

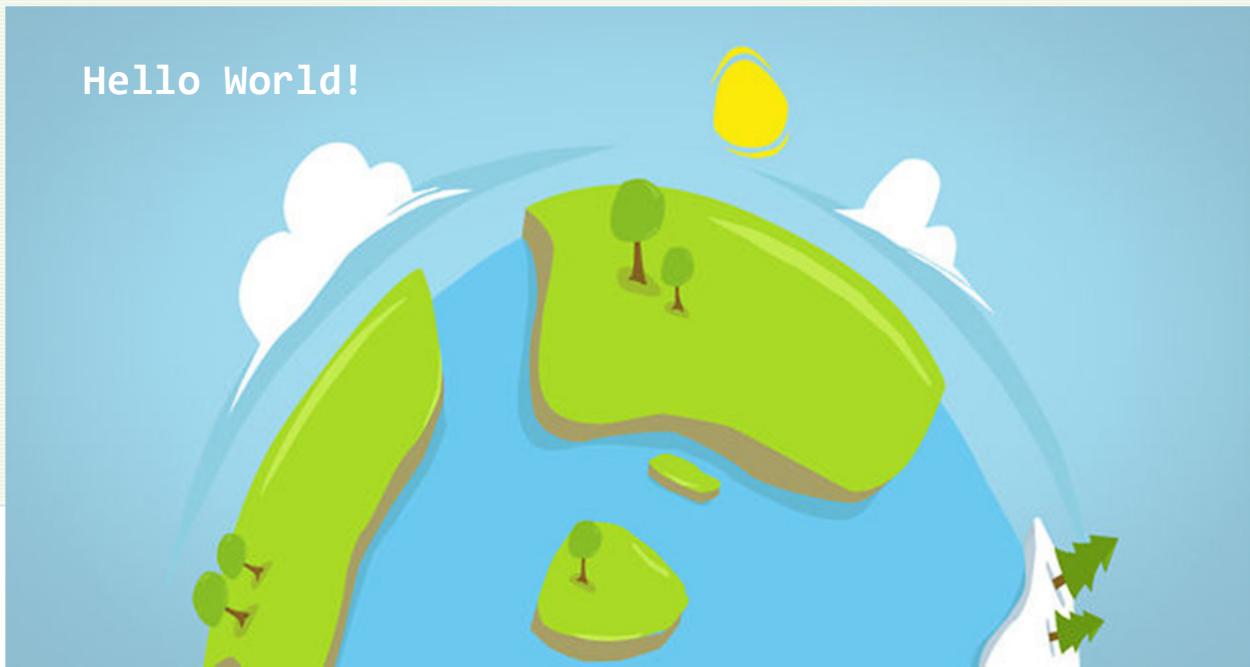


وزارت آموزش و پرورش
مرکز ملی پژوهش استعدادهای درخشان و دانش پژوهان جوان
دیستراستان علامه حلی (۳) تهران

برنامه نویسی به زبان [C++]

پدیدآورندگان:

محسن غفوریان، حسام علیزاده، میلاد مهدیزاده



به نام خداوند بخشندۀ مهریان

مقدمه:

مقصود ما از تالیف این کتاب آماده نمودن یک خودآموز به زبان ساده برای دانشآموزان دبیرستانی، بدون هیچ فرضی بر سابقه‌ی پیشین آشنایی با برنامه‌نویسی بوده است. محتوای این کتاب حاصل چند سال تجربه‌ی تدریس مولفین در مراکز استعدادهای درخشان می‌باشد. شاید یکی از ویژگی‌های ممتاز این نوشه آن باشد که تا جای ممکن سعی شده است با حذف بسیاری از جزئیات غیر ضروری زبان، یک مجموعه‌ی حداقلی لازم، کافی و نه چندان پیچیده آماده شود که به راحتی برای دانشآموزان دبیرستانی به منظور حل مسائل معمول، قابل استفاده باشد.

از جمله موارد دیگری که در دستور کار قرار دادیم این بود که بیش از آن که به نحو زبان پردازیم، ایده‌های حل مسائل را مورد تجزیه و تحلیل قرار دهیم و از همین رو بوده است که یکی از فصل‌های ابتدایی این کتاب را منحصراً به پرداختن به این ایده‌ها، مستقل از زبان برنامه‌نویسی اختصاص داده‌ایم، چرا که معتقدیم که مهم‌ترین کارکرد آموزش برنامه‌نویسی برای دانشآموزان دبیرستانی، یادگیری نظام تفکر و استراتژی حل مسئله می‌باشد. البته ناگفته پیدا است که آن‌چه در این مجال کوتاه آمده، به هیچ‌وجه پاسخگوی نیاز وسیع دانش‌پژوه در این زمینه نیست و اکیداً توصیه می‌گردد دانشآموزان در این مورد مطالعات فراتری انجام دهند.

و اما در مورد محتوای کتاب و این که این نوشتار درباره‌ی C هست یا C++ در واقع اصلی‌ترین ویژگی جدید C++ نسبت به C امکان آن برای برنامه‌نویسی شی‌گرایی است، اما علاوه بر آن تغییرات به نسبت فرعی‌تری نیز اعمال شده است و از آن جا که در این کتاب فرستاد ورود به مبحث شی‌گرایی وجود نداشت، مطالعه‌ی این مبحث بر عهده‌ی خواننده قرار داده شده است. اما از طرفی به منظور سهولت بیشتر نحو، از برخی ویژگی‌های دیگر C++ از جمله جریان ورودی و خروجی استفاده کرده‌ایم. از این رو، آن‌چه در این کتاب می‌بینید ترکیبی از زبان‌های برنامه‌نویسی C و C++ است (C++) نه هرکدام به صورت خالص.

به یاری خدا ویرایش دوم این کتاب پس از یک سال تدریس ویرایش اول آن به پایان رسید، اما بی‌گمان اشتباهاتی سهواً در این اثر جای دارند، امیدواریم انتقادات و پیشنهادات خردمندانه‌ی شما یاری‌گر ما برای بهبود کارایی آن باشد.

خواهشمندیم نظرات ارزشمند خود را به آدرس الکترونیکی cIntroductoryBook@gmail.com ارسال نمایید.

در پایان جا دارد کمال تشکر خود را از جناب آقای سید فخرالدین نصیری مدیریت محترم دبیرستان علامه حلی^۳، همچنین دیگر همکاران گرانقدرمان آقایان دکتر ناصر کریمی، ابراهیم مقتضی و محسن پیرحسین‌لو که نهایت همکاری را با ما برای تهیه‌ی این اثر داشتند، ابراز بداریم.

محسن غفوریان، حسام علیزاده، میلاد مهدی‌زاده

تابستان ۹۱

فهرست مطالب:

فصل اول: مبانی برنامه‌نویسی کامپیوتر

۷	۱. اهمیت آموختن برنامه‌نویسی
۸	۲. از برنامه‌ها تا صفر و یک
۱۰	۳. زبان برنامه‌نویسی C
۱۱	تمرین

فصل دوم: الگوریتم، ایده‌ی حل مسئله

۱۳	۱. الگوریتم چیست؟
۱۳	۲. مثال شوت زدن
۱۴	۳. نمایش الگوریتم
۱۶	۴. مثال شوت زدن (فلوچارت)
۱۶	۵. مثال درست کردن چای
۱۷	۶. مثال درست کردن چای (فلوچارت)
۱۸	۷. مثال نمایش عدد بزرگ‌تر
۱۸	۸. مثال نمایش عدد بزرگ‌تر (فلوچارت)
۱۹	۹. مثال نمایش بزرگ‌ترین عدد
۲۰	۱۰. مثال نمایش بزرگ‌ترین عدد (فلوچارت)
۲۱	تمرین

فصل سوم: ورودی و خروجی، متغیرها

۲۳	۱. اولین برنامه‌ی ما – Hello World!
۲۴	۲. معرفی متغیرها و گرفتن مقادیر از کاربر
۲۸	۳. محاسبه‌ی میانگین دو عدد صحیح
۲۹	۴. جایجایی دو متغیر
۳۱	۵. کمی در باره‌ی کاراکترها
۳۳	تمرین

فصل چهارم: گرافیک

۳۵	۱. مقدمه و راهاندازی گرافیک
۳۵	۲. آشنایی با محیط گرافیکی

۳۶.....	رسم شکل‌های اولیه
۳۸.....	استفاده از رنگ
۴۲.....	تمرین
۴۳.....	فصل پنجم: ساختار شرطی
۴۴.....	۱. معرفی ساختار شرطی
۴۴.....	۲. اندکی در باب شرط‌ها
۴۷.....	۳. شرط‌های ترکیبی
۴۹.....	۴. بلاک‌ها
۵۰.....	۵. و گرنی!
۵۲.....	تمرین
۵۳.....	فصل ششم: حلقه‌ی while
۵۴.....	۱. حلقه چیست؟
۵۴.....	۲. ساختار حلقه‌ی while
۵۵.....	۳. مثال چاپ ستاره در یک سطر
۵۵.....	۴. مثال توان عامل ۲
۵۶.....	۵. مثال فاکتوریل
۵۷.....	۶. متغیر پرچم
۵۷.....	۷. مثال شمارش اعداد زوج
۵۸.....	۸. حلقه‌ی do while
۵۹.....	۹. مثال چاپ بزرگ‌ترین عدد
۶۰.....	۱۰. مثال کد اسکی
۶۱.....	۱۱. مثال رسم دایره
۶۲.....	۱۲. مثال مجموع مربعات
۶۳.....	تمرین

۶۴.....	فصل هفتم: حلقه‌ی for
۶۵.....	۱. معرفی حلقه‌ی for
۶۶.....	۲. ساختار حلقه‌ی for
۶۷.....	۳. مثال اعداد میانی
۶۸.....	۴. مثال ترتیب نزولی
۶۹.....	۵. مثال رسم دایره

۷۰.....	۶ مثال عدد π
۷۲.....	تمرین
۷۳.....	فصل هشتم: آرایه‌ها
۷۴.....	۱. آشنایی اولیه با آرایه‌ها
۷۵.....	۲. مقداردهی اولیه به آرایه
۷۶.....	۳. مثال بدست آوردن میانگین اعداد
۷۷.....	۴. جستجو
۷۸.....	۵. پیدا کردن ماکریم یک آرایه
۷۹.....	۶ رشته‌ها
۸۴.....	۷. آرایه‌های دو بعدی
۸۷.....	تمرین
۸۸.....	فصل نهم: حلقه‌های تو در تو
۸۹.....	۱. حلقه‌ی تو در تو
۹۰.....	۲. مثال جدول ضرب
۹۱.....	۳. مثال ضرب
۹۱.....	۴. الگوریتم مرتب کردن اعداد
۹۲.....	۵. برنامه‌ی مرتب کردن اعداد
۹۴.....	تمرین
۹۵.....	فصل دهم: توابع
۹۶.....	۱. اهمیت استفاده از تابع - مثال رسم ستاره
۱۰۰.....	۲. پیدا کردن ماکریم سه عدد صحیح
۱۰۱.....	۳. فرستادن آرایه به عنوان پارامتر به تابع - مثال پیدا کردن میانگین اعداد یک آرایه
۱۰۳.....	۴. حوزه‌ی تعریف متغیرها در تابع
۱۰۴.....	۵. اعداد اول کوچک‌تر از عدد ورودی
۱۰۵.....	۶. مزایای استفاده از تابع
۱۰۸.....	تمرین
۱۰۹.....	فصل یازدهم: توابع بازگشته‌ی
۱۱۰.....	۱. معرفی برنامه‌نویسی بازگشته‌ی

۲. مثال محاسبه‌ی فاکتوریل.....	۱۱۰
۳. مثال محاسبه‌ی عدد ۱۷ام فیبوناچی.....	۱۱۵
۴. مثال برج‌های هانوی.....	۱۱۶
۵. مثال جستجوی دودویی.....	۱۱۹
تمرین.....	۱۲۲
فصل دوازدهم: ساختارها.....	۱۲۳
۱. ضرورت استفاده از ساختار.....	۱۲۴
۲. معرفی ساختار.....	۱۲۶
۳. استفاده از ساختار برای محیط گرافیکی.....	۱۲۸
۴. مثلث سرپینسکی.....	۱۳۱
۵. جمع‌بندی مزایای استفاده از ساختار.....	۱۳۳
تمرین.....	۱۳۴
فصل سیزدهم: فایل‌ها.....	۱۳۵
۱. چرا به فایل نیاز داریم؟.....	۱۳۶
۲. فایل چیست؟.....	۱۳۷
۳. کارکردن با فایل.....	۱۳۸
۴. مثال خواندن نمره از فایل.....	۱۳۸
۵. مثال خواندن متن از فایل.....	۱۴۰
تمرین.....	۱۴۲
فصل چهاردهم: اشاره‌گرها.....	۱۴۳
۱. اهمیت و معرفی اشاره‌گرها.....	۱۴۴
۲. ارتباط آرایه‌ها و اشاره‌گرها.....	۱۴۹
۳. آرایه‌های پویا.....	۱۵۱
۴. مثال مرتب کردن دانش‌آموزان در آرایه‌ی پویا.....	۱۵۳
تمرین.....	۱۵۵
ضمیمه (جدول کدهای اسکی).....	۱۵۶

فصل اول : مبانی برنامه نویسی کامپیوتر

۱. اهمیت آموختن برنامه نویسی
۲. از برنامه ها تا صفر و یک
۳. زبان برنامه نویسی C

۱) اهمیت آموختن برنامه‌نویسی

همه‌ی ما برای ادامه‌ی زندگی مجبوریم مسائل پیش پایمان را حل کنیم. البته ممکن است راه حل برخی از مسائل بدیهی به نظر برسد. مثلاً ممکن است تشنه باشید، شاید نیاز پیدا کنید که برای دوست قدیمی‌تان که دلتنگش شده‌اید، نامه‌ای بفرستید یا بخواهید برای آرامش بیشتر، کمی موسیقی گوش دهید. برای رسیدن به اهدافتان می‌توانید از مجرای دنیای واقعی دست به کار شوید. مثلاً می‌توانید پای یخچال بروید و یک لیوان آب بنوشید. یا یک خودکار و صفحه‌ای کاغذ بردارید و شروع به نوشتن نامه کرده و پس از پشت سر گذاشتن مراحل لازم، نامه را پست کنید یا این‌که از یک گروه موسیقی دعوت کنید که در خانه‌ی شما برای تان موسیقی دلخواه‌تان را بنوازن! همانطور که می‌بینید بعضی از این راه‌حل‌های فیزیکی ساده و سرراست، بعضی سخت، و ممکن است برخی از آن‌ها تقریباً غیرعملی باشند. در این میان شاید ما به نسبت اجدادمان خوش‌شانس‌تر بوده باشیم که ابزار قدرتمندی برای حل بسیاری از مسائل، در اختیار داریم: کامپیوتر!

واضح است که در مورد بعضی مسائل، کامپیوتر نمی‌تواند کمکی به شما بکند، مثل خوردن آب. باید خودتان بروید پای یخچال و به میزان دلخواه آب بنوشید. اما گاهی حل مسئله را برای شما ساده‌تر می‌کند، مثلاً شما می‌توانید از سرویس ایمیلی که غول‌های صنعت کامپیوتر، مثل یاهو و گوگل زحمت ایجاد یا به اصطلاح کامپیوتری برنامه‌نویسی آن را کشیده‌اند، استفاده کرده و با زحمت خیلی کم‌تر این نامه را به دوست-تان ارسال کنید. در مورد مثال موسیقی هم که دیگر نیازی به توضیح بیشتر نیست.

کامپیوترها به ما کمک می‌کنند مسائل را راحت‌تر حل کنیم، اما این کار را چطور انجام می‌دهد؟ چطور کامپیوتر می‌تواند آهنگ پخش کند یا نامه‌ی الکترونیکی ما را به دست دوست‌مان برساند؟ علاوه بر نیاز ما به وجود یک بستر سخت‌افزاری برای بعضی از امور (مثل نیاز به بلندگو برای گوش کردن به موسیقی)، به یک شرح دقیق از جزئیات کارهایی که باید توسط کامپیوترها انجام شود، نیاز است. این شرح دقیق را برنامه^۱ نامیم. پس شرکت محترم گوگل نیاز به برنامه‌نویسان توانایی دارد که برنامه‌های مربوط به ارسال موفقیت‌آمیز نامه‌ی الکترونیکی را تولید کنند.



تا این‌جا دیدیم که حل مسائل توسط کامپیوتر نیاز به برنامه‌نویسی دارد و این راه حل‌ها موجب ساده‌تر شدن زندگی ما شده است. خوشبختانه شرکت‌های خیلی زیادی وجود داشته‌اند که از روی خیرخواهی یا برای رسیدن به سود بیشتر، اقدام به برنامه‌نویسی برای حل بسیاری از نیازهای موجود کرده‌اند و در این میان سیستم‌های عامل ویندوز و لینوکس، برنامه‌های تهیه‌ی متن، برنامه‌های پخش موسیقی، شبکه‌های اجتماعی، ویروس‌ها، آتی‌ویروس‌ها و ... پدید آمده‌اند.

در این‌جا ممکن است این سوال به وجود بیاید که با وجود حل شدن همه‌ی این مسائل توسط این شرکت‌های بزرگ و وجود هزاران نرم افزار با کیفیت در بازار، چرا باید همچنان برنامه‌نویسی در دنیا باقی مانده باشد و اصلاً چرا ما باید برنامه‌نویسی را یاد بگیریم؟ برای پاسخ به این سوال، باید به سه نکته توجه کرد:

اول این‌که این برنامه‌های معروف هر روز و هر روز بهتر می‌شوند، همچنین بسته به تغییر کاربری، نیاز است تغییراتی در برنامه‌ها ایجاد شود. پس این شرکت‌ها برای بهبود نرم افزارهای موجودشان و فروش و کسب سود بیشتر همچنان به برنامه‌نویسی (و تبعاً برنامه‌نویسان) نیاز دارند.

¹ Program

نکته‌ی بعدی در این جاست که با پیشرفت روزافزون فن‌آوری، گاهی بسترهایی فراهم می‌شود که راه‌حل‌های جدید و بهتری در اختیار ما قرار می‌دهند، یا بعضاً برخی مسائل که قبلاً ممکن نبودند را ممکن می‌کنند. مثلاً در دهه ۱۹۹۰ که اینترنت رشد کرد و همگانی شد، امکاناتی برای برنامه‌نویسی در حوزه‌های جدید (برنامه‌نویسی وب) فراهم شد که باعث شد این‌بوهی از برنامه‌های جدید در این زمینه تولید شوند.

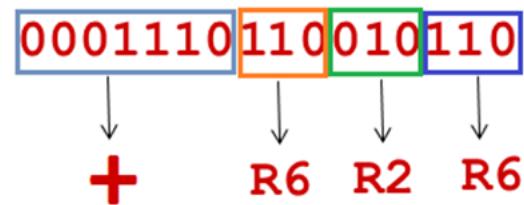
نکته‌ی سوم که بسیار مهم‌تر از دو نکته‌ی قبلی نیز هست این است که شرکت‌های بزرگ تنها نیازهای کلی مشترک بین میلیون‌ها انسان را حل کرده و روانه بازار کرده‌اند. اما برای دسته‌های خاصی از انسان‌ها مثل محققین، هر روز ممکن است مسائل جدید خاص‌منظره‌ای مطرح شود. این جاست که اگر خودتان دست به کار نشده و برنامه‌ی مورد نیاز خودتان را تولید نکنید، مشکل‌تان بی‌راحل باقی می‌ماند. به عنوان مثال ممکن است که در حین تحقیقات‌تان یک رابطه برای اعداد اول پیدا کنید. برای آزمایش این رابطه، باید یک برنامه بنویسید تا صحت آن تایید شود.

۲) از برنامه‌ها تا صفر و یک

حافظه اصلی

حافظه اصلی
00101101
⋮
10100010

همهی کامپیووترها یک حافظه‌ی اصلی دارند که ما آن را با نام **RAM** می‌شناسیم. دستورات برنامه‌ها به ترتیب روی خانه‌های متوالی حافظه قرار می‌گیرند و دستور به دستور، به ترتیب اجرا می‌شوند. پس هر خانه‌ی حافظه حاوی یک دستور از برنامه خواهد بود. اما همان‌طور که می‌دانید، هر خانه از حافظه به صورت دو دویی^۲ می‌باشد، یعنی فقط می‌تواند رشته‌ای از ۰ و ۱ ها را در خود جای دهد. پس باید هر دستور ما هم رشته‌ای از ۰ و ۱ ها باشد. به این ترتیب، باید قراردادی داشته باشیم که هر رشته از ۰ و ۱ ها به چه معنی است. به عنوان مثال، اگر فرض کنیم دستورها ۱۶ بیتی باشند، می‌توانیم قرارداد کنیم که ۷ بیت سمت چپ، نشان دهنده‌ی نوع عملگر^۳ (مثل جمع، تفریق، ضرب و ...) و هر کدام از سه بیت‌های سمت راست آن، نشان دهنده‌ی یک عملوند^۴ (آن‌چه محاسبات روی آن انجام می‌شود) باشد. مثلاً رشته‌ی بیتی مشخص شده در شکل زیر نشان دهنده‌ی دستور $R6 + R2 = R6$ خواهد بود. یعنی محتوای $R6$ با $R2$ جمع شده و حاصل در $R6$ قرار می‌گیرد.



به زبانی که هر دستور آن، با رشته‌ای از صفر و یک نوشته شده، **زبان ماشین**^۵ می‌گوییم. در این زبان هر رشته از صفر و یک‌ها با یک قرارداد مشخص، نشان‌دهنده‌ی یک دستور خاص خواهد بود.

با این حساب شما برای برنامه‌نویسی با زبان ماشین کار سختی پیش‌رو خواهید داشت. اولاً باید قرارداد ماشینی که با آن کار می‌کنید را به خاطر داشته باشید و دستورات‌تان را با آن قرارداد به صورت صفر و یک بنویسید. اگر مشکل کوچکی در برنامه‌تان وجود داشته باشد، مثلاً یک ۱ را به اشتباه ۰ زده باشید، پیدا کردن این اشتباه در این‌بوهی از ۰ و ۱ ها کار واقعاً طاقت‌فرسایی است. از طرفی خواندن و متوجه شدن برنامه‌ای که به زبان ماشین نوشته شده برای ما بسیار مشکل است.

² Binary

³ Operator

⁴ Operand

⁵ Machine Language (ML)

همه‌ی این مشکلات از این‌جا بوجود آمده که زبانی که ما با آن صحبت و فکر می‌کنیم، برای ماشین قابل فهم نیست و متقابلاً زبان ماشین نیز برای ما غیر قابل فهم است، مثل حالی که دو نفر با دو زبان متفاوت می‌خواهند با هم صحبت کنند. نوشتن برنامه به زبان ماشین، شبیه به این است که یکی از طرفین (متاسفانه در این‌جا ما) زحمت یادگرفتن یک زبان جدید را قبول کرده و بعد از یادگیری به سختی و با اشتباهات زیاد، به زبان او صحبت کند.

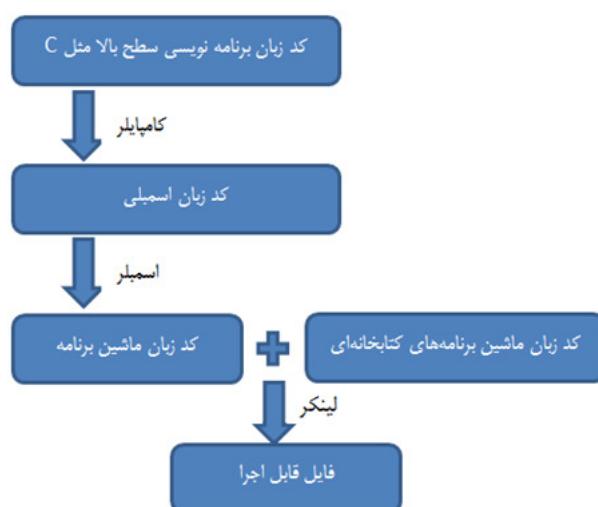
اما راه حل‌های ساده‌تری هم می‌توانند به کار برد شوند. در شرایطی که نیاز به مکالمه‌ی دو نفر با دو زبان متفاوت است، معمولاً از یک مترجم

~~0001110110010110~~

ADD R6, R2, R6

که به زبان هر دو نفر آشنایی دارد، استفاده می‌شود. این راه حل ساده می‌تواند الهام بخش ما برای حل مشکل برنامه‌نویسی به زبان ماشین باشد. ما برنامه‌مان را به جای این که به زبان ماشین بنویسیم، به زبانی نمادین که نزدیک‌تر به زبان خودمان است، می‌نویسیم و زحمت ترجمه را به عهده‌ی یک برنامه‌ی دیگر می‌گذاریم. به عنوان مثال، به جای این که برای عملیاتِ جمع، رشته‌ی 7 بیتی مربوط به آن را بنویسیم از عبارت ADD استفاده می‌کنیم. به این ترتیب نوشتن، خواندن و خطایابی برنامه خیلی ساده‌تر می‌شود. به این زبان که برای انسان قابلیت فهم بیشتری دارد، زبان اسembly⁶ گفته می‌شود و به برنامه‌ی میانجی، که هر خط از برنامه‌ی زبان اسembly را به رشته‌ی معادل صفر و یک در زبان ماشین تبدیل می‌کند، اسembler⁷ گفته می‌شود.

اما هنوز هم شرایط به اندازه‌ی کافی برای ما مطلوب نیست. زبان اسembly یک زبان بسیار سطح پایین است. به این معنی که برای انجام هر کار ساده‌ای، نیاز به چندین دستور است. به عنوان مثال برای چاپ کردن یک عبارت ساده روی مانیتور، ممکن است به بیش از ۱۰ دستور اسembly نیاز باشد. از طرفی زبان اسembly و زبان ماشین، واسطه به ماشینی هستند که روی آن اجرا می‌شوند. یعنی اگر شما برای یک پردازشگر خاص برنامه‌ای به زبان اسembly نوشتید، روی کامپیوترا دیگر با پردازشگری متفاوت، قابل فهم نیست. در عوض ما به زبان‌های برنامه‌نویسی سطح بالایی علاقه‌داریم که نیازمند حجم کمتری از برنامه باشد و قابلیت حمل⁸ داشته باشد، به این معنی که روی کامپیوتراها متفاوت قابل اجرا باشد. به این منظور زبان‌های سطح بالایی مثل C به وجود آمده‌اند.



برای تبدیل برنامه‌ها از زبان‌های سطح بالا مانند C به زبان سطح پایین اسembly، هم نیاز به مترجم دیگری داریم که هر دستور از زبان مبدا را به یک یا احتمالاً چند دستور اسembly تبدیل کند. این کار را هم یک برنامه‌ی مجزا به نام کامپایلر⁹ انجام می‌دهد.

برنامه‌های نوشته شده به زبان C، در فایلی با پسوند C. ذخیره می‌شوند که به این فایل، کد مبدای¹⁰ گفته می‌شود. مرحله‌ی بعدی،

⁶ Assembly Language

⁷ Assembler

⁸ Portability

⁹ Compiler

¹⁰ Source Code

تبديل کد مبدأ به زبان اسمبلي و سپس تبديل آن به کد ماشين است که به آن **کد مقصده**^{۱۱} گفته می‌شود. همان‌طور که قبلاً نیز گفتیم، برای انجام این کارها به ترتیب از کامپایلر و اسمبيلر استفاده می‌شود. کامپایلرهای متعددی برای زبان C توسط شرکت‌های مختلف و برای سیستم‌عامل‌های مختلف نوشته شده است که می‌توانید بر حسب نیاز از هر یک از آن‌ها استفاده نمایید.

اما هنوز برنامه برای اجرا آماده نیست. معمولاً برنامه‌نویسان در برنامه‌های خود از یک سری کدهای از پیش آماده شده، برای انجام عملیات متداول (مانند محاسبه جذر یک عدد و یا سینوس یک زاویه) استفاده می‌کنند، که برنامه‌ی آن‌ها قبلاً به بهترین نحو ممکن نوشته و ترجمه شده است. این برنامه‌ها یا در قالب کتابخانه‌های استاندارد توسط شرکت‌های ارائه‌کننده نرم افزار عرضه شده است و یا توسط دیگر همکاران برنامه‌نویس نوشته و در اختیار ما قرار داده شده است. در این مرحله، باید کد مقصده برنامه‌ی اصلی با کدهای مربوط به این برنامه‌های کمکی پیوند زده شود. برای این کار نیاز به یک **پیوند زننده**^{۱۲} داریم، نتیجه‌ی این کار پیوند زننده، یک فایل قابل اجرا خواهد بود. (در ویندوز این فایل پسوند **exe** خواهد داشت).

۳) زبان برنامه‌نویسی C



این زبان برنامه‌نویسی در سال‌های اولیه دهه ۱۹۷۰، توسط دنیس ریچی در آزمایشگاه-های بل تولید شد. ریچی C را با الهام‌گیری از زبان برنامه‌نویسی B، که خود از زبان دیگری به نام BCPL مشتق شده بود، طراحی کرد و دلیل نام‌گذاری آن هم همین است! هدف عمدۀ از طراحی زبان C، به کارگیری در نوشتن برنامه‌ی سیستم عامل بوده و به همین دلیل طوری طراحی شده که بسیار بهینه و سریع باشد. در تمام تصمیم‌گیری‌های زبان C ردپای فلسفه‌ی سرعت و بهینگی اجرا را می‌توان دید. علی‌رغم این هدف گذاری، زبان C به دلیل طراحی مناسب، در بسیاری از کاربردهای دیگر نیز به کار رفته و امروزه به عنوان پرطرفدارترین زبان از زمان آغاز برنامه‌نویسی شناخته می‌شود. در سال ۱۹۸۳ زبان C++ پایه زبان C و با اضافه کردن چند قابلیت جالب دیگر از جمله **تئسی گروابی**^{۱۳} بوجود آمد.

یکی دیگر از ویژگی‌های زبان برنامه نویسی C، قابلیت حمل آن است. تقریباً برای همه‌ی انواع ماشین‌ها و انواع سیستم‌های عامل، کامپایلر C وجود دارد و برنامه‌ی نوشته شده به زبان C روی یک پلتفرم خاص، با تغییراتی اندک روی پلتفرمی دیگر قابل اجرا می‌باشد.

در عمل برای تولید برنامه، شما می‌توانید متن کد برنامه را با یک نرم‌افزار ویرایش متن مثل Notepad تولید کرده و جداگانه توسط یک کامپایلر، کامپایل کنید. اما برای C محیط‌های تخصصی متنوعی برای برنامه‌نویسی وجود دارد که به آنها IDE^{۱۴} می‌گوییم. این محیط‌ها معمولاً تسهیلاتی برای توسعه‌ی متن برنامه، کامپایل کردن و اشکال زدایی برنامه در اختیار ما قرار می‌دهند. از جمله‌ی این IDE‌ها می‌توان Microsoft Visual Studio و Dev-CPP، Code Blocks، Borland C پیش‌فرض ما استفاده از محیط Dev-CPP خواهد بود.

¹¹ Object Code

¹² Linker

¹³ Object Oriented

¹⁴ Integrated Development Environment

تمرین



۱. به نظر شما، چرا آن‌چه که ما در حافظه نگه‌داری می‌کنیم، به صورت صفر و یک است؟ به عنوان مثال، چرا به جای این‌که هر واحد حافظه بیت (صفر یا یک) باشد، یک رقم در مبنای ده نیست؟
۲. هر کدام از زبان‌های برنامه‌نویسی سطح بالا، با یک هدف و فلسفه‌ای ایجاد شده‌اند. برای حداقل ۵ زبان سطح بالا، این مورد را بررسی کرده و گزارش کنید.
۳. ۳ نوع معروف از خطاهای برنامه‌نویسی موسوم به **خطای نحوی^{۱۵}**، **خطای زمان اجرا^{۱۶}** و **خطای منطقی^{۱۷}** هستند، در مورد هر کدام از ۳ نوع بالا، تحقیق کرده و حاصل را گزارش کنید.
۴. روی کامپیوتر خود نصب کرده و از گردش در محیط و منوهای آن لذت ببرید!

¹⁵ Syntax Error

¹⁶ Run Time Error

¹⁷ Logic Error

فصل دوم: الگوریتم، ایده‌ی حل مسئله

۱. الگوریتم چیست؟
۲. مثال شوت زدن
۳. نمایش الگوریتم
۴. مثال شوت زدن (فلوچارت)
۵. مثال درست کردن چای
۶. مثال درست کردن چای (فلوچارت)
۷. مثال نمایش عدد بزرگ‌تر
۸. مثال نمایش عدد بزرگ‌تر (فلوچارت)
۹. مثال نمایش بزرگ‌ترین عدد
۱۰. مثال نمایش بزرگ‌ترین عدد (فلوچارت)

۱) الگوریتم چیست؟

دانستن صرف یک زبان برنامه‌نویسی برای نوشتن برنامه برای حل یک مسئله کافی نیست. این قضیه مثل این است که شما به زبان فرانسه مسلط باشید اما جمله‌ی مناسبی در ذهن نداشته باشید که به این زبان بیان کنید. همان‌طور که مسئله‌ی اصلی برای انجام یک سخنرانی تاثیرگذار داشتن حرف‌ها، ایده‌ها و مفاهیم تاثیرگذار است و تبدیل کردن این مفاهیم به جملات معمولاً کار پیچیده‌ای نیست، برای تولید یک برنامه‌ی موثر و مناسب هم مسئله‌ی اصلی داشتن یک الگوریتم^{۱۸} مناسب برای حل آن در ذهن است. الگوریتم به زبان ساده، راه حلی قدم به قدم برای حل یک مساله می‌باشد. از دیدگاهی دیگر می‌توان به الگوریتم به عنوان ابزاری نگاه کرد که با دریافت تعدادی ورودی، خروجی دلخواه را به ما ارائه می‌کند.

هر الگوریتم باید دارای سه خاصیت باشد:

- **عدم ابهام:** هر یک از قدم‌ها باید بدون ابهام و برای کامپیوتر تعریف شده باشند. به عبارت دیگر هر قدم به معنای استفاده از یکی از دستورات کامپیوتر است، که این دستورات از پیش برای کامپیوتر تعریف شده‌اند.
- **گام‌های متناهی:** الگوریتم باید شروع و پایان مشخصی داشته باشد و در تمامی حالات پس از اجرای تعداد متناهی از دستورات پایان یابد.
- **خروچی:** هر الگوریتم باید حداقل یک خروچی داشته باشد، که خروچی در واقع همان نتایج به دست آمده از حل مسأله می‌باشد.

نکته‌ی فنی

الگوریتم می‌تواند هیچ یا چند ورودی داشته باشد، یعنی اجباری برای وجود ورودی وجود ندارد.

نکته‌ی تفننی

الگوریتم از نام ابو جعفر محمد بن موسی خوارزمی (الخوارزمی) ریاضیدان و ستاره‌شناس بزرگ ایرانی آمده است. در قرون وسطی کتاب‌های خوارزمی و به خصوص کتاب جبر او، شهرت فراوانی نزد ریاضیدانان اروپا پیدا کرد. در قرن ۱۲ میلادی کتاب او در مورد اعداد هندی به زبان لاتین ترجمه شد، که مترجم از نام «الگوریتمی» برای او در زبان لاتین استفاده کرد و به مرور زمان و در قرن ۱۷ نام او در زبان فرانسه به الگوریتم تغییر یافت. به نظر من علاقه + تلاش = جاودانگی! نظر شما چیست؟

برای آشنایی بیشتر با مفهوم الگوریتم، مثال زیر را در نظر بگیرید که با انجام تعدادی قدم ابتدائی با ترتیبی مشخص، هدف مورد نظر برآورده می‌شود

۲) مثال شوت زدن

الگوریتمی برای شوت زدن به یک توپ با فرض تعریف شده بودن دستورات زیر برای کامپیوتر بنویسید.

¹⁸ Algorithm

توب را پیدا کن، پایت را پشت توب قرار بده، شوت بزن، به سمت توب بچرخ، به میزان دلخواه حرکت کن، فاصله‌ات با توب را بستج و ذخیره کن، شروع، پایان.

۱- شروع.

۲- توب را پیدا کن.

۳- به سمت توب بچرخ.

۴- فاصله‌ات با توب را بستج و ذخیره کن.

۵- به میزان فاصله حرکت کن.

۶- پایت را پشت توب قرار بده.

۷- شوت بزن.

۸- پایان

همان طور که می‌بینید دستورات برنامه دارای ترتیب می‌باشند، و پس از اجرای هر قدم، قدم بعدی انجام می‌شود.

۳) نمایش الگوریتم

فرض کنید الگوریتمی برای حل یک مسئله به ذهن تان خطور کرده و شما قصد دارید این ایده را با افراد دیگر در میان بگذارید. یک راه برای انجام این کار پیاده‌سازی این الگوریتم به یکی از زبان‌های برنامه‌نویسی مثل C و نشان دادن آن به افراد مورد نظر است. این کار به چند دلیل کار عاقلانه‌ای نیست:

- شما باید به زبان برنامه‌نویسی آشنایی داشته باشید
- نوشتن برنامه به زبان برنامه‌نویسی کار نسبتاً پردردسری است. به این علت که ما باید برنامه را به زبان کامپیوتر بنویسیم که تا حدی با زبان مورد استفاده‌ی ما فرق دارد.
- به همان دلیل بالا، خواندن این برنامه سخت است.
- ممکن است تعدادی از مخاطبین کلأاً زبان‌های برنامه‌نویسی یا حداقل زبان برنامه‌نویسی نمایش شما، آشنایی نداشته باشند.

برای بیان الگوریتم به طریقی که خوانایی آن برای دیگر افراد به سهولت می‌سر باشد، می‌توان از چند روش ساده‌تر و عاقلانه‌تر استفاده کرد، که دو روش رایج برای این کار را در ادامه می‌بینید:

- ۱- **شبه برنامه**^{۱۹}: شبه برنامه، زبانی بین زبان انسان و زبان ماشین است که اولاً خوانایی بیشتری نسبت به دستورات کامپیوتر دارد و به نوعی به زبان انسان‌ها نزدیک‌تر می‌باشد و در عین حال، مانند دستورات کامپیوتر، بدون ابهام و دقیق است.

مثال ضرب کردن

الگوریتمی بنویسید که دو عدد از کاربر دریافت کرده، در یکدیگر ضرب کند و حاصل را نمایش دهد.

¹⁹ Pseudo-Code

۱- شروع

۲- دو عدد را از کاربر دریافت کن.

۳- عدد اول را در عدد دوم ضرب کن.

۴- حاصل را ذخیره کن.

۵- حاصل را نمایش بده.

۶- پایان.

سوال: برای کامپیوتری که برنامه‌ی بالا را اجرا می‌نماید، چه دستوراتی باید تعریف شده باشد؟

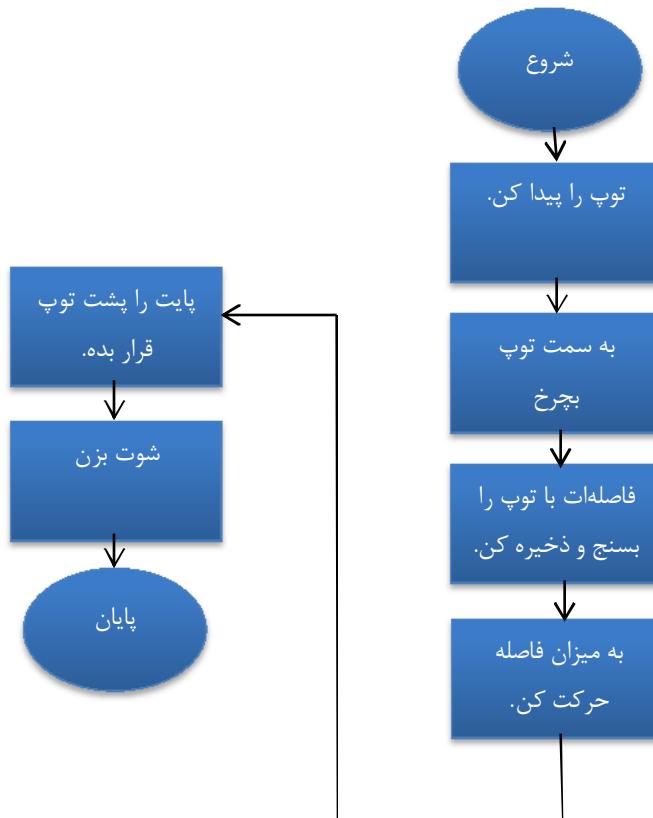
۲- **فلوچارت:** نمایشی گرافیکی از الگوریتم با هدف خوانایی بیشتر آن است، که هر دستور را توجه به نوع آن، درون شکل هندسی مربوط به آن نوشته شده و این اشکال با خطوطی به یکدیگر متصل می‌شوند، که این خطوط ترتیب اجرای دستورات را نمایش می‌دهند. انواع اشکال هندسی مورد استفاده در رسم فلوچارت را در ادامه مشاهده می‌کنید:



²⁰ Flowchart

۴) مثال شوت زدن (فلوچارت)

برنامه‌ای که برای شوت زدن نوشته بودید را به وسیله‌ی فلوچارت نمایش دهید.



در مثال شوت زدن با نمونه‌ای ساده از الگوریتم آشنا شدیم. در این مثال دستورات شرطی و یا دریافت ورودی و خروجی وجود نداشت و به نوعی همه‌ی دستورات محاسباتی بودند. در مثال زیر با دستورات شرطی آشنا می‌شونیم:

۵) مثال درست کردن چای

الگوریتمی (شیه برنامه) برای درست کردن چای با فرض تعریف شده بودن دستورات زیر برای کامپیوتر بنویسید.

یک قاشق چای در قوری بریز، منتظر بمان، قوری را بر روی کتری قرار بده، آب کتری را در قوری بریز، درون کتری آب بریز، گاز را روشن کن، گاز را خاموش کن، دما را بسنجد، شروع، پایان.

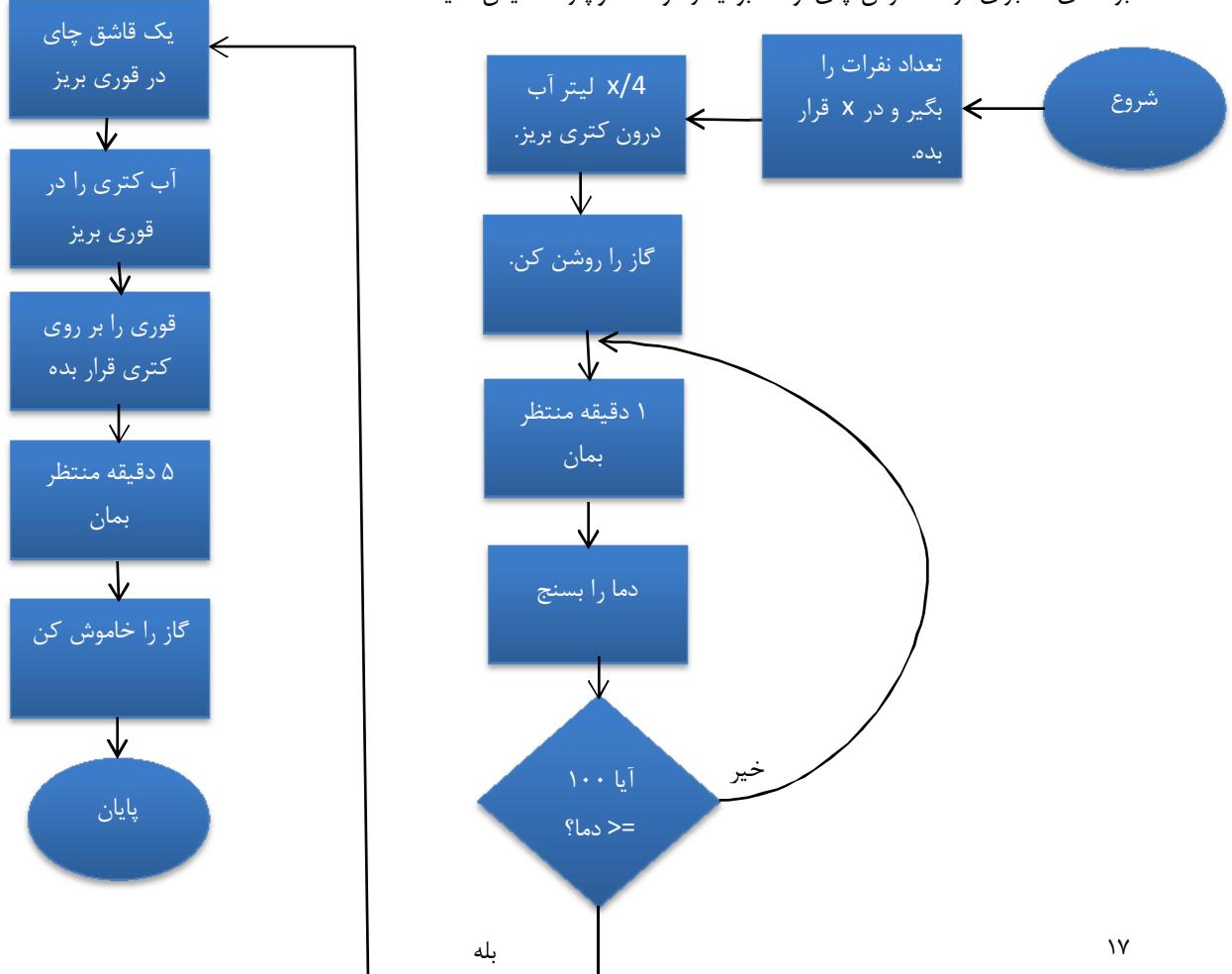
- شروع.
- تعداد نفرات را بگیر و در X قرار بده.
- $X/4$ لیتر آب درون کتری آب بریز.
- گاز را روشن کن.

- ۵ ۱ دقیقه منتظر بمان.
- ۶ دما را بسنج.
- ۷ آیا دما به 100° درجه رسیده است؟
- ۸ یک قاشق چای در قوری بریز.
- ۹ آب کتری را در قوری بریز.
- ۱۰ قوری را بروی کتری قرار بده.
- ۱۱ ۵ دقیقه منتظر بمان.
- ۱۲ گاز را خاموش کن.
- ۱۳ پایان.

همان طور که در خط ۶ می‌بینید، در صورتی که دما پائین‌تر از 100° باشد، روند اجرا عوض شده و برنامه به خط ۵ برمی‌گردد، و در غیر این صورت، روند عادی برنامه طی می‌شود. این نوع دستور که در آن با بررسی یک شرط و با توجه به شرایط مساله، یکی از دو روند اجرا انتخاب می‌شود، **دستور شرطی** نامیده می‌شود. نحوه استفاده از این نوع دستور در فلوچارت را در مثال زیر می‌بینید.

۶) مثال درست کردن چای (فوچارت)

برنامه‌ای که برای درست کردن چای نوشته بودید را توسط فلوچارت نمایش دهید.



نکته‌ی فنی

همان‌طور که در مثال بالا مشاهده می‌کنید، هنگامی که به دستور شرطی ($دما = 100$ می‌رسیم)، با توجه به شرایط موجود، به مرحله‌ی مورد نظر (در صورت پائین‌تر بودن از ۱۰۰) به مرحله‌ی انتظار به مدت ۱ دقیقه و در غیر این صورت به مرحله‌ی ریختن آب کتری در قوری) منتقل می‌شویم.

مثال‌های ذکر شده تا این‌جا، با دستوراتی فرضی برای ماشینی که از توانایی‌های سطح بالایی برخوردار است، نوشته شده بود. اما رایانه‌ای که ما در اختیار داریم، دارای این قابلیت‌ها نمی‌باشد و باید با زبانی ساده‌تر با آن صحبت نمود. در ادامه مثالی شبیه به زبان رایانه‌های امروزی می‌بینیم.

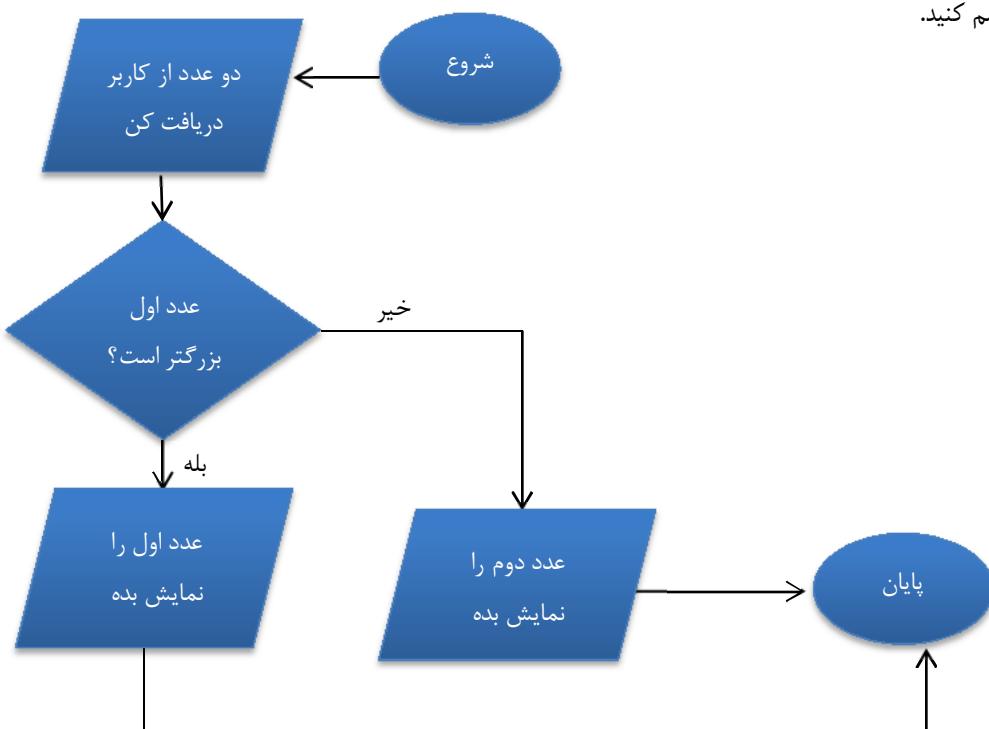
۷) مثال نمایش عدد بزرگ‌تر

شبه‌برنامه‌ای بنویسید که دو عدد را از کاربر دریافت کرده، عدد بزرگ‌تر را نمایش دهد.

- ۱- شروع
- ۲- دو عدد را از کاربر دریافت کن.
- ۳- آیا عدد اول بزرگ‌تر است؟
بله: عدد اول را نمایش بده.
خیر: عدد دوم را نمایش بده.
- ۴- پایان.

۸) مثال نمایش عدد بزرگ‌تر (فلوچارت)

برای مثال قبل فلوچارت رسم کنید.



۹) مثال نمایش بزرگترین عدد

شبه برنامه‌ای بنویسید که ۱۰ عدد دریافت کرده، بزرگترین آنها را نمایش دهد.

- ۱- شروع
- $i = 1$ -۲
- ۳- یک عدد دریافت کن و در a ببریز.
- ۴- یک عدد دریافت کن و در b ببریز.
- ۵- آیا $a < b$ بله: ؟ آیا $i > 10$ برو: ؟
- ۶- $i = i + 1$
- ۷- آیا $i > 10$ برو: ؟

خیر: برو: ۹.

بله: برو به: ۴.

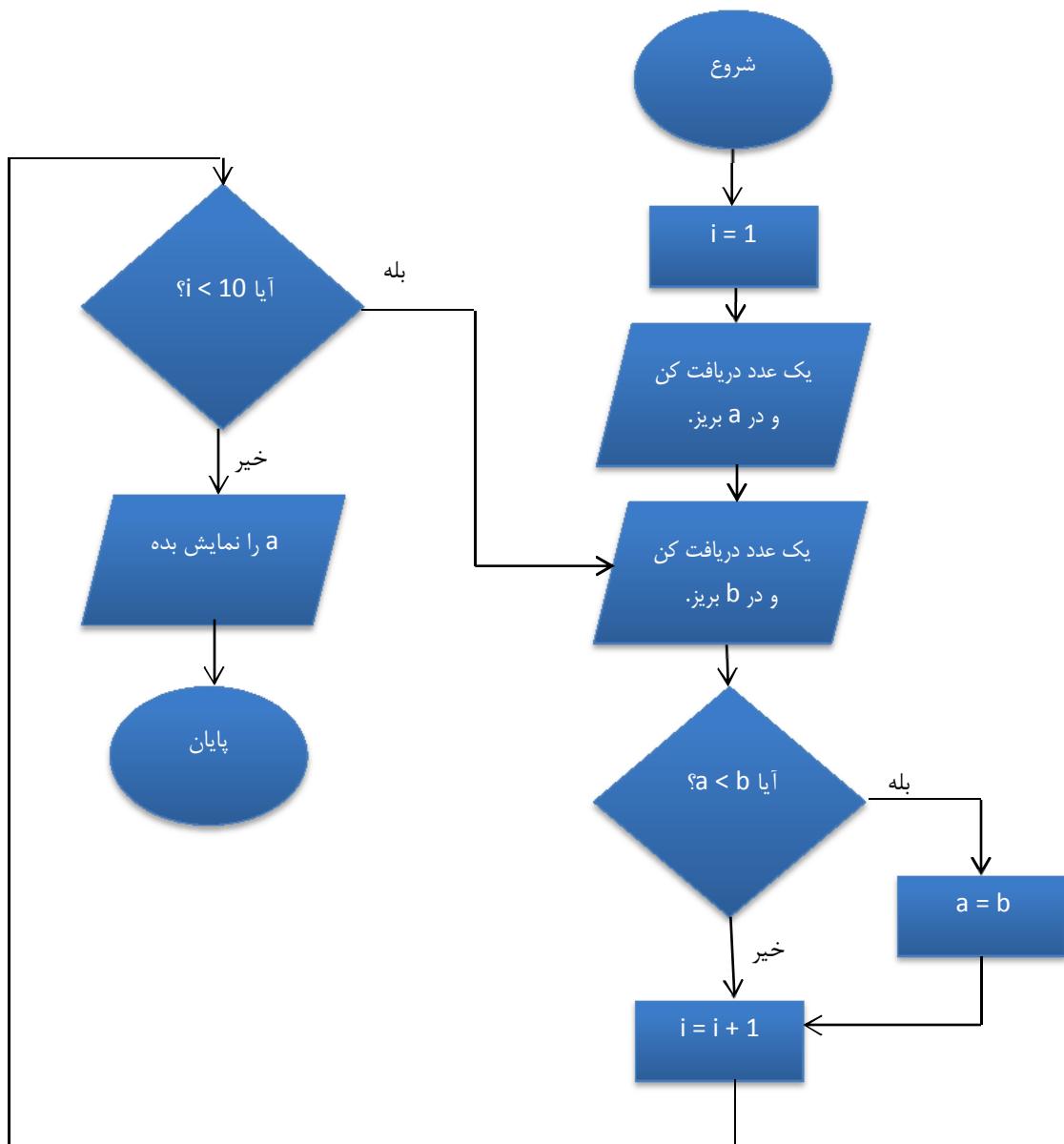
- ۸- a را چاپ کن.
- ۹- پایان.

الگوریتمی که در بالا آمده است، ابتدا دو عدد دریافت کرده، با یکدیگر مقایسه می‌کند و عدد بزرگتر را در a نگهداری می‌کند. سپس در مراحل بعدی، در هر مرحله یک عدد دریافت می‌شود و در b ریخته می‌شود، این عدد جدید با مقدار a که بزرگترین عدد تاکنون در آن ریخته شده است، مقایسه می‌شود و در صورت بزرگتر بودن عدد جدید مقدار آن در a ریخته می‌شود. این کار تا جایی که ۱۰ عدد را بخوانیم تکرار می‌شود. برای این که بفهمیم ۱۰ عدد را از دریافت کرده‌ایم یا خیر از متغیر i استفاده کرده‌ایم، پس از هر بار خواندن عدد جدید این متغیر یکی افزایش می‌یابد و وقتی که مقدار i به ۱۰ رسید یعنی ۱۰ عدد را خوانده‌ایم و کار ما به پایان رسیده است. که در این صورت مقدار a بر روی صفحه‌ی نمایش داده شده و برنامه پایان می‌یابد.

توجه کنید که در خط ۵ شرط **خیر** ذکر نشده است، بنابراین برنامه به خط ۶ می‌رود، در صورت برقرار بودن شرط نیز پس از انجام دستور $a = b$ برنامه به خط ۶ خواهد رفت.

۱۰) مثال نمایش بزرگترین عدد (فلوچارت)

فلوچارت شبه برنامه‌ای که برای نمایش بزرگترین عدد نوشته بودید را رسم کنید.



تمرین



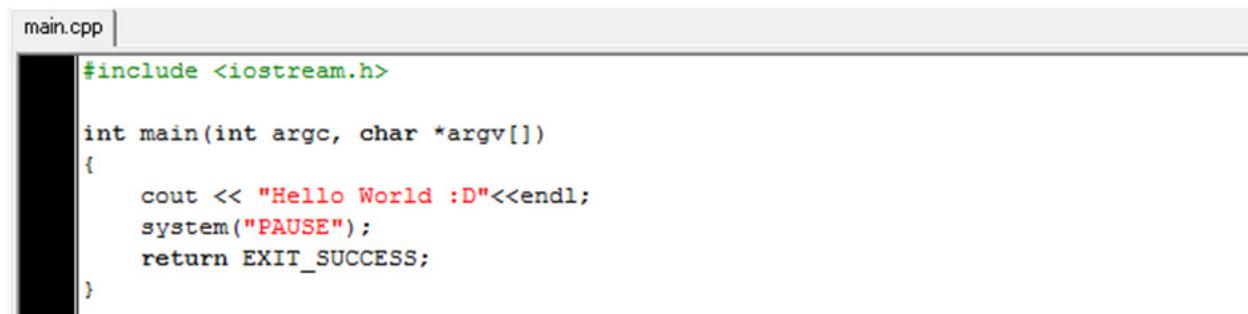
۱. شبهبرنامه و فلوچارتی بنویسید که ۱۰ عدد دریافت کرده، میانگین آن اعداد را نمایش دهد.
۲. شبهبرنامه و فلوچارتی بنویسید که عددی دریافت کرده و بگوید که عدد اول است یا خیر.
۳. شبهبرنامه‌ای بنویسید که با استفاده از شبهبرنامه‌ی سوال بالا، تجزیه‌ی یک عدد به عوامل اول را بدست بیاورد.
۴. به عنوان مثال $3^2 * 2^3 = 72$. توجه کنید که نیازی نیست شبهبرنامه‌ی سوال ۲ را دوباره بنویسید.
۵. استفاده از حل آماده‌ی یک مسئله برای حل مسئله‌ی دیگر، مثل آن‌چه در سوال ۳ انجام دادید، چه مزایایی دارد؟
۶. شبهبرنامه و فلوچارتی بنویسید که دو عدد دریافت کرده و اعداد صحیح بین آن دو عدد را نمایش دهد.
۷. شبهبرنامه و فلوچارتی بنویسید که عددی از ورودی دریافت کرده، توان عامل دوی آن عدد را نمایش دهد. (به عنوان مثال توان عامل دوی ۴۸، ۴ است)
۸. شبهبرنامه و فلوچارتی بنویسید که عددی دریافت کرده، فاکتوریل آن را محاسبه کرده و نمایش دهد.

فصل سوم: ورودی و خروجی، متغیرها

۱. اولین برنامه‌ی ما – **Hello World!**
۲. معرفی متغیرها و گرفتن مقادیر از کاربر
۳. محاسبه‌ی میانگین دو عدد صحیح
۴. جابجایی دو متغیر
۵. کمی در باره‌ی کاراکترها

Hello World! ما – (۱)

تا این جای کار، با مفاهیم اولیه‌ی برنامه‌نویسی آشنا شده‌اید و موقع آن رسیده که اولین برنامه را بنویسید. شاید تمام برنامه‌نویس‌های معروف دنیا، با سلام کردن به دنیا کار خود را شروع کرده‌اند. حال شما هم باید این برنامه را بنویسید.



```
main.cpp |  
#include <iostream.h>  
  
int main(int argc, char *argv[]){  
    cout << "Hello World :D" << endl;  
    system("PAUSE");  
    return EXIT_SUCCESS;  
}
```

شاید در نگاه اول، برنامه‌ی بالا خیلی طولانی و پیچیده به نظر برسد. در حالی که ما تنها یک خط برنامه نوشته‌ایم. ابتدا به اولین خط بعد از آکولاد می‌پردازیم:

اولین دستوری که در این کتاب می‌آموزید، دستور خروجی `cout` است. اما این دستور چگونه کار می‌کند؟ در این ساختار هر نوشته‌ای که بین " " باشد، عیناً در خروجی چاپ می‌شود. در قسمت بعد که با علامت <> جدا شده است، کلمه کلیدی `endl` آمده است که این دستور باعث می‌شود، بعد از چاپ جمله‌ی `Hello world` به ابتدای خط بعد برویم. با نگاه به خروجی برنامه‌ی فوق این موضوع را بهتر درک خواهید کرد.



اما ما فقط یک جمله چاپ کرده‌ایم، پس خط دوم چیست؟ برای اینکه پاسخ این سوال را پیدا کنید، ابتدا باید بدانید بقیه‌ی خطوط برنامه چیست؟

خط اول مربوط به **کتابخانه‌های زبان برنامه‌نویسی C** هست. شما باید قبل از این‌که از دستوری استفاده کنید، به کامپایلر بگویید که این دستور مربوط به کدام کتابخانه است. اما چرا این کار را باید انجام دهید؟ به مثال زیر توجه کنید:

شما مقاله‌ای (تحقیقی) نوشته‌اید که باید در انتهای آن، مراجع خود را معرفی کنید. این لیست به خواننده‌ها کمک می‌کند که بدانند هر بخش نوشته شده، چه مرجعی داشته است. در برنامه‌نویسی هم شبیه به همین اتفاق می‌افتد. شما به کامپایلر می‌گویید که فلان دستور مربوط به کدام بخش است که بتواند به اطلاعات مربوط به آن دستور دسترسی پیدا کند. اگر آن را معرفی نکنید، کامپایلر به این اطلاعات دست نمی‌باید و باعث اجرا نشدن برنامه شما می‌شود. ممکن است این سوال برای تان پیش بیاید که چرا به صورت پیش فرض، همه‌ی کتابخانه‌ها مورد جستجو قرار نمی‌گیرند؟ جواب سوال شما دو قسمت دارد. قسمت اول این است که تعداد این کتابخانه‌ها خیلی زیاد است و جستجو در تمام آنها زمان و حجم زیادی نیاز دارد که هر دوی این پارامترها در برنامه‌نویسی به شدت مهم هستند. اما قسمت دوم جواب که مهم‌تر هم هست، را وقتی خواهید فهمید که با مفهوم **کلاس (class)** آشنا شده باشید.

حال به قسمت اصلی برنامه (main) می‌رسیم که مهتم‌ترین بخش از برنامه است. این قسمت که به بدنی اصلی برنامه معروف است، تقریباً همیشه به همین شکل است. اما این دستورات که پشت‌سرهم آمده‌اند، چه کاری انجام می‌دهند؟ برای دانستن پاسخ این سوال باید تا پایان فصل تابع صبر کنید.

بین دو آکولاد قسمتی است که باید دستورات مورد نیازتان را وارد کنید تا با اجرای برنامه به اهدافتان برسید. cout که توضیح داده شد نوبت به دو خط انتهایی برنامه می‌رسد. اگر یادتان باشد، سوال ابتدای فصل هنوز بدون پاسخ مانده است، که خط دوم خروجی چیست؟ دستور system("PAUSE") مسئول چاپ جمله‌ی... Press any key to continue... است. این دستور با چاپ این جمله، از پایان یافتن برنامه جلوگیری می‌کند، تا کاربر بتواند خروجی برنامه را دیده و وقتی خوب متوجه آن شد، با فشاردادن یک کلید برنامه را تمام کند. در کل این دستور خروجی برنامه را تا فشردن یک کلید نگه می‌دارد

اما هرگاه هنگام اجرا به خط آخر برسیم، برنامه تمام می‌شود. در هر جای برنامه که باشیم، این دستور نشان دهنده انتهای برنامه است. پس حواس‌تان باشد که در کجا برنامه از این دستور استفاده می‌کنید.

اولین برنامه را نوشتیم. اما چگونه می‌خواهید برنامه را اجرا کنید؟ کلید F9 وظیفه اجرای برنامه را به عهده دارد. اگر این کلید را فشار دهید، می‌بینید که یک صفحه سیاه رنگ باز می‌شود که منتظر فشار یک کلید است و زمانی که این اتفاق می‌افتد صفحه سیاه بسته شده و به محیط اصلی Dev-C برمی‌گردد.

خط مربوط به cout در برنامه‌ی فوق را می‌توان به صورت زیر هم نوشت، بدون آنکه در خروجی برنامه تغییری ایجاد شود

```
cout<<"hello ";
cout<<"world" <<endl;
```

نکته‌ی فنی

در استفاده از endl دقت کنید! همان‌طور که می‌بینید در خط اول از این دستور استفاده نشده است. چرا که هرگاه به این دستور برسیم، به ابتدای خط بعد می‌رویم، پس اگر در خط اول هم از endl استفاده کرده بودیم، این دو کلمه در دو خط متوالی چاپ می‌شد.

۲) معرفی متغیرها و گرفتن مقادیر از کاربر

در مقابل دستور خروجی (cout) دستور دیگری (cin) هم هست که وظیفه‌ی دریافت اطلاعات از کاربر را به عهده دارد. به این دستورات، دستورات ورودی می‌گویند. اما قبل از این که وارد این مبحث شویم، باید با متغیر^{۲۲} ها آشنا شویم. در دنیای واقعی برای نگهداری آب، نخود و یا هر چیز دیگری نیاز به ظرف‌هایی با شکل و اندازه‌ی مشخص داریم. در برنامه‌نویسی هم ما همین ظرف‌ها را داریم که به آنها متغیر گفته می‌شود. در ابتدای کار ما با ^{۲۳} نوع از این ظرف‌ها آشنا می‌شویم. این چهار نوع بیشترین کاربرد را برای ما دارند:

²² Variable

- ظرف اول یا همان نوع متغیر اول، **int** نام دارد که در آن مقادیر صحیح قرار می‌گیرند.
- نوع دوم **float** است که مقادیر اعشاری را در خود جای می‌دهد.
- نوع سوم **char** است که تنها یک حرف را در خود جای می‌دهد
- نوع آخر **bool** است که دو حالت دارد یا **true** (صحیح) و یا **false** (غلط).

با برخی از انواع این نوع متغیرها آشنا شدیم. اما توجه کنید که هر نوع بازه‌ی مشخصی دارند. به جدول زیر نگاه کنید:

جنس متغیر	تعداد بیت در حافظه	بازه
Int	۳۲	-2,147,483,648 تا +2,147,483,647
Char	۸	0 تا 255
float	۳۲	10 ^{-۳۷} تا 10 ^{+۳۷}
Bool	۱	false یا true

در حالت کلی برای تعریف متغیر، ابتدا نوع متغیر را نوشته و سپس نام متغیر می‌آید. مثلاً برای تعریف متغیری به نام **nomre** از نوع **int** نویسیم:

```
int nomre;
```

نکته‌ی فنی

شما در نام‌گذاری متغیرها آزاد هستید، اما چند محدودیت وجود دارد که باید در نظر گرفته شود:

۱. اسم متغیر تنها می‌تواند شامل حروف انگلیسی (بزرگ یا کوچک)، اعداد و آندرلاین (_) باشد. مثلاً استفاده از نام **a\$X** مجاز نمی‌باشد.
۲. استفاده از اعداد در حرف اول قابل قبول نیست اما استفاده از حروف انگلیسی و آندرلاین در حرف اول مجاز می‌باشد. مثلاً **2X** یک نام نامعتبر و **X25** نامی معتبر است.
۳. زبان برنامه‌نویسی **C** به حروف بزرگ و کوچک حساس است. این جمله بدین معنا است که **nomre** با **Nomre** فرق دارد و دو متغیر متفاوت در نظر گرفته می‌شود. (توجه شود این قانون، شامل کلمات کلیدی نیز می‌شود و این یعنی **int** با **INT** فرق دارد. کلمات کلیدی با حروف کوچک باید نوشته شوند).

اخلاق برنامه‌نویسی



برای نام‌گذاری متغیرها، علاوه بر چند قاعده‌ی فنی که باید رعایت کنید و در نکته‌ی فنی بالا ذکر شد، یک سری قرارداد بین برنامه‌نویسان وجود دارد که رعایت آن‌ها، کیفیت برنامه‌ی شما از جنبه‌های متفاوتی بالاتر می‌برد. شاید بتوان این قراردادها را به نوعی بخشی از اخلاق خوب برنامه‌نویسی دانست! برخی از این قراردادها در ادامه آمده است:

- حتماً نام متغیرها را با مسمی انتخاب کنید، به این معنی که نام متغیر، معرف کارکردی باشد که برای برنامه خواهد داشت. به عنوان مثال اگر قرار است متغیری برای نگهداری سن یک فرد داشته باشید، به جای این که آن را `x` یا `sdjfa` (!) بنامید، بهتر است از نامهای گویا بی مثلاً `age` یا `sen` استفاده کنید.
- متغیرهایی که از یک کلمه تشکیل شده‌اند را تماماً با حروف کوچک نام‌گذاری کنید. مثلاً استفاده از `age` به جای `Age` اشکالی ندارد که نام توصیف‌کننده‌ی متغیرتان، از چند کلمه تشکیل شده باشد. برای خوانایی بیشتر در این حالت، کلمات دوم به بعد را با حرف بزرگ شروع کنید. مثلاً برای نگهداری تعداد دانش‌آموزان به جای `numberofstudents` استفاده کنید که خواناتر است.
- سعی کنید همیشه و در همه‌ی جنبه‌های زندگی‌تان، خوش اخلاق باشید!

اما چگونه این متغیرها را مقداردهی کنیم؟ این کار به چندین روش ممکن است، که فعلاً سه روش معمول آن را می‌بینیم.

۱. مقداری مشخص را به آن منتسب کنیم. (در این مثال مقدار `a` برابر با ۲۰ می‌شود)

```
a = 20;
```

۲. مقدار یک متغیر را در یک متغیر دیگر بریزیم. (چون مقدار `b` برابر با ۲۰ است مقدار `a` هم ۲۰ می‌شود)

```
int a, b;
b = 20;
a = b;
```

خطاهای معمول برنامه نویسی



گفتیم می‌توانید مقدار یک متغیر را در متغیر دیگر بریزید، اما اگر این متغیرها هم نوع نباشند چطور؟ مثلاً مقدار متغیری از نوع `float` را در متغیری از نوع `int` بریزید. در اینجا کامپایلر اعلام خطای نمی‌کند و برنامه برای شما اجرا می‌شود، اما این ممکن است منجر به نادرستی برنامه شما شود. به این نوع خطاهای که موجب جلوگیری از اجرای برنامه نمی‌شوند، اما باعث نادرستی خروجی برنامه می‌شوند، خطای منطقی^{۲۳} گفته می‌شود. به عنوان مثال اگر متغیر `pi=3.14` باشد، و آن را در `p` بریزید که از نوع `int` است، مقدار ۳ خواهد گرفت (چون قابلیت نگهداری اعشار را ندارد)، و به این ترتیب اگر محاسبات مربوط به محیط دایره را با متغیر `p` انجام دهیم، جواب درستی به دست نخواهد آمد.

۳. گرفتن مقدار از کاربر به کمک دستور ورودی `cin`. در این روش، ابتدا دستور `cin` و علامت مخصوص این دستور (`<>`) و سپس نام متغیری که قرار است مقداردهی شود، می‌آید.

²³ Logical Error

```
main.cpp | 
#include <iostream.h>

int main(int argc, char *argv[])
{
    int a;
    cin>>a;
    cout<<a<<endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

در ابتدا متغیری از نوع اعداد صحیح و با نام `a` تعریف کردیم، سپس مقداری صحیحی از کاربر گرفتیم و آن را در متغیر `a` نگهداری کردیم و آن گاه مقداری را که در `a` بود چاپ کردیم، به خروجی برنامه نگاه کنید:

```
10
10
Press any key to continue . . .
```

کاربر مقدار `10` را وارد کرده و برنامه هم همان مقدار را چاپ کرده است.



خطاهای معمول برنامه نویسی

یکی از خطاهایی که در شروع برنامه نویسی بارها دیده شده که آن را انجام می‌دهید، اشتباه کردن علامت‌های جدا کننده‌ی دستورهای `cin` و `cout` می‌باشد. یادتان باشد که علامت این دو به این شکل است:

```
cin>>a •
cout<<a •
```

نکته‌ی فنی

اگر بادتان باشد، وقتی اولین جمله را چاپ کردیم، `Hello World` را بین علامتهای `""` قرار دادیم. اما در برنامه‌ی بالا، بدون علامت‌های `""` استفاده شد. تفاوت چیست؟ برنامه‌ی بالا را این بار با استفاده از `""` مجدداً اجرا کنید و به خروجی برنامه نگاه کنید. خروجی چه تغییری کرد؟

پس اگر عبارتی را درون علامت `""` بیاوریم، آن عبارت عیناً چاپ می‌شود. مثلاً `cout<<"a"<<endl;` روی صفحه‌ی نمایش حرف `a` را چاپ می‌کند.

اما اگر عبارتی بدون علامت `""` بیاید، به عنوان یک متغیر در نظر گرفته شده و مقدار آن متغیر چاپ می‌شود. به عنوان مثال، مقدار متغیر `a`، مثلاً `10` را چاپ می‌کند.

این بار نوبت شماست تا بگویید خروجی برنامه‌ی زیر چیست با فرض این که کاربر مقدار `20` را وارد کند.

```
int a;  
cin>>a;  
cout<<a*a <<endl;
```

به احتمال زیاد درست حدس زدید، این برنامه، توان دوم مقداری را که کاربر وارد کند، چاپ می‌کند. پس عدد ۴۰۰ به نمایش در می‌آید.

نکته‌ی فنی

وقتی متغیری تعریف می‌شود، مقدار نامشخصی درون آن است. این نکته را به یاد بسپارید که همیشه قبل از استفاده از متغیرها در برنامه، آن‌ها را مقدار دهی نمایید.

۳) محاسبه‌ی میانگین دو عدد صحیح

اگر از شما بخواهند برنامه‌ای بنویسید که دو عدد صحیح از کاربر بگیرد و میانگین آن دو را چاپ کند، ممکن است نتیجه‌ی اولین تلاش شما به شکل زیر باشد:

```
int num1, num2;  
int miangin;  
cin>>num1>>num2;  
miangin = (num1 + num2) /2;  
cout << "miangin = " <<miangin<<endl;
```

اجرای این برنامه وقوع یک خطای منطقی را به ما نشان می‌دهد، چراکه مقداری که روی خروجی دیده می‌شود، برابر انتظار ما نیست:

```
1  
2  
miangin = 1  
Press any key to continue . . .
```

ما انتظار عدد ۱.۰۵ به عنوان میانگین ۱ و ۲ داشتیم، اما ۱ به عنوان خروجی مشخص شد. اگر به برنامه بالا دقت کنید، خیلی ساده به دلیل این اشتباه پی خواهید برد. میانگین ۲ عدد صحیح لزوماً صحیح نیست و ممکن است اعشاری باشد. پس باید متغیر **miangin** را از نوع **float** تعریف کنیم. پس برنامه را به این شکل بازنویسی می‌کنیم:

```
int num1, num2;  
float miangin;  
cin>>num1>>num2;  
miangin = (num1 + num2) /2;  
cout << "miangin = " <<miangin<<endl;
```

به خروجی برنامه‌ی جدید دقت کنید:

```
1
2
miangin = 1
Press any key to continue . . .
```

خطای منطقی دیگری که همچنان در خروجی برنامه مشهود است، این فرصت را به ما می‌دهد که نکته‌ی مهم و طریفی را به شما تذکر دهیم:

نکته‌ی فنی

در دستورات محاسباتی مانند خط چهارم برنامه بالا، ابتدا حاصل عبارت ریاضی، مستقل از متغیری که قرار است حاصل عبارت در آن ریخته شود (در اینجا **miangin**)، محاسبه شده و سپس مقداردهی انجام می‌شود.

دقت انجام محاسبات (**double**, **int**, **float** و ..) همواره برابر بیشترین دقتی است که در عبارت ریاضی سمت راست معادله وجود دارد. در اینجا دو متغیر **num1** و **num2** و عدد 2 از نوع **int** هستند، پس دقت حاصل محاسبات هم از نوع **int** خواهد بود.

برای حل مشکل مثال قبل، باید یکی از این سه مقدار حاضر در محاسبات را از نوع **float** کنیم تا حاصل محاسبات هم اعشاری شود. ساده‌ترین کار ممکن این است به جای 2 که از نوع صحیح است، از 2.0 که از نوع اعشاری است، استفاده نماییم.

با توجه به نکته فنی بالا، می‌توانیم برنامه‌ی قبل را به این صورت بازنویسی کنیم:

```
int num1, num2;
float miangin;
cin>>num1>>num2;
miangin = (num1 + num2) /2.0;
cout << "miangin = " <<miangin<<endl;
```

خوشبختانه برنامه این بار خروجی درست را نمایش می‌دهد:

```
1
2
miangin = 1.5
Press any key to continue . . .
```

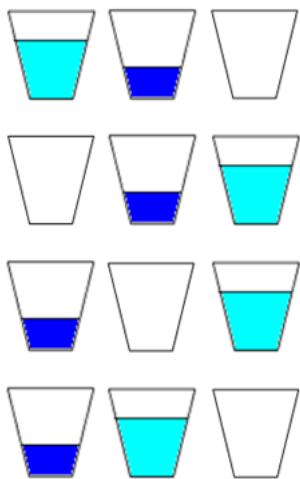
۴) جابجایی دو متغیر

یکی از اعمالی که در آینده‌ی نزدیک، حتماً به آن نیاز پیدا خواهد کرد، این است که مقادیری که داخل دو متغیر هم نوع قرارداد را با هم جابجا^{۲۴} کنید. مثلاً اگر در شروع در متغیر a مقدار ۲ و در متغیر b مقدار ۳ وجود داشته باشد، انتظار داریم بعد از جابجایی داشته باشیم: a=3 و b=2

این کار را می‌توانیم به راحتی با استفاده از یک متغیر اضافی انجام دهیم. نحوه انجام کار را با یک مثال شهودی بهتر درک خواهید کرد. فرض کنید دو لیوان دارید که مقداری آب داخل هر کدام آنها هست و می‌خواهید آب‌های داخل این دو ظرف را عوض کنید.

²⁴ Swap

ساده‌ترین روش آن است که ابتدا آب لیوان اول را داخل لیوان سوم بریزید. سپس لیوان دوم را در لیوان اول خالی کنید. حال آب لیوان سوم را در لیوان دوم بریزید. با این روش مقدار آبی که داخل دو لیوان بود با هم جایجا می‌شود. شکل رویرو مرحله انجام این کار را نشان می‌دهد:



به کد برنامه‌ی جایجا می‌دقت کنید. یادتان باشد باید هر سه متغیر از یک نوع باشند.

```
main.cpp | 
#include <iostream.h>

int main(int argc, char *argv[])
{
    int a = 5, b = 7;

    int temp = a;
    a = b;
    b = temp;

    cout << "a=" << a << " b=" << b << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

خروجی برنامه:

```
a=7 b=5
Press any key to continue . . .
```

به نظر شما، آیا می‌توانیم جایجا می‌دو متغیر را بدون متغیر اضافی انجام دهیم؟ بله! با استفاده از جمع و تفریق‌های متوالی، این امکان وجود دارد که متغیرها را بدون متغیر اضافی با هم جایجا کنیم. بهتر است راجع به این مسئله کمی فکر کنید. پیدا کردن این روش نشان‌دهنده‌ی فکر باز و خلاق شماست.

ابتدا به کد این روش نگاه کنید. سپس با کمک یک مثال این روش را توضیح خواهیم داد.

```

n1 = n1 + n2;
n2 = n1 - n2;
n1 = n1 - n2;

```

معادله	n1	n2
مقدار اولیه	۱۰	۷
n1 = n1 + n2	۱۷	۷
n2 = n1 - n2	۱۷	۱۰
n1 = n1 - n2	۷	۱۰

(۵) کمی درباره کاراکترها

قبل‌اً گفتیم که برای نگه‌داری کاراکترها، از متغیرهایی از نوع `char` استفاده می‌کنیم. در فصل اول اشاره کردیم که خانه‌های حافظه، که متغیرها در آن‌ها نگه‌داری می‌شوند، قابلیت نگه‌داری یک عدد در مبنای ۲ را دارند. اما کاراکترهایی مثل '`a`' و '`@`' هیچ شباهتی به اعداد ندارند. سوال مهمی که در اینجا مطرح می‌شود، این است که کاراکترها چطور در حافظه نگه‌داری می‌شوند؟

با توجه به این‌که هر خانه‌ی حافظه، تنها قابلیت نگه‌داری اعداد را دارد، تنها جواب ممکن این خواهد بود که هر کاراکتری قبل از نگه‌داری در حافظه، تبدیل به یک عدد می‌شود. پس ما نیاز به یک جدول داریم که هر کاراکتر در آن با یک عدد متناظر شده باشد و طبیعتاً هیچ دو کاراکتر متفاوتی نباید متناظر با یک عدد باشند. (چرا؟)

این جدول استاندارد، معروف به **جدول کدهای اسکی**^{۲۵} است. اگر به جدول کدهای اسکی که در ضمیمه آمده است، توجه کنید می‌بینید که به عنوان مثال برای کاراکتر '`A`' کد ۶۵ در نظر گرفته شده است. پس اگر در متغیری از نوع `char`، کاراکتر '`A`' را بزیم، در این متغیر در واقع عدد ۶۵ ذخیره می‌شود. از آنجایی کاراکترها در واقع حاوی اعداد هستند، در زبان C می‌توان با آن‌ها به عنوان عدد هم رفتار کرد. مثلاً عملیات ریاضی را می‌توان روی آن‌ها انجام داد. به برنامه‌ی زیر توجه کنید:

```

char c = 'A';
c = c+1;
cout <<c<<endl;

```

در برنامه‌ی بالا، بعد از ریختن '`A`' در متغیر `C`، مقدار ۶۵ در آن قرار می‌گیرد و بعد از اجرای خط بعدی، مقدار آن به ۶۶ تغییر پیدا می‌کند. انتظار دارید بعد از خط سوم چه مقداری چاپ شود؟ همان‌طور که احتمالاً حدس زدید، کاراکتری چاپ می‌شود که کد اسکی آن برابر ۶۶ باشد. اگر نگاهی به جدول کدهای اسکی بیندازید، کاراکتر '`B`' معادل این کد می‌باشد:

²⁵ ASCII code table

B
Press any key to continue . . .

در ادامه می‌خواهیم برنامه‌ای بنویسیم که یک حرف بزرگ انگلیسی را از ورودی گرفته و حرف کوچک معادل آن را چاپ کند.
با توجه به جدول کدهای اسکی می‌بینیم که کد اسکی هر حرف کوچک، ۳۲ واحد از کد اسکی حرف بزرگ متناظر، بیشتر است. پس کافیست
کد اسکی هر کاراکتر ورودی را با ۳۲ جمع کنیم:

```
char c;  
cin>>c;  
c = c + 32;  
cout <<c<<endl;
```

خروجی برنامه:

D
d
Press any key to continue . . .

تمرین



۱. برنامه‌ای بنویسید که خروجی زیر را تولید کند:

```
*
**
***
****
*****
*****
```

۲. هر کدام از دو روشی که برای جابجایی دو متغیر ذکر شد، یک حُسن و یک ابراد دارند. آن‌ها را پیدا کنید.
۳. پنج عدد از ورودی بخوانید و میانگین آن‌ها را چاپ کنید.
۴. دو کاراکتر از ورودی بخوانید، با فرض این‌که این دو کاراکتر رقم باشند، (${}^9 \dots {}^1$)، با توجه به جدول کدهای اسکی حاصل ضرب این دو عدد را پیدا کنید.
۵. برنامه‌ای بنویسید که یک عدد چهار رقمی از ورودی بخواند و جای ارقام آن را برعکس کرده و چاپ کند. به عنوان مثال، برای ورودی 1234 باید 4321 چاپ کند.

فصل چهارم: گرافیک

۱. مقدمه و راهاندازی گرافیک
۲. آشنایی با محیط گرافیکی
۳. رسم شکل‌های اولیه
۴. استفاده از رنگ

(۱) مقدمه و راه اندازی گرافیک

شاید تقریباً همه‌ی شما، بازی‌های کامپیوتری را دوست داشته باشید. اگر کمی هم به برنامه‌نویسی علاقه‌مند باشید، احتمالاً این فکر و سوشهانگیز به ذهن‌تان خطرور کرده که یک بازی کامپیوتری را طراحی و برنامه‌نویسی کنید. برنامه‌های ساده‌ای که تا این‌جا نوشته‌ایم، شامل یک کنسول بوده‌اند که متن دلخواهی را می‌توانستید در آن چاپ کنید. اما بعید است بازی محبوبی را سراغ داشته باشید که به این شکل، تنها از متن استفاده کند. بازی‌ها معمولاً از شکل‌ها و تصاویر گرافیکی استفاده می‌کنند. در این فصل سعی می‌کنیم، مقدماتی از برنامه‌نویسی گرافیکی را بیینیم، تا توانیم بازی‌های ساده‌ای را تولید کنیم. طبق تجربه‌ای که ما داشته‌ایم، این مطلب، همیشه جزء محبوب‌ترین قسمت‌های برنامه‌نویسی، برای دانش‌آموزان بوده است.

در ادامه می‌بینیم که در محیط برنامه‌نویسی **Dev-Cpp** چطور می‌توانیم محیط گرافیکی را راهاندازی کنیم.

راهاندازی محیط گرافیکی در Dev-Cpp

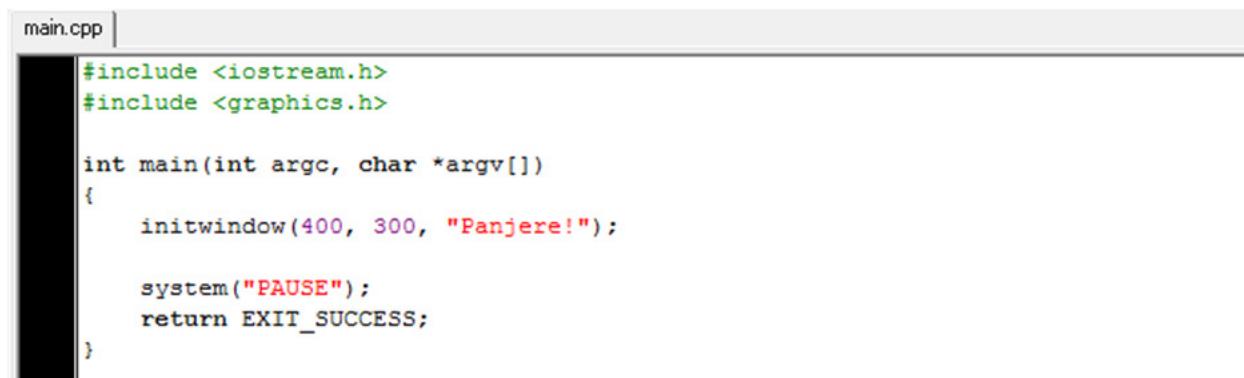
برای استفاده از محیط گرافیکی نیاز به انجام مقدمات راهاندازی محیط گرافیکی داریم. بعد از ساختن **project** برای برنامه خود، از منوی **Project Options**، گزینه‌ی **Project** را انتخاب کرده (یا کلیدهای **Alt+P** را فشار دهید)، سپس در قسمت **Linker** و در پنجره‌ی مربوط به **Parameters** چندخط زیر را با دقت وارد کنید:

```
-lbg1
-lgdi32
-lcomdlg32
-luuid
-loleaut32
-lole32
```

(۲) آشنایی با محیط گرافیکی

برای استفاده از محیط گرافیکی، بعد از انجام مراحل بالا، باید از کتابخانه‌ی گرافیک استفاده کنیم. این کار را با استفاده از دستور **#include <graphics.h>** انجام می‌دهیم.

به‌منظور باز کردن پنجره برای محیط گرافیکی، از دستور **initwindow** استفاده می‌کنیم. در اینجا باید ابعاد پنجره را مشخص کنیم، ابعاد پنجره، شامل عرض و ارتفاع آن می‌باشد. به برنامه‌ی زیر توجه کنید:



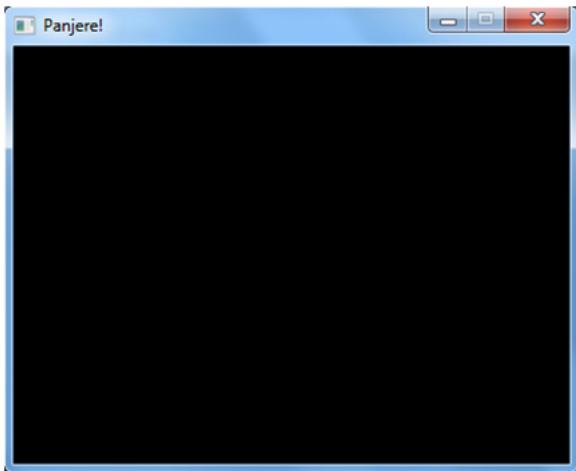
```
main.cpp

#include <iostream.h>
#include <graphics.h>

int main(int argc, char *argv[])
{
    initwindow(400, 300, "Panjere!");

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

در این‌جا یک پنجره به عرض ۴۰۰ و ارتفاع ۳۰۰ تعریف کرده‌ایم و نام آن را که در نوار بالای آن نمایش داده می‌شود، هم "Panjere!" قرار داده‌ایم. خروجی این برنامه، یک پنجره‌ی سیاه بدون شکل خواهد بود:



این پنجره در واقع ۳۰۰ سطر و در هر سطر ۴۰۰ نقطه خواهد داشت که به این نقاط، **پیکسل**^{۲۶} می‌گوییم. هر پیکسل را می‌توانیم با مختصات آن مشخص کنیم. مختصات هر پیکسل، با شماره‌ی ستون (X) و شماره‌ی سطر (Y) تعیین می‌شود. مختصات نقطه‌ی گوشی بالا سمت چپ، $(0, 0)$ است و با حرکت به سمت راست و پایین، به ترتیب، مختصات X و Y افزایش پیدا می‌کند. به این ترتیب مختصات نقطه‌ی گوشی پایین، سمت راست، در این مثال $(299, 299)$ خواهد بود.

۳) رسم شکل‌های اولیه

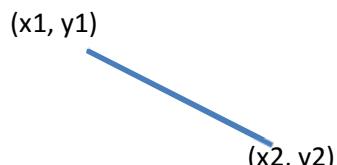
برای رسم برخی اشکال پرکاربرد، دستورهایی وجود دارد که برخی از آن‌ها را در ادامه می‌بینید. در اینجا شماره‌ی ستون را با X و شماره‌ی سطر را با Y نمایش می‌دهیم:

- **نقطه:** با استفاده از دستور زیر، نقطه‌ای در مختصات (y, x) رسم می‌شود:

```
putpixel(x, y);
```

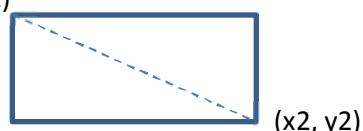
- **خط:** این دستور خطی بین نقاط با مختصات $(x1, y1)$ و $(x2, y2)$ رسم می‌کند:

```
line (x1, y1, x2, y2);
```



- **مستطیل:** با استفاده از دستور زیر، مستطیلی رسم می‌شود که دو سر یکی از قطرهای آن با استفاده از $(x1, y1)$ و $(x2, y2)$ تعیین شده است:

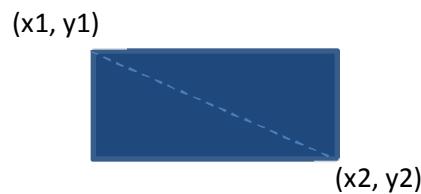
```
rectangle (x1, y1, x2, y2);
```



²⁶ Pixel

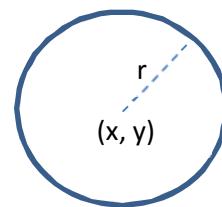
• مستطیل توپر: مشابه رسم مستطیل، با استفاده از دستور زیر:

```
bar(x1, y1, x2, y2);
```



• دایره: دایره‌ای به مختصات مرکز (x, y) و شعاع r می‌کشد.

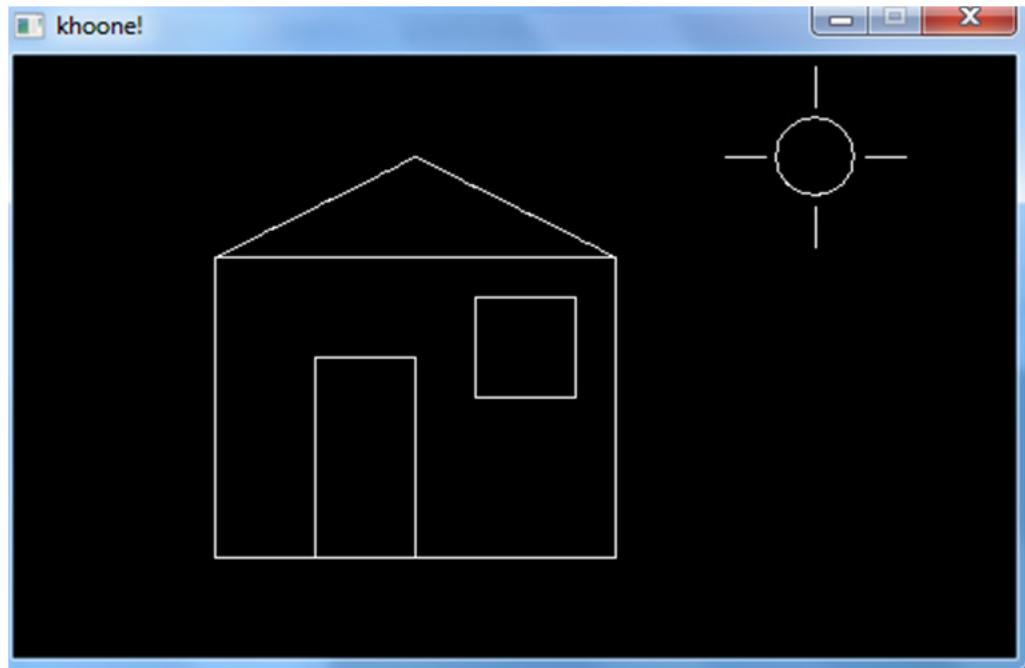
```
circle (x, y, r);
```



در ادامه با استفاده از همین شکل‌های اولیه، به یاد روزهای خوب کودکی، یک نقاشی ساده رسم می‌کنیم. ☺

```
main.cpp |  
#include <iostream.h>  
#include <graphics.h>  
  
int main(int argc, char *argv[]){  
    initwindow(500, 300, "khoone!");  
  
    rectangle(100, 100, 300, 250);  
    rectangle(150, 150, 200, 250);  
    line(100, 100, 200, 50);  
    line(200, 50, 300, 100);  
    rectangle(230, 120, 280, 170);  
  
    circle(400, 50, 20);  
    line(400, 25, 400, 5);  
    line(425, 50, 445, 50);  
    line(375, 50, 355, 50);  
    line(400, 75, 400, 95);  
  
    system("PAUSE");  
    return EXIT_SUCCESS;  
}
```

خروجی برنامه در ادامه آمده است، اما سعی کنید خودتان از روی دستورات، این شکل را روی کاغذ رسم کنید.



۴) استفاده از رنگ

رنگ، یکی از عناصری است که در رسایی و زیبایی تصاویر می‌تواند بسیار موثر باشد. در کتابخانه‌ی گرافیکی مورد استفاده‌ی ما، ۱۶ رنگ برای استفاده در شکل‌ها تعریف شده است. به هر کدام از این رنگ‌ها یک شماره بین ۰ تا ۱۵ نسبت داده شده است، که لیستی از آن‌ها را جدول زیر مشاهده می‌کنید:

رنگ	شماره	رنگ	شماره
خاکستری روشن	۸	سیاه	۰
آبی روشن	۹	آبی	۱
سبز روشن	۱۰	سبز	۲
آبی دریایی روشن	۱۱	آبی دریایی	۳
قرمز روشن	۱۲	قرمز	۴
سرخابی روشن	۱۳	سرخابی	۵
زرد	۱۴	قهوه‌ای	۶
سفید	۱۵	خاکستری تیره	۷

برای این‌که شکل‌های رسم شده در مثال بالا، با خطوط رنگی رسم شوند، باید با استفاده از دستور `setcolor` رنگ را تغییر دهیم. در مثال زیر، پس از کشیدن خانه با رنگ سفید و قبل از رسم خورشید، با استفاده از این دستور رنگ جاری را به زرد تغییر می‌دهیم، تا خورشید به رنگ زرد نمایش داده شود.

```

main.cpp | 
#include <graphics.h>

int main(int argc, char *argv[])
{
    initwindow(500, 300, "khoone!");

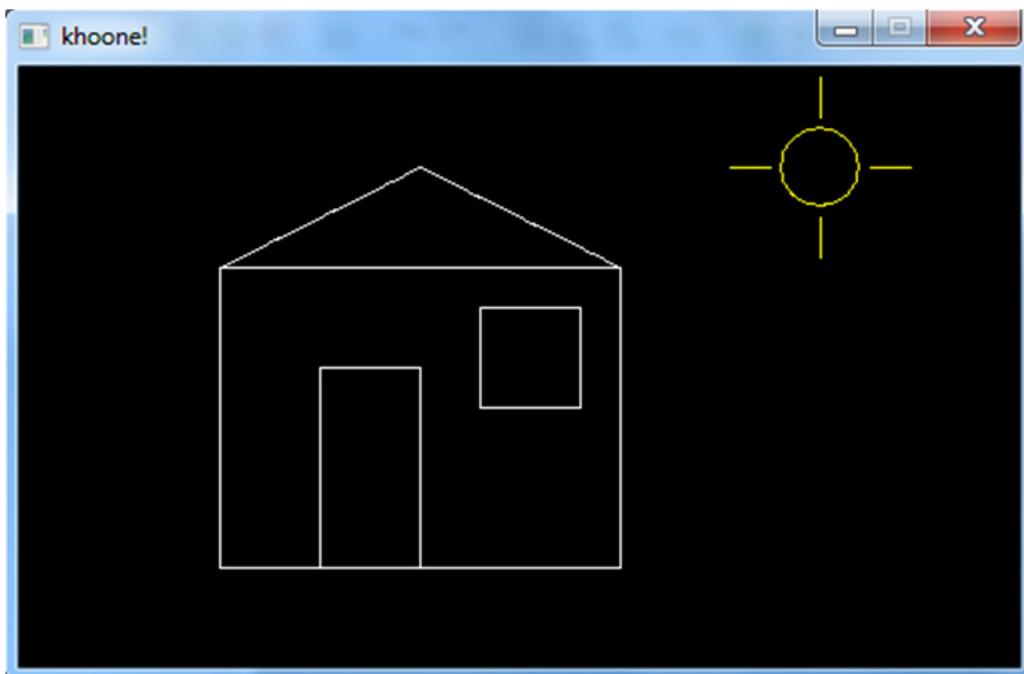
    rectangle(100, 100, 300, 250);
    rectangle(150, 150, 200, 250);
    line(100, 100, 200, 50);
    line(200, 50, 300, 100);
    rectangle(230, 120, 280, 170);

    setcolor(14); ←
    circle(400, 50, 20);
    line (400, 25, 400, 5);
    line (425, 50, 445, 50);
    line (375, 50, 355, 50);
    line (400, 75, 400, 95);

    system("PAUSE");
    return EXIT_SUCCESS;
}

```

این برنامه، شکل زیر را ترسیم می‌کند:



حال اگر بخواهیم خورشید را توپر رسم کنیم، چطور؟ برای انجام این کار دستور **floodfill** وجود دارد، که برای آن یک نقطه‌ی شروع و یک رنگ مرزی تعیین می‌کنیم و به این شکل از نقطه‌ی شروع تا رسیدن به نقاط به مرزی به رنگ مشخص شده، رنگ‌آمیزی می‌شود. این دستور به شکل زیر کار می‌کند:

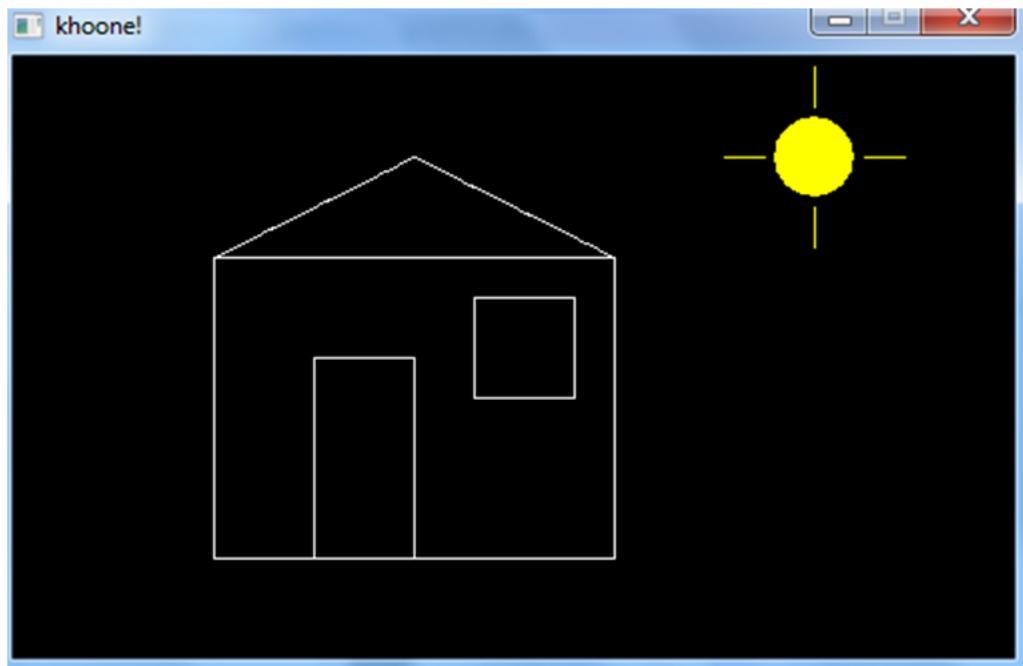
floodfill (x, y); (رنگ مرزی، y)

اما این رنگ‌آمیزی به چه رنگ و مدلی انجام می‌شود؟ این‌ها را با دستور **setfillstyle** مشخص می‌کنیم:

setfillstyle(رنگ، مدل رنگ‌آمیزی);

در این دستور، مدل رنگ‌آمیزی، یک عدد صحیح است که تعیین می‌کند که این شکل با خط، هاشور، رنگ توپر یا ... رنگ آمیزی شود. ما معمولاً از حالت رنگ توپر استفاده می‌کنیم، که با عدد 1 مشخص می‌شود. تغییر مورد نظر در این برنامه اعمال شده است:

۴۰



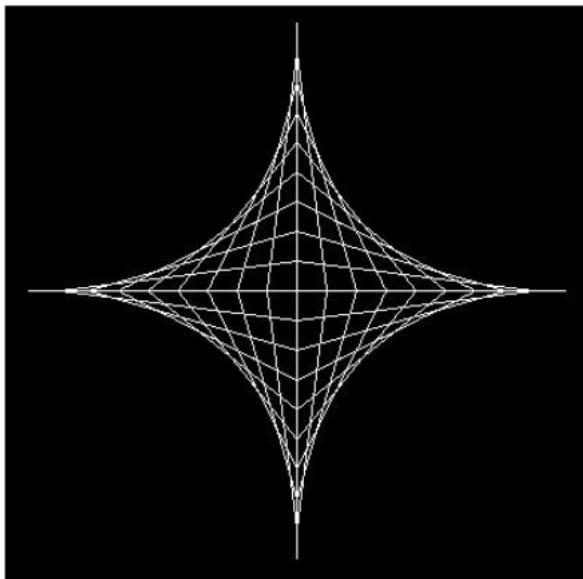
به شکل مشابه، می‌توانید باقی این شکل را رنگ‌آمیزی کنید.

دستورات گرافیکی زیادی در این کتابخانه وجود دارد، که در اینجا فرصت نمی‌شود به همه‌ی آن‌ها بپردازیم، اما در صورت علاقه‌مندی، به راحتی می‌توانید آن‌ها را در اینترنت پیدا کنید.

تمرین



۱. برنامه‌ای بنویسید که مختصات سه راس یک مثلث را از ورودی گرفته و مثلث را به همراه میانه‌های آن رسم کند.
۲. برنامه‌ای بنویسید که مختصات سه راس یک مثلث را از ورودی گرفته و مثلث را به همراه ارتفاعات آن رسم کند. (دقت کنید که حاصلضرب شیب دو خط عمود ۱- است.)
۳. یک مکعب با زاویه‌ی دلخواه روی صفحه‌ی خروجی رسم کنید.
۴. برنامه‌ای بنویسید که یک زمین چمن فوتبال را به همراه خطوط آن رسم کند.
۵. برنامه‌ای بنویسید که شکل زیر را بکشد.



۱. معرفی ساختار شرطی
۲. اندکی در باب شرطها
۳. شرطهای ترکیبی
۴. بلاکها

فصل پنجم: ساختار شرطی

(1) معرفی ساختار شرطی

زندگی ما انسان‌ها پر از شرط‌های مختلف است، شرط‌هایی که شاید هیچ‌وقت به آن‌ها توجه نکردیم. ساده‌ترین این شرط‌ها حالتی است که شما از این منطق استفاده می‌کنید: "اگر این اتفاق افتاد، آن‌گاه من آن کار را انجام می‌دهم!" در برنامه‌نویسی به هر زبانی ما نیاز مبرمی به این ساختارها داریم! این ساختارها به اشکال متفاوتی می‌توانند در زبان‌های برنامه‌نویسی ظاهر شوند، که پرکاربردترین آن‌ها، ساختار `if` است.

در زبان C، این ساختار در ابتدایی‌ترین حالت خود، از سه بخش تشکیل شده است:

`if (شرط)`

کاری که قرار است صورت گیرد

در ابتدای خط کلمه کلیدی `if` می‌آید. پس از آن و داخل پرانتز، شرطی که در صورت درست بودن آن، کاری که در ادامه آمده است، انجام می‌شود. برای مثال، می‌خواهیم برنامه‌ای بنویسیم که در صورتی که متغیر `a` بزرگ‌تر از ۰ بود، به ما بگوید این عدد مثبت است.



```
if (a>0)
    cout<<"+";
```

توجه کنید که در برنامه‌ی فوق، اگر مقدار `a` صفر باشد، شرط نادرست بوده و "+" چاپ نمی‌گردد.

خطاهای معمول برنامه نویسی



قرار دادن علامت `;` بعد از دستور `if` مثلاً به شکل

```
if (a>0) ;
    cout << "+";
```

باعث می‌شود که خط پایین (`;` `+` `<<` `cout`) مستقل از شرط باشد و در هر صورت اجرا شود و این یعنی بروز یک خطای منطقی!

(2) اندکی در باب شرط‌ها

همانطور که گفتیم در ساختار دستور `if` بعد از کلمه‌ی کلیدی `if`، داخل پرانتز شرطی را می‌آوریم که بسته به درست بودن یا نبودن آن، دستورات مربوطه ممکن است اجرا شوند. در ادامه کمی مفصل‌تر به ساختار شرط‌ها خواهیم پرداخت.

شرط‌ها در زبان C دارای یک ساختار کلی با استفاده از **عملگرهای رابطه‌ای**^۷ می‌باشند. شکل کلی آن را در ادامه می‌بینید:

^۷ Relational Operators

مقدار یا متغیر ۱

عملگر رابطه‌ای

مقدار یا متغیر ۲

a > 0

مثالی که قبلاً برای مثبت بودن متغیر a دیدیم، نیز در این ساختار جای می‌گیرد: در این مثال به جای مقدار یا متغیر ۱، متغیر a آمده و به جای مقدار یا متغیر ۲، مقدار ۰ آمده است. در ادامه جدولی آمده است که در آن، انواع عملگرهای شرطی و معنی هر کدام را می‌توانید بینید:

عملگر رابطه‌ای	معنی
>	بزرگ‌تر
<	کوچک‌تر
>=	بزرگ‌تر مساوی
<=	کوچک‌تر مساوی
==	مساوی
!=	نامساوی

مثال‌هایی از استفاده از بعضی از این عملگرهای رابطه‌ای را در ادامه خواهید دید.

بعضی وقت‌ها نیاز پیدا می‌کنیم که بفهمیم متغیری بزرگ‌تر مساوی مقدار یا متغیر دیگری هست یا نه. در این صورت از علامت = > استفاده می‌کنیم. به مثال پایین دقت نمایید:

```
if (a>=0)
    cout<<"+";
```

و یا در صورت نیاز به کوچک‌تر مساوی:

```
if (a<=b)
    cout<<"max = a";
```

و در بعضی شرایط، برای ما فقط تساوی دو مقدار مهم است:

```
if (a==20)
    cout<<"a = 20";
```

نکته‌ی فنی

توجه به این نکته ضروری است که برای بررسی تساوی دو عدد باید از دو علامت == پشت سرهم استفاده کنید. اگر از اشتباه‌اً از یک تساوی استفاده کنید، برنامه اعلام خطأ نخواهد کرد، اما نحوه‌ی اجرا متفاوت خواهد بود. مثلاً اگر از دستور if(a=1) استفاده کنید، ابتدا مقدار 1 در متغیر a قرار داده می‌شود و سپس به دلیلی که به زودی می‌فهمید، شرط صحیح در نظر گرفته می‌شود، حتی اگر مقدار a، 1 نباشد.

خطاهای معمول برنامه نویسی



یکی از شایع‌ترین خطاهایی که ممکن است هنگام برنامه‌نویسی با آن مواجه شوید، استفاده از یک علامت مساوی (`==`) می‌باشد. این خطا از نوع خطاهای منطقی است، به این معنی که خطای شما در زمان کامپایل شدن برنامه اعلام نمی‌شود، اما برنامه ممکن است به درستی کار نکند. لطفاً حواستان باشد این اشتباه را مرتكب نشوید!

همان‌طور که در نکته‌ی فنی و خطاهای معمول برنامه‌نویسی قبیل گفته شد، استفاده از یک تساوی می‌تواند منجر به بروز خطاهای منطقی شود. برای درک بهتر این مطلب به برنامه‌ی زیر توجه کنید:

```
main.cpp |  
include <iostream.h>  
  
int main(int argc, char *argv[]){  
    int a = 5;  
    if (a = 20)  
        cout <<a<<endl;  
    system("PAUSE");  
    return EXIT_SUCCESS;  
}
```

در برنامه‌ی بالا، ما این اشتباه را انجام داده‌ایم و می‌خواهیم ببینیم چه مشکلی برای برنامه‌ی ما پیش می‌آید.

```
20  
Press any key to continue . . .
```

مطابق نکته‌ای که خدمت‌تان عرض کردیم، شرط علی‌رغم این‌که `a` مقدار 5 داشت نه 20، درست در نظر گرفته شده و مقدار 20 هم در متغیر `a` ریخته شده است، هرچند ما هرگز چنین قصدی نداشتمیم. در این‌جا می‌خواهیم توضیح دهیم که چرا این اتفاق افتاده است. ابتدا برای فهم بهتر به نکته‌ی فنی زیر توجه کنید:

نکته‌ی فنی

اگر در ساختار `if` بجای شرط، مقدار یا متغیری مخالف صفر قرار دهیم، بدین معنا است که شرط برقرار بوده و دستور درون `if` اجرا می‌گردد. عنوان مثال در برنامه‌ی زیر

```
if ( 10)  
    cout <<"!"<<endl;
```

چون به جای شرط، عدد غیر صفر (10) آمده است، علامت `!` چاپ خواهد شد.

اتفاقی که برای اجرای دستور `if (a=20)` می‌افتد این است که این دستور به دو دستور متوالی

```
a = 20;  
if (a)
```

تبديل می‌شود. یعنی در ابتدا عدد 20 در متغیر `a` قرار گرفته، سپس [طبق نکته‌ی فنی بالا] بررسی می‌شود که مقدار `a` صفر است یا غیر صفر. چون مقدار `a` برابر 20 و در نتیجه غیر صفر است، شرط درست درنظر گرفته شده و `if` اجرا می‌شود. پس دلیل این که چرا در این مثال مقدار 20 چاپ می‌گردد، تا حدودی مشخص شد.

(۳) شرط‌های ترکیبی

تا اینجا با ساختار `if` آشنا شدیم و توانستیم بر اساس درست بودن یک شرط، کار معینی را انجام دهیم، اما در بسیاری از موارد، ما نیاز داریم که کاری را با در نظر گرفتن همزمان چند شرط انجام دهیم. برای مثال، ممکن است بخواهیم برنامه‌ای بنویسیم که نمره‌ی یک دانش‌آموز را بگیرد و اگر بین ۱۰ تا ۱۲ بود، به او نمره ۱۲ بدهد. این برنامه را هنگامی می‌توانیم بنویسیم که با مفهوم `if`‌های تو در تو آشنا باشیم.

این ساختار به‌گونه‌ای است که دو `if` بدون فاصله از هم می‌آیند و `if` دوم تنها در صورتی اجرا می‌شود که شرط `if` اول درست باشد و در صورتی که شرط `if` دوم هم درست باشد، آن‌گاه کل شرط درست می‌شود و می‌توانیم دستور مربوطه را اجرا کنیم. حالا برنامه‌ی بالا (که قرار بود در صورتی که نمره بین ۱۰ و ۱۲ باشد آن را به ۱۲ تبدیل کنیم) را با هم می‌نویسیم:

```
if (nomre >= 10)  
    if (nomre <= 12)  
        nomre = 12;
```

همان‌طور که در بالا اشاره شد، فقط زمانی خط `nomre=12` اجرا می‌شود که هر دو شرط برقرار باشد و اگر هر کدام از آن‌ها نادرست باشند، این خط اجرا نمی‌شود.

حالا برای تمرین بیشتر روی این مبحث، دو سوال زیر را حل کنید:

۱. برنامه‌ای بنویسید که دو عدد گرفته و اگر هر دوی آن‌ها بزرگ‌تر از ۲۰ بودند، پیغام مناسب چاپ کند.
۲. برنامه‌ای بنویسید که دو عدد گرفته و اگر حداقل یکی از آن‌ها بزرگ‌تر از ۲۰ بود، پیغام مناسب چاپ کند.

به طور کلی موقع زیادی وجود دارند که در آن‌ها، شرط‌های ترکیبی به کار می‌آیند:

۱. می‌خواهید شرطی برقرار نباشد.
۲. می‌خواهید شرط اول و شرط دوم، هر دو برقرار باشند. (مانند مثال بالا)
۳. می‌خواهید حداقل یکی از شرط‌های اول یا دوم برقرار باشند.

احتمالاً هنگام حل هر کدام از ۲ سوال بالا، که مثال‌هایی از ۳ حالت ذکر شده هستند، از ۲ دستور `if` استفاده کرداید و ممکن است این سوال برای تان مطرح شده باشد که آیا می‌توان این برنامه‌ها را ساده‌تر نوشت؟ (مثلاً با استفاده از تنها یک دستور `if`)

زبان C شیوه‌ی ساده‌تری برای برنامه‌نویسی در حالت‌های بالا، که بسیار پرکاربرد هم هستند، در نظر گرفته است. اگر فرض کنید C1 و C2 دو شرط باشند، (مثلاً `a>20` و `b>20`: C1 و C2)، هر کدام از سه حالت بالا را می‌توان به اشکال زیر بیان کرد:

۱. شرط C1 برقرار نباشد:

- از علامت `!` (بخوانید `not`) استفاده کنید: (C1) `!`
- مثلاً می‌توانید بنویسید:

```
if (! (a>20))
    cout <<"a bozorgtar az 20 nist!"<<endl;
```

یعنی اگر شرط `a>20` برقرار نباشد، که معادل این است که `a<=20` باشد.

۲. هر دو شرط C1 و C2 برقرار باشند:

- از علامت `&&` (بخوانید `and`) استفاده کنید: (C1 `&&` C2)
- مثلاً برای حل ساده‌تر سوال ۱ از سوال‌های بالا:

```
if ( a>20 && b>20 )
    cout <<"a va b har 2 bozorgtar az 20 hastand!"<<endl;
```

۳. حداقل یکی از شرط‌های C1 یا C2 برقرار باشند:

- از علامت `||` (بخوانید `or`) استفاده کنید: (C1 `||` C2)
- مثلاً برای حل ساده‌تر سوال ۲ بالا:

```
if (a>20 || b>20)
    cout <<"hadeaghah yeki az a va b az 20 bozorgtar ast"<<endl;
```

ماهیت این فصل پایه‌ای به گونه‌ای است که برای تسلط به مبحث، نیاز به حل تمرین‌های زیادی دارد، پس لطفاً تمرین‌های زیر را حل کنید:

۱. برنامه‌ای بنویسید که عددی گرفته و اگر این عدد بین ۲۵ و ۱۸ بود، پنج برابر آن عدد را چاپ کند.
۲. برنامه‌ای بنویسید که دو عدد گرفته و اگر عدد دوم بزرگ‌تر از عدد اول بود و همچین از ۳ برابر عدد اول کوچک‌تر بود، پیغام مناسب چاپ کند.
۳. برنامه‌ای بنویسید که دو عدد گرفته و اگر فاصله‌ی آن‌ها کمتر از ۱۰ بود، پیغام مناسب چاپ کند.
۴. برنامه‌ای بنویسید که دو عدد گرفته و اگر حداقل یکی از آن‌ها کوچک‌تر از ۵ بود، پیغام مناسب چاپ کند.
۵. برنامه‌ای بنویسید که دو عدد گرفته و اگر تنها یکی از آن‌ها بزرگ‌تر از ۲۰ بود، پیغام مناسب چاپ کند.

تا اینجا تمامی کارهایی که در صورت درستی شرط انجام می‌شده، یک خطی بود. اما در حالت‌هایی لازم است در صورت درستی شرط، چند خط اجرا شود. اما تا الان ما فقط می‌توانستیم از خطی که بلافصله بعد از `if` می‌آمد، استفاده کنیم، برای این‌که این مشکل حل شود، باید از **بلاک^{۲۸}**‌ها کمک بگیریم که در مباحث بعدی نیز به آن نیاز خواهیم داشت.

بلاک چیست؟ به محدوده‌ای که بین دو آکولاد باز و بسته می‌آید، اصطلاحاً **بلاک** گفته می‌شود. بلاک‌ها قسمتی از برنامه هستند که با هم مرتبط بوده و برای مشخص کردن محدوده‌ی یک وظیفه‌ی معین به کار می‌رود.

حالا که با مفهوم بلاک آشنا شدید، به مواردی که نیاز به استفاده‌ی از آن دارد، اشاره می‌شود. فرض کنید که باید برنامه‌ای بنویسیم که یک عدد از کاربر گرفته و اگر کمتر از ۲۰ بود با نمایش پیغام مناسب، آن را دو برابر کنیم.

```
cin>>a;
if (a<20)
{
    a=a*2;
    cout<<"a = " <<a <<endl;
}
```

خروجی برنامه‌ی بالا را با ورودی‌های مختلف را می‌بینیم:

```
5
a = 10
Press any key to continue . . .
```

```
25
Press any key to continue . . .
```

حال فرض کنید که برنامه‌ی بالا را بدون بلاک می‌نوشتم. فکر می‌کنید چه اتفاقی می‌افتد؟

```
cin>>a;
if (a<20)
    a=a*2;
cout<<"a = " <<a <<endl;
```

برنامه را با همان مثال‌های قبل اجرا می‌کنیم:

```
5
a = 10
Press any key to continue . . .
```

²⁸ Block

```
25
a = 25
Press any key to continue . . .
```

مشکل چیست؟ خط آخر برنامه که وظیفه‌ی نمایش پیغام مناسب را دارد، در هر صورت (چه شرط درست باشد، چه نباشد) اجرا شده است. در حالی که قرار بوده تنها زمانی اجرا گردد که شرط برقرار باشد. نتیجه‌ی مربوط به این مثال‌ها را در نکته‌ی فنی زیر جمع‌بندی می‌کنیم:

نکته‌ی فنی

در شرایطی که نیاز باشد در صورت درستی یک شرط، بیش از یک دستور اجرا شود، آن دستورها را در یک بلاک قرار می‌دهیم. توجه کنید که اشکالی ندارد که تنها یک دستور را در بلاک قرار دهید.

خطاهای معمول برنامه نویسی



در شرایطی که قصد داریم بیش از یک دستور را در صورت درستی شرط اجرا نماییم، اگر از بلاک استفاده نکنیم، دچار یک خطای منطقی خواهیم شد. به این صورت که تنها اولین دستور به صورت مشروط و بقیه دستورها در همه‌ی حالتها اجرا می‌شوند. (مانند مثال قبل)

۵) و گرنه!

یادگیری ساختار **if** همچنان ادامه دارد، خیلی وقت‌ها حالتی پیش می‌آید که لازم است در صورت درست بودن شرط کار یا کارهای خاصی صورت گیرد و در صورت نادرست بودن آن، کار یا کارهای دیگری انجام دهیم. به این حالت اصطلاحاً **else-if** (و گرنه - اگر!) گفته می‌شود. ساختار کلی آن به این صورت است:

if (شرط)

کار یا کارهایی که قرار است در صورت درست بودن شرط انجام شود

else

کاری(هایی) که قرار است در صورت نادرست بودن شرط انجام شود

برای درک بهتر این مطلب به مثال روبرو توجه کنید:

```
cin>>a;
if (a>12)
    a=a+1;
else
    a=12;
```

کمی به این مثال فکر کنید. این برنامه چه کاری انجام می‌دهد؟ سعی کنید قبل از این‌که توضیح آن را در ادامه بخوانید، این مسئله را در ذهن‌تان حل کنید.

این برنامه قرار است عملیات محبوب ارافق روی نمرات دانش‌آموزان را انجام بدهد. برنامه‌ی فوق یک عدد [نمره] از کاربر می‌گیرد و در صورتی که بزرگ‌تر از ۱۲ باشد، به آن یک واحد اضافه می‌کند. در غیر این صورت، اگر کوچک‌تر یا مساوی ۱۲ باشد، مقدار a را برابر ۱۲ قرار می‌دهد. حالا اگر قرار باشد برنامه را طوری تغییر دهید که با پیغام‌های مناسب این کار را انجام دهد، به استفاده از بلاک نیاز پیدا می‌کنید، پس باید حواس‌تان را جمع کنید. حال تلاش کنید این برنامه را بنویسید و برنامه‌ی خودتان را با برنامه‌ی زیر مقایسه کنید. امیدواریم که برنامه را درست نوشته باشید.

```
cin>>a;
if (a>12)
{
    a=a+1;
    cout<<"a=a+1" <<endl;
}
else
{
    a=12;
    cout<<"a=12" <<endl;
}
```

دقیق کنید که بعد از **else** می‌توان از شرط دیگری هم استفاده کرد و به این ترتیب از ساختار **if else** استفاده نمود.

تمرین



۱. برنامه‌ای بنویسید که دو عدد بگیرد و بگوید عدد کوچکتر زوج است یا فرد.
۲. برنامه‌ای بنویسید که مجموع امتیاز یک شخص در بازی ۱-۳-۵ را بگیرد و بگوید کدام پرتابها گل شده اند.
(در بازی ۱-۳-۵ سه پرتاب داریم که پرتاب اول ۵ امتیاز، دوم ۳ امتیاز و سوم ۱ امتیاز دارد)
۳. برنامه‌ای بنویسید که ۴ عدد را در ۴ متغیر **d**، **c**، **b**، **a**، **د**، **س**، **ب**، **ا** بگیرد و سپس آنها را مرتب کند. (به گونه‌ای که بزرگ‌ترین در **a** و ... باشد)(نکته: می‌دانیم که دومین عدد بزرگ‌ترین و سومین عدد کوچک‌ترین است، اما درباره‌ی بقیه چیزی نمی‌دانیم)
۴. برنامه‌ای بنویسید که اندازه سه ضلع مثلث را پرسد و نوع مثلث را از بین این انواع تعیین کند: معمولی، متساوی‌الساقین، متساوی‌الاضلاع. (می‌دانیم که با این سه عدد می‌توان مثلث ساخت.)
۵. برنامه‌ای بنویسید که سه عدد به عنوان اندازه‌ی اضلاع مثلث بگیرد و بگوید مثلث قائم‌الزاویه است یا نه.
۶. برنامه‌ای بنویسید که دو عدد بگیرد و اگر تنها یکی از آن‌ها (و نه هر دوی آن‌ها) بر ۷ بخش پذیر بود، پیغام دهد **Yes** و در غیر این صورت **No**.
۷. برنامه‌ای بنویسید که عددی بگیرد و اگر بر ۲ و ۳ و حداقل یکی از اعداد ۵ و ۷ بخش پذیر بود، پیغام دهد.
۸. برنامه‌ای بنویسید که دو ساعت را بگیرد و جمع آن‌ها را نمایش دهد. (مثلاً ساعت ۱۱:۲۰:۳۴ را به صورت سه عدد ۱۱، ۲۰، ۳۴ می‌گیرد.)
۹. برنامه‌ای بنویسید که ۴ عدد بگیرد و کوچک‌ترین عدد زوج در میان آن‌ها را نمایش دهد.
۱۰. برنامه‌ای بنویسید که به ترتیب محل یک فیل و یک سریاز را در صفحه شطرنج بگیرد و بگوید آیا فیل سریاز را تهدید می‌کند یا نه. (محل یک مهره شطرنج را به صورت دو عدد می‌گیرد. مثلاً دو عدد ۱ و ۱ نشان‌دهنده‌ی قرار داشتن مهره در خانه‌ی ۱ و ۱ شطرنج است.)

فصل ششم : حلقوی while

۱. حلقه چیست؟
۲. ساختار حلقه‌ی `while`
۳. مثال چاپ ستاره در یک سطر
۴. مثال توان عامل ۲
۵. مثال فاکتوریل
۶. متغیر پرچم
۷. مثال شمارش اعداد زوج
۸. حلقه‌ی `do while`
۹. مثال چاپ بزرگ‌ترین عدد
۱۰. مثال کد اسکی
۱۱. مثال رسم دایره
۱۲. مثال مجموع مربعات

۱) حلقه چیست؟

حلقه^{۲۹} ساختاری است که امکان تکرار یک عمل به تعداد دلخواه را برای ما فراهم می‌کند. به بیانی دقیق‌تر، حلقه ساختاری، کنترلی است، که یک شرط اجرا دارد که تا هنگامی که این شرط برقرار باشد، دستورات درون بلوک حلقه، تکرار می‌شوند.

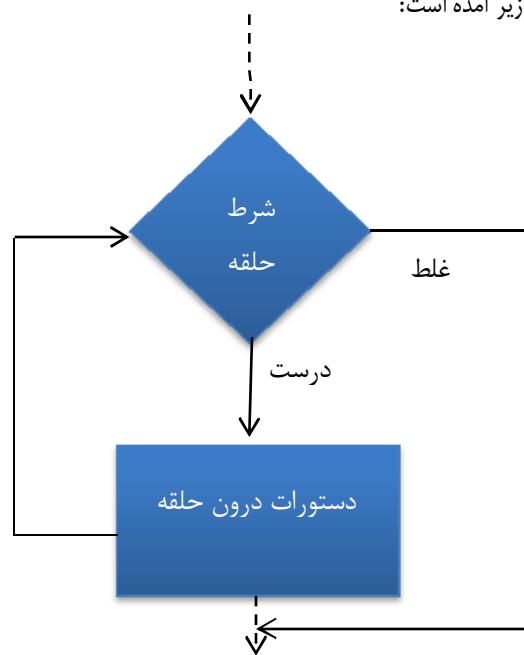
چرا به تکرار نیاز داریم؟

فرض کنید می‌خواهیم میانگین نمره‌ی درس کامپیوتر دانش‌آموزان یک کلاس که به ترتیب توسط یک کاربر وارد می‌شوند، را حساب کنیم با فرض این که بدانیم این کلاس ۲۰ نفر دانش‌آموز دارد، اگر از حلقه استفاده نکنیم باید ۲۰ بار دستور `cin` (خواندن مقدار) و ریختن در متغیر را بنویسیم. اما توسط حلقه و با یک بار نوشتن این عمل درون حلقه، می‌توان عمل میانگین‌گیری را انجام داد. پس یکی از خوبی‌های حلقه، کاهش حجم برنامه و در نتیجه بالا رفتن خوانایی و کاهش میزان خطای احتمالی است. حال اگر بخواهیم این برنامه، قابل استفاده برای کلاس‌هایی با تعداد دانش‌آموز متفاوت باشد، آیا بدون حلقه امکان نوشتن این برنامه را داشتیم؟ پس استفاده از حلقه امری اجتناب‌ناپذیر در جایی است که از قبل تعداد تکرار آن را نمی‌دانیم.

۲) ساختار حلقه‌ی `while`

ساختار حلقه‌ی `while` از دو بخش شرط حلقه و بلوک حلقه تشکیل شده است. در هنگام اجرای این ساختار، ابتدا شرط حلقه بررسی می‌شود. سپس در صورت برقرار بودن شرط، قطعه برنامه‌ی درون بلوک اجرا می‌شود. این حلقه تا هنگامی که شرط حلقه برقرار باشد، تکرار می‌شود. شما از حلقه‌ی `while` در زیر آمده است:

```
while (شرط حلقه)
{
    دستورات درون حلقه
}
```



²⁹ Loop

۳) مثال چاپ ستاره در یک سطر

برنامه‌ای با استفاده از حلقه‌ی **while** بنویسید که ۲۰ بار کاراکتر ستاره (*۲۰) را بر روی صفحه چاپ کند.

برای این‌که حلقه ۲۰ بار تکرار شود، از یک متغیر **int** (**adad**) استفاده کرده‌ایم و در هر بار تکرار حلقه، مقدار آن را یکی اضافه می‌کنیم. شرط حلقه، کوچکتر بودن **adad** از ۲۰ است و در ابتدای کار متغیر به مقدار صفر مقداردهی شده است.

```
main.cpp
#include <iostream.h>

int main(int argc, char *argv[])
{
    int adad = 0;
    while (adad < 20)
    {
        cout << "*";
        adad = adad + 1;
    }
    cout << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

حاصل اجرای برنامه در زیر آمده است:

```
*****
Press any key to continue . . .
```

خطاهای معمول برنامه نویسی



توجه کنید که اگر پس از دستور **while** و در پایان خط از کاراکتر **و** (سمیکالن) استفاده شود، در صورتی که شرط حلقه برقرار باشد، کامپیوتر فکر می‌کند که حلقه‌ی **while** به پایان رسیده است و سراغ دستورات درون حلقه نمی‌رود و دوباره شرط حلقه را چک می‌کند. این کار تکرار می‌شود و برنامه متوقف نمی‌شود، اصطلاحاً می‌گوییم که در حلقه‌ی بینهایت افتاده‌ایم.

سوال: در چه حالات‌های دیگری ممکن است درون حلقه‌ی بینهایت بیافتیم؟

۴) مثال توان عامل دو

برنامه‌ای با استفاده از حلقه‌ی **while** بنویسید که عددی از ورودی دریافت کرده، توان عامل دوی آن عدد را نمایش دهد. در این مثال برای این که عامل دوی عدد ورودی را بیابیم، باید تا جایی که عدد بر ۲ بخش‌بذیر است، آن را بر دو تقسیم کنیم. تعداد دفعاتی که عدد را بر دو تقسیم کرده‌ایم، توان عامل دو خواهد بود. بنابراین شرط حلقه، باید بخش‌بذیری عدد بر ۲ باشد. عملکر $\% \text{ باقیمانده‌ی عامل چپ بر عامل سمت راست را برمی‌گرداند}$ و اگر باقیمانده‌ی عددی بر ۲ صفر باشد، یعنی بر ۲ بخش‌بذیر است:

```
main.cpp | #include <iostream.h>

int main(int argc, char *argv[])
{
    int adad;
    cout << "yek adad vared konid: ";
    cin >> adad;
    int amele_do = 0;
    while (adad % 2 == 0)
    {
        adad = adad / 2;
        amele_do++;
    }
    cout << "amele do: " << amele_do << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

برای مثال برای پیدا کردن توان عامل دو برای عدد ۹۶ برنامه عدد ۵ را برمی‌گرداند:

```
yek adad vared konid: 96
amele do: 5
Press any key to continue . . .
```

(۵) مثال فاکتوریل

برنامه‌ای بنویسید که با دریافت یک عدد، فاکتوریل آن عدد را محاسبه و چاپ نماید.

```
main.cpp | #include <iostream.h>

int main(int argc, char *argv[])
{
    int adad;
    cout << "yek adad vared konid: ";
    cin >> adad;
    int factorial = 1;
    while (adad > 0)
    {
        factorial = factorial * adad;
        adad = adad - 1;
    }
    cout << factorial << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

فاکتوریل یک عدد، حاصل ضرب تمام اعداد طبیعی کوچکتر و مساوی آن عدد در یکدیگر می‌باشد، بنابراین کافی است، حلقه‌ای بنویسیم که تا وقتی که به صفر نرسیده‌ایم، عدد را یکی کم کند و در حاصل تاکنون به دست آمده ضرب کند. توجه کنید در ابتدای کار، متغیر فاکتوریل را مساوی یک قرار داده‌ایم، زیرا در غیر این صورت با توجه به مقدار اولیه‌ی نامشخص متغیر، برنامه درست کار نکند.

برای مثال اگر عدد ۶ را وارد نماییم، برنامه مقدار !۶ را محاسبه کرده برای ما نمایش می‌دهد:

```
yek adad vared konid: 6
720
Press any key to continue . . .
```

۶) متغیر پرچم

در جایی که چند شرط برای پایان حلقه داریم، می‌توانیم با استفاده از عملگرهای عطف و فصل شرط‌ها را با هم به عنوان شرط حلقه قرار دهیم روشی دیگر برای این کار استفاده از **متغیر پرچم**^{۳۰} می‌باشد. متغیر پرچم یک متغیر از جنس **bool** می‌باشد که در ابتدا به مقدار صحیح (true) مقداردهی می‌شود و به عنوان شرط حلقه از آن استفاده می‌شود و اگر یکی از شرط‌هایی که برای پایان حلقه داریم برقرار نباشند، این متغیر را به مقدار غلط (false) مقداردهی می‌کنیم و بدین ترتیب حلقه پایان می‌یابد. ساختار کلی استفاده از متغیر پرچم به شکل زیر است:

```
bool parcham = true;
while (parcham == true)
{
    دستورات دون حلقه
}
```

نکته‌ی فنی

هنگامی که از متغیر پرچم به عنوان شرط حلقه استفاده می‌کنیم، حلقه **while** حداقل یک بار تکرار می‌شود، زیرا در ابتدا متغیر پرچم صحیح است و حتماً وارد حلقه می‌شویم، در ادامه با ساختار **do while** که خاصیت مشابهی دارد آشنا می‌شویم.

۷) مثال شمارش اعداد زوج

برنامه‌ای بنویسید که تا هنگامی که عددی فرد وارد نشده است، از کاربر عدد دریافت کند، و سپس تعداد اعداد زوج وارد شده را چاپ کند.

³⁰ Flag Variable

```
main.cpp |
#include <iostream.h>

int main(int argc, char *argv[])
{
    int adad;
    bool parcham = true;
    int zoj = 0;
    cout << "adadi vared konid: (baraye khorooj adadi fard vared konid" << endl;

    while (parcham == true)
    {
        cin >> adad;
        if (adad % 2 == 0)
            zoj++;
        else
            parcham = false;
        if (parcham == true)
            cout << "adade ba'di ra vared konid: " << endl;
    }
    cout << "Te'dade a'daade zoje vared shode: " << zoj << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

توجه کنید که متغیر `ZOJ` که در آن تعداد اعداد زوج ذخیره می‌شود، در ابتدا به مقدار صفر مقداردهی شده است. اگر عددی زوج وارد شود یک به مقدار این متغیر افزوده می‌شود و اگر عددی فرد وارد شود، `parcham` به مقدار غلط مقداردهی می‌شود و در نتیجه حلقه پایان می‌یابد.

```
adadi vared konid: <baraye khorooj adadi fard vared konid
4
adade ba'di ra vared konid:
12
adade ba'di ra vared konid:
18
adade ba'di ra vared konid:
16
adade ba'di ra vared konid:
19
Te'dade a'daade zoje vared shode: 4
Press any key to continue . . .
```

do while حلقه‌ی (۸)

حلقه‌ی `do while` حلقه‌ای است همانند حلقه‌ی `while` با این تفاوت که در این ساختار در اولین اجراء شرط حلقه بررسی نمی‌شود و دستورات درون بلوک حلقه، حداقل یک بار اجرا می‌شوند، اما تکرارهای بعدی در صورت برقراری شرط حلقه صورت می‌گیرد.

```
do
{
    دستورات درون حلقه
} while (شرط حلقه);
```



توجه کنید که در ساختار `do while` بخلاف `while` باید در انتهای خط و پس از دستور `while` علامت `;` قرار دهد.

۹) مثال چاپ بزرگ‌ترین عدد

برنامه‌ای بنویسید که تا هنگامی که عدد صفر توسط کاربر وارد نشده، عدد دریافت کند، در صورت ورود عدد صفر به کار خود پایان داده، سپس بزرگ‌ترین عدد خوانده شده را چاپ نماید.

برای این کار باید هر بار که عددی خوانده می‌شود با عدد ماکریمی که تا به حال به دست آمده مقایسه شده در صورت بزرگ‌تر بودن از ماکریم به عنوان ماکریم جدید ذخیره شود، شرط حلقه نیز مساوی بودن عدد وارد شده با صفر است.

```
main.cpp | #include <iostream.h>

int main(int argc, char *argv[])
{
    int adad;
    int max = 0;
    do
    {
        cout << "Adadi vared konid:(baraye paian 0 ra vared konid) " << endl;
        cin >> adad;
        if (adad > max)
            max = adad;
    } while (adad != 0);
    cout << "Maximum: " << max << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

برای مثال پس از ورود چند عدد زیر، مقدار ماکریم عددهای وارد شده (۵۷) چاپ شده است:

```
Adadi vared konid:(baraye paian 0 ra vared konid)
34
Adadi vared konid:(baraye paian 0 ra vared konid)
57
Adadi vared konid:(baraye paian 0 ra vared konid)
34
Adadi vared konid:(baraye paian 0 ra vared konid)
23
Adadi vared konid:(baraye paian 0 ra vared konid)
12
Adadi vared konid:(baraye paian 0 ra vared konid)
0
Maximum: 57
Press any key to continue . . .
```

نکته‌ی فنی

همان طور که در مثال بالا می‌بینید، در اینجا نیاز است که حداقل یک عدد توسط کاربر وارد شود، بنابراین از حلقه‌ی `do while` استفاده شده است.

(۱۰) مثال کد اسکی

برنامه‌ای بنویسید که تا وقتی حرف `p` وارد نشده است، یک حرف کوچک انگلیسی دریافت کرده، کد ASCII و حرف بزرگ آن را چاپ کند.

برای این کار باید از کد ASCII حروف استفاده کنیم. متناظر هر کاراکتری که در صفحه‌ی نمایش کامپیوتر نمایش داده می‌شود، یک عدد در جدولی به نام جدول ASCII ذخیره شده است و کامپیوتر برای نمایش حروف با مراجعه به این جدول متوجه می‌شود که چه کاراکتری را باید بر روی صفحه نمایش دهد، اما حروف کوچک انگلیسی از شماره‌ی ۹۷ تا شماره‌ی ۱۲۲ جدول ASCII می‌باشند و حروف بزرگ انگلیسی از شماره‌ی ۶۵ شروع می‌شوند بنابراین برای تبدیل حرف کوچک به حرف بزرگ باید ۳۲ تا از شماره‌ی آن حرف کم کنیم.

نکته‌ی فنی

دقت کنید برای شرط پایان و چک کردن وارد شدن حرف `p` از علامت `'` (کتیشن) استفاده شده است. و اگر از این علامت استفاده نمی‌کردید، آن را به عنوان متغیر در نظر می‌گرفت.

main.cpp

```
#include <iostream.h>

int main(int argc, char *argv[])
{
    char harf;
    char harfeBozorg;
    int ascii;
    cout<<"yek harf vared konid (baraye khorooj 0 ra bezanid):"<<endl;
    do {
        cin >> harf;
        if (harf <= 122 && harf >= 97) {
            harfeBozorg = harf - 32;
            ascii = harf;
            cout << harfeBozorg << " " << ascii << endl;
        }
    } while (harf != 'p');
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

نتیجه‌ی ورود چند کاراکتر مختلف پس از اجرا در ادامه آمده است:

```
yek harf vared konid <baraye khorooj 0 ra bezanid>:
a 97
h 104
p 112
Press any key to continue . . .
```

(۱) مثال رسم دایره

برنامه‌ای بنویسید که از کاربر سه عدد (x و y به عنوان مرکز دایره و r به عنوان شعاع) دریافت کند و دایره‌ای به مشخصات وارد شده رسم کند و در صورتی که یک کدام از این سه عدد صفر داده شد، پایان یابد.

با توجه به این که در صورت سوال ذکر شده است که اگر یکی از سه عدد صفر باشد برنامه پایان یابد، بنابراین شرط حلقه حاصل عطف (And) صفر نبودن سه متغیر است، یعنی اگر هر کدام از این سه متغیر صفر باشند، شرط حلقه نقض شده، حلقه پایان می‌یابد.

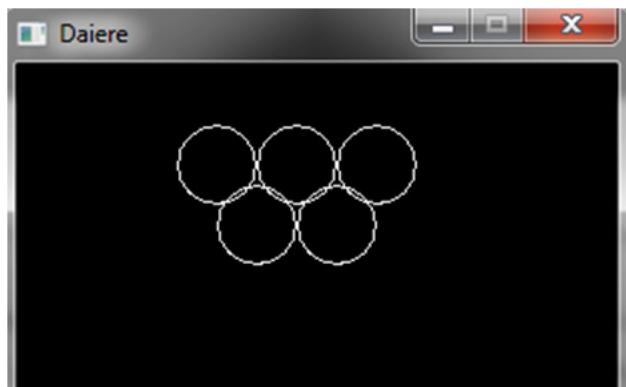
```
main.cpp

#include <iostream.h>
#include <graphics.h>

int main(int argc, char *argv[])
{
    initwindow(300, 300, "Daiere");
    int x, y, shoa;
    do
    {
        cout << "x, y va r ra vared konid: " << endl;
        cin >> x;
        cin >> y;
        cin >> shoa;
        circle(x, y, shoa);
    } while (x != 0 && y != 0 && shoa && != 0);
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

نتیجه‌ی رسم چند دایره در زیر آمده است:

```
x, y va r ra vared konid:
100 50 20
x, y va r ra vared konid:
140 50 20
x, y va r ra vared konid:
180 50 20
x, y va r ra vared konid:
120 80 20
x, y va r ra vared konid:
160 80 20
x, y va r ra vared konid:
0 0 0
Press any key to continue . . .
```



۱۲) مثال مجموع مربعات

برنامه‌ای بنویسید که تا هنگامی که کاربر عدد صفر را وارد نکرده، عدد دریافت کرده در نهایت مجموع مربعات آنها را نمایش دهد.

متغیر majmoo برای ذخیره‌ی حاصل جمع مربعات اعداد وارد شده تعریف شده است، توجه کنید که در ابتدای کار این متغیر به صفر مقداردهی اولیه شده است، اگر این امر رعایت نمی‌شد، در ابتدای کار majmoo عددی نامعلوم بود و نتیجه‌ای که به دست می‌آمد اشتباه بود.

```
main.cpp | #include <iostream.h>

int main(int argc, char *argv[])
{
    int adad;
    int majmoo = 0;
    cout << "chand adad vared konid (baraye payan 0 vared konid):" << endl;
    do{
        cin >> adad;
        majmoo += adad*adad;
    } while(adad!=0);
    cout << "majmooe morabae adade vared shode:" << majmoo << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

حاصل اجرای این برنامه را در زیر می‌بینید:

```
chand adad vared konid (baraye payan 0 vared konid):
6
3
4
0
majmooe morabae adade vared shode:61
Press any key to continue . . .
```

تمرین



- برنامه‌ای بنویسید که یک عدد گرفته و بگوید آیا ده‌تاب هست یا نه. عدد ده‌تاب عددی است که جمع تعدادی از ارقام سمت راست آن برابر با ۱۰ باشد. به عنوان مثال ۱۸۲ ده‌تاب هست اما ۵۳۶۲ ده‌تاب نیست.
- برنامه‌ی زیر چه کاری انجام می‌دهد؟

main1.cpp |

```
#include <iostream.h>

int main()
{
    int y;
    int x = 1;
    int total = 0;
    while ( x <= 10 )
    {
        y = x * x;
        cout << y << endl;
        total += y;
        x++;
    }

    cout << "Total is " << total << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

- برنامه‌ای بنویسید که عددی از کاربر دریافت کرده، اعلام کند که این عدد اول است یا خیر. (یک عدد اول فقط بر خودش و بر یک بخش‌پذیر است.)

برنامه‌ای بنویسید که دو عدد مثبت را از ورودی خوانده، آنها را به روش تفریق بر هم تقسیم نماید.

- برنامه‌ای بنویسید که تا وقتی عدد صفر وارد نشده، عدد سه‌رقمی دریافت شده را به روش زیر رمزگذاری کرده و نمایش دهد. برای رمزگذاری هر عدد در ۳ ضرب شده و منهای ۵۷ می‌شود.

۶. حرکت یک توب را شبیه‌سازی کنید که با برخورد به هر کدام از ۴ خط مرزی بالا، پایین، چپ و راست، بازتاب پیدا کند. برای نمایش حرکت توب، باید در هر مرحله محل قبلی توب را پاک کنید و توب را در محل جدید رسم نمایید.

- برنامه‌ای بنویسید که دو عدد گرفته و ب.م. آنها را به روش نرdbانی محاسبه و نمایش دهد. (برای مثال ب.م. دو عدد ۴۲۰ و ۶۶ برابر است با ۶)

	۶	۲	۱	۳	
۴۲۰	۶۶	۲۴	۱۸	۶	.
۳۹۶	۴۸	۱۸	۱۸		

فصل هفتم: حلقهی `for`

۱. معرفی حلقهی `for`
۲. ساختار حلقهی `for`
۳. مثال اعداد میانی
۴. مثال ترتیب نزولی
۵. مثال رسم دایره
۶. مثال عدد π

۱) معرفی حلقه‌ی for

نوع دیگری از حلقه در زبان C حلقه‌ی **for** می‌باشد، حلقه‌ی **for** معمولاً در جایی که تعداد دفعات تکرار حلقه مشخص است به کار می‌رود. مزیت اصلی **for** نسبت به **while**، ساختار ساده‌تر آن برای حلقه‌های با تعداد تکرار مشخص می‌باشد. ساختار **for** علاوه بر شرط حلقه، دارای یک شمارنده می‌باشد که این شمارنده به همراه شرط حلقه، تعداد تکرار حلقه را مشخص می‌کنند، برای آشنایی اولیه با این ساختار، مثال محاسبه‌ی معدل نمرات دانش‌آموزان کلاس ۱۰ نفری را که در ابتدای فصل به آن اشاره کردیم را با استفاده از ساختار **for** پیاده‌سازی می‌کنیم:

```
main.cpp |  
#include <iostream.h>  
  
int main(int argc, char *argv[])
{
    int daneshamooz;
    int nomre;
    double miangin = 0;
    for (daneshamooz = 0; daneshamooz < 10; daneshamooz++)
    {
        cout << "Nomreie daneshamooze ba'di ra vared konid: " << endl;
        cin >> nomre;
        miangin = miangin + nomre;
    }
    miangin = miangin / 10;
    cout << "Nomreie miangine kelas: " << miangin << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

نتیجه‌ی اجرای این برنامه برای ۱۰ نمره‌ی فرضی در زیر آمده است:

```
Nomreie daneshamooze ba'di ra vared konid:
16
Nomreie daneshamooze ba'di ra vared konid:
16.5
Nomreie daneshamooze ba'di ra vared konid:
16
Nomreie daneshamooze ba'di ra vared konid:
17
Nomreie daneshamooze ba'di ra vared konid:
18
Nomreie daneshamooze ba'di ra vared konid:
14
Nomreie daneshamooze ba'di ra vared konid:
15.5
Nomreie daneshamooze ba'di ra vared konid:
19
Nomreie daneshamooze ba'di ra vared konid:
18
Nomreie daneshamooze ba'di ra vared konid:
16
Nomreie miangine kelas: 16.6
Press any key to continue . . .
```

همان طور که در مثال بالا می‌بینید، در ساختار **for** مقداردهی اولیه، شرط پایان حلقه و قدمهای حلقه گنجانده شده است و متغیر شمارندهٔ حلقه می‌باشد، که ابتدا به مقدار صفر مقداردهی می‌شود و در هر تکرار افزایش یافته و تا وقتی به ۱۰ نرسیده است، حلقه تکرار می‌شود.

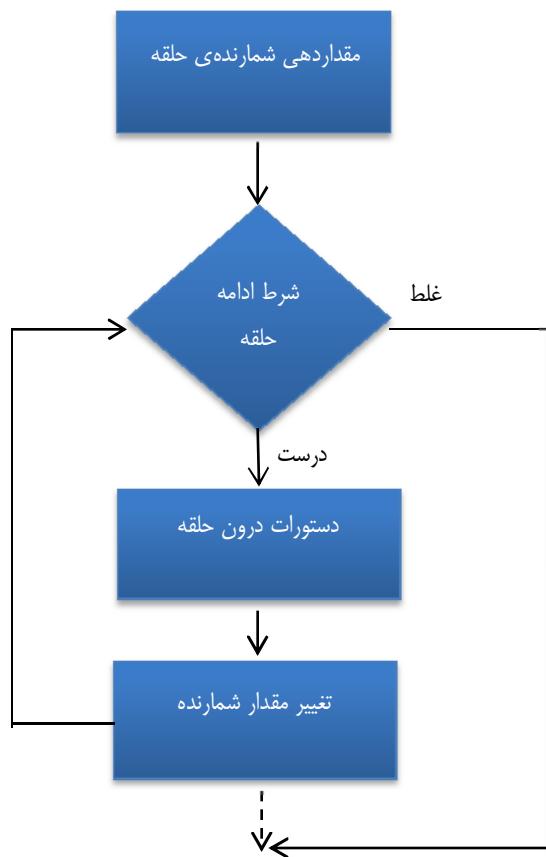
نکته‌ی دیگری که در این مثال رعایت شده است تعریف متغیر **miangin** از نوع **double** می‌باشد که اگر از جنس **int** تعریف می‌شد، معدل حساب شده به سمت پایین گرد می‌شود و قسمت اعشاری آن نمایش داده نمی‌شود.

۲) ساختار حلقه‌ی **for**

تعداد دفعات تکرار حلقه‌ی **for**، توسط متغیری که آن را متغیر حلقه یا شمارنده می‌نامیم کنترل می‌شود، همان‌طور که در شکل زیر می‌بینید، حلقه‌ی **for** از ۴ بخش (مقداردهی اولیه‌ی متغیر، شرط ادامه حلقه، تغییر مقدار شمارنده و دستورات درون حلقه) تشکیل شده است:

```
for (تغییر مقدار شمارنده ; شرط ادامه حلقه ; مقداردهی اولیه‌ی شمارنده)
{
    دستورات درون حلقه
}
```

مقداردهی شمارندهٔ حلقه تنها در اولین اجرای حلقه انجام می‌شود، شرط ادامه حلقه برای هر بار تکرار بررسی شده در صورت برقرار بودن شرط، حلقه اجرا می‌شود، دستورات درون حلقه همانند حلقه‌ی **while** در هر بار تکرار حلقه اجرا می‌شوند و تغییر مقدار شمارنده در انتهای هر بار تکرار انجام می‌گیرد. برای درک بهتر عملکرد حلقه‌ی **for**، فلوچارت زیر را در نظر بگیرید:



خطاهای معمول برنامه نویسی



قرار دادن علامت ؛ بعد از خط اول `for` به شکل زیر، دستورات مربوط به حلقه را مستقل از حلقه می‌کند و به این ترتیب، این دستورات تنها یکبار اجرا می‌شوند.

```
for (int i=1; i<10; i++) ;  
    cout << "!" ;
```

مثلاً در برنامه‌ی بالا، علامت تعجب تنها یکبار چاپ می‌شود

نکته‌ی فنی

هر برنامه‌ای که را با `for` بنویسید، با `while` هم می‌توانید آن را شبیه‌سازی کنید و برعکس. به این ترتیب سوالی که مطرح می‌شود، این است که در چه شرایطی از `for` و در چه شرایطی از `while` استفاده کنیم؟ به این مسئله کمی فکر کنید...

پاسخ این است که در شرایطی که تعداد دفعات انجام حلقه مشخص نباشد و یک شرط که از قبل مشخص نیست کی برقرار می‌شود به عنوان شرط پایان در نظر گرفته شده باشد، از `while` استفاده می‌کنیم. مثلاً شرایطی که بخواهیم در حلقه‌ای از کاربر عدد بگیریم و این کار را زمانی پایان دهیم که عددی منفی وارد شود. اما اگر تعداد دفعات اجرای حلقه مشخص باشد، یا کار مورد نیاز، یک کار شمارشی باشد، ساختار حلقه‌ی `for` برای انجام آن‌ها بهینه‌تر است. مثلاً اگر بخواهید اعداد فرد بین `a` و `b` را چاپ کنید، بهتر است از `for` استفاده کنید.

(۳) مثال اعداد میانی

برنامه‌ای با استفاده از حلقه `for` بنویسید که دو عدد از ورودی دریافت کرده (عدد اول کوچک‌تر است) و اعداد بین آن دو را چاپ کند.

```
main.cpp |  
#include <iostream.h>  
  
int main(int argc, char *argv[]){  
    int adad_aval;  
    int adad_dovom;  
    int shomarande;  
    cout<<"2 adad vared konid ta adade beine 2 adad chap shavand" << endl;  
    cin >> adad_aval;  
    cin >> adad_dovom;  
    for (shomarande = adad_aval; shomarande <= adad_dovom; shomarande ++)  
    {  
        cout << shomarande << endl;  
    }  
    system("PAUSE");  
    return EXIT_SUCCESS;  
}
```

توجه کنید اگر این مثال را با استفاده از حلقه‌ی **While** می‌خواستیم بنویسیم، باید در یک خط ابتدا شمارنده را به عدد اول مقداردهی می‌کردیم، در یک خط شرط حلقه را می‌نوشتیم و در درون حلقه، شمارنده را افزایش می‌دادیم، اما با استفاده از حلقه‌ی **for** همه‌ی اعمال مرتبط با تکرار حلقه در یک خط نوشته شده است که این امر ساختار بهتری برای ما فراهم می‌کند و احتمال فراموشی و یا خطا در نوشتن این سه دستور را کاهش می‌دهد.

نمونه‌ای از اجرای این برنامه در زیر آمده است:

```
2 adad vared konid ta adade beine 2 adad chap shavand
2 10
2
3
4
5
6
7
8
9
10
Press any key to continue . . .
```

نکته‌ی فنی

در مرحله‌ی **تغییر مقدار شمارنده در حلقه‌ی for**، لزومی بر افزایش شمارنده‌ی حلقه و استفاده از عملگر **++** وجود ندارد و گام حلقه می‌تواند بزرگتر از یک باشد، همچنین حلقه می‌تواند کاهشی باشد. مثلاً اگر بخواهید اعداد فرد را چاپ کنید، باید از ۱ شروع کرده و گام حلقه را ۲ قرار دهید.

۴) مثال ترتیب نزولی

برنامه‌ی قبل را طوری تغییر دهید که دو عدد دریافت کرده و اعداد بین آن دو را به ترتیب نزولی چاپ کند.

در این مثال فرضی بر روی کوچکتر یا بزرگتر بودن عدد اول نشده است، بنابراین ابتدا باید عدد بزرگتر را تشخیص دهیم و سپس با استفاده از حلقه‌ی نزولی از عدد بزرگ تا عدد کوچک را نمایش دهیم.

main.cpp |

```
#include <iostream.h>

int main(int argc, char *argv[])
{
    int adad_aval;
    int adad_dovom;
    int temp;
    int shomarande;
    cout<<"2 adad vared konid ta adade beine 2 adad chap shavand" << endl;
    cin >> adad_aval;
    cin >> adad_dovom;
    if (adad_dovom < adad_aval) {
        temp = adad_dovom;
        adad_dovom = adad_aval;
        adad_aval = temp;
    }

    for (shomarande = adad_dovom; shomarande >= adad_aval; shomarande--)
    {
        cout << shomarande << endl;
    }
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

حاصل اجرای این برنامه برای دو عدد ۵ و ۱۵ را در زیر می‌بینید:

```
2 adad vared konid ta adade beine 2 adad chap shavand
5 15
15
14
13
12
11
10
9
8
7
6
5
Press any key to continue . . .
```

(۵) مثال رسم دایره

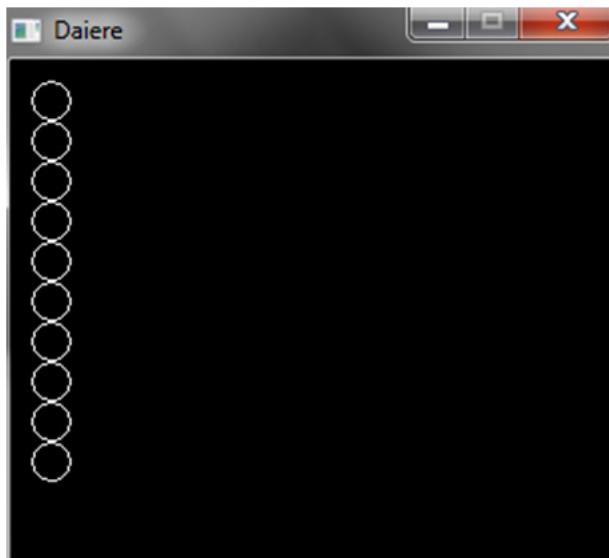
برنامه‌ای با استفاده از حلقه‌ی **for** بنویسید که ۱۰ دایره زیر هم که برهم مماس هستند را بر روی صفحه چاپ کند.

در اینجا برای این که هر بار دایره‌ی جدید، زیر دایره‌ی جدید، مماس بر آن کشیده شود، باید مرکز دایره‌ی جدید به اندازه‌ی قطر دایره‌ی پایین‌تر آمده باشد، بنابراین با توجه به این که نقطه‌ی $(0, 0)$ دستگاه مختصات در گوشی بالا سمت چپ قرار دارد، هر بار باید مختصات **adad** را به اندازه‌ی قطر دایره افزایش دهیم، که این امر با ضرب **adad** که هر بار یکی افزایش می‌یابد، در قطر (دو برابر شاع) حاصل می‌شود.

```
main.cpp | 
#include <graphics.h>

int main(int argc, char *argv[])
{
    initwindow(300, 300, "Daiere");
    int shoa = 10;
    int adad;
    for (adad = 0; adad < 10; adad++)
    {
        circle(20, 20 + (adad * shoa * 2), shoa);
    }
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

پس از اجرای برنامه، شکل زیر رسم می‌شود:



سوال: اگر بخواهیم دایره‌های تو در تو بکشیم چه تغییری باید در این برنامه بدهیم؟ اگر بخواهیم به تعداد دلخواه کاربر دایره رسم شود چطور؟

π) مثال عدد π

می‌خواهیم با استفاده از سری لایبنتیز عدد π را محاسبه کنیم، برنامه‌ای بنویسید که این عدد را تا تعداد جمله‌ی دلخواه برای ما محاسبه نماید.

سری لایبنتیز برای محاسبه‌ی عدد π به فرم روبرو می‌باشد: $\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \dots$ بنابراین با توجه به این که کاربر می‌خواهد تا چند جمله‌ی اول این سری را محاسبه نماید، باید این سری را ادامه داد. ابتدا تعداد جمله‌ی مورد نظر کاربر را دریافت می‌کنیم، سپس حلقه‌ای به طول عدد دریافت شده ایجاد می‌کنیم و درون حلقه با توجه به این که جملات حلقه یک در میان مثبت و منفی می‌شوند، جملات زوج را با عدد حاصل، جمع و جملات منفی را از آن کم می‌کنیم، توجه کنید که متغیری که برای عدد π در نظر گرفته‌ایم، **double** تعریف شده است.

```
main.cpp |  
#include <iostream.h>  
  
int main(int argc, char *argv[])  
{  
    int i;  
    int jomle;  
    double pi = 0;  
    cout<<"ta jomleie chandome seri ra niaz darid: "<<endl;  
    cin >> jomle;  
    for (i = 0; i < jomle; i++) {  
        if (i % 2 == 0)  
            pi += 4.0 / ((2 * i) + 1);  
        else  
            pi -= 4.0 / ((2 * i) + 1);  
    }  
    cout << "PI: " << pi << endl;  
    system("PAUSE");  
    return EXIT_SUCCESS;  
}
```

برای مثال حاصل حساب کردن این سری تا جمله‌ی ۱۰۰۰ام در شکل زیر آمده است:

```
ta jomleie chandome seri ra niaz darid:  
1000  
PI: 3.14059  
Press any key to continue . . .
```

تمرین



- برنامه‌ای بنویسید که عددی پنج رقمی دریافت کرده، ارقام آن را از هم جدا کرده و به ترتیب بر روی صفحه نمایش دهد.
- برنامه‌ای بنویسید که حاصل سری زیر را تا جمله‌ی n ام محاسبه نماید و در صفحه نمایش دهد. (مقدار n از کاربر دریافت می‌شود)

$$1 + \frac{1}{2!} + \frac{1}{3!} + \dots + \frac{1}{n!}$$

- برنامه‌ای بنویسید که سری فیبوناچی را تا جمله‌ی n ام نمایش دهد. (مقدار n از کاربر دریافت می‌شود)
سری فیبوناچی: ...
 $1, 1, 2, 3, 5, 8, \dots$
- برنامه‌ای بنویسید که معدل n دانش‌آموز را از ورودی خوانده، دانش‌آموز با دومین معدل را پیدا کرده و نمایش دهد.
- برنامه‌ای بنویسید که عددی پنج رقمی دریافت کرده و به کاربر بگوید که این عدد آینه‌ای است یا خیر. (عدد آینه‌ای عددی است که اگر آن را از چپ به راست و از راست به چپ بخوانیم یکسان باشد، مانند ۱۲۳۲۱).
- برنامه‌ای بنویسید که با گرفتن ضرایب یک معادله‌ی درجه سه به شکل $ay^3 + bx^2 + cx + d = 0$ ، نمودار y - x آن را روی محور مختصات رسم کند.

فصل هشتم : آرایه‌ها

۱. آشنایی اولیه با آرایه‌ها
۲. مقداردهی اولیه به آرایه
۳. مثال بدست آوردن میانگین اعداد
۴. جستجو
۵. پیدا کردن ماکریم یک آرایه
۶. رشته‌ها
۷. آرایه‌های دو بعدی

۱) آشنایی اولیه با آرایه‌ها

در فصل‌های قبل با انواع متغیرها آشنا شدید و دیدید که برای گرفتن ۱۰ عدد از کاربر و نگه‌داری آن‌ها، مجبور به تعریف ۱۰ متغیر مختلف هستید. اگر تعداد متغیرها زیاد باشد، این کار تقریباً محال است. مثلاً فرض کنید برای تحلیل وزن افراد ایرانی، نیاز پیدا کنید وزن همه‌ی هم‌وطنان (که حدود ۷۰ میلیون نفر هستند) را در برنامه‌ی خود نگه‌دارید. این یعنی تعریف ۷۰ میلیون متغیر! فکر می‌کنم با این اوصاف، دیگر علاقه‌ای به نوشتمن این برنامه نداشته باشید.

برای رفع این مشکل، در زبان C ساختاری وجود دارد که به کمک آن، می‌توانید به سادگی تعداد زیادی متغیر از یک نوع تعریف کنید. به این ساختار آرایه^{۳۱} گفته می‌شود. آرایه عبارت است از تعداد مشخصی از متغیرهای هم نوع. تعریف آرایه در زبان C بسیار شبیه به تعریف یک متغیر است، تنها با این تفاوت که در اینجا باید تعداد متغیرهای این آرایه را مشخص کرد.

```
int b;
int a[100];
```

در خط اول یک متغیر از نوع صحیح (با همان رویه‌ی قبلی که بلد بودید) و در خط دوم صد متغیر صحیح تعریف کردیدم. هر کدام از این ۱۰۰ متغیر یک شماره دارند که از طریق این شماره هر کدام مشخص می‌شوند. مثلاً ممکن است بگوییم مقدار خانه‌ی شماره‌ی ۱۰ از آرایه‌ی a چاپ کن. به این شماره‌ها که هر کدام مشخص کننده‌ی یک خانه از آرایه هستند، آندیس^{۳۲} گفته می‌شود.

پس برای دسترسی به هر کدام از این متغیرها، باید آندیس آن را جلوی نام متغیر بیاورید. به مثال رویرو توجه نمایید:

```
int a[100];
a[1] = 10;
cin >> a[2];
a[3] = a[1] + a[2];
cout << a[3] << endl;
```

نکته‌ی فنی

در زبان C آرایه‌ها از صفر شروع می‌شود. وقتی آرایه‌ای ۱۰۰ عضوی تعریف می‌کنیم، شماره‌ی آندیس‌ها از ۰ شروع و تا ۹۹ ادامه پیدا می‌کند و شماره‌ی آندیس ۱۰۰ جزء آرایه نمی‌باشد. یعنی متغیرهای آرایه به شکل زیر قابل دسترسی هستند:

a[0], a[1], a[2], a[3], ..., a[99]

³¹ Array

³² Index

خطاهای معمول برنامه نویسی



اگر آرایه‌ای به طول مثلاً ۱۰۰ تعریف کرده باشد، دسترسی به خانه‌ای به اندیس ۱۰۰، که طبق نکته‌ی فنی قبلی موجود نیست، سبب بروز خطا خواهد شد.

(۲) مقدار دهی اولیه به آرایه

قبلاً به شما تذکر دادیم که وقتی یک متغیر تعریف می‌کنید، بلافصله بعد از تعریف، مقدار نامشخصی در آن قرار دارد و به همین دلیل از شما خواستیم که همیشه بعد از تعریف و قبل از استفاده از متغیر، به آن یک مقدار اولیه معقول بدهید. آرایه‌ها هم که مجموعه‌ای از متغیرها هستند، از این قاعده مستثنی نیستند، بعد از تعریف آرایه، هر کدام از خانه‌های آرایه مقدار نامشخصی دارند. اگر باور ندارید به برنامه‌ی زیر و خروجی آن دقت کنید:

```
main.cpp |  
#include <iostream.h>  
  
int main(int argc, char *argv[])  
{  
    int a[10];  
    for (int i=0;i<10;i++)  
        cout <<a[i]<<endl;  
    system("PAUSE");  
    return EXIT_SUCCESS;  
}
```

این برنامه بدون این‌که به خانه‌های آرایه‌ی ۱۰ عضوی **a** مقدار بدهد، مقدار این خانه‌ها را چاپ می‌کند. خروجی برنامه را در زیر مشاهده می‌کنید:

```
3411696  
3411576  
2293560  
1985257245  
0  
0  
2293616  
108  
100  
5513680  
Press any key to continue . . .
```

انشالاً... که تا این‌جا قانع شده‌اید که حتماً آرایه‌ها را هم باید مقداردهی اولیه کنید. مثلاً اگر بخواهید مقدار خانه‌های آرایه را در حالت اولیه برابر ۱ قرار دهید، می‌توانید از یک **for** استفاده کنید:

```
int a[10];  
for (int i=0;i<10;i++)  
    a[i] = 1;
```

در شرایطی که تعداد خانه‌های آرایه کم باشد، (مثل اینجا که ۱۰ خانه داریم)، می‌توانید از روش ساده‌تری استفاده کنید. به یک خط کد زیر توجه کنید:

```
int a[10] = {1,1,1,1,1,1,1,1,1,1};
```

همانطور که می‌بینید، شما می‌توانید مقدار خانه‌های آرایه را بلا فاصله بعد از تعریف، با آوردن مقادیر مورد نظر در یک آکولاد باز و بسته، مشخص کنید.

حتماً این سوال برایتان پیش آمده که، اگر تعداد اعدادی که درون آکولاد مشخص می‌کنید، کم‌تر از تعداد خانه‌های آرایه باشد، تکلیف باقی خانه‌ها چه می‌شود؟ چیزی شبیه خط زیر:

```
int a[10] = {1,1,1};
```

تصمیمی که زبان C برای این شرایط گرفته این است که باقی خانه‌های آرایه را با صفر پر می‌کند. اگر خانه‌های آرایه‌ای که به شکل بالا مقداردهی اولیه شده است را چاپ کنیم، این قضیه را تایید می‌کند:

```
1
1
1
0
0
0
0
0
0
0
0
Press any key to continue . . .
```

با توجه به آن‌چه گفتیم، برای صفر کردن همه‌ی خانه‌های آرایه، که پراستفاده‌ترین مقداردهی اولیه نیز می‌باشد، یک راه ساده، به شکل زیر است:

```
int a[10] = {0};
```

(۳) مثال بdst آوردن میانگین اعداد

همان‌طور که دیدید، تمام کارهایی را که با متغیرهای معمولی می‌توانستید انجام دهید، (خواندن مقدار یا ریختن یک مقدار دیگر در آن)، با هر خانه از آرایه هم می‌توانید انجام دهید، چرا که هر خانه از آرایه، خود یک متغیر است. در ادامه برنامه‌ای که ۱۰۰ عدد می‌گیرد و میانگین آنها را محاسبه و چاپ می‌نماید، آمده است:

```
float avg;
int i;
int a[100];
for(i=0; i<100; i++)
{
    cin>>a[i];
    avg = avg + a[i];
}
cout<<avg/100;
```

با این روش، علاوه بر این که نیاز به تعریف تعداد زیادی متغیر با نام‌های مختلف نداریم، مقدار آن‌ها را برای محاسبات بعدی نگهداشته‌ایم. شاید با خودتان فکر کنید که این برنامه را می‌توانستیم بدون آرایه هم، به صورت زیر هم بنویسیم:

```
float avg;
int i, x;
for(i=0; i<100; i++)
{
    cin>>x;
    avg = avg + x;
}
cout<<avg/100;
```

حرف تان درست است، اما آیا همه‌ی مسائل را به این شکل بدون آرایه می‌توان نوشت؟ خیر! مثلاً فرض کنید بین ۱۰۰ عددی که از ورودی به ما داده می‌شود، عددی را می‌خواهیم که از همه، به میانگین اعداد نزدیک‌تر باشد. در این صورت ابتدا باید میانگین اعداد را بدست آوریم، سپس یک‌بار دیگر همه اعداد را با میانگین مقایسه کنیم و نزدیک‌ترین را پیدا کنیم. در این حالت، دیگر حتماً نیاز داریم که ۱۰۰ عدد وارد شده را در آرایه نگه داریم. نوشتن این برنامه را به عنوان یکی از تمرین‌های پایان فصل، بر عهده‌هی شما می‌گذاریم!

۴) جستجو

برنامه‌های بسیاری به کمک ساختار آرایه نوشته می‌شوند. یکی از پرکاربردترین این برنامه‌ها، جستجو در بین اعداد وارد شده است. اولین برنامه‌ای که خواهیم دید، بدست آوردن اندیس عدد ۲۰ در آرایه است. فرض می‌کنیم آرایه‌ی **a** یک آرایه‌ی ۱۰ عضوی است.

```
int a[10] = {2,3,5,8,9,20,7,5,10,1};
int index = -1;
for (int i=0;i<10;i++)
{
    if (a[i]==8)
        index = i;
if (index == -1)
    cout <<"8 dar in araye vojud nadarad."<<endl;
else
    cout <<"8 dar andise "<< index << " ast."<<endl;
```

در این برنامه، متغیر **index** برای نگه‌داری اندیس عدد ۸ در نظر گرفته شده است. در ابتدا که هنوز این عدد را پیدا نکرده‌ایم، این متغیر با مقدار -۱ - مقداردهی اولیه شده است. بعد در یک حلقه روی تمامی عناصر آرایه، بررسی می‌کنیم که در صورتی که در خانه‌ی **i** ام آرایه عدد ۸ موجود باشد، **i** را در متغیر **index** می‌ریزیم. در پایان اگر **index** برابر ۱ - باقی مانده باشد، به این معنی است که ۸ در آرایه موجود نبوده و گرنه اندیس آن را چاپ می‌کنیم.

خروجی برنامه:

```
8 dar andise 3 ast.
Press any key to continue . . . -
```

توجه کنید که برای این مسئله یک روش سریع‌تر وجود دارد که در فصل یازدهم (برنامه‌نویسی بازگشتی) با آن آشنا خواهید شد.

ابتدا برنامه‌ای بنویسید که تعداد تکرارهای عددی که توسط کاربر وارد می‌شود (مثلًاً X)، را بدست آورد.

```
int a[10] = {2,3,5,8,9,20,7,5,10,1};
int x;
cin>>x;
int index = -1;
int tekrar = 0;
for (int i=0;i<10;i++)
    if (a[i]==x)
        tekrar++;
cout <<"tedad tekrare " <<x<<" dar araye: "<<tekrar<<endl;
```

در این برنامه متغیر **tekrar** مسئول نگهداری تعداد تکرار عدد X در آرایه است. هر جا در آرایه عنصری برابر X دیدیم، متغیر شمارندهٔ تکرار، افزایش می‌یابد. خروجی برنامه به ازای $X=5$ را در ادامه می‌بینید:

```
5
tedad tekrare 5 dar araye: 2
Press any key to continue . . .
```

(۵) پیدا کردن ماقریزم یک آرایه

در بسیاری از موارد، در برنامه‌ها با این مواجه هستیم که عنصری با بزرگترین [یا کوچکترین] مقدار را در آرایه بیابیم. مثلًاً ممکن است بخواهید بلندقدترین فرد را در یک کشور، پیدا کرده و به اردوی تیم ملی بسکتبال دعوت کنید! با توجه به این که این برنامه بسیار کاربردی است، خوب آن را یاد گرفته و به خاطر بسپارید.

شما برای حل این مشکل نباید مشکل خاصی داشته باشید، پس سعی کنید خودتان این مسئله را حل کنید و سپس به برنامه زیر نگاه کنید:

```
int a[10] = {2,3,5,8,9,20,7,5,10,1};
int max = a[0];
for (int i=1;i<10;i++)
    if (a[i]>max)
        max = a[i];
cout <<"max dar in araye: "<<max<<endl;
```

در روش بالا، ابتدا اولین عضو آرایه را بزرگترین عدد فرض کردیم و آن را در متغیر **max** ریخته‌ایم. در ادامه به کمک یک حلقهٔ **for** تک عناصر آرایه را با **max** مقایسه می‌کنیم، اگر هر کدام از این عناصر از **max** بزرگ‌تر بودنده، مقدار آن عنصر را در متغیر **max** می‌ریزیم و از آن پس این عدد بزرگترین عددی است که تا اینجا دیده شده است. این مقایسه را تا انتهای آرایه ادامه می‌دهیم.

این هم شکلی برای فهم بهتر نحوهٔ اجرای برنامه:

شماره متغیر	a[0]	a[1]	a[2]	...	a[8]	a[9]
مقدار	2	3	5		10	1
I	0	1	2	...	8	9
if(a[i] > max)	--	True	True	...	False	False
Max	2	3	5	...	20	20

این هم خروجی برنامه:

```
max dar in araye: 20
Press any key to continue . . .
```

در بسیاری از موارد به جای این که مقدار مаксیمم را بخواهید که در مثال بالا ۲۰ است، اندیس ماکزیمم در آرایه را می‌خواهید، که در مثال بالا برابر ۵ است. یعنی ماکزیمم در خانه‌ای با اندیس ۵ قرار دارد. بدیهی است که شما با داشتن اندیس عنصر ماکزیمم به راحتی به مقدار آن هم دسترسی خواهید داشت.

```
int a[10] = {2,3,5,8,9,20,7,5,10,1};
int maxIndex = 0;
for (int i=1;i<10;i++)
    if (a[i]>a[maxIndex])
        maxIndex = i;
cout <<"andise max: "<<maxIndex<<, meghdare max: "<<a[maxIndex]<<endl;
```

در این برنامه، به جای این که یک متغیر برای مقدار ماکزیمم نگه‌داری کنیم، یک متغیر به نام `maxIndex` برای نگه‌داری اندیس ماکزیمم داریم. در حلقه `for` در صورتی که عنصر `i`ام بزرگ‌تر از عنصر آرایه با اندیس `maxIndex` باشد، `maxIndex` را برابر `i` می‌کنیم.

خروجی برنامه را هم در ادامه مشاهده می‌کنید:

```
andise max: 5, meghdare max: 20
Press any key to continue . . .
```

بدست آوردن کوچک‌ترین عدد تنها کافی است یک تغییر کوچک در برنامه‌ی بالا بدهید. برای تمرین بیشتر برنامه‌ای بنویسید که دو عدد بزرگ‌تر را در آرایه بیابد. (در مثال بالا باید مقادیر ۱۲۴ و ۱۰۰ را باید چاپ نمایید)

(۶) رشته‌ها

همانطور که می‌دانید برای نگه‌داری هر حرف از نوع `char` استفاده می‌کنیم. ممکن است این سوال برای شما پیش آمده باشد که چطور می‌توانیم یک کلمه را نگه‌داری کنیم. مثلاً برای نگه‌داری نام یک کاربر، یک مشتری یا یک دانش‌آموز، برنامه‌های ما نیاز دارند که بتوانند یک یا چند کلمه را به نوعی نگه‌داری کنند. اما چطور می‌توانیم این کار را انجام دهیم؟

از آن جایی که هر کلمه از تعدادی حرف (کاراکتر) تشکیل شده است، برای نگهدازی هر رشته به تعدادی متغیر از نوع `char` نیاز داریم. هر آرایه هم که مجموعه‌ای از متغیرهای همجنس است. پس کاملاً منطقی است که برای نگهداشتن رشته‌ها کافی است آرایه‌ای از کاراکتر‌ها تعریف کنیم. مثلاً برای نگهدازی نام یک فرد (که مثلاً می‌دانیم بیش از ۵۰ حرف نخواهد بود)، می‌توانیم به این شکل یک آرایه تعریف کنیم:

```
char name[50];
```

حالا که این رشته را تعریف کردیم، می‌توانیم هر کلمه یا جمله‌ای که طول آن از ۵۰ کمتر باشد، را در آن نگهدازی کنیم. مثلاً می‌توانیم کلمه‌ی `"ali"` را در آن بروزیم. در این حالت تنها ۳ حرف اولیه از آرایه را استفاده کرده‌ایم و باقی ۴۷ حرف رزرو شده استفاده نشده و مقادیری که در آنها وجود دارد برای ما بی‌معنی و نامعتبر هستند.

حالا احتمالاً این سوال برای تان پیش آمده که بعد از ریختن یک عبارت دلخواه داخل این آرایه، از کجا بدانیم که چند حرف اول این آرایه معنی‌دار و بقیه بی‌معنی هستند؟

سوال به جایی است. برای رسیدن به جواب، به نکته‌ی فنی زیر توجه کنید:

نکته‌ی فنی

زبان C برای مشخص کردن قسمت معنی‌دار یک رشته، بعد از آخرین خانه‌ی معنی‌دار، یک علامت `NULL` می‌گذارد. `NULL` در زبان C یعنی هیچی! و معادل کاراکتری با کد اسکی `0` است. به عنوان مثال اگر بخواهیم در یک رشته، کلمه‌ی `"Ali"` را قراردهیم، باید بعد از آخرین حرف (یعنی در خانه‌ای با اندیس `3`) `NULL` قرار دهیم. در بعضی از مراجع، `NULL` را با `\0` نمایش می‌دهند.



همانطوری که دستور `cin` زحمت گرفتن مقدار از کاربر را برای هر نوع متغیری، تقبل کرده است، در مورد رشته‌ها هم می‌توانیم از `cin` کمک بگیریم. مثلاً

```
char name[50];
cin>>name;
```

دستور `cin` هر رشته‌ای را که به عنوان ورودی می‌گیرد، حرف به حرف درون آرایه قرار داده و خوشبختانه حواسش هم هست که بعد از آخرین حرف معنی‌دار `NULL` قرار دهد.

به طور مشابه دستور `cout` می‌تواند هر رشته‌ای را چاپ کند. به برنامه‌ی زیر توجه کنید:

```
char name[50];
cin>>name;
cout <<"Hello "<<name<<endl;
```

خروجی برنامه:

```
Ali
Hello Ali
Press any key to continue . . .
```

مقداردهی به متغیرها

همچنان، مانند هر متغیر دیگری باید بتوانیم به یک رشته، یک مقدار مشخص بدهیم، توجه کنید که از طریق عملگر `=` نمی‌توانید مقداری در یک رشته برویزید.



خطاهای معمول برنامه نویسی

در صورتی که از عملگر `=` برای ریختن مقدار در یک رشته استفاده کنید، یک خطای زمان کامپایل، جریمه‌ی این عمل نسنجیده شما خواهد بود!

```
char name[50];
name = "Ali";
```

پس چطور می‌توانیم در یک رشته، یک مقدار برویزیم؟

توسط دستور `strcpy` ! به نحوی استفاده از این دستور توجه کنید:

؛) رشته‌ای که قرار است ریخته شود و رشته‌ی مقصد `strcpy()`

رشته‌ای که قرار است ریخته شود، می‌تواند یک رشته‌ی ثابت مثل `"ali"`، یا خود یک متغیر رشته‌ای باشد. به عنوان مثال:

```
char name[50], copy[50];
strcpy(name, "ali");
strcpy(copy, name);
```

در مثال بالا از هر دو حالت استفاده شده است: در ابتدا در رشته‌ی `name` رشته‌ی ثابت `"ali"` ریخته شده است. سپس رشته‌ی `name` که مقدار آن `"ali"` است، در رشته‌ی `copy` ریخته می‌شود.

توجه کنید که دستور `strcpy`، در انتهای رشته، علامت `NULL` را قرار می‌دهد، پس از این بابت آسوده خاطر باشید.

نکته‌ی فنی

در صورتی که نیاز به رشته‌ای دارید که می‌خواهید تعداد مشخصی حرف، مثلاً ۱۰ کاراکتر در آن نگه‌داری کنید، باید هنگام تعریف رشته، طول آن را یک کاراکتر بیش‌تر در نظر بگیرید که بتوانید **NULL** را هم در انتهای آن نگه‌داری کنید. مثلاً اگر می‌خواهید رشته‌ی "ali" را نگه‌داری کنید، طول رشته را باید ۴ تعریف کنید.

خطاهای معمول برنامه نویسی



در صورتی که هنگام تعریف رشته، برای نگه‌داری **NULL** یک کاراکتر بیش‌تر از طول مورد نیاز، در نظر نگیرید، بسته به نوع کامپایلر مورد استفاده، چهار خطای منطقی یا خطای زمان اجرا خواهد شد.

مقایسه‌ی دو رشته

در بسیاری از موارد نیاز پیدا می‌کنید دو رشته را با هم مقایسه کنید و بسته به تساوی یا عدم تساوی آن‌ها، کارهای متفاوتی انجام دهید. مثلاً ممکن است در شروع یک برنامه، اسم رمز (که یک رشته است) را از کاربر گرفته و با مقدار موردنظر مقایسه کنید و اگر باهم برابر بودند (رمز کاربر درست بود)، برنامه را ادامه دهید. دستور **strcmp** این کار را انجام می‌دهد. به ساختار آن توجه کنید:

strcmp(رشته‌ی دوم **،** رشته‌ی اول **)**

این دستور با گرفتن دو رشته، آن‌ها را باهم مقایسه می‌کند. اگر باهم برابر بودند، مقدار صفر را به عنوان نشانه‌ی برابر بودنشان به ما می‌دهد. بدیهی است که اگر جواب آن صفر نبود، یعنی برابر نبوده‌اند.

به کد زیر توجه کنید که اسم رمز را بررسی می‌کند، با فرض این‌که اسم رمز "salam" باشد.

```
char ramz[20];
cout << "ramze khod ra vared konid: ";
cin >> ramz;
if (strcmp(ramz, "salam") == 0)
    cout << "ramze shoma dorost ast!" << endl;
else
    cout << "ramze shoma eshtebah ast!" << endl;
```

در این برنامه، رشته‌ای که کاربر وارد کرده (**ramz**) با رشته‌ی "salam" مقایسه می‌شود و در صورتی که برابر باشند، یعنی جواب دستور **strcmp**، صفر شود، پیام مناسب چاپ می‌شود. خروجی برنامه را یک‌بار برای رمز درست، و یک‌بار برای رمز نادرست، در ادامه ملاحظه می‌فرمایید:

```
ramze khod ra vared konid: salam
ramze shoma dorost ast!
Press any key to continue . . .
```

```
ramze khod ra vared konid: khodahafez
ramze shoma eshtebah ast!
Press any key to continue . . .
```

بقیهی دستورات مورد نیاز

دو دستور دیگر باقیمانده که برای نوشتن برنامه‌هایی که با رشته‌ها کار می‌کنند، به آن‌ها نیاز پیدا خواهید کرد. بدون مقدمه به آن‌ها خواهیم پرداخت:

- دستور **strlen** برای بدست آوردن طول رشته:

```
strlen( رشته‌ی مورد نظر );
```

این دستور به عنوان جواب، طول رشته را به ما می‌دهد. در واقع این دستور تعداد کاراکترهایی را می‌شمارد که از شروع آرایه تا رسیدن به **NULL** قرار دارند.

- دستور **strcat** برای چسباندن یک رشته به انتهای رشته‌ای دیگر:

```
strcat( رشته‌ی دوم + رشته‌ی اول );
```

این دستور رشته‌ی دوم را به انتهای رشته‌ی اول می‌چسباند.

برای درک بهتر این دو دستور، لطفاً به مثال زیر توجه کنید:

```
char str1[10], str2[10];
strcpy(str1, "Hi");
strcpy(str2, "Bye");
strcat(str1, str2);
cout << str1 << endl;
cout << strlen(str1) << endl;
```

در برنامه‌ی بالا، با استفاده از دستور **strcpy**، ابتدا رشته‌های **“Hi”** و **“Bye”** در **str1** و **str2** ریخته شده است. سپس با رشته‌ی **str2** را به انتهای **str1** چسبانده‌ایم. بعد از این با چاپ **str1** انتظار داریم، **“HiBye”** چاپ شود. در نهایت با استفاده از **strlen** طول رشته‌ی **str1** را چاپ کرده‌ایم که باید برابر مقدار ۵ باشد.

```
HiBye
5
Press any key to continue . . .
```

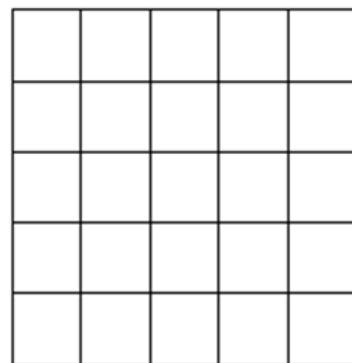
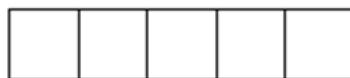
خب، مبحث نسبتاً طولانی رشته‌ها به پایان رسید، امیدواریم که با رشته‌ها، ارتباط دوستانه‌ای برقرار کرده باشید!

۷) آرایه‌های دو بعدی

حال که با آرایه آشنا شدید، نوبت به آرایه‌های دو بعدی می‌رسد. همان‌طور که ما این امکان را داشتیم که مجموعه‌ای از متغیرها را تعریف کنیم (آرایه‌ی یک بعدی)، این امکان را هم داریم که مجموعه‌ای از آرایه‌ها را تعریف کنیم که به آن آرایه‌ی دو بعدی ۳۳ گفته می‌شود. آرایه‌های بک بعدی یک ر دیف از متغیرها بودند، در حالی که آرایه‌های دوبعدی، جدولی از متغیرها هستند. به شکل‌های زیر توجه کنید.

آرایه‌ی یک بعدی ۵ تایی

۵*۵ آرایه‌ی دو بعدی



په نحوهی تعریف آرایهی ۲ بعدی دقت کنید:

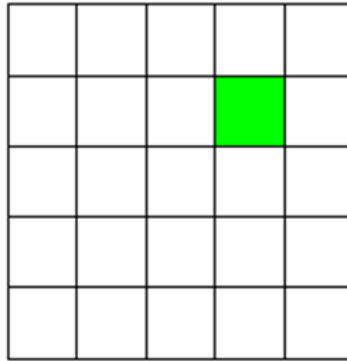
تعداد سطون‌ها [تعداد سطره‌ها] نام آرایه جنس آرایه

به عنوان نمونه، نحوه تعریف آرایه‌ی دو بعدی بالا را در ادامه می‌بینید:

```
int b[5][5];
```

همان طور که می توانید حدس بزنید، برای دسترسی به خانه های آرایه دو بعدی، بعد از آوردن نام آرایه، باید در کروشهی اول شمارهی سطر و در کروشهی دوم، شمارهی ستون مورد نظر را قرار دهید. به عنوان مثال، برای دسترسی به خانهی مشخص شده در آرایه دو بعدی زیر، باید بنویسیم [3][1].a

33 2-Dimensional Array



مقداردهی اولیه به آرایه‌ی دو بعدی

قطعًا یادتان نرفته که چطور آرایه‌های یک بعدی را هنگام تعریف، با آوردن مقادیر خانه‌های آرایه در آکولاد، مقداردهی می‌کردیم. اگر هم یادتان رفته، به خط کد زیر نگاه کنید.

```
int a[5]={4,8,1,5,9};
```

آیا می توانیم از روش مشابهی برای مقداردهی آرایه های دو بعدی استفاده کنیم؟ پاسخ مثبت است. کافیست همهی سطرهای آرایه را مانند آن چه در اینجا می پسندید، مقداردهی کنید.

```
int a[2][3]={{5, 4, 2}, {7, 6, 9}};
```

برای آن که متوجه شوید که آرایه‌های دو بعدی را کامل درک کرده‌اید یا نه، بیایید جدول ضرب 10×10 را در یک آرایه‌ی دو بعدی ذخیره کرده، سپس آن را چاپ کنیم.

```
int a[10][10];
for (int i=0;i<10;i++)
{
    for (int j=0;j<10;j++)
        a[i][j] = (i+1)*(j+1);
}
for (int i=0;i<10;i++)
{
    for (int j=0;j<10;j++)
        cout <<a[i][j]<<"\t";
    cout <<endl;
}
```

در پر نامه ی پالا، "t" یک tab بین هر ستون فاصله می‌اندازد.

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

Press any key to continue . . . =

اگر برنامه‌ی بالا را به درستی نوشته‌ید، نوبت به آن می‌رسد که مدرسه‌ی خود را در محاسبه‌ی معدل دانش‌آموزان کمک نمایید. نحوه‌ی محاسبه‌ی معدل: هر درس یک نمره دارد و یک ضریب که باید در هم ضرب شده و در انتهای مجموع کل بر مجموع ضرایب تقسیم گردد. اگر تعداد درس‌ها ۵ باشد برنامه‌ی فوق را به بهترین نحو بنویسید.

```

float a[2][5];
int i, j;
float avg=0, sum=0;
for(j=0; j<5; j++)
{
    cout<<"nomre :";
    cin>>a[0][j];
    cout<<"zarib :";
    cin>>a[1][j];
}
for(i=0; i<5; i++)
{
    avg+= a[0][i] * a[1][i];
    sum+= a[1][i];
}
cout<<avg/sum <<endl;

```

یادآوری این نکته بی‌فایده نیست که به دلیل آن‌که قرار بوده در `avg` و `sum` مجموع را حساب کنیم، ابتدا آن‌ها را برابر با مقدار صفر قرار داده‌ایم.

تمرین



۱. برنامه‌ای بنویسید که از بین ۱۰۰ عدد ورودی، عددی که بیش از بقیه به میانگین اعداد نزدیک‌تر است را پیدا کرده و چاپ کند.
۲. برنامه‌هایی بنویسید که بدون استفاده از دستورات زیر، عملکرد آن‌ها را شبیه‌سازی کند：
 ۳. برنامه‌ای بنویسید که مشخص کند که آیا رشته‌ی ورودی با معکوس خود برابر است یا نه.
 ۴. برنامه‌ای بنویسید که یک رشته را از ورودی خوانده و حروف موجود در رشته را به همراه تعداد تکرار آن‌ها مشخص کند.
 ۵. برنامه‌ای بنویسید که یک عدد را از کاربر دریافت کرده و نمایش آن را به صورت باینری بدست آورده و چاپ کند.
 ۶. برنامه‌ای بنویسید که یک رشته را به عنوان ورودی گرفته و بزرگ‌ترین عدد صحیح یا اعشاری موجود در رشته (از لحاظ مقدار عددی) را چاپ کند. به عنوان مثال：

"x123addfa4563sd7600d9"	→ 7600
"ad230.65dhg42sdd"	→ 230.65
"2234 323 adf 32 "	→ 2234
 ۷. بازی **0-X** را شبیه‌سازی کنید. دارا بودن
 - محیط گرافیکی کاربر پسند
 - بررسی برد و باخت و اعلام نتیجه در زمان مناسبالزامیست.

فصل نهم: حلقه‌ای تو در تو

۱. حلقه‌ی تو در تو
۲. مثال جدول ضرب
۳. مثال ضرب
۴. الگوریتم مرتب کردن اعداد
۵. برنامه‌ی مرتب کردن اعداد

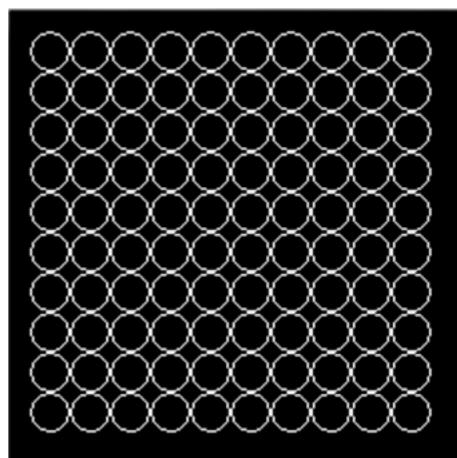
۱) حلقه‌ی تو در تو

فرض کنید بخواهیم با استفاده از حلقه در یک خط ۱۰ عدد دایره‌ی مماس بر هم رسم کنیم و این کار را در ۱۰ خط تکرار کنیم. برای این کار نیاز داریم که در یک حلقه ۱۰ دایره بر روی یک خط رسم کنیم و این کار را ۱۰ بار تکرار کنیم. بنابراین یک حلقه برای رسم ۱۰ دایره در یک خط و حلقه‌ای دیگر برای تکرار این کار در خط‌های دیگر نیاز داریم، به عبارت دیگر نیاز به **حلقه‌های تو در تو**^{۳۴} داریم:

```
main.cpp |  
#include <graphics.h>  
  
int main(int argc, char *argv[])  
{  
    initwindow(300, 300, "Daiere");  
    int shoa = 10;  
    int x;  
    int y;  
    for (x = 0; x < 10; x++)  
    {  
        for (y = 0; y < 10; y++)  
        {  
            circle(20 + (x * shoa * 2), 20 + (y * shoa * 2), shoa);  
        }  
    }  
    system("PAUSE");  
    return EXIT_SUCCESS;  
}
```

همان‌طور که می‌بینید، حلقه‌ی بیرونی مسئول تغییر **X** (طول) مرکز دایره و حلقه‌ی درونی مسئول تغییر **Y** (عرض) مرکز دایره می‌باشد، روند اجرای برنامه بدین صورت می‌باشد که در هر بار اجرای حلقه‌ی بیرونی، حلقه‌ی درونی به طور کامل (در این مثال ۱۰ بار اجرا می‌شود) بنابراین هر بار که **X** تغییر می‌کند، **Y** از صفر تا ۱۰ تغییر می‌کند.

حاصل اجرای این برنامه در شکل زیر آمده است:



³⁴ Nested Loops

سوال: در مثال بالا اگر جای حلقه‌ی بیرونی و درونی عوض می‌شد چه تغییری در اجرای برنامه و نتیجه‌ی کار حاصل می‌شد؟

(۲) مثال جدول ضرب

برنامه‌ای بنویسید که جدول ضرب را بر روی صفحه‌ی نمایش چاپ کند.

برای این کار باید حلقه‌ای تو در تو تعریف کنیم و متغیرهای حلقه را به مقدار ۱ مقداردهی اولیه کنیم، و در هر بار تکرار، حاصل ضرب متغیرهای حلقه را چاپ کنیم. برای حفظ زیبائی جدول رسم شده، تعداد کاراکتر چاپ شده برای هر عدد را یکسان می‌گیریم (با استفاده از دستور **cout.width**) تا تفاوتی میان اعداد دو رقمی و یک رقمی نباشد، برای این منظور از حداقل طول ۴ استفاده کردایم، یعنی هر بار که دستور **cout** اجرا می‌شود، چهار کاراکتر در صفحه چاپ می‌شود و در صورتی که تعداد کاراکترهای مورد نظر کمتر از چهار باشد، به جای بقیه‌ی کاراکترها “ ” (space) چاپ می‌شود.

```
main.cpp |  
-----  
#include <iostream.h>  
  
int main(int argc, char *argv[])  
{  
    int x;  
    int y;  
    for (x = 1; x < 10; x++)  
    {  
        for (y = 1; y < 10; y++)  
        {  
            cout.width(4);  
            cout << x * y;  
        }  
        cout << endl;  
    }  
    system("PAUSE");  
    return EXIT_SUCCESS;  
}
```

جدول ضرب حاصل در شکل زیر آمده است:

1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18
3	6	9	12	15	18	21	24	27
4	8	12	16	20	24	28	32	36
5	10	15	20	25	30	35	40	45
6	12	18	24	30	36	42	48	54
7	14	21	28	35	42	49	56	63
8	16	24	32	40	48	56	64	72
9	18	27	36	45	54	63	72	81

Press any key to continue . . . -

دقیق کنید باید در پایان هر خط جدول ضرب، دستور چاپ `endl` داده شود که به خط بعد برویم.

۳) مثال مثلث

برنامه‌ای بنویسید که با کاراکتر *، شکلی شبیه مثلث در صفحه‌ی نمایش ایجاد کند.

```
main.cpp | #include <iostream.h>

int main(int argc, char *argv[])
{
    int x;
    int y;
    for (x = 0; x < 10; x++)
    {
        for (y = 0; y < x; y++)
        {
            cout << "*";
        }
        cout << endl;
    }
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

شکل حاصل به صورت زیر به دست می‌آید:

```
*  
**  
***  
****  
*****  
*****  
*****  
*****  
*****  
Press any key to continue . . .
```

۴) الگوریتم مرتب کردن اعداد

یکی از کاربردهای مهم حلقه‌ی تو در تو در برنامه‌نویسی، مرتب کردن اعداد است. یکی از ساده‌ترین روش‌های مرتب‌سازی اعداد، روش مرتب‌سازی حبابی^{۳۵} است، در این روش در هر قدم دو خانه‌ی مجاور آرایه مقایسه می‌شوند و در صورتی که عدد بزرگ‌تر در خانه‌ی با

³⁵ Bubble Sort

اندیس کوچک‌تر باشد، مقدار دو خانه تعویض می‌شود، این کار از مقایسه‌ی خانه‌ی اول و دوم آرایه شروع شده و تا خانه‌ی آخر آرایه جلو می‌رود، اما این حلقه، خود باید در یک حلقه‌ی دیگر تکرار شود. به عبارت دیگر عمل مقایسه‌ی دو خانه‌ی کناری و جابجایی باید در یک حلقه‌ی تو در تو انجام شود. در ادامه و با مثال زیر، دلیل این امر را بررسی می‌کنیم. آرایه‌ی زیر را در نظر بگیرید:

19 5 18 10 11 16 9

می‌خواهیم این آرایه را با روش مرتب‌سازی حبابی مرتب کنیم، در گام اول دو عنصر 5 و 19 مقایسه می‌شوند و با توجه به این که 19 از 5 بزرگ‌تر است جای این دو عدد عوض می‌شود:

5 19 18 10 11 16 9

در گام بعدی دو عدد 18 و 19 مقایسه می‌شوند و با توجه به بزرگ‌تر بودن 19 جابجا می‌شوند:

5 18 19 10 11 16 9

این کار تا رسیدن به خانه‌ی پایانی آرایه تکرار می‌شود و با توجه به این که 19 از سایر اعضای آرایه بزرگ‌تر است، در انتهایی حلقه به خانه‌ی پایانی منتقل می‌شود:

5 18 10 11 16 9 19

پس با یک بار اجرای حلقه‌ی داخلی، بزرگ‌ترین عنصر آرایه به جای درست خود (سمت راست آرایه) منتقل می‌شود، اما همان طور که می‌بینید، هنوز آرایه مرتب نشده است، در واقع در هر بار تکرار، بزرگ‌ترین عنصر بین باقی عناصر، به جای خود منتقل می‌شود، یعنی در تکرار بعدی 18 که بزرگ‌ترین عنصر پس از 19 می‌باشد به جای درست خود منتقل می‌شود:

5 10 11 16 9 18 19

پس با فرض قرار داشتن آرایه به ترتیب اولیه‌ی نزولی (که نامرتب‌ترین حالت ممکن است)، این کار باید به اندازه‌ی طول آرایه تکرار شود تا همه‌ی عناصر به جای درست خود منتقل شوند.

۵) برنامه‌ی مرتب کردن اعداد

برنامه‌ای بنویسید که نمره‌ی ۸ درس یک دانش‌آموز را دریافت کرده، آن‌ها را مرتب کرده و نمایش دهد.

```

#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    double nomre[8];
    cout <<"nomarahaie daneshamooz ra vared konid: "<<endl;
    for (int i=0;i<8;i++)
        cin>>nomre[i];
    for (int i=0;i<n-1;i++)
        for (int j=0;j<n-i-1;j++)
            if (a[j]>a[j+1])
            {
                double temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
            }
    cout <<"Nomrehaie moratab shode: "<<endl;
    for (int i=0;i<8;i++)
        cout <<nomre[i]<<endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

برای این کار ابتدا به یک حلقه برای دریافت نمره‌ی ۸ درس دانش‌آموز نیاز داریم، سپس باید در یک حلقه‌ی تو در تو عمل مرتب‌سازی را انجام دهیم، و در نهایت پس از مرتب شدن آرایه آن را در صفحه‌ی نمایش می‌دهیم، برای مثال ۸ نمره‌ی زیر توسط برنامه مرتب شده‌اند:

```

Nomrehaie daneshamooz ra vared konid:
19
18
17.5
20
16.5
17.75
18.5
14
Nomrehaie moratab shode:
14
16.5
17.5
17.75
18
18.5
19
20
Press any key to continue . . .

```

تمرین



۱. برنامه‌ای بنویسید که خروجی زیر را در صفحه‌ی نمایش دهد. تعداد سطرها از کاربر دریافت می‌شود. (تعداد ستاره‌های سطر اول برابر با تعداد سطرهای است)

**
*

۲. برنامه‌ای بنویسید که تا زمانی که کاربر عدد صفر را وارد نکرده، عدد دریافت کرده و مشخص کند که عدد کامل است یا نه. (عدد کامل عددی است که مجموع مقسوم‌علیه‌هایش (جز خودش) برابر با خودش باشند، مثلاً $6 = 1 + 2 + 3$ عددی کامل است زیرا $1 + 2 + 3 = 6$).
۳. برنامه‌ای بنویسید که عددی از کاربر گرفته، همه‌ی اعداد اول قبل از آن عدد را در صفحه نمایش دهد.
۴. در مورد الگوریتم مرتب‌سازی انتخابی^{۳۶} تحقیق کرده و برنامه‌ای برای اجرای آن بنویسید.
۵. برنامه‌ای بنویسید که در حلقه‌ای عدد دریافت کرده (تا هنگامی که صفر وارد شود) و وارون آن‌ها را چاپ نماید.
۶. صفحه‌ی شطرنج (بدون کشیدن مهره‌های آن) را روی صفحه‌ی نمایش رسم کنید.
۷. ۱۰۰ سرباز در یک صف هر کدام در جهت چپ یا راست ایستاده‌اند. با هر بار فرمان مأمور، هر دو سرباز هم‌جواری که رو بروی هم ایستاده‌اند جهت خود را برعکس می‌کنند، اما اگر رو بروی هم نباشند جهت خود را تغییر نمی‌دهند. به عنوان مثال اگر سربازها به صورت $\leftrightarrow \leftrightarrow \leftrightarrow \leftrightarrow$ قرار گرفته باشند، تنها دو نفر میانی ($\leftrightarrow \rightarrow$) رو بروی هم قرار گرفته‌اند و بعد از فرمان سربازها به صورت $\leftrightarrow \leftrightarrow \leftrightarrow \leftrightarrow$ در می‌آیند. برنامه‌ای بنویسید که یک ترتیب اولیه برای سربازها به صورت تصادفی ایجاد کرده و تا زمانی که با فرمان مأمور، صف تغییر نکند، فرمان را اعلام کرده و صفحه‌ها را تغییر دهد. شکل صف را در هر مرحله نمایش دهید.
۸. برنامه‌ای بنویسید که یک جدول 10×10 را با مقادیر تصادفی پر کرده، سپس تمام حلقه‌های آن را به صورت ساعت‌گرد یک خانه دوران دهد. هر دو جدول را (قبل و بعد از دوران) نمایش دهید. به مثال زیر توجه کنید:

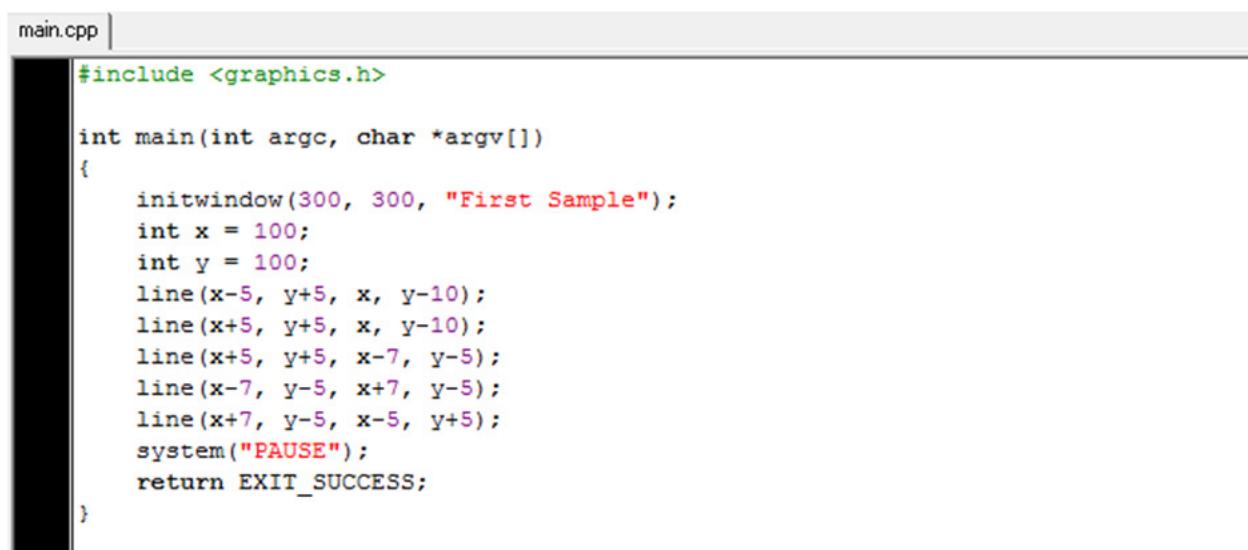
$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 12 & 13 & 14 & 5 \\ 11 & 16 & 15 & 6 \\ 10 & 9 & 8 & 7 \end{bmatrix} \rightarrow \begin{bmatrix} 12 & 1 & 2 & 3 \\ 11 & 16 & 13 & 4 \\ 10 & 15 & 14 & 5 \\ 9 & 8 & 7 & 6 \end{bmatrix}$$

فصل دهم: توابع

۱. اهمیت استفاده از تابع – مثال رسم ستاره
۲. پیدا کردن ماکریم سه عدد صحیح
۳. فرستادن آرایه به عنوان پارامتر به تابع – مثال پیدا کردن میانگین اعداد یک آرایه
۴. حوزه‌ی تعریف متغیرها در تابع
۵. اعداد اول کوچک‌تر از عدد ورودی
۶. مزایای استفاده از تابع

۱) اهمیت استفاده از تابع – مثال رسم ستاره

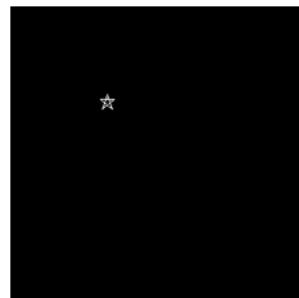
فرض کنید در حال نوشتن یک بازی هستید، که محیط انجام این بازی، شامل آسمان شب است و می‌خواهید در آسمان بازی تان ستاره‌هایی قرار دهید. از آن‌چه که از گرافیک آموخته‌اید، به راحتی می‌توانید کدی مشابه برنامه‌ی زیر برای رسم یک ستاره، تولید کنید:



```
main.cpp | #include <graphics.h>

int main(int argc, char *argv[])
{
    initwindow(300, 300, "First Sample");
    int x = 100;
    int y = 100;
    line(x-5, y+5, x, y-10);
    line(x+5, y+5, x, y-10);
    line(x+5, y+5, x-7, y-5);
    line(x-7, y-5, x+7, y-5);
    line(x+7, y-5, x-5, y+5);
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

در اینجا متغیرهای `X` و `Y` برای مشخص کردن مرکز ستاره در نظر گرفته شده‌اند و ۵ خطی که در ادامه آمده است، بر حسب مکان مرکز ستاره، به سادگی با کشیدن ۵ خط آن را رسم می‌کند. خروجی این برنامه به این شکل خواهد بود:



از آنجایی که آسمان خیلی بیشتر از یک ستاره دارد، برای هر چه طبیعی‌تر شدن محیط بازی، باید تعداد زیادی از این ستاره‌ها را در آسمان قرار دهیم. طبیعتاً هر ستاره باید مرکز متفاوتی داشته باشد، تا ستاره‌ها در مناطق مختلف قرار بگیرند.

پس در طی برنامه، هر جا که لازم شد ستاره رسم کنید، باید ۷ خط مربوط به رسم ستاره را قرار دهید. به عنوان مثال، اگر در ۵۰ مکان از برنامه، نیاز به رسم ستاره داشتید، باید این ۷ خط را در هر ۵۰ مکان بیاورید. به نظر می‌رسد با در اختیار داشتن امکان `Copy` و `Paste`، این کار در دسربازی برای شما نداشته باشد، اما واقعیت چیز دیگری است!

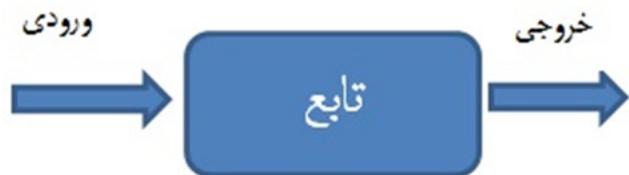
برای پی بردن به عمق فاجعه(!)، فرض کنید بعد از این‌که کد برنامه را نوشته‌ید، تصمیم بگیرید در شکل ستاره تغییری ایجاد کنید. مثلاً آن را توپر یکشید یا رنگ آن را عوض کنید. در این صورت شما باید در تمام کد برنامه، دنبال ۷ خطی‌های رسم ستاره بگردید و در همه‌ی آن‌ها تغییر ایجاد

کنید. این کار بسیار طاقت فرساست، به خصوص اگر ستاره‌های زیادی در برنامه رسم کرده باشید. علاوه بر این، این احتمال وجود دارد که ستاره‌ای از قلم بیفتند و تغییر نکرده باقی بماند.

به نظر می‌رسد که تصمیم ما برای استفاده از **Copy** و **Paste**، تصمیم عاقلانه‌ای نبوده است. برای حل این مشکل، باید در ابتدا بهمیم مشکل از کجا آمده است. منبع مشکل این جاست که کد رسم ستاره را در برنامه تکرار کرده‌ایم و در نتیجه برای تغییر شکل ستاره، باید همه تکرارها را تغییر دهیم.

اگر این امکان وجود داشت که ما یکبار کد مربوط به رسم ستاره را می‌آوردیم و هر بار برای رسم، صرفاً برنامه را به آن کد ارجاع می‌دادیم، این مشکل کاملاً برطرف می‌شد، چراکه برای تغییر در شکل ستاره‌های برنامه، تنها کافی بود همان کدی را که آن را تنها یک بار آورده‌ایم، تغییر دهیم.

خوبی‌خانه در زبان C، این امکان در نظر گرفته شده است. استفاده از **تابع**^{۳۷}، امکانی است که C در اختیار ما قرار داده است. کافیست قطعه کدی که قرار است تکرار شود را یک جای برنامه بیاوریم و به آن قطعه کد، نامی نسبت بدهیم و هر جای برنامه که نیاز است این کد اجرا شود، فقط نام آن قطعه کد را بیاوریم، به این قطعه کدها که هر کدام نامی دارند، تابع می‌گوییم. هر تابع، مانند یک دستگاه است: تعدادی ورودی دارد، یک سری کار انجام می‌دهد و بر این اساس، خروجی را تولید می‌کند.



ورودی‌های تابع، مقادیری هستند که قطعه کد مورد نظر برای انجام کار خود، به آن‌ها احتیاج دارد. به عنوان مثال برای رسم ستاره، مختصات مرکز آن، مقادیر ورودی هستند. به ورودی‌های تابع، پارامتر^{۳۸} یا آرگومان^{۳۹} نیز گفته می‌شود. در بعضی موارده حاصل محاسباتی که در کد تابع انجام شده به صورت یک عدد محاسبه خواهد شد. مثلاً ممکن است تابعی داشته باشید که به عنوان ورودی، ۲ عدد بگیرد و میانگین آن‌ها را حساب کند. در اینجا این میانگین به عنوان خروجی، در نظر گرفته می‌شود.

نکته‌ی فنی

دقت کنید ممکن است تابعی نیاز به ورودی نداشته باشد. همچنین توجه کنید که تابعی ممکن است عددی را به عنوان خروجی خود معرفی نکند. به عنوان مثال، تابع رسم ستاره، صرفاً یک ستاره را روی صفحه‌ی نمایش رسم می‌کند و عددی را به عنوان خروجی خود ندارد.

فرمت کلی تابع در زبان C به شکل زیر است:

³⁷ Function

³⁸ Parameter

³⁹ Argument

```

        نوع خروجی    (لیست ورودی‌ها )  اسم تابع
{
    کد تابع
}

```

در ادامه، به عنوان نمونه، تابعی را که یک ستاره را با گرفتن مختصات مرکز آن رسم می‌کند، مشاهده می‌کنید:

```

نوع خروجی    نام تابع    نوع و نام ورودی اول    نوع و نام ورودی دوم
                ↑          ↑          ↑          ↑
                |          |          |          |
                void rasmeSetare(int x, int y)
{
    line(x-5, y+5, x, y-10);
    line(x+5, y+5, x, y-10);
    line(x+5, y+5, x-7, y-5);
    line(x-7, y-5, x+7, y-5);
    line(x+7, y-5, x-5, y+5);
}

```

کد تابع

کلمه‌ی کلیدی **void** که در برنامه‌ی بالا به عنوان نوع خروجی آمده، نشان دهنده این است که تابع، عددی به عنوان خروجی ندارد. اسم تابع نام دلخواهی است که قواعد نام متغیرها برای آن صدق می‌کند. (به توصیه‌ی اخلاق برنامه‌نویسی زیر دقت کنید). در قسمت لیست ورودی‌ها که داخل پرانتز آمده است، هر متغیر ورودی به ترتیب با تعیین کردن نوع و نام آن معرفی می‌شود. همان‌طور که دیده می‌شود، بین تعریف هر دو ورودی، یک علامت کاما قرار می‌گیرد.

اخلاق برنامه‌نویسی

علاوه بر قواعد زبان C که برای نام‌گذاری توابع، مشابه نام‌گذاری متغیرها وجود دارد، یک سری قرارداد یا به عبارت دیگر، توصیه‌ی اخلاقی برای نام‌گذاری توابع، بین برنامه‌نویسان وجود دارد.

اول این‌که لطفاً نام توابع را متناسب با کاری که انجام می‌دهند، انتخاب کنید. هیچ اشکالی ندارد اگر این نام بیش از یک کلمه شود، (مانند **rasmeSetare**، فقط در این صورت باید کلمه‌ی دوم را با حرف بزرگ شروع کنید تا خوانایی آن ساده‌تر شود. (عدم نام‌گذاری به شکل **rasmesetare** که در آن کلمه‌ی دوم با حرف کوچک شروع شده است.)

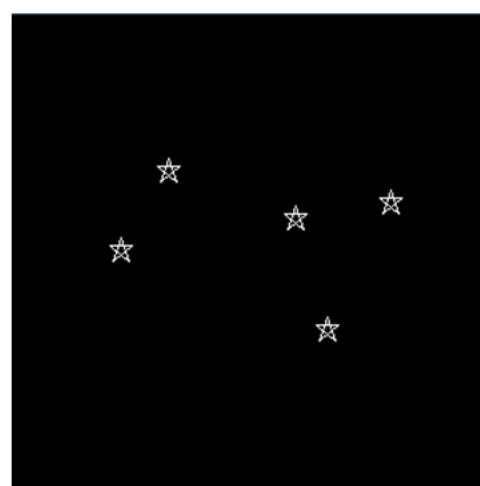
حالا که کد مربوط به رسم ستاره را یکبار به صورت یک تابع مشخص کردیم، هریکار که نیاز بود از این کد استفاده کنیم، کافی است تابع را صدا بزنیم، یعنی نام تابع را با ورودی‌های مورد نیاز بیاوریم.

مثالاً در جایی از برنامه می‌توانیم تابع را به صورت `rasmeSetare(100, 150);` صدا بزنیم. با این فراخوانی مقدارهای 100 و 150 به ترتیب در متغیرهای ورودی `X` و `Y` قرار می‌گیرند و کد تابع با این مقادیر `X` و `Y` اجرا می‌شود. نتیجه‌ی این صدا زدن تابع، مطابق انتظار ما رسم یک ستاره به مختصات مرکز (100, 150) خواهد بود.

در صفحه‌ی بعد، برنامه‌ای که 5 ستاره را با استفاده از تابع رسم می‌کند، آمده است:

```
main.cpp |  
#include <graphics.h>  
  
void rasmeSetare(int x, int y)  
{  
    line(x-5, y+5, x, y-10);  
    line(x+5, y+5, x, y-10);  
    line(x+5, y+5, x-7, y-5);  
    line(x-7, y-5, x+7, y-5);  
    line(x+7, y-5, x-5, y+5);  
}  
  
int main(int argc, char *argv[])  
{  
    initwindow(300, 300, "Aseman!");  
  
    rasmeSetare(100, 100);  
    rasmeSetare(200, 200);  
    rasmeSetare(70, 150);  
    rasmeSetare(240, 120);  
    rasmeSetare(180, 130);  
  
    system("PAUSE");  
    return EXIT_SUCCESS;  
}
```

خروجی این برنامه هم به شکل زیر خواهد بود:



نکته‌ی فنی

در قطعه کد بالا، ملاحظه می‌فرمایید که تابع `rasmeSetare` قبل از تابع `main` (جایی که از آن استفاده شده) تعریف شده است. دلیل این است که اگر ابتدا از آن استفاده و سپس آن را معرفی می‌کردیم، بدین وسیله موجبات شگفت‌زدگی کامپایلر را فراهم کرده و یک خطای زمان کامپایل را نصیب خود نموده بودیم!

خطاهای معمول برنامه نویسی



تعریف یک تابع بعد از جایی که استفاده شده است، سبب ایجاد یک خطای زمان کامپایل خواهد شد.

۲) پیدا کردن ماکزیمم ۳ عدد صحیح

این بار قرار است برنامه‌ای بنویسیم که بین سه عدد صحیح، ماکزیمم آن را چاپ می‌کند. در اینجا برای پیدا کردن ماکزیمم ۳ عدد، یک تابع ایجاد می‌کنیم:

```
main.cpp |  
#include <iostream.h>  
  
int maximum3(int a, int b, int c)  
{  
    int max = a;  
    if (b>max)  
        max = b;  
    if (c>max)  
        max = c;  
    return max;  
}  
  
int main(int argc, char *argv[])  
{  
    int x = 7, y = 10, z = 3;  
    int m = maximum3(x, y, z);  
    cout << "Maximume in 3 adad: " << m << endl;  
    system("PAUSE");  
    return EXIT_SUCCESS;  
}
```

خروجی این برنامه به این شکل خواهد بود:

```
Maximume in 3 adad: 10  
Press any key to continue . . .
```

اولاً توجه کنید که خروجی تابع `maximum3` از نوع `int` معرفی شده است. دلیل این است که ماکریزم `3` عدد از نوع صحیح، خود از نوع عدد صحیح (`int`) خواهد بود. اما یک کلمه‌ی کلیدی جدید در تابع `maximum3` دیده می‌شود: کلمه‌ی کلیدی `.return`.

در توابعی که مقدار خروجی دارند، یعنی خروجی تابع از نوع `void` نیست، بعد از این که مقدار خروجی مشخص شد، باید به برنامه مقدار خروجی را اعلام کنیم. این کار توسط دستور `return` انجام می‌شود و پس از آن، اجرای تابع تمام می‌شود. به عنوان مثال در اینجا بعد از دو دستور `if` در تابع `maximum3` مقدار ماکریزم `3` عدد، در متغیر `max` قرار گرفته است (در اینجا عدد `10`). حالا که این مقدار محاسبه شده و کار تابع به پایان رسیده است، باید با استفاده از دستور `return` این مقدار را اعلام کنیم و به این ترتیب از تابع خارج شویم.

همان‌طور که می‌بینید در خط دوم `main` متغیر `m` قرار داده‌ایم. این بدان معنی است که مقدار خروجی تابع `maximum3` (همان مقداری که درون تابع `return` شده است) را در متغیر `m` بروزیم.

نکته‌ی فنی

توجه کنید که `return` آخرین دستوری است که در هر تابع اجرا می‌شود. پس اگر قطعه کدی بعد از `return` قرار داشته باشد، اجرا نمی‌شود.

خطاهای معمول برنامه نویسی



قراردادن علامت `;` بعد از خط اول تعریف تابع، خطای خیلی رایجی هنگام شروع برنامه‌نویسی با توابع می‌باشد:

```
void rasmeSetare(int x, int y)  ;  
{  
    .....  
}
```

نکته‌ی فنی

آخرین نکته‌ی مربوط به این مثال، این است که شما، قبل از این که با مفهوم تابع آشناشی پیدا کنید، هم از تابع استفاده کرده بودید، چرا که از `main` در برنامه‌ی خود استفاده می‌کردید. `main` خود یک تابع است که در هر برنامه‌ای باید وجود داشته باشد. همان‌طور که می‌بینید خروجی تابع `main` از نوع `int` می‌باشد که مقدار مورد نظر، در آخرین خط `return` شده است. (`EXIT_SUCCESS` در واقع یک عدد صحیح است که نشان می‌دهد برنامه با موفقیت اجرا شده است، چرا که اگر در اجرای برنامه مشکلی پیش می‌آمد، به پایان برنامه نمی‌رسیدیم).

۳) فرستادن آرایه به عنوان پارامتر به تابع - مثال پیدا کردن میانگین اعداد یک آرایه

همان طور که متغیرها را می‌توانیم به یک تابع بفرستیم، آرایه‌ها را هم می‌توانیم به عنوان پارامتر به یک تابع ارسال کرده و از مقادیر خانه‌های آن استفاده کنیم. حتیً یادتان هست که هنگام نوشتن لیست پارامترها، برای هر پارامتر به ترتیب نوع و اسم آن را مشخص می‌کردیم. برای مشخص کردن آرایه در لیست پارامترها، از قالب زیر استفاده می‌کنیم:

نوع خروجی جنس آرایه (..., [] اسم آرایه و ...) اسم تابع

به عنوان مثال، اگر بخواهیم آرایه‌ای از نوع اعداد صحیح به نام `a` در پارامترها داشته باشیم، از `[]` در تعریف تابع استفاده می‌کنیم. دقت کنید که در این جا نیازی نیست مانند وقتی که آرایه را تعریف می‌کردیم، طول آرایه را درون کروشه مشخص کنیم. هنگام صدا زدن تابع هم کافیست تنها اسم آرایه را که می‌خواهیم بفرستیم، بیاوریم.

برای این که قضیه روش شود، به مثال زیر توجه کنید:

فرض کنید قرار است میانگین اعداد یک آرایه از اعداد صحیح را محاسبه کنیم. برای این منظور تابعی ایجاد می کنیم که آرایه و تعداد عناصر آن را بگیرد و میانگین را برگرداند:

```
main.cpp | #include <iostream.h>

float miangin(int a[], int tool)
{
    float sum = 0;
    for (int i=0;i<tool;i++)
    {
        sum += a[i];
    }
    sum /= tool;
    return sum;
}

int main(int argc, char *argv[])
{
    int a[4] = {2, 3, 4, 5};
    float m = miangin(a, 4);
    cout << "Miangine in 4 adad: " << m << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

فرمت آرایه به عنوان پارامتر در تعریف تابع

آوردن نام آرایه هنگام صدار زدن تابع

خروجی، این برنامه به شکل زیر است:

Miangine in 4 adad: 3.5
Press any key to continue . . .

به فرمت تعریف پارامتر آرایه در تعریف تابع توجه کنید که علامت [] بعد از نام پارامتر، نشان دهنده آرایه بودن آن است. همانطور که در برنامه‌ی بالا مشهود است، هنگام صدا زدن تابع هم کافیست اسم آرایه را به عنوان پارامتر مربوطه بیاوریم. لطفاً این ساختار را به خاطر بسپارید.

۴) حوزه‌ی تعریف متغیرها در توابع

به برنامه‌ی ساده‌ای که در ادامه آمده، توجه کنید:

```
#include <iostream.h>

float taghir(int a)
{
    a = 20;
}

int main(int argc, char *argv[])
{
    int a = 10;
    taghir(a);
    cout << "meghdare a:" << a << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

همان‌طور که از نام تابع در ابتدای برنامه بر می‌آید، به نظر می‌رسد این تابع قصد دارد مقدار متغیر **a** را که مقدار اولیه‌ی 10 دارد، به 20 تغییر دهد. همه چیز هم مرتب به نظر می‌رسد: متغیر **a** به عنوان پارامتر به تابع **taghir** داده شده و در آن‌جا مقدار این پارامتر به 20 تغییر می‌یابد. اما وقتی این برنامه را اجرا کنید، خروجی چیز دیگری می‌گوید:

```
meghdare a:10
Press any key to continue . . .
```

دلیل این‌که **a** مقدار 20 به خود نگرفته و همچنان مقدار 10 را نگه داشته، چیست؟ دلیل آن این نکته مهم است که وقتی یک متغیر را هنگام فراخوانی به یک تابع ارسال می‌کنید، خود این متغیر را به تابع نمی‌دهید، بلکه تنها مقدار آن را می‌دهید و پارامتر متناظر موجود در تابع، متغیر دیگری است. پس ما در این برنامه، دو متغیر متفاوت به نام **a** داریم:

- متغیر **a** که متعلق به تابع **main** است.
- متغیر **a** که متعلق به تابع **taghir** است.

همان‌طور که گفته شد، تنها ربط این دو متغیر این است که در شروع تابع **taghir**، مقدار هر دو برابر است. اما اگر در تابع **taghir** مقدار **a** را عوض کنیم، تاثیری در متغیری که متعلق به **main** است، نخواهد داشت. اصولاً به دلیل همین بی‌ربطی است که لزومی ندارد این متغیر ارسال شده و پارامتر تابع، نامهای یکسانی داشته باشند. (همانطور که در مثال محاسبه ماتریس ۳ عدد دیدیم،

اما در این مورد یک استثنا وجود دارد. مثال صفحه‌ی بعد، این استثنا را نشان می‌دهد:

```

main.cpp |

#include <iostream.h>

void zarbdar2(int b[], int tool)
{
    for (int i=0;i<tool;i++)
    {
        b[i] *= 2;
    }
}

int main(int argc, char *argv[])
{
    int a[5] = {1, 2, 3, 4, 5};
    zarbdar2(a, 5);

    cout <<"a[]=";
    for (int i=0;i<5;i++)
        cout<< a[i]<< " ";
    cout <<")" << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

در این برنامه، آرایه `a` به تابع `zarbdar2` داده شده و این تابع سعی دارد هر کدام از خانه‌های آن را دو برابر کند. اگر طبق نکته‌ی قبلی بخواهیم نتیجه‌گیری کنیم، این تابع نباید بتواند خانه‌های آرایه‌ی تعریف شده در تابع `main` را تغییر دهد، اما اگر نگاهی به خروجی برنامه زیر بیندازید، می‌بینید که این تغییر رخ داده است:

```

a[]={2 4 6 8 10 }
Press any key to continue . . .

```

نکته‌ی فنی

می‌توان این استثنای این طور بیان کرد که برخلاف متغیرهای عادی، در حالتی که پارامتر ارسالی از نوع آرایه باشد، مقدار خانه‌های آن با تغییر در تابع، بیرون تابع هم تغییر می‌کنند. برای فهمیدن دلیل این نکته تا فصل چهاردهم (اشارة‌گرها) باید صبر کنید!

۵) اعداد اول کوچک‌تر از عدد ورودی

در این مثال از ما خواسته شده است، برنامه بنویسیم که یک عدد از ورودی گرفته و اعداد اول کوچک‌تر از آن را چاپ کند. قسمتی از کد که تکرار می‌شود، تشخیص اول بودن یا نبودن یک عدد است. اگر این مسئله را حل کنیم (برای آن یک تابع بنویسیم)، کافی است اول بودن یا نبودن اعداد کوچک‌تر از عدد وارد شده را با استفاده از این تابع مشخص کنیم، پس منطقی است که یک تابع داشته باشیم که یک عدد به عنوان پارامتر گرفته، مشخص کند که اول است یا نه.

```

main.cpp | 
#include <iostream.h>

bool avval(int a)
{
    for (int i=2;i<=a/2;i++)
    {
        if (a%i == 0)
            return false;
    }
    return true;
}

int main(int argc, char *argv[])
{
    cout <<"adade n ra vared konid:";
    int n;
    cin>>n;
    for (int i=2;i<n;i++)
    {
        if (avval(i)==true)
            cout <<i<<" ";
    }
    cout <<endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

این هم خروجی برنامه:

```

adade n ra vared konid:20
2 3 5 7 11 13 17 19
Press any key to continue . . .

```

۶) مزایای استفاده از تابع

از آن جایی که قبلاً از آشنایی با تابع هم برنامه‌هایی نوشته بودیم، واضح است که این امکان وجود دارد که بدون استفاده از تابع (غیر از تابع **main**) هم برنامه‌هایمان را بنویسیم. پس دلیل این که یک فرمت جدید از زبان را برای توابع یاد گرفته و استفاده کنیم، چیست؟ خوشبختانه دلایل خیلی زیادی وجود دارد که برای ساده‌تر شدن کار خودمان، زحمت [اندک] استفاده از تابع را به جان بخربیم. برای جمعبندی مبحث مربوط به تابع، این فصل را با اشاره‌ای به برخی محسن استفاده از توابع در برنامه‌سازی به پایان می‌بریم:

- **حجم پایین‌تر کرد**

توابع اصولاً در شرایطی به کار می‌روند که یک یا چند قطعه کد، چندین بار قرار است در برنامه استفاده شوند. مثلاً یک قطعه کد ۵۰ خطی ممکن است در برنامه‌ای ۲۰۰ بار نیاز به اجرا داشته باشد. در این شرایط استفاده از تابع، حجم کد را به شکل قابل ملاحظه‌ای کاهش می‌دهد. از آن جایی که برنامه‌ها روی حافظه بارگذاری می‌شوند، به این ترتیب به حافظه‌ی اصلی کمتری هم نیاز خواهیم داشت.

برنامه‌نویسی سریع‌تر

طبعیتاً با پایین آمدن حجم کد، زمان برنامه‌نویسی آن نیز کاهش پیدا می‌کند.

تغییر راحت‌تر

همان‌طور که در مثال اول این فصل متذکر شدیم، با قرار دادن کدهای تکرار شونده در یک تابع، در صورت نیاز به تغییر کارکرد این قطعه کد، دیگر نیاز نیست در تمام برنامه دنبال این کد بگردیم و آن را بارها تغییر دهیم. تنها کافیست کد درون تابع را یک‌بار تغییر دهیم. در این حالت، اصطلاحاً گفته می‌شود که برنامه انعطاف‌پذیری^{۴۰} بالایی دارد.

اشکال زدایی^{۴۱} ساده‌تر

در صورتی که در یک قطعه کد، ایرادی وجود داشته باشد، اگر این کد تکرار شود، این ایراد در تمامی برنامه پخش شده و تکرار می‌شود. اما اگر این کد مشکل‌دار در یک تابع باشد، کافی خواهد بود تنها یک‌بار اشکال زدایی را در آن تابع انجام دهیم. و این یعنی راحتی بیشتر برای اشکال زدایی.

خوانایی بیشتر

مثال رسم ستاره‌ها را به خاطر بیاورید. در صورتی که برای رسم هر ستاره از تابع استفاده نمی‌کردیم، باید در مکان‌های متفاوتی از برنامه این ۷ خط را کپی می‌کردیم که این باعث کاهش خوانایی برنامه می‌شد. اما در شرایطی که از تابع استفاده می‌کنیم، با توجه به بامعنی بودن اسم تابع در هر قسمت از برنامه، مشخص است که چه کاری در حال انجام شدن است. به عبارتی دیگر ما با استفاده از توابع، برای قسمت‌های مختلف برنامه، یک اسم توصیف کننده (ن‌تابع) خواهیم داشت که برای فهم بهتر برنامه، خیلی موثر است.

استفاده‌ی مجدد

وقتی کدهای مربوط به برخی وظایف مشخص را از کدهای دیگر برنامه‌تان جدا می‌کنید و در توابع قرار می‌دهید، می‌توانید در صورت نیاز مجدد به این وظایف در برنامه‌های دیگر، این توابع را به راحتی در آن‌ها استفاده کنید. بنابراین استفاده از توابع قابلیت استفاده‌ی مجدد^{۴۲} برنامه‌ی شما را افزایش می‌دهد.

استفاده از تکنیک "تفرقه بیانداز و حکومت کن"^{۴۳}!

مسائلی که در دنیای واقعی با آن مواجه هستیم، مسائل بزرگی هستند، که برنامه‌ی مربوط به آن‌ها ممکن است به هزاران خط برسد. در این حالت شما می‌توانید وظایف برنامه‌ی خود به چندین مسئله کوچک‌تر (چندین تابع مختلف) تقسیم کنید، که حل هر کدام از این مسائل کوچک‌تر، بسیار ساده‌تر خواهد بود. در این‌جا این امکان نیز وجود دارد که برنامه‌نویسی برای هر کدام از این تابع‌ها، به یک فرد از اعضای تیم برنامه‌نویسی سپرده شود و در نتیجه سرعت برنامه‌نویسی برای آن مسئله، بالاتر برود.

⁴⁰ Flexibility

⁴¹ Debugging

⁴² Reusability

⁴³ Divide and Conquer

اخلاق برنامه‌نویسی



همه‌ی موارد ذکر شده، اهمیت فراوان عادت خوب برنامه‌نویسی با استفاده‌ی مناسب از توابع را نشان می‌دهد. به این نوع برنامه‌نویسی با استفاده از توابع در موقعیت‌های مناسب، **برنامه‌نویسی پیمانه‌ای^{۴۴}** گفته می‌شود. لطفاً سعی کنید اخلاق برنامه‌نویسی خود به برنامه‌نویسی پیمانه‌ای، مزین کنید!

⁴⁴ Modular Programming

تمرین



۱. تابعی بنویسید که عملیات به توان رساندن را انجام دهد. این تابع دو پارامتر می‌گیرد و پارامتر اول را که از نوع عدد اعشاری است، به توان پارامتر دوم که عددی صحیح (مثبت یا منفی) می‌باشد، می‌رساند.

۲. تابعی بنویسید که با گرفتن X و تعداد جملات تقریب (n) به عنوان پارامتر، از روی سری زیر مقدار $\cos(x)$ را تقریب بزند. دقت کنید که تابع شما باید مقدار $\cos(x)$ را برگرداند.

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots$$

۳. در مورد توابع ریاضی زیر که در کتابخانه `math` قرار دارند، تحقیق کنید که چه کاری انجام می‌دهند و ورودی‌ها و چه خروجی‌آن‌ها از چه نوعی هستند:

الف. `floor`

ب. `sin`

ج. `pow`

۴. تابع `rand()` یک عدد صحیح تصادفی بزرگتر از صفر برمی‌گرداند. تابعی بنویسید که دو عدد صحیح `a` و `b` را به عنوان پارامتر گرفته و با استفاده از تابع `rand()`، یک عدد صحیح بزرگتر از `a` و کوچکتر از `b` برگرداند.

۵. تابعی بنویسید که عدد `n` را به عنوان پارامتر بگیرد و تعداد اعداد اول بین `n!` و `(n+1)!` که در ارقامشان عدد ۷ وجود دارد اما ۸ نیست را محاسبه کرده و بازگرداند. (به عنوان مثال اگر `n=3` به عنوان پارامتر داده شود تنها اعداد ۷ و ۱۷ شامل شروط بالا هستند و در نتیجه تابع باید عدد ۲ را به عنوان مقدار خروجی برگرداند.)

۶. شرکت `X` تصمیم به اعطای جایزه به مشتریانی که بیشترین خرید را از محصولات این شرکت داشته‌اند، گرفته است. مسئول بخش انفورماتیک شرکت، اطلاعات مربوط به خرید مشتریان را در آرایه‌ای ریخته است، بدین ترتیب که به ازای هر بار خرید یک مشتری، شناسه‌اش یک بار در آرایه آمده است. به او کمک کنید که مشتریان را بر اساس دفعات خرید مرتب نماید. به بیان دیگر تابعی بنویسید که آرایه‌ای از شناسه‌ها و طول آرایه را دریافت نموده، آرایه‌ی مرتب شده مشتریان بر اساس تعداد خرید را بازگرداند.

مثال: اگر اطلاعات مربوط به خریدهای ۳ مشتری با شناسه‌های ۲۲، ۳۴، ۵۷ به صورت ۲۲، ۳۴، ۵۷ باشد. خروجی تابع به صورت ۲۲، ۵۷، ۳۴ خواهد بود.

۷. تابعی بنویسید که دو رشته به عنوان پارامتر گرفته و رشته‌ی دوم را در رشته‌ی اول جستجو کرده و اندیس اولین محل وقوع رشته‌ی دوم در رشته‌ی اول را برگرداند. به عنوان مثال محل وقوع رشته‌ی "C++" در رشته‌ی "I like C++" ۷ می‌باشد. توجه کنید که رشته‌ی دوم در رشته‌ی اول موجود نباشد، تابع شما باید ۱-برگرداند.

۸. می‌گوییم یک آرایه `n`-پاره است اگر بتوان آن را به `n` قسمت تقسیم کرد به طوری که این قسمت‌ها به ترتیب صعودی و نزولی باشند. مثلاً آرایه‌ی ۱، ۳، ۵، ۴، ۲ دوپاره است. تابعی بنویسید که یک آرایه و طول آن را گرفته و به عنوان مقدار بازگشته مشخص کند چندپاره است.

فصل یازدهم: توابع بازگشتی

۱. معرفی برنامه‌نویسی بازگشتی
۲. مثال محاسبه‌ی فاکتوریل
۳. مثال محاسبه‌ی عدد 127 فیبوناچی
۴. مثال برج‌های هانوی
۵. مثال جستجوی باینری

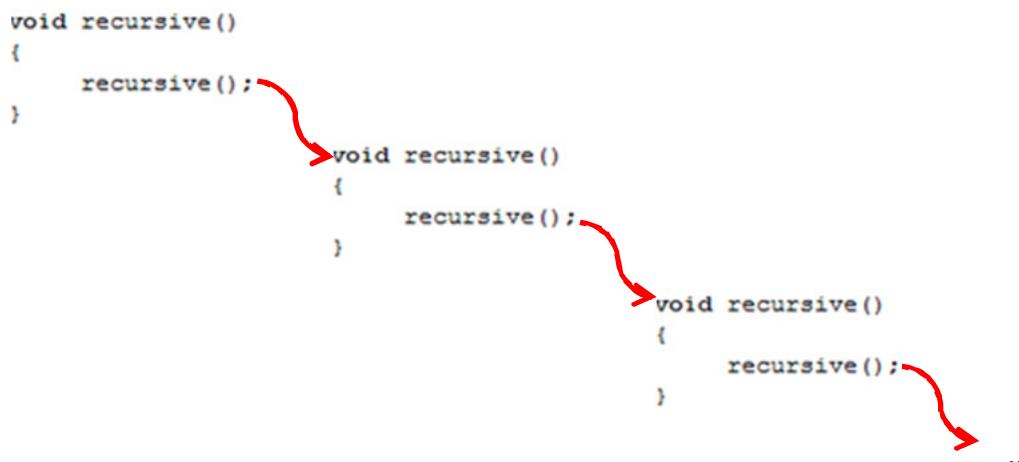
(۱) معرفی برنامه نویسی بازگشته

حتماً یادتان هست که در فصل گذشته توابعی را تعریف کرده و در صورت نیاز آنها را صدای کردیم. فرض کنید تابعی به نام `f` دارید که برای انجام وظایف محله‌اش چندین بار نیاز به اجرای قطعه کد مشخصی داشته باشد. همانطور که قبلاً هم گفتیم، در این شرایط، راه عاقلانه این است که این قطعه کد تکراری را در تابع دیگری مثلاً^{۴۵} به نام `g` قرار دهیم و هربار که نیاز بود، این تابع را صدای بزنیم. پس نه تنها هیچ اشکالی ندارد که در یک تابع مثل `f`، تابع دیگری مثل `g` را صدای بزنیم، بلکه می‌تواند بسیار هم پسندیده باشد.

اما اگر تابع `f` در متن خود، خودش را صدای بزنند چطور؟ آیا اصلاً این کار امکان‌پذیر است؟ پاسخ این است که علاوه بر این کار در زبان برنامه‌نویسی C ممکن است، به عنوان یکی از جالب‌ترین و خلاقانه‌ترین تکنیک‌های برنامه نویسی شناخته شده به آن برنامه‌نویسی بازگشته^{۴۵} گفته می‌شود. این روش منجر به حل مسائلی می‌شود که بدون استفاده از این تکنیک، تقریباً غیر ممکن هستند. در این فصل سعی داریم یاد بگیریم چطور از این روش استفاده کنیم. برنامه‌ی ساده‌ی زیر را در نظر بگیرید:

```
main.cpp |  
-----  
#include <iostream.h>  
  
void recursive()  
{  
    recursive();  
}  
  
int main(int argc, char *argv[])  
{  
    recursive();  
    system("PAUSE");  
    return EXIT_SUCCESS;  
}
```

اتفاقی که در این برنامه می‌افتد، این است که در تابع `main` تابع `recursive` صدای زده می‌شود. در تابع `recursive` مجدداً همین تابع فراخوانی می‌شود و ... شکل زیر تصویری از این فراخوانی‌ها ارائه می‌دهد:



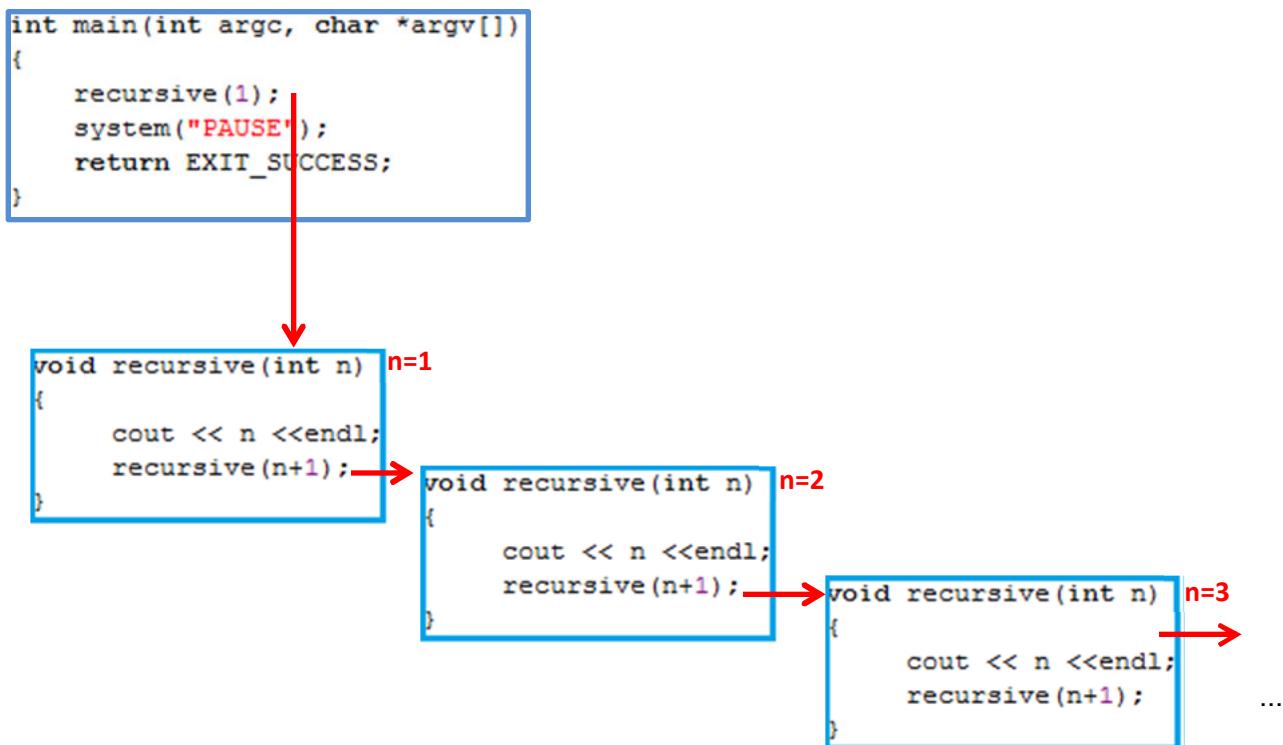
⁴⁵ Recursive

هر تابعی، تابع دیگری از نوع خود را صدای زند و این فراخوانی‌ها تا ابد ادامه دارد...

این بار می‌خواهیم هر تابعی که فراخوانی می‌شود، این‌که چندمین تابع است را چاپ کند. این کار را به این شکل می‌توانیم انجام می‌دهیم:

```
main.cpp |  
#include <iostream.h>  
  
void recursive(int n)  
{  
    cout << n << endl;  
    recursive(n+1);  
}  
  
int main(int argc, char *argv[])  
{  
    recursive(1);  
    system("PAUSE");  
    return EXIT_SUCCESS;  
}
```

باید ببینیم چطور هر تابع شماره‌ی خود را بدست می‌آورد: یک پارامتر به نام n در تابع اضافه کردیم که قرار است شماره‌ی تابع را در خود نگه دارد. اولین بار که تابع صدای زده می‌شود (در تابع `main` مقدار 1 به عنوان شماره‌ی اولین تابع به آن داده شده است. در هر مرحله از اجرای تابع، بعد از این‌که شماره‌ی تابع چاپ شد، شماره‌ی تابع بعدی یکی بیشتر می‌شود ($n+1$). شکل زیر، اجرای این برنامه را به تصویر کشیده است:



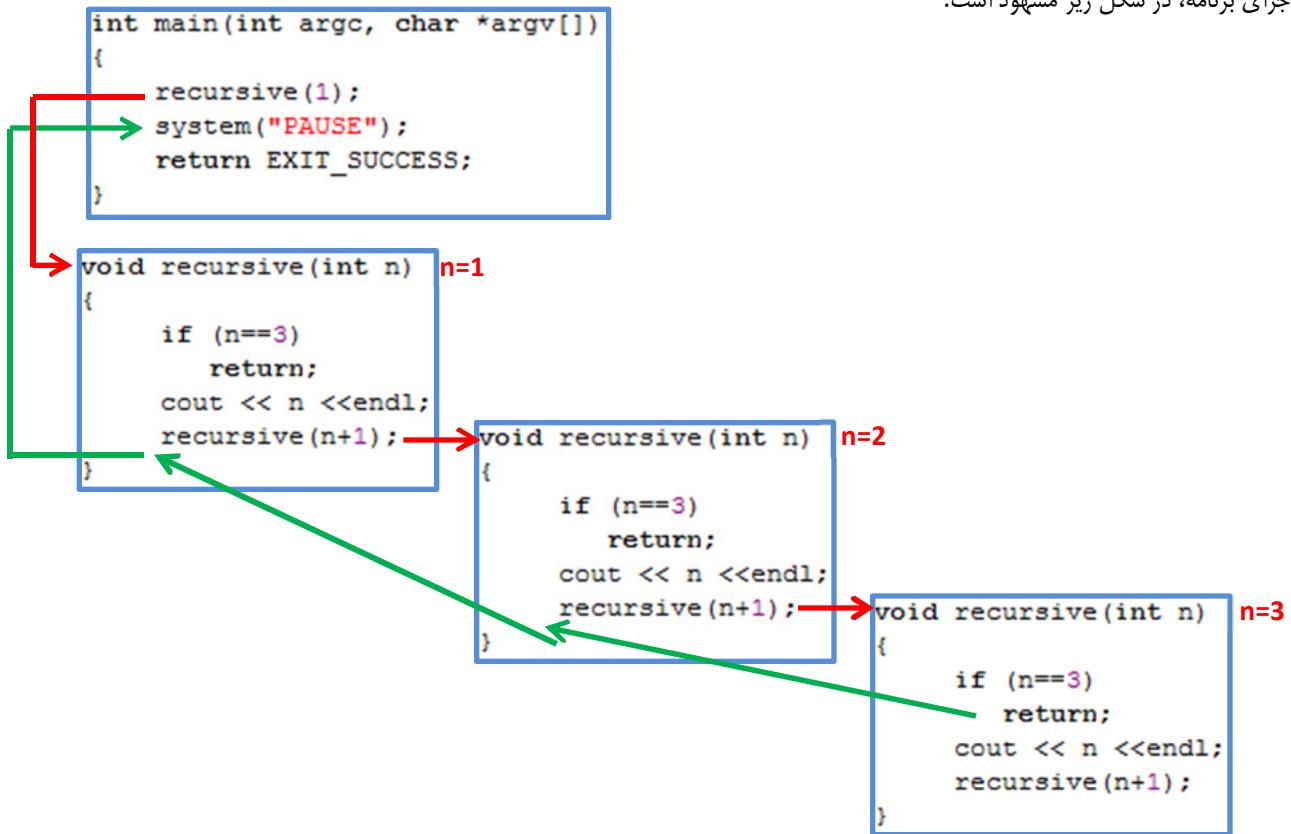
همان طور که در شکل هم مشخص است، این فرآخوانی‌ها تمام نخواهند شد و اجرای برنامه هیچ‌گاه متوقف نمی‌شود. از آن جایی که برنامه‌ای که تمام نشود، به هیچ دردی نمی‌خورد، می‌خواهیم کاری کنیم که برنامه‌مان پایان داشته باشد. مثلاً زمانی که شماره‌ی تابع به عدد ۳ برسد، برنامه تمام شود. برای انجام این کار کافیست قبل از این‌که در تابع خودش را صدا بزنیم، بررسی کنیم که آیا شماره‌ی تابع برابر ۳ شده است یا نه. اگر به عدد ۳ رسیده بود با استفاده از دستور `return` تابع را پایان می‌دهیم، وگرنه به فرآخوانی ادامه می‌دهیم:

```
main.cpp
#include <iostream.h>

void recursive(int n)
{
    if (n==3)
        return;
    cout << n << endl;
    recursive(n+1);
}

int main(int argc, char *argv[])
{
    recursive(1);
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

نحوه اجرای برنامه، در شکل زیر مشهود است:



روند اجرا به این شکل است که بعد از این که تابع `recursive` برای بار دوم خودش را صدای زند (سومین تابع)، مقدار `n` برابر ۳ است، پس شرط دستور `if` برقرار است، تابع دستور `return` را انجام می‌دهد و به این ترتیب از تابع سوم خارج می‌شود. طبیعتاً بعد از بازگشت از تابع خط بعدی تابع صدای زند (تابع دوم) باید اجرا شود. چون خط بعدی وجود ندارد، تابع دوم تمام شده و به خط بعدی در تابع اول می‌رود و به این ترتیب تابع اول هم تمام شده و به تابع `main` باز می‌گردیم و بعد دستور `system("PAUSE")` اجرا می‌گردد.

پس در تابع بازگشتی (تابعی که خود را صدای زند)، حتماً باید در شروع تابع، به نوعی شرط پایان فراخوانی را داشته باشیم تا اجرای تابع بازگشتی بتواند تمام شود.

تا این جای کار، چند برنامه بازگشتی را دیدیم که فایده‌ی خاصی برای ما نداشتند و مسائل معنی داری را برای ما حل نمی‌کردند. هدف ما از ذکر این مثال‌ها آشنایی اولیه‌ی شما با نحوه‌ی کارکرد تابع بازگشتی بود. در ادامه، مثال‌های کلاسیکی از استفاده‌ی معنادار از برنامه‌نویسی بازگشتی خواهیم دید.

(۲) مثال محاسبه‌ی فاکتوریل

حتماً با مفهوم و رابطه‌ی ریاضی فاکتوریل آشنایی دارید:

$$n! = n * (n-1) * (n-2) * \dots * 2 * 1$$

تابعی به نام `fact` را در نظر بگیرید که با گرفتن عدد ورودی `n`، مقدار `n!` را به عنوان خروجی محاسبه می‌کند. حال فرض کنید با فراخوانی این تابع مقدار `fact(n)` را محاسبه کرده باشیم و نیاز به محاسبه‌ی مقدار `fact(n+1)` داشته باشیم. این مقدار از روی رابطه‌ی بازگشتی زیر قابل محاسبه است:

$$fact(n+1) = (n+1) * fact(n)$$

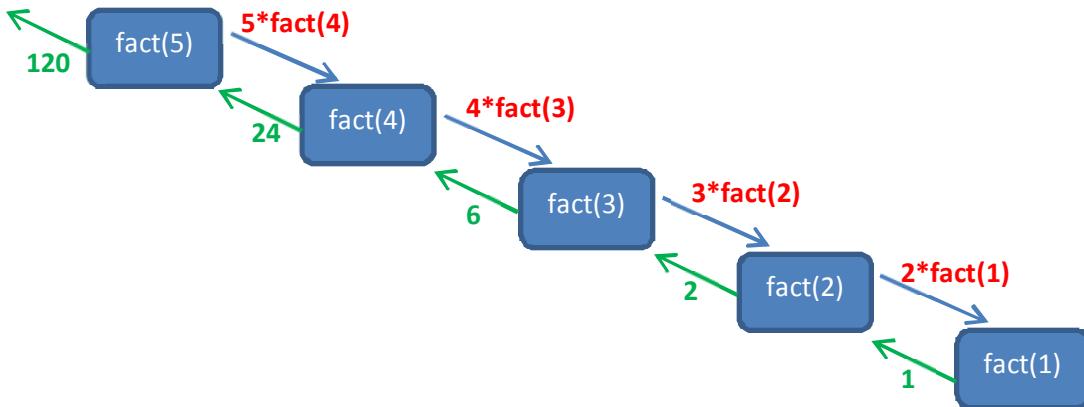
در اینجا می‌توانیم از مقدار تابع در `n` برای محاسبه‌ی مقدار تابع برای `n+1` استفاده کنیم و این یعنی یک ایده برای نوشتن یک تابع بازگشتی برای محاسبه‌ی فاکتوریل:

```
main.cpp
#include <iostream.h>

int fact(int n)
{
    if(n == 1)
        return 1;
    return n*fact(n-1);
}

int main(int argc, char *argv[])
{
    cout <<fact(5)<<endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

در ادامه باز هم برای فهم بهتر، شکلی برای اجرای برنامه می‌بینید:



برای محاسبه $fact(5)$ از رابطه $5*fact(4)$ استفاده کردایم، که این باعث صدا زدن $fact(4)$ می‌شود. برای محاسبه $fact(4)$ ، تابع $fact(3)$ فراخوانی می‌شود و ... تا این‌که برای محاسبه $fact(1)$ ، تابع $fact(1)$ را صدا می‌زنیم. در این‌جا چون n برابر ۱ است، شرط if موجود در $fact(1)$ برقرار شده و مقدار ۱ به عنوان خروجی $fact(1)$ به تابع صدا زنده $fact(2)$ داده می‌شود. خروجی تابع $fact(2)$ برابر $2*fact(1)$ است که با داشتن مقدار خروجی $fact(1)$ ، خروجی $fact(2)$ می‌شود. این روند ادامه پیدا می‌کند و خروجی توابع $fact(3)$ و $fact(4)$ و $fact(5)$ به ترتیب برابر ۶ و ۲۴ و ۱۲۰ خواهد بود. چون در تابع $main$ مقدار خروجی $fact(5)$ چاپ شده است، مطابق انتظار عدد ۱۲۰ روی کنسول خروجی دیده می‌شود:

120
Press any key to continue . . .

لطفاً به نکته‌ی فنی مهم زیر خوب دقت کنید:

نکته‌ی فنی

اگر کمی در مورد برنامه‌ی بالا و به طور کلی به مفهوم برنامه‌نویسی بازگشته تفکر کنیم، متوجه می‌شویم که در همه‌ی مسائلی که به روش بازگشته حل می‌شوند، پارامتری وجود دارد که اندازه‌ی مسئله را مشخص می‌کند. به عنوان مثال در مسئله‌ی فاکتوریل n اندازه‌ی مسئله را نشان می‌دهد. ماهیت این پارامتر طوری است که هر چه اندازه‌ی مسئله کوچک‌تر باشد، حل مسئله ساده‌تر و کم‌هزینه‌تر می‌باشد. استراتری ما برای حل این گونه مسائل، بدست آوردن حل مسئله‌ی با اندازه‌ی بزرگ‌تر، با مفروض داشتن حل مسئله‌ی با اندازه‌ی کوچک‌تر است. بنابراین هر برنامه‌ی بازگشته از دو قسمت اصلی تشکیل شده است:

- (۱) شرط پایان تابع (حالت بدیهی)
- معمولًاً در ابتدای تابع می‌آید و معمولاً مقدار تابع را در یک حالت بدیهی (مسئله در اندازه‌ی بسیار کوچک) مشخص می‌کند. مثل برنامه‌ی بالا که در آن، حالت بدیهی مقدار خروجی ۱ برای $fact(1)$ است.

(۲) رابطه‌ی بازگشته

نحوه‌ی بدست آوردن حل مسئله برای مقدار (اندازه‌ی) بزرگ‌تر، با فرض داشتن حل مسئله برای یک مقدار (اندازه‌ی) کوچک‌تر. به عنوان مثال با داشتن رابطه‌ی $fact(n) = n * fact(n-1)$ ، ما حل مسئله برای مقدار بزرگ‌تر n را از روی حل مسئله

برای مقدار کوچک‌تر $n-1$ بدست آورده‌ایم:

```
int fact (int n)
{
    if (n == 1)
        return 1;
    return n*fact(n-1); }
```

حالت بدیهی \rightarrow
رابطه بازگشتی \rightarrow

پس برای نوشتن یک تابع بازگشتی، کافیست که رابطه‌ی بازگشتی و حالت بدیهی را مشخص کنیم و بقیه‌ی کار، که تبدیل کردن این دو قسمت به کد زبان C است، مشکل خاصی ندارد.

خطاهای معمول برنامه نویسی

نیاوردن شرط پایان در توابع بازگشتی یا آوردن آن پس از رابطه‌ی بازگشتی، باعث می‌شود که تابع بازگشتی هیچ‌گاه پایان نیافته و برنامه دچار یک خطای زمان اجرا شود.

۳) مثال محاسبه‌ی عدد n ام فیبوناچی

سری اعداد فیبوناچی را که به خاطر دارید؟

n	1	2	3	4	5	6	7	8	..
$fib(n)$	1	1	2	3	5	8	13	21	..

از دنباله‌ی اعداد فیبوناچی می‌توان دید که هر عدد فیبوناچی، برابر مجموع دو عدد قبلی خود است و این یعنی بدست آمدن یک رابطه‌ی بازگشتی! پس ما می‌توانیم به راحتی یک برنامه‌ی بازگشتی برای محاسبه‌ی عدد n ام فیبوناچی بنویسیم.

در ادامه دو مرحله‌ی اساسی نوشتن این تابع بازگشتی را می‌بینیم:

۱. حالت بدیهی: $fib(1) = 1$ و $fib(2) = 1$

۲. رابطه بازگشتی: $fib(n) = fib(n-1) + fib(n-2)$

لطفاً توجه کنید که این‌بار به جای مقدار تابع برای یک حالت بدیهی، مقدار تابع برای دو حالت بدیهی را آورده‌ایم. دلیل آن، این است که این‌بار برخلاف رابطه‌ی بازگشتی برای فاکتوریل که تنها بستگی به مقدار فاکتوریل در یک مقدار کوچک‌تر $n-1$ داشت، رابطه‌ی بازگشتی ما در این حالت به مقدار فیبوناچی برای دو مقدار کوچک‌تر $n-1$ و $n-2$ دارد.

برنامه‌ی زیر شامل یک تابع بازگشتی برای محاسبه‌ی عدد n ام فیبوناچی است:

```
main.cpp | 
#include <iostream.h>

int fib(int n)
{
    if (n<3)
        return 1;
    return fib(n-1)+fib(n-2);
}

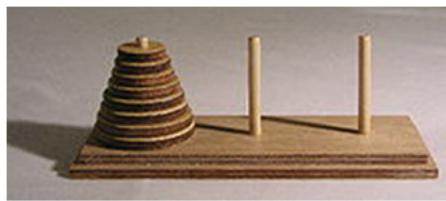
int main(int argc, char *argv[])
{
    cout <<fib(8)<endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

حالت بدهی
رابطه بازگشتی

خروجی برنامه:

```
21
Press any key to continue . . .
```

۴۶) مثال برج‌های هانوی



در افسانه‌ها آمده است که در محوطه‌ی معبدی در آسیای دور، سه میله‌ی الماسی قرار داشته و میله‌ی اول حاوی ۶۴ دیسک از جنس طلا بوده که از بزرگ به کوچک روی هم قرار گرفته بودند. کاهنان معبد در تلاش بودند تا دیسک‌ها را از میله‌ی اول به یکی دیگر از میله‌ها، مثلاً میله‌ی سوم انتقال دهند. البته هر انتقالی باید سه قانون زیر را رعایت می‌کرده:

- در هر مرحله تنها یک دیسک انتقال داده شود.
 - دیسک انتخابی باید بالای یکی از میله‌ها قرار داشته باشد.
 - دیسک منتقل شده، نباید روی دیسکی کوچک‌تر از خودش قرار بگیرد.
- آن‌ها باور داشتند که با تمام شدن انتقال دیسک‌ها، عمر جهان نیز به پایان خواهد رسید!

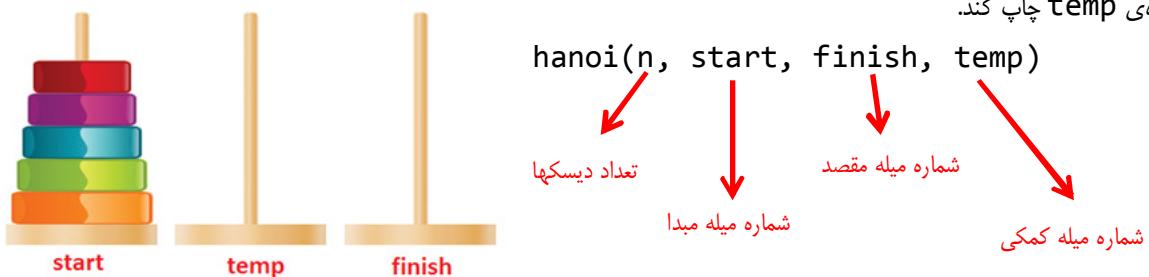
⁴⁶ Hanoi Towers

نکته‌ی تفننی

اگر این افسانه حقیقت داشته باشد و انتقال هر دیسک تنها یک ثانیه طول بکشد، حدود ۵۸۵ میلیارد سال طول خواهد کشید تا این انتقال پایان یابد و دنیا نیز همزمان تمام شود! چراکه انتقال این ۶۴ دیسک نیاز به $18,446,744,073,709,551,615$ حرکت خواهد داشت! (چرا؟)

ما در اینجا قصد داریم، برنامه‌ای بنویسیم که با گرفتن تعداد دیسک‌ها، شماره‌ی میله‌ی مبدأ و شماره‌ی میله‌ی مقصد، تمام حرکت‌های لازم برای رساندن دیسک‌ها به میله‌ی مقصد را مشخص کنیم، تا کمک اندکی برای پایان هرچه زودتر عمر دنیا کرده باشیم!

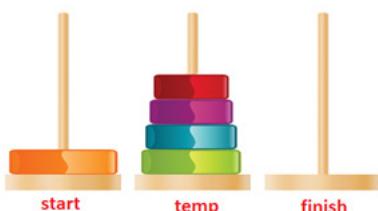
فرض کنید تابع زیر، قرار است حرکات لازم را برای انتقال n دیسک از میله با شماره‌ی **start** به میله با شماره‌ی **finish**، با کمک میله‌ی دیگر با شماره‌ی **temp** چاپ کند.



طبق معمول حل مسائل بازگشته، می‌توانیم فرض کنیم که حل مسئله را برای اعداد کوچک‌تر (تعداد دیسک‌های کمتر) بلد هستیم. (مثال ۱-۱ دیسک) با این فرض قصد داریم حل مسئله برای n دیسک را تبدیل به مسئله‌هایی با $n-1$ دیسک کنیم.

این کار را در سه مرحله انجام می‌دهیم:

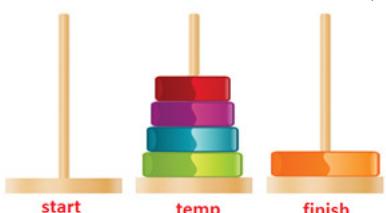
- با انجام تمام حرکات لازم، $n-1$ دیسک را از میله‌ی **start** به میله‌ی **temp** ببریم. در پایان این مرحله، وضعیت میله‌ها و دیسک‌ها به این شکل در خواهد آمد:



در این مرحله میله‌ی مبدأ، میله‌ی **start** و میله‌ی مقصد، میله‌ی **temp** است و از میله‌ی **finish** به عنوان میله‌ی کمکی استفاده می‌کنیم. پس با فرض بلد بودن حل مسئله برای $n-1$ دیسک، این مرحله را می‌توان با استفاده از فراخوانی **hanoi (n-1, start, temp, finish);**

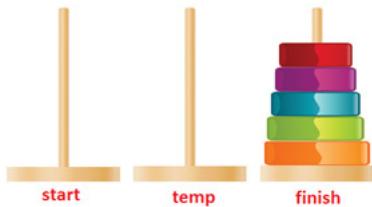
انجام داد.

- بزرگ‌ترین دیسک را که در میله‌ی **start** باقی مانده است، به میله‌ی **finish** ببریم.



انجام این کار تنها شامل یک حرکت می‌باشد. کافیست این حرکت را چاپ کنیم:

```
cout << start << " -> " << finish << endl;
```



(۳) در نهایت با انجام تمام حرکات لازم، $n-1$ دیسک باقی‌مانده روی میله‌ی **temp** را به میله‌ی **finish** می‌بریم تا وضعیت میله‌ها و دیسک‌ها به این شکل شود:

این مرحله هم معادل فراخوانی زیر از تابع **hanoi** می‌باشد:

```
hanoi(n-1, temp, finish, start);
```

۳ مرحله‌ی بالا نشان داد که چگونه حل مسئله برای n دیسک را تبدیل به حل مسئله با $n-1$ دیسک کنیم. چیزی که در بالا توضیح دادیم در واقع پیدا کردن رابطه‌ی بازگشتی برای حل مسئله بود.

اما برای تکمیل برنامه‌نویسی بازگشتی، یک مرحله‌ی دیگر باقی مانده است، حل مسئله برای حالت بدیهی: در شرایطی که تنها یک دیسک داشته باشیم ($n==1$) کافی است که تنها دیسک‌مان را از میله‌ی **start** به میله‌ی **finish** ببریم. یعنی تنها یک حرکت. کدی که در ادامه خواهد آمد، در واقع یک جمع‌بندی است بر آن‌چه تا کنون در مورد این مسئله گفته‌یم:

```
main.cpp |
```

```
#include <iostream.h>

void hanoi (int n, int start, int finish, int temp)
{
    if (n==1)
    {
        cout << start << " -> " << finish << endl;
    }
    else
    {
        hanoi(n-1, start, temp, finish);
        cout << start << " -> " << finish << endl;
        hanoi(n-1, temp, finish, start);
    }
}

int main(int argc, char *argv[])
{
    hanoi(3, 1, 3, 2);
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

حالات بدیهی

حل بازگشتی

خروجی برنامه در ادامه دیده می‌شود که حرکات لازم را به ترتیب نشان می‌دهد:

```

1 -> 3
1 -> 2
3 -> 2
1 -> 3
2 -> 1
2 -> 3
1 -> 3
Press any key to continue . . .

```

همان طور که دیدید، لزومی ندارد از توابع بازگشتی صرفاً برای محاسبات ریاضی مثل اعداد فیبوناچی و فاکتوریل استفاده کنیم، در ادامه یک مسئله‌ی جالب دیگر را هم خواهید دید.

(۵) مثال جستجوی دودویی

فرض کنید a آرایه‌ای مرتب از اعداد به طول n باشد، می‌خواهیم در این آرایه عددی را جستجو کنیم. مثلاً عدد X به ما داده شده است و ما می‌خواهیم اندیس خانه‌ای از آرایه، که X در آن قرار دارد را پیدا کنیم.

یک روش خیلی ساده و سر راست، که خیلی زود هم به ذهن می‌رسد، این است که با یک حلقه‌ی fOR تمام خانه‌های آرایه را بررسی کنیم، اما این روش ساده، به اندازه‌ی کافی سریع نیست. احتمالاً مشکل از این جا ناشی می‌شود که ما هیچ استفاده‌ای از فرض مرتب بودن آرایه نکردایم. شاید با استفاده از این فرض بتوانیم این جستجو را سریع‌تر انجام دهیم. اما چطور؟ با استفاده از روش **جستجوی دودویی**^{۴۷}.

فرض کنید آرایه‌ی مرتب a شامل عناصر زیر باشد:

1	5	8	10	11	16	19
---	---	---	----	----	----	----

عنصر میانی آرایه را در نظر بگیرید. به علت مرتب بودن آرایه به صورت صعودی، تمام عناصر سمت چپ از عنصر وسط کوچک‌تر هستند و عناصر سمت راست همگی از عنصر وسط آرایه بزرگ‌ترند.

1	5	8	10	11	16	19
---	---	---	----	----	----	----

حالا بباید X را با عنصر وسط آرایه مقایسه کنیم:

- اگر X با این عنصر برابر بود که جواب مسئله پیدا شده است.
- اگر X از عنصر وسط آرایه بزرگ‌تر باشد، در این صورت حتماً از تمام عناصر سمت چپ هم بزرگ‌تر است. بنابراین عنصر X قطعاً سمت چپ آرایه نخواهد بود و باید در سمت راست آرایه دنبال آن بگردیم.
- اگر X از عنصر میانی آرایه کوچک‌تر باشد، حتماً از همه‌ی عناصر سمت راست (که همگی از عنصر میانی بزرگ‌ترند) هم کوچک‌تر است پس باید در سمت چپ X را جستجو کنیم.

به این ترتیب با یک مقایسه یا به جواب می‌رسیم یا اندازه‌ی آرایه‌ای که در آن جستجو می‌کنیم، نصف می‌شود. کافی است همین رویه را روی نیمه‌ی آرایه‌ی بدست آمده تکرار کنیم. اگر هر مرحله از این رویه (مقایسه‌ی X با عنصر وسط) را در یک تابع قرار داده و انجام دهیم، به این دلیل که همین رویه دوباره باید روی آرایه‌ی نصف شده انجام شود، این تابع می‌تواند برای انجام مرحله‌ی بعدی خودش را فراخوانی کند و به این ترتیب ما یک تابع بازگشتی خواهیم داشت.

⁴⁷ Binary Search

در هر مرحله از اجرا، تابع باید بداند محدودهای که این بار باید جستجو را در آن جا انجام دهد، کجاست. پس تابع بازگشتی به عنوان پارامتر باید عدد مورد جستجو، آرایه و اندیس شروع و پایان ناحیه‌ی جستجو را بگیرد:

```
int binarySearch(int x, int[] a, int start, int end)
```

دقیق کنید که برای این تابع یک خروجی از نوع `int` در نظر گرفته شده است. در صورت پیدا شدن `X`، اندیس آن در آرایه برگردانده می‌شود و در غیر این صورت عدد `1` برای مشخص کردن این که `X` در آرایه وجود نداشته بازگردانده خواهد شد.

شرط پایان این تابع بازگشتی حالتی است که آرایه به طول صفر برسد. (start>end) در این حالت آرایه حاوی عنصر X نبوده و جواب - 1 به معنی پیدا نشدن عدد X در آرایه، برگردانه می‌شود.

کد این: ب‌نامه‌د، ادامه‌آمده است. خوب به آن توجه کنید:

```
main.cpp | #include <iostream.h>

int binarySearch(int x, int a[], int start, int end)
{
    if (start>end)
        return -1;

    int m = (start + end)/2;
    if (a[m]==x)
        return m;
    if (a[m]>x)
        return binarySearch(x, a, start, m-1);
    else
        return binarySearch(x, a, m+1, end);
}

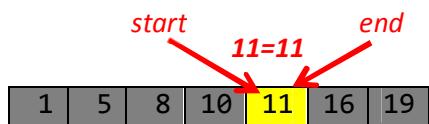
int main(int argc, char *argv[])
{
    int n = 8;
    int x = 11;
    int a[10] = {1, 5, 8, 10, 11, 16, 19};
    int index = binarySearch(x, a, 0, n-1);
    if (index == -1)
        cout << "x dar araye nist!"<<endl;
    else
        cout << "x dar andise "<< index << " ast!"<<endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

شرط بیان

حل بازگشته

د، ادامه، نمود مصور، اجرایی، برنامه، ای براي، $x=11$ می بینند:

A horizontal list of numbers: 1, 5, 8, 10, 11, 16, 19. The number 10 is highlighted with a yellow background. Above the list, the word "start" is in red with a red arrow pointing to the first element (1). The word "end" is in red with a red arrow pointing to the last element (19). Between the start and end indices, the text "10<11" is written in red.



خروجی برنامه به این شکل است:

```
x dar andise 4 ast!
Press any key to continue . . .
```

نکته‌ی فنی

جستجوی دودویی با روشی که معرفی شد، بسیار سریع‌تر از روش جستجوی عادی به جواب می‌رسد. در آرایه‌ای با یک میلیون عنصر، با استفاده از روش عادی ممکن است نیاز به انجام یک میلیون مقایسه داشته باشیم. (اگر در جستجوی بزرگ‌ترین عدد باشیم یا دنبال عددی باشیم که در آرایه وجود ندارد.) اما با روش جستجوی دودویی در این آرایه، به دنبال هیچ عددی، هرگز بیش از ۲۰ مقایسه نخواهیم داشت. (چرا؟) این یعنی پنجاه هزار برابر سریع‌تر در این مثال. با بزرگ‌تر شدن اندازه‌ی آرایه، این اختلاف وحشتناک‌تر هم خواهد شد!

بررسی مثال بالا و مثال‌های فراوان موجود دیگر، اهمیت فوق‌العاده‌ی خلاقیت در طراحی الگوریتم‌های بهتر و سریع‌تر را نشان می‌دهد.

تمرین



۱. به کمک یک تابع بازگشتی، شمارش معکوس از عدد ورودی n ، تا صفر را روی خروجی چاپ کنید.
۲. برنامه‌ای بنویسید که با تقسیمات متوالی بر ۲ و بدون استفاده از آرایه، نمایش عدد ورودی را در مبنای ۲ چاپ کند.
۳. یک تابع بازگشتی بنویسید که عملیات به توان رساندن را انجام دهد. این تابع دو پارامتر می‌گیرد و پارامتر اول را که از نوع عدد اعشاری است، به توان پارامتر دوم که عددی صحیح (مثبت یا منفی) می‌باشد، می‌رساند.
۴. یک تابع بازگشتی به نام gcd برای محاسبه‌ی ب.م.م دو پارامتر ورودی بنویسید. اگر خاطرтан باشد، $\text{gcd}(x, 0) = \text{gcd}(y, x \% y) = \text{gcd}(y, 0)$ که در آن $\%$ عملگر باقیمانده‌گیری است.
۵. تابع بازگشتی زیر چه کاری انجام می‌دهد؟

```
Int f(int x, int y)
{
    if (y==1)
        return x;
    return x + f(x, y-1);
}
```

- پس از این که متوجه شدید این برنامه چه کاری انجام می‌دهد، آن را طوری تغییر دهید که بتواند برای عواید منفی هم کار کند.
۶. مجموعه‌ی A از تمامی اعداد ۱ تا n مفروض است. برنامه‌ای بنویسید که تمام زیر مجموعه‌های این مجموعه را نمایش دهد. تعداد این زیر مجموعه‌ها چندتاست؟
 ۷. برای مجموعه‌ی A از سوال بالا، تمام جایگشت‌های (ترتیبها) ممکن را نمایش دهید. به عنوان مثال برای مجموعه‌ی $\{1, 2, 3\}$ دو ترتیب $1, 2, 3$ و $3, 2, 1$ وجود دارند. تعداد این ترتیبها چندتاست؟
 ۸. مرتب‌سازی ادغامی^{۴۸}، به صورت بازگشتی، آرایه‌ای از اعداد را مرتب می‌کند. تحقیق کنید که از چه الگوریتمی برای این کار استفاده می‌کند. پس از تحقیق در مورد این الگوریتم، آن را پیاده‌سازی کنید.
 ۹. بازی تاس یک بازی تک نفره است که در هر مرحله بازیکن تاس را می‌اندازد. اگر حاصل ۱ یا ۲ بود بازیکن بازی را باخته و اگر برابر مقدار تاس در مرحله‌ی قبل باشد، بازی را برده است. اگر هیچ یک از این دو اتفاق رخ نداد، تاس یک بار دیگر انداخته می‌شود. یک تابع بازگشتی برای انجام این بازی بنویسید که خروجی آن یک متغیر بولین است که اگر true باشد به معنی برد بازیکن می‌باشد.
 ۱۰. تابعی بازگشتی بنویسید که با گرفتن آرایه‌ای از اعداد صحیح و طول آن به عنوان پارامتر، در صورت امکان طوری $+ - + - \dots$ را بین آنها قرار دهد که حاصل آن صفر شود. به عنوان مثال آرایه‌ی $6, 5, 3, 2, 2$ داریم:

$2 - 5 - 3 + 6 = 0$ دقت کنید که برنامه‌ی شما باید تمام حالت‌های ممکن را چاپ کند.

فصل دوازدهم : ساختمانها

۱. ضرورت استفاده از ساختمان

۲. معرفی ساختمان

۳. استفاده از ساختمان برای محیط گرافیکی

۴. مثلث سرپینسکی

۵. مزایای استفاده از ساختمان

۱) ضرورت استفاده از ساختار

فرض کنید از شما خواسته شده است نرم افزاری برای مدیریت اطلاعات دبیرستان تولید کنید. طبیعتاً مهم‌ترین اطلاعات موجود مدرسه برای نگهداری و پردازش، اطلاعات مربوط به دانش‌آموزان می‌باشد. از میان اطلاعات متفاوتی که یک دانش‌آموز می‌تواند داشته باشد، موارد زیر برای ما اهمیت دارند:

- نام و نام خانوادگی
- شماره‌ی دانش‌آموزی
- شماره‌ی کلاس
- معدل

با فرض این‌که این مدرسه ۲۰۰ دانش‌آموز دارد، ما به دنبال راهی برای نگهداری اطلاعات دانش‌آموزان به صورت منظم هستیم، به طوری که بعداً بتوانیم هر گونه سوالی در مورد اطلاعات دانش‌آموزان را پاسخ دهیم. مثلاً بتوانیم با گرفتن نام و نام خانوادگی یک دانش‌آموز، دیگر اطلاعات مربوط به این دانش‌آموز را چاپ کنیم. چطور می‌توانیم این کار را انجام دهیم؟

ساده‌ترین راهی که به ذهن‌مان می‌رسد این است که برای هر کدام از این اطلاعات (نام، نام خانوادگی، شمار دانش‌آموزی و ...) یک آرایه‌ی ۲۰۰ عنصری در نظر بگیریم، یعنی در مجموع ۴ آرایه به طول ۲۰۰ داشته باشیم:

```
char name[200][50];
int id[200];
int classNumber[200];
float average[200];
```

با داشتن این آرایه‌ها می‌توانیم اطلاعات مربوط به دانش‌آموزان را به طور منظم ذخیره کنیم. به این صورت که نام دانش‌آموز اول در خانه‌ی اول آرایه‌ی `name`، شماره‌ی دانش‌آموزی او در خانه‌ی اول آرایه‌ی `id`، شماره‌ی کلاس او در خانه‌ی اول آرایه‌ی `classNumber` و ... قرار داده شود

name	“Ali Alavi”	“Reza Razavi”	...	“Amir Amiri”
id	1055	1023	...	1057
classNumber	101	102	...	101
average	19.21	18.88	...	19.05

اطلاعات مربوط به دانش‌آموز اول

اطلاعات مربوط به دانش‌آموز دوم

اطلاعات مربوط به دانش‌آموز ۲۰۰م

حالا فرض کنید قوار است یک دانش آموز (مثالاً دانش آموز اول) از این مدرسه، به مدرسه‌ی دیگری منتقل شود. طبیعی است که باید اطلاعات مربوط به این دانش آموز پاک شود. اما نکته‌ای که در اینجا مهم است، این است که در همه‌ی آرایه‌ها باید از عنصر بعدی دانش آموز حذفی، همه‌ی خانه‌های آرایه‌ها را یکی به سمت چپ شیفت دهیم. حالا اگر یادمان برود که مثلاً آرایه‌ی `classNumber` را شیفت دهیم، چه اتفاقی رخ می‌دهد؟ همان‌طور که در شکل پایین ملاحظه می‌کنید، اشتباهاتی در اطلاعات مربوط به شماره‌ی کلاس دانش آموزان ایجاد می‌شود، مثلاً رضا رضوی به اشتباه، دانش آموز کلاس ۱۰۱ شناخته می‌شود!

name	“Reza Razavi”	...	“Amin Amiri”
id	1023	...	1057
classNumber	101	102	...
average	19.21	...	19.05

اطلاعات مربوط به دانش آموز اول اطلاعات مربوط به دانش آموز ۱۹۹ ام

در مثال بالا هر دانش آموز، تنها ۴ مشخصه داشت، اما برای نگه‌داری اطلاعات مربوط به دنیای واقعی، این تعداد مشخصه‌ها بیشتر خواهد بود. مثلاً برای یک دانش آموز، شما باید شماره‌ی شناسنامه، شماره‌ی تلفن منزل، آدرس منزل، نام پدر، تاریخ تولد و ... را نیز نگه‌داری کنید. در این حالت، تعداد آرایه‌های مورد نیاز شما، سر به فلک خواهد گذاشت و شما به عنوان یک انسان جایز‌الخطا، کاملاً حق خواهید داشت که شیفت دادن یکی از این آرایه‌ها را فراموش کنید!

بنابراین، در چنین حالتی (نگه داشتن چندین آرایه برای نگه‌داری اطلاعات مربوط به موجودانی با چندین مشخصه) احتمال بروز خطأ در برنامه، بسیار بالا خواهد بود. سوالی که احتمالاً تا اینجا برای شما مطرح شده است، این است که چطور می‌توانیم این مشکل را حل کنیم؟

برای حل این مشکل، در ابتدا باید به منشا بروز آن فکر کنیم. همان‌طور که در شکل بالا دیدید، هر ستون از آرایه‌ها، نمایانگر اطلاعات مربوط به یکی از دانش آموزان است. مشکل این‌جاست که به علت جدا بودن اطلاعات مختلف مربوط به هر دانش آموز، (وجود هر کدام از ویژگی‌ها در یک آرایه‌ی جدا) امکان جدایی این اطلاعات و در نتیجه به هم ریختن نظم اطلاعات وجود دارد. به مستطیل‌های فرضی، که دور هر ستون از اطلاعات در شکل‌های بالا وجود دارند، توجه کنید. این مستطیل‌های فرضی اطلاعات مربوط به هر دانش آموز را به هم می‌چسبانند. اگر ما در برنامه‌مان به نوعی می‌توانستیم چیزی شبیه این مستطیل‌ها داشته باشیم، که اطلاعات مربوط به یک موجود را به هم بچسبانند و اجازه‌ی جدا شدن را به آن‌ها را ندهنده، مشکل‌مان برطرف می‌شد. در واقع چیزی که ما به دنبال آن هستیم، یک متغیر مرکب است که مانند این مستطیل‌ها از چندین متغیر ساده، تشکیل شده باشد.

۲) معرفی ساختار

خوبی‌خوانه زبان C، برای این مشکل ما هم فکری کرده است: **ساختار**^{۴۹}. به این وسیله، ما می‌توانیم یک متغیر مرکب تشکیل دهیم. فرمت کلی آن در زبان C به شکل زیر است:

```
struct      نام ساختار {  
    لیستی از نوع و نام متغیرها  
};
```

خطاهای معمول برنامه نویسی



فراموش کردن علامت ; در انتهای تعریف ساختار (بعد از علامت آکولاد بسته)، موجب ایجاد خطای نحوی در زمان کامپایل خواهد شد.

برای فهم بهتر شما، برای مثال مربوط به اطلاعات دانش‌آموزان، یک ساختار تعریف می‌کنیم:

```
struct Student{  
    char name[20];  
    int id;  
    int classNumber;  
    float average;  
};
```

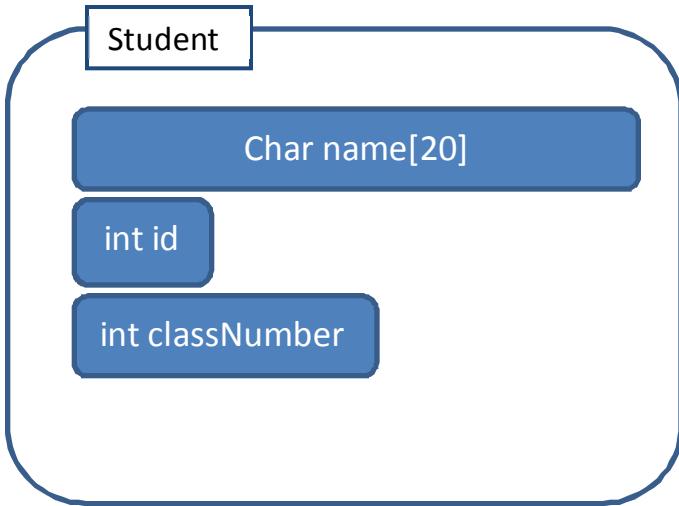
در اینجا یک نوع متغیر جدید تعریف کرده‌ایم، به نام **Student**. دقت کنید که بعد از این تعریف، **int** یک نوع جدید خواهد بود مثل **char** یا **float** که نوع متغیرهای از قبل تعریف شده هستند. پس همان‌طور که قبلاً می‌توانستید متغیری از نوع **Student** تعریف کنید، حالا باید بتوانید متغیری از نوع **Student** تعریف کنید:

```
int a;  
Student s;
```

در دو خط بالا به همان شکل که متغیری به نام **a** از نوع **int** تعریف کرده‌اید، متغیر دیگری به نام **s** از نوع **Student** معرفی کرده‌اید. اما تفاوت‌هایی هم بین این دو وجود دارد:

- نوع **Student** را خودتان تعریف کرده‌اید، اما **int** از قبل در زبان C تعریف شده است.
- نوع متغیر **Student** یک نوع مرکب است، به این معنی که خود شامل چندین متغیر ساده، مانند **int** و **float** و غیره می‌باشد.

⁴⁹ Structure



پس نوع متغیر مرکب **Student** در شکل بالا، خود شامل ۴ متغیر ساده است.

اخلاق برنامه‌نویسی

برای ساختارها هم مشابه متغیرها و توابع، باید نام مناسب و گویایی انتخاب کنیم، که برای خواننده‌ای که اطلاع چندانی از برنامه‌ی ما ندارد، نشان‌دهنده‌ی اطلاعات درون این ساختار باشد. به عنوان مثال **Student**، نام مناسبی می‌باشد، برای مشخص کردن این که این ساختار اطلاعات مربوط به یک دانش‌آموز را نگه‌داری خواهد کرد، اما نامی مانند **S** نمی‌تواند چنین توصیفی از محتویات داشته باشد. علاوه‌بر این به خاطر داشته باشید که نام ساختارها بهتر است با حرف بزرگ شروع شوند (برخلاف نام توابع و متغیرها) تا با متغیرها اشتباه گرفته نشوند.

همان‌طور که می‌توانیم متغیری از نوع **int** را مقدار دهی کنیم، منطقی است بتوانیم به متغیری از نوع **Student** هم مقدار دهیم:

```

Student s;
s.id = 1023;
s.classNumber = 102;
s.average = 19.22;
strcpy(s.name, "Reza Razavi");
  
```

همان‌طور که در قطعه کد بالا دیده می‌شود، ابتدا متغیری به نام **s** از نوع مرکب **Student** تعریف شده است. برای دسترسی به هر کدام از متغیرهای این متغیر مرکب، کافیست بعد از نام متغیر (در اینجا **s** نقطه (.) قرار داده و بعد نام متغیر درونی، (مثل **id**) آورده شود. (مثلاً **s.id**) به همان ترتیبی که سابقاً می‌توانستیم آرایه‌ای از یک نوع مانند **int** سازیم، قادر خواهیم بود که آرایه‌ای از نوع **Student** داشته باشیم.

```

int aa[200];
Student students[200];
  
```

در خط دوم قطعه کد بالا، آرایه‌ای به نام `Student` از نوع `students` تعریف شده است. قطعه کدی که در ادامه آمده است، اطلاعات مربوط به دانش‌آموزان را از ورودی خوانده و در آرایه‌ای از `Student`‌ها می‌ریزد.

```
main.cpp |  
#include <iostream.h>  
  
struct Student{  
    char name[20];  
    int id;  
    int classNumber;  
    float average;  
};  
  
int main(int argc, char *argv[]){  
    Student students[200];  
    for (int i=0;i<200;i++)  
    {  
        cin>>students[i].name;  
        cin>>students[i].id;  
        cin>>students[i].classNumber;  
        cin>>students[i].average;  
    }  
    system("PAUSE");  
    return EXIT_SUCCESS;  
}
```

۳) استفاده از ساختار برای محیط گرافیکی

می‌دانید که هر نقطه در محیط گرافیکی، شامل دو مولفه‌ی `X` و `Y` است. پس اگر قرار باشد که از یک نقطه توسط متغیرها نگه‌داری کنید، نیاز به دو متغیر خواهد داشت. اگر نیاز پیدا کنید تعداد بیشتری نقطه را نگه‌داری کنید، تعداد متغیرهای شما بیش‌تر خواهد شد. از آنجایی یک نقطه‌ی تصویر در واقع یک موجودیت است، که بهتر است توسط یک متغیر تعریف شود، عاقلانه است برای نقطه، یک متغیر مرکب (ساختار) تعريف کنیم:

```
struct Point{  
    int x;  
    int y;  
};
```

استفاده‌ای ساده‌ای از این ساختار را در برنامه‌ی زیر ملاحظه می‌فرمایید. در اینجا ابتدا ساختار نقطه تعریف شده است و سپس با تعريف متغیری به نام `p` از نوع `Point` و مقداردهی به آن، آن نقطه را رسم کرده‌ایم:

```
main.cpp
#include <iostream.h>
#include <graphics.h>

struct Point{
    int x;
    int y;
};

int main(int argc, char *argv[])
{
    initwindow(300, 300, "Draw Point");
    Point p;
    p.x = 150; p.y=150;
    putpixel(p.x, p.y, 15);
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

شاید مثال بالا به اندازه کافی نشان ندهد که تعریف ساختار برای نقطه واقعاً مفید است. برای درک بهتر، این بار فرض کنید قرار است که تابعی بنویسید که با گرفتن مختصات رئوس یک مثلث، آن را رسم کند. اگر نخواهیم از ساختار برای نقطه استفاده کنیم، باید به تابعی که وظیفه رسم مثلث را دارد، ۶ پارامتر بدهیم. چون هر مثلث سه نقطه‌ی گوش، و هر نقطه یک x و یک y دارد، به شکل زیر:

```
void drawTriangle (int x1, int y1, int x2, int y2, int x3, int y3)
```

اما با استفاده از ساختار برای نقطه، خط اول تعریف تابع به شکل زیر در می‌آید:

```
void drawTriangle(Point p1, Point p2, Point p3)
```

همان‌طور که دیده می‌شود به این شکل، نوشتن برنامه ساده‌تر و کد بسیار خواناتر خواهد شد.

هنوز هم می‌توانیم بهتر عمل کنیم! ما با فرستادن ۳ پارامتر از نوع **Point** قصد داریم یک نوع موجودیت مثلث را مشخص کنیم، اما این کار را با فرستادن ۳ پارامتر انجام داده‌ایم. این یعنی این که ما می‌توانیم باز هم با استفاده از ساختار و یکی کردن این سه متغیر در یک ساختار جدید، برنامه‌مان را باز هم ساده‌تر و خواناتر کنیم:

```
struct Triangle{
    Point p1;
    Point p2;
    Point p3;
};
```

با استفاده از این ساختار، تابع رسم مثلث به ساده‌ترین و خواناترین شکل ممکن در می‌آید:

```
void drawTriangle ( Triangle t)
```

در ادامه، برنامه‌ی رسم مثلث را با استفاده از ساختار مثلث، مشاهده می‌فرمایید:

```

main.cpp |

struct Point{
    int x;
    int y;
};

struct Triangle{
    Point p1;
    Point p2;
    Point p3;
};

void drawTriangle (Triangle t)
{
    line(t.p1.x, t.p1.y, t.p2.x, t.p2.y);
    line(t.p2.x, t.p2.y, t.p3.x, t.p3.y);
    line(t.p3.x, t.p3.y, t.p1.x, t.p1.y);
}

int main(int argc, char *argv[])
{
    initwindow(300, 300, "Draw Triangle");
    Point p1, p2, p3;
    p1.x = 150; p1.y = 50;
    p2.x = 100; p2.y = 150;
    p3.x = 200; p3.y = 150;
    Triangle t;
    t.p1=p1; t.p2=p2; t.p3=p3;
    drawTriangle(t);
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

نکته‌ی فنی



قبل‌اً گفته بودیم که هر متغیر مرکب از چندین متغیر دیگر تشکیل شده است. اما حالا می‌بینیم که لزومی ندارد که این متغیرها ساده باشند. همان‌طور که در برنامه‌ی بالا مشاهده می‌کنید، در متغیر مرکب **Triangle** از متغیر مرکب دیگری استفاده شده است (**Point**). استفاده از یک متغیر مرکب در یک متغیر مرکب دیگر، هیچ مشکلی ایجاد نمی‌کند و در بسیاری از موارد، چنین ساختاری می‌تواند بسیار هم مفید باشد.

خطاهای معمول برنامه نویسی



دقت کنید که با توجه به این که متغیر مرکب **Point** در متغیر مرکب **Triangle** استفاده شده است، تعریف **Point** باید قبل از تعریف **Triangle** بیاید. و گرنه هنگام تعریف **Triangle**، کامپایلر به یک موجود ناشناخته به نام **Point** برخورد می‌کرده، که این موضوع باعث ایجاد یک خطای زمان کامپایل می‌گردد.

۴) مثلث سرپینسکی

در این بخش، یک مثال جالب خواهید دید، که مروری بر مباحث توابع بازگشتی و ساختار خواهد بود.

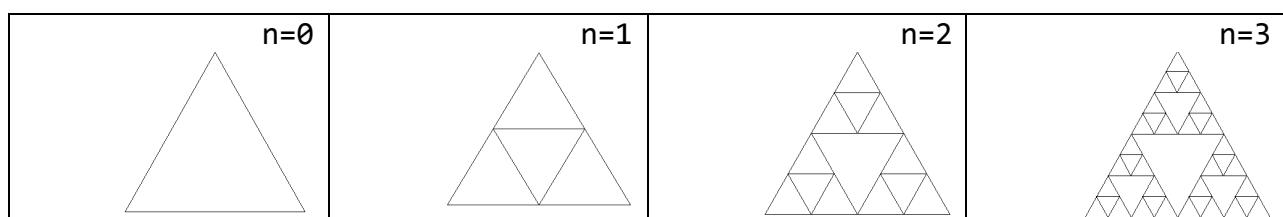
مثلث سرپینسکی^{۵۰} یکی از ساده‌ترین و جالب‌ترین فراکتال‌های خود مشابه است، که نحوه‌ی ساخت آن به شکل زیر می‌باشد:

- در ابتدا یک مثلث ساده‌ی متساوی‌الاضلاع را در نظر بگیرید.

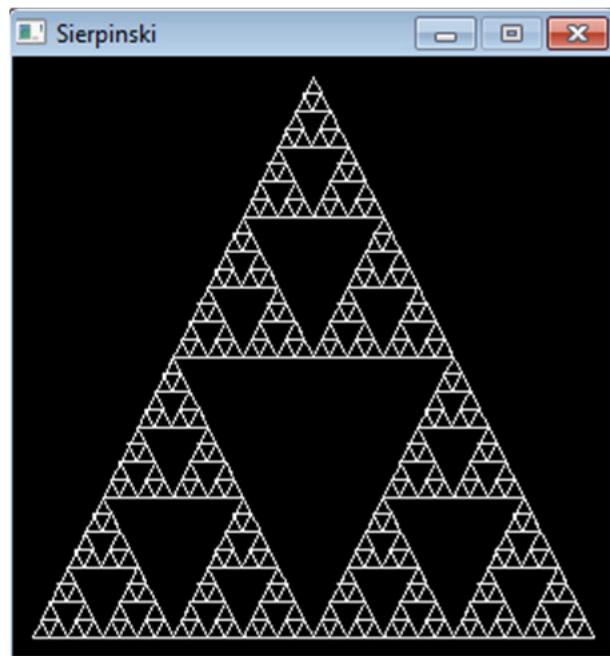
- نقاط میانی سه ضلع مثلث را به هم وصل کنید.

- همین روش را برای هر یک از سه گوش‌های تشکیل شده، انجام دهید.

شکل مثلث، بر اساس شماره‌ی مرحله در زیر آمده است:



اجرای برنامه برای $n=5$ و برنامه‌ی پیاده‌سازی شده‌ی آن به ترتیب در ادامه آمده است. توصیه می‌کنیم این برنامه را برای $n=11$ اجرا کرده و مراحل جالب کشیده شدن شکل را مشاهده کنید.



⁵⁰ Sierpinski Triangle

```

#include <iostream.h>
#include <graphics.h>

struct Point{
    int x;
    int y;
};

struct Triangle{
    Point p1;
    Point p2;
    Point p3;
};

void drawTriangle (Triangle t)
{
    line(t.p1.x, t.p1.y, t.p2.x, t.p2.y);
    line(t.p2.x, t.p2.y, t.p3.x, t.p3.y);
    line(t.p3.x, t.p3.y, t.p1.x, t.p1.y);
}

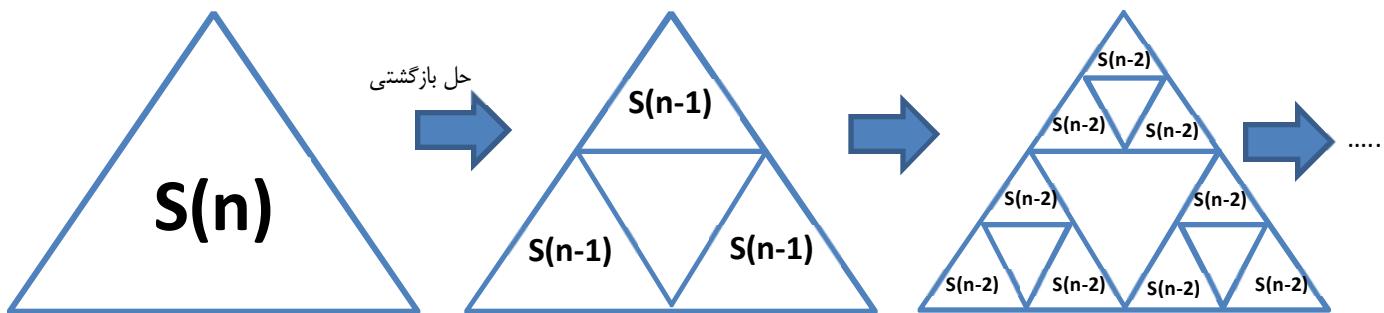
void sierpinski(Triangle t, int n)
{
    if (n==0)      } → شرط بابان
    return;
    Point m1, m2, m3;
    m1.x = (t.p1.x + t.p2.x)/2; m1.y = (t.p1.y + t.p2.y)/2;
    m2.x = (t.p2.x + t.p3.x)/2; m2.y = (t.p2.y + t.p3.y)/2;
    m3.x = (t.p1.x + t.p3.x)/2; m3.y = (t.p1.y + t.p3.y)/2;
    Triangle tt; tt.p1 = m1; tt.p2 = m2; tt.p3 = m3;
    drawTriangle(tt);
    Triangle t1, t2, t3;
    t1.p1 = t.p1; t1.p2 = m1; t1.p3 = m3;
    t2.p1 = t.p2; t2.p2 = m1; t2.p3 = m2;
    t3.p1 = t.p3; t3.p2 = m2; t3.p3 = m3;
    sierpinski(t1, n-1);
    sierpinski(t2, n-1);
    sierpinski(t3, n-1);
}

int main(int argc, char *argv[])
{
    initwindow(300, 300, "Sierpinski");
    Point p1, p2, p3;
    p1.x = 150; p1.y = 10;
    p2.x = 10;  p2.y = 290;
    p3.x = 290; p3.y = 290;
    Triangle t;
    t.p1=p1; t.p2=p2; t.p3=p3;
    drawTriangle(t);
    sierpinski(t, 5);
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

در ابتدا مثلث اصلی را با استفاده از تابع `drawTriangle` کشیده‌ایم و رسم محتویات داخل این مثلث اصلی به تابع بازگشته `sierpinski` سپرده شده است. پارامتر اول این تابع، مثلثی است که قرار است به صورت فراکتالی پر شود و پارامتر دوم عمق یا تعداد مرحله‌یست که کشیدن این مثلث‌های خود مشابه در آن ادامه خواهد یافت. دو مرحله‌ی همیشگی توابع بازگشته در اینجا هم ظاهر می‌شوند:

- شرط پایان یا حالت بدیهی: در شرایطی که تعداد مراحل رسم فراکتالی در مثلثی صفر شود، ($n==0$)، طبیعتاً نیازی نیست که بیش از این کاری انجام شود، کافیست به سادگی از تابع خارج شویم. (`return;`)
- حل بازگشته مسئله: در اینجا مسئله‌ای با اندازه‌ی بزرگتر را با استفاده از حل مسئله با اندازه‌ی کوچکتر حل می‌کردیم. مشخصاً در اینجا تعداد مراحل رسم فراکتالی (n) نشان‌دهنده‌ی اندازه‌ی مسئله می‌باشد. برای این‌که یک مثلث با n مرحله رسم فراکتالی داشته باشیم، کافی است نقاط وسط مثلث را به هم وصل کرده و در هر یک از سه مثلث گوش‌های $n-1$ مرحله رسم فراکتالی داشته باشیم. برای فهم بهتر به شکل زیر توجه کنید:



(۵) جمع‌بندی مزایای استفاده از ساختار

گرچه هر کدام از این نکات به تنهایی به صورت پراکنده در متن ذکر شده است، انجام یک جمع‌بندی در مورد مزیت‌های استفاده از ساختار در پایان کار، به فهم بهتر مطلب کمک خواهد کرد:

- احتمال کمتر بروز خط: به همان دلایلی که در قسمت اول این فصل، در مثال نگهداری اطلاعات دانش‌آموزان گفتیم، استفاده از ساختار باعث مطمئن‌تر شدن روند برنامه‌نویسی خواهد شد.
- خوانایی بیشتر: نامهای مناسب ساختارهای موجود در برنامه، دید درست را به شما در مورد کارکرد هر متغیری هنگام خواندن کد برنامه خواهد داد. به عنوان یک نمونه، مثال رسم مثلث در حالتی که ۶ متغیر به عنوان پارامتر به تابع داده می‌شد را با حالتی که یک متغیر از نوع `Triangle` به کار می‌رفت، مقایسه کنید.

استفاده‌ی مجلد: وقتی برنامه‌ی شما از ساختارهایی برای انجام هر چه بهتر وظایف خود استفاده می‌کند (مثلاً `Student` و `Triangle Point` و ...)، به راحتی می‌توانید این ساختارها را به برنامه‌های دیگر تان منتقل کرده و از آنها مجدداً استفاده کنید و به این ترتیب برنامه‌نویسی تان را سریع‌تر و کم‌خطاطر کنید.



با استناد به همه‌ی دلایل بالا، اکیداً به شما توصیه می‌کنیم، موارد مناسب استفاده از ساختار را در برنامه‌های تان شناسایی کرده و آن را به کار ببرید، تا از مزایای بکارگیری آن بهره‌مند شوید.

تمرین



- تابعی بنویسید که ضرایب یک معادله درجه دوم را به عنوان پارامتر گرفته و ریشه‌های این معادله را به **main** برگرداند، سپس در تابع **main** ریشه‌ها را چاپ کنید.
- ساختاری برای نگهداری زمان (ساعت، دقیقه و ثانیه) ایجاد کنید. سپس تابعی برای چاپ کردن آن بنویسید.
- تابعی به نام **add** بنویسید که دو زمان با ساختار تعریف شده در مثال بالا را گرفته و آن‌ها را با هم جمع کند و حاصل را به عنوان خروجی برگرداند.
- ساختاری برای نگهداری اطلاعات کارمندان بنویسید. این اطلاعات شامل نام، تعداد ساعت کارکرد، کارمزد هر ساعت و درصد مالیات مربوط به هر فرد خواهد بود. اندازه هر متغیر از نوع کارمند با ساختاری که طراحی کرده‌اید، چند بایت است؟
- تابعی بنویسید که اطلاعات مربوط به کارمندان (که در سوال بالا برای آن ساختار تعریف کردید) را در یک آرایه گرفته و میزان حقوقی که باید به هر کارمند پرداخت شود را محاسبه و چاپ کند.

فصل سیزدهم: فایل‌ها

۱. چرا به فایل نیاز داریم؟
۲. فایل چیست؟
۳. کارکردن با فایل
۴. مثال خواندن نمره از فایل
۵. مثال خواندن متن از فایل

۱) چرا به فایل نیاز داریم؟

فرض کنید می‌خواهیم برنامه‌ی دفترچه‌ی تلفن را برای کامپیوتر بنویسیم، برای نوشتن این برنامه نیاز داریم که تعدادی شماره‌ی تلفن و نام صاحب شماره را دریافت کرده و نگه‌داری کنیم. سپس در صورت نیاز کاربر به شماره‌ی یک شخص، با انتخاب نام او، شماره‌ی تلفن شخص مورد نظر به کاربر برگردانده شود. برای این کار باید اطلاعات وارد شده، در جایی ذخیره شده باشد تا در موقع نیاز از آن استفاده شود. برای ذخیره‌ی تعداد زیادی عدد و رشته استفاده از آرایه گرنیه‌ی مناسبی است، اما مشکل اینجاست که اطلاعات درون آرایه در صورت خاموش شدن کامپیوتر از دست می‌روند. چه باید کرد؟ به نظر می‌رسد نیاز داریم که اطلاعات وارد شده را در جایی ذخیره کنیم که با خاموش شدن کامپیوتر از بین نزوند. به این نوع حافظه (حافظه‌ای که اطلاعاتش با خاموش شدن کامپیوتر از دست نمی‌رود) حافظه‌ی جانبی می‌گوییم. تصویری از یک حافظه‌ی جانبی را در زیر می‌بینید:



فایل ابزاری است که امکان ذخیره اطلاعات در حافظه‌ی جانبی را برای ما فراهم می‌کند.

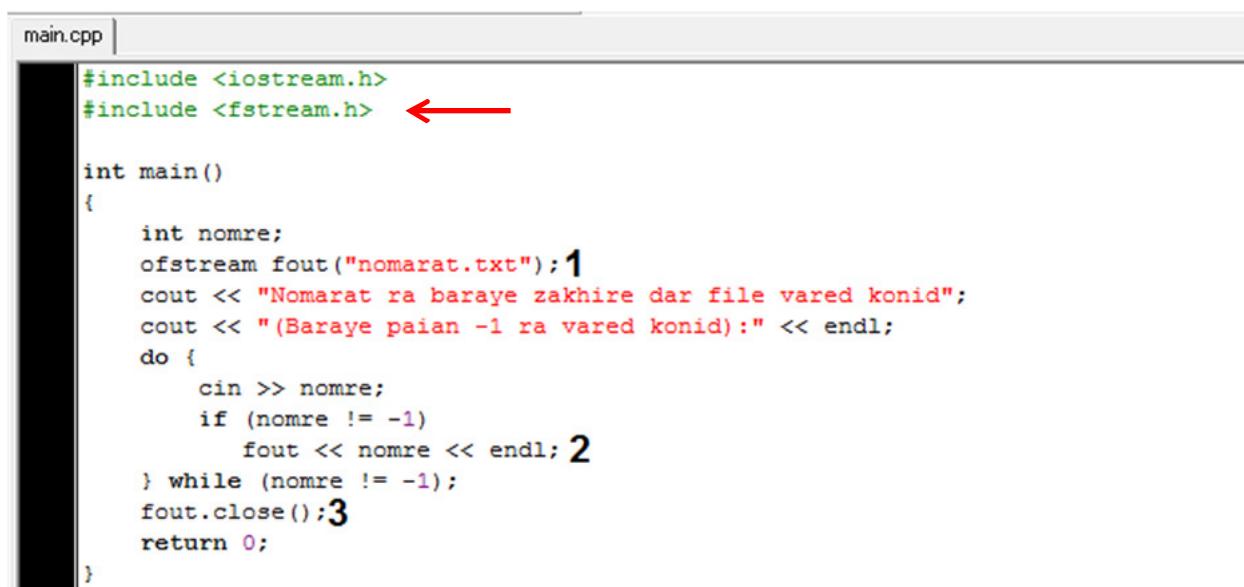
۲) فایل چیست؟

معنای لغوی کلمه‌ی فایل، پرونده است. پرونده‌ی کاغذی ابزاری است برای ذخیره اطلاعات، تا در صورت نیاز بتوانیم به آن رجوع کنیم. فایل کامپیوترا می‌توان همتای مدرن پرونده‌های کاغذی دانست که امکان ذخیره کردن اطلاعات در کامپیوتر را برای ما فراهم می‌کند.

برای ذخیره اطلاعات، باید ابتدا یک فایل ایجاد و سپس اطلاعات مورد نظر برای ذخیره شدن را در آن وارد نمود و در پایان باید فایل را بست. برای خواندن اطلاعات نیز باید ابتدا فایل را باز نمود، سپس اطلاعات را از فایل خواند و در انتهای فایل را بست. هر فایل یک نام و یک آدرس دارد و حاوی اطلاعات دلخواه کاربر است. در صورت نیاز به اطلاعات ذخیره شده، با دانستن نام فایل، اطلاعات لازم را از آن می‌خوانیم.

قبل از این‌که به ادامه‌ی کار پردازیم، تذکر این نکته ضروری است که برای سهولت در یادگیری، با فرمتی مشابه با آن‌چه در فصل ۳ برای ورودی و خروجی دیدید، با فایل‌ها کار خواهیم کرد، که هر دوی این‌ها مربوط به C++ هستند.

مثال ساده زیر را در نظر بگیرید که در آن می‌خواهیم تعدادی نمره را در یک فایل ذخیره کنیم:



```
main.cpp
#include <iostream.h>
#include <fstream.h> ←

int main()
{
    int nomre;
    ofstream fout("nomarat.txt"); 1
    cout << "Nomarat ra baraye zakhire dar file vared konid";
    cout << "(Baraye paian -1 ra vared konid):" << endl;
    do {
        cin >> nomre;
        if (nomre != -1)
            fout << nomre << endl; 2
    } while (nomre != -1);
    fout.close(); 3
    return 0;
}
```

همان‌طور که در بالا توضیح داده شد، برای نوشتن در فایل ابتدا باید فایل مورد نظر را بازکرد (۱)، سپس در آن نوشت (۲) و در پایان کار فایل را بست (۳).

در این مثال آن‌طور که مشاهده می‌کنید، یک **متغیر فایل** به نام **fout** برای کار کردن با فایل مورد نظر تعریف شده و در همان‌جا فایل مربوطه ("nomarat.txt") نیز به آن نسبت داده شده است. توضیح کامل‌تر درباره‌ی متغیر فایل در بخش بعدی می‌آید. (۱)

سپس در حلقه‌ای نمرات دریافت شده و در فایل نوشته می‌شوند. (۲)

و در انتهای با ورود ۱- توسط کاربر برنامه، حلقه پایان می‌یابد و با دستور **close** فایل سته می‌شود (۳)

نکته‌ی فنی

برای کار کردن با فایل و استفاده از دستورات کتابخانه‌ای آن، باید کتابخانه‌ی **fstream** را در ابتدای برنامه **include** کرد.

در مثال بالا با دو دستور جدید (برای باز کردن و بستن فایل) آشنا شدیم، در ادامه توضیح بیشتری در مورد آنها می‌دهیم.

۳) کار کردن با فایل

در زبان C++ نحوه‌ی باز کردن فایل با توجه به استفاده‌ای که از فایل داریم متفاوت است. برای انجام عملیات خواندن و نوشتن به ترتیب از دو نوع متغیر فایل `ifstream` و `ofstream` استفاده می‌شود، نحوه‌ی باز کردن فایل (مثلاً برای نوشتن در فایل) به صورت زیر است:

```
ofstream ("نام فایل") متغیر فایل ؛
```

متغیر فایل، متغیری است که از طریق آن می‌توانیم با فایل مورد نظر کار کنیم، دقیقاً مشابه حالتی که می‌توانستیم با استفاده از `cin` و `cout` با ورودی و خروجی روی کنسول کار می‌کنیم. در قسمت نام فایل، اگر نام فایل را وارد کنیم، کامپایلر در پوشه‌ای که برنامه‌ی C++ قرار دارد، به دنبال آن فایل می‌گردد، و اگر بخواهیم از فایلی که در جای دیگری از کامپیوتر ذخیره شده است، استفاده کنیم، باید آدرس کامل آن فایل را در قسمت نام فایل وارد کنیم.

از `ofstream` برای نوشتن در فایل و از `ifstream` برای برای خواندن از فایل استفاده می‌شود. یعنی با متغیر فایلی که با `ofstream` تعریف شود، می‌توان مانند `cout` کار کرد (توسط عملگر `<>`). به عبارت دیگر، فقط برای نوشتن در فایل می‌توان از آن استفاده کرد، به همین ترتیب با متغیر فایلی که با `ifstream` تعریف شده، می‌توان مانند `cin` (خواندن از فایل) کار کرد (توسط عملگر `<>`).

خطاهای معمول برنامه نویسی



توجه کنید که در فایلی که آن را به منظور خواندن باز کرده باشیم، نمی‌توانیم بنویسیم و همچنین از فایلی که آن را به منظور نوشتن باز کرده‌ایم، نمی‌توانیم بخوانیم.

۴) مثال خواندن نمره از فایل

برنامه‌ای بنویسید که در آن نمره‌ی تعدادی دانش‌آموز که در یک فایل ذخیره شده را دریافت کرده و بیشترین نمره را به همراه نام دانش‌آموز صاحب نمره، در صفحه نمایش دهد. در هر خط نام و سپس نمره‌ی دانش‌آموز نوشته شده است.

برای این کار باید تا وقتی به پایان فایل نرسیده‌ایم، در یک حلقه ابتدا نام دانش‌آموز و نمره‌ی او را بخوانیم و در صورتی که نمره‌ی او بزرگ‌تر از نمرات قبلی بود نام و نمره‌ی او را به عنوان بالاترین نمره نگه داریم. در انتهای و پس از رسیدن به پایان فایل نام و نمره‌ی دانش‌آموز با بالاترین نمره چاپ می‌شود.

```

main.cpp

#include <iostream.h>
#include <fstream.h>
#include <string.h> ←

int main()
{
    int i, nomre, maximum;
    maximum = 0;
    char naam [50];
    char naam_maximum [50];
    ifstream fin("daneshamooz.txt");
    while (!fin.eof()) { ←
        fin >> naam;
        fin >> nomre;
        if (nomre > maximum) {
            maximum = nomre;
            strcpy (naam_maximum, naam); ←
        }
    }
    cout << "Balatarin nomre:" << endl;
    cout << naam_maximum << " " << maximum << endl;
    fin.close();
    system("pause");
    return 0;
}

```

دقت کنید که متغیر **maximum** که بزرگ‌ترین نمره در آن نگهداری می‌شود، در ابتدای کار به عدد صفر مقداردهی شده است، که در غیر این صورت با توجه به مقدار نامشخص این متغیر جواب درستی به دست نمی‌آوردیم.

در این مثال از یک تابع جدید به نام `(eof)` استفاده شده است. وظیفه‌ی `(eof)` (مخفف شده‌ی `end of file`) است که مشخص کند که به انتهای فایل رسیده‌ایم یا نه. در صورتی که به پایان فایل رسیده باشیم، این تابع مقدار `TRUE` برمی‌گرداند و در غیر این صورت (اگر به آخر فایل نرسیده باشیم)، مقدار `FALSE` برمی‌گرداند. بنابراین معنای شرط حلقه این است که تا وقتی به پایان فایل نرسیده‌ایم، حلقه را تکرار کن. حاصل اجرای برنامه برای فایلی با محتویات زیر را می‌بینید:

```

Mohamad 15
Kiavash 16
Kourosh 17
Ali 18
Gholi 10
Taghi 14
Saeed 17

```

```

Balatarin nomre:
Ali 18
Press any key to continue . . .

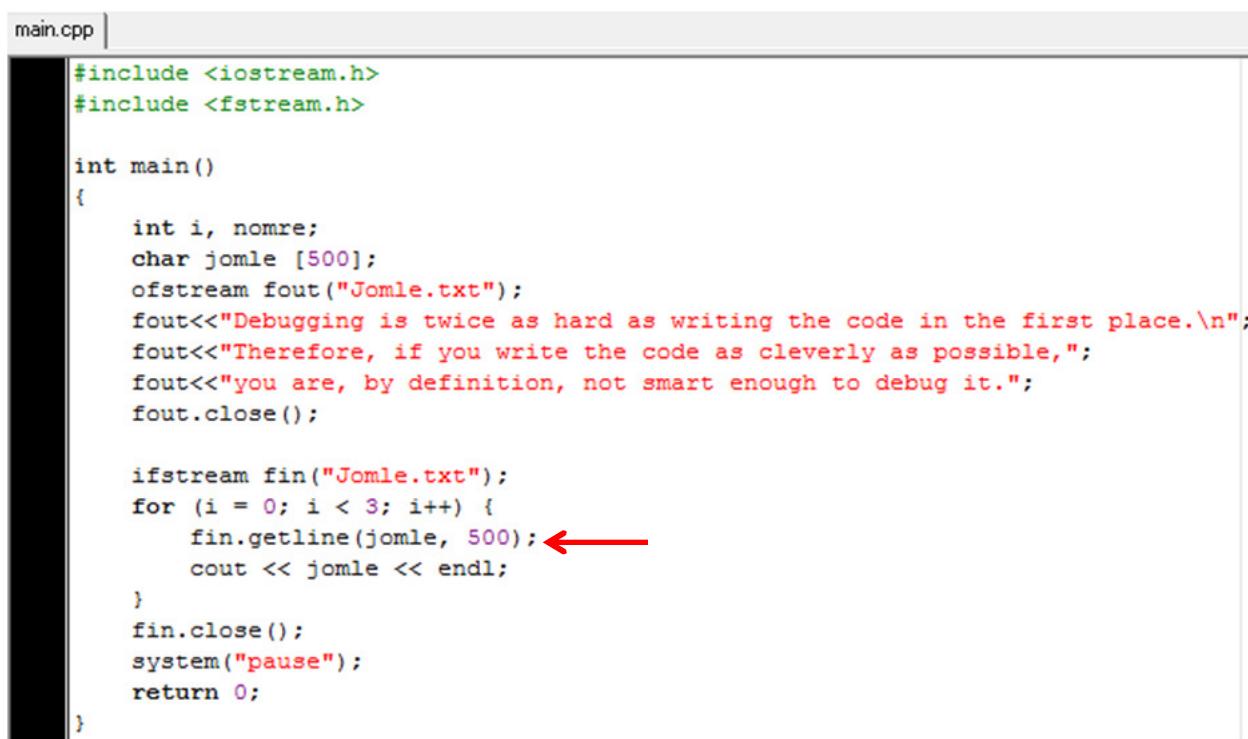
```

۵) مثال خواندن متن از فایل

فرض کنید می‌خواهیم متنی که در یک فایل نوشته شده است را بخوانیم و در جایی ذخیره کنیم اما دستور خواندن از فایل (توسط عملگر <>) تا رسیدن به کاراکتر فاصله (" ") یا پایان خط جلو می‌رود، و در نتیجه فقط یک کلمه را می‌توانیم بخوانیم، بنابراین اگر بخواهیم یک خط کامل را از فایل بخوانیم باید از دستور **getline** استفاده کنیم. طریقه‌ی استفاده از دستور **getline** را در مثال زیر می‌بینیم:

برنامه‌ای بنویسید که جمله‌ی زیر را در فایلی بنویسد و سپس آن را از فایل خوانده نمایش دهد:

“Debugging is twice as hard as writing the code in the first place.
Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”



```
main.cpp |  
#include <iostream.h>  
#include <fstream.h>  
  
int main()  
{  
    int i, nomre;  
    char jomle [500];  
    ofstream fout("Jomle.txt");  
    fout<<"Debugging is twice as hard as writing the code in the first place.\n";  
    fout<<"Therefore, if you write the code as cleverly as possible,";  
    fout<<"you are, by definition, not smart enough to debug it.";  
    fout.close();  
  
    ifstream fin("Jomle.txt");  
    for (i = 0; i < 3; i++) {  
        fin.getline(jomle, 500); ←  
        cout << jomle << endl;  
    }  
    fin.close();  
    system("pause");  
    return 0;  
}
```

برای این کار ابتدا باید فایل را برای نوشتن باز کنیم و جمله‌ی بالا (که از سه خط تشکیل شده است) را در آن چاپ کنیم، سپس باید فایل را بسته و آن را برای خواندن باز کنیم، و با دستور **getline** درون یک حلقه به طول سه (با توجه به این که جمله‌ی ما در سه خط چاپ شده است) جمله‌ی نوشته شده در فایل را از آن بخوانیم، سپس در صفحه‌ی نمایش چاپ کنیم. همان‌طور که در مثال بالا می‌بینید، دستور **getline** دو ورودی دارد، ورودی اول نام آرایه‌ای از جنس **char** است که پس از خواندن یک خط از فایل، اطلاعات خوانده شده در آن ذخیره می‌شود، ورودی دوم حداکثر تعداد کاراکتری است که می‌خواهیم بخوانیم، که در اینجا ما آن را ۵۰۰ فرض کردی‌ایم. نتیجه‌ی اجرای برنامه در زیر آمده است:

```
Debugging is twice as hard as writing the code in the first place.  
Therefore, if you write the code as cleverly as possible,  
you are, by definition, not smart enough to debug it.  
Press any key to continue . . .
```

توجه کنید که پس از پایان نوشتن در فایل و قبل از خواندن از آن، فایل را بسته‌ایم در غیر این صورت با خطا مواجه می‌شدیم.

خطاهای معمول برنامه نویسی



در صورتی که فایلی که می‌خواهیم در آن بنویسیم در توسط برنامه‌ی دیگری مورد استفاده باشد و یا اجازه‌ی ساختن فایل را در آدرس داده شده نداشته باشیم با پیغام خطای زمان اجرا روبرو خواهیم شد.

تمرین



- برنامه‌ای بنویسید که آدرس فایلی را دریافت کرده و محتويات آن را در در آدرس مقصد (که آن هم از کاربر دریافت می‌کنید) کپی کند.
- ساختاری برای **Student** تعریف کنید شامل نام، شماره دانش‌آموزی، شماره کلاس و معدل. سپس تابعی بنویسید که اطلاعات دانش‌آموزان مدرسه به صورت آرایه‌ای از **Student**‌ها و نام فایل مقصد را به عنوان پارامتر گرفته و اطلاعات دانش‌آموزان را در فایلی با این نام بریزد، به شکلی که بعداً بتوانیم به راحتی این اطلاعات را از فایل بخوانیم.
- تابعی بنویسید که نام فایلی که با قرارداد سوال ۲ ساخته شده است را به عنوان پارامتر گرفته و آرایه‌ی مربوطه از **Student**‌ها را بسازد و برگرداند.
- با استفاده از دو تابع سوال ۲ و ۳، تابعی بنویسید که آدرس یک فایل که اطلاعات دانش‌آموزان در آن قرار دارد را به عنوان پارامتر گرفته و محتوای فایل را بر اساس معدل مرتب کند.

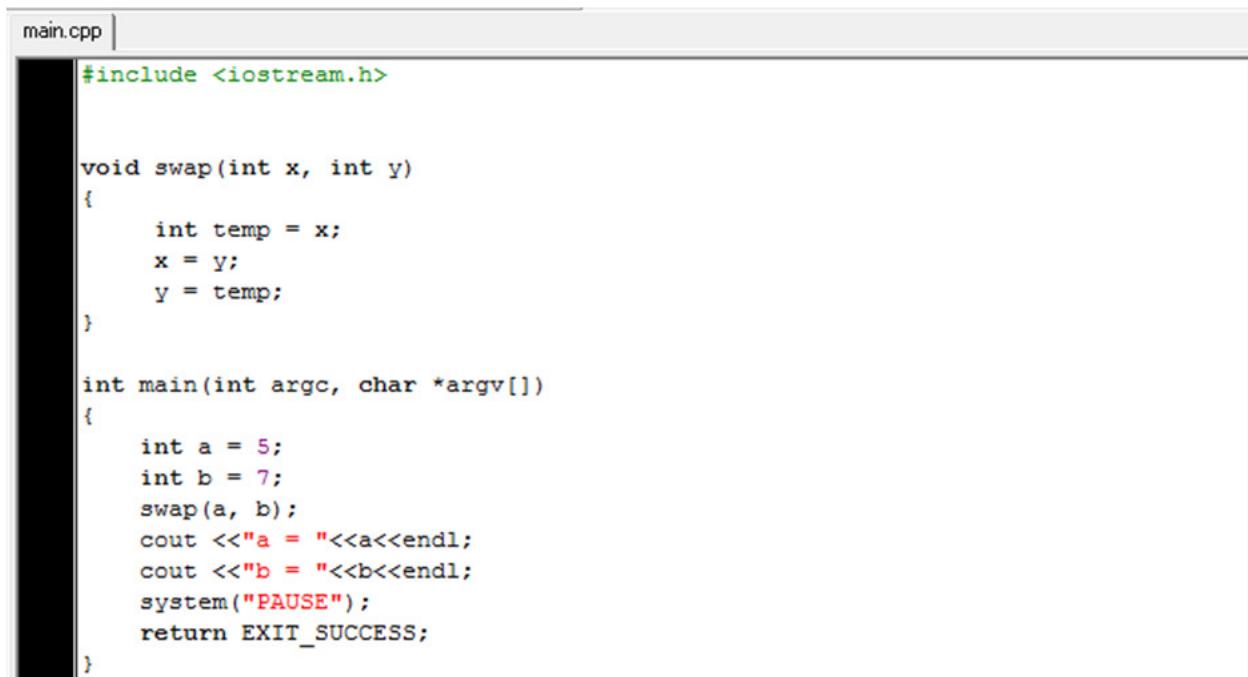
فصل چهاردهم: اشاره‌گرها

۱. اهمیت و معرفی اشاره‌گرها
۲. ارتباط آرایه‌ها و اشاره‌گرها
۳. آرایه‌های پویا
۴. مثال مرتب کردن دانش‌آموزان در آرایه‌ی پویا

۱) اهمیت و معرفی اشاره‌گرها

فرض کنید قرار است برنامه‌ی خیلی ساده‌ای بنویسیم، که مقدار دو متغیر را با هم عوض کند. از آن‌جایی که این قطعه کد در بسیاری از برنامه‌ها، از جمله در الگوریتم‌های مرتب‌سازی، بسیار پرکاربرد است، عقل حکم می‌کند که این قطعه کد تکرار شونده را در یک تابع قرار دهیم و در مواقعي که به این عملیات نیاز داریم، این تابع را صدا بزنیم.

یک تلاش اولیه برای پیاده‌سازی و استفاده از این تابع را در ادامه مشاهده می‌کنید:



```
main.cpp | #include <iostream.h>

void swap(int x, int y)
{
    int temp = x;
    x = y;
    y = temp;
}

int main(int argc, char *argv[])
{
    int a = 5;
    int b = 7;
    swap(a, b);
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

اگر نگاهی به برنامه‌ی بالا بیاندازید، می‌بینید که بعد از تعریف، به متغیرهای **a** و **b** به ترتیب مقادیر 5 و 7 داده‌ایم، سپس تابع **swap** را صدایدهایم. در تابع **swap**، همان سه خط کد معروف، برای عوض کردن متغیرها آمده است. بعد از فراخوانی تابع **swap** مقادیر **a** و **b** را چاپ کرده‌ایم. انتظار داریم که روی مانیتور، به ترتیب اعداد 7 و 5 را مشاهده کنیم، اما شکل زیر آن‌چیزی است که از اجرای این برنامه به دست می‌آید:

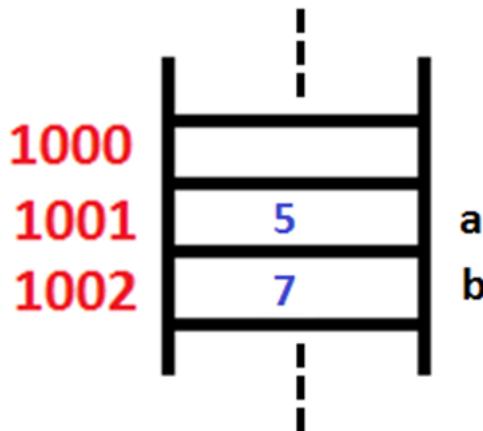
```
a = 5
b = 7
Press any key to continue . . .
```

همان‌طور که ملاحظه می‌فرمایید، این خروجی مطابق انتظار اولیه‌ی ما نبوده است و به نظر می‌رسد که تابع **swap** کار خاصی انجام نداده است.

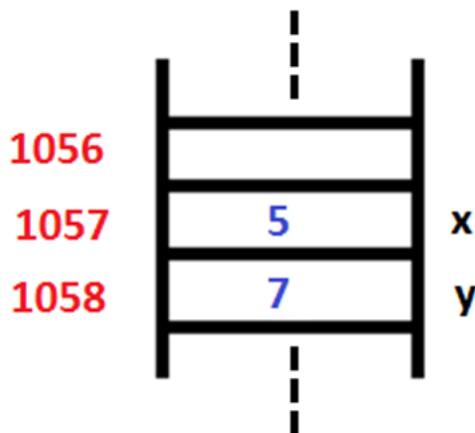
اگر مباحث مربوط به تابع را به خاطر بیاورید، (که حتماً می‌آورید!) آن‌جا عرض کردیم که متغیرهای مربوط به هر تابع، نسبت به متغیرهای تابع دیگر مستقل است. یعنی ربط خاصی به هم ندارند، جز این‌که مقدار متغیرها در شروع تابع با هم برابرند و بنایراین با تغییر دادن یکی،

دیگری تغییری نمی‌کند. پس نتیجه بدست آمده از برنامه بالا، چندان هم دور از انتظار نبوده است. برای این‌که دید عمیق‌تری از این مطلب پیدا کنید، در ادامه کمی مفصل‌تر به آن می‌پردازیم:

همانطور که می‌دانید، هر متغیر در یک خانه‌ی حافظه علاوه بر این‌که می‌تواند یک مقدار را درون خود نگه دارد، یک آدرس هم دارد. (همان‌طور که همه‌ی خانه‌های یک شهر، یک آدرس دارند). اگر به مثال بالا برگردیم، می‌توانیم به عنوان مثال، فرض کنیم متغیرهای a و b که در تابع `main` تعریف شده‌اند، در خانه‌های حافظه به آدرس‌های 1001 و 1002 قرار گرفته‌اند، مانند شکل پایین:



وقتی تابع `swap` صدای زده می‌شود، پارامترهای X و Y در آدرس‌های دیگری از حافظه تولید می‌شوند و مقادیر a و b به ترتیب در آن‌ها قرار می‌گیرند. فرض کنید که آدرس‌های مربوط به X و Y به ترتیب 1057 و 1058 باشند:



بعد از اجرای تابع `swap`، مقادیر مربوط به متغیرهای X و Y (در آدرس‌های 1057 و 1058) تغییر می‌کنند، اما متغیرهای a و b که در مکان متفاوتی قرار دارند (در آدرس‌های 1001 و 1002)، دست نخورده باقی می‌مانند.

در هر صورت این‌که تلاش ما برای تغییر دادن مقدار دو متغیر توسط یک تابع، ناموفق بوده است، برای ما مطلوب نیست. در واقع مشکل ما از این‌جا نشات می‌گیرد که پارامترهایی که به تابع می‌دهیم (a و b ، به نسبت پارامترهایی که در تابع داریم (X و Y ، در مکان‌های متفاوتی از حافظه قرار دارند).

اما چطور می‌توانیم این مشکل را حل کنیم؟

اگر می‌توانستیم به جای این که مقدار متغیرهای *a* و *b* را به تابع بدھیم (5 و 7)، آدرس آن‌ها را به تابع بدھیم (1002 و 1001)، به راحتی می‌توانستیم این مشکل را حل کنیم. به جای این که بگوییم: "مقدار *X* و *Y* را با هم عوض کن"، می‌توانستیم بگوییم: "خانه‌هایی که آدرس 1002 و 1001 دارند (در واقع خود *a* و *b*) را با هم عوض کن." و به این ترتیب واقعاً خود *a* و *b* عوض می‌شوند.

برای این که بتوانیم این شرح فارسی را به برنامه تبدیل کنیم، زبان C باید دو امکان را برای ما فراهم کند:

۱. از روی یک متغیر (مثل *a*) بتوانیم آدرس آن را بدست بیاوریم تا به جای آن که مقدار آن را به تابع بدھیم، آدرس آن را بدھیم.
۲. با داشتن آدرس یک متغیر، بتوانیم به خود متغیر دسترسی پیدا کنیم. (مقدار آن را بخوانیم یا تغییر دهیم)

خوبی‌خوبی مثل همیشه، زبان C نیازهای ما را بی‌آن که نیازی به مطرح کردن آن باشد!- شناسایی و برطرف کرده است. لطفاً به آن‌چه در ادامه می‌آید، خوب توجه کنید:

امکانات مربوط به نیاز اول:

هرگاه متغیری از هر نوع دلخواهی (char, int, ...) داشته باشید، قرار دادن علامت `&` پشت سر نام آن، آدرس آن متغیر را به شما می‌دهد. مثلاً اگر متغیر *a* را داشته باشیم، `&a` نشان دهنده آدرس مربوط به آن متغیر است.

به عنوان مثال دو خط برنامه‌ی زیر، آدرس متغیر *a* در حافظه را چاپ می‌کند:

```
int a = 5;
cout << &a << endl;
```

اما همیشه وقتی پایی یک مقدار وسط باشد، مانند آدرس یک متغیر که یک مقدار است، (مثلاً 1057) ما به متغیری نیاز پیدا می‌کنیم که آن مقدار را در خود ذخیره کند. هر مقدار یک نوع دارد. مثلاً مقدار 100، 3.14 و 'k' به ترتیب نیاز به متغیرهایی از نوع int، float و char دارند. مقدار هر آدرس هم نیاز به نوع خاص و جدیدی از متغیرها دارد:

- آدرس متغیری که از نوع int باشد، نیاز به متغیری از نوع *int دارد.
- آدرس متغیری که از نوع float باشد، نیاز به متغیری از نوع *float دارد.
- آدرس متغیری که از نوع char باشد، نیاز به متغیری از نوع *char دارد.

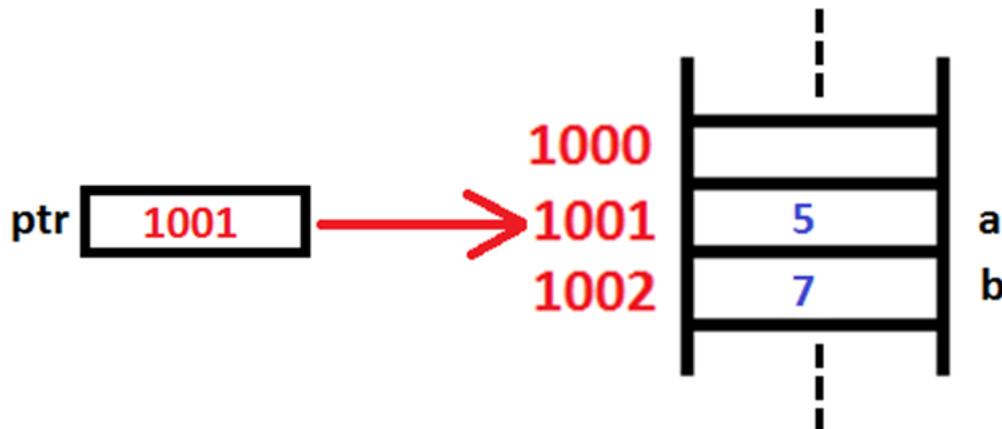
همانطور که می‌توانید حدس بزنید:

آدرس متغیری از نوع *X* نیاز به متغیری از نوع *X** دارد! (در اینجا به جای *X* هر کدام از انواع int, char, float، و ... می‌تواند باشد.)

در برنامه‌ی بالا، می‌توانیم در ابتدا مقدار آدرس متغیر *a* را در متغیر دیگری ذخیره کنیم، بعد مقدار آن متغیر که حاوی آن آدرس است را چاپ کنیم، همان‌طور که در ادامه می‌بینید:

```
int a = 5;
int* ptr = &a;
cout << ptr << endl;
```

در مثال بالا اگر فرض کنیم a در آدرس 1001 قرار داشته باشد، در این صورت متغیر ptr آدرس متغیر a را نگه خواهد داشت. پس ptr عدد 1001 را در خود خواهد داشت.



به متغیرهایی که آدرس یک متغیر دیگر را در خود نگه می‌دارند، **اشاره‌گر**^۱ گفته می‌شود. مثلاً در مثال بالا ptr اشاره‌گری به متغیر a می‌باشد.

امکانات مربوط به نیاز دوم

در صورتی که آدرس یک متغیر را داشته باشیم، با قرار دادن علامت * قبل از نام اشاره‌گر، می‌توانیم به خود متغیر دسترسی پیدا کنیم. مثلاً اگر متغیر ptr آدرس متغیر a را داشته باشد، *ptr دقیقاً معادل متغیر a خواهد بود.

به مثال زیر توجه کنید:

```
main.cpp | #include <iostream.h>

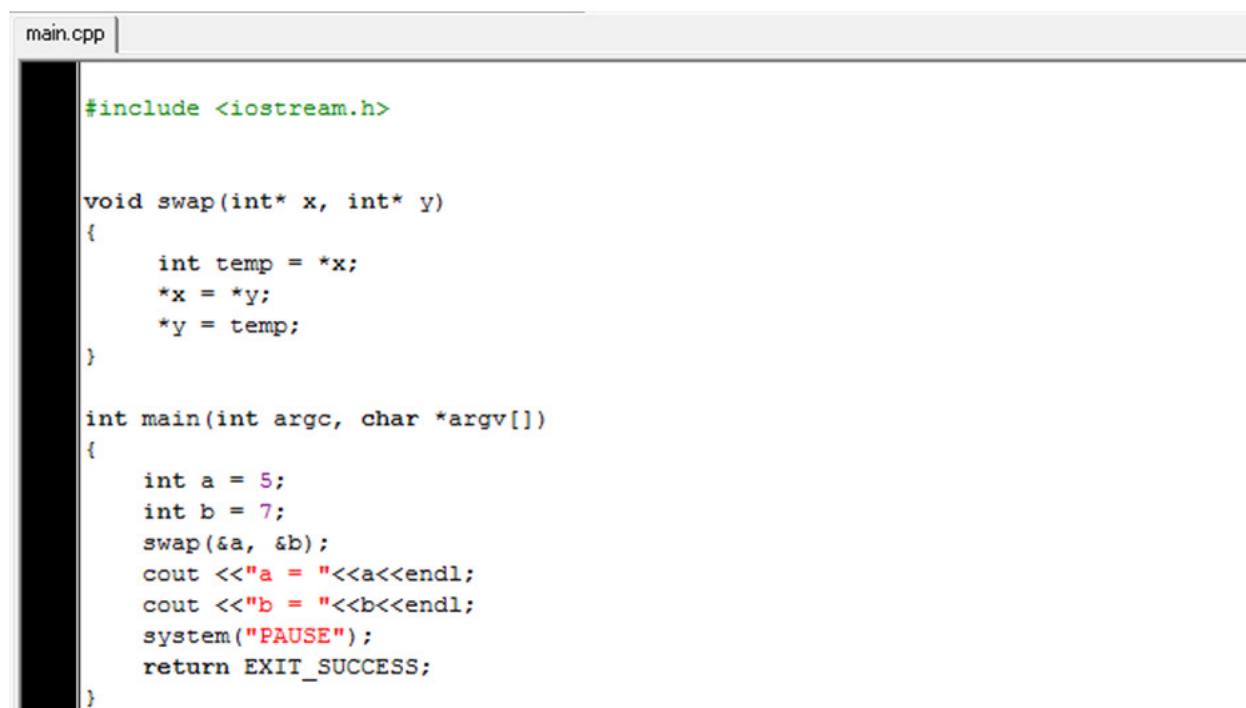
int main(int argc, char *argv[])
{
    int a = 5;
    int* ptr = &a;
    *ptr = 2;
    cout << "a = " << a << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

در این برنامه در خط سوم تابع `main` عدد 2 در خط دوم آدرس متغیر `a` را در خود گرفته است. چون `ptr` در خط دوم ریخته شده است. چون `ptr` در خط دوم آدرس متغیر `a` را در خود گرفته است، `*ptr` معادل متغیر `a` خواهد بود و در نتیجه بعد از اجرای خط سوم، متغیر `a` مقدار 2 به خود می‌گیرد.

خروجی این برنامه مطابق انتظار به شکل زیر است:

```
a = 2
Press any key to continue . . .
```

حالا که تا حدی با نحوه کارکردن با اشاره‌گرها آشنا شده‌ایم، می‌توانیم برنامه‌ی مربوط به جایگایی متغیرها را به شکل زیر بازنویسی کنیم:



```
main.cpp | #include <iostream.h>

void swap(int* x, int* y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main(int argc, char *argv[])
{
    int a = 5;
    int b = 7;
    swap(&a, &b);
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

در برنامه‌ی بالا در خط سوم تابع `main`، به جای این‌که `a` و `b` را به تابع ارسال کنیم، آدرس آن‌ها را ارسال کردکاریم. به این علت که `a` و `b` از نوع `int` هستند، در تابع `swap` پارامترها را باید از نوع `int*` تعریف کنیم. پس تا این‌جا می‌دانیم که `X` آدرس `a` و `y` آدرس `b` خواهد بود. در تابع `swap`، قصد داریم مقادیر متغیرهایی که آدرس آن‌ها را در `X` و `y` داریم، تغییر دهیم. همان‌طور که قبلاً ذکر شد، کافیست قبل از متغیر آدرس یک `*` قرار دهیم. یعنی `X*` و `y*` دقیقاً معادل `a` و `b` هستند.

خروجی برنامه هم درستی اجرای برنامه را تایید می‌کند:

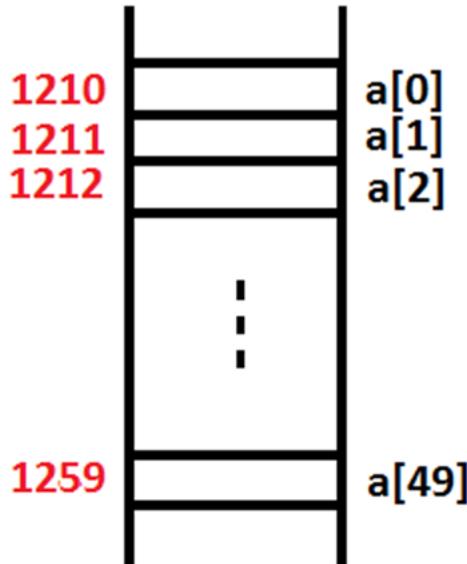
```
a = 7
b = 5
Press any key to continue . . .
```

۲) ارتباط آرایه‌ها و اشاره‌گرها

حتماً آرایه‌ها را از یاد نبرده‌اید: مجموعه‌ای از خانه‌های هم‌جنس پشت سر هم در حافظه که نمونه‌ای از آن را به شکل زیر تعریف می‌کردیم:

```
int a[50];
```

در شکل زیر این حقیقت که خانه‌های آرایه، دقیقاً خانه‌های متوالی آرایه هستند، دیده می‌شود



برای دسترسی به خانه با اندیس i ام آرایه، کافیست که به اولین خانه آرایه مراجعه کرده و i خانه جلوتر برویم. پس با داشتن آدرس اولین خانه آرایه، می‌توانیم به تمامی خانه‌های آن دسترسی پیدا کنیم.

بد نیست بدانید که نام آرایه، خود متغیری است که آدرس اولین خانه‌ی حافظه را در خود نگه می‌دارد. به عنوان مثال، دو خط برنامه زیر آدرس اولین خانه آرایه **a** را چاپ می‌کند:

```
int a[50];
cout<<a<<endl;
```

باتوجه به این‌که آدرس اولین خانه آرایه در متغیر **a** (که طبیعتاً از نوع **int*** است) قرار دارد، مقدار خانه اول آرایه را (که قبلاً آن را با **a[0]** بدست می‌آوردیم) می‌توانیم با **a*** هم بدست بیاوریم. همچنین آدرس خانه **a[1]** یکی بیشتر از آدرس **a[0]** است، یعنی آدرس آن **a+1** است. پس می‌توانیم به طریق مشابه، به جای **a[1]** **a[i]** بنویسیم (**a+i**)* و در حالت کلی‌تر **a[i]** معادل **(a+i)** خواهد بود.

برنامه‌ی زیر قرار است تابعی داشته باشد که با استفاده از شکل معادل جدید دسترسی به اندیس‌های آرایه، میانگین آن را به دست بیاورد:

```
main.cpp |
#include <iostream.h>

float average(int* a, int length)
{
    float sum = 0;
    for (int i=0;i<length;i++)
        sum += *(a+i);
    return sum/length;
}

int main(int argc, char *argv[])
{
    int a[6] = {5, 8, 1, 3, 4, 9};
    cout << average(a, 6)<<endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

نکته‌ی فنی

به تعریف تابع **average** دقت کنید. تا اینجا برای گرفتن آرایه‌ای از جنس **int a[]** به عنوان پارامتر، می‌نوشتیم: **int a[]**. اما چون نام آرایه، که به تابع داده می‌شود، آدرسی از خانه‌ی اول آرایه است و خانه‌ی اول هم از نوع **int** می‌باشد، نوع نام آرایه ***int** خواهد بود. پس تعجبی ندارد که این بار سمحض تنوع و استفاده از چیزهای جدیدی که یاد گرفتید- به جای **[]** نوشتیم: **a[]**. دسترسی به خانه‌های حافظه هم که دقیقاً مشابه چیزی است که توضیح آن را دادیم: **(a+i)** به جای **[a][i]** (سمضم تنوع!).

خروجی برنامه:

```
5
Press any key to continue . . .
```

نکته‌ی فنی

یادتان هست در فصل ۱۰ (تابع)، خدمتتان عرض کردیم که وقتی یک تابع می‌دهید، با تغییر دادن متغیر متناظر در تابع، متغیر اصلی تغییر نمی‌کند؟ اما در پایان آن قسمت گفتیم که آرایه‌ها در این مورد استثناء هستند. یعنی اگر آرایه‌ای را به تابعی بدهید و خانه‌های آرایه را در تابع تغییر بدهید، آرایه اصلی هم تغییر می‌کند. ذکر علت آن را هم به این فصل موکول کرده بودیم. احتمالاً خودتان دیگر می‌توانید دلیل آن را متوجه شوید: وقتی نام آرایه را به تابع می‌دهید، در واقع آدرس اولین عنصر آرایه را به تابع داده‌اید و با این کار واقعاً به خود آرایه دسترسی دارید، نه به یک کپی از آن. به این ترتیب با تغییر دادن عناصر آرایه در تابع، آرایه‌ی اصلی هم تغییر می‌کند.

۳) آرایه‌های پویا

چیزی که تا این جای کار از آرایه‌ها یاد گرفته‌اید، این است که:

- آرایه‌ها خانه‌های هم‌جنس پشت‌سر هم در حافظه هستند.
- نام آرایه اشاره‌گری به اولین عنصر آرایه است.
- طول آرایه باید یک عدد صحیح مشخص باشد.

به مورد سوم توجه کنید، این یعنی این که شما هنگام تعریف آرایه، نمی‌توانید طول آرایه را توسط یک متغیر بیان کنید.



خطاهای معمول برنامه نویسی

تعریف کردن آرایه به طولی که توسط یک متغیر تعیین شده است، یک خطای آشکار نحوی در برنامه‌ها است! به عنوان مثال قطعه کد زیر موجب بروز خطا در زمان کامپایل برنامه می‌گردد:

```
int len = 10;  
int a[len];
```

به این آرایه‌ها، که قبل از اجرا باید طول آن‌ها مشخص باشد، (طول توسط یک عدد ثابت مشخص می‌شود) آرایه‌ی ^{۵۲}ایستا گفته می‌شود.

اما هنگام برنامه نویسی موارد خیلی زیادی پیش می‌آید که ما نیاز به آرایه‌ای داریم که طول آن، قبل از اجرا کردن مشخص نیست. مثلاً ممکن است بخواهید قد دانش‌آموزان یک کلاس را گرفته و سپس مرتب کنید، اما از قبل مشخص نباشد آن کلاس، چند دانش‌آموز دارد. معمولاً راه حلی که در این موارد در نظر می‌گرفتیم، این بود که فرض می‌کردیم که یک حداکثری برای طول آرایه‌ی مورد نیاز، وجود داشته باشد و آرایه را از قبل به طول حداکثر تعریف می‌کردیم. مثلاً می‌توانیم فرض کنیم تعداد دانش‌آموزان یک کلاس، بیش از ۴۰ نفر نیست و از ابتدا یک آرایه‌ی ^{۵۳}عضوی تعریف کنیم.

اما این کار همیشه مقرن به صرفه نیست. موارد زیادی وجود دارند که این حداکثر تعداد، عدد بزرگی است اما عملاً ممکن است تنها چند خانه از آرایه استفاده شود و بقیه‌ی خانه‌ها هدر بروند. بنابراین ما نیاز به روشی برای ساخت آرایه‌هایی خواهیم داشت که طول آن‌ها در زمان اجرا (توسط یک متغیر) تعیین شود. به چنین آرایه‌هایی که طول آن‌ها توسط یک متغیر مشخص می‌شود، آرایه‌ی ^{۵۴}پویا گفته می‌شود.

در زبان C++ به شکل زیر می‌توانیم این کار انجام دهیم:

```
new [طول] جنس
```

مثال:

⁵² Static Array

⁵³ Dynamic Array

```
int* a = new int[len];
```

در خط بالا با استفاده از دستور `new`، مشخص شده است که آرایه‌ای از جنس `int` می‌خواهیم که طول آن توسط متغیر `len` مشخص شده است. این دستور، آرایه را در قسمتی از حافظه که امکان‌پذیر است، به برنامه اختصاص می‌دهد و آدرس اولین خانه‌ی آن را به ما برمی‌گرداند. از آن‌جا که داشتن آدرس اولین خانه برای دسترسی به هرکدام از خانه‌ها کافی و لازم است، آدرس اولین خانه را در یک متغیر اشاره‌گر، نگه‌داری می‌کنیم. چون جنس اولین خانه‌ی آرایه در اینجا `int` است، آدرس آن باید از نوع `int*` باشد. به همین دلیل، متغیری از نوع `a` در نظر گرفته شده و آدرس اولین خانه در آن ریخته شده است.

حالا که آدرس اولین خانه‌ی آرایه را داریم، برای دسترسی به `نامین` خانه‌ی آن کافیست از یکی از عبارات `a[i]` یا `(a+i)*` استفاده کنیم. پس در آرایه‌های پویا، تنها تعریف آرایه متفاوت با قبل است و هنگام استفاده از آن دقیقاً مانند آرایه‌های قبلی با آن برخورد می‌کنیم.

توجه کنید که فضایی که از آن برای آرایه پویا به شما اختصاص داده می‌شود، با فضای آرایه ایستا متفاوت است. آرایه‌ی ایستا در استک^{۵۴} و آرایه‌ی پویا روی هیپ^{۵۵} قرار دارند. از آنجایی که فضای استک بسیار محدودتر از فضای هیپ می‌باشد، ممکن است آرایه‌های بزرگی که استک حافظه‌ی لازم برای اختصاص دادن به آن نداشته باشد را بتوانید بدون مشکل به صورت پویا تعریف کنید.

C++ بسیار با شخصیت است! تا زمانی که خودتان ذکر نکنید که دیگر احتیاجی به آرایه‌ای که به صورت پویا به شما اختصاص داده شده است (با استفاده از دستور `new` ندارید، به آرایه‌ی شما دست نمی‌زند. اوج شخصیت C++ آن‌جا نمایان می‌شود که حتی اگر برنامه‌ی شما تمام شود و شما هنوز اعلام نکرده باشید که دیگر به آن آرایه نیاز ندارید، آرایه‌ی شما همچنان محفوظ باقی می‌ماند و هیچ کسی [برنامه‌ای]^{۵۶} حق ندارد به آن قسمت از حافظه تعلیم کند! و به این صورت است که شما قطعه‌ای از حافظه را از دست داده‌اید، چون نه شما، نه هیچ کس دیگری به آن دسترسی ندارد. در این‌جا اصطلاحاً گفته می‌شود که در حافظه زیاله^{۵۷} تولید کرده‌اید و باید کامپیوترا را `Restart` کنید، تا مجدداً بتوانید از تمام حافظه استفاده کنید.

عاقلانه‌ترین و طبیعی‌ترین راه جلوگیری از این مشکل، این است که خیلی ساده اعلام کنید که دیگر نیازی به حافظه‌ی مورد نظر ندارید. (پر واضح است که وقتی این کار را می‌کنید، که دیگر نیازی به آن حافظه نداشته باشید!) این کار توسط تابع `delete[]` انجام می‌شود. کافی است آدرس اولین خانه‌ی آرایه‌ای را که دیگر به آن احتیاج ندارید، به آن بدهید، مثل قطعه کد پایین:

```
int len = 100;
float* marks = new float[len];
//...
delete[] (marks);
```

⁵⁴ Stack

⁵⁵ Heap

⁵⁶ Garbage

۴) مثال مرتب کردن دانشآموزان در آرایه‌ی پویا

در ادامه، برنامه‌ی مربوط به مرتب کردن قد دانشآموزان را خواهیم دید. به این برنامه به عنوان ورودی، ابتدا تعداد دانشآموزان و سپس به همین تعداد اطلاعات دانشآموزان (نام و قد) داده می‌شود

در این برنامه به نکات زیر توجه کنید:

- چون دانشآموزان شامل دو فقره اطلاع نام و قد می‌باشند، از یک ساختار به نام **Student** برای آن استفاده کرده‌ایم.
- استفاده از **new**، محدود به نوع متغیرهای از پیش تعریف شده، مثل **int** و **char** و ... نیست. در اینجا با استفاده از آن یک آرایه‌ی پویا از جنس **Student** ساخته شده است.
- کل برنامه به وظایف معنی‌دار مشخص، تقسیم شده است: یک تابع برای خواندن ورودی، یک تابع برای مرتب کردن و یک تابع هم برای چاپ کردن خروجی. به این ترتیب، برنامه خواناتر شده است. این یعنی استفاده از برنامه‌نویسی پیمانه‌ای. سعی کنید به این عادات حسنی برنامه‌نویسی خوب بگیرید!
- برای مرتب‌سازی نزولی دانشآموزان بر اساس قد، از روش مرتب‌سازی جبابی (که معرف حضورتان هستند) استفاده شده است.
- در انتهای برنامه فراموش نکرده‌ایم که آرایه را از حافظه پاک کنیم. (**delete[] (bacheha)**)

این بار برای تنوع، خروجی برنامه را قبل از خود برنامه می‌بینیم. یک نمونه خروجی برنامه:

```
tedade danesh amoozan: 3
nam: ali
ghad: 173
nam: hasan
ghad: 165
nam: reza
ghad: 184
-----
nam: reza, ghad: 184
nam: ali, ghad: 173
nam: hasan, ghad: 165
Press any key to continue . . .
```

```
#include <iostream.h>

struct Student{
    char name[20];
    int ghad;
};

void input(Student* bacheha, int n)
{
    for (int i=0;i<n;i++)
    {
        cout<<"nam: ";
        cin>>bacheha[i].name;
        cout <<"ghad: ";
        cin>>bacheha[i].ghad;
    }
}

void output(Student* bacheha, int n)
{
    cout<<"-----"<<endl;
    for (int i=0;i<n;i++)
    {
        cout<<"nam: "<<bacheha[i].name;;
        cout <<", ghad: "<<bacheha[i].ghad<<endl;
    }
}

void sort(Student* bacheha, int n)
{
    for (int i=0;i<n;i++)
        for (int j=0;j<n-i-1;j++)
            if (bacheha[j].ghad < bacheha[j+1].ghad)
            {
                Student temp = bacheha[j];
                bacheha[j] = bacheha[j+1];
                bacheha[j+1] = temp;
            }
}

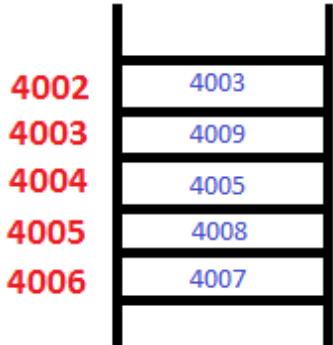
int main(int argc, char *argv[])
{
    int n;
    cout <<"tedade danesh amoozan: ";
    cin>>n;
    Student* bacheha = new Student[n];
    input(bacheha, n);
    sort(bacheha, n);
    output(bacheha, n);
    delete[] (bacheha);
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

تمرین



۱. با فرض اینکه `X` یک متغیر از نوع `int*` بوده و در آدرس `4002` حافظه قرار دارد، مقدار عبارت زیر را مشخص کنید: محتويات حافظه هم در شکل زیر مشخص شده است.

- الف. `X`
- ب. `*X`
- ج. `&X`



۲. تحقیق کنید چطور می‌توان یک آرایه‌ی دو بعدی پویا با ابعاد $m \times n$ ساخت. همچنین راهی برای پاک کردن کامل آن از حافظه پیدا کنید.
۳. بعضی از توابع ممکن است نیاز داشته باشند بیش از یک مقدار برگردانند. با استفاده از اشاره‌گرها چطور می‌توان بیش از یک مقدار را به عنوان خروجی تابع برگرداند؟
۴. خروجی برنامه‌ی زیر چیست؟

```
#include <iostream.h>

void f(int param1, int** param3)
{
    param1 = 10;
    int* var = &param1;
    **param3 = 57;
    param3 = &var;
    **param3 = 21;
    cout << **param3 << " " << param1 << " " << *var << endl;
}

int main(int argc, char *argv[])
{
    int k = 7;
    int m = 2;
    int* ptr1 = &m;
    int** ptr2 = &ptr1;
    f(k, ptr2);
    cout << m << " " << k << " " << *ptr1 << " " << **ptr2 << endl;
    return EXIT_SUCCESS;
}
```

ضمیمه ۴

جدول کدهای اسکی

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	Ø	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	\$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(72	48	H	104	68	h
9	09	Horizontal tab	41	29)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	Ø	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	:	91	5B	[123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	
29	1D	Group separator	61	3D	=	93	5D]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□