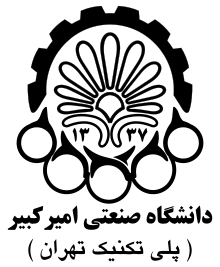
**بسم الله الرحمن الرحیم**



**گزارش تمرین سری چهارم**

**برنامه سازی پیشرفته**

**سیده معصومه سجادی**

**ش.د:۹۳۲۳۰۸۷**

اردیبهشت 1398

سوالات C++

سوال 1:

* 1. Dive deeper into C++ with move semantics, a key component of C++ resource management. Move semantics allow you to optimize the transfer of data and reallocate memory without extra copy operations. In this course, Bill Weinman helps you understand rvalues (temporary values that are the key to move semantics); create move constructors and operations; and use the copy-and-swap idiom to "swap in" new data while destroying the old.
  2. The word polymorphism means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance. C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.
  3. pure abstract . کلاس ارث دهنده ای با وضعیت pure abstract ، دو ویژگی خواهد داشت:

**ویژگی اول:** امکان تعریف تابع pure abstract آن داخل خودش نیست.

**ویژگی دوم:** تمامی کلاس های ارث برنده، موظف به تعریف آن تابع می باشند.

* 1. override . با قرار دادن این عبارت جلوی توابع مشترک بین کلاس ارث دهنده و کلاس ارث برنده، کامپایلر کاربر را موظف می کند که آن تابع را در کلاس ارث برنده بازتعریف کند. در نتیجه کاربرد این عبارت، در ارث بری کلاس ها می باشد.
  2. inline . با نوشتن عبارت inline در ابتدای تابع، کامپایلر کد تابع را عینا در محل فراخوانی شدن آن قرار می دهد، بدون اینکه نیازی به انجام مراحل واقعی فراخوانی تابع و پروسه های مربوط به آن باشد. این کار منجر به افزایش سرعت کد می شود، ولی در عین حال، حجم فایل اجرایی را نیز افزایش می دهد. به همین دلیل استفاده از inline در توابع کوچک بسیار مفید و کاربردی می باشد، ولی استفاده از آن در توابع بزرگتر ممکن است غیرمفید و یا حتی امکان ناپذیر باشد.
  3. . explicit قرار دادن عبارت explicit در ابتدای تعریف تابع، منجر به جلوگیری از فراخوانی تابع به صورت ضمنی می گردد. به عبارت دیگر، تابع تنها زمانی فراخوانی می شود که ورودی تعریف شده برای آن، با ورودی اعمال شده به آن عینا یکسان باشد و تابع به طور صریح فراخوانی شده باشد.

سوال ۲:

سوال ۳ و سوال ۴:

در این سوال قصد داریم کلاس Shape را به صورت خلاصه و به عنوان کلاس پایه تعریف کنیم. در مرحله بعد دو کلاس TwoDimensionalShape و ThreeDimensionalShape را به ترتیب برای اشکال دو بعدی و اشکال سه بعدی که از کلاس پایه Shape ارث می برند تعریف می نماییم. در مرحله بعد دو کلاس Circle و Square را به عنوان ارث برنده از کلاس TwoDimensionalShape و دو کلاس Sphere و Cube را به عنوان ارث برنده از کلاس ThreeDimensionalShape پیاده سازی می کنیم. در نهایت همه کلاس های پیاده سازی شده باید تابع main داده شده را کاملا دقیق اجرا کنند و خروجی مورد نظر باید مثل خروجی داده شده در صورت سوال باشد.

در کلاس هایی که از کلاس مبنا ارث می برند توابع area() و volume() که محاسبات مساحت و حجم را برای اشکال انجام می دهند تعریف می کنیم و توجه می کنیم که برای اشکال دو بعدی حجم بی معنی خواهد بود. این توابع را به صورت virtual در کلاس ها استفاده می نماییم. در واقع در prototype توابعی که قرار است روی آنها late binding انجام شود عبارت virtual قرار می دهیم. اصطلاح late binding به این معناست که فرض کنید چند تابع با یک نام در کلاس های مختلف وجود دارد و در تابع اصلی main این تابع با نام مورد نظر روی یک شیء می خواهد اعمال شود. کامپایلر هنگامی که به این خط از کد می رسد باید تشخیص دهد که شیء مورد نظر مربوط به کدام یک از کلاس های ارث دهنده یا ارث برنده است، سپس تابع اعمال شده را از همان کلاس برای اجرا روی شیء فراخوانی کند نه توابع با همان نام از کلاس های دیگر که این حالت در زمان run time توسط کامپایلر انجام می شود. این دقیقا یکی از مصداق های کلاس های polymorphism می باشد. طبق همین تعریف چون مساحت و حجم هر یک از اشکال با روابط مخصوص به خود به دست می آید بنابراین باید این توابع را در همه کلاس ها تعریف کرده و از عبارت virtual در prototype آنها در فایل های header استفاده کنیم. همچنین چون خروجی های مختلفی برای هر یک از اشکال داریم روی توابع print هم late binding انجام داده و عبارت virtual را در آنها به کار می بریم.

در قسمت پایانی تابع main یک حلقه for داده شده که برای 4 شیء ساخته شده باید 4 خروجی مربوط به مشخصات اشکال را چاپ کند. همه چیز باید با یک خط کد cout << \*( ptr[ x ] ) << '\n'; چاپ شود. بنابراین باید اپراتور << را به گونه ای تعریف کنیم که دو ورودی cout به صورت رفرنسی از کلاس std::ostram و دیگری \*( ptr[ x ] ) به صورت رفرنسی از کلاس Shape ( یک شیء) بگیرد و رفرنس گرفته شده از کلاس std::ostram را به عنوان ورودی تابع print در هر یک از کلاس ها قرار دهد و درون خود اپراتور این تابع را روی همان شیء گرفته شده اعمال کند و در نهایت یک رفرنس از کلاس std::ostram برگرداند که منجر به خروجی های مورد نظر شود. این اپراتور را فقط در کلاس پایه Shape تعریف می نماییم.

در تمرین قبلی کلاس نقطه را تعریف کرده و پیاده سازی نمودیم. در مرحله آخر اپراتور جمع را برای کلاس های ارث برنده طوری به کار می بریم که با جمع عنصری از کلاس نقطه و کلاس های مشتق شده بتواند مرکز اشکال را به اندازه مختصات داده شده برای تست جابجا نماید. این بخش به صورت یک حلقه for در آخر تابع main اضافه شده است که یکی یکی مرکز اولیه اشکال ساخته شده را می گیرد با مختصات یک نقطه که برای تست داده شده (3 و 2) جمع می کند و مراکز جدید اشکال را در خروجی نمایش می دهد.

سوال ۵:

در این سوال قصد داریم کلاسی با نام Stack پیاده سازی نماییم به طوریکه بتواند کلاس های مختلف را در خود ذخیره کند و توسط کد زیر فراخوانی شود.

**# کد داده شده در صورت سوال:**



**-** **منطق کد ارائه شده:**

در این کد، ابتدا یک stack از نوع shared\_ptr<CText> تولید می نماییم. این نوع پشته، معادل با این است که یک پوینتر به نوع CText ، (CText\* ) ساخته باشیم با این تفاوت که در آخر برنامه پوینتر دینامیکی را خود کامپایلر از بین می برد. در مرحله بعد، این پشته را توسط make\_shared<CText>(base + a) پر می کنیم. می توان تابع make\_shared را معادل با new کردن دانست. به عبارت دیگر، این خط معادل با نوشتن new CText {base+a} می باشد. base ، یک متغیر stringاست که به کمک اپراتور + موجود در کتابخانه string ، با کاراکتر داخلی حلقه for یعنی متغیر a ترکیب می شود، در نتیجه عباراتی مانند TEXTA ، TEXTB و... ساخته شده و درون stack قرار می گیرد. در مرحله ی بعد، تابع getCount() را به منظور شمارش تعداد اعضای موجود در stack می نویسیم. در این تابع، با ساختن یک stack به صورت template و برابر قرار دادن آن با stack موجود و پیش رفتن به سمت درون آن تا رسیدن به nullptr ، تعداد اعضای stack را به دست میآوریم. سپس، متغیری به نام N را با خروجی این تابع برابر قرار می دهیم. حال به کمک تابع isEmpty() ، بررسی می کنیم که stack خالی شده است یا نه. با توجه به خطوط قبلی کد، این شرط برقرار نخواهد بود. اگر این شرط برقرار می شد، دیگر نیازی به اجرای بقیه کد نبود و می توانستیم از برنامه خارج شویم. پس از آن، به تعداد اعضای موجود در stack ، عملیات pop را انجام می دهیم. در نهایت، مجددا خالی بودن stack را به کمک تابع isEmpty() بررسی می نماییم. متوجه می شویم که این بار شرط به درستی برقرار بوده و stack تخلیه شده است.

**-** **خروجی کد به صورت زیر می باشد:**



**-** **نحوه ی به دست آمدن خروجی کد:**

stack به صورت یک حافظه LIFO عمل می کند که سرواژه ی عبارت Last In First Out می باشد. این عبارت یعنی آخرین عضو وارد شده به stack، اولین عضوی است که از آن خارج می گردد. در کد داده شده، اولین عضو ورودی TEXTA و آخرین عضو ورودی TEXTZ می باشد. بنابراین، همانطور که در خروجی کد مشاهده می گردد، اولین عضو خروجی TEXTZ و آخرین آنها TEXTA می باشد. همچنین همانطور که انتظار داشتیم، عبارت Stack is empty مربوط به if دوم، در انتهای خروجی چاپ شده است.

**سوال ۶:**

**الف)** برای بخش اول این سوال دو نوع راه حل نوشته ایم که در دو فایل اول ذخیره شده اند.

در حالت کلی مشاهده میکنیم که دستور remove قادر به \اک کردن درایه مورد نظر ما نمیباشد و برای این که عدد ۲ را از داخل وکتور مورد نظر حذف کنیم از دستور eraseاستفاده میکنیم .البته در کد دوم سعی شده با هر دوی این دستورات خروجی دلخواه را بدست آوریم اما باید دقت شود که حذف کردن در واقع با دستور erase صورت میگیرد.

**ب)**در بخش دوم برای اینکه از حلقه استفاده نکنیم از دستور زیر بهره میبریم و دو وکتور قبلی که ساخته بودیم را در وکتور جدیدی ذخیره میکنیم البته تمام این ها با همان دو وکتور قبلی قابل انجام بود اما برای وضوح کار آنها را جداگانه نمایش میدهیم:

// vector\*2 without loop

std::transform (vec1.begin(), vec1.end(), vec2.begin(), vec3.begin(), std::plus<int>());

.ج) در این بخش نیز با استفاده از سورت کردن باید مرتب سازی را انجام دهیم .ابتدا میزان میانگین را از وکتورمان کم کرده و تفاضل را بدست می آوریم و بعد آن را سورت میکنیم.

د)برای حذف مقادیر تکراری از دستورات ست استفاده میکنیم.

ر) این بخش را نیز با استفاده از کدی متغیر و تعاریف توابع به گونه ای مینویسیم ک بتوان عدد دلخواه را برای تعیین مقدار مطلوب وکتور تعیین کرد .

**:سوال ۷**

همانطور که گفته شده در این سوال استفاه از حلقه ها ممنوع است. یک container از نوع vector تعریف می کنیم به طول 50 و باید آن را با مقادیر1 تا 50 پر کنیم. یک تابع به نام fill() برای پر کردن وکتور می نویسیم، این تابع یک تابع بازگشتی است و دو ورودی می گیرد، یک ورودی وکتور و یک ورودی، عنصر اول وکتور و هر بار یک واحد از عنصر اول وکتور کم می کند و به ورودی خودش می فرستد در درون خود ( به صورت بازگشتی ) و همزمان عنصر را به داخل یک وکتور push\_back می کند. از یک تابع به نام print() برای چاپ کردن عناصر درون وکتور و از یک حلقه استفاده می نماییم. در مرحله بعد، عناصر درون وکتور را به صورت random به هم میریزیم، این کار را به کمک تابع shuffle() که سه ورودی می گیرد ( عنصر اول وکتور، عنصر آخر وکتور و متغیر تعریف شده برای اندازه گیری زمان سیستم ) و زمان سیستم ( std::chrono::system\_clock ) انجام می دهیم و دوباره تابع print() را برای نمایش عناصر فراخوانی می کنیم.

طبق گقته ی سوال جلو میرویم وبرای آن که به مشکل نخوریم بجای گرفتن یک آرایه ی جدید آرایه ای را که داریم و اعداد آن بین ۱ تا ۵۰ میباشد را بهم میریزیم و به عنوان آرایه های b , c استفاده میکنیم تا دیگر مقدار تکراری ای وجود نداشته باشد و بتوانیم عملیات های مختلف را انجام دهیم.خروجی به شکل زیر نمایش داده میشود :

