوار ورودی تشکیل شده است.
- برای پیاده‌سازی‌های کامپیوترهای فیزیکی که آن‌ها را سیستم‌های کامپیوتری می‌نامیم از یک واحد محاسبات مرکزی [2] و یک واحد حافظه [3] استفاده می‌شود و برای ارسال ورودی‌ها به کامپیوتر از واحد ورودی [4] شامل موس و کیبورد و غیره و برای دریافت خروجی‌ها از یک واحد خروجی [5] شامل مانیتور استفاده می‌شود.

---

[1] Charles Babbage Analytical Engine
[2] Universal Turing Machine
[2] Central Processing Unit (CPU)
[3] Memory unit
[4] Input unit
[5] output unit

معرفی سیستم‌های کامپیوتری

- سیستم‌های کامپیوتر در ده‌ها سال گذشته به اندازهٔ یک اتاق بودند و میلیون‌ها دلار برای تولید آن‌ها هزینه می‌شد ولی سیستم‌های کامپیوتری امروزی با استفاده از قطعات سیلیکونی تولید می‌شوند و علاوه بر حجم بسیار کمی که دارند، هزینهٔ بسیار پایینی نیز برای تولید آنها صرف می‌شود.
- در مورد فراوانی مادهٔ سیلیکون باید گفت که کرهٔ زمین $۵/۹۷ \times ۱۰^{۲۴}$ کیلوگرم وزن دارد که ۳۲ درصد آن را آهن، ۳۰ درصد را اکسیژن، و ۱۵ درصد آن را سیلیکون تشکیل داده است. به عبارت دیگر، سیلیکون سومین عنصری است که در زمین به وفور یافت می‌شود و بنابراین هزینهٔ بسیار پایینی دارد.

معرفی سیستم‌های کامپیوتری

- واحدهای اصلی سازندهٔ یک سیستم کامپیوتری واحد پردازندهٔ مرکزی و واحد حافظه هستند.

معرفی سیستم‌های کامپیوتری

- واحد حافظه داده‌های ورودی را پس از دریافت از واحد ورودی و داده‌های خروجی را قبل از ارسال به واحد خروجی در خود نگهداری می‌کند. همچنین در هنگام پردازش داده‌ها، ممکن است نیاز به تولید داده‌هایی باشد که این داده‌ها نیز بر روی حافظه نگهداری می‌شوند. واحد حافظه معمولاً حافظه دسترسی تصادفی [1] (RAM) نامیده می‌شود زیرا به هر قسمت از داده‌ها می‌توان به طور تصادفی دسترسی پیدا کرد.
- در سیستم‌های کامپیوتری امروزی مقدار حافظه به ده‌ها گیگابایت [2] می‌رسد. یک گیگابایت در حدود یک میلیارد بایت است و یک بایت از هشت بیت تشکیل شده است. یک بیت می‌تواند مقدار صفر یا یک را در خود نگهدار کند.

[1] Random Access Memory
[2] gigabyte

معرفی سیستم‌های کامپیوتری

- واحد پردازندهٔ مرکزی قسمت کنترل کنندهٔ یک سیستم کامپیوتری است که به وسیلهٔ آن اطلاعات از ورودی دریافت می‌شوند، توسط عملیات حسابی (مانند جمع و تفریق و ضرب و تقسیم) و عملیات رابطه‌ای (مانند مقایسه) و عملیات منطقی (مانند عطف و فصل و نقیض) محاسبه می‌شوند و اطلاعات پردازش شده به واحد خروجی ارسال می‌شوند.
- امروزه بسیاری از پردازنده‌ها از چند هسته [1] تشکیل شده‌اند و هسته‌ها می‌توانند به طور مجزا محاسبات را انجام دهند و بدین ترتیب با استفاده از پردازش موازی (پردازش توسط چند هسته) محاسبات می‌توانند سریع‌تر انجام شوند.

[1] multicore

معرفی سیستم‌های کامپیوتری

- کوچکترین واحد اطلاعات در سیستم‌های کامپیوتری یک بیت است. یک بیت می‌تواند مقدار صفر یا یک را نگهداری کند. توسط بیت‌ها می‌توان اعداد را در مبنای دو نمایش داد. همهٔ اطلاعات در سیستم‌های کامپیوتری به صورت دودویی نگهداری می‌شوند.
- یک بایت (B) برابر با $۲^۳$ یا ۸ بیت است.
- یک کیلوبایت (KB) برابر با $۲^{۱۰}$ یا ۱۰۲۴ بایت است.
- به همین ترتیب یک مگابایت (MB) برابر با $۲^{۲۰}$ بایت یا ۱۰۲۴ کیلوبایت، یک گیگابایت (GB) برابر با $۲^{۳۰}$ بایت یا ۱۰۲۴ مگابایت، یک ترابایت (TB) برابر با $۲^{۴۰}$ بایت یا ۱۰۲۴ گیگابایت، یک پتابایت (PB) برابر با $۲^{۵۰}$ بایت یا ۱۰۲۴ ترابایت، می‌باشد.

معرفی سیستم‌های کامپیوتری

- برای نمایش حروف از یک سیستم کدگذاری استفاده می‌شود. در استاندارد کدگذاری آمریکایی برای تبادل اطلاعات [1] (ASCII) هر کاراکتر با یک عدد یک بایتی نمایش داده می‌شود، بنابراین در این سیستم کدگذاری می‌توان تنها ۱۲۸ حرف یا کاراکتر را نمایش داد. در کامپیوترهای ابتدایی یک بیت برای بررسی خطاها در نظر گرفته شده بود، بنابراین از ۷ بیت برای کدگذاری استفاده می‌شد.
- سیستم کدگذاری یونیکد [2] برای نمایش حروف زبان‌های غیر انگلیسی استفاده می‌شود. در این سیستم کدگذاری یک حرف می‌تواند با یک عدد ۲، ۳ یا ۴ نمایش داده شود و بنابراین با استفاده از یک نمایش ۴ بایتی می‌توان تا حدود ۴ میلیارد حرف را نمایش داد.

[1] American Standard Code for Information Interchange
[2] unicode

مبنای اعداد

- تبدیل اعداد دهدهی [1] به دودویی [2]: $(x)_{10} = (a_n a_{n-1} \cdots a_1 a_0)_2$ به طوری که $x = \Sigma_{i=0}^{n} a_i \times 2^i$ و $a_i \in \{0, 1\}$
- مثال: $(42)_{10} = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$
  $(42)_{10} = 32 + 8 + 2 = (101010)_2$
- اعداد دهدهی می‌توانند علاوه بر قسمت صحیح [3] قسمت اعشاری [4] نیز داشته باشند.
- تبدیل اعداد دهدهی اعشاری به دودویی: $(0.x)_{10} = (0.a_1 a_2 \cdots a_{n-1} a_n)_2$ به طوری که $0.x = \Sigma_{i=1}^{n} a_i \times 2^{-i}$ و $a_i \in \{0, 1\}$
- مثال: $(0.75)_{10} = 1 \times 2^{-1} + 1 \times 2^{-2}$
  $(0.75)_{10} = 0.5 + 0.25 = (0.11)_2$

[1] decimal
[2] binary
[3] integer part
[4] fractional part

مبنای اعداد

- روش تبدیل اعداد دهدهی اعشاری به دودویی: عدد دهدهی را n بار در ۲ ضرب می‌کنیم تا یا عدد به دست آمده قسمت اعشاری نداشته باشد و یا n از تعداد ارقام اعشاری مورد نیاز در عدد دودویی بیشتر شود. سپس عدد دهدهی بدون قسمت اعشاری را به دودویی تبدیل می‌کنیم و در عدد دودویی به دست آمده n رقم از سمت راست جدا می‌کنیم و ممیز اعشار را بعد از n رقم قرار می‌دهیم (در واقع عدد دودویی به دست آمده را n بار بر ۲ تقسیم می‌کنیم).
- مثال: معادل عدد دهدهی $(۴.۷۵)_{۱۰}$ را در مبنای دو را محاسبه کنید.
- $۴.۷۵ \times ۲ \times ۲ = ۱۹$
  $(۱۹)_{۱۰} = (۱۰۰۱۱)_{۲}$
  $(۱۰۰۱۱)_{۲} \div ۲ \div ۲ = (۱۰۰.۱۱)_{۲}$
  $(۴.۷۵)_{۱۰} = (۱۰۰.۱۱)_{۲}$

مبنای اعداد

- مثال: معادل عدد دهدهی $(۰.۳)_{۱۰}$ را در مبنای دو تا ۱۴ رقم اعشار محاسبه کنید.

- $۰.۳ \times ۲^{۱۴} = ۴۹۱۵.۲$
$(۴۹۱۵)_{۱۰} = (۱۰۰۱۱۰۰۱۱۰۰۱۱)_{۲}$
$(۱۰۰۱۱۰۰۱۱۰۰۱۱)_{۲} \div ۲^{۱۴} = (۰.۰۱۰۰۱۱۰۰۱۱۰۰۱۱)_{۲}$
$(۰.۰۱۰۰۱۱۰۰۱۱۰۰۱۱)_{۲} = (۰.۳)_{۱۰}$

مبنای اعداد

- روش تبدیل اعداد دودویی اعشاری به دهدهی: عدد دودویی را n بار در ۲ ضرب می‌کنیم تا عدد به دست آمده قسمت اعشاری نداشته باشد. سپس عدد دودویی بدون قسمت اعشاری را به دهدهی تبدیل می‌کنیم و عدد دهدهی به دست آمده را n بار بر ۲ تقسیم می‌کنیم.
- مثال: عدد دودویی $(۱۰۰.۱۱)_۲$ را به دهدهی تبدیل کنید.
- $(۱۰۰.۱۱)_۲ \times ۲ \times ۲ = (۱۰۰۱۱)_۲$
$(۱۰۰۱۱)_۲ = (۱۹)_{۱۰}$
$۱۹ \div ۲ \div ۲ = ۴.۷۵$
$(۱۰۰.۱۱)_۲ = (۴.۷۵)_{۱۰}$

مبنای اعداد

- یک عدد دودویی را می‌توانیم به صورت یک عدد علامت‌دار [1] یا یک عدد بدون علامت [2] تعبیر کنیم.
- اولین بیت (رقم) از سمت چپ یک عدد را برای نشان دادن علامت آن عدد استفاده می‌کنیم و آن را بیت علامت [3] می‌گوییم.
- اگر بیت علامت برابر با ۱ باشد عدد منفی است و اگر بیت علامت برابر با صفر باشد عدد مثبت است.

[1] signed
[2] unsigned
[3] sign bit

مبنای اعداد

- برای تبدیل یک عدد دودویی علامتدار $(b)_۲$ با بیت علامت ۱ به مبنای دهدهی ابتدا مکمل دو b را محاسبه می‌کنیم. فرض کنیم عدد به دست آمده عدد c است. حال عدد c را به مبنای ده تبدیل می‌کنیم و فرض می‌کنیم عدد به دست آمده برابر با d است: $(c)_۲ = (d)_{۱۰}$
- عدد دودویی b یک عدد منفی است که در مبنای ده برابر است با $-d$ بنابراین: $(b)_۲ = (-d)_{۱۰}$
- برای محاسبهٔ مکمل‌دو [1] یک عدد صفرها را به یک و یک‌ها را به صفر تبدیل کرده، سپس یک واحد به آن عدد می‌افزاییم.
- مثال: عدد بدون علامت $(۱۰۰۱)_۲$ در مبنای دو برابر است با ۹.
- اما عدد علامت‌دار $(۱۰۰۱)_۲$ در مبنای دو برابر است با -۷.

[1] two's complement

مبنای اعداد

- برای تبدیل یک عدد منفی دهدهی به یک عدد منفی دودویی، ابتدا آن عدد را به صورت مثبت در نظر گرفته، آن را به مبنای دو تبدیل کرده، سپس مکمل‌دو آن را محاسبه می‌کنیم.
- مثال: معادل عدد ۴۲− را در مبنای دو محاسبه کنید.
- $(۴۲)_{۱۰} = (۰۱۰۱۰۱۰)_{۲}$
  $(-۴۲)_{۱۰} = (۱۰۱۰۱۱۰)_{۲}$

مبنای اعداد

- اعداد دودویی را می‌توانیم با استفاده از روش زیر به اعداد پایهٔ شانزده (شانزده شانزدهی یا هگزادسیمال [1]) تبدیل کنیم.
- عدد دودویی را از سمت راست چهار بیت چهار بیت جدا می‌کنیم و معادل هگزادسیمال هر چهاربیت را از سمت راست می‌نویسیم. اعداد چهاربیتی می‌توانند بیت ۰ تا ۱۵ باشند. در مبنای شانزده، عدد ۱۰ را با A ، ۱۱ را با B ، ۱۲ را با C ، ۱۳ را با D ، ۱۴ را با E ، و ۱۵ را با F نشان می‌دهیم.
- مثال: معادل عدد ۴۲ را در مبنای شانزده محاسبه کنید.
- $(۴۲)_{۱۰} = (۱۰۱۰۱۰)_{۲} = (۲A)_{۱۶}$

[1] hexadecimal

ماشین و زبان اسمبلی

- هر پردازنده معمولاً یک زبان مختص به خود دارد که به آن زبان ماشین [1] گفته می‌شود. این زبان توسط طراحان پردازنده طراحی می‌شود.
- برای مثال برای جمع دو عدد، یک پردازنده به طور قراردادی یک عدد دودویی را به عنوان عملگر جمع در نظر می‌گیرد و دو عملوند ورودی را به صورت دو عدد دودویی برای عملگر جمع دریافت می‌کند.
- زبان اسمبلی [2] زبانی است بسیار شبیه به زبان ماشین که در آن دستوراتی که توسط پردازنده انجام می‌شوند به نحوی نامگذاری شده‌اند که توسط انسان قابل خواندن و فهمیدن هستند. پردازنده‌ها معمولاً زبان اسمبلی مخصوص خود را دارند. در زبان اسمبلی از کلمات زبان انگلیسی به طور قراردادی برای انجام عملیات توسط پردازنده‌ها استفاده شده است. برای مثال در یک زبان اسمبلی ممکن است از دستور `add` برای جمع دو عدد استفاده شود.
- برنامه مترجمی که زبان اسمبلی را به زبان ماشین تبدیل می‌کند اسمبلر [3] نامیده می‌شود.

[1] machine langague
[2] assembly language
[3] assembler

زبان سطح بالا

- اگر کامپیوتر و زبان مختص دستورات کامپیوتری را در پایین‌ترین سطح در نظر بگیریم و انسان و زبان‌های طبیعی را در بالاترین سطح، آنگاه زبان ماشین یک زبان سطح پایین محسوب می‌شود که در پایین‌ترین سطح این تقسیم‌بندی قرار دارد. زبان اسمبلی در سطحی بالاتر از زبان ماشین قرار می‌گیرد ولی همچنان به آن یک زبان سطح پایین گفته می‌شود چون به زبان کامپیوتر نزدیک‌تر است.
- با افزایش پردازنده‌ها و تنوع زبان‌های اسمبلی نیاز به زبان یا زبان‌هایی بود که به صورت یکپارچه برنامه‌نویسان بتوانند با استفاده از این زبان‌ها برای همهٔ پردازنده‌ها و معماری‌های متفاوت برنامه بنویسند. همچنین زبان اسمبلی به زبان ماشین بسیار نزدیک بود و بهتر بود زبان یا زبان‌هایی به وجود می‌آمدند که به زبان برنامه‌نویسان نزدیک‌تر باشند.
- چنین زبان‌هایی، که در دهه ۱۹۵۰ به وجود آمدند، زبان‌های سطح بالا [1] نامیده می‌شوند.

[1] high-level languages

زبان سطح بالا

- برای تبدیل یک برنامه در زبان سطح بالا به یک برنامه به زبان اسمبلی از برنامه‌ای به نام مترجم یا کامپایلر [1] استفاده می‌شود. با استفاده از یک زبان سطح بالا می‌توان دستوراتی نوشت که به زبان انگلیسی و زبان ریاضی شباهت بیشتری دارند تا زبان ماشین.
- اولین زبان برنامه‌نویسی، در یک رسالهٔ دکتری در سال ۱۹۵۱ توسط کورادو بوهم [2] در دانشگاه ای‌تی‌اچ زوریخ توصیف شد و به همراه یک کامپایلر عرضه شد.
- دو زبان مهم تجاری که در این دهه به وجود آمدند، عبارتند از فورترن [3] و کوبول [4] .
- نوآوری جدید فورترن این بود که به برنامه‌نویس کمک می‌کرد تا بتواند فرمول‌های ریاضی را به همان صورتی که برروی کاغذ نوشته می‌شوند بنویسید. در واقع کلمهٔ فورترن مخفف کلمهٔ ترجمهٔ فرمول [5] بود.

[1] compiler
[2] Corrado Bohm
[3] Fortran
[4] Cobol
[5] Formula Translation

زبان سطح بالا

- قبل از به وجود آمدن زبان فورترن برنامه‌نویسان برای نوشتن فرمول $i + ۲ * j$ نیاز بود مقدار i و j را در قسمتی از حافظه ذخیره کنند . سپس مقدار j را دو برابر کنند و سپس مقدار i را با دو برابر j جمع کنند، اما با استفاده از فورترن برنامه‌نویسان می‌توانستند فرمول را به شکلی که روی کاغذ می‌نوشتند در برنامه خود وارد کنند.

سیستم عامل

- سیستم عامل [1] برنامه‌ای است که برای مدیریت قطعات جانبی یک سیستم کامپیوتری مانند ورودی، خروجی‌ها و همچنین مدیریت منابع مانند حافظه و پردازنده استفاده می‌شود. با استفاده از یک سیستم عامل می‌توان برنامه‌ها را به طور همزمان اجرا کرد و برای اجرای همزمان برنامه‌ها نیاز به تخصیص حافظه و زمان پردازنده به برنامه‌ها به طور بهینه است.

[1] operating system

سیستم عامل

- سیستم عامل یونیکس [1] که یکی از مهم‌ترین سیستم عامل‌های ابتدایی است در سال ۱۹۶۹ بر روی یک کامپیوتر PDP۷ با استفاده از زبان اسمبلی توسط دنیس ریچی [2] و کن تامسون [3] در آزمایشگاه‌های بل طراحی و پیاده‌سازی شد.

[1] Unix
[2] Dennis Richie
[3] Ken Thompson

زبان سی

- یونیکس در نسخهٔ بعدی برای یک کامپیوتر PDP۱۱ پیاده‌سازی شد و از آنجایی که برای کامپیوتر جدید به تعدادی ابزار نیاز بود، طراحان تصمیم گرفتند کامپایلری برای یک زبان سطح بالا طراحی کنند تا ابزارها را بتوان با استفاده از آن زبان سطح بالا راحت‌تر پیاده‌سازی کرد. در آن زمان زبان BCPL طراحی شده بود. طراحان یونیکس با استفاده از ایده‌های این زبان، و همچنین زبان الگول [4] کامپایلری برای یک زبان جدید طراحی و پیاده‌سازی کردند و زبان جدید را B نامیدند.
- بین سال‌های ۱۹۷۱ و ۱۹۷۲ به تدریج امکاناتی به زبان B افزوده شد و در نتیجه زبان جدیدی به وجود آمد که بعدها زبان C نامیده شد.
- در سال ۱۹۷۸ اولین نسخه از کتاب زبان برنامه نویسی سی [1] منتشر کرد.

[4] Algol

[1] The C Programming Language

زبان سی

- زبان سی به عنوان یکی از زبان‌های بسیار کارامد برای سال‌ها در کنار اسمبلی تنها زبان سطح بالایی بود که در توسعه سیستم عامل لینوکس استفاده می‌شد. در سال ۲۰۲۲ برای اولین بار از زبان سطح بالای راست [1] برای توسعه سیستم عامل لینوکس استفاده شد.

[1] Rust

زبان سی

- علاوه بر سیستم عامل‌ها، زبان سی در بسیاری از سیستم‌های نهفته [1] شامل سیستم‌های تعبیه شده در لوازم خانگی، ربات‌ها، سیستم‌های هواپیمایی و سیستم‌های کنترلی استفاده می‌شود.
- علت عمده اهمیت زبان سی در کارامدی بالا و سادگی استفاده از این زبان است.
- برای مثال در یک سامانهٔ کنترل ترافیکی کوچکترین تأخیری در سیستم می‌تواند خطرآفرین باشد و بنابراین نیاز به زبانی است که تا حد امکان با سرعت بالایی برنامه‌ها را اجرا کند.
- به عنوان مثال دیگر در سامانه‌های ارتباطی نیاز به ارسال و دریافت سریع اطلاعات صوتی و تصویری و کدگذاری و کدگشایی آنهاست و تأخیر در این سامانه‌ها باعث اختلال در ارتباط می‌شود و بنابراین به زبانی با سرعت اجرای بالا نیاز است.

[1] embedded systems

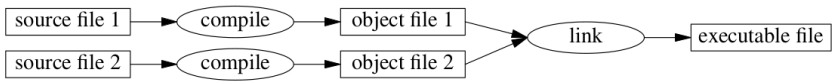
زبان سی

- زبان سی در زمان ابداع به شیوه‌های متفاوتی تعمیم داده شده و پیاده‌سازی شد و در سال ۱۹۸۹ توسط مؤسسهٔ استاندارد ملی آمریکا [1] به صورت استاندارد درآمد.

[1] Amercan National Standards Institute (ANSI)

زبان سی

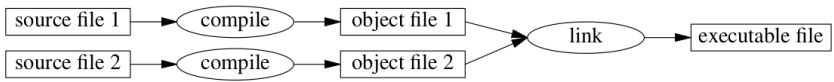
- کامپایلر سی متن برنامه را که در یک فایل منبع یا فایل سورس [1] نگهداری می‌شود به زبان ماشین ترجمه می‌کند و فایل‌هایی به نام فایل آبجکت [2] می‌سازد که حاوی برنامه به زبان ماشین مقصد برای اجرا است.
- سپس فایل‌های آبجکت باید توسط پیوند دهنده یا لینکر [3] به یکدیگر پیوند داده شوند و یک فایل اجرایی برای اجرا تهیه شود. معمولاً یک برنامه از تعداد زیادی فایل سورس تشکیل شده است.



[1] source file
[2] object file
[3] linker

زبان سی

- گاهی برای برنامه‌ای که توسط برنامه‌نویسان دیگر نوشته شده است فایل سورس در اختیار برنامه‌نویس قرار نمی‌گیرد بلکه فایل‌های آبجکت ارائه می‌شوند.
- در چنین مواردی لینکر وظیفه دارد فایل‌های آبجکت یک برنامه و فایل‌های آبجکت برنامه‌های مورد نیاز دیگر را به صورت یک فایل یکپارچه اجرای درآورد.



- در نهایت یک برنامه اجرایی [1] در قالب یک فایل اجرایی [2] برای یک پردازندهٔ مقصد تهیه می‌شود. این فایل قابل انتقال [3] نیست، بدین معنی که نمی‌توان آن را از یک ماشین یا سیستم عامل به یک ماشین یا سیستم عامل متفاوت دیگر انتقال داد و اجرا کرد.

[1] executable program
[2] executable file
[3] portable

الگوریتم

مقدمه

- کلمهٔ الگوریتم از نام دانشمند ایرانی محمدبن موسی الخوارزمی گرفته شده است.
- خوارزم منطقه‌ای است در آسیای مرکزی که در حال حاضر در ازبکستان و ترکمنستان قرار دارد و در کنار دریاچهٔ آرال (دریاچهٔ خوارزم) قرار گرفته است. خوارزمی کتاب الجبرو المقابله را نیز به تألیف رسانده است که کلمه جبر [1] در زبان انگلیسی نیز از همین کتاب گرفته شده است.
- تا سال ۱۹۵۰ کلمهٔ الگوریتم بیشتر برای الگوریتم اقلیدس [2] برای پیدا کردن بزرگ‌ترین مقسوم‌علیه مشترک [3] دو عدد به‌کار می‌رفت که در کتاب اصول اقلیدس [4] توصیف شده است.

---

[1] algebra
[2] Euclid's algorithm
[3] greatest common divisor
[4] Euclid's Element

- الگوریتم پیدا کردن بزرگ‌ترین مقسوم‌علیه مشترک را می‌توانیم به صورت زیر وصف کنیم.

۱. (پیدا کردن باقیمانده.) عدد m را بر n تقسیم می‌کنیم. فرض کنید باقیمانده r باشد خواهیم داشت $0 \leqslant r < n$

۲. (آیا باقیمانده صفر است؟) اگر $r = 0$ ، الگوریتم پایان می‌یابد و n جواب مسئله است.

۳. (کاهش.) قرار می‌دهیم $m \leftarrow n$ و $n \leftarrow r$ و به مرحلهٔ ۱ می‌رویم.

- الگوریتم در واقع یک روند [1] یا دستورالعمل [2] برای حل یک مسئلهٔ محاسباتی است.
- به طور غیر رسمی می‌توانیم بگوییم یک الگوریتم در واقع یک روند محاسباتی گام‌به‌گام است که مجموعه‌ای از مقادیر را که ورودی الگوریتم نامیده می‌شوند دریافت می‌کند و مجموعه‌ای از مقادیر را که خروجی الگوریتم نامیده می‌شوند در زمان محدود تولید می‌کند. بنابراین یک الگوریتم دنباله‌ای است از گام‌های محاسباتی که ورودی‌ها را به خروجی تبدیل می‌کند.

[1] procedure
[2] recipe

مقدمه

- می‌توان گفت یک الگوریتم ابزاری است برای حل یک مسئله محاسباتی معین.
- یک مسئله با تعدادی گزاره رابطهٔ بین ورودی‌ها و خروجی‌ها را در حالت کلی مشخص می‌کند. یک نمونه از مسئله، در واقع با جایگذاری اعداد و مقادیر برای مسئله کلی به دست می‌آید. یک الگوریتم روشی گام‌به‌گام را شرح می‌دهد که با استفاده از آن در حالت کلی برای همهٔ نمونه‌های یک مسئله، خروجی‌ها با دریافت ورودی‌ها تولید شوند. بنابراین روند یک الگوریتم در رابطهٔ بین ورودی‌ها و خروجی‌ها صدق می‌کند.
- به عنوان مثال، فرض کنید می‌خواهید دنباله‌ای از اعداد را با ترتیب صعودی مرتب کنید. این مسئله که مسئله مرتب سازی [1] نام دارد، یک مسئله بنیادین در علوم کامپیوتر به حساب می‌آید که منشأ به وجود آمدن بسیاری از روش‌های طراحی الگوریتم نیز می‌باشد.

[1] sorting problem

مقدمه

- مسئله مرتب سازی را به طور رسمی به صورت زیر تعریف می‌کنیم.
- ورودی مسئله مرتب سازی عبارت است از دنباله‌ای از n عدد به صورت $\langle a_1, a_2, ..., a_n \rangle$ و خروجی مسئله عبارت است از دنباله‌ای به صورت $\langle a'_1, a'_2, ..., a'_n \rangle$ که از جابجا کردن عناصر دنبالهٔ ورودی به‌دست آمده است به طوری که $a'_1 \leq a'_2 \leq ... \leq a'_n$.
- بنابراین به ازای دنباله ورودی $\langle 58, 42, 36, 42 \rangle$ دنباله خروجی $\langle 36, 42, 42, 58 \rangle$ جواب مسئله است.
- یک نمونه از یک مسئله [1] تشکیل شده است از یک ورودی معین و شرح ویژگی خروجی مسئله. بنابراین دنبالهٔ ورودی $\langle 58, 42, 36, 42 \rangle$ به علاوه شرح مسئله مرتب سازی یک نمونه از مسئلهٔ مرتب سازی نامیده می‌شود.

[1] instance of a problem

مقدمه

- بنابراین به طور خلاصه، یک مسئله تشکیل شده است از (۱) شرحی از چندین پارامتر یا متغیر آزاد، و (۲) شرحی از ویژگی‌هایی که جواب مسئله دارد.
- یک پارامتر یا متغیر آزاد کمیتی است که مقدار آن مشخص نشده و توسط حروف و یا کلمات، نامی بر آن نهاده شده است.
- یک نمونهٔ مسئله با تعیین مقادیر پارامترهای مسئله به دست می‌آید.
- یک الگوریتم، روندی گام به گام است برای پیدا کردن جواب یک مسئله است.

مقدمه

- سرعت اجرای مسئله مرتب سازی به اندازه ورودی یعنی تعداد عناصر دنباله نامرتب و روند الگوریتم بستگی دارد.
- الگوریتم‌های زیادی برای حل مسئله مرتب سازی وجود دارند که هر کدام می‌توانند مزایا و معایبی داشته باشند. به طور مثال یک الگوریتم از میزان حافظهٔ بیشتری استفاده می‌کند، اما زمان کمتری برای محاسبه نیاز دارد و الگوریتم دیگر با میزان حافظهٔ کمتر در زمان بیشتری محاسبه می‌شود که به فراخور نیاز می‌توان از یک از الگوریتم‌ها استفاده کرد.
- عوامل دیگری مانند معماری کامپیوتر، نوع پردازنده و میزان حافظه نیز در زمان اجرای یک الگوریتم مؤثرترند اما این عوامل فیزیکی هستند و صرف نظر از عوامل فیزیکی می‌توان الگوریتم‌ها را از لحاظ میزان حافظه مورد نیاز و زمان اجرا با یکدیگر مقایسه کرد.

مقدمه

- یک الگوریتم برای یک مسئله محاسباتی درست است اگر به ازای هر نمونه از مسئله که با تعدادی ورودی معین شده است، (۱) توقف کند، بدین که در زمان محدود به اتمام برسد و (۲) خروجی تعیین شده توسط شرح مسئله را تولید کند. یک الگوریتم درست در واقع یک مسئلهٔ محاسباتی را حل می‌کند.
- یک الگوریتم نادرست ممکن است به ازای برخی از ورودی‌ها توقف نکند یا ممکن است به ازای برخی از ورودی‌ها خروجی نادرست تولید کند.
- یک الگوریتم را می‌توان با استفاده از یک زبان طبیعی مانند فارسی یا انگلیسی توصیف کرد و یا برای توصیف آن از یک برنامهٔ کامپیوتری یا یک زبان ساده شده مانند فلوچارت یا شبه‌کد استفاده کرد. تنها نیازمندی یک الگوریتم توصیف دقیق گام‌های الگوریتم است و زبان مورد استفاده برای توصیف اهمیتی ندارد.

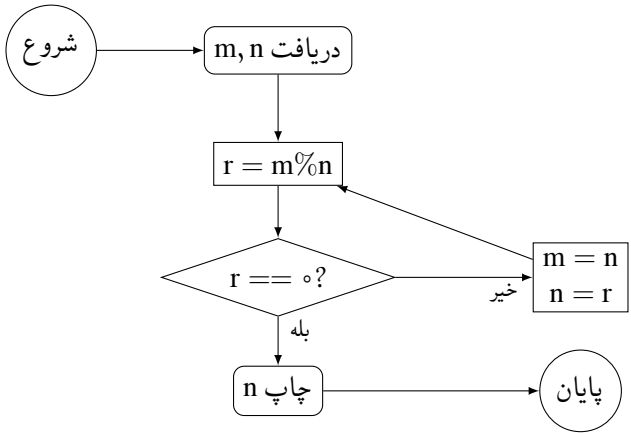
مقدمه

- یک الگوریتم را به صورت یک فلوچارت [1] می‌توانیم رسم کنیم.
- یک فلوچارت یا روندنما نموداری است که روند انجام کاری را نشان می‌دهد.
- یک فلوچارت،الگوریتم را به صورت تصویری به نمایش می‌گذارد. در یک فلوچارت معمولاً برای گام‌های محاسباتی از مستطیل و برای گام‌های شرطی از بیضی یا لوزی استفاده می‌شود. همچنین در گام‌هایی که ورودی از کاربر گرفته می‌شود یا خروجی برای نمایش به کاربر چاپ می‌شود از شکل هایی مانند متوازی‌الاضلاع یا شکل‌های قراردادی دیگر شبیه به مستطیل استفاده می‌شود. هر گام به گام بعدی توسط یک علامت فلش متصل می‌شود. شروع و پایان الگوریتم را معمولا با دایره نشان می‌دهند.

[1] flowchart

مقدمه

- برای مثال الگوریتم اقلیدس را می‌توان به صورت زیر رسم کرد.



- در الگوریتم‌ها معمولا از علامت ← یا =: یا = برای عملیات انتساب استفاده می‌شود. برای مثال $m \leftarrow n$ یا $m = n$ یعنی m را با مقدار فعلی n مقدار دهی می‌کنیم.
- معمولاً از علامت == برای تساوی استفاده می‌شود. برای مثال می‌توانیم بپرسیم آیا مقدار m برابراست با مقدار n و می‌نویسیم اگر $m == n$ یا $m == n?$.
- به عنوان مثال دیگر، برای افزایش مقدار یک متغیر به اندازهٔ یک واحد می‌نویسیم $n \leftarrow n + ۱$ یعنی مقدار n برابراست با مقدار فعلی n به علاوهٔ یک. معمولاً این عبارت را به این صورت می‌خوانیم : مقدار n برابر می‌شود با n+۱ . همچنین می‌توانیم بنویسیم $n = n + ۱$ که این عملیات انتساب با عملگر تساوی در ریاضی متفاوت است.

مقدمه

- در نشانه گذاری ریاضی معمولاً دنباله‌ها را با استفاده از اندیس‌ها نمایش می‌دهیم برای مثال دنباله $v_1, v_2, \cdots, v_n$ یک دنباله از n متغیر است. در الگوریتم‌ها معمولاً از عملگر زیرنویس [1] که با دو براکت باز و بسته [ ] نمایش داده می‌شود استفاده می‌کنیم. بنابراین i امین عنصر دنبالهٔ $v_1, \cdots, v_n$ را به صورت v[i] نمایش می‌دهیم.

[1] subscript

مقدمه

- الگوریتم‌ها در زمینه‌های زیاد و متنوعی کاربرد دارند.
- به عنوان مثال در پروژه ژنوم‌های انسانی هدف پیدا کردن الگوهای ژن‌ها در دی‌ان‌ای [1] انسان است که برای این کار از الگوریتم‌های کامپیوتری استفاده می‌شود. به عنوان چند مثال دیگر می‌توان از الگوریتم کوتاهترین مسیر برای مسیریابی بسته‌های اینترنتی در شبکه‌های کامپیوتری، الگوریتم‌های رمز نگاری برای تبادل امن اطلاعات، الگوریتم‌های تخصیص منابع و زمانبندی در کاربردهایی مانند زمانبندی پروازها و تخصیص خلبان و خدمه به هواپیماها با کمترین هزینه ممکن و الگوریتم‌های فشرده سازی داده‌ها نام برد.
- معمولاً یک مسئلهٔ محاسباتی راه حل‌های زیادی دارد که بنابر معیارهای مورد اهمیت برای استفاده کنندهٔ الگوریتم، الگوریتمی انتخاب می‌شود که در یک یا چند معیار مورد نظر بهترین باشد. برای مثال در یک سامانهٔ حمل و نقل هرچه به ازای یک سفر مسیر کوتاه‌تری طی شود، هزینه پایین می‌آید.

[1] DNA

الگوریتم

- الگوریتمی بنویسید که عدد n را دریافت کند و مجموع $n + \cdots + ۲ + ۱$ را چاپ کند.

الگوریتم

۱. عدد n را دریافت کن.
۲. اگر `n < 1` ، برو به ۷
۳. `num = 1` ، `sum = 0`
۴. `sum = sum + num`
۵. `num = num + 1`
۶. اگر `num > n` برو به ۸ در غیراینصورت برو به ۴
۷. چاپ کن عدد ورودی غیر معتبر است. پایان.
۸. مقدار `sum` را چاپ کن. پایان.

الگوریتم

- خط ۴ در این الگوریتم چند بار تکرار می‌شود؟

الگوریتم

- خطوط ۴ و ۵ و ۶ در این الگوریتم n بار تکرار می‌شوند.
- می‌گوییم این الگوریتم از مرتبهٔ n است، بدین معنی که اگر عدد n به الگوریتم داده شود، عمل محاسباتی جمع باید n بار تکرار شود. اگر هر عمل محاسباتی در زمان ۱ میلی ثانیه انجام شود، این الگوریتم در n میلی ثانیه محاسبات را انجام می‌دهد.
- آیا می‌توانید الگوریتم بهتری بنویسید که در زمان کمتری اجرا شود؟

الگوریتم

۱. عدد `n` را دریافت کن.

۲. اگر `n < 1` ، برو به ۴

۳. `sum = n (n + 1) / 2`

۴. چاپ کن عدد ورودی غیر متغیر است. پایان.

۵. مقدار `sum` را چاپ کن. پایان.

الگوریتم

- الگوریتمی بنویسید که عدد n را دریافت کند و زوج و فرد بودن آن را تعیین کند.

الگوریتم

۱. عدد `n` را دریافت کن.

۲. `r = n%2`

۳. اگر `r == 0` برو به ۴ در غیر اینصورت برو به ۵

۴. چاپ کن عدد زوج است. پایان.

۵. چاپ کن عدد فرد است. پایان.

الگوریتم

- الگوریتمی بنویسید که عدد n را دریافت کند و !n را چاپ کند.

الگوریتم

۱. عدد `n` را دریافت کن.
۲. اگر `n < 0` آنگاه برو به ۷
۳. `fact = 1` ، `i = 1`
۴. `fact = fact * i`
۵. `i = i + 1`
۶. اگر `i > n` آنگاه برو به ۸. در غیر اینصورت برو به ۴
۷. چاپ کن ورودی غیر معتبر است. پایان.
۸. مقدار `fact` را چاپ کن. پایان.

الگوریتم

- الگوریتمی بنویسید که عدد n را دریافت کرده و عدد فیبوناچی n ام را چاپ کند.

الگوریتم

۱. عدد `n` را دریافت کن.
۲. اگر `n < 0` چاپ کن عدد ورودی غیر معتبر است. پایان.
۳. اگر `n <= 1` عدد `n` را چاپ کن. پایان.
۴. `i = 1` ، `prev = 0` ، `curr = 1`
۵. `next = prev + curr`
۶. `prev = curr`
۷. `curr = next`
۸. `i = i + 1`
۹. اگر `i < n` برو به ۵
۱۰. عدد `curr` را چاپ کن. پایان.

الگوریتم

- الگوریتمی بنویسید که انتگرال $f(n)$ را به طور تقریبی از i تا j محاسبه کند.

الگوریتم

۱. اعداد `i` و `j` را دریافت کن.

۲. `d = 1.0`، `sum = 0`

۳. `sum = sum + f(i) * d`

۴. `i = i + d`

۵. اگر `i > j` آنگاه برو به ۶ در غیراینصورت برو به ۳

۶. مقدار `sum` را چاپ کن. پایان.

الگوریتم

- الگوریتمی بنویسید که عدد n را دریافت کرده، و ارقام آن را از سمت راست به چپ چاپ کند.

الگوریتم

۱. عدد `n` را دریافت کن.
۲. اگر `n < 0` چاپ کن عدد ورودی غیر معتبر است. پایان.
۳. `r = n % 10`
۴. `n = n / 10`
۵. رقم `r` را چاپ کن.
۶. اگر `n > 0` آنگاه برو به ۳.
۷. پایان.

الگوریتم

- الگوریتمی بنویسید که عدد n را دریافت کرده، آن را به مبنای ۲ تبدیل کند.

الگوریتم

۱. عدد `n` را دریافت کن.

۲. اگر `n < 0` چاپ کن ورودی غیر معتبر است. پایان.

۳. `res = 0` ، `base = 1`

۴. `r = n % 2`

۵. `n = n / 2`

۶. `res = res + base * r`

۷. `base = base * 10`

۸. اگر `n > 0` برو به ۴

۹. مقدار `res` را چاپ کن.

الگوریتم

- الگوریتمی بنویسید که یک عدد دودویی را دریافت کرده، به مبنای ۱۰ تبدیل کند.

الگوریتم

۱. عدد `n` را دریافت کن.

۲. `res = 0` ، `base = 1`

۳. `r = n % 10`

۴. `n = n / 10`

۵. `res = res + base * r`

۶. `base = base * 2`

۷. اگر `n > 0` برو به ۳.

۸. مقدار `res` را چاپ کن.

الگوریتم

- الگوریتمی بنویسید که دو عدد را دریافت کرده، کوچک‌ترین مضرب مشترک آنها را محاسبه کند.

الگوریتم

۱. دو عدد `m` و `n` را دریافت کن.

۲. اگر `m <= 0` یا `n <= 0` چاپ کن ورودی غیر معتبر است. پایان.

۳. بزرگ‌ترین مقسوم‌علیه مشترک `m` و `n` را با استفاده از الگوریتم ب.م.م محاسبه کن و برابر با `gcd` قرار بده.

۴. `lcm = (n * m) / gcd`

۵. مقدار `lcm` را چاپ کن.

الگوریتم

۱. دو عدد `m` و `n` را دریافت کن.

۲. اگر `m <= 0` یا `n <= 0` چاپ کن ورودی غیر معتبر است. پایان.

۳. اگر `m > n` آنگاه `m` و `n` را جابجا کن (`k=m; m=n; n=k`).

۴. `d = 1`

۵. `lcm = d * m`

۶. `d = d + 1`

۷. اگر `lcm % n != 0` آنگاه برو به ۵.

۸. مقدار `lcm` را چاپ کن.

الگوریتم

- الگوریتمی بنویسید که یک عدد را دریافت کرده بررسی کند آن عدد اول است یا خیر.

الگوریتم

۱. عدد `n` را دریافت کن.

۲. اگر `n<=1` چاپ کن عدد ورودی غیر معتبر است. اگر `n==2` برو به ۷.

۳. `d = 2`

۴. `r = n % d`

۵. `d = d + 1`

۶. اگر `r == 0` برو به ۸ در غیراینصورت اگر `d <= n/2` برو به ۴.

۷. عدد `n` اول است. پایان.

۸. عدد `n` اول نیست. پایان.

الگوریتم

- الگوریتمی بنویسید که مجموع دنبالهٔ $\cdots + ۱/۱۶ + ۱/۸ + ۱/۴ + ۱/۲ + ۱$ را محاسبه کند.

الگوریتم

۱. `sum=0.0 ; d=1.0 ; t=0.0001`

۲. `sum = sum + d`

۳. `d = d/2`

۴. اگر `d > t` برو به ۲

۵. مقدار `sum` را چاپ کن. پایان.

الگوریتم

- الگوریتمی بنویسید که یک دنباله از اعداد را دریافت کرده، آنها را مرتب کند.

الگوریتم

۱. عدد `n` (تعداد اعداد دنباله) را دریافت کن.

۲. `k = 1`

۳. عدد `A[k]` را دریافت کن.

۴. `k = k + 1`

۵. اگر `k <= n` برو به ۳

الگوریتم

۶. `i = 1`

۷. `j = i+1`

۸. اگر `A[i] > A[j]` مقدار `A[i]` و `A[j]` را جابجا کن (`k=A[i]; A[i]=A[j]; A[j]=k`).

۹. `j = j + 1`

۱۰. اگر `n => j` برو به ۸

۱۱. `i = i + 1`

۱۲. اگر `n > i` برو به ۷

الگوریتم

۱۳. `k = 1`

۱۴. عدد `A[k]` را چاپ کن.

۱۵. `k = k + 1`

۱۶. اگر `k <= n` برو به ۱۴.

برنامه‌نویسی سی

برنامه‌نویسی سی

- در این قسمت قصد داریم همهٔ ویژگی‌های مهم و ساختار کلی زبان سی را بدون ورود به جزئیات شرح دهیم.
- بنابراین در این قسمت هدف توصیف شرح کلی زبان سی است به طوری که بتوانیم با استفاده از آن الگوریتم‌های ساده را پیاده‌سازی کنیم. در فصل‌های بعدی بیشتر به شرح جزئیات خواهیم پرداخت.
- در این قسمت به طور خلاصه در مورد متغیرها و ثابت‌ها، محاسبات عددی، ساختارهای کنترلی، توابع و مقدمات ورودی و خروجی صحبت خواهیم کرد و قسمت‌های پیشرفته‌تر مانند اشاره‌گرها و ساختمان‌ها را در قسمت‌های بعد توضیح خواهیم داد.

برنامه‌نویسی سی

- اولین برنامه‌ای که در اولین درس همهٔ زبان‌های برنامه‌نویسی نوشته می‌شود برنامه‌ای است که جملهٔ "hello, world" را بر روی خروجی چاپ می‌کند.
- این برنامه در زبان سی به صورت زیر است.

```c
#include <stdio.h>
int main() {
  printf("hello, world\n");
}
```

- این برنامه در یک فایل سورس با پسوند c. ذخیره می‌شود. برای کامپایل این برنامه می‌توان از ابزار gcc استفاده کرد. دستور gcc hello.c یک فایل آبجکت a.out تولید می‌کند که قابلیت اجرا شدن دارد و جمله "hello, world" را چاپ می‌کند.

برنامه‌نویسی سی

- یک برنامهٔ سی به طور کلی از تعدادی تابع تشکیل شده است. یک تابع با استفاده از تعدادی دستورات محاسباتی و ساختارهای کنترلی تعدادی ورودی را دریافت کرده و عملیاتی را بر روی ورودی اعمال و صفر یا یک خروجی تولید می‌کند.
- تابع `main` تابع اصلی برنامه است یعنی یک برنامه با اجرای تابع `main` آغاز می‌شود. توابع می‌توانند نام‌های دلخواه داشته باشند، اما تابع `main` یک تابع ویژه است که باید برای شروع برنامه وجود داشته باشد. در دستورات تابع `main` می‌توانند توابع دیگر فراخوانی شوند.
- در ابتدای برنامه کتابخانه‌های مورد نیاز اضافه می‌شوند. هر کتابخانه شامل توابعی است که کاربرد خاصی دارند و وظایف خاصی انجام می‌دهند. برای مثال کتابخانه `stdio` یک کتابخانه استاندارد برای انجام عملیات ورودی خروجی [1] است.

[1] standard input output library

برنامه‌نویسی سی

- در اولین برنامه در تابع `main` از تابع `printf` برای چاپ یک رشته استفاده کردیم. یک تابع معمولاً تعدادی پارامتر [1] دریافت می‌کند و عملیات خود را بر روی ورودی‌ها انجام می‌دهد و خروجی تولید می‌کند. یک تابع همچنین می‌تواند بدون پارامتر باشد. متغیرهایی که به پارامترهای یک تابع ارسال می‌شوند را آرگومان [2] می‌نامیم.
- همهٔ دستورات یک تابع در بین در آکولاد بازو بسته {} قرار می‌گیرند.
- به تابع `printf` یک رشته (`"hello, world\n"`) به عنوان آرگومان ارسال می‌شود. یک رشته عبارتی است که از تعدادی حرف (کاراکتر) تشکیل شده است. کاراکتر `\n` برای اتمام خط فعلی و ادامهٔ چاپ با شروع خط بعدی استفاده می‌شود.
- هنگام چاپ رشته همچنین می‌توان از کاراکترهای ویژهٔ دیگر مانند `\t` برای فاصلهٔ ستونی، `\"` برای چاپ علامت نقل قول و `\\` برای چاپ بک‌اسلش [3] استفاده کرد.

[1] parameter
[2] argument
[3] backslash

متغیرها و ساختارهای کنترل

- حال می‌خواهیم برنامه‌ای نویسیم که دمای اندازه‌گیری شده در واحد فارنهایت را به سلسیوس تبدیل کند. فرمول تبدیل فارنهایت به سلیوس به صورت زیر است.

$$C = (5/9)(F-32)$$

متغیرها و ساختارهای کنترل

- این برنامه به صورت زیر نوشته می‌شود.

```
۱  #include <stdio.h>
۲  /* print Fahrenheit-Celsius table for fahr = 0, 20, ..., 300 */
۳  int main () {
۴    int fahr, celsius;
۵    int lower, upper, step;
۶    lower = 0; /* lower limit of temperature scale */
۷    upper = 300; /* upper limit */
۸    step = 20; /* step size */
۹    fahr = lower;
۱۰   while (fahr <= upper) {
۱۱       celsius = 5 * (fahr - 32) / 9;
۱۲       printf ("%d\t%d\n", fahr, celsius);
۱۳       fahr = fahr + step;
۱۴   }
۱۵ }
```

- پس از معرفی کتابخانه `stdio` در ابتدای برنامه (برای انجام عملیات چاپ بر روی صفحه نمایش)، توضیحاتی [1] درمورد برنامه در خط دوم داده شده است. این توضیحات، تنها برای شرح عملکرد برنامه استفاده می‌شوند و کامپایلر در هنگام کامپایل برنامه آنها را نادیده می‌گیرد. هر متنی بین دو علامت `*/` و `/*` قرار بگیرد جزء توضیحات محسوب می‌شود. توضیحات باعث می‌شوند عملکرد برنامه توسط برنامه‌نویسان دیگر بهتر فهمیده شود.

[1] comment

متغیرها و ساختارهای کنترل

- با استفاده از متغیرها [1] می‌توان مقادیری را توسط یک نام معین ذخیره و نگهداری کرد. مقدار متغیرها در طول برنامه قابل تغییر است.
- در زبان سی متغیرها به همراه نوع‌شان تعریف می‌شوند. یک متغیر می‌تواند از نوع عدد صحیح، عدد اعشاری، کاراکتر یا غیره باشد.
- در برنامه تبدیل فارنهایت به سلسیوس می‌خواهیم مقادیر فارنهایت و سلسیوس را نگهداری کنیم. همچنین می‌خواهیم کمترین و بیشترین مقدار دما به فارنهایت که در برنامه قابل تبدیل است را نگهداری کنیم و به علاوه می‌خواهیم برای نمونه برداری دما به فارنهایت فاصله بین دو دما را در یک متغیر ذخیره کنیم.
- بنابراین به ۵ متغیر نیاز داریم که همه از نوع صحیح هستند.

```
1 int fahr, celsius;
2 int lower, upper, step;
```

[1] variables

متغیرها و ساختارهای کنترل

- نوع `int` برای عدد صحیح، نوع `float` برای اعداد اعشاری، نوع `double` برای اعداد اعشاری با دقت دو برابر، و نوع `char` برای حروف به کار می‌رود.
- هر نوع متغیر اندازهٔ معینی دارد. برای مثال متغیر `int` در ماشین‌های ۳۲ بیتی ۲ بایت و در ماشین‌های ۶۴ بیتی ۴ بایت است.
- در ماشین‌های ۳۲ بیتی آدرس‌ها ۳۲ هستند و بنابراین تعداد $۲^{۳۲}$ مکان حافظه را می‌توان آدرس دهی کرد. در ماشین‌های ۶۴ بیتی تعداد $۲^{۶۴}$ آدرس حافظه وجود دارد.
- بنابراین یک ماشین ۳۲ بیتی حداکثر ۴ گیگابایت حافظه (معادل $۲^{۳۲}$ بایت) را می‌توانند آدرس دهی کند و یک ماشین ۶۴ بیتی حداکثر ۱۶ میلیون ترابایت.

متغیرها و ساختارهای کنترل

- یک متغیر صحیح در یک کامپیوتر ۶۴ بیتی ۴ بایت است. بنابراین برای اعداد صحیح مثبت می‌توان از صفر تا ۴۲۹۴۹۶۷۲۹۵ را در آنها ذخیره کرد.
- نوع‌های داده‌ای دیگر نیز وجود دارند که بعدها بیشتر درمورد آنها صحبت خواهیم کرد.
- در برنامهٔ تبدیل فارنهایت به سانتیگراد یک حلقه وجود دارد که در آن حلقه می‌خواهیم یک روند را تکرار کنیم. به ازای هر یک از مقادیر در واحد فارنهایت، عملیات لازم برای تبدیل به سانتی‌گراد یکسان است، پس این عملیات به ازای همهٔ اعداد باید تکرار شود. این حلقه باید تا زمانی تکرار شود که مقدار متغیر `fahr` کمتر از کران بالای `upper` باشد. بنابراین حلقه به صورت `while (fahr <= upper){}` نوشته می‌شود یعنی تا زمانی که `fahr` کوچکتر یا مساوی `upper` باشد عملیات تکرار شود.

- اگر بدنهٔ حلقه شامل تنها یک دستور باشد می‌توانیم از علامت آکولاد استفاده نکنیم. همچنین معمولاً برای خوانایی یک برنامه از دندانه‌گذاری [1] استفاده می‌کنیم، بدین معنی که دستورات درون حلقه، با چند خط فاصله شروع می‌شوند.
- تبدیل فارنهایت به سلسیوس با استفاده از دستور `celsius = 5 * (fahr - 32) / 9` انجام می‌شود. دلیل اینکه مقدار `(fahr - 32)` را ابتدا در ۵ ضرب و سپس بر ۹ تقسیم می‌کنیم این است که در غیراینصورت مقدار `5/9` که تقسیم دو عدد صحیح است برابر است با عدد صحیح صفر خواهد بود و مقدار `(fahr - 32)*(5/9)` همیشه صفر خواهد بود.

[1] indentation

متغیرها و ساختارهای کنترل

- در برنامه تبدیل فارنهایت به سلسیوس همچنین از تابع `printf` استفاده کردیم. این تابع برای چاپ یک رشته استفاده می‌شود. هریک از کلمات این رشته، یا به عبارت دیگر زیررشته‌ها، که با علامت درصد آغاز می‌شود، با مقادیر پارامترهای دوم به بعد جایگزین می‌شود.
- در مثال قبل دو زیر رشتهٔ `%d` که در آرگومان اول به تابع چاپ ارسال شده است با مقادیر آرگومان‌های دوم و سوم جایگزین می‌شود و بنابراین مقادیر فارنهایت و سلسیوس در خروجی چاپ می‌شوند.
- تابع `printf` جزئی از زبان سی نیست بلکه تابعی در کتابخانه استاندارد مربوط به ورودی و خروجی `stdio` است.

- ورودی اول تابع `printf` رشته‌ای است که در خروجی استاندارد چاپ می‌شود. این رشته می‌تواند شامل زیر رشته‌هایی باشد که نحوهٔ نمایش (فرمت) [1] خروجی را تعیین می‌کند. این زیر رشته‌ها که با علامت % شروع می‌شوند را تعیین کنندهٔ فرمت [2] می‌نامیم.
- تعیین کنندهٔ فرمت با مقادیر ورودی‌های دوم به بعد تابع `printf` جایگزین می‌شود و اعداد و رشته‌ها را با فرمت تعیین شده در خروجی استاندارد چاپ می‌کند.
- یک تعیین کنندهٔ فرمت می‌تواند `c%` برای چاپ کاراکتر، `d%` برای چاپ اعداد صحیح دهدهی، `f%` برای چاپ اعداد اعشاری باشد.

[1] format
[2] format specifier

- برنامهٔ تبدیل فارنهایت به سلسیوس می‌تواند خروجی را به گونه‌ای چاپ کند که هر دو ستون اعداد (مقادیر فارنهایت و سلسیوس) از سمت راست همتراز شوند. برای این کار از دستور زیر استفاده می‌کنیم.

```
printf("%3d %6d \n" , fahr, celsius);
```

- در این دستور عدد ۳ قبل از حرف d تعیین می‌کند که عرض ستون چاپ برای عدد صحیح اول ۳ باشد.
- همچنین در این برنامه می‌توانیم از نوع اعداد اعشاری به جای اعداد صحیح برای مقادیر استفاده کنیم.

- برای چاپ کردن اعداد اعشاری از f% استفاده می‌کنیم. همچنین می‌توانیم تعیین کنیم اعداد اعشاری با چه دقتی چاپ شوند. برای مثال در دستور زیر دمای سلسیوس با ۱ رقم بعد از اعشار چاپ می‌شود.

```
printf("%3.0f %6.1f \n" , fahr, celsius);
```

- همچنین برای تبدیل فارنهایت و سلسیوس می‌توانیم بنویسیم.

```
celsius = ( 5.0 / 9.0 ) * ( fahr - 32.0 )
```

- در اینجا با تقسیم عدد اعشاری 5.0 بر 9.0 یک عدد اعشاری به دست می‌آید و محاسبات به درستی انجام می‌شود.

- یک برنامه می‌تواند معمولاً به شکل‌های مختلف نوشته شود. برای مثال چندین نوع ساختار حلقه در زبان سی وجود دارد که می‌توان آنها را به جای یکدیگر استفاده کرد.
- برنامهٔ تبدیل دماهای فارنهایت به سلسیوس به شکلی دیگر در زیر نوشته شده است.

```
۱ #include <stdio.h>
۲ /* print Fahrenheit-Celsius table */
۳ int main ()
۴ {
۵   int fahr;
۶   for (fahr = 0; fahr <= 300; fahr = fahr + 20)
۷     printf ("%3d %6.1f\n", fahr, (5.0 / 9.0) * (fahr - 32));
۸ }
```

نماد ثابت

- در برنامه قبل از اعداد ۳۰۰ و ۲۰ در برنامه بدون نام استفاده کردیم. این نوع برنامه‌نویسی گرچه برنامه را مختصر می‌کند، ولی باعث می‌شود خواندن برنامه برای دیگر برنامه‌نویسان مشکل شود.
- با استفاده از کلید واژهٔ `#define` می‌توانیم یک نماد ثابت [1] تعریف کنیم.

[1] constant

نماد ثابت

- برای مثال در برنامه قبل می‌توانستیم اعداد کران بالا و کران پایین و مقدار افزایش را به صورت زیر تعریف کنیم.

```
۱  #include <stdio.h>
۲  #define LOWER 0 /* lower limit of table */
۳  #define UPPER 300 /* upper limit */
۴  #define STEP 20 /* step size */
۵  /* print Fahrenheit-Celsius table */
۶  int main ()
۷  {
۸    int fahr;
۹    for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
۱۰     printf ("%3d %6.1f\n", fahr, (5.0 / 9.0) * (fahr - 32));
۱۱ }
```

ورودی و خروجی

- ورودی و خروجی در زبان سی به صورت جریان‌های متنی [1] یا استریم‌های متنی هستند. یک استریم متنی دنباله‌ای است از تعدادی رشته که می‌توانند از یکدیگر با کاراکتر خط جدید [2] جدا شده باشند. هر رشته شامل تعدادی حرف یا کاراکتر است.
- در کتابخانهٔ استاندارد تعدادی تابع برای دریافت ورودی و چاپ خروجی طراحی شده‌اند. برای مثال تابع `getchar()` حرف بعدی را از ورودی استاندارد می‌خواند و به عنوان یک متغیر از نوع کاراکتر بازمی‌گرداند. همچنین تابع `putchar(c)` محتوای یک کاراکتر را در خروجی استاندارد چاپ می‌کند.

[1] text streams
[2] newline character

- برای مثال می‌خواهیم برنامه‌ای بنویسیم که یک کاراکتر را از ورودی دریافت کند و در صورتی که ورودی، کاراکتر پایان فایل [1] نبود، آن کاراکتر را در خروجی چاپ کند.

[1] end of file character

ورودی و خروجی

- این برنامه به صورت زیر است :

```
۱  #include <stdio.h>
۲  /* copy input to output; */
۳  int main ()
۴  {
۵    int c;
۶    c = getchar();
۷    while (c != EOF)
۸      {
۹        putchar(c);
۱۰       c = getchar();
۱۱     }
۱۲ }
```

ورودی و خروجی

- در برنامهٔ قبل خروجی `getchar()` را به جای یک کاراکتر برابر با یک عدد صحیح `int` قرار دادیم. باید دقت کرد که یک کاراکتر در واقع یک عدد را در حافظه نگهداری می‌کند بنابراین `char` و `int` می‌توانند به جای یکدیگر استفاده شوند. البته نوع کاراکتر یک بایت است و نوع عدد صحیح ۲ یا ۴ بایت.
- در ورودی استاندارد از `Ctrl + D` برای وارد کردن کاراکتر پایان فایل استفاده می‌کنیم.

- برنامه قبل را می‌توانیم به صورت مختصرتر به صورت زیر بنویسیم :

```
۱ #include <stdio.h>
۲ /* copy input to output; 2nd version */
۳ int main ()
۴ {
۵   int c;
۶   while ((c = getchar ()) != EOF)
۷     putchar (c);
۸ }
```

- در واقع یک عبارت انتساب دارای یک مقدار است که مقدار آن برابر با مقدار سمت چپ عبارت انتساب است. بنابراین می‌توانیم مقدار (`c = getchar()`) را با `EOF` مقایسه کنیم.
- همچنین پرانتزگذاری عبارت (`c = getchar()`) `!= EOF` با اهمیت است. عملگرها در زبان دارای تقدم یا اولویت [1] هستند. عملگرهای با اولویت بالاتر در یک عبارت زودتر اجرا می‌شوند. اولویت `!=` از اولویت `=` بالاتر است. بنابراین `c = getchar() != EOF` معادل است با

```
c = (getchar() != EOF)
```

- معنی این عبارت این است که مقدار عبارت `getchar() != EOF` سنجیده شود (این مقدار یا برابر با صفر است یا یک) و سپس مقدار به دست آمده در متغیر `c` ذخیره شود.

[1] precedeuce

- حال می‌خواهیم برنامه‌ای بنویسیم که کاراکترهای ورودی را بخواند و تعداد آنها را در پایان چاپ کند. این برنامه به صورت زیر است.

```
۱  #include <stdio.h>
۲  /* count characters in input; 1st version */
۳  int main ()
۴  {
۵    long nc;
۶    nc = 0;
۷    while (getchar() != EOF)
۸      ++nc;
۹    printf ("%ld\n", nc);
۱۰ }
```

ورودی و خروجی

- عملگر ++ یک واحد به مقدار متغیر اضافه می‌کند. می‌توانیم عبارت nc++ را به صورت nc = nc+1 نیز بنویسیم.
- همچنین ++nc یک واحد به متغیر می‌افزاید. تفاوت افزایش پیشوندی و پسوندی در این است که در عبارت پیشوندی x = ++nc ابتدا مقدار متغیر یک واحد افزایش پیدا می‌کند و سپس عملیات انتساب انجام می‌شود ولی در عبارت پسوندی x = nc++ ابتدا عملیات انتساب انجام می‌شود و سپس متغیر یک واحد افزایش می‌یابد.
- همچنین عملگر -- یک واحد از مقدار یک متغیر می‌کاهد.
- برای ذخیرهٔ متغیر nc از نوع int استفاده نکردیم زیرا ممکن است تعداد کاراکترها آنقدر زیاد باشد که در متغیر int نگنجد. یک متغیر از نوع long یا عدد صحیح بزرگ در کامپیوترهای ۳۲ بیتی ۴ بایت و در کامپیوترهای ۶۴ بیتی ۸ بایت است.
- برای چاپ یک متغیر از نوع long در تابع چاپ از تعیین کنندهٔ فرمت %ld استفاده می‌کنیم.

- می‌توانستیم از نوع داده‌ای double نیز به صورت زیر در یک حلقهٔ for استفاده کنیم.

```
۱ #include <stdio.h>
۲ /* count characters in input; 2nd version */
۳ int main ()
۴ {
۵   double nc;
۶   for (nc = 0; getchar () != EOF; ++nc);
۷   printf ("%.0f\n", nc);
۸ }
```

ورودی و خروجی

- بدنهٔ حلقهٔ `for` در مثال قبل خالی است، زیرا در تعریف حلقه همهٔ کارها انجام می‌شود.
- همچنین در مثال‌های قبل اگر هیچ کاراکتری وارد نشود، تعداد کاراکترهای ورودی صفر اعلام خواهد شد که نتیجه مورد نظر است.

ورودی و خروجی

- حال می‌خواهیم برنامه‌ای بنویسیم که تعداد خط‌های برنامه را بشمارد. در اینجا باید بررسی کنیم آیا یک حرف وارد شده برابر کاراکتر خط جدید \n است یا خیر.
- این برنامه به صورت زیر نوشته می‌شود.

```
۱  #include <stdio.h>
۲  /* count lines in input */
۳  int main ()
۴  {
۵    int c, nl;
۶    nl = 0;
۷    while ((c = getchar ()) != EOF)
۸      if (c == '\n')
۹        ++nl;
۱۰   printf ("%d\n", nl);
۱۱ }
```

- عملگر == به معنای برابری است، در حالی که دیدیم عملگر = به معنی انتساب است. ممکن است با اشتباه در استفاده این عملگرها برنامه‌ای به دست آید که به درستی کامپایل می‌شود ولی نتیجه درست به دست نمی‌دهد.
- یک کاراکتر را با یک علامت نقل قول نشان می‌دهیم. بنابراین `'\n'` به معنای کاراکتر خط جدید است. هر کاراکتری یک کد معادل استاندارد اسکی [1] دارد. برای مثال کد معادل `'A'` برابر است با ۶۵.
- کاراکتر `'\n'` برابر با کد اسکی ۱۰ است. توجه کنید که `'\n'` یک کاراکتر است نه دو کاراکتر. برای نمایش حرف `n` از `'n'` و برای نمایش حرف \ از `'\\'` استفاده می‌کنیم.

[1] American Standard Code for Information Interchange (ASCII)

ورودی و خروجی

- حال فرض کنید می‌خواهیم تعداد خطوط، کلمات و حروف را در ورودی بشماریم. کلمات از یکدیگر با کاراکتر خط فاصله یا کاراکتر خط جدید یا ستون جدید جدا می‌شوند.

- این برنامه به صورت زیر نوشته می‌شود. برنامه wc در لینوکس همین عملیات را انجام می‌دهد.

```
۱ #include <stdio.h>
۲ #define IN 1     /* inside a word */
۳ #define OUT 0    /* outside a word */
۴ /* count lines, words, and characters in input */
۵ int main ()
۶ {
۷   int c, nl, nw, nc, state;
۸   state = OUT;
۹   nl = nw = nc = 0;
```

```
11    while ((c = getchar ()) != EOF)
12      {
13            ++nc;
14            if (c == '\n')
15                ++nl;
16            if (c == ' ' || c == '\n' || c == '\t')
17                state = OUT;
18            else if (state == OUT)
19                {
20                      state = IN;
21                      ++nw;
22                }
23      }
24    printf ("%d %d %d\n", nl, nw, nc);
25  }
```

ورودی و خروجی

- در این برنامه با استفاده از یک متغیر `state` بررسی می‌کنیم آیا در حین پردازش یک کلمه هستیم یا کلمهٔ قبلی به پایان رسیده است. برای این کار از دو مقدار `IN` و `OUT` که در واقع مقادیر ۰ و ۱ هستند استفاده می‌کنیم. استفاده از این نام‌ها به جای مقادیر برنامه را خواناتر می‌کند.

- گرچه در برنامه‌های کوچک ممکن است خوانایی برنامه نسبت به حجم برنامه و میزان استفاده از حافظه اهمیت کمتری داشته باشد ولی در برنامه‌های بزرگ‌تر خوانایی برنامه اهمیت بسیار زیادی پیدا می‌کند زیرا هزینهٔ نگهداری برنامه نیز یکی از معیارهای مهم در طراحی برنامه است. هرچه یک برنامه خواناتر باشد، پیدا کردن مشکلات و همچنین ایجاد تغییرات در آن آسان‌تر می‌شود و بنابراین هزینهٔ نگهداری آن پایین می‌آید.

- در ابتدای برنامه از عبارت انتساب `nl = nw = nc = 0;` استفاده کردیم. با توجه به این‌که وابستگی عملگر [1] تساوی از راست به چپ است، این عبارت به صورت `nl = (nw = (nc = 0));` ترجمه می‌شود.

[1] operator associativity

- عملگر || به معنی فصل منطقی است که یای منطقی نیز نامیده می‌شود. عطف منطقی به صورت && است. اولویت عطف بالاتر از فصل است.
- مقدار یک عبارت منطقی از چپ به راست ارزیابی می‌شود و به محض اینکه مقدار منطقی آن تعیین شده محاسبات ادامه پیدا نمی‌کند. برای مثال در برنامه قبل در عبارت
`if ( c==' ' || c=='\n' || c='\t' )`
اگر مقدار c برابر با کاراکتر خط فاصله بود ارزیابی عبارت منطقی ادامه پیدا نمی‌کند زیرا مقدار آن در هر صورت برابر با ۱ خواهد بود.

- در زبان سی ساختار شرطی به صورت زیر نوشته می‌شود.

```
if (expression)
    statement1
else
    statement2
```

- این بدان معنی است که اگر مقدار منطقی `expression` برابر صحیح بود عبارت یا مجموعه عبارات `statement1` اجرا می‌شود و در غیر اینصورت عبارت مجموعه عبارات `statement2` اجرا می‌شوند.

آرایه‌ها

- حال فرض کنید می‌خواهیم برنامه‌ای بنویسیم که میانگین ۵ عدد را پیدا کند.
- می‌توانیم برای هر یک از این ۵ عدد یک متغیر در نظر بگیریم، اما ممکن است در آینده بخواهیم میانگین ۵۰۰۰ عدد را محاسبه کنیم.
- در چنین مواردی از آرایه‌ها استفاده می‌کنیم. یک آرایه متغیری است که می‌تواند تعدادی مقادیر از یک نوع را در خود نگهداری کند.

آرایه‌ها

- برنامه محاسبه میانگین را به صورت زیر می‌نویسیم.

```
۱  #include <stdio.h>
۲  int main() {
۳      float marks[5];
۴      marks[0] = 19;
۵      marks[1] = 20;
۶      marks[2] = 18;
۷      marks[3] = 15;
۸      marks[4] = 16.5;
۹      float sum = 0;
۱۰     for (int i=0; i<5; i++) {
۱۱             sum = sum + marks[i];
۱۲     }
۱۳     float average = sum / 5;
۱۴     printf("Average : %.2f\n", average);
۱۵ }
```

- عناصر آرایه را به صورت زیر نیز می‌توانیم مقداردهی اولیه کنیم.

```c
#include <stdio.h>
int main() {
    float marks[5] = {19, 20, 18, 15, 16.5};
    float sum = 0;
    for (int i=0; i<5; i++) {
            sum = sum + marks[i];
    }
    float average = sum / 5;
    printf("Average : %.2f\n", average);
}
```

آرایه‌ها

- حال فرض کنید می‌خواهیم برنامه‌ای بنویسیم که تعداد رخداد هر رقم و حروف خطوط فاصله را در یک متن بشمارد.
- برای شمردن ارقام به ۱۰ متغیر نیاز داریم. به جای ۱۰ متغیر می‌توانیم از یک آرایه با ۱۰ عنصر به صورت زیر استفاده کنیم.

```c
#include <stdio.h>
/* count digits, white space, others */
int main ()
{
  int c, i, nwhite, nother;
  int ndigit[10];
  nwhite = nother = 0;
  for (i = 0; i < 10; ++i)
    ndigit[i] = 0;
```

```
11    while ((c = getchar ()) != EOF)
12      if (c >= '0' && c <= '9')
13        ++ndigit[c - '0'];
14      else if (c == ' ' || c == '\n' || c == '\t')
15        ++nwhite;
16      else
17        ++nother;
18    printf ("digits =");
19    for (i = 0; i < 10; ++i)
20      printf (" %d", ndigit[i]);
21    printf (", white space = %d, other = %d\n", nwhite, nother);
22  }
```

آرایه‌ها

- متغیری از نوع آرایه به صورت `int ndigit[10]` تعریف کردیم که شامل ۱۰ عنصر آن یک عدد صحیح است. هر عنصر آن یک عدد صحیح است. برای مثال `ndigit[0]` یک عدد صحیح است برای شمردن ارقام صفر، و به همین ترتیب `ndigit[9]` یک عدد صحیح است برای شمردن ارقام ۹ در متن وارد شده.
- اندیس یک آرایه همیشه یک عدد صحیح است.
- همانطور که گفتیم یک کاراکتر یک مقدار عددی دارد بنابراین می‌توانیم بنویسیم `if(c >='0' && c<='9')` بدین معنی که اگر معادل عددی حرف وارد شده از معادل عددی حرف `'0'` بیشتر و از معادل عددی حرف `'9'` کمتر است.
- فاصلهٔ یک کاراکتر با کاراکتر `'0'` معادل مکان آن در آرایه خواهد بود زیرا معادل عددی کاراکترهای ارقام به ترتیب به صورت اعداد متوالی هستند. پس می‌توانیم بنویسیم `ndigit[C-'0']`
- برای دستورات شرطی پی‌درپی از `if` و `else if` استفاده کنیم.

توابع

- یک تابع تعدادی دستورات را به صورت یک گروه در می‌آورد و با یک نام نامگذاری می‌کند به طوری که با فراخوانی نام تابع تمام دستورات اجرا می‌شوند.
- در فراخوانی یک تابع درواقع نیازی نداریم بدانیم تابع و عملیات آن چگونه اجرا می‌شوند، بلکه تنها کافی است بدانیم آن تابع چه کاری انجام می‌دهد.
- با استفاده از توابع همچنین می‌توانیم که برنامه نظم بیشتری بدهیم.

توابع

- فرض کنید می‌خواهیم تابعی بنویسیم که مقدار x^y را محاسبه می‌کند. در واقع این تابع دو متغیر `x` و `y` را دریافت می‌کند و مقداری را از تابع `pow(x,y)` بازمی‌گرداند.

توابع

- این تابع به صورت زیر پیاده‌سازی و فراخوانی می‌شود.

```c
#include <stdio.h>
int power (int m, int n);
/* test power function */
int main ()
{
  int i;
  for (i = 0; i < 10; ++i)
    printf ("%d %d %d\n", i, power (2, i), power (-3, i));
  return 0;
}
```

```
۱۱  /* power: raise base to n-th power; n >= 0 */
۱۲  int power (int base, int n)
۱۳  {
۱۴    int i, p;
۱۵    p = 1;
۱۶    for (i = 1; i <= n; ++i)
۱۷      p = p * base;
۱۸    return p;
۱۹  }
```

توابع

- یک تابع در حالت کلی تشکیل شده است از یک نوع داده بازگشتی، نام تابع، متغیرهای ورودی که پارامتر نامیده می‌شوند و بدنهٔ تابع.
- یک تابع در یک فایل تعریف می‌شود و نمی‌توان یک تابع را به دو قسمت تقسیم کرد. وقتی یک تابع در یک فایل جداگانه از تابع فراخوانی کننده قرار می‌گیرند، برای استفاده از آن باید فایل‌های آبجکت ساخته شده به یکدیگر پیوند داده شوند.
- تابع `power` در مثال قبل دارای دو پارامتر ورودی از نوع عدد صحیح و یک مقدار بازگشتی از نوع عدد صحیح است.
- متغیرهایی که در تعریف تابع به کار می‌روند را پارامتر و متغیرهایی که در فراخوانی تابع به تابع ارسال می‌شوند را آرگومان می‌نامیم.
- کلمهٔ کلیدی `return` برای بازگرداندن یک مقدار از تابع به کار می‌رود.

توابع

- در مثال قبل تابع `main` هم یک مقدار باز می‌گرداند. مقدار صفر نشان دهنده این است که برنامه بدون خطا متوقف شده است.
- یک تابع را می‌توانیم در یک قسمت اعلام کنیم و سپس در قسمتی دیگر تعریف کنیم. برای اعلام [1] تابع باید نوع خروجی، نام تابع و پارامترهای ورودی و نوع آنها مشخص شود. در اعلام تابع درواقع تنها پروتوتایپ تابع [2] مشخص می‌شود.
- یک تابع پس از اعلام باید تعریف شود، در تعریف تابع [3] بدنهٔ تابع نیز باید مشخص شود.
- در اعلام تابع، الزامی به ذکر نام پارامترها وجود ندارد. بنابراین می‌توانیم بنویسیم.

```
int power (int, int);
```

[1] declare
[2] function prototype
[3] definition

فراخوانی با مقدار

- در زبان سی فراخوانی توابع به طور پیش فرض با مقدار است، بدین معنی که مقدار آرگومان‌ها در مقدار پارامترها کپی می‌شوند و تنها مقدار آرگومان‌ها در دسترس است نه مکان حافظه و آدرس آنها بنابراین متغیرهای ارسال شده به عنوان آرگومان در بدنه تابع در دسترس نیستند.
- فراخوانی با مقدار [1] درکنار فراخوانی با ارجاع [2] دو نوع فراخوانی در زبان سی هستند. فراخوانی به طور پیش‌فرض با مقدار است. در مورد فراخوانی با ارجاع در آینده توضیح خواهیم داد.

[1] call by value
[2] call by reference

فراخوانی با مقدار

- در مثال زیر، مقدار n در تابع کاهش می‌یابد ولی از آنجایی که n یک کپی از آرگومان دوم ارسال شده به تابع است، مقدار آرگومان دوم تغییر نخواهد کرد.

```
۱ /* power: raise base to n-th power; n >= 0; version 2 */
۲ int power (int base, int n)
۳ {
۴   int p;
۵   for (p = 1; n > 0; --n)
۶     p = p * base;
۷   return p;
۸ }
```

فراخوانی با مقدار

- وقتی نیاز داشته باشیم مقدار یک آرگومان را تغییر دهیم باید از فراخوانی با ارجاع استفاده کنیم. در فراخوانی با ارجاع پارامترها به صورت اشاره‌گر تعریف می‌شوند.
- وقتی یک آرایه به عنوان آرگومان به یک تابع ارسال می‌شود، فراخوانی با ارجاع است و نام تابع به آدرس اولین عنصر آرایه که به عنوان آرگومان ارسال شده است اشاره می‌کند. عناصر یک آرایه از آرگومان به پارامتر کپی نمی‌شوند.

آرایهٔ حروف

- آرایه‌ای که در زبان سی بیشترین استفاده را دارد، آرایهٔ حروف است. یک آرایهٔ حروف، آرایه‌ای است که نوع داده‌ای عناصر آن حروف یا کاراکترها هستند.
- می‌خواهیم برنامه‌ای بنویسیم که متنی را به صورت دنباله‌ای از خطوط دریافت می‌کند و بلندترین خط را چاپ می‌کند.

آرایهٔ حروف

- این برنامه را می‌توانیم ابتدا به صورت شبه‌کد بنویسیم :

۱. تا وقتی که یک خط دیگر وجود دارد.
۲. اگر طول خط فعلی از طول بلندترین خطی که تاکنون وارد شده بیشتر است.
۳. خط فعلی را ذخیره کن.
۴. طول خط فعلی را ذخیره کن.
۵. بلندترین خط را چاپ کن.

آرایهٔ حروف

- در این برنامه طرح کلی برنامه را در چند خط بیان کردیم. در هریک از این خطوط تنها آنچه باید انجام شود، بدون شرح چگونگی انجام آن توصیف شد. پس هریک از این قسمت‌ها را می‌توانیم به صورت یک تابع بنویسیم.
- ابتدا به تابعی نیاز داریم که خط بعدی را دریافت کند.
- تابعی با نام `getline` تعریف کنیم و در تعریف این تابع سعی می‌کنیم آن را تا حد امکان به صورت عمومی تعریف کنیم یعنی تابع را به صورتی تعریف کنیم که در برنامه‌های مختلف بتواند استفاده شود.

آرایهٔ حروف

- برای مثال تابع `getline` را طوری طراحی می‌کنیم که طول یک خط را بازگرداند و در صورتی که به پایان متن رسیده‌ایم و خطی وجود ندارد مقدار صفر را بازگرداند. توجه کنید اگر در خطی هیچ کاراکتری وجود نداشته باشد، کاراکتر `'\n'` وجود دارد پس طول آن برابر با ۱ است.
- همچنین وقتی طول یک خط از طولانی‌ترین خط بیشتر باشد باید آن را در مکانی ذخیره کنیم. پس نیاز به یک تابع کپی `copy` داریم که یک خط را در یک متغیر که آرایه‌ای از کاراکترها ذخیره کند.

آرایهٔ حروف

- این برنامه را به صورت زیر می‌توانیم بنویسیم.

```
۱ #include <stdio.h>
۲ #define MAXLINE 1000
۳ /* maximum input line length */
۴ int _getline (char line[], int maxline);
۵ void _copy (char to[], char from[]);
```

```
۶  /* print the longest input line */
۷  int main ()
۸  {
۹    int len;
۱۰ /* current line length */
۱۱   int max;
۱۲ /* maximum length seen so far */
۱۳   char line[MAXLINE];
۱۴ /* current input line */
۱۵   char longest[MAXLINE];^^I/* longest line saved here */
۱۶   max = 0;
۱۷   while ((len = _getline (line, MAXLINE)) > 0)
۱۸     if (len > max)
۱۹       {
۲۰     max = len;
۲۱     _copy (longest, line);
۲۲       }
```

آرایهٔ حروف

```
۲۴    if (max > 0)       /* there was a line */
۲۵          printf ("%s", longest);
۲۶    return 0;
۲۷  }
۲۸  /* getline: read a line into s, return length */
۲۹  int
۳۰  _getline (char s[], int lim)
۳۱  {
۳۲    int c, i;
۳۳    for (i = 0; i < lim - 1 && (c = getchar ()) != EOF && c != '\n'; ++i)
۳۴      s[i] = c;
۳۵    if (c == '\n')
۳۶      {
۳۷        s[i] = c;
۳۸        ++i;
۳۹      }
```

آرایهٔ حروف

```
۴۰    s[i] = '\0';
۴۱    return i;
۴۲  }
۴۳  /* copy: copy 'from' into 'to'; assume to is big enough */
۴۴  void _copy (char to[], char from[])
۴۵  {
۴۶    int i;
۴۷    i = 0;
۴۸    while ((to[i] = from[i]) != '\0')
۴۹      ++i;
۵۰  }
```

آرایهٔ حروف

- اولین پارامتر تابع `getline` یک آرایه است که عناصر آن را کاراکترها تشکیل داده‌اند. و دومین پارامتر آن یک عدد صحیح است که ماکزیمم طول آرایه را مشخص می‌کند. از آنجایی که طول آرایه در `main` مشخص می‌شود و تابع `getline` از آن مطلع نیست، بنابراین نیاز داریم این مقدار را به عنوان آرگومان به تابع `getline` ارسال کنیم.
- همچنین تابع `getline` یک عدد صحیح به تابع `main` بازمی‌گرداند که طول خط خوانده شده از ورودی است.
- تابع `getline` کاراکتر `'\0'` را در پایان آرایه قرار می‌دهد. این کاراکتر را کاراکتر تهی [1] نیز می‌نامیم. کاراکتر تهی بدین معنی است که رشته به پایان رسیده است. دنباله‌ای از حروف که با علامت پایان مشخص می‌شوند را رشته [2] می‌نامیم.

[1] null character
[2] string

آرایهٔ حروف

- در زبان سی و کتابخانه‌های استاندارد زبان سی این قرار داد استفاده شده است که یک رشته با کاراکتر تهی پایان می‌یابد. برای مثال رشتهٔ `"hello\n"` را در حافظه ذخیره می‌کنیم، درواقع `"hello\n\0"` در حافظه ذخیره می‌شود.
- برای مثال در تابع `printf` می‌توانیم از تعیین کنندهٔ فرمت `%s` برای چاپ رشته استفاده کنیم. آرایهٔ حروف مورد نظر توسط تابع `printf` دریافت می‌شود و چاپ رشته تا جایی ادامه می‌یابد که به کاراکتر تهی برخورد کنیم.
- همچنین در تابع `copy` کپی کردن کاراکترها تا جایی ادامه می‌یابد که به کاراکتر تهی برخورد کنیم.
- در این مثال برخی از حالات را مدیریت نکردیم. برای مثال وقتی طول یک خط بیشتر از حد مجاز باشد، برنامه هیچ پیامی صادر نمی‌کند. در یک برنامهٔ کامل معمولاً همه شرایط استثنا و حالات خاص باید مدیریت شوند و خطاها بررسی شوند.
- در این مثال تابع `main` می‌تواند آخرین حرف یک خط را بررسی کند و اگر طول رشته برابر با اندازهٔ بیشینه بود و حرف آخر کاراکتر تهی نبود، نتیجه گرفته می‌شود که طول خط وارد شده از حد مجاز بیشتر بوده است.

حوزهٔ تعریف

- در برنامهٔ متغیرهای `line` و `longest` را در تابع `main` تعریف کردیم. این متغیرها فقط در تابع `main` تعریف شده‌اند و بیرون از تابع دسترسی به آنها امکان پذیر نیست.
- متغیرهایی که در یک تابع تعریف می‌شوند را متغیرهای محلی [1] تابع می‌نامیم. به عبارت دیگر این متغیرها تنها در حوزهٔ تابع [2] تعریف شده‌اند.
- یک متغیر محلی با فراخوانی تابع تعریف می‌شود و در حافظه قرار می‌گیرد و به محض اتمام اجرای تابع از حافظه حذف می‌شود و مقدار خود را از دست می‌دهد و دسترسی به آن امکان‌پذیر نیست. بنابراین در ابتدای تابع نیاز به مقداردهی اولیه این متغیرهای محلی داریم.

[1] local variables
[2] scope

حوزهٔ تعریف

- یک متغیر را می‌توانیم بیرون از توابع نیز تعریف کنیم. چنین متغیرهایی عمومی [1] می‌نامیم. به متغیرهای عمومی متغیرهای خارجی [2] نیز گفته می‌شود.
- یک متغیر عمومی با شروع برنامه تعریف می‌شود و با اتمام برنامه از بین می‌رود بنابراین مقدار خود را در طول برنامه نگه‌می‌دارد.
- وقتی یک متغیر خارج از یک تابع تعریف شده باشد، می‌توانیم با استفاده از کلمهٔ کلیدی `extern` آن را اعلام کنیم.

[1] global variable
[2] external variable

حوزهٔ تعریف

- در مثال از متغیرهای عمومی به جای متغیرهای محلی استفاده شده است.

```
۱  #include <stdio.h>
۲  #define MAXLINE 1000     /* maximum input line size */
۳  int max;
۴  char line[MAXLINE];
۵  char longest[MAXLINE];    /* maximum length seen so far */
۶  /* current input line */
۷  /* longest line saved here */
۸  int _getline (void);
۹  void _copy (void);
۱۰ /* print longest input line; specialized version */
۱۱ int main ()
۱۲ {
۱۳   int len;
۱۴   extern int max;
```

```
۱۶    extern char longest[];
۱۷    max = 0;
۱۸    while ((len = _getline ()) > 0)
۱۹      if (len > max)
۲۰        {
۲۱      max = len;
۲۲      _copy ();
۲۳        }
۲۴    if (max > 0)        /* there was a line */
۲۵      printf ("%s", longest);
۲۶    return 0;
۲۷  }
```

```
۲۸ /* getline: specialized version */
۲۹ int
۳۰ _getline (void)
۳۱ {
۳۲   int c, i;
۳۳   extern char line[];
۳۴   for (i = 0; i < MAXLINE - 1
۳۵                        && (c = getchar ()) != EOF && c != '\n'; ++i)
۳۶     line[i] = c;
۳۷   if (c == '\n')
۳۸     {
۳۹       line[i] = c;
۴۰       ++i;
۴۱     }
۴۲   line[i] = '\0';
۴۳   return i;
۴۴ }
```

```
۴۴ /* copy: specialized version */
۴۵ void _copy (void)
۴۶ {
۴۷   int i;
۴۸   extern char line[], longest[];
۴۹   i = 0;
۵۰   while ((longest[i] = line[i]) != '\0')
۵۱   ++i;
۵۲ }
```

حوزهٔ تعریف

- اگر تعریف متغیر قبل از تابع استفاده از آن صورت گرفته باشد، نیازی اعلام متغیر با کلمه `extern` نداریم. همچنین اگر یک برنامه از چند فایل تشکیل شده باشد و یک متغیر در یک فایل تعریف شده باشد و بخواهیم در یک فایل دیگر از آن استفاده کنیم، نیاز داریم با استفاده از کلمهٔ `extern` آن را اعلام کنیم.
- توابعی مانند `printf` که از آنها استفاده کردیم، فایل‌های دیگر تعریف شده‌اند در ابتدای برنامه فایل `stdio.h` را ضمیمه کردیم. توابع ورودی و خروجی توسط توسعه دهندگان زبان سی در این فایل تعریف شده‌اند.
- نوع `void` برای یک متغیر به معنی نوع تهی است. تابعی که هیچ متغیری بازنمی‌گرداند، مقدار بازگشتی آن از نوع `void` است.

حوزهٔ تعریف

- وقتی یک متغیر را تعریف می‌کنیم، در واقع متغیر باید ساخته شود و بر روی حافظه قرار بگیرد. وقتی یک متغیر را اعلام می‌کنیم در واقع به کامپایلر می‌گوییم آن متغیر قبلا تعریف شده و اکنون می‌خواهیم از آن استفاده کنیم.

- تا آنجایی که امکان دارد بهتر است از متغیرهای عمومی و خارجی استفاده نکنیم. دلیل اول این است که استفاده زیاد از متغیرهای خارجی از خوانایی برنامه می‌کاهد. دلیل دوم این است که گاهی ممکن است توابع مختلف مقدار یک متغیر عمومی را به نحوی تغییر دهند که برنامه نویس نسبت به آن آگاه نباشد و این امر باعث ایجاد برنامه‌ای شود که از نظر منطقی دچار مشکل شود. سومین دلیل این است که ممکن است چند تابع در یک اجرای همزمان (در برنامه نویسی همروند) به طور همزمان یک متغیر را تغییر دهند که باعث ایجاد اشکال در اجرای برنامه شود. چهارمبن دلیل این است که توابعی که پارامتر دریافت می‌کنند به طور عمومی تعریف می‌شوند و در همهٔ برنامه‌ها قابل استفاده هستند چون به صورت یک تابع مستقل عمل می‌کنند. بنابراین برنامه قبلی را ترجیح می‌دهیم با متغیرهای محلی پیاده‌سازی کنیم و نه متغیرهای عمومی.

عبارات و نوع‌های داده‌ای

نوع‌های داده‌ای

- متغیرها [1] و ثابت‌ها [2] داده‌هایی هستند که در یک برنامه تغییر داده می‌شوند
- در تعریف یک متغیر نوع متغیر تعیین می‌شود و می‌توان یک مقدار اولیه نیز برای آن تعیین کرد. از عملگرها [3] برای انجام عملیات محاسباتی و منطقی برروی متغیرها و ثابت‌ها که عملوندها [4] در یک عبارت هستند استفاده می‌شود.
- یک عبارت [5] با ترکیب متغیرها و ثابت‌ها به عنوان عملوند و همچنین عملگرها مقادیر مورد نیاز محاسبه می‌کند.

[1] variables
[2] constants
[3] operator
[4] operand
[5] expression

نوع‌های داده‌ای

- نوع یک متغیر با ثابت مشخص می‌کند آن عملوند چه مقادیری می‌تواند داشته باشد و چگونه عملگر بر روی آن عملیات انجام می‌دهد.
- در این قسمت به معرفی نوع‌های داده‌ای می‌پردازیم.

نوع‌های داده‌ای

- نام متغیرها می‌تواند از حروف الفبا و ارقام و زیرخط [1] (_) تشکیل شده باشد ولی اولین کاراکتر نمی‌تواند یک رقم باشد.
- حروف بزرگ و کوچک الفبا با یکدیگر متفاوت‌اند، بنابراین x و X دو متغیر متفاوت هستند. برنامه نویسان سی معمولا از حروف کوچک برای نام متغیرها استفاده می‌کنند و نام نمادهای ثابت معمولا با حروف بزرگ نوشته می‌شود.
- از کلمات کلیدی مانند `if` ، `else` ، `int` نمی‌توان به عنوان نام متغیرها استفاده کرد.
- بهتر است برای متغیرها از اسامی معنی‌دار استفاده شود تا برنامه خوانایی بیشتری داشته باشد.

[1] underscope

نوع‌های داده‌ای

- نوع داده‌های اصلی در زبان سی عبارتند از :
- نوع کاراکتر [1] `char` که یک بایت است.
- نوع عدد صحیح [2] `int` که اندازهٔ آن به ماشین بستگی دارد ولی معمولاً ۴ بایت است.
- نوع عدد اعشاری [3] `float` یا ممیز شناور که ۴ بایت است.
- نوع اعشاری با دقت دو برابر [4] `double` که ۸ بایت است.

[1] character
[2] integer
[3] floation point
[4] double-precision float point

نوع‌های داده‌ای

- همچنین می‌توان از کلمات کلیدی توصیفی short و long قبل از تعریف متغیرها استفاده کرد. برای مثال متغیر از نوع int مقدار ۴ بایت را اشغال می‌کند و متغیر از نوع short int مقدار ۲ همچنین متغیر از نوع long int مقدار ۸ بایت را اشغال می‌کند.
- از کلمات توصیفی signed و unsigned نیز می‌توان برای تعیین علامت‌دار بودن یا بدون علامت بودن متغیرها استفاده کرد.
- وقتی یک متغیر علامت‌دار است یک بیت آن برای علامت آن در نظر گرفته می‌شود. برای مثال unsigned short int دو بایت است و مقادیر ۰ تا ۶۵۵۳۵ را خود نگه‌می‌دارد، اما signed short int مقادیر -۳۲۷۶۸ تا ۳۲۷۶۸ را در خود نگه‌می‌دارد.

نوع‌های داده‌ای

- یک عدد صحیح را توسط ارقام نشان می‌دهیم. عدد 1234 یک عدد صحیح ثابت است. برای عددهای ثابت صحیح بزرگ از کاراکتر L در پایان عدد استفاده می‌کنیم. برای مثال عدد 123456789L یک عدد ثابت صحیح بزرگ است.
- اعداد اعشاری را با ممیز اعشاری (برای مثال به صورت 4.123) یا به صورت نمایش علمی (برای مثال به صورت 1e-2) نشان می‌دهیم.
- برای نمایش اعداد در مبنای ۱۶ از پیشوند 0X استفاده می‌کنیم. برای مثال 0X1F یک عدد صحیح در مبنای ۱۶ است.
- کاراکترها را با یک علامت نقل قول نشان می‌دهیم. برای مثال 'x' یک کاراکتر شامل حرف x است که مقداری است که در یک بایت ذخیره می‌شود.

نوع‌های داده‌ای

- برخی کاراکترهای خاص مانند کاراکتر خاص مانند کاراکتر خط جدید '\n' و کاراکتر ستون جدید '\t' نیز وجود دارند. یک کاراکتر را می‌توان با کد اسکی آن نیز به صورت '\xhh' نمایش داد به طوری‌که hh تعدادی ارقام در مبنای شانزده هستند.
- کاراکتر '0' کاراکتر تهی است.
- یک نماد ثابت را با `define #` می‌توان تعریف کرد. نماد ثابت در زمان کامپایل جایگزین مقدار ثابت می‌شود. برای مثال می‌نویسیم:

```
۱ #define MAXLINE 1000
۲ char line[MAXLIN + 1];
```

نوع‌های داده‌ای

- یک ثابت رشته [1] دنباله‌ای است از کاراکترها که با علامت نقل قول دوگانه تعریف می‌شود. برای مثال `"I am a string"` یک رشته است.
- برای استفاده از علامت نقل قول در یک رشته از `\"` استفاده می‌کنیم. همچنین دو رشته با درکنار یکدیگر قرار گرفتن، به یکدیگر الحاق می‌شوند. برای مثال `"hello" "world"` ، `"hello , world"` برابراست با `"hello , world"` . برای تقسیم یک رشته طولانی در چند خط می‌توانیم از این تکنیک استفاده کنیم.
- یک رشته به صورت آرایه‌ای از کاراکترها تعریف می‌شود و آخرین کاراکتر یک رشته به طور قراردادی برابراست با `'\0'`.

[1] string constant

نوع‌های داده‌ای

- برای مثال، جهت به دست آوردن طول یک رشته می‌توانیم از تابع زیر استفاده کنیم.

```
۱  /* strlen: return length of s */
۲  int
۳  strlen (char s[])
۴  {
۵    int i;
۶    while (s[i] != '\0')
۷          ++i;
۸    return i;
۹  }
```

نوع‌های داده‌ای

- تابع `strlen` در کتابخانه `<string.h>` تعریف شده است.
- یک ثابت شمارشی [1] عدد صحیحی است که عضو یک نوع داده شمارشی [2] است. نوع دادهٔ شمارشی برای نگهداری مجموعه‌ای از اعداد ثابت تحت عنوان یک نام به کار می‌رود.
- برای مثال مقدار نوع داده‌ای `boolean` که در زیر تعریف شده است می‌تواند `No` باشد که معادل عدد صحیح صفر است و یا `Yes` باشد که معادل عدد صحیح یک است.

```
۱ enum boolean {No, Yes};
```

- برای ثابت‌های شمارشی می‌توان مقدار نیز تعیین کرد. در مثال زیر مقدار ثابت شمارشی `JAN` برابراست با یک.

[1] enumeration conctant
[2] enumeration type

نوع‌های داده‌ای

```
۱ enum escapes
۲ { BELL = '\a', BACKSPACE = '\b', TAB = '\t',
۳   NEWLINE = '\n', VTAB = '\v', RETURN = '\r'
۴ };
۵ enum months
۶ { JAN = 1, FEB, MAR, APR, MAY, JUN,
۷   JUL, AUG, SEP, OCT, NOV, DEC
۸ };
۹ /* FEB = 2, MAR = 3, etc. */
```

نوع‌های داده‌ای

- یک متغیر توسط نوع داده‌ای آن تعریف می‌شود. در تعریف متغیر ابتدا نوع و سپس نام متغیر و سپس به طور اختیاری مقدار اولیه آن مشخص می‌شود. برای مثال :

```
int lower, upper;
int i = 0
```

- با استفاده از کلمه کلیدی const می‌توان یک ثابت تعریف کرد. مقدار یک ثابت در طول اجرای برنامه غیر قابل تغییر است. برای مثال :

```
const double e = 2.7182;
```

- یک ثابت مانند یک متغیر است پس می‌توان آن را به یک تابع ارسال کرد.

- همچنین می‌توانیم پارامترهای یک تابع را ثابت تعریف کنیم تا مانع تغییر مقدار آرگومان‌ها (در صورتی که آرایه باشند یا فراخوانی با ارجاع باشد) شوند. برای مثال :

```
int strlen (const char[]);
```

عملگرها

- در یک عبارت می‌توان از عملگرهای حسابی [1] مانند + ، - ، * ، / استفاده کرد و برای به دست آوردن باقیمانده از عملگر % استفاده می‌شود.
- برای مثال برای مشخص کردن کبیسه بودن سال می‌توانیم از برنامه زیر استفاده کنیم.

```
1 if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
2   printf ("%d is a leap year\n", year);
3 else
4   printf ("%d is not a leap year\n", year);
```

- عملگرهای یگانی + و - بالاترین اولویت را دارند و پس از آنها * ، / ، % هم اولویت بوده و در درجهٔ دوم اولویت قرار دارند و در نهایت + و - اولویت سوم قرار می‌گیرند.

[1] arithmetic operators

عملگرها

- عملگرهای رابطه‌ای [1] < ، =< ، > ، => دارای بالاترین اولویت هستند و پس از آنها عملگرهای == و =! در درجه دوم اولویت قرار می‌گیرند.
- عملگرهای رابطه نسبت به عملگرهای حسابی اولویت کمتری دارند.
- عملگرهای منطقی [2] عطف && و فصل || و نقیض ! نیز برای ترکیب عبارات منطقی به کار می‌روند.
- اولویت عطف بیشتر از فصل است و هر دو اولویت کمتری نسبت به عملگرهای حسابی و رابطه‌ای دارند.

[1] relational operators
[2] logical operator

عملگرها

- در عبارت `(c = getchar()) != '\n'` به پرانتزگذاری نیاز داریم، زیرا اولویت `!=` بالاتر از عملگر `=` است.
- هر عبارت منطقی یا رابطه‌ای دارای مقدار یک است اگر درست باشد و مقدار آن برابر صفر است اگر نادرست باشد.
- عملگر نقیض `!` مقدار صفر را به یک و مقدار یک را به صفر تبدیل می‌کند.
- بنابراین به جای `if(valid == 0)` می‌نویسیم `if(!valid)`.

تبدیل نوع

- وقتی یک عملگر دارای چند عملوند از نوع‌های مختلف است، نوع عملوند با اندازهٔ کوچک‌تر به نوع عملوند با اندازهٔ بزرگ‌تر تبدیل می‌شود. بدین ترتیب هیچ اطلاعاتی را بین نمی‌رود. برای مثال در عبارت `i + f` که جمع یک عدد اعشاری و یک عدد صحیح است، عدد صحیح به اعشاری تبدیل می‌شود.
- در جایی که تبدیل نوع ممکن است به از دست رفتن اطلاعات منجر شود. کامپایلر پیام خطا صادر می‌کند. برای مثال در انتساب یک عدد صحیح طولانی به یک عدد صحیح ممکن است اطلاعات از بین برود.
- طول `char` از `int` کمتر است، پس می‌تواند بدون خطا تبدیل کاراکتر به عدد صحیح انجام شود.

تبدیل نوع

- در مثال زیر یک رشته به معادل عددی آن تبدیل می‌شود.

```
۱  /* atoi: convert s to integer */
۲  int
۳  atoi (char s[])
۴  {
۵    int i, n;
۶    n = 0;
۷    for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)
۸      n = 10 * n + (s[i] - '0');
۹    return n;
۱۰ }
```

تبدیل نوع

- در مثال قبل `s[i]-'0'` مقدار عددی کاراکتر را مشخص می‌کند. معادل اسکی کاراکتر `'0'` برابراست با ۴۸ و معادل اسکی کاراکترهای `'1'` ، `'2'` و ... به ترتیب برابراست با ۴۹ ، ۵۰ و ... . بنابراین تفاضل محاسبه شده معادل عددی یک کاراکتر را تعیین می‌کند.
- به عنوان یک مثال دیگر، در برنامه زیر یک حروف بزرگ به حروف کوچک تبدیل می‌شود.

```
۱ /* lower: convert c to lower case; ASCII only */
۲ int
۳ lower (int c)
۴ {
۵   if (c >= 'A' && c <= 'Z')
۶     return c + 'a' - 'A';
۷   else
۸     return c;
۹ }
```

تبدیل نوع

- در کتابخانه <ctype.h> بسیاری از توابع کاربردی برای تبدیل کاراکترها مانند توابع `tolower` یا `isdigit` پیاده‌سازی شده‌اند.

تبدیل نوع

- یک عبارت منطقی در صورتی که درست باشد مقدار یک و در غیراینصورت مقدار صفر را بازمی‌گرداند. بنابراین مقدار متغیر در عبارت `d = c >= '0' && c <= '9'` برابر با یک است اگر `c` یک رقم باشد و در غیراینصورت مقدار آن برابر با صفر است.
- در هنگام استفاده از اعداد در عبارت منطقی هر مقدار غیر صفر درست [1] است و مقدار صفر برابر با نادرست [2] است.

[1] true
[2] false

تبدیل نوع

- در یک عبارت اگر یکی از عملوندها `long double` باشد بقیه عملوندها نیز به این نوع تبدیل می‌شوند در غیراینصورت اگر یکی از عملوندها `double` باشد بقیه عملوندها به `double` تبدیل می‌شوند. در غیراینصورت اگر یکی از عملوندها `float` باشد، بقیه عملوندها به `float` تبدیل می‌شوند. در غیراینصورت اگر یکی از عملوندها `int` باشد، بقیه عملوندها به `int` تبدیل می‌شوند.
- اگر در یک عملیات انتساب یک متغیر از نوع `char` را برابر با یک متغیر از نوع `int` قرار دهیم، بیت‌های پرارزش‌تر از بین می‌روند.
- در تبدیل `float` به `int` قسمت اعشاری از بین می‌رود. در تبدیل `double` به `float` رقم‌های اعشار با تقریب کاهش می‌یابند.
- تبدیل نوع در ارسال آرگومان‌ها به پارامترها نیز صورت می‌گیرد اگر نوع‌ها همخوانی نداشته باشند.

تبدیل نوع

- یک نوع را می‌توان به صورت صریح نیز به یک نوع دیگر تبدیل کرد. در عبارت `(type name)expression` نوع عبارت به طور صریح تبدیل می‌شود. به این پرانتزگذاری و تعیین نوع عملگر تبدیل نوع [1] می‌گوییم.
- برای مثال برای تبدیل عدد `n` به `double` در تابع `sqrt` که جذر یک عدد را محاسبه می‌کند می‌نویسیم `sqrt((double)n)`.

[1] cast operator

عملگرهای افزایش و کاهش

- در زبان سی دو عملگر وجود دارد که در زبان ریاضی وجود ندارند. عملگر افزایش ++ که یک واحد به مقدار یک متغیر می‌افزاید و عملگر کاهش -- که یک واحد از مقدار یک تغییر می‌کاهد.
- این دو عملگر به دو صورت می‌توانند استفاده شوند : به صورت پیشوند [1] (یعنی قبل از متغیر n++) و به صورت پسوند [2] (یعنی بعد از متغیر ++n).
- در هر دو مشکل عملگر افزایش یک واحد به مقدار متغیر می‌افزاید با این تفاوت که در حالت پیشوند افزایش قبل از استفاده از متغیر اعمال می‌شود و در حالت پسوند افزایش بعد از استفاده متغیر اعمال می‌شود.

[1] prefix
[2] postfix

عملگرهای افزایش و کاهش

- برای مثال اگر مقدار `n` برابر با ۵ باشد، آنگاه پس از عبارت `x = n++` مقدار `x` برابر با ۵ خواهد بود چرا چراکه ابتدا `n` با مقدار قبلی در عبارت استفاده می‌شود و سپس یک واحد به آن افزوده می‌شود. اما پس از عبارت `x = ++n` مقدار `x` برابر با ۶ خواهد بود چراکه ابتدا یک واحد به `n` افزوده می‌شود و سپس `n` با مقدار جدید در عبارت استفاده می‌شود.
- این عملگرها تنها برروی تک متغیر اعمال می‌شوند، بنابراین `++(i+j)` یک عبارت غیر معتبر و بی معنا است.
- در برخی موارد هدف تنها افزایش یا کاهش یک متغیر است و در این موارد پیشوند و پسوند تفاوتی ندارند. اما وقتی این عملگرها در یک عبارت استفاده شوند، تفاوت آنها مشهود است. همچنین با استفاده از این عملگرها می‌توان برنامه را مختصرتر نوشت.

عملگرهای افزایش و کاهش

- در مثال زیر که همهٔ حروف c را از رشته s حذف می‌کند از عملگر افزایش استفاده شده است.

```
۱  /* squeeze: delete all c from s */
۲  void
۳  squeeze (char s[], int c)
۴  {
۵    int i, j;
۶    for (i = j = 0; s[i] != '\0'; i++)
۷      if (s[i] != c)
۸        s[j++] = s[i];
۹    s[j] = '\0';
۱۰ }
```

عملگرهای افزایش و کاهش

- اگر عملگر ++ وجود نداشت، مجبور بودیم بدنهٔ شرط را به صورت زیر بنویسیم.

```
۱ if (s[i] != c ) {
۲      s[j] = s[i];
۳      j = j + 1;
۴ }
```

عملگرهای افزایش و کاهش

- حال تابع strcat را در نظر بگیرید که رشته t را در انتها رشته s الحاق می‌کند.

```
۱  /* strcat: concatenate t to end of s;
۲   * s must be big enough */
۳  void
۴  strcat (char s[], char t[])
۵  {
۶    int i, j;
۷    i = j = 0;
۸    while (s[i] != '\0')^^I^^I/* find end of s */
۹      i++;
۱۰   while ((s[i++] = t[j++]) != '\0')^^I/* copy t */
۱۱     ;
۱۲ }
```

عملگرهای افزایش و کاهش

- پس از هر بار کپی کردن یک کاراکتر از رشته `t` به رشته `s` ، مقدار اندیس `i` و `j` یک واحد افزایش پیدا می‌کند.

عملگرهای بیتی

- در زبان سی تعدادی عملگر بیتی وجود دارد که بر روی عملوندهای صحیح مانند `char` ، `short` ، `int` و `long` اعمال می‌شوند. عملگرهای بیتی دوگانی [1] بر روی بیت‌های دو عملوند دریافتی خود عملیات بیتی شامل عطف [2] و فصل [3] و فصل انحصاری [4] و انتقال به چپ [5] و انتقال به راست [6] انجام می‌دهند.

[1] binary bitwise operators
[2] conjunction
[3] disjunction
[4] exclusive disjunction
[5] left shift
[6] right shift

عملگرهای بیتی

- عملگر بیتی عطف AND با & ، عملگر بیتی فصل OR با | ، عملگر بیتی فصل انحصاری XOR با $\wedge$ ، عملگر انتقال به چپ با $\ll$ و عملگر انتقال به راست با $\gg$ نشان داده می‌شوند.
- همچنین عملگر یگانی [1] مکمل با ~ نشان داده می‌شود.

[1] unary bitwise operator

عملگرهای بیتی

- برای مثال در عبارت `n = n & 0177` ، عدد 0177 عددی در مبنای ۸ معادل عدد 0000000001111111 است. بنابراین همهٔ بیت‌های عدد n بعد از ۷ بیت سمت راست برابر با صفر قرار می‌گیرند.
- در عبارت `x = x | SET_ON` همهٔ بیت‌هایی که در x صفر هستند و در SET_ON یک هستند را به یک تبدیل می‌کند.
- توجه داشته باشد که عملگر منطقی && با عملگر بیتی & متفاوت است. اگر x برابر با ۱ و y برابر با ۲ باشد، مقدار x&y برابر با صفر است، در حالی که مقدار x&&y برابر با مقدار درست یا یک است.

عملگرهای بیتی

- عملگر انتقال ≫ و ≪ انتقال به چپ و راست انجام می‌دهند. در سمت چپ عملگر متغیری قرار می‌گیرد که عملیات انتقال برای آن انجام می‌شود و در سمت راست عملگر تعداد انتقال‌ها قرار می‌گیرد.
- برای مثال `x≪2` مقدار `x` را دو واحد به سمت چپ انتقال می‌دهد و مقدار دو بیت کم ارزش را برابر با صفر قرار می‌دهد. این عملیات معادل ضرب عدد در ۴ است.
- عملگر یگانی بیتی مکمل ~ بیت‌های صفر را به یک و بیت‌های یک را به صفر تبدیل می‌کند.

عملگرهای انتساب

- عبارت `i = i + 2` را می‌توانیم به صورت `i += 2` نیز بنویسیم. عملگر `+=` یک عملگر انتساب [1] نامیده می‌شود.
- همهٔ عملگرهای `op=` در زبان سی تعریف شده‌اند به طوری‌که مقدار `op` می‌تواند `+` ، `-` ، `*` ، `/` ، `%` ، $\ll$ ، $\gg$ ، `&` ، `|` ، $\wedge$ باشد
- برای مثال `x *= y+1` معادل است با `x = x * (y+1)`.

[1] assignment operator

عملگرهای انتساب

- در برنامهٔ زیر تعداد بیت‌های یک در متغیر x شمرده می‌شود.

```
۱  /* bitcount: count 1 bits in x */
۲  int
۳  bitcount (unsigned x)
۴  {
۵    int b;
۶    for (b = 0; x != 0; x >>= 1)
۷      if (x & 01)
۸        b++;
۹    return b;
۱۰ }
```

عملگرهای انتساب

- علاوه بر این که با استفاده از عملگرهای انتسال کد را می‌توان به صورت مختصر نوشت، در برخی موارد باعث خوانایی بیشتر برنامه نیز می‌شود. برای مثال عبارت
`yyval [yypv[p3+p4] + yypv[p1]] += 2`
را در نظر بگیرید. علاوه بر این که با استفاده از عملگر += برنامه کوتاهتر می‌شود، اگر عبارت را به صورت عادی عملگر تساوی می‌نوشتیم، خوانندهٔ برنامه مجبور بود بررسی کند آیا در سمت چپ و راست عملگر تساوی دو عبارت یکسان‌اند یا خیر و اگر دو عبارت یکسان نبودند نمی‌توانست مطمئن باشد که آیا برنامه‌نویس خطایی در نوشتن برنامه انجام داده یا اینکه در عبارت یکسان نیستند.
- یک عبارت انتساب دارای یک مقدار است. مقدار یک عبارت، برابر است با مقدار سمت چپ عبارت بعد از عملیات و نوع آن برابر با نوع متغیر سمت چپ عبارت.

عبارات شرطی

- قطعه برنامه زیر را در نظر بگیرید.

```
if ( a >b )
    z = a;
else
    z = b;
```

- این قطعه برنامه ماکزیمم بین دو متغیر را محاسبه می‌کند. این عبارت را به طور مختصرتر می‌توان با استفاده از عملگر :? به صورت زیر نوشت.

```
z = (a > b) ? a : b;      /* z = max(a,b)  */
```

عبارات شرطی

- در واقع مقدار عبارت `cond ? expr1 : expr2` برابراست با `expr1` اگر `cond` صحیح باشد و برابر است با `expr2` اگر `cond` نادرست باشد.
- اگر نوع `expr1` و `expr2` متفاوت باشد، نوع عبارت شرطی برابر با نوعی است که وسیع‌تر باشد. برای مثال نوع عبارت `(n > 0) ? f : i` برابراست با `float` جایی که `f` یک `float` است و `i` یک `int`.

عبارات شرطی

- در برنامهٔ زیر کاراکتر خط جدید هر ۱۰ خط یک بار چاپ می‌شود و همچنین وقتی آخرین عنصر آرایه چاپ می‌شود.

```
for (i = 0; i < n; i++)
    printf("%6d%c", a[i], (i%10==9 || i==n-1) ? '\n' : ' ');
```

- در مثال زیر اگر یک عنصر وجود داشت می‌خواهیم از کلمهٔ مفرد استفاده کنیم و اگر چند عنصر وجود داشت از کلمهٔ جمع.

```
printf("You have %d item%s.\n", n, n==1 ? "" : "s");
```

اولویت عملگرها

- در جدول زیر قوانین اولویت ذکر شده‌اند. عملگرهایی که در سطرهای بالاتر قرار دارند، اولویت بالاتری دارند. عملگرهایی که در یک سطر هستند اولویت یکسان دارند.

Operators	Associativity
`() [] -> .`	left to right
`! ~ ++ -- + - *(type) sizeof`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`= += -= *= /= %= &= ^= |= <<= >>=`	right to left
`,`	left to right

- برای مثال در عبارت `if((x & MASK) == 0 )` اگر از پرانتزگذاری استفاده نکنیم، معنای عبارت متفاوت خواهد بود.

اولویت عملگرها

- همچنین باید توجه داشت که استاندارد زبان سی مشخص نمی‌کند که در یک عبارت کدام یک از پارامترها زودتر محاسبه می‌شوند، بنابراین در پیاده‌سازی‌های متفاوت زبان سی و کامپایلرهای متفاوت نتیجه یک عبارت متفاوت باشد.
- برای مثال در عبارت زیر مشخص نیست آیا ابتدا n++ محاسبه می‌شود و یا مقدار power(2,n) و بنابراین بهتر است برنامه‌نویس سی از نوشتن چنین عبارت‌هایی پرهیز کند.

```
1 printf("\%d \%d \bksl n" , ++n, power(2,n));    /* WRONG */
```

ساختارهای کنترلی

ساختارهای کنترلی

- ساختارهای کنترلی [1] در زبان‌های برنامه‌نویسی جهت اعمال تصمیمات بر روی ترتیب اجرای دستورات به کار می‌روند.
- یک دستور [2] در یک زبان برنامه‌نویسی با علامت نقطه ویرگول (;) پایان می‌یابد. یک دستور می‌تواند یک عملیات انتساب و یا فراخوانی یک تابع باشد. علاوه بر این دستورات، شامل تعاریف [3] و اعلام‌ها [4] می‌شوند.
- علامت آکولاد باز و بسته {} برای گروه بندی دستورات استفاده می‌شوند. یک گروه از دستورات را یک بلوک [5] می‌نامیم. برای مثال دستورات تعریف یک تابع در یک بلوک قرار می‌گیرند.
- دستورات متعلق به یک ساختار کنترلی نیز که در این قسمت توضیح خواهیم داد، در یک بلوک قرار می‌گیرند.

---

[1] control flow structures
[2] statement
[3] definition
[4] declaration
[5] block

دستورات شرطی

- دستورات شرطی اگر–وگرنه `if-else` برای بیان یک تصمیم در انتخاب دستورات به کار می‌رود.
- ساختار نحوی دستور شرطی `if-else` به صورت زیر است.

```
۱ if (condition)
۲       statement1
۳ else
۴       statement2
```

- قسمت `else` اختیاری است. شرط `condition` سنجیده می‌شود و در صورتی که مقدار آن درست بود `statement1` اجرا می‌شود و در صورتی که مقدار آن نادرست بود `statement2` اجرا می‌شود.

دستورات شرطی

- مقدار درست [1] در زبان سی برابر با مقدار غیر صفر است و مقدار نادرست [2] برابر با مقدار صفر.
- از آنجایی که در شرط `if` یک مقدار سنجیده می‌شود، به جای `if(expression != 0)` می‌توانیم بنویسیم `if(expression)`

[1] true
[2] false

دستورات شرطی

- عبارت else همیشه متعلق به نزدیک‌ترین عبارت if است.
- برای مثال در قطعه برنامه زیر else متعلق به if دوم است.

```
if (n > 0)
    if (a > b)
        z = a;
    else
        z = b;
```

- اگر بخواهیم در برنامه فوق else متعلق به if اول باشد، باید قطعه برنامه را به صورت زیر بنویسیم.

```
if (n > 0){
    if (a > b)
        z = a;
    }
    else
        z = b;
```

دستورات شرطی

- برنامهٔ زیر را در نظر بگیرید. گرچه برنامه‌نویس با دندانه‌گذاری [1] کد مقصود خود را بیان داشته و هدفش این بوده که `else` متعلق به `if` اول باشد، ولی کامپایلر برنامه را به نحوی دیگر اجرا می‌کند، زیرا `else` متعلق به نزدیک‌ترین `if` یعنی `if` دوم است.

```
۱ if (n > 0)
۲   for (i = 0; i < n; i++)
۳     if (s[i] > 0) {
۴       printf("...");
۵       return i;
۶     }
۷ else  /* WRONG */
۸   printf("error -- n is negative\n");
```

[1] indentation

دستورات شرطی

- در دستور شرطی `if-else` تنها دو حالت سنجیده می‌شود یعنی یا `condition` درست است و یا نادرست.
- اگر بخواهیم چندین شرط با بسنجیم می‌توانیم از دستورات `if else` استفاده کنیم.
- دستورات `if else` را می‌توانیم به صورت زیر بنویسیم.

```
if (condition1)
     statement1
else if (condition2)
     statement2
...
else
     statementN
```

دستورات شرطی

- این شرط‌ها به ترتیب سنجیده می‌شوند. اگر `condition1` درست باشد `statement1` اجرا می‌شود. در غیراینصورت اگر `condition2` درست باشد `statement2` اجرا می‌شود، در غیراینصورت `statementN` اجرا می‌شود.
- قسمت `else` آخر وقتی اجرا می‌شود که هیچ کدام از شرط‌ها درست نباشند. این قسمت را می‌توان حذف کرد.

دستورات شرطی

- فرض کنید v یک آرایهٔ مرتب شده از اعداد صحیح است. در برنامه زیر می‌خواهیم یک عنصر را در آرایه جستجو کنیم. فرض کنید مقدار این عنصر در x ذخیره می‌شود.
- برای این جستجو از الگوریتم جستجوی دودویی [1] استفاده می‌کنیم.
- در جستجوی دودویی ابتدا مقدار x را با عنصر وسط آرایه v مقایسه می‌کنیم. اگر مقدار x کمتر از مقدار عنصر وسط آرایه بود، جستجو را بر روی نیمه اول آرایه ادامه می‌دهیم. اگر مقدار x بیشتر از مقدار عنصر وسط آرایه بود، جستجو را بر روی نیمه دوم آرایه ادامه می‌دهیم. و اما اگر مقدار x برابر با عنصر وسط آرایه بود، مکان عنصر وسط را باز می‌گردانیم.

[1] binary search

دستورات شرطی

- الگوریتم جستجوی دودویی به صورت زیر است.

```
۱  /* binsearch: find x in v[0] <= v[1] <= ... <= v[n-1] */
۲  int
۳  binsearch (int x, int v[], int n)
۴  {
۵    int low, high, mid;
۶    low = 0;
۷    high = n - 1;
```

دستورات شرطی

```
 8    while (low <= high)
 9          {
10                mid = (low + high) / 2;
11                if (x < v[mid])
12                      high = mid + 1;
13                else if (x > v[mid])
14                      low = mid + 1;
15                else         /* found match */
16                      return mid;
17          }
18    return -1;          /* no match */
19  }
```

دستورات شرطی

- ساختار کنترلی switch یک ساختار شرطی چندگزینه‌ای است. در این ساختار شرطی چند شرط وجود دارد که در آن بررسی می‌شود کدام یک از شرط‌ها برقرار هستند. هر کدام از شرط‌ها که برقرار بود، دستورات متعلق به آن شرط اجرا می‌شوند.
- ساختار کنترلی switch به صورت زیر است.

```
switch (expression) {
    case const-expr1 : statement1
    case const-expr2 : statement2
    ...
    defult : statement N
```

- اگر مقدار عبارت expression برابر با const-expr1 بود، statement1 اجرا می‌شود، اگر مقدار عبارت expression برابر با const-expr2 بود، statement2 اجرا می‌شود و در غیراینصورت در حالت پیش‌فرض defult دستور statement N اجرا می‌شود.

دستورات شرطی

- فرض کنید می‌خواهیم برنامه‌ای بنویسیم که رخداد هر یک از ارقام را در یک رشته و کاراکترهای خط فاصله، خط جدید و ستون جدید را بشمارد.

دستورات شرطی

- این برنامه را می‌توانیم به صورت زیر بنویسیم.

```
۱  #include <stdio.h>
۲  int main ()^^I      /* count digits, white space, others */
۳  {
۴    int c, i, nwhite, nother, ndigit[10];
۵    nwhite = nother = 0;
۶    for (i = 0; i < 10; i++)
۷        ndigit[i] = 0;
۸    while ((c = getchar ()) != EOF)
۹      {
۱۰     switch (c)
۱۱       {
۱۲           case '0':
۱۳           case '1':
۱۴           case '2':
۱۵           case '3':
```

دستورات شرطی

```
۱۶            case '4':
۱۷            case '5':
۱۸            case '6':
۱۹            case '7':
۲۰            case '8':
۲۱            case '9':
۲۲                 ndigit[c - '0']++;
۲۳                 break;
۲۴            case ' ':
۲۵            case '\n':
۲۶            case '\t':
۲۷                 nwhite++;
۲۸                 break;
۲۹            default:
۳۰                 nother++;
```

```
                    break;
                }
            }
    printf ("digits =");
    for (i = 0; i < 10; i++)
        printf (" %d", ndigit[i]);
    printf (", white space = %d, other = %d\n", nwhite, nother);
    return 0;
}
```

دستورات شرطی

- دستور توقف `break` باعث می‌شود کنترل برنامه از `switch` خارج شود. در ساختار `switch` همهٔ شرط‌ها بررسی می‌شوند، بنابراین اگر چند شرط برقرار باشند، دستورات همهٔ شروط درست اجرا می‌شوند، مگر اینکه به طور صریح توسط دستور `break` در یکی از شاخه‌های `switch` از آن خارج شویم.
- دستور توقف `break` در ساختارهای حلقه تکرار مانند `while` و `for` و `do` نیز به کار می‌رود و باعث می‌شود کنترل برنامه از حلقه تکرار خارج شود قبل از اینکه حلقه به پایان برسد.

دستورات شرطی

- بنابراین زبان سی به گونه‌ای طراحی شده که اگر چند حالت در `switch` درست بودند همهٔ حالت‌ها بررسی شوند. مزیت این طراحی این است که به برنامه‌نویس اجازه می‌دهد برای چند حالت متفاوت یک دسته دستورات واحد اجرا کند. مشکل این طراحی این است که همیشه وقتی حالت‌ها متمایزند نیاز است بعد از هر حالت دستور `break` نوشته شود.
- توجه کنید با این‌که از نظر منطقی نیازی به قرار دادن دستور `break` بعد از دستورات شاخه `default` نیست، اما بهتر است دستور `break` قرار داده شود، زیرا ممکن است در آینده برنامه‌نویس دیگری تعدادی شاخه به انتهای `switch` اضافه کند و قرار دادن `break` پس از شاخه `default` مانع بروز خطاهای احتمالی می‌شود.

ساختارهای حلقه تکرار

- ساختارهای حلقه [1] برای تکرار بلوکی از دستورات به تعداد معین به کار می‌روند.
- چند نوع ساختار حلقه وجود دارد که عبارتند از حلقه `for` و `while` و `do`.
- ساختار `while` (مادامی‌که) به صورت زیر است.

```
while(condition)
        statement
```

- مادامی‌که مقدار شرط `condition` صحیح است، دستورات `statement` اجرا می‌شوند. به عبارت دیگر پس از اتمام اجرای دستورات، دوباره شرط بررسی می‌شود و اگر شرط همچنان درست بود دستورات برای بار دیگر اجرا می‌شوند.

[1] loop structures

ساختارهای حلقه تکرار

- ساختار `for` (برای) به صورت زیر است.

```
for (expr1 ; cond ; expr2)
        statement
```

- این ساختار معادل است با :

```
expr1;
while(cond) {
          statement
          expr2;
}
```

- هر یک از سه عبارت در ساختار `for` می‌تواند حذف شوند. اگر `expr1` حذف شوند، این عبارات می‌تواند قبل از حلقه قرار بگیرند. اگر `expr2` حذف شود، این عبارت می‌تواند در درون حلقه قرار بگیرند. اگر `cond` حذف شود، شرط حلقه همیشه درست است و خاتمه نمی‌یابد مگر با دستور `break` یا `return`.

ساختارهای حلقه تکرار

- عبارت `for(;;){ ... }` یک حلقه بینهایت است و فقط با `break` یا `return` خاتمه می‌یابد.
- معمولا وقتی از ساختار `for` استفاده می‌کنیم که نیاز به مقدار دهی اولیه باشد و نیاز به اعمال تغییرات در متغیرها برای تکرار بعدی در حلقه داشته باشیم.

ساختارهای حلقه تکرار

- در مثال زیر به اعمال تغییرات در تکرارهای حلقه و مقدار دهی اولیه نداریم، پس `while` مناسب‌تر است.

```
while ((c = getchar()) == ' ' || c == '\n' || c = '\t')
; /* skip white space characters */
```

- معمولا در هنگام انجام عملیات بر روی عناصر یک آرایه به یک حلقه به صورت `for(i = 0 ; i < n ; i++)` نیاز داریم.

- فرض کنید می‌خواهیم برنامه‌ای بنویسیم که یک رشته را به عددی که محتوای رشته است تبدیل کنیم. در ابتدا باید همه کاراکترهای خط فاصله را نادیده بگیریم. سپس یک کاراکتر را به عنوان علامت (در صورت وجود) دریافت کنیم و سپس ارقام را به ترتیب از رشته بخوانیم و تبدیل به عدد کنیم.

ساختارهای حلقه تکرار

- برنامه تبدیل رشته به عدد به صورت زیر نوشته می‌شود.

```
۱  #include <ctype.h>
۲  /* atoi: convert s to integer; version 2 */
۳  int
۴  atoi (char s[])
۵  {
۶    int i, n, sign;
۷    for (i = 0; isspace (s[i]); i++)  /* skip white space */
۸        ;
۹    sign = (s[i] == '-') ? -1 : 1;
۱۰   if (s[i] == '+' || s[i] == '-')  /* skip sign */
۱۱      i++;
۱۲   for (n = 0; isdigit (s[i]); i++)
۱۳     n = 10 * n + (s[i] - '0');
۱۴   return sign * n;
۱۵ }
```

ساختارهای حلقه تکرار

- در کتابخانه استاندارد تابع `strtol` برای تبدیل یک رشته به عدد صحیح طولانی وجود دارد.

ساختارهای حلقه تکرار

- می‌خواهیم یک آرایه‌ای از اعداد صحیح را توسط الگوریتم مرتب‌سازی شل [1] مرتب کنیم. این روش مرتب‌سازی در سال ۱۹۵۹ توسط شل ابداع شد. در این روش مرتب‌سازی ابتدا عناصر با فاصله دور مقایسه می‌شوند. پس در گام‌های ابتدایی عناصر با فاصله مرتب می‌شوند و در گام‌های انتهایی عناصر نزدیک به هم مرتب می‌شوند. الگوریتم مرتب‌سازی شل به صورت زیر است.

[1] Shell sort

ساختارهای حلقه تکرار

```
۱  /* shellsort: sort v[0]...v[n-1] into increasing order */
۲  void
۳  shellsort (int v[], int n)
۴  {
۵    int gap, i, j, temp;
۶    for (gap = n / 2; gap > 0; gap /= 2)
۷        for (i = gap; i < n; i++)
۸           for (j = i - gap; j >= 0 && v[j] > v[j + gap]; j -= gap)
۹           {
۱۰              temp = v[j];
۱۱              v[j] = v[j + gap];
۱۲              v[j + gap] = temp;
۱۳          }
۱۴ }
```

ساختارهای حلقه تکرار

- سه حلقه تودرتو در این مثال وجود دارند. در حلقهٔ بیرونی فاصلهٔ بین عناصر کنترل می‌شود، و در هر بار این فاصله نصف می‌شود. حلقه میانی بررسی عناصر آرایه را کنترل می‌کند و در حلقه درونی جفت عناصر با فاصله معین را مقایسه و در صورت نیاز آنها را جابجا می‌کند.

ساختارهای حلقه تکرار

- یکی از عملگرها در زبان سی عملگر ویرگول (`;`) است. در حلقه `for` می‌توان با استفاده از این عملگر چند عبارت مقداردهی اولیه یا چند عبارت اعمال گام قرار داد.

ساختارهای حلقه تکرار

- در مثال زیر وارون یک رشته محاسبه می‌شود.

```
۱  #include <string.h>
۲  /* reverse: reverse string s in place */
۳  void
۴  reverse (char s[])
۵  {
۶    int c, i, j;
۷    for (i = 0, j = strlen (s) - 1; i < j; i++, j--)
۸      {
۹        c = s[i];
۱۰       s[i] = s[j];
۱۱       s[j] = c;
۱۲     }
۱۳ }
```

ساختارهای حلقه تکرار

- دقت کنید که در حلقه `for` عبارات جدا شده با کاما از چپ به راست اجرا می‌شوند ولی عبارات جدا شده توسط کاما در آرگومان‌های تابع و تعریف و اعلام متغیرها توسط عملگر کاما از چپ به راست اجرا نمی‌شوند، زیرا این کاما (ویرگول) ها عملگر ویرگول نیستند.

ساختارهای حلقه تکرار

- یکی دیگر از ساختارهای حلقه `do-while` است. این ساختار به صورت زیر نوشته می‌شود.

```
do
      statement
while(condition);
```

- در این ساختار دستور `statement` اجرا می‌شود، و پس از آن شرط حلقه بررسی می‌شود. این روند ادامه پیدا می‌کند تا وقتی که شرط حلقه برابر با صفر (مقدار نادرست) شود. از این ساختار معمولاً کمتر از `for` و `while` استفاده می‌شود.

ساختارهای حلقه تکرار

- در برنامه زیر یک عدد صحیح به رشته تبدیل می‌شود.

```
۱  /* itoa: convert n to characters in s */
۲  void
۳  itoa (int n, char s[])
۴  {
۵    int i, sign;
۶    if ((sign = n) < 0)   /* record sign */
۷       n = -n;    /* make n positive */
۸    i = 0;
```

ساختارهای حلقه تکرار

```
 ۹    do
۱۰      {          /* generate digits in reverse order */
۱۱            s[i++] = n % 10 + '0';    /* get next digit */
۱۲      }
۱۳    while ((n /= 10) > 0);    /* delete it */
۱۴    if (sign < 0)
۱۵       s[i++] = '-';
۱۶       s[i] = '\0';
۱۷    reverse (s);
۱۸ }
```

ساختارهای حلقه تکرار

- در اینجا از ساختار `do-while` استفاده می‌کنیم زیرا لازم است حداقل یک کاراکتر در آرایهٔ `s` باشد، حتی اگر `n` برابر با صفر باشد.

توقف و ادامه

- گاهی نیاز داریم از یک حلقه خارج شویم، قبل از اینکه حلقه به پایان رسیده باشد. دستور break برای خروج از حلقه به کار برده می‌شود.
- در تابع زیر، کاراکترهای خط فاصله و خط جدید و ستون جدید از انتهای یک رشته حذف می‌شوند و این کار ادامه پیدا می‌کند تا جایی‌که یا به ابتدای رشته برسیم یا به یک کاراکتر معمولی برخورد کنیم.

```
۱  /* trim: remove trailing blanks, tabs, newlines */
۲  int
۳  trim (char s[])
۴  {
۵    int n;
۶    for (n = strlen (s) - 1; n >= 0; n--)
۷        if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
۸        break;
۹    s[n + 1] = '\0';
۱۰   return n;
۱۱ }
```

توقف و ادامه

- دستور ادامه یا `continue` وقتی به کار می‌رود که می‌خواهیم ادامه دستورات در بدنه حلقه را ادامه ندهیم و اجرای حلقه را با تکرار بعدی ادامه دهیم.
- در حلقه `while` و `do-while` پس از اجرای دستور `continue` شرط حلقه بررسی می‌شود. ما در حلقه `for` پس از اجرای دستور `continue` ابتدا دستور افزایش حلقه اجرا می‌شود و پس از آن شرط حلقه بررسی می‌شود.
- دستور `break` هم در حلقه‌ها استفاده می‌شود و هم در `switch` ، اما دستور `continue` تنها در حلقه‌ها استفاده می‌شود.

توقف و ادامه

- در مثال زیر تنها عناصر غیر منفی در حلقه پردازش می‌شوند.

```
for (i = 0; i < n; i++)
   if (a[i] < 0)   /* skip negative elements */
    continue
   ... /* do positive elements */
```

توابع و ساختارهای برنامه

توابع

- توسط توابع می‌توان محاسبات طولانی را به قسمت‌های کوچکتر تقسیم کرد. این تقسیم‌بندی کمک می‌کند که یک برنامه خوانایی بیشتری داشته باشد، علاوه بر این که وقتی یک محاسبات معین توسط یک واحد مشخص تعریف شود، از آن واحد محاسباتی می‌توان در برنامه‌های دیگر نیز استفاده کرد.
- یک واحد محاسباتی توسط یک نام مشخص می‌شود. واحد محاسباتی برای انجام محاسبات تعدادی ورودی دریافت می‌کند و تعدادی باز می‌گرداند. چنین واحد محاسباتی در زبان سی یک تابع نامیده می‌شود.
- یک تابع یک واحد محاسباتی است که توسط یک نام مشخص می‌شود و تعدادی ورودی دریافت کرده و صفر یا یک خروجی باز می‌گرداند.

توابع

- برنامهٔ سی از تعداد زیادی تابع تشکیل شده است که هر یک وظیفه مشخصی دارند. این توابع می‌توانند در یک یا چند فایل مختلف قرار بگیرند. هر فایل به طور جداگانه کامپایل می‌شود و فایل‌های کامپایل شده توسط لینکر به یکدیگر متصل می‌شوند. در بسیاری از مواقع توابع مورد نیاز توسط افراد دیگر نوشته شده‌اند و فایل‌های کامپایل شده برای استفاده در اختیار ما قرار می‌گیرند. مجموعه توابعی که در یک برنامه تعریف شده و برای استفاده در اختیار دیگر برنامه‌ها قرار می‌گیرند، یک کتابخانه نامیده می‌شود.
- یک تابع را می‌توانیم قبل از تعریف کردن [1] اعلام کنیم [2]. در اعلام تابع فقط نام تابع و نوع ورودی و خروجی‌های آن مشخص می‌شوند. این اعلام جهت اطلاع کامپایلر است از اینکه چنین تابعی وجود دارد. در تعریف تابع، علاوه بر مشخص کردن نام و نوع ورودی‌ها و خروجی تابع باید مجموعه دستورات تابع در یک بلوک تعریف شوند. تعریف و اعلام یک تابع باید با یکدیگر همخوانی داشته باشند.

[1] definition
[2] declaration

توابع

- فرض کنید می‌خواهیم برنامه‌ای بنویسیم که در ورودی تعدادی خط را دریافت کرده و اگر در یک خط یک رشته داده شده وجود داشت، آن خط را در خروجی چاپ کند.
- پس الگوریتم این برنامه بدین صورت است : تا وقتی که خط بعدی وجود دارد، خط بعدی را از ورودی استاندارد بخوان و اگر آن خط شامل رشتهٔ داده شده است، آن خط را چاپ کن.

توابع

- گرچه همهٔ دستورات را می‌توانیم در بدنه تابع اصلی `main` قرار دهیم، راه بهتری این است که برنامه را طوری ساختاربندی کنیم که دستورات بدنهٔ اصلی کم و برنامه اصلی خوانا باشد. طراحی بهتر این است جزئیات را در توابع قرار دهیم و در بدنهٔ اصلی تنها بنویسیم چه‌کاری باید انجام نشود و به این که چگونه باید انجام شود.
- برای دریافت خط بعدی قبلا تابع `getline` را نوشتیم. برای چاپ یک جمله نیز از تابع `printf` استفاده می‌کنیم. تنها تابع مورد نیاز، تابعی است که یک رشته (یک خط) را دریافت کند و یک زیررشته (یک الگو) را در آن جستجو کند.
- می‌توانیم تابعی بنویسیم که به عنوان ورودی اول یک جمله و به عنوان ورودی دوم یک الگو را دریافت کرده و در جمله، الگوی مورد نظر را جستجو کند. این تابع در صورتی که الگوی ورودی را پیدا کرد اندیس کاراکتر اول الگوی یافت شده را در جمله باز می‌گرداند و در غیر اینصورت مقدار ۱- را باز می‌گرداند.
- در کتابخانهٔ استاندارد زبان سی تابعی به نام `strstr` وجود دارد که عملیات مشابه انجام می‌دهد.

توابع

- برنامه جستجوی رشته در ورودی استاندارد به صورت زیر نوشته می‌شود.

```
۱ #include <stdio.h>
۲ #define MAXLINE 1000      /* maximum input line length */
۳ int _getline (char line[], int max);
۴ int strindex (char source[], char searchfor[]);
۵ char pattern[] = "ould";    /* pattern to search for */
```

```c
/* find all lines matching pattern */
int main ()
{
  char line[MAXLINE];
  int found = 0;
  while (_getline (line, MAXLINE) > 0)
    if (strindex (line, pattern) >= 0)
      {
    printf ("%s", line);
    found++;
      }
  return found;
}
```

توابع

```
۲۰ /* getline: get line into s, return length */
۲۱ int
۲۲ _getline (char s[], int lim)
۲۳ {
۲۴   int c, i;
۲۵   i = 0;
۲۶   while (--lim > 0 && (c = getchar ()) != EOF && c != '\n')
۲۷     s[i++] = c;
۲۸   if (c == '\n')
۲۹      s[i++] = c;
۳۰   s[i] = '\0';
۳۱   return i;
۳۲ }
```

توابع

```
/* strindex: return index of t in s, -1 if none */
int
strindex (char s[], char t[])
{
  int i, j, k;
  for (i = 0; s[i] != '\0'; i++)
    {
        for (j = i, k = 0; t[k] != '\0' && s[j] == t[k]; j++, k++)
        ;
      if (k > 0 && t[k] == '\0')
         return i;
    }
  return -1;
}
```

توابع

- همانطور که دیدیم یک تابع توسط نوع خروجی، نام تابع، پارامترهای ورودی (متغیرهای ورودی) و نوع آنها که توسط کاما از یکدیگر جدا می‌شوند، و بدنه تابع تعریف می‌شود.
- بدنهٔ تابع می‌تواند تهی باشد، بنابراین کوچک‌ترین تابع به صورت `void f() {}` تعریف می‌شود. اگر نوع خروجی تابع تعیین نشود و نوع پیش فرض `int` است ولی در کامپایلرهای امروز پیام هشدار صادر می‌شود و نوع خروجی تابع باید مشخص شود.
- توابع با یکدیگر توسط پارامترها ارتباط برقرار می‌کنند و یک تابع توسط `return` یک مقدار باز می‌گرداند. وقتی یک تابع عبارت `expression` را باز می‌گرداند، نوع مقدار بازگشتی به نوع بازگشتی تابع تبدیل می‌شود.
- دستور `return` به تنهایی باعث خروج از تابع بدون بازگرداندن مقدار می‌شود.

توابع

- همچنین تابعی که یک تابع دیگر را فراخوانی می‌کند، می‌تواند مقدار بازگشتی تابع فراخوانی شده را نادیده بگیرد.
- مقداری که توسط تابع `main` بازگردانده می‌شود به محیط اجرا کنندهٔ آن یعنی سیستم عامل بازگردانده می‌شود. سیستم عامل می‌تواند تصمیم بگیرد که با مقدار بازگردانده شده چه عملیاتی انجام دهد.
- در برنامه‌ای که نوشتیم همهٔ توابع را در یک فایل قرار دادیم. می‌توانیم توابع را در فایل‌های جداگانه قرار دهیم و در هنگام کامپایل باید همهٔ فایل‌های مورد نیاز را مشخص کنیم. برای مثال برنامه قبل را می‌توانستیم به صورت `gcc main.c getline.c strindex.c` کامپایل کنیم.
- همچنین اگر فایل‌های `getline.c` و `strindex.c` قبلاً کامپایل شده بودند و می‌خواستیم آنها را به برنامه اصلی پیوند دهیم، می‌توانستیم به صورت `gcc main.c getline.o strindex.o` نیز برنامه را کامپایل کنیم.

توابع

- توابع می‌توانند هر نوع مقداری را بازگردانند. برای مثال فرض کنید می‌خواهیم تابعی بنویسیم که یک رشته را به عدد اعشاری تبدیل کند. در اینصورت تابع مورد نظر باید مقدار `double` بازگرداند.

توابع

- تابع تبدیل رشته به عدد اعشاری به صورت زیر نوشته می‌شود.

```
۱  #include <ctype.h>
۲  /* atof: convert string s to double */
۳  double
۴  atof (char s[])
۵  {
۶    double val, power;
۷    int i, sign;
۸    for (i = 0; isspace (s[i]); i++)   /* skip white space */
۹      ;
۱۰   sign = (s[i] == '-') ? -1 : 1;
۱۱   if (s[i] == '+' || s[i] == '-')
۱۲     i++;
```

```
۱۳    for (val = 0.0; isdigit (s[i]); i++)
۱۴      val = 10.0 * val + (s[i] - '0');
۱۵    if (s[i] == '.')
۱۶      i++;
۱۷    for (power = 1.0; isdigit (s[i]); i++)
۱۸      {
۱۹        val = 10.0 * val + (s[i] - '0');
۲۰        power *= 10;
۲۱      }
۲۲    return sign * val / power;
۲۳  }
```

توابع

- یک تابع را می‌توان در کنار تعریف متغیرها اعلام کرد. مزیت اعلام کرد. مزیت اعلام یک تابع این است که استفاده کنندهٔ آن خواهد دانست چگونه از تابع استفاده کند.

توابع

- در برنامه زیر تابع atof اعلام شده است.

```
۱  #include <stdio.h>
۲  #define MAXLINE 100
۳  /* rudimentary calculator */
۴  int main ()
۵  {
۶    double sum, atof (char[]);
۷    char line[MAXLINE];
۸    int getline (char line[], int max);
۹    sum = 0;
۱۰   while (_getline (line, MAXLINE) > 0)
۱۱     printf ("\t%g\n", sum += atof (line));
۱۲   return 0;
۱۳ }
```

توابع

- الزامی به اعلام توابع وجود ندارد و توابع را می‌توانیم تنها تعریف کنیم. در اینصورت در هنگام فراخوانی، کامپایلر به دنبال تعریف تابع می‌گردد.
- با فرض اینکه تابع atof برای تبدیل رشته به اعشاری تعریف شده است، می‌توانیم تابعی تعریف کنیم که یک رشته را به یک عدد صحیح تبدیل می‌کند.

```
۱ /* atoi: convert string s to integer using atof */
۲ int
۳ atoi (char s[])
۴ {
۵   double atof (char s[]);
۶   return (int) atof (s);
۷ }
```

توابع

- توابع می‌توانند توسط متغیرهای عمومی و متغیرهای خارجی نیز با یکدیگر اطلاعات به اشتراک بگذارند. متغیرهای محلی یک تابع در هنگام فراخوانی تعریف شده و در هنگام اتمام اجرای تابع از بین می‌روند، ولی متغیرهای عمومی مقدار خود را نگه می‌دارند.

توابع

- فرض کنید می‌خواهیم یک ماشین حساب بنویسیم که عملیات جمع و تفریق و ضرب و تقسیم انجام دهد. از آنجایی که محاسبهٔ عبارت‌های ریاضی که به صورت پسوندی [1] بیان می‌شوند برای کامپیوتر آسان‌تر است، به جای دریافت عبارت‌ها به صورت میانوندی [2] از نشانه گذاری پسوندی استفاده می‌کنیم.
- برای مثال عبارت میانوندی (1-2)*(4+5) در نشانه گذاری پسوندی به صورت 12-45+* بیان می‌شود. نشانه گذاری پسوندی غیر مبهم است پس به پرانتز گذاری نیاز ندارد.

[1] postfix
[2] infix

توابع

- پیاده‌سازی این ماشین حساب به یک پشته نیاز دارد. هر عملوند در یک پشته ذخیره می‌شود. وقتی به یک عملگر می‌رسیم، دو عملوند را از پشته خارج می‌کنیم و عملگر مورد نظر را بر روی آنها اعمال می‌کنیم. نتیجه محاسبات مجدداً در پشته ذخیره می‌شود.

- الگوریتم این ماشین حساب بدین صورت است.

۱. تا وقتی که مقدار بعدی در رشته عملگر یا عملوند است.
۲. اگر به یک عدد رسیدی
۳. عدد را در پشته وارد کن
۴. در غیراینصورت اگر به یک عملگر رسیدی
۵. دو عملوند را از پشته خارج کن و عملگر را بر روی آنها اعمال کن و نتیجه را در پشته ذخیره کن.
۶. در غیراینصورت اگر به کاراکتر خط جدید رسیدی
۷. مقدار نهایی را از پشته خارج کن و چاپ کن.
۸. در غیراینصورت پیام خطا چاپ کن.

توابع

- برای هر یک از عملیات پشته یک تابع تعریف می‌کنیم. این توابع نیاز دارند مکان آخر پشته را به یکدیگر منتقل کنند. آخرین مکان پشته را می‌توانیم به صورت یک متغیر عمومی تعریف کنیم.
- این برنامه را در چند فایل می‌نویسیم. در یک فایل بدنه اصلی برنامه تعریف می‌شود، در فایل دیگر توابع مربوط به پشته تعریف می‌شوند و در فایل دیگر توابع مورد نیاز برای دریافت ورودی.

توابع

- بدنه اصلی برنامه ماشین حساب به صورت زیر نوشته می‌شود.

```
۱  #include <stdio.h>
۲  #include <stdlib.h>        /* for atof() */
۳  #define MAXOP 100   /* max size of operand or operator */
۴  #define NUMBER '0'   /* signal that a number was found */
۵  int getop (char[]);
۶  void push (double);
۷  double pop (void);
۸  /* reverse Polish calculator */
۹  int main ()
۱۰ {
۱۱   int type;
۱۲   double op2;
۱۳    char s[MAXOP];
۱۴     while ((type = getop (s)) != EOF)
```

```
16      {
17    switch (type)
18       {
19          case NUMBER:
20               push (atof (s));
21               break;
22          case '+':
23               push (pop () + pop ());
24               break;
25          case '*':
26               push (pop () * pop ());
27               break;
28          case '-':
29               op2 = pop ();
30               push (pop () - op2);
31               break;
```

```
۳۲            case '/':
۳۳                 op2 = pop ();
۳۴                 if (op2 != 0.0)
۳۵                      push (pop () / op2);
۳۶                 else
۳۷                      printf ("error: zero divisor\n");
۳۸                 break;
۳۹            case '\n':
۴۰                 printf ("\t%.8g\n", pop ());
۴۱                 break;
۴۲            default:
۴۳                 printf ("error: unknown command %s\n", s);
۴۴                 break;
۴۵            }
۴۶       }
۴۷     return 0;
۴۸  }
```

توابع

- پشته مورد نیاز در برنامه ماشین حساب به صورت زیر نوشته می‌شود.

```
۱  #include <stdio.h>
۲  #define MAXVAL 100    /* maximum depth of val stack */
۳  int sp = 0;     /* next free stack position */
۴  double val[MAXVAL];     /* value stack */
۵  /* push: push f onto value stack */
۶  void
۷  push (double f)
۸  {
۹    if (sp < MAXVAL)
۱۰         val[sp++] = f;
۱۱   else
۱۲         printf ("error: stack full, can't push %g\n", f);
۱۳ }
```

توابع

```
۱۴ /* pop: pop and return top value from stack */
۱۵ double
۱۶ pop (void)
۱۷ {
۱۸   if (sp > 0)
۱۹         return val[--sp];
۲۰   else
۲۱     {
۲۲         printf ("error: stack empty\n");
۲۳         return 0.0;
۲۴     }
۲۵ }
```

توابع

- توابع مورد نیاز برای دریافت عملوندها به صورت زیر تعریف می‌شود.

```
۱  #include <stdio.h>
۲  #include <ctype.h>
۳  #define NUMBER '0'     /* signal that a number was found */
۴  int getch (void);
۵  void ungetch (int);
۶  /* getop: get next character or numeric operand */
۷  int
۸  getop (char s[])
۹    int i, c;
۱۰   while ((s[0] = c = getch ()) == ' ' || c == '\t')
```

```
11      ;
12    s[1] = '\0';
13    if (!isdigit (c) && c != '.')
14      return c;       /* not a number */
15    i = 0;
16    if (isdigit (c))    /* collect integer part */
17      while (isdigit (s[++i] = c = getch ()))
18        ;
19    if (c == '.')     /* collect fraction part */
20      while (isdigit (s[++i] = c = getch ()))
21        ;
22    s[i] = '\0';
23    if (c != EOF)
24      ungetch (c);
25    return NUMBER;
26  }
```

توابع

- در این توابع نیاز به یک تابع برای دریافت یک کاراکتر داریم که آن را با نام `getch` تعریف می‌کنیم. وقتی می‌خواهیم یک عدد دریافت کنیم نیاز داریم کاراکترها به ترتیب دریافت کنیم تا زمانی که کاراکتر بعدی یک رقم نباشد. از آنجایی که طول یک عدد مشخص نیست باید یکی یکی کاراکترها را دریافت کنیم و در نهایت کاراکتری دریافت خواهیم کرد که رقم نیست. در اینصورت نیاز داریم این کاراکتر خوانده شده را بازگردانیم. تابع `ungetch` برای بازگرداندن یک کاراکتر استفاده می‌شود.

توابع

- توابع دریافت کاراکتر به صورت زیر تعریف می‌شوند.

```
۱ #include <stdio.h>
۲ #define BUFSIZE 100
۳ char buf[BUFSIZE];  /* buffer for ungetch */
۴ int bufp = 0;  /* next free position in buf */
۵ int
۶ getch (void)  /* get a (possibly pushed-back) character */
۷ {
۸   return (bufp > 0) ? buf[--bufp] : getchar ();
۹ }
```

```
۱۰ void
۱۱ ungetch (int c)  /* push character back on input */
۱۲ {
۱۳   if (bufp >= BUFSIZE)
۱۴     printf ("ungetch: too many characters\n");
۱۵   else
۱۶     buf[bufp++] = c;
۱۷ }
```

برنامه در چند فایل

- معمولاً برنامه‌ها در زبان سی در چند فایل ذخیره می‌شوند و فایل‌ها به طور جداگانه کامپایل شده و به یکدیگر پیوند داده می‌شوند.
- حوزهٔ تعریف [1] یک نام، قسمتی از برنامه است که در آن نام تعریف شده است. برای مثال حوزهٔ تعریف یک متغیر که در ابتدای یک تابع تعریف شده است، بلوک دستورات آن تابع است. دو متغیر محلی در دو تابع مختلف دو متغیر کاملا متفاوت هستند. حوزهٔ تعریف یک متغیر عمومی در ابتدای یک فایل، همهٔ آن فایل است که شامل توابع تعریف شده در آن فایل می‌شود. همچنین یک تابع که در یک فایل تعریف و اعلام شده است در همهٔ آن فایل قابل استفاده است و بنابراین حوزهٔ تعریف آن همه آن فایل است.

[1] scope

برنامه در چند فایل

- اگر تابع `f` بعد از تابع `g` تعریف شده باشد، تابع `f` در تابع `g` تابع استفاده نیست مگر اینکه `f` قبل از `g` اعلام شده باشد.
- اگر یک متغیر در یک فایل دیگر تعریف شده باشد، توسط کلیدواژه `extern` می‌توان آن متغیر را در فایل جاری اعلام کرد. در این صورت فایل دیگر باید توسط `include` به فایل جاری معرفی شود. یک متغیر یا یک تابع فقط یک بار می‌تواند تعریف شود. ولی می‌توان در چند مکان اعلام شود.

برنامه در چند فایل

- معمولاً توابع در یک برنامه سی در یک فایل سرتیتر [1] با پسوند `h`. اعلام می‌شوند و در یک فایل سورس با پسوند `c`. تعریف می‌شوند.
- برای مثال در برنامهٔ ماشین حساب می‌توان همهٔ توابع را در یک فایل `calc.h` اعلام کرد و توابع مربوط به پشته را در فایل `stack.h` ، توابع دریافت عملوندها را در فایل `getop.c` ،توابع مربوط به دریافت کاراکتر را در فایل `getch.c` و تابع بدنه اصلی را در فایل `main.c` ذخیره کرد. برای اینکه هر یک از فایل‌های سورس که به توابع فایل‌های سورس دیگر نیاز دارند بتوانند کامپایل شوند، نیاز داریم در همهٔ فایل‌ها `calc.h` را اضافه کنیم. همچنین در فایل `main.c` نیاز به اعلام توابع داریم که برای اعلام آنها `calc.h` را به فایل اصلی اضافه می‌کنیم. وقتی یک متغیر در یک فایل تعریف شود، آن متغیر می‌تواند توسط کلیدواژهٔ `extern` به عنوان متغیر خارجی در یک فایل دیگر استفاده شود.

[1] header file

متغیرهای ایستا

- اگر بخواهیم یک متغیر به طور خصوصی برای یک فایل تعریف شود و در فایل‌های دیگر قابل استفاده نباشد، آن متغیر باید به صورت ایستا با کلیدواژه `static` تعریف شود.
- یک متغیر عمومی ایستا در یک فایل در فایل‌های دیگر قابل استفاده نیست.
- همینطور می‌توان یک تابع را به طور ایستا تعریف کرد. تابع ایستا در یک فایل در فایل‌های دیگر قابل استفاده نیست.
- اگر یک متغیر به صورت ایستا در یک تابع تعریف شود، پس از اتمام فراخوانی تابع مقدار آن از بین نمی‌رود. در واقع متغیرهای ایستا حتی اگر محلی باشند، پس از تعریف تا اتمام برنامه در حافظه باقی می‌مانند.

متغیرهای رجیستر

- متغیرهای پر استفاده معمولاً با استفاده از کلیدواژهٔ `register` تعریف می‌شوند. متغیرهای رجیستر در واقع برروی رجیستر پردازنده قرار می‌گیرند و نه برروی حافظه. بدین ترتیب سرعت دسترسی به آنها بسیار بیشتر است. از آنجایی که تعداد رجیسترهای پردازنده محدود است، در استفاده از متغیرهای رجیستر نیز محدودیت وجود دارد.

بلوک‌ها

- در زبان سی، یک متغیر می‌تواند در یک بلوک تعریف شود. این بلوک می‌تواند هرگونه بلوکی باشد مثلاً در بلوک متعلق به یک دستور `if` یا `for` می‌توان متغیر تعریف کرد. در این صورت متغیر فقط در آن بلوک تعریف شده و خارج از بلوک قابل دسترسی نیست.
- در مثال زیر متغیر `i` در بلوک `if` تعریف شده است و خارج از بلوک `if` قابل استفاده نیست.

```
۱ if (n > 0) {
۲   int i;  /* declare a new i */
۳   for (i = 0; i < n; i++)
۴     ...
۵ }
```

بلوک‌ها

- یک متغیر می‌تواند به صورت عمومی تعریف شود و سپس در یک بلوک مجدداً با همان نام تعریف شود. در این صورت، در بلوک متغیر محلی استفاده می‌شود و در خارج از بلوک متغیر عمومی.
- در مثال زیر متغیر x در داخل تابع از نوع double است و مقدار متفاوتی از متغیر عمومی x دارد.

```
۱ int x;
۲ int y;
۳ void f(double x) {
۴   double y;
۵ }
```

بلوک‌ها

- متغیر پس از تعریف باید مقداردهی اولیه شوند، در غیراینصورت مقدار آنها معتبر نخواهد بود.
- مقداردهی اولیه با عملگر تساوی انجام می‌شود. برای مثال `int low = 0` متغیر `low` را مقداردهی اولیه می‌کند.
- آرایه‌ها را می‌توانند با تعیین مقادیر عناصر آرایه مقداردهی اولیه شوند. برای مثال `int days[] = { 31,24 }` یک آرایه با دو عنصر با مقادیر 31 و 24 می‌سازد.
- مقداردهی اولیه رشته‌ها می‌تواند به دو صورت انجام شود. یک رشته آرایه‌ای از کاراکترهاست بنابراین می‌توانیم بنویسیم `char str[] = {'h' , 'i'}` و همچنین یک رشته می‌تواند به صورت `char str = "h"` نیز مقداردهی اولیه شود.

توابع بازگشتی

- توابع در زبان سی می‌توانند به صورت بازگشتی نیز فراخوانی شوند. یک تابع می‌تواند خود را فراخوانی کند که به این فراخوانی بازگشتی گفته می‌شود. توابعی که فراخوانی بازگشتی دارند را توابع بازگشتی [1] می‌نامیم. توابع می‌توانند خود را به صورت مستقیم یا غیر مستقیم فراخوانی کنند.
- فرض کنید می‌خواهیم برنامه‌ای بنویسیم که یک عدد صحیح را به یک رشته تبدیل کند. روش اول این است که ارقام با ارزش پایین‌تر به ترتیب از عدد جدا شده و رشته از راست به چپ ساخته شود. روش دوم این است که یک فراخوانی بازگشتی استفاده کنیم.
- وقتی می‌خواهیم عدد `Ax` را به رشته تبدیل کنیم به طوری‌که `A` یک عدد و `x` یک رقم است در واقع باید ابتدا `A` را به رشته تبدیل کرده و سپس `x` را به صورت کاراکتر چاپ کنیم.

[1] recursive function

توابع بازگشتی

- این برنامه را می‌توانیم به صورت زیر بنویسیم.

```
۱  #include <stdio.h>
۲  /* printd: print n in decimal */
۳  void
۴  printd (int n)
۵  {
۶    if (n < 0)
۷      {
۸        putchar ('-');
۹        n = -n;
۱۰     }
۱۱   if (n / 10)
۱۲     printd (n / 10);
۱۳   putchar (n % 10 + '0');
۱۴ }
```

توابع بازگشتی

- یک مثال دیگر برای یک تابع بازگشتی الگوریتم مرتب‌سازی سریع است. با استفاده از این الگوریتم مرتب‌سازی، یک عنصر از آرایه انتخاب می‌شود و عناصر کوچک‌تر از آن به سمت چپ و عناصر بزرگ‌تر از آن به سمت راست عنصر انتخاب شده انتقال داده می‌شوند. این عملیات به صورت بازگشتی برای زیر آرایه‌ها تکرار می‌شود تا کل آرایه مرتب شود.

توابع بازگشتی

- الگوریتم مرتب‌سازی سریع به صورت زیر است.

```
۱  /* qsort: sort v[left]...v[right] into increasing order */
۲  void
۳  qsort (int v[], int left, int right)
۴  {
۵    int i, last;
۶    void swap (int v[], int i, int j);
۷    if (left >= right)  /* do nothing if array contains */
۸      return;  /* fewer than two elements */
۹    swap (v, left, (left + right) / 2);  /* move partition elem */
۱۰   last = left;  /* to v[0] */
۱۱   for (i = left + 1; i <= right; i++)  /* partition */
۱۲     if (v[i] < v[left])
۱۳       swap (v, ++last, i);
```

توابع بازگشتی

```
14    swap (v, left, last);  /* restore partition elem */
15    qsort (v, left, last - 1);
16    qsort (v, last + 1, right);
17  }
```

توابع بازگشتی

- عملیات جابجایی دو عنصر آرایه به صورت زیر انجام می‌شود.

```
۱  /* swap: interchange v[i] and v[j] */
۲  void
۳  swap (int v[], int i, int j)
۴  {
۵    int temp;
۶    temp = v[i];
۷    v[i] = v[j];
۸    v[j] = temp;
۹  }
```

توابع بازگشتی

- در یک فراخوانی بازگشتی، درواقع فراخوانی‌های پی‌درپی برروی پشته‌ای در حافظه ذخیره می‌شوند تا مقادیر آنها بعدا مورد استفاده قرار بگیرد.
- یک تابع بازگشتی از تابع غیربازگشتی معادل آن سریع‌تر نیست، اما کوتاهترین نوشته می‌شود و نوشتن و خواندن برنامه را آسان‌تر می‌کند.

پیش پردازش

- قبل از مرحله کامپایل در زبان سی، یک مرحله پیش پردازش [1] وجود دارد. در این مرحله فایل‌هایی که نام آنها توسط کلمه include به فایل اضافه شده‌اند، محتوایشان نیز به فایل اضافه می‌شود. همچنین نمادهای ثابتی که توسط define تعریف شده‌اند، مقادیرشان در فایل جایگزین نمادها می‌شوند.
- فایل‌های به صورت #include <filename> مشخص شده‌اند، در آدرس‌های تعیین شده توسط سیستم عامل جستجو می‌شوند و فایل‌های که به صورت #include "filename" استفاده شد، در مکانی که برنامه ذخیره شده است جستجو می‌شوند.
- توسط کلمه define در زبان سی می‌توان یک می‌توان یک نماد تعریف کرد. این نمادها ماکرو نامیده می‌شوند. این نمادها می‌توانند جایگزین هر قسمتی از برنامه شوند. برای مثال برای تعریف یک حلقه بینهایت می‌توانیم بنویسیم.

```
1 #define forever for (;;)
```

[1] prepoccessing

پیش پردازش

- ماکروها را می‌توان با استفاده از پارامتر نیز تعریف کرد. در مثال زیر، نماد max با دو پارامتر با معادل آن در کد جایگزین می‌شود.

```
1 #define max(A,B) ((A) > (B) ? (A) : (B))
```

یک برنامه‌ای که از این ماکرو استفاده می‌کند max(p+q , r+s) با عبارت
((p + q) > (r + s) ? (p + q) : (r + s)) جایگزین می‌شود.

- در ماکروها باید به پرانتز گزاری توجه کرد. برای مثال اگر ماکرویی به صورت

```
1 #define square(x) x*x       /* WRONG */
```

تعریف کنیم، آنگاه square(z+1) معادل خواهد بود، z+1 * z+1 که معادل است با 2z+1

پیش پردازش

- اگر یک نماد توسط یک ماکرو تعریف شود، می‌توان در قطعه‌ای از برنامه تعریف را توسط کلیدواژه undef لغو کرد.
- برای مثال :

```
۱ #undef getchar
۲ int getchar() {...}
```

- اگر در عبارت جایگزین شده در یک ماکرو قبل از یک متغیر از علامت # استفاده کنیم، آن متغیر در عبارت جایگزین شده در میان دو علامت نقل قول قرار می‌گیرد.
- برای مثال اگر داشته باشیم :

```
۱ #define dprint(expr) printf(#expr " = %g\n" , expr)
```

آنگاه با اجرای dprint(x/y) در واقع دستور printf("x/y" "=% g \n" , x/y) اجرا می‌شود.

پیش پردازش

- در ماکروها می‌توانیم از علامت ## استفاده کنیم برای اتصال دو پارامتر به یکدیگر استفاده می‌شود.
- برای مثال اگر داشته باشیم :

```
1 #define paste(front, back) front ## back
```

آنگاه با اجرای `paste(name,1)` عبارت `name1` به دست می‌آید.

پیش پردازش

- دستورات ماکرویی برای کنترل کردن ماکروها و اجرای شرطی آنها وجود دارد. برای مثال دستور ماکروی `#if` یک ورودی دریافت می‌کند و در صورتی که ورودی غیر صفر باشد دستورات ماکروی بعد از آن تا رسیدن به ماکروی `#endif` اجرا می‌شوند. همچنین دستورات ماکروی `#elif` و `#else` نیز برای اجرای دستورات `else if` و `else` در ماکرو وجود دارند.

پیش پردازش

- فرض کنید می‌خواهیم در یک برنامه مطمئن شویم که یک فایل سر تیتر تنها یک بار به یک فایل افزوده می‌شود.
- برای این کار یک متغیر در هنگام اعلام توابع در فایل سر تیتر تعریف می‌کنیم و سپس اطمینان حاصل می‌کنیم که این متغیر تنها یک بار تعریف شده است.
- برای مثال فایل `hdr.h` به صورت زیر تعریف شده است.

```
۱ #if !defined(HDR)
۲ #define HDR
۳ /* contents of hdr.h go here */
۴ #endif
```

پیش پردازش

- در برنامه زیر با توجه به نوع سیستم عامل عملیات متفاوت انجام می‌دهیم.

```
۱  #if SYSTEM == SYSV
۲  #define HDR "sysv.h"
۳  #elif SYSTEM == BSD
۴  #define HDR "bsd.h"
۵  #elif SYSTEM == MSDOS
۶  #define HDR "msdos.h"
۷  #else
۸  #define HDR "default.h"
۹  #endif
۱۰ #include HDR
```

پیش پردازش

- ماکروی `ifndef` با یک پارامتر ورودی، بدین معنی است که اگر پارامتر ورودی تعریف نشده بود، آنگاه عملیات بعدی را تا رسیدن به `endif` انجام بده.

```
۱ #ifndef HDR
۲ #define HDR
۳ /* contents of hdr.h go here */
۴ #endif
```

اشاره‌گرها و آرایه‌ها

اشاره‌گرها

- یک اشاره‌گر [1] متغیری است که آدرس یک متغیر را ذخیره می‌کند.
- اشاره‌گرها به کثرت در زبان سی استفاده می‌شوند. گاهی استفاده از اشاره‌گرها بدین دلیل است که برنامه‌نویسی را ساده‌تر می‌کنند و بدون آنها توصیف محاسبات پیچیده می‌شود و گاهی بدین دلیل است که با استفاده از آنها می‌توان برنامه‌های فشرده‌تر و کارامدتری نوشت.
- اشاره‌گرها و آرایه بسیار به یکدیگر شبیه هستند و در این قسمت به معرفی اشاره‌گرها و بررسی آنها در کنار آرایه‌ها خواهیم پرداخت.

[1] pointer

اشاره‌گرها

- یک سیستم کامپیوتری معمولاً دارای آرایه‌ای از مکان‌های حافظه است که به صورت پی‌درپی و شماره‌گذاری شده به دنبال یکدیگر قرار گرفته شده‌اند.
- مقدار یک متغیر از نوع `char` در یک بایت از این مکان‌های حافظه قرار می‌گیرد. مقدار یک متغیر از نوع `short` در دو بایت پی‌درپی در حافظه ذخیره می‌شود.
- یک اشاره‌گر یک متغیر ۴ یا ۸ بایتی (بسته به نوع ماشین) است که آدرس یک مکان حافظه را نگهداری می‌کند.
- بنابراین اگر `c` یک متغیر از نوع `char` باشد و `p` یک متغیر از نوع اشاره‌گر باشد، متغیر `p` می‌تواند آدرس متغیر `c` در حافظه را نگهداری کند.

اشاره‌گرها

- آدرس یک متغیر را می‌توانیم با عملگر امپرسند & دریافت کنیم، بنابراین عبارت `p = &c` آدرس متغیر `c` را دریافت می‌کند و این آدرس را در اشاره‌گر `p` ذخیره می‌کند. می‌گوییم اشاره‌گر `p` به متغیر `c` اشاره می‌کند.
- عملگر & تنها بر روی متغیرهایی که بر روی حافظه هستند عمل می‌کند و برروی عبارات و نمادهای ثابت و متغیرهای رجیستر نمی‌توان آن را اعمال کرد.
- عملگر ستاره * یک عملگر رفع ارجاع [1] است. وقتی این عملگر بر روی یک اشاره‌گر اعمال می‌شود، مقدار متغیری را که اشاره‌گر به آن اشاره می‌کند بازمی‌گرداند.
- فرض کنید `x` و `y` دو متغیر از نوع عدد صحیح هستند و `ip` یک اشاره‌گر از نوع عدد صحیح (متغیر از نوع اشاره‌گر به عدد صحیح) است.

[1] dereference

اشاره‌گرها

- در عبارات زیر نشان داده شده است چگونه از اشاره‌گرها استفاده می‌کنیم.

```
۱ int x = 1, y = 2, z[10];
۲ int *ip; /* ip is a pointer to int */
۳ ip = &x; /* ip now points to x */
۴ y = *ip; /* y is now 1 */
۵ *ip = 0; /* x is now 0 */
۶ ip = &z[0]; /* ip now points to z[0] */
```

شاره‌گرها

- در تعریف اشاره‌گر `ip` می‌گوییم `ip*` یک عدد صحیح است، یعنی مقداری که `ip` به آن اشاره می‌کند یک عدد صحیح است و بنابراین `ip` یک اشاره‌گر است به یک متغیر از نوع عدد صحیح.
- در عبارت زیر در واقع می‌گوییم `dp*` و تابع `atof` از نوع `double` هستند و تابع `atof` یک اشاره‌گر از نوع کاراکتر دریافت می‌کند. در واقع `dp` یک اشاره‌گر به یک عدد `double` است و تابع `atof` مقداری از نوع `double` بازمی‌گرداند.

```
1 double *dp, atof(char *);
```

- هر اشاره‌گر به یک متغیر از نوع معین اشاره می‌کند. البته یک استثنا برای اشاره‌گر به نوع `void` وجود دارد که در مورد آن بعدها بیشتر صحبت خواهیم کرد.

اشاره‌گرها

- فرض کنید `ip` اشاره‌گری است که به متغیر `x` اشاره می‌کند. در این‌صورت عبارت `*ip = *ip + 10` مقدار متغیر `x` را ۱۰ واحد افزایش می‌دهد. عبارت `y = *ip + 1` مقدار متغیری که `ip` به آن اشاره می‌کند را دریافت می‌کند و پس از افزودن یک واحد مقدار به دست آمده را در متغیر `y` ذخیره می‌کند. عبارت `*ip += 1` مقدار متغیری که `ip` به آن اشاره می‌کند را یک واحد افزایش می‌دهد. که معادل است با `++*ip` یا `++(*ip)`.

- پرانتزگذاری در عبارت `++(*ip)` ضروری است، زیرا وابستگی [1] عملگر `++` و `*` از راست به چپ است. بنابراین در عبارت `++*ip` ابتدا مقدار `ip` یک واحد افزایش می‌یابد (به خانه حافظه بعدی اشاره می‌کند) و سپس مقدار آن دریافت می‌شود.

- همچنین می‌توانیم مقدار یک اشاره‌گر (که یک آدرس است) را در یک اشاره‌گر دیگر ذخیره کنیم برای مثال اگر `ip` و `iq` دو اشاره‌گر باشند، می‌توانیم بنویسیم `iq = ip` که در اینصورت `iq` به متغیری اشاره خواهد کرد که `ip` نیز به آن اشاره می‌کند.

[1] associativity

اشاره‌گرها به عنوان پارامتر تابع

- از آنجایی که در زبان سی آرگومان‌ها با مقدار به توابع ارسال می‌شوند، یک تابع نمی‌تواند مقدار یک آرگومان را تغییر دهد.
- برای مثال فرض کنید در یک تابع می‌خواهیم مقدار دو آرگومان را توسط تابع swap جابجا کنیم. این جابجایی ممکن است در یک الگوریتم مرتب‌سازی برای جابجایی دو عنصر مورد استفاده قرار بگیرد.
- حال فرض کنید تابع swap را به صورت زیر بنویسیم.

```
۱ void swap(int x, int y)   /* WRONG */
۲ {
۳   int temp;
۴   temp = x;
۵   x = y;
۶   y = temp;
۷ }
```

اشاره‌گرها به عنوان پارامتر تابع

- چون فراخوانی توابع با مقدار است، تابع `swap` مقدار دو متغیر را جابجا نمی‌کند، بلکه تنها مقدار دو متغیر پارامتر خود را جابجا می‌کند که در مقدار آرگومان‌ها تأثیری نمی‌گذارد، زیرا مقدار آرگومان تابع `swap` در مقدار پارامترهای `x` و `y` کپی می‌شود.
- به عبارت دیگر با فراخوانی تابع `swap(a,b)` در واقع مقدار `a` در `x` و مقدار `b` در `y` کپی می‌شود و تغییر متغیرهای `x` و `y` در مقدار متغیرهای `a` و `b` بی تأثیر است.
- برای اینکه تابع بتواند مقدار آرگومان‌ها را تغییر دهد، باید اشاره‌گری از آرگومان‌ها را به تابع ارسال کنیم تا تابع با در دست داشتن آدرس آرگومان‌ها بتواند مقدار آنها را تغییر دهد پس تابع `swap` باید دو پارامتر اشاره‌گر دریافت کند.

اشاره‌گرها به عنوان پارامتر تابع

- تابع زیر مقدار آرگومان‌هایی که با تابع ارسال می‌شوند را جابجا می‌کند.

```
۱ void swap(int *px, int *py) /* interchange *px and *py */
۲ {
۳   int temp;
۴   temp = *px;
۵   *px = *py;
۶   *py = temp;
۷ }
```

- حال می‌توانیم تابع swap(&a,&b) را فراخوانی کنیم. در این صورت اشاره‌گر px آدرس a و اشاره‌گر py آدرس را نگهداری خواهد کرد و بنابراین می‌تواند با استفاده از آدرس آنها به مقدارشان دسترسی پیدا کرده و مقادیر آنها را جابجا کند.

اشاره‌گرها به عنوان پارامتر تابع

- پارامترهایی که از نوع اشاره‌گر هستند توابع را قادر می‌سازند که آرگومان‌هایی که به آنها ارسال می‌شوند را تغییر دهند.

اشاره‌گرها به عنوان پارامتر تابع

- فرض کنید می‌خواهیم تابعی به نام `getint` بنویسیم که در هر بار فراخوانی یک عدد صحیح را از ورودی استاندارد دریافت می‌کند.
- برای دریافت یک عدد ترتیب ارقام دریافت شده تا وقتی که ورودی یک رقم نباشد. این ارقام تبدیل به عدد می‌شوند و در اشاره‌گر ورودی ذخیره می‌شوند. در صورتی که در ورودی کاراکتر پایان فایل دریافت شود، تابع این کاراکتر را باز می‌گرداند.
- درواقع در طراحی این تابع می‌توانیم به صورتی دیگر عمل کنیم بدین صورت که تابع `getint` مقدار عدد دریافت شده از ورودی را بازگرداند، اما در این صورت چگونه به فراخوانی کنندهٔ `getint` بگوییم عدد دیگری وجود ندارد ؟ درواقع در این تابع نیاز داریم دو مقدار را به فراخوانی کنندهٔ تابع بازگردانیم : مقدار عدد خوانده شده و کاراکتر پایان فایل در صورتی که به پایان فایل برسیم.
- در چنین مواقعی معمولاً مقادیر کنترلی را در خروجی تابع باز می‌گردانیم و مقادیر مورد نیاز را در پارامترهای اشاره‌گر در ورودی ذخیره می‌کنیم.

اشاره‌گرها به عنوان پارامتر تابع

- بنابراین از تابع getint می‌توانیم به صورت زیر برای دریافت آرایه‌ای از اعداد استفاده کنیم.

```
۱ int n, array[SIZE], getint(int *);
۲ for (n = 0; n < SIZE && getint(&array[n]) != EOF; n++);
```

- در هر یک از عناصر array[n] عدد ورودی بعدی قرار می‌گیرد تا وقتی که یا به پایان رشتهٔ ورودی برسیم و یا ظرفیت آرایهٔ array تکمیل شود.

اشاره‌گرها به عنوان پارامتر تابع

- برنامهٔ دریافت اعداد صحیح به صورت زیر نوشته می‌شود.

```
۱  int getch (void);
۲  void ungetch (int);
۳  /* getint: get next integer from input into *pn */
۴  int
۵  getint (int *pn)
۶  {
۷    int c, sign;
۸    while (isspace (c = getch ()))  /* skip white space */
۹      ;
۱۰   if (!isdigit (c) && c != EOF && c != '+' && c != '-')
۱۱     {
۱۲       ungetch (c);  /* it is not a number */
۱۳       return 0;
۱۴     }
```

اشاره‌گرها به عنوان پارامتر تابع

```
۱۵    sign = (c == '-') ? -1 : 1;
۱۶    if (c == '+' || c == '-')
۱۷      c = getch ();
۱۸    for (*pn = 0; isdigit (c), c = getch ())
۱۹      *pn = 10 * *pn + (c - '0');
۲۰    *pn *= sign;
۲۱    if (c != EOF)
۲۲      ungetch (c);
۲۳    return c;
۲۴  }
```

اشاره‌گرها و آرایه‌ها

- اشاره‌گرها و آرایه‌ها به یکدیگر شباهت زیادی دارند.
- عملیاتی که برروی آرایه‌ها توسط عملگر زیرنویس انجام می‌دهیم، در اشاره‌گرها نیز امکان‌پذیر است.
- تعریف آرایهٔ `a[10]` را در نظر بگیرید. توسط این تعریف درواقع آرایه‌ای از ۱۰ بلوک حافظه را که هر کدام حاوی یک عدد صحیح خواهند بود در اختیار می‌گیریم. به هریک از عناصر آرایه می‌توانیم توسط عملگر زیرنویس به صورت `a[0]` ، `a[1]` ، ... و `a[9]` دسترسی پیدا کنیم.

اشاره‌گرها و آرایه‌ها

- حال اگر اشاره‌گر `pa*` را به صورت `int *pa` تعریف کنیم، می‌توانیم مقدار آن را برابر با آدرس عنصر اول آرایه با دستور `pa = &a[0]` قرار دهیم. بنابراین `pa` به اولین عنصر آرایه اشاره خواهد کرد.
- حال می‌توانیم مقدار متغیری که `pa` به آن اشاره می‌کند را توسط `x = *pa` دریافت کنیم. اگر `pa` به عنصر صفرم آرایه اشاره کند، `pa + 1` به عنصر اول و `pa + i` به عنصر `i` ام آرایه اشاره می‌کند. بنابراین `*(pa + 1)` مقدار `a[1]` و `*(pa + i)` مقدار `a[i]` را بازمی‌گردانند.
- عبارت `pa + 1` در واقع به سلول بعدی در حافظه اشاره می‌کند و به نوع متغیری که `pa` به آن اشاره می‌کند بستگی ندارد. پس اگر `pa` از نوع `char` باشد `pa + 1` در واقع یک بایت به جلو حرکت می‌کند و اگر `pa` از نوع `int` باشد، `pa + 1` در واقع چهار بایت در حافظه حرکت می‌کند.

اشاره‌گرها و آرایه‌ها

- نام یک آرایه درواقع یک اشاره‌گر است و یک متغیر از نوع آرایه درواقع اولین عنصر آرایه را نگهداری می‌کند.
- بنابراین به جای `pa = &a[0]` می‌توانیم بنویسیم `pa = a` زیرا نام آرایه برابراست با آدرس اولین عنصر آرایه.

اشاره‌گرها و آرایه‌ها

- درواقع به عناصر آرایه نیز می‌توانیم مانند اشاره‌گرها به عملگر * دسترسی پیدا کنیم. بنابراین برای دسترسی به عنصر i ام آرایه a می‌توانیم بنویسیم `*(a+i)` که معادل است با `a[i]`.
- درواقع در زبان سی `a[i]` در هنگام ارزیابی به `*(a+i)` تبدیل می‌شود پس هر دو عبارت معادل هستند.
- همچنین `&a[i]` معادل است با `a+i`
- اگر `pa` یک اشاره‌گر باشد، عبارت `*(pa+i)` می‌تواند به صورت `pa[i]` نیز نوشته شود.

اشاره‌گرها و آرایه‌ها

- نتیجه اینکه عبارات نوشته شده به صورت نام آرایه و عملگر زیرنویس معادل هستند با عبارات نوشته شده به صورت نام اشاره‌گر و آفست از عنصر اول.
- با این حال، یک تفاوت بین اشاره‌گرها و آرایه‌ها وجود دارد. از آنجایی که اشاره‌گرها متغیر هستند می‌توانیم عملیاتی مانند متغیرها برروی آنها انجام دهیم برای مثال می‌توانیم بنویسیم `pa = a` یا `pa++` اما نام آرایه‌ها متغیر نیستند پس `a = pa` و `a++` عبارات نادرستی هستند.
- وقتی یک آرایه به یک تابع ارسال می‌شود، درواقع آدرس مکان اول آرایه به تابع ارسال می‌شود.

اشاره‌گرها و آرایه‌ها

- می‌خواهیم تابعی بنویسیم که طول یک رشته را محاسبه کند. با استفاده از اشاره‌گرها این تابع را به صورت زیر می‌نویسیم.

```
۱ /* strlen: return length of string s */
۲ int strlen(char *s)
۳ {
۴   int n;
۵   for (n = 0; *s != '\0', s++)
۶     n++;
۷   return n;
۸ }
```

اشاره‌گرها و آرایه‌ها

- در واقع در این تابع اشاره‌گر s یک کپی است از اشاره‌گر ارسال شده به تابع از طریق آرگومان ورودی تابع. پس اگر آرگومان ورودی تابع یک آرایه باشد، اشاره‌گر s یک کپی از آدرس آرایه را ذخیره می‌کند.
- در برنامه قبل عبارت s++ تأثیری در آرگومان ورودی تابع نمی‌گذارد.

اشاره‌گرها و آرایه‌ها

- این تابع را می‌توانیم به گونه‌های متفاوت فراخوانی کنیم.

```
1 strlen("hello, world"); /* string constant */
2 strlen(array); /* char array[100]; */
3 strlen(ptr); /* char *ptr; */
```

- در پارامتر ورودی تابع `char s[]` و `char *s` معادل یکدیگرند.

اشاره‌گرها و آرایه‌ها

- همچنین گاه ممکن است قسمتی از یک آرایه را به یک تابع ارسال کنیم. برای مثال می‌توانیم بنویسیم `f(&a[2])` و یا `f(a+2)` تا آرایه `a` را به شروع از عنصر `a[2]` به تابع `f` ارسال کنیم.
- تابع `f` می‌تواند به دو صورت `f(int arr[])` یا `f(int *arr)` تعریف شود.
- همچنین اگر بدانیم که عناصر قبل از عنصر `p[0]` در حافظه وجود دارند، می‌توانیم به آنها با اندیس‌های منفی به صورت `p[-1]` و `p[-2]` و ... دسترسی پیدا کنیم.

محاسبات آدرس‌ها

- اگر `p` یک اشاره‌گر باشد، آنگاه `++p` یک واحد به `p` می‌افزاید تا به عنصر بعدی در حافظه اشاره کند و `p+=i` درواقع `i` واحد به آدرس `p` می‌افزاید.
- فرض کنید می‌خواهیم تابعی به نام `alloc(n)` بنویسیم که اشاره‌گری به ابتدای `n` عنصر در حافظه بازمی‌گرداند. فراخوانی کنندهٔ تابع `alloc(n)` می‌تواند با فراخوانی این تابع `n` مکان در حافظه را برای ذخیره یک رشته `n` حرفی در اختیار بگیرد.
- همچنین می‌خواهیم `afree(p)` را پیاده‌سازی کنیم که مکانی که اشاره‌گر `p` در حافظه اشغال کرده را آزاد کند. تا بتوانیم از آن مکان حافظه مجددا استفاده کنیم.
- در کتابخانه استاندارد دو تابع `malloc` و `free` بدین منظور پیاده‌سازی شده‌اند. در اینجا به مطالعه پیاده‌سازی ساده این دو تابع می‌پردازیم.

محاسبات آدرس‌ها

- فرض کنید بافری به نام `allocbuf` در اختیار داریم که تخصیص و آزادسازی حافظه را برروی آن بافر انجام می‌دهیم.
- این بافر را به صورت ایستا تعریف می‌کنیم چون نمی‌خواهیم برنامه‌های دیگر به صورت مستقیم به این بافر دسترسی داشته باشند.
- همچنین در این برنامه به یک اشاره‌گر به نام `allocp` نیاز داریم که به آدرس مکان در دسترس بعدی در بافر اشاره می‌کند.
- وقتی می‌خواهیم توسط تابع `alloc` تعداد `n` کاراکتر در حافظه را رزرو کنیم، باید ابتدا بررسی شود آیا `allocbuf` این مقدار مکان آزاد در اختیار دارد یا خیر. اگر مکان آزاد موجود بود آدرس فعلی `allocp` بازگردانده می‌شود و اشاره‌گر `allocp` به `allocp + n` اشاره می‌کند.

محاسبات آدرس‌ها

- اگر فضای مورد نیاز در بافر وجود نداشت تابع `alloc` مقدار صفر را بازمی‌گرداند.
- همچنین با فراخوانی `afree(p)` اشاره‌گر `allocp` به `p` اشاره خواهد کرد.

محاسبات آدرس‌ها

- این توابع به صورت زیر نوشته می‌شوند.

```
۱  #define ALLOCSIZE 10000   /* size of available space */
۲  static char allocbuf[ALLOCSIZE];  /* storage for alloc */
۳  static char *allocp = allocbuf;    /* next free position */
۴  char *
۵  alloc (int n)    /* return pointer to n characters */
۶  {
۷    if (allocbuf + ALLOCSIZE - allocp >= n)
۸      {     /* it fits */
۹        allocp += n;
۱۰       return allocp - n;^^I/* old p */
۱۱     }
۱۲   else    /* not enough room */
۱۳     return 0;
۱۴ }
```

محاسبات آدرس‌ها

```
۱۵ void
۱۶ afree (char *p)    /* free storage pointed to by p */
۱۷ {
۱۸   if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
۱۹     allocp = p;
۲۰ }
```

محاسبات آدرس‌ها

- در عبارت

```
1 static char *allocp = allocbuf
```

در واقع `allocp` به اولین مکان آزاد در بافر `allocbuf` اشاره می‌کند. می‌توانستیم این عبارت به صورت

```
1 static char  *allocp = &allocbuf[0]
```

نیز بنویسیم، زیرا نام آرایه‌ها درواقع معادل است با آدرس اولین عنصر آنها.

محاسبات آدرس‌ها

- در برنامه قبل توسط

```
1 if(allocbuf + ALLOCSIZE - allocp > = n)
```

بررسی می‌کنیم که فضای کافی برای n کاراکتر بر روی بافر وجود داشته. اگر فضای کافی برروی بافر وجود داشته باشد، مقدار جدید allocp تنظیم خواهد و اشاره‌گری به ابتدای حافظهٔ تخصیص داده شده بازگردانده خواهد شد. در غیر اینصورت باید سیگنالی به فراخوانی کننده بازگردانده شود مبنی بر اینکه فضای خالی در حافظه وجود ندارد. از عدد صفر برای این سیگنال استفاده می‌شود.

- انتساب اعداد به اشاره‌گرها ممکن نیست، اما امکان انتساب عدد صفر به یک اشاره‌گر وجود دارد. اشاره‌گری که مقدار آن برابر با صفر باشد، به جایی در حافظه اشاره نمی‌کند. نماد ثابت NULL برای اشاره‌گر صفر در کتابخانه استاندارد تعریف شده است.

محاسبات آدرس‌ها

- اشاره‌گرها را می‌توان با یکدیگر توسط عملگرهای رابطه‌ای مانند == ، =! ، > ، < ، => یا =< مقایسه کرد.
- برای مثال p < q مقدار درست را بازمی‌گرداند اگر p به مکانی در حافظه قبل از q اشاره کند به عبارت دیگر اگر آدرس حافظه‌ای که p نگه می‌دارد از آدرس حافظه‌ای که q نگه می‌دارد کوچکتر باشد، p < q است.
- جمع یک اشاره‌گر با یک عدد صحیح نیز وجود دارد. عبارت p + n مکان حافظه‌ای نشان می‌دهد که n واحد از مکان حافظهٔ اشاره‌گر p بیشتر باشد. این جمع به نوع متغیر p بستگی دارد. اگر p یک عدد صحیح باشد، هر واحد از n درواقع ۴ بایت است.
- تفریق اشاره‌گرها از یکدیگر نیز امکان پذیر است. برای مثال اگر p و q به دو مکان از حافظه اشاره کنند و داشته باشیم p < q ، آنگاه q - p + 1 تعداد عناصر بین آنها را مشخص می‌کند.

محاسبات آدرس‌ها

- برنامه زیر طول یک رشته را توسط محاسبات برروی اشاره‌گرها به دست می‌آورد.

```
۱ /* strlen: return length of string s */
۲ int
۳ strlen (char *s)
۴ {
۵   char *p = s;
۶   while (*p != '\0')
۷     p++;
۸   return p - s;
۹ }
```

محاسبات آدرس‌ها

- در کتابخانه استاندارد مقدار بازگشتی تابع `strlen` از نوع `size-t` است که یک عدد صحیح بدون علامت است.
- عملیات مجاز بر روی اشاره‌گرها عبارتند از : انتساب اشاره‌گرهای هم نوع به یکدیگر، افزودن یک عدد صحیح به یک اشاره‌گر، کاستن یک عدد صحیح از یک اشاره‌گر، مقایسهٔ دو اشاره‌گر که به یک آرایهٔ واحد اشاره می‌کنند، انتساب عدد صفر به اشاره‌گر و مقایسه یک اشاره‌گر با عدد صفر.
- عملیات دیگر مانند ضرب کردن دو اشاره‌گر مجاز نیست، زیرا این عملیات بی معنی است.

اشاره‌گر نوع کاراکتر

- یک رشته آرایه‌ای است از حروف (کاراکترها). ذخیره‌سازی بیتی رشته‌ها به نحوی است که کاراکتر آخر آنها `'\0'` است. بنابراین برای ذخیرهٔ یک رشته با طول `n` به `n + 1` مکان در حافظه نیاز است.
- یک تابع می‌تواند یک اشاره‌گر نوع کاراکتر دریافت کند. برای مثال تابع `printf` به عنوان پارامتر یک اشاره‌گر نوع کاراکتر دریافت می‌کند.
- یک رشته در حافظه در واقع آرایه‌ای از کاراکترهاست. بنابراین در برنامهٔ زیر یک اشاره‌گر به یک اشاره‌گر دیگر انتساب داده می‌شود.

```
1 char *pmessage;
2 pmessage = "hello" ;
```

اشاره‌گر نوع کاراکتر

- توجه کنید که عبارت قبل رشته را کپی نمی‌کند، بلکه تنها آدرس اشاره‌گر را تغییر می‌دهد.
- یک تفاوت در مقداردهی اولیه بین آرایه‌ها و اشاره‌گرها وجود دارد.
- دو دستور زیر را در نظر بگیرید :

```
1 char amessage[] = "now is the time"; /* an array */
2 char *pmessage = "now is the time"; /* a pointer */
```

- متغیر amessage یک آرایه است و در مقداردهی اولیه اندازهٔ آن توسط رشته داده شده تعیین می‌شود. محتوای این آرایه رشته‌ای است که در مقداردهی اولیه برابر با آن قرارگرفته شده است. اما متغیر pmessage یک اشاره‌گر است. در مقداردهی اولیه، رشته‌ای در مکانی بر روی حافظه قرار می‌گیرد و اشاره‌گر به مکان آن رشته اشاره می‌کند. در طول اجرای برنامه اشاره‌گر ممکن است به مکانی دیگر اشاره کند و رشته را از دست بدهد ولی آرایه نمی‌تواند به مکانی دیگر اشاره کند.

اشاره‌گر نوع کاراکتر

- می‌خواهیم تابعی بنویسیم که دو اشاره‌گر نوع کاراکتر را دریافت کند و محتوای رشته‌ای که توسط اشاره‌گر دوم مشخص شده است را در مکان حافظه‌ای که توسط رشته اول مشخص شده کپی کند.
- این تابع با نام `strcpy` در کتابخانه استاندارد پیاده سازی شده است. تابع `strcpy(s,t)` رشته `t` را در رشته `s` کپی می‌کند.
- توجه کنید که دستور `s = t` تنها آدرس اشاره‌گر `t` را در اشاره‌گر `s` کپی می‌کند.

اشاره‌گر نوع کاراکتر

- تابع کپی رشته به صورت زیر پیاده‌سازی می‌شود.

```
۱ /* strcpy: copy t to s; array subscript version */
۲ void
۳ strcpy (char *s, char *t)
۴ {
۵   int i;
۶   i = 0;
۷   while ((s[i] = t[i]) != '\0')
۸     i++;
۹ }
```

اشاره‌گر نوع کاراکتر

- تابع کپی رشته را می‌توان به صورت زیر نیز پیاده‌سازی کرد.

```
۱  /* strcpy: copy t to s; pointer version */
۲  void
۳  strcpy (char *s, char *t)
۴  {
۵    int i;
۶    i = 0;
۷    while ((*s = *t) != '\0')
۸      {
۹        s++;
۱۰       t++;
۱۱     }
۱۲ }
```

اشاره‌گر نوع کاراکتر

- از آنجایی که فراخوانی در زبان سی با مقدار است، مقدار آرگومان‌ها در پارامترهای تابع کپی می‌شوند، بنابراین تابع می‌تواند پارامترهای `s` و `t` را تغییر دهد بدون اینکه مقدار آرگومان‌های تابع (که اشاره‌گر هستند) تغییر کند. با این حال، چون تابع به مکان‌های حافظه از طریق پارامترهای ورودی دسترسی دارد، می‌تواند محتوای حافظه‌ای که توسط اشاره‌گرها مشخص شده است را تغییر دهد.

اشاره‌گر نوع کاراکتر

- تابع کپی رشته را می‌توان به طور فشرده‌تر نیز به صورت زیر پیاده‌سازی کرد.

```
۱ /* strcpy: copy t to s; pointer version 2 */
۲ void strcpy(char *s, char *t)
۳ {
۴   while ((*s++ = *t++) != '\0');
۵ }
```

اشاره‌گر نوع کاراکتر

- توجه کنید که مقدار '\0' درواقع صفر است و در حلقه نیازی به مقایسه مقدار عبارت شرطی با '\0' نداریم.
- تابع کپی رشته را می‌توان به طور فشرده‌تر نیز به صورت زیر پیاده‌سازی کرد.

```
1 /* strcpy: copy t to s; pointer version 3 */
2 void strcpy(char *s, char *t)
3 {
4   while (*s++ = *t++);
5 }
```

اشاره‌گر نوع کاراکتر

- تابع دیگری که در اینجا بررسی می‌کنیم، دو رشته را دریافت کرده، با یکدیگر مقایسه می‌کند. اگر رشته اول از رشته دوم از نظر لغوی کوچکتر بود یک عدد منفی بازگردانده می‌شود. اگر دو رشته برابر بودند، عدد صفر و اگر رشته اول از رشته دوم بزرگ‌تر بود، یک عدد مثبت بازگردانده می‌شود. مقداری که بازگردانده می‌شود، تفاضل اولین حرفی که در رشته اول در رشته دوم متفاوت است.

اشاره‌گر نوع کاراکتر

- تابع مقایسه دو رشته به صورت زیر نوشته می‌شود.

```
۱  /* strcmp: return <0 if s<t, 0 if s==t, >0 if s>t */
۲  int
۳  strcmp (char *s, char *t)
۴  {
۵    int i;
۶    for (i = 0; s[i] == t[i]; i++)
۷      if (s[i] == '\0')
۸        return 0;
۹    return s[i] - t[i];
۱۰ }
```

اشاره‌گر نوع کاراکتر

- تابع مقایسه دو رشته را می‌توان به صورت زیر با استفاده از عملگرهای اشاره‌گرها نیز پیاده‌سازی کرد.

```
۱ /* strcmp: return <0 if s<t, 0 if s==t, >0 if s>t */
۲ int
۳ strcmp (char *s, char *t)
۴ {
۵   for (; *s == *t; s++, t++)
۶     if (*s == '\0')
۷       return 0;
۸   return *s - *t;
۹ }
```

اشاره‌گر نوع کاراکتر

- عملگرهای ++ و -- هم به صورت پسوندی استفاده می‌شوند و هم به صورت پیشوندی. برای مثال عبارت `val = *p++` مقدار `val` را در مکان حافظهٔ `p` کپی می‌کند و سپس اشاره‌گر را یک واحد به جلو حرکت می‌دهد. عبارت `val = *--p` ابتدا اشاره‌گر را یک واحد به عقب حرکت می‌دهد و سپس مقداری که اشاره‌گر به آن اشاره می‌کند را در `val` ذخیره می‌کند.

اشاره‌گر به اشاره‌گر

- از آنجایی که اشاره‌گرها خود متغیر هستند، می‌توانند مانند بقیهٔ متغیرها در آرایه‌ها نگهداری شوند.
- فرض کنید می‌خواهیم برنامه‌ای بنویسیم که مجموعه‌ای از رشته‌ها را به ترتیب الفبایی مرتب کند.
- در گذشته الگوریتم مرتب‌سازی را برای اعداد صحیح بررسی کردیم.
- برای نگهداری یک مجموعه از رشته‌ها به آرایه‌ای از اشاره‌گرها نیاز داریم. هر عنصر آرایه به یک رشته (یا یک خط) اشاره خواهد کرد. دو رشته را می‌توانیم با استفاده از تابع `stdcmp` مقایسه کنیم. وقتی می‌خواهیم جای دو رشته را در این آرایه عوض کنیم، کافی است اشاره‌گرها را جابجا کنیم. بنابراین در تعویض مکان دو رشته در آرایه، اشاره‌گرهای آنها با یکدیگر تعویض می‌کنیم، به محتوای رشته‌ها.

اشاره‌گر به اشاره‌گر

- می‌خواهیم برنامه‌ای بنویسیم که تعدادی رشته را هر یک در یک خط از ورودی دریافت کند، رشته‌ها را به ترتیب حروف الفبایی مرتب کند و در نهایت رشته‌های مرتب شده را چاپ کند.
- قبل از نوشتن تابع مرتب‌سازی ساختار کلی برنامه را بررسی می‌کنیم.

اشاره‌گر به اشاره‌گر

```
۱ #include <stdio.h>
۲ #include <string.h>
۳ #define MAXLINES 5000^^I^^I/* max #lines to be sorted */
۴ char *lineptr[MAXLINES];^^I/* pointers to text lines */
۵ int readlines (char *lineptr[], int nlines);
۶ void writelines (char *lineptr[], int nlines);
۷ void qsort (char *lineptr[], int left, int right);
```

اشاره‌گر به اشاره‌گر

```
 ۸  /* sort input lines */
 ۹  int main ()
۱۰  {
۱۱    int nlines;      /* number of input lines read */
۱۲    if ((nlines = readlines (lineptr, MAXLINES)) >= 0)
۱۳      {
۱۴        qsort (lineptr, 0, nlines - 1);
۱۵        writelines (lineptr, nlines);
۱۶        return 0;
۱۷      }
۱۸    else
۱۹      {
۲۰        printf ("error: input too big to sort\n");
۲۱        return 1;
۲۲      }
۲۳  }
```

اشاره‌گر به اشاره‌گر

```
۲۴ #define MAXLEN 1000      /* max length of any input line */
۲۵ int getline (char *, int);
۲۶ char *alloc (int);
۲۷ /* readlines: read input lines */
۲۸ int
۲۹ readlines (char *lineptr[], int maxlines)
۳۰ {
۳۱   int len, nlines;
۳۲   char *p, line[MAXLEN];
۳۳   nlines = 0;
۳۴    while ((len = getline (line, MAXLEN)) > 0)
۳۵     if (nlines >= maxlines || p = alloc (len) == NULL)
۳۶     return -1;
```

اشاره‌گر به اشاره‌گر

```
۳۷      else
۳۸        {
۳۹      line[len - 1] = '\0';     /* delete newline */
۴۰      strcpy (p, line);
۴۱      lineptr[nlines++] = p;
۴۲        }
۴۳    return nlines;
۴۴  }
۴۵  /* writelines: write output lines */
۴۶  void
۴۷  writelines (char *lineptr[], int nlines)
۴۸  {
۴۹     int i;
۵۰     for (i = 0; i < nlines; i++)
۵۱     printf ("%s\n", lineptr[i]);
۵۲  }
```

اشاره‌گر به اشاره‌گر

- در این برنامه آرایه به رشته‌ها را به صورت آرایه‌ای از اشاره‌گرها به صورت

```
1 char *lineptr [MAXLINES]
```

تعریف کردیم.

- بنابراین `lineptr` یک آرایه با `MAXLINES` عنصر است که هر عنصر آن یک اشاره‌گر از نوع `char` است. بنابراین `lineptr[i]` یک اشاره‌گر از نوع کاراکتر است و `*lineptr[i]` کاراکتری است که آن عنصر به آن اشاره می‌کند، که در واقع اولین کاراکتر رشته است.

اشاره‌گر به اشاره‌گر

- از آنجایی که `lineptr` نام یک آرایه است، می‌تواند مانند یک اشاره‌گر مورد استفاده قرار بگیرد، بنابراین تابع `writelines` می‌تواند به صورت زیر نوشته شود.

```
۱ /* writelines: write output lines */
۲ void
۳ writelines (char *lineptr[], int nlines)
۴ {
۵   while (nlines-- > 0)
۶     printf ("%s\n", *lineptr++);
۷ }
```

- در ابتدا `lineptr*` به اولین خط اشاره می‌کند. سپس اشاره‌گر به سمت جلو حرکت می‌کند و به عناصر بعدی اشاره می‌کند.

اشاره‌گر به اشاره‌گر

- حال که می‌توانیم ورودی و خروجی را کنترل کنیم، الگوریتم مرتب‌سازی را پیاده سازی کنیم.
- الگوریتم مرتب‌سازی سریع را که برای اعداد صحیح پیاده‌سازی کردیم، کمی تغییر می‌دهیم تا رشته‌ها را دریافت کند. برای مقایسهٔ دو رشته از تابع `strcmp` استفاده می‌کنیم.

اشاره‌گر به اشاره‌گر

- تابع مرتب‌سازی برای رشته‌ها به صورت زیر پیاده‌سازی می‌شود.

```
۱  /* qsort: sort v[left]...v[right] into increasing order */
۲  void
۳  qsort (char *v[], int left, int right)
۴  {
۵    int i, last;
۶    void swap (char *v[], int i, int j);
۷    if (left >= right)    /* do nothing if array contains */
۸      return;        /* fewer than two elements */
۹    swap (v, left, (left + right) / 2);
۱۰   last = left;
۱۱   for (i = left + 1; i <= right; i++)
```

اشاره‌گر به اشاره‌گر

```
۱۲      if (strcmp (v[i], v[left]) < 0)
۱۳        swap (v, ++last, i);
۱۴    swap (v, left, last);
۱۵    qsort (v, left, last - 1);
۱۶    qsort (v, last + 1, right);
۱۷  }
```

اشاره‌گر به اشاره‌گر

- همچنین برای جابجایی دو رشته در آرایه از تابع swap به صورت زیر استفاده می‌کنیم.

```
۱ /* swap: interchange v[i] and v[j] */
۲ void
۳ swap (char *v[], int i, int j)
۴ {
۵   char *temp;
۶   temp = v[i];
۷   v[i] = v[j];
۸   v[j] = temp;
۹ }
```

آرایه‌های چند بعدی

- در زبان سی امکان ایجاد آرایه‌های چند بعدی نیز وجود دارد، اگر آرایه‌های یک بعدی کاربرد بیشتری دارند.
- می‌خواهیم برنامه‌ای بنویسیم که توسط آن تعیین کنیم یک روز از یک ماه چندمین روز از سال است و همچنین یک روز معین از سال چه روزی از چه ماهی است.
- تابع `day-of-year` را برای تبدیل یک روز از یک ماه به یک روز از سال و تابع `month-year` را برای تبدیل یک روز از سال به یک روز از یک ماه تعریف می‌کنیم.
- از آنجایی که تابع دوم دو خروجی دارد، می‌توانیم خروجی‌ها را توسط اشاره‌گرها در ورودی از تابع دریافت کنیم. برای مثال ۶۰ امین روز سال ۱۹۸۸ برابر با ۲۹ فوریه است که آن را با فراخوانی تابع `month-day(1988,60, &m,&d)` به دست می‌آوریم. تابع `m` را برابر با ۲ و `d` را برابر با ۹ قرار خواهد داد.

آرایه‌های چند بعدی

- تعداد روزهای ماه در سال‌های کبیسه متفاوت است از تعداد روزها در سال‌های معمولی، بنابراین از یک جدول با ۲ سطر و ۱۲ ستون استفاده می‌کنیم که سطر اول تعداد روزها در سال معمولی و سطر دوم تعداد روزها در سال کبیسه را تعیین می‌کند. این جدول را به صورت یک آرایهٔ دو بعدی تعریف می‌کنیم.

آرایه‌های چند بعدی

- برنامهٔ تبدیل روز ماه به روز سال به صورت زیر نوشته می‌شود.

```
۱  static char daytab[2][13] = {
۲    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
۳    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
۴  };
۵  /* day_of_year: set day of year from month & day */
۶  int
۷  day_of_year (int year, int month, int day)
۸  {
۹    int i, leap;
۱۰   leap = year % 4 == 0 && year % 100 != 0 || year % 400 == 0;
۱۱   for (i = 1; i < month; i++)
۱۲     day += daytab[leap][i];
۱۳   return day;
۱۴ }
```

آرایه‌های چند بعدی

- برنامهٔ تبدیل روز سال به روز ماه به صورت زیر نوشته می‌شود.

```
۱  /* month_day: set month, day from day of year */
۲  void
۳  month_day (int year, int yearday, int *pmonth, int *pday)
۴  {
۵    int i, leap;
۶    leap = year % 4 == 0 && year % 100 != 0 || year % 400 == 0;
۷    for (i = 1; yearday > daytab[leap][i]; i++)
۸      yearday -= daytab[leap][i];
۹    *pmonth = i;
۱۰   *pday = yearday;
۱۱ }
```

آرایه‌های چند بعدی

- دقت کنید که مقدار منطقی به دست آمده در متغیر `leap` یا صفر است و یا یک. بنابراین این مقدار را می‌توان به عنوان اندیس آرایهٔ دو بعدی `daytab` استفاده کرد.
- برای تعریف این آرایه از نوع `char` استفاده کردیم، زیر می‌خواهیم مقادیر کوچک را نگهداری کنیم که یک بایت برای آنها کافی است.
- یک آرایهٔ دو بعدی در واقع یک آرایه از آرایه‌ها است و بنابراین برای دسترسی به هر عنصر آن می‌نویسیم `daytab[i][j]`.
- در حافظه، مقادیر یک آرایهٔ دو بعدی سطر به سطر ذخیره می‌شوند، بنابراین دو مقدار `daytab[i][j]` و `daytab[i][j+1]` در کنار یکدیگر در حافظه قرار می‌گیرند.

آرایه‌های چند بعدی

- برای مقداردهی اولیه یک آرایه می‌توانیم از علامت آکولاد استفاده کنیم. مقادیری که در آکولاد قرار می‌گیرند به ترتیب عناصر آرایه را تشکیل می‌دهند. همچنین در یک آرایهٔ دو بعدی هر سطر درون آکولاد و همهٔ ستون‌ها در یک آکولاد بیرونی قرار می‌گیرند.
- در این برنامه اولین ستون جدول `daytab` را برابر با صفر قرار دادیم تا بتوانیم ماه‌ها را از ۱ تا ۱۲ شماره‌گذاری کنیم به جای ۰ تا ۱۱.
- وقتی می‌خواهیم یک آرایهٔ دو بعدی را به یک تابع ارسال کنیم، باید تعداد ستون‌های آرایه‌ها را در امضای تابع مشخص کنیم، زیرا آنچه به تابع ارسال می‌شود، اشاره‌گری است از آرایه‌ها و برای خواندن صحیح مقادیر از حافظه باید اندازهٔ آرایه‌ها مشخص باشد.
- بنابراین تابعی که آرایهٔ دو بعدی `daytab` را دریافت می‌کند می‌تواند به صورت `if(daytab[2][13])` یا `if(int daytab[][13])` یا `if(int (*daytab)[13])` تعریف شود.

آرایه‌های چند بعدی

- در تعریف آخر درواقع می‌گوییم پارامتر تابع اشاره‌گری است از آرایه‌های ۱۳ عنصری. پرانتزها ضروری هستند، زیرا اولویت براکت [] بیشتر از اولویت عملگر ستاره * است.
- درواقع `int *daytab[13]` یک آرایهٔ ۱۳ عنصری است از اشاره‌گرها به اعداد صحیح.
- در حالت کلی برای یک آرایهٔ چند بعدی تنها بعد اول را می‌توان بدون تعداد عناصر قید کرد و تعداد عناصر ابعاد دیگر باید دقیقا در تعاریف مشخص شوند.

مقداردهی اولیهٔ آرایه از اشاره‌گرها

- فرض کنید می‌خواهیم تابعی بنویسیم که با دریافت عدد یک ماه، اشاره‌گری به نام آن ماه بازگرداند. نام ماه‌ها را می‌توانیم در یک آرایه از رشته‌ها (اشاره‌گرهای کاراکتری) نگهداری کنیم و این آرایه می‌تواند در درون تابع قرار بگیرد، اما چون مقادیر آن آرایه باید توسط دیگر تابع در دسترس باشد، آرایه را به صورت ایستا تعریف می‌کنیم.

مقداردهی اولیهٔ آرایه از اشاره‌گرها

- این تابع را می‌توانیم به صورت زیر تعریف کنیم.

```
/* month_name: return name of n-th month */
char *
month_name (int n)
{
  static char *name[] = {
    "Illegal month",
    "January", "February", "March",
    "April", "May", "June",
    "July", "August", "September",
    "October", "November", "December"
  };
  return (n < 1 || n > 12) ? name[0] : name[n];
}
```

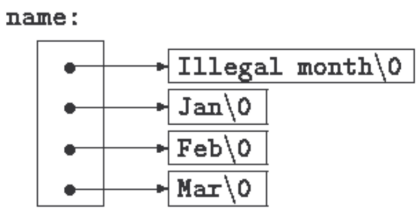
اشاره‌گرها و آرایه‌های چند بعدی

- یک آرایهٔ دو بعدی را به صورت `int a[10][20]` تعریف می‌کنیم و یک آرایه به اشاره‌گر را به صورت `int *b[10]` . در اینصورت `a[3][4]` و `b[3][4]` هر دوبه یک `int` اشاره می‌کنند، اما `a` یک آرایهٔ دو بعدی واقعی است که دقیقاً در آن ۲۰۰ مکان حافظه برای مقادیر صحیح در نظر گرفته شده است و برای به دست آوردن خانهٔ حافظه‌ای که در آن سطر `row` و ستون `col` قرار می‌گیرد. می‌توان از عبارت `20 * row + col` استفاده کرد.
- اما متغیر `b` تنها یک آرایهٔ ۱۰ عنصری از اشاره‌گرها را تعریف می‌کند. فرض کنید هر عنصر `b` به یک آرایهٔ ۲۰ عنصری اشاره کند در این صورت ۲۰۰ مکان حافظه برای نگهداری اعداد صحیح داریم و به علاوه ۱۰ مکان حافظه برای نگهداری آدرس اشاره‌گرهای هر سطر.
- روش دوم فضای بیشتری اشغال می‌کند اما مزیت آن این است که هر سطر از آرایهٔ دو بعدی می‌تواند طول متفاوتی از سطرهای دیگر داشته باشد.

اشاره‌گرها و آرایه‌های چند بعدی

- برای مثال آرایه‌ای از رشته‌ها را در نظر بگیرید. هر عنصری در این آرایه در واقع اشاره‌گری است که به رشته‌ها با طول‌های متفاوت اشاره می‌کند.

```
1 char * name[] = {"Illegal month" , "Jan" , "Feb" , "Mar"};
```



اشاره‌گرها و آرایه‌های چند بعدی

- اما آرایهٔ دو بعدی در حافظه به صورت زیر است.

```
1  char aname[][15] = {"Illegal month" , "Jan" , "Feb" , "Mar"};
```

aname:

Illegal month\0	Jan\0	Feb\0	Mar\0
0	15	30	45

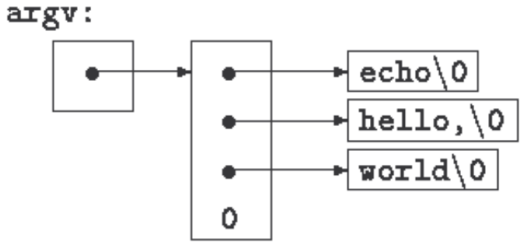
آرگومان‌های ورودی برنامه

- در محیطی که برنامهٔ سی در آن اجرا می‌شود، امکان دریافت آرگومان توسط برنامه وجود دارد. بدین ترتیب با اجرای یک برنامه تعدادی ورودی به برنامه ارسال می‌شوند. این ورودی‌ها توسط برنامه دریافت شده، در برنامه استفاده می‌شوند.
- وقتی تابع `main` فراخوانی می‌شود، معمولا با دو آرگومان فراخوانی انجام می‌شود. آرگومان اول تعداد آرگومان‌ها را مشخص می‌کند و `argc` (تعداد آرگومان) [1] نامیده می‌شود و آرگومان دوم اشاره‌گری است به یک آرایه از رشته‌های کاراکتری و شامل مقدار همهٔ آرگومان است و `argv` (وکتور آرگومان‌ها) [2] نامیده می‌شود.
- برای مثال برنامه `echo` را در نظر بگیرید. این برنامه یک ورودی می‌گیرد و مقدار ورودی را چاپ می‌کند. برای مثال `world` ، `echo hello` رشته `world` ، `hello` را چاپ می‌کند.

[1] argument count
[2] argument vector

آرگومان‌های ورودی برنامه

- مقدار argv[0] نام برنامه است، بنابراین مقدار argc حداقل ۱ است. اگر مقدار argc برابر با ۱ باشد، درواقع هیچ آرگومانی به برنامه ارسال نشده است. در مثال echo درواقع argc برابر با ۳ است. مقادیر argv[0] ، argv[1] و argv[2] به ترتیب "echo" ، "hello" و "world" می‌باشد. بنابراین مقدار اولین آرگومان در argv[1] و آخرین آرگومان در argv[argc-1] قرار می‌گیرد.



آرگومان‌های ورودی برنامه

- برنامهٔ echo به صورت زیر نوشته می‌شود.

```
۱  #include <stdio.h>
۲  /* echo command-line arguments; 1st version */
۳  int main (int argc, char *argv[])
۴  {
۵    int i;
۶    for (i = 1; i < argc; i++)
۷      printf ("%s%s", argv[i], (i < argc - 1) ? " " : "");
۸    printf ("\n");
۹    return 0;
۱۰ }
```

آرگومان‌های ورودی برنامه

- از آنجایی که argv یک اشاره‌گر به آرایه‌ای از اشاره‌گرهاست می‌توانیم به جای استفاده از آن به صورت آرایه، از آن به صورت اشاره‌گر استفاده کنیم.

```
۱ /* echo command-line arguments; 2nd version */
۲ int main (int argc, char *argv[])
۳ {
۴   while (--argc > 0)
۵     printf ("%s%s", *++argv, (argc > 1) ? " " : "");
۶   printf ("\n");
۷   return 0;
۸ }
```

آرگومان‌های ورودی برنامه

- با هربار افزایش اشاره‌گر argv اشاره‌گر به مکان حافظه بعدی اشاره می‌کند و argv* یک آرگومان اشاره می‌کند.
- عبارت چاپ رشته را می‌توانیم به صورت زیر بنویسیم.

```
printf ((argc > 1) ? "%s " : "%s" , *++ argv)
```

- حال می‌خواهیم برنامه‌ای بنویسیم که ورودی‌های دریافت شده از آرگومان‌های برنامه را در یک متن که از ورودی استاندارد دریافت می‌شود، جستجو کند.

آرگومان‌های ورودی برنامه

- این برنامه جستجو که در یونیکس و لینوکس grep نامیده می‌شود به صورت زیر است.

```
۱  #include <stdio.h>
۲  #include <string.h>
۳  #define MAXLINE 1000
۴  int getline (char *line, int max);
۵  /* find: print lines that match pattern from 1st arg */
۶  int main (int argc, char *argv[])
۷  {
۸    char line[MAXLINE];
۹    int found = 0;
۱۰   if (argc != 2)
۱۱       printf ("Usage: find pattern\n");
۱۲   else
```

آرگومان‌های ورودی برنامه

```
۱۳      while (getline (line, MAXLINE) > 0)
۱۴      if (strstr (line, argv[1]) != NULL)
۱۵     {
۱۶     printf ("%s", line);
۱۷     found++;
۱۸     }
۱۹     return found;
۲۰  }
```

آرگومان‌های ورودی برنامه

- تابع `strstr` یک اشاره‌گر به اولین وقوع رشته `t` در رشته `s` بازمی‌گرداند و در صورتی که رشته پیدا نشود مقدار `NULL` را بازمی‌گرداند.
- حال فرض کنید می‌خواهیم این برنامه را تعمیم دهیم به طوری‌که در برخی مواقع همهٔ خطوط دریافت شده از کاربر را به جز خطوطی که شامل یک عبارت معین هستند را چاپ کند و در برخی مواقع شماره خطوط دریافت شده توسط کاربر را قبل از خط مورد نظر چاپ کند. در سیستم عامل یونیکس برای مشخص کردن آرگومان‌های ورودی، از یک حرف و یک خط تیره استفاده می‌شود. برای مثال می‌توانیم از آرگومان `x-` برای نشان دادن انتخاب به جز `(except)` استفاده کنیم و از آرگومان `n-` برای نشان دادن انتخاب شماره خط `(number)`.
- بنابراین `find -x -n pattern` بدین معناست که می‌خواهیم تمام خطوطی را که حاوی الگوی `pattern` نیستند را با شماره خط آنها چاپ کنیم. همچنین ممکن است بخواهیم این دستور را به صورت `find -nx pattern` وارد کنیم.

آرگومان‌های ورودی برنامه

- این برنامهٔ جستجو به صورت زیر نوشته می‌شود.

```
۱  #include <stdio.h>
۲  #include <string.h>
۳  #define MAXLINE 1000
۴  int getline (char *line, int max);
۵  /* find: print lines that match pattern from 1st arg */
۶  int main (int argc, char *argv[])
۷  {
۸    char line[MAXLINE];
۹    long lineno = 0;
۱۰   int c, except = 0, number = 0, found = 0;
۱۱   while (--argc > 0 && (*++argv)[0] == '-')
```

آرگومان‌های ورودی برنامه

```
۱۲        while (c = *++argv[0])
۱۳            switch (c)
۱۴                {
۱۵                case 'x':
۱۶                     except = 1;
۱۷                     break;
۱۸                case 'n':
۱۹                     number = 1;
۲۰                     break;
۲۱                default:
۲۲                     printf ("find: illegal option %c\n", c);
۲۳                     argc = 0;
۲۴                     found = -1;
۲۵                     break;
۲۶                  }
```

آرگومان‌های ورودی برنامه

```
۲۷    if (argc != 1)
۲۸      printf ("Usage: find -x -n pattern\n");
۲۹    else
۳۰       while (getline (line, MAXLINE) > 0)
۳۱          {
۳۲               lineno++;
۳۳               if ((strstr (line, *argv) != NULL) != except)
۳۴                   {
۳۵                          if (number)
۳۶                                 printf ("%ld:", lineno);
۳۷                          printf ("%s", line);
۳۸                          found++;
۳۹                   }
۴۰       }
۴۱     return found;
۴۲  }
```

آرگومان‌های ورودی برنامه

- دقت کنید که در این برنامه `argv++*` اشاره‌گری است به یک رشته، بنابراین `[0] (argv++*)` اولین کاراکتر این رشته است. می‌توانستیم از `argv++**` نیز برای دریافت اولین کاراکتر استفاده کنیم، ولی روش اول خواناتر است.
- پرانتزگذاری در عبارت مذکور ضروری است زیرا عبارت `argv[0]++*` معادل است با `(argv[0])++*` که کاراکتری را که در اندیس ۱ قرار دارد را باز می‌گرداند.

اشاره‌گر به توابع

- در زبان سی گرچه نام توابع را نمی‌توان به عنوان متغیر استفاده کرد، ولی می‌توان اشاره‌گر به توابع تعریف کرد. چنین اشاره‌گرهایی را می‌توان به یکدیگر انتساب کرد یا در آرایه قرار داد و یا به تابع به عنوان آرگومان ارسال کرد و یا از تابع به عنوان خروجی بازگرداند.

- فرض کنید می‌خواهیم برنامه‌ای بنویسیم که در آن با دریافت آرگومان n- جستجو را بر روی اعداد به جای رشته‌ها انجام دهد.

- یک الگوریتم مرتب‌سازی از چند بخش تشکیل شده است. یک الگوریتم مرتب‌سازی شامل یک تابع مقایسه‌گر است که دو عنصر از یک آرایه را با یکدیگر مقایسه می‌کند (برای مثال رشته‌ها به صورت الفبایی مقایسه می‌شوند و اعداد صحیح یه صورت عددی). یک الگوریتم مرتب‌سازی همچنین تعیین می‌کند که عناصر به صورت صعودی مقایسه شوند یا به صورت نزولی. همچنین یک الگوریتم مرتب‌سازی روش مرتب کردن عناصر را مشخص می‌کند.

اشاره‌گر به توابع

- نوع مقایسهٔ دو عنصر و همچنین صعودی یا نزولی بودن الگوریتم مرتب‌سازی می‌تواند به عنوان آرگومان به تابع مرتب‌سازی ارسال شود.
- مقایسه الفبایی دو رشته نوع `strcmp` انجام می‌شود. می‌توانیم تابعی به نام `numcmp` بنویسیم که شبیه به `strcmp` عمل کند ولی برای مقایسهٔ دو مقدار عددی به کار می‌رود. بدین ترتیب اگر اشاره‌گری به تابع `strcmp` به تابع مرتب‌سازی `qsort` ارسال شود، مقادیر به صورت الفبایی مقایسه می‌شوند و در صورتی که `numcmp` به تابع `qsort` ارسال شود، مقادیر به صورت عددی مرتب می‌شوند.

اشاره‌گر به توابع

- بنابراین مرتب‌سازی در حالت کلی برای رشته‌ها و اعداد به صورت زیر انجام می‌شود.

```
۱  #include <stdio.h>
۲  #include <string.h>
۳  #define MAXLINES 5000^^I^^I/* max #lines to be sorted */
۴  char *lineptr[MAXLINES];^^I/* pointers to text lines */
۵  int readlines (char *lineptr[], int nlines);
۶  void writelines (char *lineptr[], int nlines);
۷  void qsort (void *lineptr[], int left, int right,
۸  int (*comp) (void *, void *));
۹  int numcmp (char *, char *);
۱۰ /* sort input lines */
۱۱ int main (int argc, char *argv[])
۱۲ {
۱۳   int nlines;      /* number of input lines read */
۱۴   int numeric = 0;    /* 1 if numeric sort */
```

اشاره‌گر به توابع

```
16    if (argc > 1 && strcmp (argv[1], "-n") == 0)
17      numeric = 1;
18    if ((nlines = readlines (lineptr, MAXLINES)) >= 0)
19      {
20        qsort ((void **) lineptr, 0, nlines - 1,
21         (int (*)(void *, void *)) (numeric ? numcmp : strcmp));
22        writelines (lineptr, nlines);
23        return 0;
24      }
25    else
26      {
27        printf ("input too big to sort\n");
28        return 1;
29      }
30  }
```

اشاره‌گر به توابع

- بنابراین به یک تابع `qsort` نیاز داریم که هر نوع آرایه‌ای را بتوان پردازش کند و نه فقط رشته‌ها. در این حالت تابع `qsort` یک آرایه از اشاره‌گرها را دریافت می‌کند که این اشاره‌گرها می‌توانند به اعداد و یا رشته‌ها اشاره‌کنند. برای این که این تابع در حالت کلی عمل کند از اشاره‌گر `*void` استفاده می‌کنیم که یک اشاره‌گر بدون نوع است. هر نوع اشاره‌گری را می‌توان به اشاره‌گر نوع `*void` تبدیل کرد و همچنین اشاره‌گر نوع `*void` را می‌توان به هر نوع اشاره‌گری تبدیل کرد.

اشاره‌گر به توابع

- تابع مرتب‌سازی qsort در حالت کلی به صورت زیر نوشته می‌شود.

```
۱  /* qsort: sort v[left]...v[right] into increasing order */
۲  void
۳  qsort (void *v[], int left, int right, int (*comp) (void *, void *))
۴  {
۵    int i, last;
۶    void swap (void *v[], int, int);
۷    if (left >= right)   /* do nothing if array contains */
۸      return;      /* fewer than two elements */
۹    swap (v, left, (left + right) / 2);
۱۰   last = left;
۱۱   for (i = left + 1; i <= right; i++)
۱۲     if ((*comp) (v[i], v[left]) < 0)
۱۳       swap (v, ++last, i);
```

اشاره‌گر به توابع

```
14    swap (v, left, last);
15    qsort (v, left, last - 1, comp);
16    qsort (v, last + 1, right, comp);
17  }
```

اشاره‌گر به توابع

- چهارمین پارامتر تابع `qsort` یک اشاره‌گر به تابع به صورت `int (*comp)(void* , void*)` است که به تابعی اشاره می‌کند که دو ورودی از نوع `*void` و یک خروجی از نوع `int` دارد. هر دو تابع `strcmp` و `numcmp` چنین خروجی و ورودی‌هایی دارند و هر دو می‌توانند به تابع `qsort` ارسال شوند.
- مجموعهٔ ورودی‌ها و خروجی یک تابع امضای تابع نامیده می‌شود. در ارسال یک تابع به عنوان اشاره‌گر به یک تابع دیگر، امضای تابع ارسال شده با امضای اشاره‌گر به تابع در ورودی تابع دیگر باید یکسان باشند.
- از آنجایی که در مثال قبل `comp` یک اشاره‌گر است، پس به خود تابع توسط `comp*` دسترسی پیدا می‌کنیم و بنابراین می‌نویسیم `(*comp)(v[i] , v[left])`

اشاره‌گر به توابع

- تابع مقایسهٔ دو مقدار عددی به صورت زیر تعریف می‌شود.

```
۱  #include <stdlib.h>
۲  /* numcmp: compare s1 and s2 numerically */
۳  int
۴  numcmp (char *s1, char *s2)
۵  {
۶    double v1, v2;
۷    v1 = atof (s1);
۸    v2 = atof (s2);
۹    if (v1 < v2)
۱۰     return -1;
۱۱   else if (v1 > v2)
۱۲     return 1;
۱۳   else
۱۴     return 0;
۱۵ }
```

اشاره‌گر به توابع

- تابع جابجایی دو اشاره‌گر در حالت کلی به صورت زیر خواهد بود.

```
۱ void swap (void *v[],
۲                 {
۳                    void *temp; int i, int j;
۴         )temp = v[i];
۵   v[i] = v[j];
۶   v[j] = temp;
۷ }
```

ساختمان‌ها

ساختمان‌ها

- یک ساختمان [1] مجموعه‌ای است از یک یا چند متغیر که می‌توانند از چندین نوع متفاوت باشند. این مجموعه از متغیرها تحت عنوان یک نام تعریف می‌شوند. در زبان‌های دیگر ساختمان یک رکورد [2] نیز نامیده می‌شود.

- ساختمان‌ها کمک می‌کنند داده‌های پیچیده در برنامه‌های بزرگ سازمان‌دهی شوند، زیرا توسط ساختمان‌ها می‌توانیم به یک دسته از متغیرهای مرتبط با یکدیگر با یک عنوان واحد دسترسی پیدا کنیم.

- برای مثال یک دانشجو دارای ویژگی‌هایی از جمله نام و نام‌خانوادگی، شماره‌ملی، آدرس و شمارهٔدانشجویی است. آدرس خود می‌توانند دارای قسمت‌های مختلف از جمله شهر، خیابان و پلاک باشد. بنابراین برای نگهداری اطلاعات یک دانشجو به یک ساختار پیچیده نیاز داریم که همهٔ اطلاعات ذکر شده را در بر می‌گیرد. به عنوان مثال دیگر، در یک برنامهٔ رسم کامپیوتری، یک مستطیل شامل اطلاعات چهار نقطه است و هر نقطه شامل یک مکان در راستای محور افقی و یک مکان در راستای محور عمودی است.

---

[1] structure

[2] record

ساختمان‌ها

- یک نقطه که دارای دو مختصات در راستای محور افقی و یک مختصات در راستای محور عمودی است را به صورت زیر تعریف می‌کنیم.

```
۱ struct point {
۲         int x;
۳         int y;
۴ };
```

- کلمهٔ کلیدی `struct` برای تعریف یک ساختمان به کار می‌رود. در درون بلوک `struct` مجموعه‌ای از متغیرهای متعلق به ساختمان قرار می‌گیرند. متغیرهایی که در ساختمان تعریف می‌شوند، اعضای ساختمان نام دارند. به دنبال کلیدی کلیدی `struct` نام یا برچسب [1] ساختمان ذکر می‌شود.

- نام یک متغیر در یک برنامه و نام یک متغیر در یک ساختمان می‌توانند یکسان باشند، زیرا حوزهٔ تعریف آنها متفاوت است.

[1] tag

ساختمان‌ها

- توسط کلید واژهٔ `struct` درواقع یک نوع داده تعریف می‌کنیم. بنابراین از این نوع داده می‌توانیم مشابه نوع‌های اصلی، متغیر کنیم.
- در مثال زیر، از یک ساختمان سه متغیر تعریف شده است.

```
۱ struct { ... } x,y,z ;
```

- توجه کنید که در این مثال نام یا برچسب ساختمان ذکر نشده است، زیرا قید کردن نام ساختمان اختیاری است.
- اگر یک ساختمان دارای نام باشد، می‌توانیم پس از تعریف ساختمان از آن متغیر بسازیم. برای مثال یک متغیر از نوع نقطه به صورت زیر تعریف می‌شود.

```
۱ struct point pt ;
```

ساختمان‌ها

- یک متغیر را می‌توانیم با یک لیست مقداردهی اولیه نیز مقداردهی کنیم. برای مثال :

```
struct point maxpt = {320 , 200} ;
```

- یک متغیر از نوع ساختمان را می‌توانیم به یک تابع ارسال کنیم یا از یک تابع بازگردانیم. همچنین توسط عملگر نقطه (.) می‌توانیم از طریق نام متغیر به اعضای آن دسترسی پیدا کنیم.
- برای مثال اعضای ساختمان point را می‌توانیم به صورت زیر چاپ کنیم.

```
printf("%d , %d" , pt.x , pt.y) ;
```

- فاصلهٔ بین یک متغیر از نوع نقطه از مبدأ مختصات (0,0) را می‌توانیم به صورت زیر محاسبه کنیم.

```
double dist, sqrt (double);
dist = sqrt ((double) pt.x * pt.x + (double) pt.y * pt.y);
```

ساختمان‌ها

- حال فرض کنید می‌خواهیم یک مستطیل را توسط ویژگی‌های آن تعریف کنیم. برای تعریف یک مستطیل که طول و عرض آن و محورهای مختصات موازی است، می‌توانیم از دو نقطه استفاده کنیم که مختصات نقطه جنوب غربی و نقطه شمال شرقی مستطیل را تعیین می‌کنند.
- ساختمان چنین مستطیلی به صورت زیر تعریف می‌شود.

```
۱ struct rect {
۲           struct point pt1;
۳           struct point pt2;
۴ };
```

- سپس می‌توانیم یک متغیر از نوع مستطیل به صورت زیر تعریف کنیم.

```
۱ struct rect screen ;
```

- به مختصات نقطه جنوب غربی در راستای افقی می‌توانیم توسط `screen pt1.x` دسترسی پیدا کنیم.

ساختمان‌ها و توابع

- ساختمان‌ها را می‌توانیم به توابع ارسال کنیم و همچنین توسط توابع می‌توانیم یک متغیر از نوع ساختمان بازگردانیم.
- فرض کنید می‌خواهیم تابعی بنویسیم که دو عدد صحیح را به عنوان مختصات یک نقطه دریافت کند و یک متغیر از نوع نقطه بازگرداند. این تابع به صورت زیر نوشته می‌شود.

```
۱ /* makepoint: make a point from x and y components */
۲ struct point
۳ makepoint (int x, int y)
۴ {
۵   struct point temp;
۶   temp.x = x;
۷   temp.y = y;
۸   return temp;
۹ }
```

ساختمان‌ها و توابع

- از تابع makepoint می‌توانیم به صورت زیر برای ساخت یک نقطه استفاده کنیم.

```
۱ struct rect screen;
۲ struct point middle;
۳ struct point makepoint(int, int);
۴ screen.pt1 = makepoint(0,0);
۵ screen.pt2 = makepoint(XMAX, YMAX);
۶ middle = makepoint((screen.pt1.x + screen.pt2.x)/2,
۷                                 (screen.pt1.y + screen.pt2.y)/2);
```

ساختمان‌ها و توابع

- تابع زیر مختصات نقطهٔ دوم را به نقطه اول اضافه می‌کند و نقطه به دست آمده را بازمی‌گرداند.

```
۱ /* addpoints: add two points */
۲ struct addpoint (struct point p1, struct point p2)
۳ {
۴   p1.x += p2.x;
۵   p1.y += p2.y;
۶   return p1;
۷ }
```

ساختمان‌ها و توابع

- در این مثال ورودی و خروجی تابع هر دو از نوع ساختمان هستند. توجه کنید که در این مثال (به جای تعریف یک متغیر موقت) مقدار p1 را افزایش دادیم. این کار بدین دلیل است که متغیرها در زبان سی با مقدار به توابع ارسال می‌شوند و بنابراین تغییر مقدار پارامتر p1 در مقدار آرگومان ارسال شده به تابع تأثیری نمی‌گذارد.

- فرض کنید می‌خواهیم برنامه‌ای بنویسیم که بررسی کند آیا یک نقطه در یک مستطیل قرار دارد یا خیر. این برنامه به صورت زیر نوشته می‌شود.

```
۱ /* ptinrect: return 1 if p in r, 0 if not */
۲ int
۳ ptinrect (struct point p, struct rect r)
۴ {
۵       return p.x >= r.pt1.x && p.x < r.pt2.x
۶       && p.y >= r.pt1.y && p.y < r.pt2.y;
۷ }
```

- در مثال قبل فرض کردیم مستطیل در فرم استاندارد ذخیره شده است، بدین معنی که مختصات `pt1` کوچکتر از مختصات `pt2` هستند.

ساختمان‌ها و توابع

- می‌توانیم تابعی به صورت زیر بنویسیم که یک مستطیل را دریافت کرده، آن را به فرم استاندارد تبدیل کند. با دریافت یک مستطیل در فرم غیر استاندارد این تابع یک مستطیل در فرم استاندارد باز می‌گرداند.

```
۱  #define min(a, b) ((a) < (b) ? (a) : (b))
۲  #define max(a, b) ((a) > (b) ? (a) : (b))
۳  /* canonrect: canonicalize coordinates of rectangle */
۴  struct rect
۵  canonrect (struct rect r)
۶  {
۷    struct rect temp;
۸    temp.pt1.x = min (r.pt1.x, r.pt2.x);
۹    temp.pt1.y = min (r.pt1.y, r.pt2.y);
۱۰   temp.pt2.x = max (r.pt1.x, r.pt2.x);
۱۱   temp.pt2.y = max (r.pt1.y, r.pt2.y);
۱۲   return temp;
۱۳ }
```

ساختمان‌ها و توابع

- وقتی می‌خواهیم یک ساختمان بزرگ را به یک تابع ارسال کنیم بهتر است که آن را توسط اشاره‌گر ارسال کنیم یا به عبارت دیگر فراخوانی را با ارجاع انجام دهیم. اشاره‌گر به ساختمان شبیه به اشاره‌گر به نوع‌های داده‌ای اصلی تعریف می‌شوند.
- برای مثال `struct point *pp;` یک اشاره‌گر از نوع ساختمان `point` تعریف می‌کند. درواقع اگر `pp` به یک ساختمان نقطه اشاره کند، `pp*` مقدار متغیر است و `(*pp).x` و `(*pp).y` مختصات آن در راستای افقی و عمودی هستند.
- در مثال زیر از اشاره‌گری به یک ساختمان استفاده می‌کنیم.

```
۱ struct point origin, *pp;
۲ pp = &origin;
۳ printf("origin is (%d,%d)\n", (*pp).x, (*pp).y);
```

ساختمان‌ها و توابع

- در این مثال نیاز به پرانتزگذاری وجود دارد، زیرا اولویت عملگر (.) بالاتر از عملگر (*) است. عبارت `*pp.x` معادل است با `(*pp).x` که در اینجا غیر قانونی است زیرا `x` اشاره‌گر نیست.
- از آنجایی که اشاره‌گر به ساختمان‌ها بسیار مورد استفاده است، برای دسترسی که اعضای اشاره‌گری که از نوع ساختمان است، یک عملگر خاص تعریف شده است. عملگر `->` یا عملگر فلش برای دسترسی به اعضای اشاره‌گرها به ساختمان‌ها استفاده می‌شود.
- برای مثال

```
1 printf("origin is (%d, %d) \n" , pp -> x , pp -> y);
```

- اگر داشته باشیم `struct rect r, *rp = &r;` آنگاه چهار عبارت زیر معادل یکدیگرند.

$$r.pt1.x \equiv rp \text{ -> } pt1.x \equiv (r.pt1).x \equiv (rp \text{ -> } pt1).x$$

ساختمان‌ها و توابع

- عملگرهای . ، <- ، () و [] بالاترین اولویت‌ها را در میان عملگرها دارند.
- فرض کنید ساختمانی به صورت زیر به همراه یک اشاره‌گر به آن تعریف کنیم.

```
۱ struct {
۲       int len;
۳       char *str;
۴ }*p ;
```

- در اینصورت p++ -> len مقدار len را می‌افزاید، نه مقدار اشاره‌گر p . این عبارت در واقع معادل است با ++(p -> len)
- اگر بخواهیم ابتدا اشاره‌گر p را بیافزاییم و سپس ویژگی len را از آن بازگردانیم، عبارت را باید به صورت (++p) -> len بنویسیم. در صورتی که بخواهیم ابتدا مقدار len را بازگردانیم و سپس اشاره‌گر را حرکت دهیم، باید عبارت را به صورت (p++) -> len بنویسیم.

آرایه از ساختمان‌ها

- فرض کنید می‌خواهیم برنامه‌ای بنویسیم که تعداد وقوع کلمات کلیدی زبان سی را در یک متن بشمارد. یک آرایه از متغیرها از نوع
برای نگهداری نام کلمات کلیدی و یک آرایه از اعداد صحیح برای نگهداری تعداد وقوع هر یک از کلمات کلیدی نیاز داریم.
- برای مثال می‌توانیم به صورت زیر تعریف کنیم.

```
1 char* keyword [NKEYS];
2 int keycount [NKEYS];
```

آرایه از ساختمان‌ها

- اما از آنجایی که این دو متغیر با یکدیگر به کار می‌روند، بهتر است طراحی متغیرها را به طوری سازماندهی کنیم که ارتباط بین این دو متغیر مشخص باشد که از خطای برنامه نویسی نیز کاسته خواهد شد.
- این دو متغیر را می‌توانیم به صورت زیر سازماندهی کنیم.

```
1 struct key {
2         char* word;
3         int count;
4 };
5 struct key keytab [NKEYS]
```

آرایه از ساختمان‌ها

- از آنجایی که این آرایه تعداد معینی رشته را در برمی‌گیرد، می‌توانیم آن را در ابتدای برنامه به صورت زیر مقداردهی اولیه کنیم.

```
۱  struct key
۲  {
۳    char *word;
۴    int count;
۵  } keytab[] = {
۶    "auto", 0,
۷    "break", 0,
۸    "case", 0,
۹    "char", 0,
۱۰   "const", 0,
۱۱   "continue", 0,
۱۲   "default", 0,
```

آرایه از ساختمان‌ها

```
    /* ... */
  "unsigned", 0,
  "void", 0,
  "volatile", 0,
  "while", 0
};
```

آرایه از ساختمان‌ها

- البته بهتر است از آکولاد گذاری برای مشخص شدن مرز رکوردها استفاده کنیم.

```
{{"auto" , 0} , {"break" , 0} , ...}
```

آرایه از ساختمان‌ها

- حال به مسئله شمارش کلمات کلیدی باز می‌گردیم. به تابعی نیاز داریم که کلمات را از ورودی یک‌به‌یک دریافت کند. سپس هر کلمه در آرایهٔ کلمات کلیدی پیدا می‌شود و شمارندهٔ آن یک واحد افزایش پیدا کند.
- برنامهٔ شمارش کلمات کلیدی به صورت زیر نوشته می‌شود.

```
۱ #include <stdio.h>
۲ #include <ctype.h>
۳ #include <string.h>
۴ #define MAXWORD 100
۵ int getword (char *, int);
۶ int binsearch (char *, struct key *, int);
۷ /* count C keywords */
۸ int main ()
```

آرایه از ساختمان‌ها

```
۱۰  {
۱۱    int n;
۱۲    char word[MAXWORD];
۱۳    while (getword (word, MAXWORD) != EOF)
۱۴      if (isalpha (word[0]))
۱۵        if ((n = binsearch (word, keytab, NKEYS)) >= 0)
۱۶        keytab[n].count++;
۱۷    for (n = 0; n < NKEYS; n++)
۱۸      if (keytab[n].count > 0)
۱۹        printf ("%4d %s\n", keytab[n].count, keytab[n].word);
۲۰    return 0;
۲۱  }
۲۲  /* binsearch: find word in tab[0]...tab[n-1] */
۲۳  int
۲۴  binsearch (char *word, struct key tab[], int n)
۲۵  {
```

آرایه از ساختمان‌ها

```
۲۶    int cond;
۲۷    int low, high, mid;
۲۸    low = 0;
۲۹    high = n - 1;
۳۰    while (low <= high)
۳۱      {
۳۲        mid = (low + high) / 2;
۳۳        if ((cond = strcmp (word, tab[mid].word)) < 0)
۳۴           high = mid - 1;
۳۵        else if (cond > 0)
۳۶           low = mid + 1;
۳۷        else
۳۸           return mid;
۳۹      }
۴۰    return -1;
۴۱  }
```

آرایه از ساختمان‌ها

- متغیر NKEYS تعداد کلمات کلیدی را مشخص می‌کند. گرچه می‌توانیم تعداد کلمات کلیدی را به صورت دستی بشماریم، اما بهتر است از یک متغیر استفاده کنیم، چرا که ممکن است در آینده بخواهیم تعداد کلمات را افزایش دهیم. همچنین بهتر است برنامه را طوری بنویسیم که تعداد کلمات کلیدی به طور خودکار توسط برنامه تعیین شود.
- تعداد عناصر آرایه برابر است با اندازهٔ آرایهٔ keytab تقسیم بر اندازهٔ هر یک از عناصر آرایه.
- در زبان سی عملگر sizeof برای محاسبه اندازهٔ یک شیء تعریف شده است. می‌توانیم اندازه یک شیء در حافظه را به صورت sizeof object و اندازهٔ یک نوع داده را توسط sizeof (type) به دست آوریم. اندازه‌ای که این عملگر محاسبه می‌کند یک عدد صحیح بدون علامت از نوع size-t است. یک شیء ممکن است یک متغیر، یک آرایه یا یک متغیر از نوع ساختمان باشد. همچنین یک نوع می‌تواند یک نوع اولیه مانند int و double یا یک نوع تعریف شده توسط کاربر مانند یک ساختمان یا یک نوع مشتق شده مانند اشاره‌گر باشد.

آرایه از ساختمان‌ها

- بنابراین می‌توانیم تعداد عناصر آرایه را با تقسیم `sizeof keytab` اندازهٔ هر عنصر آرایه یعنی `sizeof (struct key)` به دست آوریم.

```
1 #define NKEYS (sizeof keytab/sizeof (struct key))
```

آرایه از ساختمان‌ها

- همچنین می‌توانیم به جای محاسبه اندازه ساختمان `key` اندازهٔ اولین عنصر آرایهٔ `keytab` را به صورت `sizeof (keytab[0])` محاسبه کنیم.
- مزیت روش دوم این است که اگر نام ساختمان تغییر کرد این خط از کد نیازی به تغییر ندارد.
- در برنامه قبل به یک تابع دریافت کلمات ورودی به نام `getword` نیز نیاز داشتیم. این تابع کلمهٔ بعدی را از ورودی دریافت می‌کند. کلمهٔ دریافتی کلمهٔ دریافتی کلمه‌ای است که از ترکیب کلمات و حروف تشکیل شده است و با یک حرف آغاز می‌شود. مقداری که این تابع باز می‌گرداند اولین کاراکتر کلمهٔ دریافتی است . در صورتی که رشته ورودی به پایان رسیده باشد، مقدار بازگشت داده شده توسط تابع مقدار `EOF` است.

آرایه از ساختمان‌ها

- تابع دریافت ورودی به صورت زیر نوشته می‌شود.

```
۱  /* getword: get next word or character from input */
۲  int
۳  getword (char *word, int lim)
۴  {
۵    int c, getch (void);
۶    void ungetch (int);
۷    char *w = word;
۸    while (isspace (c = getch ()))
۹      ;
۱۰   if (c != EOF)
۱۱     *w++ = c;
۱۲   if (!isalpha (c))
۱۳     {
۱۴       *w = '\0';
```

آرایه از ساختمان‌ها

```
۱۵        return c;
۱۶      }
۱۷    for (; --lim > 0; w++)
۱۸      if (!isalnum (*w = getch ()))
۱۹        {
۲۰           ungetch (*w);
۲۱           break;
۲۲        }
۲۳    *w = '\0';
۲۴    return word[0];
۲۵  }
```

آرایه از ساختمان‌ها

- در این برنامه از تابع `isspace` برای تشخیص کاراکتر خط فاصله، از تابع `isalpha` برای تشخیص حروف الفبا و از تابع `isalnum` برای تشخیص حروف و ارقام استفاده شده است که همگی در کتابخانه ⟨ `ctype.h` ⟩ تعریف شده‌اند.

اشاره‌گر به ساختمان‌ها

- حال می‌خواهیم به جای آرایه‌ای از یک ساختمان از یک اشاره‌گر به ساختمان استفاده کنیم. برنامهٔ شمارش کلمات کلیدی را یک بار دیگر با استفاده از اشاره‌گرها پیاده‌سازی می‌کنیم.

```
۱ #include <stdio.h>
۲ #include <ctype.h>
۳ #include <string.h>
۴ #define MAXWORD 100
۵ int getword (char *, int);
۶ struct key *binsearch (char *, struct key *, int);
۷ /* count C keywords; pointer version */
۸ int main ()
۹ {
```

اشاره‌گر به ساختمان‌ها

```
۱۰    char word[MAXWORD];
۱۱    struct key *p;
۱۲    while (getword (word, MAXWORD) != EOF)
۱۳      if (isalpha (word[0]))
۱۴        if ((p = binsearch (word, keytab, NKEYS)) != NULL)
۱۵              p->count++;
۱۶    for (p = keytab; p < keytab + NKEYS; p++)
۱۷      if (p->count > 0)
۱۸        printf ("%4d %s\n", p->count, p->word);
۱۹        return 0;
۲۰  }
۲۱  /*binsearch:find word in tab[0] ... tab[n - 1] */
۲۲  struct key *
۲۳  binsearch (char *word, struck key * tab, int n)
۲۴  {
```

اشاره‌گر به ساختمان‌ها

```
۲۵    int cond;
۲۶    struct key *low = &tab[0];
۲۷    struct key *high = &tab[n];
۲۸    struct key *mid;
۲۹    while (low < high)
۳۰      {
۳۱        mid = low + (high - low) / 2;
۳۲        if ((cond = strcmp (word, mid->word)) < 0)
۳۳            high = mid;
۳۴        else if (cond > 0)
۳۵            low = mid + 1;
۳۶        else
۳۷            return mid;
۳۸      }
۳۹    return NULL;
۴۰  }
```

اشاره‌گر به ساختمان‌ها

- در این برنامه اگر تابع `binsearch` یک کلمه را پیدا کند اشاره‌گری به ساختمان پیدا شده بازمی‌گرداند در غیراینصورت مقدار `NULL` را باز می‌گرداند.
- همچنین `low` و `hight` دو اشاره‌گر هستند و از آنجایی که جمع دو اشاره‌گر بی معنی و غیر مجاز است، بنابراین عنصر وسط از طریق رابطهٔ `mid = low + (high - low)/2` محاسبه می‌شود
- حلقهٔ `for` به صورت `for (p = keytab; p < keytab + NKEYS ; p++)` نوشته شده است. از آنجایی که `p` به یک ساختمان اشاره می‌کند، با هر بار افزودن آن به میزان یک واحد، در حافظه به میزان فضای یک ساختمان به جلو حرکت می‌کنیم.
- البته توجه کنید که اندازهٔ یک ساختمان دقیقا به اندازهٔ مجموع اعضایش نیست. ممکن است برای دسترسی سریع‌تر به اعضای ساختمان، اعضای آن به گونه‌ای در حافظه قرار بگیرند که فضای تخصیص داده شده به یک متغیر از ساختمان بیشتر از مجموع اندازهٔ اعضای آن باشد.

اشاره‌گر به ساختمان‌ها

- برای مثال ساختمان زیر ۸ بایت اشغال می‌کند و نه ۵ باید، چراکه برای همگون شدن دسترسی به حافظه برای هر دو متغیر `c` و `i` مقدار ۴ باید در نظر گرفته می‌شود.

```
1 struct {
2         char c;
3         int i;
4 };
```

ساختمان‌های خود ارجاع

- فرض کنید می‌خواهیم برنامه‌ای بنویسیم که همهٔ رخدادهای هر کلمه را در یک متن بشمارد. لیست همهٔ کلمان قبل مشخص نیست، بنابراین نمی‌توانیم از جستجوی دودویی به شکلی که قبلاً استفاده کردیم، استفاده کنیم. همچنین نمی‌توانیم از یک جستجوی خطی استفاده کنیم، چراکه چنین جستجوی زمان زیادی برای اجرا نیاز خواهد داشت.
- یک راه‌حل این است که لیست کلمات را همیشه مرتب شده نگه‌داریم، بتوانیم با سرعت زیاد در آن جستجو کنیم. هر کلمهٔ جدیدی که وارد لیست می‌شود، باید در جای مناسب خود در لیست قرار بگیرد. این کار را نمی‌توانیم در یک آرایه انجام دهیم، زیرا هربار یک عنصر جدید در میان آرایه قرار می‌گیرد، عناصر باید جابجا شوند که چنین کاری زمان زیادی صرف خواهد کرد.
- برای حل این مسئله از یک ساختمان داده به نام درخت دودویی [1] استفاده می‌کنیم.

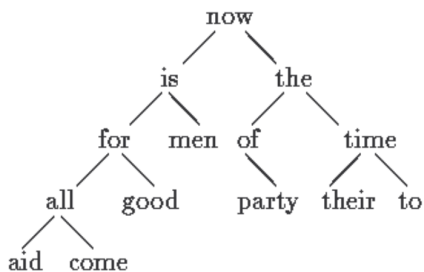
[1] binary tree

ساختمان‌های خود ارجاع

- درخت دودویی درختی است که از یک رأس به نام ریشه تشکیل شده و در آن رأس دارای حداکثر دو فرزند است که فرزند سمت چپ و فرزند سمت راست نام دارند.
- ساختمان دادهٔ مورد استفاده در این مسئله یک درخت دودویی است که هر رأس آن شامل یک کلمه و تعداد تکرار آن در متن است.
- اگر فرزند سمت چپ `N` را `L` بنامیم، فرزندان سمت چپ `N` شامل `L` و همهٔ فرزندان `L` می‌شود.
- درخت جستجوی دودویی به گونه‌ای ساخته می‌شود که فرزندان سمت چپ رأس `N` از لحاظ الفبایی کوچکتر از رأس `N` هستند و فرزندان سمت راست، بزرگتر از رأس `N`.

ساختمان‌های خود ارجاع

- برای مثال درخت جستجوی دودویی برای جملهٔ
`now is the time for all good men to come to aid of their party`
به صورت زیر است.



ساختمان‌های خود ارجاع

- برای جستجوی کلمه‌ای که در درخت وجود دارد، از ریشه آغاز می‌کنیم و مقدار آن را با ریشه مقایسه می‌کنیم. اگر مقدار آن کلمه برابر با محتوای ریشه برابر بود، کلمه پیدا شده است، در غیر اینصورت اگر مقدار آن کلمه از محتوای ریشه کوچکتر بود، فرزند سمت چپ را در نظر می‌گیریم و اگر محتوای آن کلمه از ریشه بزرگ‌تر بود، فرزند سمت راست را در نظر می‌گیریم. این فرایند ادامه پیدا می‌کند تا اینکه یا کلمهٔ مورد نظر در یکی از رئوس پیدا شود و یا به رأسی برخورد کنیم که هیچ فرزندی نداشته باشد که در اینصورت کلمهٔ در درخت وجود ندارد و در همان مکان آن را درج می‌کنیم. این عملیات به صورت بازگشتی انجام می‌شود.

- برای نگه‌داری یک رأس از این درخت ساختمان زیر را تعریف می‌کنیم.

```
۱ struct tnode
۲ {         /* the tree node: */
۳   char *word;         /* points to the text */
۴   int count;          /* number of occurrences */
۵   struct tnode *left;       /* left child */
۶   struct tnode *right;      /* right child */
۷ };
```

ساختمان‌های خود ارجاع

- این تعریف یک تعریف بازگشتی است، بدین معنی که در تعریف یک رأس از خود رأس استفاده می‌کنیم. به چنین ساختمانی یک ساختمان خود ارجاع [1] گفته می‌شود.
- یک ساختمان نمی‌تواند عضوی از نوع خودش را شامل شود ولی تعریف یک اشاره‌گر به ساختمانی از نوع خود ساختمان امکان پذیر است. بنابراین فرزندان سمت چپ و راست به صورت اشاره‌گر تعریف شده‌اند.

[1] self-referential structure

ساختمان‌های خود ارجاع

- بدنهٔ اصلی این برنامه به صورت زیر نوشته می‌شود.

```
۱ #include <stdio.h>
۲ #include <ctype.h>
۳ #include <string.h>
۴ #define MAXWORD 100
۵ struct tnode *addtree (struct tnode *, char *);
۶ void treeprint (struct tnode *);
۷ int getword (char *, int);
```

ساختمان‌های خود ارجاع

```
 ۸  /* word frequency count */
 ۹  int main ()
۱۰  {
۱۱    struct tnode *root;
۱۲    char word[MAXWORD];
۱۳    root = NULL;
۱۴    while (getword (word, MAXWORD) != EOF)
۱۵      if (isalpha (word[0]))
۱۶            root = addtree (root, word);
۱۷    treeprint (root);
۱۸    return 0;
۱۹  }
```

ساختمان‌های خود ارجاع

- تابع `addtree` یک تابع بازگشتی است. این تابع با دریافت ریشهٔ درخت به عنوان ورودی در درخت دودویی جستجو انجام می‌دهد تا اینکه یا به کلمهٔ مورد جستجو برسد و یا در صورت یافته نشدن کلمهٔ مورد جستجو آن را ایجاد کند.

ساختمان‌های خود ارجاع

- تابع جستجو در درخت به صورت زیر نوشته می‌شود.

```
۱  struct tnode *talloc (void);
۲  char *strdup (char *);
۳  /* addtree: add a node with w, at or below p */
۴  struct treenode *
۵  addtree (struct tnode *p, char *w)
۶  {
۷    int cond;
۸    if (p == NULL)
۹      {           /* a new word has arrived */
۱۰       p = talloc ();      /* make a new node */
۱۱       p->word = strdup (w);
```

ساختمان‌های خود ارجاع

```
۱۲        p->count = 1;
۱۳        p->left = p->right = NULL;
۱۴      }
۱۵    else if ((cond = strcmp (w, p->word)) == 0)
۱۶      p->count++;              /* repeated word */
۱۷    else if (cond < 0)       /* less than into left subtree */
۱۸      p->left = addtree (p->left, w);
۱۹    else                /* greater than into right subtree */
۲۰      p->right = addtree (p->right, w);
۲۱    return p;
۲۲  }
```

ساختمان‌های خود ارجاع

- در این تابع برای تخصیص حافظه به یک رأس از درخت از تابع talloc استفاده شده است، که یک مکان از حافظه برای ذخیره‌سازی کلمهٔ جدید بازمی‌گرداند.
- تابع treeprint محتوای یک درخت را چاپ می‌کند. این تابع به صورت زیر نوشته می‌شود.

```
۱  /* treeprint: in-order print of tree p */
۲  void
۳  treeprint (struct tnode *p)
۴  {
۵    if (p != NULL)
۶      {
۷        treeprint (p->left);
۸        printf ("%4d %s\n", p->count, p->word);
۹        treeprint (p->right);
۱۰     }
۱۱ }
```

ساختمان‌های خود ارجاع

- توجه کنید که این درخت ممکن است نامتوازن شود، زیرا کلمات با هر ترتیبی می‌توانند وارد شوند. در بدترین حالت همهٔ کلماتی که وارد درخت می‌شوند مرتب شده هستند و بنابراین (در یک ترتیب صعودی) هر رأس در سمت راست رأس قبلی خود وارد می‌شود و برای جستجو همهٔ رئوس باید بررسی شوند. ساختمان‌های داده‌ای برای بهبود این درخت و تولید درخت متوازن وجود دارند که به آن نمی‌پردازیم.

ساختمان‌های خود ارجاع

- همانطور که گفته شد اندازهٔ یک ساختمان دقیقاً به اندازهٔ مجموع اعضای آن نیست و ممکن است به دلیل تسهیل دسترسی، اندازهٔ یک ساختمان از مجموع اعضای آن بیشتر باشد. تابع `malloc` در زبان سی، برای تخصیص حافظه استفاده می‌شود که اندازهٔ مورد نیاز یک ساختمان را محاسبه کرده، فضای مورد نیاز را در حافظه تخصیص می‌شود و اشاره‌گری به ابتدای حافظهٔ تخصیص داده شده بر می‌گرداند.

ساختمان‌های خود ارجاع

- تابع talloc برای تخصیص حافظه به یک رأس به صورت زیر نوشته می‌شود.

```
۱ #include <stdlib.h>
۲ /* talloc: make a tnode */
۳ struct tnode *
۴ talloc (void)
۵ {
۶   return (struct tnode *) malloc (sizeof (struct tnode));
۷ }
```

ساختمان‌های خود ارجاع

- تابع strdup برای تخصیص فضای حافظه به یک رشته جهت کپی کردن رشته به کار می‌رود که به صورت زیر نوشته می‌شود.

```
۱ char *
۲ strdup (char *s)
۳ {
۴   char *p;          /* make a duplicate of s */
۵   p = (char *) malloc (strlen (s) + 1);    /* +1 for '\0' */
۶   if (p != NULL)
۷     strcpy (p, s);
۸   return p;
۹ }
```

ساختمان‌های خود ارجاع

- در صورتی که تابع `malloc` فضای مورد نیاز در حافظه را پیدا نکند مقدار `NULL` بازمی‌گرداند. برای آزادسازی حافظه‌ای که توسط `malloc` تخصیص داده شده است از `free` استفاده می‌شود.

تعریف نوع

- در زبان سی، می‌توان برای یک نوع نام دیگری تعریف کرد. برای مثال می‌توانیم نوعی به نام Length تعریف کنیم که همان نوع int است.

```
۱ typedef int Length;
۲ Length len, maxlen;
```

تعریف نوع

- به عنوان مثال دیگر می‌توانیم نوعی به نام string برای جایگزین کردن * char تعریف کنیم.

```
1 typedef char* string;
2 string p;
3 p = (string) malloc (100);
```

- معمولاً از typedef برای نامگذاری ساختمان‌ها استفاده می‌شود.

```
1 typedef struct tnode *Treeptr;
2 typedef struct tnode
3 {           /* the tree node: */
4   char *word;         /* points to the text */
5   int count;            /* number of occurrences */
6   struct tnode *left;     /* left child */
7   struct tnode *right;     /* right child */
8 } Treenode;
```

تعریف نوع

- بدین ترتیب می‌توانیم یک متغیر از ساختمان را با نام جدید تعریف کنیم. برای مثال :

```
1  Treeptr talloc(void) {
2          return (Treeptr) malloc (sizeof (Treenode));
3  }
```

تعریف نوع

- توجه کنید که `typedef` نوع جدید نمی‌سازد بلکه یک نام نمادین برای یک نوع موجود به وجود می‌آورد.
- چند دلیل برای استفاده از `typedef` وجود دارد. دلیل اول این است که گاهی برای تغییر یک برنامه و جابجایی آن از یک ماشین به ماشین دیگر ممکن است بخواهیم نوع‌ها را تغییر دهیم. در این‌صورت اگر از `typedef` استفاده کرده باشیم تنها نیاز به تغییر یک خط از برنامه داریم، در غیراینصورت باید در همهٔ برنامه جستجو کنیم و نوع مورد نظر را تغییر دهیم. دلیل دوم این است که می‌توانیم نام‌های معنی‌دارتر به نوع‌ها بدهیم. برای مثال اشاره‌گر به یک ساختمان را با `treeptr` نامگذاری کردیم که خوانایی برنامه را افزایش می‌دهد. همچنین نام‌های پیچیده را می‌توانیم با نام‌های کوتاهتر جایگزین کنیم تا خوانایی برنامه افزایش پیدا کند.

اجتماع

- یک اجتماع نوع داده‌ای است که می‌تواند چندین شیء متفاوت از نوع‌ها و اندازه‌های متفاوت را دربر بگیرد، به طوری‌که فقط از یک مکان حافظه برای ذخیره‌سازی همهٔ آن اشیاء استفاده می‌شود. بنابراین اندازهٔ یک اجتماع برابراست با اندازهٔ بزرگترین شیء آن اجتماع.

اجتماع

- برای مثال فرض کنید یک مقدار داشته باشیم که می‌تواند عدد صحیح، عدد اعشاری یا رشته باشد. اگر سه متغیر برای این کار در نظر بگیریم، درواقع در استفاده از حافظه اسراف کرده‌ایم، زیرا همیشه یکی از سه متغیر حاوی مقدار و دو متغیر دیگر بدون استفاده باقی می‌مانند. بهتر است برای این سه متغیر از یک فضای حافظه استفاده کنیم. برای این‌کار از اجتماع استفاده می‌کنیم.
- اجتماع زیر سه متغیر با نوع‌های صحیح، اعشاری و اشاره‌گر به رشته تعریف می‌کند.

```
۱ union u-tag {
۲           int inval;
۳           float fval;
۴           char* sval;
۵ }u;
```

اجتماع

- برای مثال اگر متغیر utype نوعی که در u ذخیره می‌شود را نگهداری کند، می‌توانیم از متغیر u به صورت زیر استفاده کنیم.

```
۱ if (utype == INT)
۲     printf("%d\n", u.ival);
۳ if (utype == FLOAT)
۴     printf("%f\n", u.fval);
۵ if (utype == STRING)
۶     printf("%s\n", u.sval);
۷ else
۸     printf("bad type %d in utype\n", utype);
```

اجتماع

- از یک اجتماع می‌توان درون یک ساختمان نیز استفاده کرد. برای مثال عضو u از ساختمان زیر می‌تواند عدد صحیح، اعشاری یا رشته باشد.

```
۱  struct
۲  {
۳    char *name;
۴    int flags;
۵    int utype;
۶    union
۷    {
۸      int ival;
۹      float fval;
۱۰     char *sval;
۱۱   } u;
۱۲ } symtab[NSYM];
```

اجتماع

- سپس می‌توانیم به مقادیر اجتماع از این ساختمان به صورت زیر دسترسی پیدا کنیم.

```
symtab[i].u.ival;
*symtab[i].u.sval;
symtab[i].u.sval[0];
symtab[i].u.fval;
```

اجتماع

- اندازهٔ یک متغیر از نوع اجتماع به مقدار اندازهٔ بزرگ‌ترین عضو آن است. اجتماع‌ها را می‌توان با عملگر انتساب به یکدیگر نسبت داد یا آدرس آنها را دریافت کرد یا به اعضای آنها دسترسی پیدا کرد.

ورودی و خروجی

ورودی و خروجی استاندارد

- توابع مربوط به ورودی و خروجی که برای دریافت ورودی از ورودی استاندارد و فایل‌ها و درج خروجی برروی خروجی استاندارد و فایل‌ها طراحی شده‌اند، جزئی از زبان سی نیستند بلکه در کتابخانه‌ها پیاده‌سازی شده‌اند.
- یک جریان متنی [1] یا استریم متنی دنباله‌ای است از چند خط به طوری که هر خط با کاراکتر خط جدید پایان می‌یابد.
- ساده‌ترین سازوکار دریافت ورودی توسط یک برنامه از محیط، دریافت یک کاراکتر از ورودی استاندارد است که از طریق صفحه کلید دریافت می‌شود. تابع پیاده‌سازی در کتابخانهٔ استاندارد بدین منظور `int getchar(void)` است.
- تابع `getchar` یک کاراکتر از ورودی استاندارد دریافت کرده و بازمی‌گرداند. در صورت دریافت کاراکتر پایان فایل مقدار `EOF` توسط تابع بازگردانده می‌شود. این تابع در کتابخانه `stdio.h` تعریف شده است.

[1] text stream

ورودی و خروجی استاندارد

- در یک برنامهٔ سی می‌توان یک فایل را جایگزین ورودی استاندارد کرد، بدین معنی که برنامه به جای دریافت برنامه از ورودی استاندارد مقادیر ورودی را از فایل دریافت کند. این کار توسط علامت > انجام می‌شود.
- برای مثال فرض کنید برنامهٔ `prog` توسط تابع `getchar` مقادیری را از ورودی استاندارد دریافت می‌کند. در این‌صورت می‌توان توسط اجرای `prog < infile` فایل ورودی `infile` را جایگزین ورودی استاندارد کرد. توجه کنید برای این‌کار هیچ نیازی به تغییر برنامه `prog` نیست و همچنین `prog` نیازی نیست فایل ورودی را در لیست ورودی‌های خود توسط `argv` دریافت کند.
- همچنین در سیستم عامل‌های پایهٔ یونیکس می‌توان خروجی یک برنامه را به عنوان ورودی به برنامهٔ دیگر ارسال کرد. برای مثال اجرای `prog | otherprog` خروجی برنامهٔ `otherprog` را به عنوان ورودی برنامهٔ `prog` استفاده می‌کند.

ورودی و خروجی استاندارد

- تابع `int putchar (int)` برای ارسال یک خروجی از برنامهٔ سی استفاده می‌شود. با فراخوانی تابع `putchar (c)` کاراکتر `c` برروی خروجی استاندارد قرار می‌گیرد که به صورت پیش فرض صفحه نمایش است. تابع مقدار نوشته شده را باز می‌گرداند و در صورت بروز خط مقدار `EOF` را بازمی‌گرداند. خروجی استاندارد را می‌توان توسط علامت `>` برروی فایل ارسال کرد.
- برای مثال اگر برنامهٔ `prog` از تابع `putchar` استفاده کند، می‌توان توسط اجرای `prog > outfile` خروجی برنامه را به جای نمایش بر روی صفحه، برروی فایل ذخیره کرد. همچنین اجرای `prog | anotherprog` خروجی `prog` را به عنوان ورودی به `anotherprog` ارسال می‌کند.
- خروجی تابع `printf` نیز بر روی خروجی استاندارد چاپ می‌شود.
- همهٔ این توابع در کتابخانهٔ `stdio` تعریف شده‌اند. این کتابخانه را می‌توان توسط `#include ⟨ stdio.h ⟩` به برنامه ضمیمه کرد. در سیستم‌عامل‌های پایهٔ یونیکس فایل `stdio.h` به طور پیش‌فرض در آدرس `/usr/include` قرار گرفته است.

ورودی و خروجی استاندارد

- برنامهٔ زیر رشته‌ای را از ورودی دریافت می‌کند و پس از تبدیل حروف بزرگ به حروف کوچک، نتیجه را برروی خروجی استاندارد چاپ می‌کند.

```
۱ #include <stdio.h>
۲ #include <ctype.h>
۳ int main ()     /* lower: convert input to lower case */
۴ {
۵   int c while ((c = getchar ()) != EOF)
۶     putchar (tolower (c));
۷   return 0;
۸ }
```

- تابع `tolower` در کتابخانهٔ `ctype.h` تعریف شده است که یک حرف بزرگ را به حرف کوچک تبدیل می‌کند.

خروجی قالب‌بندی شده

- تابع `printf` تعدادی از مقادیر را به صورت رشته قالب‌بندی کرده و برروی خروجی استاندارد چاپ می‌کند.
- تابع `int printf (char *format, arg1, arg2, ...)` رشته‌ای که در پارامتر اول دریافت می‌کند را با جایگزین کردن زیر رشته‌های آن با مقادیر پارامترهای دوم و سوم و ... در خروجی استاندارد چاپ می‌کند. تابع تعداد کاراکترهای چاپ شده را بازمی‌گرداند.
- رشته دریافت شده در پارامتر اول رشته قالب‌بندی نامیده می‌شود. این رشته شامل تعدادی زیر رشته است که به طور معمول برروی خروجی استاندارد چاپ می‌شوند و همچنین شامل تعدادی عملگر تبدیل است که یک نوع را به رشته به صورتی قالبی معین تبدیل می‌کنند. هر عملگر تبدیل با علامت % آغاز می‌شود.

خروجی قالب‌بندی شده

- عملگر تبدیل دارای چند بخش است که در اینجا به بخش‌های آن اشاره می‌کنیم.
- علامت منفی در ابتدا باعث می‌شود مقدار از سمت چپ تراز شود. در صورتی که علامت منفی وجود نداشته باشد مقدار رشتهٔ تبدیل شده از سمت راست تراز می‌شود.
- عددی که در ابتدا عملگر نوع نوشته می‌شود، عرض رشتهٔ چاپ شده را تعیین می‌کند. به عبارت دیگر رشتهٔ تبدیل شده در یک فضا با عرض تعیین شده چاپ می‌شود.
- پس از عرض فضا، توسط علامت نقطه و یک عدد صحیح، دقت چاپ یک عدد را مشخص کرد. به عبارت دیگر برای اعداد اعشاری می‌توان تعداد ارقام مورد نیاز برای چاپ را پس از علامت نقطه تعیین کرد. برای اعداد صحیح حداقل تعداد ارقام و برای رشته‌ها حداکثر تعداد کاراکترها را می‌توان مشخص کرد.
- سپس برای متغیرهای `short` از حرف `h` و برای متغیرهای نوع `long` از `l` استفاده می‌کنیم.

خروجی قالب‌بندی شده

- برای مثال با فراخوانی `printf ("%*s", max, s)` حداکثر تعداد `max` کاراکتر از رشتهٔ `s` بر خروجی استاندارد چاپ می‌شود.
- مثال‌های زیر روش استفاده از عملگر تبدیل نوع را برای رشته‌ها نشان می‌دهد.

```
:%s:          :hello, world:
:%10s:        :hello, world:
:%.10s:       :hello, wor:
:%-10s:       :hello, world:
:%.15s:       :hello, world:
:%-15s:       :hello, world   :
:%15.10s:     :     hello, wor:
:%-15.10s:    :hello, wor     :
```

خروجی قالب‌بندی شده

- تابع `sprintf` شبیه `printf` عمل می‌کند با این تفاوت که خروجی قالب بندی شده را به جای چاپ برروی خروجی استاندارد در یک رشته کپی می‌کند.

```
int sprintf (char *string, char *format, arg1, arg2, ...)
```

لیست آرگومان‌ها با طول متغیر

- تعداد پارامترهای تابع `printf` متغیر است. در زبان سی می‌توان توابع با تعداد پارامترهای متغیر تعریف کرد.
- در اینجا می‌خواهیم تابعی شبیه به `printf` با تعداد پارامترهای متغیر پیاده‌سازی کنیم.
- در کتابخانهٔ `stdarg` تعدادی ماکرو برای ایجاد امکان تعداد پارامترهای متغیر تعریف شده است.
- نوع `va-list` یک لیست شامل تعدادی متغیر تعریف می‌کند. توسط ماکروی `va-start` می‌توان به ابتدای لیست متغیرها اشاره کرد. با هر بار فراخوانی `va-arg` در لیست متغیرها، یک متغیر به جلو حرکت می‌کنیم. در پایان `va-end` عملیات پایانی و آزادسازی حافظه‌های غیر مورد نیاز را انجام می‌دهد.

لیست آرگومان‌ها با طول متغیر

- تابع حداقلی چاپ به نام minprintf به صورت زیر پیاده‌سازی می‌شود.

```
۱  #include <stdarg.h>
۲  /* minprintf: minimal printf with variable argument list */
۳  void
۴  minprintf (char *fmt, ...)
۵  {
۶    va_list ap;         /* points to each unnamed arg in turn */
۷    char *p, *sval;
۸    int ival;
۹    double dval;
۱۰   va_start (ap, fmt);      /* make ap point to 1st unnamed arg */
```

لیست آرگومان‌ها با طول متغیر

```
11    for (p = fmt; *p; p++)
12      {
13           if (*p != '%')
14                {
15                     putchar (*p);
16                     continue;
17                }
18          switch (*++p)
19      {
20          case 'd':
21              ival = va_arg (ap, int);
22              printf ("%d", ival);
23              break;
```

لیست آرگومان‌ها با طول متغیر

```
۲۴      case 'f':
۲۵          dval = va_arg (ap, double);
۲۶          printf ("%f", dval);
۲۷          break;
۲۸      case 's':
۲۹           for (sval = va_arg (ap, char *); *sval; sval++)
۳۰                 putchar (*sval);
۳۱           break;
۳۲       default:
۳۳          putchar (*p);
۳۴          break;
۳۵        }
۳۶    }
۳۷    va_end (ap);      /* clean up when done */
۳۸  }
```

لیست آرگومان‌ها با طول متغیر

- تابع `scanf` شبیه تابع `printf` است با این تفاوت که به جای چاپ رشته قالب بندی شده برروی خروجی استاندارد، یک رشته قالب‌بندی شده را از ورودی استاندارد دریافت می‌کند.
- این تابع رشته‌ای را به صورتی که در آرگومان اول تابع تعیین شده دریافت می‌کند و عملگرهای تبدیل را با متغیرهای آرگومان‌های دوم و سوم و ... جایگزین می‌کند. به عبارت دیگر ورودی دریافت شده از کاربر را در متغیرهای آرگومان‌های دوم و سوم و ... ذخیره می‌کند.
- برای این که تابع `scanf` بتواند مقادیر متغیرهای ورودی خود را تغییر دهد، نیاز به فراخوانی با ارجاع است، بنابراین آرگومان‌های دوم به بعد از نوع اشاره‌گر هستند. برای ارسال یک متغیر به این تابع باید آدرس آن با استفاده از عملگر & ارسال شود.
- در صورتی که ورودی خوانده شده با آرگومان اول همخوانی نداشته باشد، تابع مقدار خطا بازمی‌گرداند.
- در صورت موفقیت، تابع تعداد کاراکترهای خوانده شده را بازمی‌گرداند و در صورت رسیدن به کاراکتر پایان فایل مقدار `EOF` را بازمی‌گرداند.

لیست آرگومان‌ها با طول متغیر

- تابع sscanf شبیه scanf عمل می‌کند با این تفاوت که به جای دریافت ورودی از ورودی استاندارد، آن را از یک رشته دریافت می‌کند.

```
int scanf (char *format, arg1, arg2, ...);
int sscanf (char *string, char *format, arg1, arg2, ...);
```

لیست آرگومان‌ها با طول متغیر

- برنامه زیر تعدادی عدد از ورودی دریافت کرده، مجموع آن‌ها را چاپ می‌کند.

```
۱ #include <stdio.h>
۲ int main ()        /* rudimentary calculator */
۳ {
۴   double sum, v;
۵   sum = 0;
۶   while (scanf ("%lf", &v) == 1)
۷     printf ("\t%.2f\n", sum += v);
۸   return 0;
۹ }
```

لیست آرگومان‌ها با طول متغیر

- فرض کنید می‌خواهیم یک تاریخ را از ورودی دریافت کنیم. این کار به صورت زیر انجام می‌شود.

```
1 int day, year;
2 char monthname[20];
3 scanf("%d %s %d", &day, monthname, &year);
```

لیست آرگومان‌ها با طول متغیر

- حال فرض کنید می‌خواهیم یک تاریخ را به صورت dd Month yy یا dd/mm/yy دریافت کنیم.
می‌توانیم ابتدا یک خط از ورودی را دریافت و در یک رشته ذخیره کنیم و سپس توسط sscanf مقادیر مورد نظر را از روی رشته خوانده شده بخوانیم.

```
۱ while (getline (line, sizeof (line)) > 0)
۲   {
۳     if (sscanf (line, "%d %s %d", &day, monthname, &year) == 3)
۴       printf ("valid: %s\n", line);   /* 25 Dec 1988 form */
۵     else if (sscanf (line, "%d/%d/%d", &month, &day, &year) == 3)
۶       printf ("valid: %s\n", line);   /* mm/dd/yy form */
۷     else
۸       printf ("invalid: %s\n", line);    /* invalid form */
۹   }
```

فایل‌ها

- در مثال‌های قبل برنامه‌ها مقادیر ورودی را از ورودی استاندارد (مانند کیبورد) می‌خواندند و مقادیر خروجی را بر روی خروجی استاندارد (مانند صفحه نمایش) چاپ می‌کردند.
- حال می‌خواهیم برای ورودی و خروجی از فایل‌ها استفاده کنیم. یک فایل یک واحد از اطلاعات است که برروی حافظه‌های بلندمدت ذخیره می‌شود و با یک نام و آدرس یکتا قابل دسترسی است.
- یکی از برنامه‌های جانبی که در سیستم عامل‌های بر پایهٔ یونیکس برای چاپ محتوای فایل‌ها برروی خروجی استاندارد استفاده می‌شود، برنامه `cat` است. با اجرای `cat x.c y.c` محتوای دو فایل `x.c` و `y.c` برروی صفحه نمایش نشان داده می‌شود.
- در کتابخانهٔ استاندارد دستوراتی برای کار با فایل‌ها مهیا شده‌اند.

فایل‌ها

- قبل از استفاده از یک فایل باید آدرس فایل از سیستم‌عامل گرفته شود و عملیات مقدماتی برای خواندن از فایل و نوشتن روی فایل انجام شود. برای این کار از دستور `fopen` برای بازکردن فایل استفاده می‌شود.
دستور `fopen` یک اشاره‌گر بازمی‌گرداند که اشاره‌گر فایل نامیده می‌شود. این اشاره‌گر به ساختمانی در حافظه اشاره می‌کند که اطلاعاتی در مورد فایل (مانند بافر آن در حافظه، مکان فعلی در هنگام خواندن و نوشتن فایل، وضعیت فایل و غیره) را نگهداری می‌کند.
- یک فایل به صورت زیر تعریف و باز می‌شود.

```
1 FILE * fp;
2 FILE *fopen (char *name, char *mode);
```

- در اینجا `fp` اشاره‌گری است به داده‌ای از نوع `FILE` و تابع `fopen` با دریافت نام فایل و حالت فایل، یک اشاره‌گر بازمی‌گرداند.

فایل‌ها

- یک فایل می‌تواند با استفاده از رشته "r" برای خواندن، با رشتهٔ "w" برای نوشتن و با رشته "a" برای افزودن به فایل باز شود.
- اگر یک فایل که در حالت نوشتن باز می‌شود وجود نداشته باشد، فایل ساخته می‌شود. وقتی یک فایل موجود برای نوشتن باز شود، محتوای قبلی از بین می‌رود. اگر بخواهیم از فایلی که وجود ندارد بخوانیم، با پیام خطا روبرو می‌شویم. در هنگام بروز خطا تابع fopen مقدار NULL بازمی‌گرداند.
- با استفاده از تابع int getc (FILE *fp) می‌توانیم یک کاراکتر از فایل بخوانیم. در صورتی که به پایان فایل برسیم، تابع مقدار EOF بازمی‌گرداند.
- با استفاده از تابع int putc (int c, FILE *fp) کاراکتر c برروی فایل نوشته می‌شود.
- توجه کنید که getc و putc و getchar و putchar ماکرو هستند نه تابع، بنابراین سربار فراخوانی توابع را ندارند.

فایل‌ها

- وقتی یک برنامه سی آغاز می‌شود، سه فایل توسط سیستم‌عامل باز می‌شوند که این فایل‌ها عبارتند از فایل‌های ورودی استاندارد، خروجی استاندارد و پیام خطا استاندارد. سه اشاره‌گر `stdin` ، `stdout` و `stderr` بازگردانده می‌شوند که این فایل‌ها در کتابخانه `stdio.h` تعریف شده‌اند. تابع `getchar()` معادل است با تابع `getc(stdin)` ، بنابراین ماکروهای زیر تعریف شده‌اند :

```
1 #define getchar()  get(stdin)
2 #define putchar(c)  putc((c), stdout)
```

- برای ورودی و خروجی قالب بندی شده در فایل توابع `fprintf` و `fscanf` تعریف شده‌اند که به عنوان پارامتر اول یک اشاره‌گر به فایل دریافت می‌کنند.

```
1 int fscanf (FILE *fp, char *format, ...)
2 int fprintf (FILE *fp, char *format, ...)
```

فایل‌ها

- حال می‌خواهیم برنامه‌ای شبیه به cat بنویسیم که فایل‌های دریافت شده را برروی خروجی استاندارد چاپ کند.

```
۱  #include <stdio.h>
۲  /* cat: concatenate files, version 1 */
۳  int main (int argc, char *argv[])
۴  {
۵    FILE *fp;
۶    void filecopy (FILE *, FILE *) if (argc == 1)
۷    /* no args; copy standard input */
۸      filecopy (stdin, stdout);
۹    else
۱۰     while (--argc > 0)
۱۱           if ((fp = fopen (*++argv, "r")) == NULL)
۱۲             {
۱۳                printf ("cat: can't open %s\n", *argv);
```

```
۱۵                return 1;
۱۶             }
۱۷     else
۱۸        {
۱۹           filecopy (fp, stdout);
۲۰           fclose (fp);
۲۱        }
۲۲     return 0;
۲۳  }
۲۴  /* filecopy: copy file ifp to file ofp */
۲۵  void
۲۶  filecopy (FILE * ifp, FILE * ofp)
۲۷  {
۲۸    int c;
۲۹    while ((c = getc (ifp)) != EOF)
۳۰      putc (c, ofp);
۳۱  }
```

فایل‌ها

- اشاره‌گرهای `stdin` و `stdout` از نوع `FILE` هستند.
- تابع `int fclose (FILE *fp)` برای خاتمه‌دادن به عملیات برروی فایل و بازیابی حافظه به کار می‌رود. این تابع عملیاتی برعکس عملیات مورد نیاز برای بازکردن فایل انجام می‌دهد و به عبارت دیگر فایل را می‌بندد. همچین تابع `fclose` هرآنچه در بافر نوشتن برروی فایل قرار دارد را در فایل ذخیره می‌کند.

فایل‌ها

- فرض کنید می‌خواهیم خروجی یک برنامه را برروی یک فایل ذخیره کنیم. اگر پیام‌های خطا برروی خروجی استاندارد چاپ شوند، خروجی و پیام‌های خطا همگی برروی فایل ذخیره می‌شوند. طراحی بهتر این است که پیام‌های خطا از خروجی برنامه جدا شوند تا بتوان خطاها را به طور جداگانه ذخیره و بررسی کرد.
- اشاره‌گر `stderr` یک خروجی استاندارد برای چاپ خطاهای برنامه است.

- در برنامهٔ زیر از خروجی استاندارد خطا برای چاپ خطاها استفاده شده است.

```
۱  #include <stdio.h>
۲  /* cat: concatenate files, version 2 */
۳  int main (int argc, char *argv[])
۴  {
۵    FILE *fp;
۶    void filecopy (FILE *, FILE *);
۷    char *prog = argv[0];   /* program name for errors */
۸    if (argc == 1)    /* no args; copy standard input */
۹        filecopy (stdin, stdout);
۱۰   else
۱۱       while (--argc > 0)
```

```
12        if ((fp = fopen (*++argv, "r")) == NULL)
13           {
14              fprintf (stderr, "%s: can't open %s\n", prog, *argv);
15              exit (1);
16           }
17        else
18          {
19              filecopy (fp, stdout);
20              fclose (fp);
21           }
22    if (ferror (stdout))
23      {
24           fprintf (stderr, "%s: error writing stdout\n", prog);
25           exit (2);
26       }
27    exit (0);
28  }
```

فایل‌ها

- در این برنامه پیام خطا برروی خروجی خطای استاندارد قرار می‌گیرد و همچنین در صورت بروز خطا با تابع `exit(1)` برنامه متوقف می‌شود.
- در تابع `main` دستور `return x` و `exit(x)` معادل یکدیگرند. مزیت تابع `exit` این است که از توابع دیگر غیر از `main` نیز می‌تواند فراخوانی شود که باعث توقف برنامه می‌شود.
- تابع `int ferror (FILE *fp)` درصورتی که خطایی در فایل رخ دهد مقدار غیر صفر بازمی‌گرداند.

فایل‌ها

- در کتابخانهٔ استاندارد تابعی برای خواندن یک خطا از یک فایل وجود دارد. این تابع یک اشاره‌گر به فایل دریافت می‌کند، و یک خط از ورودی را خوانده و اشاره‌گری به ابتدای آن بازمی‌گرداند.

```
char *fget (char *line, int maxline, FILE *fp)
```

- خط خوانده شده که طول آن حداکثر `maxline-1` است در متغیر `line` قرار می‌گیرد. در کاراکتر آخر مقدار `'\0'` قرار می‌گیرد.
- در صورت موفقیت این تابع اشاره‌گر `line` را بازمی‌گرداند و در غیراینصورت مقدار `NULL` بازگردانده می‌شود.
- برای نوشتن یک خط برروی یک فایل از تابع `fputs` استفاده می‌شود.

```
int fputs (char *line, FILE *fp)
```

- همچنین توابع `puts` و `gets` برای نوشتن رشته برروی خروجی استاندارد و خواندن رشته از ورودی استاندارد به‌کار می‌روند.

فایل‌ها

- توابع fgets و fputs به صورت زیر پیاده‌سازی می‌شوند.

```
۱  /* fgets: get at most n chars from iop */
۲  char *
۳  fgets (char *s, int n, FILE * iop)
۴  {
۵    register int c;
۶    register char *cs;
۷    cs = s;
۸    while (--n > 0 && (c = getc (iop)) != EOF)
۹      if ((*cs++ = c) == '\n')
۱۰       break;
۱۱   *cs = '\0';
۱۲   return (c == EOF && cs == s) ? NULL : s;
۱۳ }
```

```
۱۴ /* fputs: put string s on file iop */
۱۵ int
۱۶ fputs (char *s, FILE * iop)
۱۷ {
۱۸   int c;
۱۹   while (c = *s++)
۲۰     putc (c, iop);
۲۱   return ferror (iop) ? EOF : 0;
۲۲ }
```

- تابع getline برای دریافت یک خط از ورودی را می‌توانیم به صورت زیر با استفاده از fgets پیاده‌سازی کنیم.

```
۱ /* getline: read a line, return length */
۲ int
۳ getline (char *line, int max)
۴ {
۵   if (fgets (line, max, stdin) == NULL)
۶     return 0;
۷   else
۸     return strlen (line);
۹ }
```

توابع متفرقه

- در کتابخانه استاندارد توابع زیادی وجود دارند که می‌توانند مورد استفاده قرار بگیرند.
- در کتابخانهٔ `string.h` توابع زیر تعریف شده‌اند.

```
۱  strcat(s,t); // concatenate t to end of s
۲  strncat(s,t,n); // concatenate n characters of t to end of s
۳  strcmp(s,t);
۴      // return negative, zero, or positive for s < t, s == t, s > t
۵  strncmp(s,t,n); // same as strcmp but only in first n characters
۶  strcpy(s,t); // copy t to s
۷  strncpy(s,t,n); // copy at most n characters of t to s
۸  strlen(s); // return length of s
۹  strchr(s,c); // return pointer to first c in s, or NULL if not present
۱۰ strrchr(s,c); // return pointer to last c in s, or NULL if not present
```

توابع متفرقه

- در کتابخانهٔ ctype.h توابع زیر تعریف شده‌اند.

```
۱ isalpha(c); // non-zero if c is alphabetic, 0 if not
۲ isupper(c); // non-zero if c is upper case, 0 if not
۳ islower(c); // non-zero if c is lower case, 0 if not
۴ isdigit(c); // non-zero if c is digit, 0 if not
۵ isalnum(c); // non-zero if isalpha(c) or isdigit(c), 0 if not
۶ isspace(c); // non-zero if c is blank, tab, newline, return, formfeed,
۷ toupper(c); // return c converted to upper case
۸ tolower(c); // return c converted to lower case
```

توابع متفرقه

- در کتابخانهٔ استاندارد همچنین تابع `ungetch` تعریف شده است که یک کاراکتر خوانده شده را به جریان ورودی بازمی‌گرداند. به طور مشابه تابع `int ungetc (int c, FILE *fp)` یک کاراکتر خوانده شده از فایل را به جریان ورودی بازمی‌گرداند.
- تابع `void *malloc (size_t n)` اشاره‌گری به `n` بایت تخصیص داده شده در حافظه بازمی‌گرداند و حافظه مورد نیاز را تخصیص می‌دهد.
- تابع `void *malloc (size_t n, size_t size)` اشاره‌گری به `n` شیء هرکدام با اندازه `size` در حافظه بازمی‌گرداند و حافظه مورد نیاز را تخصیص می‌دهد.

توابع متفرقه

- برای مثال برای یک آرایه n تایی از اعداد صحیح می‌نویسیم :

```
int *ip;
ip = (int *) calloc (n, sizeof (int));
```

- تابع free(p) فضای حافظه‌ای که توسط اشاره‌گر p تخصیص داده شده است را آزاد می‌کند. اگر فضای حافظه یک بار آزاد شود، بار دوم با خطا روبرو می‌شویم.
- بنابراین در برنامه زیر با خطا روبرو می‌شویم.

```
for ( p = head; p != NULL ; p = p -> next ) /*WRONG*/
    free (p)
```

توابع متفرقه

- روش درست برای آزادسازی حافظه در این مثال به صورت زیر است:

```
for ( p = head ; p != NULL ; p = q ) {
        q = p -> next;
        free (p);
}
```

توابع متفرقه

- توابع ریاضی در کتابخانهٔ `math.h` پیاده‌سازی شده‌اند که می‌توانند مورد استفاده قرار بگیرند.

```
۱  sin(x); // sine of x, x in radians
۲  cos(x); // cosine of x, x in radians
۳  atan2(y,x); // arctangent of y/x, in radians
۴  exp(x); // exponential function e^x
۵  log(x); // natural (base e) logarithm of x (x>0)
۶  log10(x); // common (base 10) logarithm of x (x>0)
۷  pow(x,y); // x^y
۸  sqrt(x); // square root of x (x>0)
۹  fabs(x); // absolute value of x
۱۰ rand(); // generate a random number
۱۱ srand(x);  // sets the seed for rand()
```