به نام خدا

برنامه‌سازی پیشرفته

آرش شفیعی

وراثت و چندریختی

وراثت

- وقتی چند کلاس ویژگی‌ها و رفتارهای مشترک دارند، باید آن ویژگی‌ها و رفتارها را برای همهٔ آن کلاس‌ها تعریف کرد.
- تعریف ویژگی‌ها و رفتارهای واحد در چند کلاس متفاوت معایبی دارد، از جمله اینکه هزینهٔ پیاده‌سازی افزایش می‌یابد، و همچنین اعمال تغییرات در پیاده‌سازی سخت تر می‌شود، چرا که اگر یکی از رفتارهای مشترک تغییر کند، همهٔ کلاس‌هایی که آن رفتار را پیاده‌سازی کرده‌اند، تغییر می‌کنند.
- به علاوه ممکن است بعد از تعریف یک کلاس، نیاز به تعریف کلاسی باشد که بسیاری از ویژگی‌های آن کلاس را داراست و تنها در چند ویژگی و رفتار با کلاس تعریف شده متفاوت است.
- مفهوم وراثت [1] در برنامه‌سازی شیءگرا روشی برای حل این مشکلات است: با استفاده از مفهوم وراثت، یک کلاس می‌تواند تمام ویژگی‌ها و رفتارهای مشترک را تعریف کند و بقیهٔ کلاس‌ها می‌توانند آن ویژگی‌ها و رفتارها را از آن کلاس به ارث ببرند.

[1] inheritance

وراثت

- کلاسی که ویژگی‌ها و رفتارهای مشترک را تعریف می‌کند کلاس پدر [1] یا کلاس مافوق یا کلاس پایه [2] و کلاسی که ویژگی‌ها و رفتارهای مشترک را به ارث می‌برد، کلاس فرزند [3] یا زیرکلاس یا کلاس مشتق‌شده [4] نامیده می‌شود. چندین کلاس می‌توانند از یک کلاس پایه مشتق شوند.
- برای مثال در یک سامانهٔ دانشگاهی، دانشجو `student` و مدرس `lecturer` دو کلاس متفاوت هستند که هر دو دارای ویژگی‌های نام `name`، کد ملی `id` و رفتار ورود به سیستم `()login` هستند. این ویژگی‌های مشترک را می‌توان در کلاسی به نام کلاس شخص `person` پیاده‌سازی کرد.

[1] parent class
[2] super class or base class
[3] child class
[4] subclass or derived class

وراثت

- برای پیاده‌سازی ارث‌بری، کلاس پدر را به صورت یک کلاس معمولی تعریف می‌کنیم و برای کلاس فرزند تعیین می‌کنیم که از چه کلاسی و با چه نوعی به ارث ببرد.
- نوع ارث‌بری می‌تواند عمومی `public`، خصوصی `private`، و محافظت‌شده `protected` باشد، که فعلا فقط در مورد سطح دسترسی عمومی صحبت می‌کنیم.
- بنابراین دو کلاس فرزند و پدر را به صورت زیر تعریف می‌کنیم.

```
۱ class Base {
۲     ...
۳ };
۴ class Derived : public Base {
۵     ...
۶ };
```

وراثت

- در وراثت عمومی (`public`) فرزندان مانند بقیه استفاده‌کنندگان کلاس پدر، به اعضای عمومی کلاس پدر دسترسی مستقیم دارند.
- اما در این نوع وراثت، فرزندان به اعضای خصوصی کلاس پدر دسترسی مستقیم ندارند و تنها باید از طریق توابع عمومی پدر به این اعضا دسترسی پیدا کنند.
- یکی از سطوح دسترسی تعریف شده در کلاس‌ها، سطح دسترسی حفاظت‌شده (`protected`) است. اگر عضوی به صورت حفاظت‌شده در کلاس پدر تعریف شده باشد، همهٔ فرزندان به آن اعضا دسترسی مستقیم دارند، اما استفاده‌کنندگان دیگر کلاس پدر به این اعضا دسترسی مستقیم ندارند.

وراثت

- در مثال قبل گفتیم کلاس دانشجو از کلاس شخص به ارث می‌برد. پس می‌توانیم آن را به صورت زیر تعریف کنیم.

```
۱  class person {
۲  protected:
۳      string name; ...
۴  public:
۵      login(); ...
۶  };
۷  class student : public person {
۸  private:
۹      double average; ...
۱۰ public:
۱۱     int getCourse(int); ...
۱۲ };
```

وراثت

- در این مثال کلاس دانشجو همهٔ ویژگی‌ها و رفتارهای کلاس شخص را دارد، بنابراین وقتی یک شیء از کلاس دانشجو ساخته می‌شود، این شیء در حافظه همهٔ ویژگی‌های کلاس دانشجو و کلاس شخص را نگهداری می‌کند.

```
۱  class person {
۲  protected:
۳      string name; ...
۴  public:
۵      login(); ...
۶  };
۷  class student : public person {
۸  private:
۹      double average; ...
۱۰ public:
۱۱     int getCourse(int); ...
۱۲ };
```

وراثت

- وقتی یک شیء از کلاس دانشجو ساخته شود، این شیء به همهٔ اعضای عمومی کلاس دانشجو و کلاس شخص دسترسی دارد.
- وقتی یک شیء فرزند ساخته می‌شود، ابتدا سازندهٔ پیش‌فرض کلاس پدر، و سپس سازندهٔ پیش‌فرض کلاس فرزند فراخوانی می‌شود. در صورتی که سازندهٔ پیش‌فرض وجود نداشته باشد، سازندهٔ غیرپیش‌فرض در کلاس فرزند باید مقادیر اولیه برای سازندهٔ غیرپیش‌فرض در کلاس پدر را (در لیست مقداردهی اولیه) تعیین کند تا سازندهٔ غیرپیش‌فرض کلاس پدر بتواند با مقادیر مورد نیاز فراخوانی شود.
- همچنین وقتی یک شیء فرزند تخریب می‌شود، ابتدا مخرب کلاس فرزند و سپس مخرب کلاس پدر فراخوانی می‌شود.
- این ترتیب فراخوانی به این علت است که ممکن است بعد از اینکه شیء در سازندهٔ پدر مقداردهی اولیه شد، سازندهٔ فرزند نیاز به مقادیری از کلاس پدر داشته باشد و همچنین وقتی کلاس فرزند تخریب می‌شود، ممکن است به مقادیری از کلاس پدر نیاز داشته باشد.

- پس اگر هر دو کلاس پدر و فرزند سازندهٔ پیش‌فرض داشته باشند، ابتدا سازندهٔ پیش‌فرض کلاس پدر و سپس کلاس فرزند، فراخوانی می‌شوند. اما اگر سازندهٔ پیش‌فرض وجود نداشته باشد و کلاس پدر در سازنده‌های خود مقادیر ورودی دریافت کند، کلاس فرزند نیز در سازنده‌های خود باید همان مقادیر ورودی را دریافت کرده و مقادیر اولیه را به کلاس پدر ارسال کند. این کار با استفاده از لیست مقداردهی اولیه انجام می‌شود.

```
۱ class person {
۲ public:
۳     person(string name) { ... }
۴ };
۵ class student : public person {
۶ public:
۷     student(string name) : person(name) { ... }
۸ };
```

وراثت

- کلاس فرزند می‌تواند رفتار کلاس پدر خود را لغو [1] کند و رفتاری را جایگزین آن کند.
- پس اگر تابعی برای یک کلاس پدر تعریف شده باشد، کلاس فرزند می‌تواند آن تابع را بازتعریف کند و بدینسان با فراخوانی تابع بر روی یک شیء از کلاس فرزند، تابع تعریف شده در کلاس فرزند فراخوانی می‌شود.

```
۱  class person {
۲  private: string name; int id;
۳  public:
۴      void print() { cout << name << " " << id << endl; }
۵  };
۶
۷  class student : public person {
۸  private: int average;
۹  public:
۱۰     void print() { person::print(); cout << average << endl; }
۱۱ };
```

[1] override

وراثت

- وراثت چندگانه [1] در زبان سی++ امکان‌پذیر است.
- یک کلاس می‌تواند به طور همزمان از دو کلاس به ارث ببرد. به طور مثال یک تدریس‌یار هم یک دانشجوی تحصیلات تکمیلی است و هم یک محقق. پس کلاس تدریس‌یار (assistant) هم از کلاس دانشجو (student) و هم از کلاس محقق (researcher) به ارث می‌برد.

```
۱  class student {
۲  protected : int average;
۳  };
۴  class researcher {
۵  protected: string subject;
۶  };
۷  class assistant : public student, public researcher {
۸  // assistant has access to both
۹  // student and researcher class members
۱۰ };
```

[1] multiple inheritance

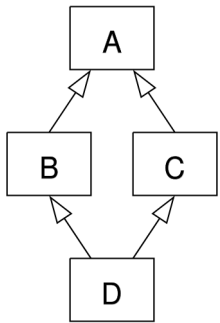
وراثت

- وراثت چندگانه ممکن است مشکلاتی را نیز به همراه داشته باشد.
- فرض کنید یک کلاس شخص (person) داریم که هم کلاس دانشجو و هم کلاس محقق از آن به ارث می‌برند. این کلاس یک ویژگی نام (name) دارد که در این مثال فرض می‌کنیم دسترسی آن عمومی است.
- حال کلاس تدریس‌یار را در نظر بگیرید. یک تدریس‌یار هم یک دانشجو است و هم یک محقق.
- یک شیء از کلاس تدریس‌یار می‌سازیم. این شیء یک متغیر نام دارد که از طریق کلاس دانشجو به ارث برده است و یک متغیر نام دارد که از طریق کلاس محقق به ارث برده است. پس با خطای کامپایلر روبرو می‌شویم.

```
۱ class person {  public: string name; ...};
۲ class student : public person { ... };
۳ class researcher : public person { ... };
۴ class assistant : public student, public researcher { ... };
۵ assistant ta;
۶ cout << ta.name; // name is ambiguous, because
۷ // name is found in different base classes.
```

وراثت

- به این مشکل، مشکل لوزی [1] نیز گفته می‌شود، زیرا وراثت چندگانه در این مواقع یک لوزی می‌سازد.
- کلاس‌های `B` و `C` از کلاس `A` به ارث می‌برند و کلاس `D` از کلاس‌های `B` و `C` به ارث می‌برد.



[1] diamond problem

وراثت

- یکی از راه‌های حل مشکل لوزی این است که استفاده‌کنندهٔ کلاس D به صراحت بیان کند آیا می‌خواهد به ویژگی مورد نظر خود از طریق کلاس B دسترسی پیدا کند و یا از طریق کلاس C.
- برای مثال اگر یک شیء از کلاس تدریس‌یار داشته باشیم، می‌توانیم به عضو داده‌ای name که از طریق کلاس‌های دانشجو و محقق به ارث برده شده است، با استفاده از عملگر تفکیک حوزه (::) [1] به صورت زیر دسترسی پیدا کنیم.

```
۱ class person {  public: string name; ...};
۲ class student : public person { ... };
۳ class researcher : public person { ... };
۴ class assistant : public student, public researcher { ... };
۵ assistant ta;
۶ cout << ta.student::name;
۷ cout << ta.researcher::name;
```

[1] scope resolution operator

- راه دوم برای حل مشکل لوزی این است که کلاس‌های B و C با استفاده از کلیدواژهٔ `virtual` از کلاس A به ارث ببرند. به این وراثت، وراثت مجازی [1] گفته می‌شود. در وراثت مجازی، کلاس D که از کلاس‌های B و C به ارث می‌برد، تنها از یک طریق ویژگی‌های کلاس A را به ارث می‌برد و یک کپی از آن ویژگی‌ها خواهد داشت.

```
۱ class person {  public: string name; ...};
۲ class student : virtual public person { ... };
۳ class researcher : virtual public person { ... };
۴ class assistant : public student, public researcher { ... };
۵ assistant ta;
۶ cout << ta.name; // ta object has only one name atrribute
```

[1] virtual inheritance

وراثت

- قوانین دسترسی در انواع وراثت عمومی، خصوصی، و حفاظت‌شده به طور خلاصه به صورت زیر است.

```
۱  class A {
۲  public: int x; protected: int y; private: int z;
۳  };
۴
۵  class B : public A {
۶  // x is public, y is protected, z is not accessible
۷  };
۸
۹  class C : protected A {
۱۰ // x is protected, y is protected, z is not accessible
۱۱ };
۱۲
۱۳ class D : private A {
۱۴ // x is private, y is private, z is not accessible
۱۵ };
```

وراثت

- پس در صورتی که کلاسی از کلاس B و C به ارث ببرد، دسترسی به متغیرهای x و y خواهد داشت، ولی اگر از کلاس D به ارث ببرد، دسترسی به هیچ یک از این متغیرها نخواهد داشت.

- از طرف دیگر اگر شیئی از کلاس‌های B ساخته شود، فقط دسترسی به متغیر x خواهد داشت، ولی اگر شیئی از کلاس‌های C یا D ساخته شود، دسترسی به هیچ یک از متغیرها نخواهد داشت.

```
۱  class A { public: int x; protected: int y; private: int z; };
۲  class B : public A {
۳  // x is public, y is protected, z is not accessible
۴  };
۵  class C : protected A {
۶  // x is protected, y is protected, z is not accessible
۷  };
۸  class D : private A {
۹  // x is private, y is private, z is not accessible
۱۰ };
```

وراثت

- از وراثت خصوصی در کلاس A وقتی استفاده می‌کنیم که به همهٔ رفتارهای یک کلاس پدر مثل B نیاز داریم، و می‌خواهیم در توابع دیگر کلاس A از این رفتارها استفاده کنیم. از طرفی نمی‌خواهیم کلاس‌هایی که از کلاس A به ارث می‌برند یا کاربران کلاس A از رفتارهای به ارث برده از کلاس B استفاده کنند.
- در چنین مواقعی B می‌تواند عضوی از A باشد. ولی به دلایلی ترجیح می‌دهیم که B پدر A باشد. یعنی به جای رابطهٔ ترکیب (یعنی A در ترکیب خود دارای B است)، از رابطهٔ وراثت (یعنی A از B به ارث برده است) استفاده می‌کنیم.

وراثت

- یکی از دلایل ترجیح رابطهٔ وراثت به ترکیب این است که کلاس فرزند بتواند از اعضای حفاظت‌شدهٔ کلاس پدر استفاده کند.
- یکی از دلایل دیگر این است که اگر کلاس پدر یعنی B هیچ دادهٔ عضوی نداشته باشد، در وراثت A از B کلاس پدر هیچ فضایی اشغال نمی‌کند ولی اگر A در ترکیب خود دارای B باشد، B در حافظه فضا اشغال می‌کند.
- از این نوع وراثت به ندرت استفاده می‌کنیم.

```
۱ class Car : private Engine {
۲ // car is not an engine, but needs all functions of an engine
۳ // car has only one engine.
۴ };
```

- موارد کاربرد وراثت حفاظت‌شده مانند وراثت خصوصی است با این تفاوت که در وراثت حفاظت شده، فرزندان A نیز می‌توانند به اعضای B به طور مستقیم دسترسی پیدا کنند. این نوع وراثت نیز به ندرت استفاده می‌شود.

وراثت

- کلاس فرزند می‌تواند سطح دسترسی به اعضای کلاس پدر خود را با استفاده از کلیدواژهٔ using برای تعدادی از ویژگی‌ها تغییر دهد.

```
۱  class A {
۲  public: int x1,x2; protected: int y1,y2; private: int z;
۳  };
۴  class B : public A { // x1 is public, y1 is protected
۵  private: using A::x2; // x2 is private
۶  public: using A::y2; // y2 is public
۷  };
۸  class C : protected A { // x1 is protected, y1,y2 are protected
۹  public : using A::x2; // x2 is public
۱۰ };
۱۱ class D : private A { // x1 is private, y1 is private
۱۲ protected: using A::x2; // x2 is protected
۱۳ public: using A::y2; // y2 is public
۱۴ };
```

وراثت

- توجه کنید که در وراثت، یک سربار زمانی به سیستم تحمیل می‌شود، چرا که در سازنده‌ها و مخرب‌ها باید چندین تابع باید اجرا شوند و همچنین در فراخوانی‌ها توابع بسته به شرایط باید از کلاس پدر یا فرزند فراخوانی شوند.
- پس بهتر است تنها در مواردی از وراثت استفاده کنیم که به آن نیاز داریم و در مواردی که وراثت ممکن است به کاهش بهره‌وری بیانجامد از آن استفاده نکنیم.

چندریختی

- در نظریهٔ زبان‌های برنامه‌نویسی، چندریختی [1] به معنای فراهم کردن یک رابط [2] برای موجوداتی از نوع‌های متفاوت است.
- در زبان سی++ نیز چندریختی قابلیتی است برای فراهم کردن امکان یک فراخوانی واحد برای اشیایی از کلاس‌های متفاوت.

[1] polymorphism
[2] interface

چندریختی

- قبل از توضیح قابلیت چندریختی، چند ویژگی تبدیل کلاس‌های فرزند به پدر و پدر به فرزند را بررسی می‌کنیم.
- فرض کنید دو شیء از کلاس فرزند و کلاس پدر می‌سازیم، و می‌خواهیم محتوای شیء از کلاس فرزند را در محتوای شیء از کلاس پدر کپی کنیم.

```
۱ class shape { ... };
۲ class circle : public shape { ... };
۳ shape shp;
۴ circle crc;
۵ shp = crc;
```

- با استفاده از عملگر تساوی که به طور پیش‌فرض سربازگذاری شده است، اعضای شیء کلاس فرزند یک به یک در اعضای شیء کلاس پدر کپی می‌شوند و از آنجایی که همهٔ اعضای کلاس پدر مقادیر مورد نیاز خود را دریافت می‌کنند هیچ مشکلی به وجود نخواهد آمد.

چندریختی

- چنانچه از اشاره‌گرهایی برای اشاره به اشیایی از کلاس‌های پدر و فرزند استفاده کنیم، همچنان امکان اشاره کردن یک شیء از کلاس پدر به شیئی از کلاس فرزند وجود خواهد داشت.

```
۱ class shape { ... };
۲ class circle : public shape { ... };
۳ shape* shp;
۴ circle* crc = new circle;
۵ shp = crc; // ok
۶ circle crc2;
۷ shp = &crc2; // ok
```

- از آنجایی که همهٔ اعضای کلاس پدر مقادیر مورد نیاز خود را از شیء فرزند دریافت می‌کنند، در اینجا نیز مشکلی به وجود نخواهد آمد.

چندریختی

- حال می‌خواهیم محتوای شیء از کلاس پدر را در محتوای شیء از کلاس فرزند کپی کنیم.

```
۱ class shape { ... };
۲ class circle : public shape { ... };
۳ shape shp;
۴ circle crc;
۵ crc = shp; // error
۶ circle * crcptr;
۷ crcptr = &shp // error
```

- اگر اعضای کلاس پدر یک‌به‌یک در اعضای کلاس فرزند کپی شوند، تعدادی از اعضای کلاس فرزند بدون مقدار باقی می‌مانند. کامپایلر در این حالت پیام خطا صادر می‌کند. به طور مشابه یک اشاره‌گر به شیئی از کلاس فرزند، نمی‌تواند به شیئی از کلاس پدر اشاره کند.

چندریختی

- حال ببینیم از نظر منطقی کامپایلر چگونه این مقداردهی‌ها را ترجمه می‌کند.
- وقتی می‌نویسیم `shp = crc` در واقع کامپایلر آن را به صورت `shp.operator=(crc)` ترجمه می‌کند. در کلاس `shape` عملگر تساوی به صورت پیش‌فرض به صورت زیر تعریف شده است.

```
1 shape& operator=(const shape& s);
```

- وقتی مقدار ورودی به این تابع یک شیء از کلاس `circle` باشد، کامپایلر از طریق روابط وراثت می‌داند که «یک `circle` یک `shape` است»، پس از نظر منطقی مشکلی به وجود نمی‌آید.

چندریختی

- اما وقتی می‌نویسیم `crc = shp` در واقع کامپایلر آن را به صورت `crc.operator=(shp)` ترجمه می‌کند. در کلاس `circle` عملگر تساوی به صورت پیش‌فرض به صورت زیر تعریف شده است.

```
1 circle& operator=(const circle& c);
```

- وقتی مقدار ورودی به این تابع یک شیء از کلاس `shape` باشد، کامپایلر از طریق روابط وراثت می‌داند که «یک `shape` یک `circle` نیست»، پس از نظر منطقی مشکلی به وجود می‌آید و کامپایلر پیام خطا صادر می‌کند.

چندریختی

- در صورتی که بخواهیم امکان کپی یک شیء از کلاس پدر را در یک شیء از کلاس فرزند فراهم کنیم، باید عملگر تساوی را برای آن تعریف کنیم.
- با تعریف عملگر تساوی، مشخص می‌کنیم متغیرهایی که در کلاس پدر وجود ندارند و در کلاس فرزند وجود دارند، چگونه باید مقداردهی شوند. به طور مثال متغیرهایی که در کلاس پدر وجود ندارند و در کلاس فرزند وجود دارند را با مقادیر صفر و رشته‌های تهی مقداردهی اولیه می‌کنیم.
- پس می‌توانیم تعریف کنیم:

```
1 circle& operator=(const shape& s) {
2  // copy members of shape s into members of this circle
3  // and initilize other members of this circle with 0 and ""
```

- حال وقتی می‌نویسیم `crc = shp` در واقع کامپایلر آن را به صورت `crc.operator=(shp)` ترجمه می‌کند که عملگر آن تعریف شده است.

چندریختی

- پس اگر کلاس پدر تابعی را تعریف کرده باشد، اشیایی از کلاس فرزند را در یک شیء از کلاس پدر کپی و تابع کلاس پدر را فراخوانی کرد.
- به طور مثال اگر آرایه‌ای از اشیایی از کلاس پدر داشته باشیم، می‌توانیم هر یک از عناصر آرایه را به یک شیء از یکی از کلاس‌های فرزند نسبت داده، و تابع کلاس پدر را بر روی آن اشیا فراخوانی کنیم.

```
۱  class shape {
۲  public: void move(int a, int b) { x+=a; y+=b; }
۳  };
۴  shape* shp[4];
۵  circle crc1,crc2;
۶  rectangle rect1, rect2;
۷  shp[0] = &crc1; shp[1] = &crc2;
۸  shp[2] = &rect1; shp[3] = &rect2;
۹  for (int i=0; i<4; i++)
۱۰     shp[i]->move(2,3); // shape::move(2,3) is called.
```

چندریختی

- همچنین می‌توانیم تابعی تعریف کنیم که به عنوان ورودی شیئی از کلاس پدر را دریافت کند و عملیاتی بر روی آن انجام دهد. بدین ترتیب اگر شیئی از یکی از کلاس‌های فرزند بدین تابع ارسال شود، تابع مورد نظر از کلاس پدر فراخوانی می‌شود.

```
۱  void move(shape * sh, int a, int b) {
۲      sh->move(a,b);
۳  }
۴  shape* shp[4];
۵  circle crc1,crc2;
۶  rectangle rect1, rect2;
۷  shp[0] = &crc1; shp[1] = &crc2;
۸  shp[2] = &rect1; shp[3] = &rect2;
۹  for (int i=0; i<4; i++)
۱۰     move(shp[i], 2, 3);
```

چندریختی

- حال فرض کنید یک اشاره‌گر به شیئی از کلاس پدر به یک شیء از کلاس فرزند اشاره می‌کند.

```
۱ class shape { ... };
۲ class circle : public shape { ... };
۳ shape* shp;
۴ circle* crc = new circle;
۵ shp = crc;
```

- اگر تابعی بر روی اشاره‌گر shp فراخوانی شود، و آن تابع هم در کلاس پدر و هم در کلاس فرزند تعریف شده باشد، آیا تابع کلاس پدر فراخوانی می‌شود و یا تابع کلاس فرزند؟

چندریختی

- هر شکل در حالت کلی یک مساحت دارد و یک دایره نیز مساحتی دارد که می‌توان آن را به نحوی خاص محاسبه کرد.

```
۱  class shape {
۲  public: int calcArea() { return 0; }
۳  };
۴  class circle : public shape {
۵  public : int calcArea() { return pi*r*r; }
۶  };
۷  shape* shp;
۸  circle* crc = new circle;
۹  shp = crc;
۱۰ shp->calcArea(); // shape::calcArea() is called.
```

- اگر تابعی بر روی اشاره‌گر shp فراخوانی شود، و آن تابع هم در کلاس پدر و هم در کلاس فرزند تعریف شده باشد، تابع کلاس پدر فراخوانی می‌شود.

چندریختی

- حال سناریوی زیر را در نظر بگیرید.
- می خواهیم آرایه‌ای از اشیایی تشکیل دهیم که همه فرزند یک پدر هستند و همه تعدادی رفتار مشابه دارند که از پدر به ارث برده‌اند، اما هر کدام این رفتار را به گونه‌ای متفاوت اجرا می‌کنند.
- برای مثال اشکال مختلف مانند دایره و مستطیل و مثلث و غیره همه می‌توانند مساحت خود را محاسبه کنند و همگی این رفتار را از پدر خود به ارث برده‌اند، اما نحوهٔ محاسبهٔ مساحت برای هر کدام از آنها متفاوت است. حال آرایه‌ای از اشکال متفاوت داریم و می‌خواهیم تابع محاسبه مساحت را برای همهٔ اعضای این لیست محاسبه کنیم.

چندریختی

- برای همهٔ اشاره‌گرهای زیر از کلاس پدر تابع calcArea از کلاس پدر فراخوانی می‌شود.

```
۱  class shape {
۲  public: int calcArea() { return 0; }
۳  };
۴  class circle : public shape {
۵  public : int calcArea() { return pi*r*r; }
۶  };
۷  class rectangle : public shape {
۸  public : int calcArea() { return w*l; }
۹  };
۱۰ shape* shp[4];
۱۱ circle crc1,crc2;
۱۲ rectangle rect1, rect2;
۱۳ shp[0] = &crc1; shp[1] = &crc2;
۱۴ shp[2] = &rect1; shp[3] = &rect2;
۱۵ for (int i=0; i<4; i++)
۱۶     shp[i]->calcArea(); // shape::calcArea() is called.
```

چندریختی

- در مثال قبل به دنبال یک قابلیت از زبان هستیم که با استفاده از آن بتوانیم بر روی اشیایی از کلاس پدر توابعی از کلاس فرزندان را فراخوانی کنیم.
- این ویژگی در زبان‌های شیءگرا وجود دارد و به آن چندریختی می‌گوییم.
- چندریختی قابلیتی است که توسط آن برای موجوداتی از نوع‌های متفاوت یک رابط واحد تعریف می‌شود.
- در اینجا موجودات متفاوت اشیای متفاوت هستند از کلاس‌هایی که همه فرزند یک پدر هستند و رابط واحد در اینجا یک تابع واحد است که توسط کلاس پدر تعریف شده است.
- اگر یک کلاس پدر تابعی را با استفاده از کلیدواژهٔ `virtual` تعریف کند و این تابع توسط فرزندان پیاده‌سازی شود، و اگر اشاره‌گری از کلاس پدر به شیئی از کلاس فرزند اشاره کند، آنگاه با فراخوانی آن تابع بر روی اشاره‌گر از کلاس پدر، تابع کلاس فرزند فراخوانی خواهد شد.

چندریختی

- تابعی که توسط کلمهٔ `virtual` تعریف شده است، و می تواند با یک نام واحد بسته به کلاسی که آن را فراخوانی می‌کند شکل‌ها یا ریخت‌های متفاوت داشته باشد، یک تابع چندریخت [1] نامیده می‌شود.
- کلاسی که یک تابع چندریخت را تعریف کند یا به ارث ببرد، یک کلاس چندریخت [2] نامیده می‌شود.

[1] polymorphic function
[2] polymorphic class

چندریختی

- برای همهٔ اشاره‌گرهای زیر از کلاس پدر تابع calcArea از کلاس فرزند فراخوانی می‌شود.

```
۱  class shape {
۲  public: virtual int calcArea() { return 0; }
۳  };
۴  class circle : public shape {
۵  public : int calcArea() { return pi*r*r; }
۶  };
۷  class rectangle : public shape {
۸  public : int calcArea() { return w*l; }
۹  };
۱۰ shape* shp[4];
۱۱ circle crc1,crc2; rectangle rect1, rect2;
۱۲ shp[0] = &crc1; shp[1] = &crc2; shp[2] = &rect1; shp[3] = &rect2;
۱۳ for (int i=0; i<4; i++)
۱۴     shp[i]->calcArea();
۱۵     // circle::calcArea() or rectangle::calcArea() is called.
```

چندریختی

- قابلیت چندریختی تنها زمانی امکان‌پذیر است که یک اشاره‌گر از نوع کلاس پدر به یک شیء از کلاس فرزند اشاره کند و تابعی مجازی را فراخوانی کند که توسط فرزند نیز پیاده‌سازی شده است.
- اگر شیئی از کلاس پدر داشته باشیم و شیئی از کلاس فرزند را به آن نسبت دهیم از چندریختی نمی‌توانیم استفاده کنیم.

```
1 shape shp; circle crc;
2 shp = crc; // here we copy members of crc into members of shp
3 shp.calcArea(); // here we call calcArea of
4 // an object of shape class,
5 // so shape::calcArea() is called,
6 // even if shape::calcArea is virtual.
```

- وقتی شیئی از کلاس پدر را با شیئی از کلاس فرزند مقداردهی می‌کنیم، در واقع شیء فرزند را در شیء پدر کپی می‌کنیم. با فراخوانی یک تابع بر روی شیء پدر، تابع مربوط به شیء پدر فراخوانی می‌شود.

چندریختی

- به عنوان یک مثال دیگر، فرض کنید می‌خواهیم تابعی بنویسیم که با استفاده از اطلاعات یک شکل عملیاتی را انجام می‌دهد. به عنوان مثال این تابع شکل را رسم می‌کند و اطلاعات شکل شامل مساحت آن را محاسبه می‌کند.

- بدون قابلیت چندریختی این تابع باید برای همهٔ اشکال ممکن پیاده‌سازی شود.

```
۱ void info(circle c) {
۲     c.draw();
۳     cout << c.calcArea() << ... ;
۴ }
۵ void info(rectangle r) {
۶     r.draw();
۷     cout << r.calcArea() << ... ;
۸ }
```

- علاوه بر اینکه تعداد زیادی تابع برای اشکال مختلف باید پیاده‌سازی شوند، هرگاه شکل جدیدی به مجموعهٔ اشکال اضافه شود، تابع `info` باید برای شکل جدید پیاده‌سازی شود.

چندریختی

- این مشکل را می‌توانیم با استفاده از قابلیت چندریختی حل کنیم.

```
۱ void info(shape * s) {
۲     s->draw();
۳     cout << s->calcArea() << ... ;
۴ }
```

- همهٔ رفتارهای مورد نیاز در تابع `info` می‌توانند به صورت مجازی تعریف شوند و در نتیجه با استفاده از قابلیت چندریختی یک تابع برای همهٔ اشکال مختلف پیاده‌سازی می‌شود.
- همچنین اگر در آینده یک شکل جدید به عنوان فرزند کلاس `shape` تعریف شود، از همین تابع می‌توان استفاده کرد.

چندریختی

- همچنین در استفاده از قابلیت چندریختی می‌توانیم به جای استفاده از یک اشاره‌گر به کلاس پدر، از یک متغیر مرجع استفاده کنیم.

```
۱ void info(shape & s) {
۲     s.draw();
۳     cout << s.calcArea() << ... ;
۴ }
```

چندریختی

- با استفاده از ویژگی چندریختی، همیشه در فراخوانی‌ها توابع مجازی، تابعی فراخوانی می‌شود که در سلسله مراتب وراثت به شیء مورد نظر نزدیک‌تر است. برای مثال اگر تابع `f` به صورت مجازی در کلاس `A` تعریف شود، و کلاس `B` از کلاس `A` ارث‌بری کند و تابع را تعریف کند، و کلاس `C` از کلاس `B` ارث‌بری کند، و تابع را تعریف نکند، آنگاه با فراخوانی تابع از طریق اشاره‌گری به کلاس `A` که به شیئی از کلاس `C` اشاره می‌کند، تابع پیاده‌سازی شده در کلاس `B` فراخوانی می‌شود.
- خاصیت مجازی بودن یک تابع در سلسله مراتب وراثت به ارث برده می‌شود، پس فرزندان نیازی به بازتعریف تابع به صورت مجازی را ندارند.

چندریختی

- یک تابع را می‌توان به صورت مجازی خالص [1] تعریف کرد.
- یک تابع مجازی خالص در یک کلاس پدر پیاده‌سازی نمی‌شود و فرزندان مجبور هستند آن تابع را پیاده‌سازی کنند. اگر فرزندی یک تابع مجازی خالص که در یک کلاس پدر تعریف شده است را پیاده‌سازی نکند کامپایلر پیام خطا صادر می‌کند.
- یک تابع مجازی خالص با استفاده از کلیدواژهٔ `virtual` در ابتدای امضای تابع و قرار دادن `0=` در انتهای امضای تابع تعریف می‌شود.

```
۱ class shape {
۲ public:
۳     virtual void draw() = 0; // pure virtual function
۴     // all sub-classes must implement this function.
۵ };
```

[1] pure virtual

چندریختی

- اگر یکی از توابع یک کلاس مجازی خالص تعریف شود، آن کلاس یک کلاس انتزاعی نامیده می‌شود و نمی‌توان از آن کلاس شیء ساخت، زیرا توابع مجازی خالص آن نمی‌توانند فراخوانی شوند.
- کلاس‌ها را می‌توانیم به دو دستهٔ کلاس‌های انضمامی [1]، و کلاس‌های انتزاعی [2]، تقسیم کنیم.
- انتزاع و انضمام دو مفهوم در فلسفه هستند. موجودات انتزاعی، موجوداتی ذهنی و مجرد هستند که در دنیای ماده وجود ندارند بلکه تنها در دنیای ذهن موجودند. موجودات انضمامی، موجوداتی عینی هستند که در دنیای ماده وجود دارند.
- به همین ترتیب کلاس‌های انتزاعی کلاس‌هایی هستند که یک مفهوم را پیاده‌سازی می‌کنند و کاری بر روی داده‌ها انجام نمی‌دهند. کلاس‌های انضمامی کلاس‌هایی هستند که یک موجود را پیاده‌سازی می‌کنند که بر روی داده‌ها تغییرات صورت می‌دهد.
- کلاس‌های انضمامی مانند انواع دادهٔ اصلی هستند. برای مثال کلاس `complex` که پیشتر تعریف کردیم یک کلاس انضمامی است.

[1] concrete classes
[2] abstract classes

- حال فرض کنید با استفاده از یک اشاره‌گر از کلاس پدر به یک شیء از کلاس فرزند اشاره می‌کنیم. سپس با استفاده از اشاره‌گر می‌خواهیم شیء را تخریب و فضای حافظه را آزاد کنیم.

```
۱ shape* shp;
۲ circle* crc = new circle;
۳ shp = crc;
۴ delete shp; // shape::~shape is called.
```

- در این حالت مخرب کلاس پدر فراخوانی می‌شود، در صورتی که نیاز داریم مخرب کلاس فرزند را نیز فراخوانی کنیم.

چندریختی

- بنابراین بهتر است مخرب کلاس پدر را به صورت مجازی تعریف کنیم، بدین ترتیب ابتدا در تخریب اشاره‌گر به کلاس پدر ابتدا مخرب کلاس فرزند و سپس مخرب کلاس پدر فراخوانی می‌شود.

```
class shape {
public: virtual ~shape() { }
};
shape* shp;
circle* crc = new circle;
shp = crc;
delete shp;
// first circle::~circle and then shape::~shape is called.
```

- از آنجایی که تابع مخرب مجازی است تابع مخرب فرزند فراخوانی می‌شود و هر گاه تابع مخرب فرزندی فراخوانی شود، تابع مخرب پدر نیز فراخوانی می‌شود.

چندریختی

- حال فرض کنید یک اشاره‌گر به یک کلاس پدر داریم و می‌خواهیم در صورتی که این اشاره‌گر به یکی از فرزندان خاص اشاره کرد تابعی از آن فرزند فراخوانی شود.
- با استفاده از تابع typeid می‌توانیم نوع یک شیء از یک کلاس فرزند را با استفاده از یک اشاره‌گر به کلاس پدر تعیین کنیم.

```
۱  void info(shape * s) {
۲      s->draw();
۳      cout << s->calcArea() << ... ;
۴
۵      if (typeid(*s)==typeid(circle)) {
۶          circle * c = (circle*) s;
۷          cout << c->getRadius();
۸          cout << typeid(*c).name();
۹      }
۱۰ }
```

چندریختی

- یک روش دیگر برای به دست آوردن نوع یک کلاس فرزند توسط اشاره‌گری به کلاس پدر، استفاده از `dynamic_cast` است.

- با استفاده از `dynamic_cast` می‌توانیم یک اشاره‌گر از کلاس پدر را به یک اشاره‌گر از کلاس فرزند تبدیل کنیم. اگر خروجی تابع `dynamic_cast` تهی نبود، اشاره‌گر به کلاس پدر، به شیئی از کلاس فرزند اشاره می‌کند. در مورد این تابع در آینده بیشتر توضیح خواهیم داد.

```
۱ void info(shape * s) {
۲     s->draw();
۳     cout << s->calcArea() << ... ;
۴
۵     circle * c = dynamic_cast<circle*>(s);
۶     if (c!=nullptr) {
۷         cout << c->getRadius();
۸     }
۹ }
```

چندریختی

- در صورتی که کلاس پدر توابع مجازی نداشته باشد، و در نتیجه کلاس پدر چندریخت [1] نباشد، نمی‌توان از تابع dynamic_cast استفاده کرد.

```
class shape {
public: int calcArea() { return 0; }
};
class circle : public shape {
public : int calcArea() { return pi*r*r; }
};
shape * s = new circle;
circle * c = dynamic_cast<circle*>(s);
// error : shape is not polymorphic
```

[1] polymorphic

چندریختی

- انتخاب توابع چندریخت به طور پویا در زمان اجرا صورت می‌گیرد. به عبارت دیگر تنها در زمان اجرا [1] مشخص می‌شود که یک اشاره‌گر به چه شیئی از اشیای چندریخت اشاره می‌کند.

- به طور مثال برنامه‌ای را در نظر بگیرید که در آن کاربر می‌تواند اشکالی را رسم کرده و از بین اشکال رسم شده، یک شکل را انتخاب می‌کند. حال بسته به این که چه شکلی توسط کاربر انتخاب شده است، مساحت توسط یک تابع چندریخت باید محاسبه شود. پس در جایی از برنامه داریم:

```
۱ shape * shp;
۲ if (/*user chooses circle*/) shp = new circle;
۳ else if (/*user chooses rectangle*/) shp = new rectangle;
۴ double area = shp->calcArea();
```

- در زمان کامپایل [2] مشخص نیست کاربر چه شکلی را انتخاب خواهد کرد و `shp` به چه شیئی اشاره می‌کند، پس کد ماشین تولید شده نمی‌تواند آدرس تابع `calcArea` مورد نظر را تعیین کند.

[1] run-time
[2] compile-time

چندریختی

- به دلیل این که آدرس تابع مورد نظر برای اجرا در توابع چندریخت به طور پویا در زمان اجرا تعیین می‌شود، چندریختی از سازوکاری به نام پیوستن پویا [1] استفاده می‌کند.
- در مقابل سازوکار پیوستن پویا، پیوستن ایستا [2] وجود دارد. در سربارگذاری توابع از پیوستن ایستا استفاده می‌کنیم.
- به عبارت دیگر، در سربارگذاری توابع، در زمان کامپایل، کامپایلر همهٔ اطلاعات مورد نیاز برای قرار دادن آدرس توابع سربارگذاری شده در کد ماشین را دارد.

[1] dynamic binding
[2] static binding

- فرض کنید در یک برنامه، بسته به این که کاربر می‌خواهد با استفاده از نام دانشجو یا شمارهٔ دانشجویی، اطلاعات دانشجو را بیابد، تابع سربارگذاری شدهٔ `getinfo` فراخوانی می‌شود.

```
۱ if (/*user chooses selection by name*/) {
۲     cin >> name; getinfo(name);
۳ } else if (/*user chooses selection by id*/) {
۴     cin >> id; getinfo(id);
۵ }
```

- در زمان کامپایل، کامپایلر می‌تواند دقیقا کد ماشین معادل کد بالا را تولید کرده و آدرس توابع سربارگذاری شده را جایگزین نام توابع کند. پس در زمان اجرا هیچ تصمیم‌گیری صورت نمی‌گیرد.

چندریختی

- از آنجایی که انتخاب تابع چند ریخت در زمان اجرا صورت می‌گیرد، چندریختی با استفاده از یک جدول توابع مجازی به نام vtable و یک اشاره‌گر به توابع مجازی به نام vptr پیاده‌سازی می‌شود.
- نحوهٔ پیاده‌سازی چندریختی توسط کامپایلر بدین صورت است که کامپایلر به هر کلاس چندریخت که توابع مجازی را تعریف می‌کند، یک اشاره‌گر vptr اضافه می‌کند که این اشاره‌گر به به یک جدول vtable اشاره می‌کند. در این جدول آدرس توابع مجازی که پیاده‌سازی شده‌اند قرار می‌گیرد.
- حال هر تابعی که از یک کلاس چندریخت ارث‌بری کند، طبق قوانین وراثت اشاره‌گر vptr را نیز به ارث می‌برد. اشاره‌گر در کلاس فرزند به جدولی دیگر اشاره می‌کند که در آن جدول آدرس توابع کلاس پیاده‌سازی شده در کلاس فرزند ذکر شده است. اگر تابعی چندریخت در کلاس فرزند تعریف نشده باشد، برای آن تابع آدرس تابعی قرار می‌گیرد که نزدیک‌ترین پدر آن را پیاده‌سازی کرده باشد.
- حال در زمان اجرا با استفاده از vptr کامپایلر می‌تواند تصمیم بگیرد چه توابعی را اجرا کند.

چندریختی

- برای مثال فرض کنید کلاس A توابع چندریخت f و g را تعریف کرده است. پس کلاس A یک اشاره‌گر مجازی vptr دارد که به جدول توابع مجازی vtable از کلاس A اشاره می‌کند. در این جدول آدرس پیاده‌سازی توابع f و g ذکر شده است.
- حال اگر کلاس B از کلاس A به ارث ببرد، اشاره‌گر را نیز به ارث می‌برد و اشاره‌گر vptr در کلاس B به جدولی مجازی مربوط به کلاس B اشاره می‌کند. حال اگر B هیچ کدام از توابع f و g را تعریف نکند، در جدول vtable کلاس B آدرس توابع f و g در کلاس پدر ذکر می‌شود. اما اگر B هر یک از این توابع را پیاده‌سازی کند، در جدول توابع مجازی آن، آدرس توابع پیاده‌سازی شده توسط خود کلاس B ذکر می‌شود.

چندریختی

- حال برنامه زیر را در نظر بگیرید.

```
A aobj; B bobj; A * aptr;
if (/*user chooses A*/) aptr = &aobj;
else if (/*user chooses B*/) aptr = &bobj;
aptr->f(); aptr->g();
```

- فرض کنید کلاس B تنها تابع f را پیاده‌سازی کند. کامپایلر این برنامه را به شکل زیر کامپایل خواهد کرد.

```
A aobj; B bobj; A * aptr;
// aobj.vptr and bobj.vptr are added.
// aobj.vtable contains addresses of f and g implemented in A.
// bobj.vtable contains the address of f implemented in B
// and the address of g implemented in A.
if (/*user chooses A*/) aptr = &aobj;
else if (/*user chooses B*/) aptr = &bobj;
aptr->f(); // => aptr->vptr->vtable[f]();
aptr->g(); //=> aptr->vptr->vtable[g]();
```

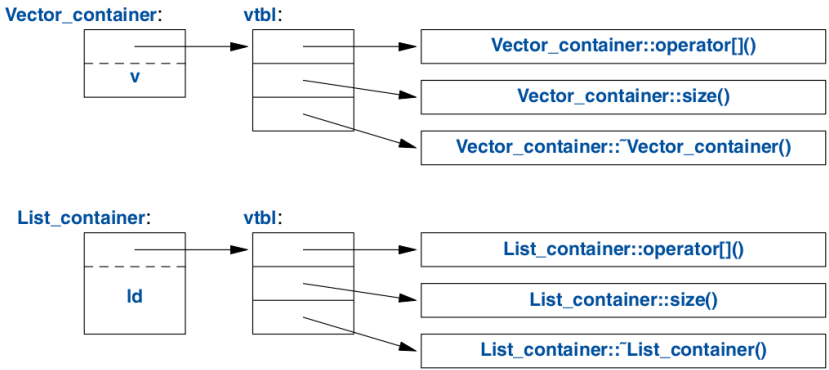
چندریختی

```
A aobj; B bobj; A * aptr;
// aobj.vptr and bobj.vptr are added.
// aobj.vtable contains addresses of f and g implemented in A.
// bobj.vtable contains the address of f implemented in B
// and the address of g implemented in A.
if (/*user chooses A*/) aptr = &aobj;
else if (/*user chooses B*/) aptr = &bobj;
aptr->f(); // => aptr->vptr->vtable[f]();
aptr->g(); //=> aptr->vptr->vtable[g]();
```

- پس در زمان اجرا بسته به اینکه vptr به چه جدولی اشاره کند و در جدول مربوطه چه آدرسی ذکر شده است، تصمیم‌گیری مبنی بر اجرای تابع چندریخت مورد نظر صورت می‌گیرد.

چندریختی

- پس اشیای چندریخت که از کلاس‌های انتزاعی به ارث برده‌اند، جدولی به نام جدول تابع مجازی [1] یا `vtbl` در حافظه نگهداری می‌کنند که در آن جدول آدرس توابعی که باید فراخوانی شوند، یادداشت شده است.



[1] virtual function table

چندریختی

- رابط کلاس [1] در واقع تعریف کلاس است که در فایل سرتیتر یا فایل هِدِر [2] آمده است.
- اما کلاس رابط [3] یک کلاس انتزاعی است که برخی توابع آن به صورت مجازی خالص تعریف شده و باید در یک زیرکلاس آن رفتارها را پیاده‌سازی کرد.

[1] class's interface or interface of the class
[2] header file
[3] interface class

مثال

- فرض کنید می‌خواهیم از مفهوم وکتور یا حامل [1] در برنامه‌های خود استفاده کنیم. یک حامل در واقع ظرفی [2] است برای نگهداری تعدادی عنصر [3]. این عناصر می‌توانند از نوع داده‌های مختلفی باشند ولی فرض کنید می‌خواهیم حامل ما شامل اعداد اعشاری `double` باشد.
- یک حامل باید یک سازنده و یک مخرب داشته باشد، یک لیست از عناصر و یک اندازه داشته باشد، و همچنین امکان دسترسی به عناصر خود را فراهم کند.

[1] vector
[2] container
[3] element

مثال

```
 1  class Vector {
 2  public:
 3      // constructor: acquire resources
 4      Vector(int s) :elem{new double[s]}, sz{s}, cur{0} {
 5          for (int i=0; i!=s; ++i) // initialize elements
 6              elem[i]=0;
 7      }
 8      // destructor: release resources
 9      ~Vector() { delete[] elem; }
10
11      double& operator[](int i);
12      int size() const;
13  private:
14      // elem points to an array of sz doubles
15      double* elem;
16      int sz;
17      int cur;
18  };
```

مثال

- تعریف تابع `size()` با کلیدواژهٔ `const` باعث می‌شود در صورتی که تابع عضوی از کلاس را تغییر دهد، کامپایلر پیام خطا صادر کند.
- هنگامی که شیئی از این کلاس ساخته می‌شود، سازنده فراخوانی شده و فضایی در حافظه برای عناصر تخصیص داده می‌شود و هنگامی که آن شیء تخریب می‌شود، مخرب فراخوانی شده و فضا آزاد می‌شود.

```
۱ void fct(int n) {
۲     Vector v(n);
۳     // ... use v ...
۴     {
۵         Vector v2(2*n);
۶         // ... use v and v2 ...
۷     } // v2 is destroyed here
۸     // ... use v ..
۹ } // v is destroyed here
```

مثال

- می‌توانیم برای این کلاس تابعی تعریف کنیم که یک عنصر به وکتور اضافه کند. فرض کنید این تابع را بدین صورت تعریف می‌کنیم:

```
۱  void push_back(double d) {
۲          elem[cur] =  d;
۳          cur++;
۴          if (cur==sz) {
۵              double * tmp = new double[2*sz];
۶              std::memcpy(tmp, elem, sz*sizeof(double));
۷              delete[] elem;
۸              elem = tmp;
۹              sz *= 2;
۱۰         }
۱۱     }
```

- در واقع وکتور با یک طول ثابت تعریف می‌شود، و هرگاه تعداد عناصر مورد نیاز از فضای تخصیص داده شده توسط وکتور بیشتر شود، وکتور اندازهٔ فضای تخصیص داده شده را افزایش می‌دهد.

مثال

- همچنین می‌توانیم سازنده‌ای تعریف کنیم که وکتور را با دریافت تعدادی عنصر در یک لیست بسازد.

```
// initialize with a list
Vector(std::initializer_list<double> lst) {
    elem = new double[lst.size()];
    sz = lst.size();
    cur = sz;
    std::copy(lst.begin(),lst.end(),elem);
}
```

- از این پس می‌توان وکتور را اینچنین مقداردهی اولیه کرد:

```
Vector v1 = {1,2,3,4,5};
Vector v2 {1.23, 3.45, 6.7, 8};
```

مثال

- یک کلاس انتزاعی کلاسی است که تنها یک مفهوم را تعریف می‌کند، ولی هیچ عملیاتی بر روی داده‌ها انجام نمی‌دهد. از یک کلاس انضمامی می‌توان یک شیء ساخت به طوری که شیء ساخته شده بر روی داده‌ها تغییرات اعمال می‌کند.
- یک وکتور یک مفهوم انضمامی است و با یک کلاس انضمامی تعریف می‌شود، و بر روی داده‌ها عملیات انجام می‌دهد. اما یک ظرف [1] یک مفهوم انتزاعی است. می‌دانیم یک وکتور یک ظرف است و همچنین یک صف [2] هم یک ظرف است. ظرف خصوصیاتی دارد که می‌توان این خصوصیات را توسط یک کلاس انتزاعی تعریف کرد.
- یک ظرف همیشه یک اندازه دارد و می‌توان به عناصر آن توسط عملگر [] دسترسی پیدا کرد.

[1] container
[2] queue

مثال

- پس یک ظرف چنین تعریف می‌شود.

```
1 class Container {
2 public:
3     virtual double& operator[](int) = 0;
4     virtual int size() const = 0;
5     virtual ~Container() {}
6 };
```

- واژهٔ `virtual` بدین معناست که تابع مجازی است و ممکن است بعدها در کلاسی دیگر که از این کلاس ارث برده است پیاده‌سازی شود.
- وقتی یک تابع مجازی برابر با صفر قرار داده می‌شود، این بدین معناست که این تابع یک تابع مجازی خالص [1] است و حتما باید توسط کلاس‌هایی که از این کلاس ارث برده‌اند پیاده‌سازی شود.

[1] pure virtual

مثال

- از یک کلاس انتزاعی نمی‌توان شیئی ساخت، اما از یک کلاس انضمامی که از یک کلاس انتزاعی ارث برده است، می‌توان شیء ساخت و استفاده کرد.

```
Container c;
// error : there can be no objects of an abstract class
Container* p = new Vector_container(10);
// OK: Container is an interface
```

- با استفاده از قابلیت چندریختی، از یک ظرف Container می‌توانیم به صورت زیر استفاده کنیم. این تابع را می‌توان توسط یک وکتور یا یک صف فراخوانی کرد.

```
void use(Container& c) {
    const int sz = c.size();
    for (int i=0; i!=sz; ++i)
        cout << c[i] << '\n';
}
```

مثال

- حال برای اینکه بتوانیم از ظرف استفاده کنیم باید آن را پیاده‌سازی کنیم. می‌توانیم یک ظرف وکتور پیاده‌سازی کنیم که از ظرف به ارث می‌برد.

```
۱  class Vector_container : public Container {
۲  // Vector_container implements Container
۳  public:
۴      Vector_container(int s) : v(s) { }
۵      // Vector of s elements
۶      ~Vector_container() {}
۷      double& operator[](int i) override { return v[i]; }
۸      int size() const override { return v.size(); }
۹  private:
۱۰     Vector v;
۱۱ };
```

مثال

- کلیدواژهٔ `public`: بدین معناست که `Vector_container` از `Container` به ارث می‌برد و به عبارت دیگر وکتور یکی از انواع ظرف است.
- ظرف وکتور یک زیرکلاس [1] از کلاس ظرف است و کلاس ظرف، کلاس مافوق یا ابرکلاس [2] ظرف وکتور است.
- وقتی یک کلاس از یک کلاس دیگر مشتق می‌شود، از رفتارهای آن کلاس استفاده می‌کند و بدین دلیل می‌گوییم کلاسی از کلاس دیگر به ارث برده است.

[1] subclass
[2] superclass

مثال

- کلاس شکل را اکنون با توابع بیشتری پیاده‌سازی می‌کنیم:

```
۱  class Shape {
۲  public:
۳      virtual Point center() const =0;
۴      virtual void move(Point to) =0; // pure virtual
۵
۶      virtual void draw() const = 0; // draw the shape
۷      virtual void rotate(int angle) = 0;
۸      virtual ~Shape() {} // destructor
۹      // ...
۱۰ };
```

- هر شکلی یک نقطهٔ مرکز خواهد داشت که توسط تابع `center` به دست می‌آید و هر شکل را می‌توان جابجا کرد و نقطهٔ مرکز آن را توسط تابع `move` تغییر داد. همچنین هر شکل را می‌توان توسط تابع `draw` رسم کرد و توسط تابع `rotate` به ازای یک درجهٔ معین دوران داد.

مثال

- حال با استفاده از ویژگی چندریختی می‌توان یک لیست از اشکال مختلف که همگی از کلاس شیء ارث‌بری کرده‌اند، دریافت کرده و دوران داد.

```
۱ // rotate 'vs elements by angle degrees
۲ void rotate_all(vector<Shape*>& v, int angle) {
۳     for (auto p : v)
۴         p->rotate(angle);
۵ }
```

مثال

- توابع مجازی را می‌توان در یک کلاس انضمامی مانند کلاس دایره که از کلاس انتزاعی شکل ارث‌بری کرده است، تعریف کرد.

```
۱  class Circle : public Shape {
۲  public:
۳      Circle(Point p, int rad); // constructor
۴      Point center() const override { return x; }
۵      void move(Point to) override { x = to; }
۶      void draw() const override;
۷      void rotate(int) override {} // nice simple algorithm
۸  private:
۹      Point x; // center
۱۰     int r; // radius
۱۱ };
```

- توابعی که تابع کلاس پدر را لغو و جایگزین می‌کنند را با کلیدواژهٔ override متمایز می‌کنیم. این کلیدواژه برای خوانایی بهتر کد به کار می‌رود.

طراحی شیءگرا

- معمولا قبل از پیاده‌سازی یک سیستم نرم‌افزاری طرحی برای آن آماده می‌کنیم. این طرح می‌تواند شامل مستندات و اشکال و فلوچارت‌هایی باشد که نحوهٔ پیاده‌سازی سیستم را توصیف می‌کند.
- با طراحی یک سیستم قبل از پیاده‌سازی اول اینکه از هزینه‌ای که ممکن است به دلیل خطاهای پیاده‌سازی تحمیل شود می‌کاهیم، دوم اینکه می‌توانیم تیم طراحی را از تیم پیاده‌سازی جدا کنیم، سوم اینکه می‌توانیم مستندات طراحی را نگهداری کنیم و امکان تغییرات سیستم در آینده را بهتر فراهم کنیم، و چهارم اینکه می‌توانیم بعد از پیاده‌سازی سیستم بررسی کنیم که آیا همهٔ خواسته‌ها بر طبق نیازها برآورده شده‌اند یا خیر.
- زبان طراحی یکپارچه [1] یا یوام‌ال زبانی است استاندارد که برای مستند کردن و توصیف کردن یک سیستم نرم‌افزاری استفاده می‌شود. گروه مدیریت اشیا [2] این زبان را در سال ۱۹۹۷ برای طراحی سیستم‌های پیچیدهٔ نرم‌افزاری ارائه کرده است.

[1] Unified Modeling Language (UML)
[2] Object Management Group (OMG)

طراحی شیءگرا

- زبان مدلسازی یکپارچه تعدادی نمودار استاندارد برای طراحی سیستم ارائه می‌کند. این نمودارها شامل نمودارهای ساختاری [1] و نمودارهای رفتاری [2] می‌شوند.
- نمودارهای ساختاری شامل نمودار کلاس [3]، نمودار اشیا [4]، نمودار اجزا [5]، و نمودار استقرار [6] می‌شوند.
- نمودارهای رفتاری شامل نمودار فعالیت [7]، نمودار توالی [8]، نمودار کارخواست [9]، و نمودار حالت [10] می‌شوند.

---

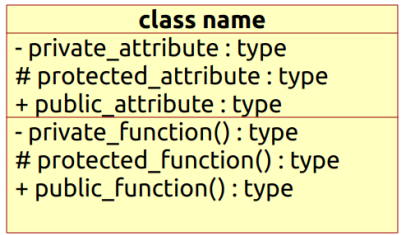
[1] structural diagrams
[2] behavioral diagrams
[3] class diagram
[4] object diagram
[5] component diagram
[6] deployment diagram
[7] activity diagram
[8] sequence diagram
[9] use case diagram
[10] state diagram

طراحی شیءگرا

- در مبحث طراحی شیءگرای سیستم‌ها همهٔ این نمودارها شرح داده می‌شوند. در اینجا تنها به نمودار کلاس می‌پردازیم.
- وقتی یک سیستم را توصیف می‌کنیم، معمولا برای هر اسم و هر مفهومی یک کلاس می‌سازیم. بنابراین وقتی سیستم مورد نظر خود را توسط مستندات توصیف کردیم، هر اسمی می‌تواند یک کلاس در سیستم باشد.

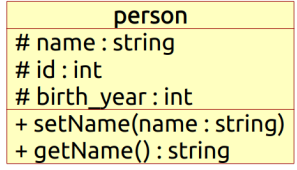
طراحی شیءگرا

- یک کلاس توسط یک مستطیل نشان داده می‌شود که در بالای آن نام کلاس نوشته می‌شود، در قسمت میانی اعضای داده‌ای کلاس، و در قسمت پایینی رفتار کلاس ذکر می‌شوند.
- اعضای عموی با علامت +، اعضای خصوصی با علامت - و اعضای حفاظت‌شده با علامت # مشخص می‌شوند.



طراحی شیءگرا

- برای مثال برای کلاس شخص (person) نمودار زیر را رسم می‌کنیم. تنها تعداد اندکی از ویژگی‌ها و رفتارها نشان داده شده‌اند.



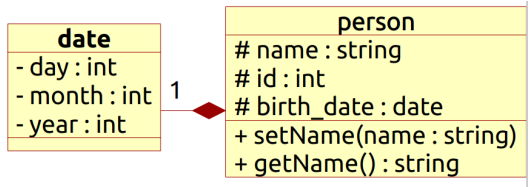
طراحی شیءگرا

- چندین نوع رابطه برای توصیف رابطهٔ بین کلاس‌ها وجود دارد. یکی از این رابطه‌ها رابطهٔ ترکیب [1] و یکی دیگر رابطهٔ تجمیع [2] می‌باشند.
- در هر دو رابطهٔ ترکیب و تجمیع می‌گوییم یک کلاس دارای عضوی یا اعضایی از کلاس دیگر است. به طور مثال یک شخص یک تاریخ تولد از کلاس تاریخ دارد یا یک شخص یک مسکن از کلاس مسکن دارد.
- تفاوت ترکیب و تجمیع در این است که در رابطهٔ ترکیب شیئی از کلاس تحت مالکیت (کلاس متعلق) نمی‌تواند بدون شیئی از کلاس مالک وجود داشته باشد. به طور مثال یک تاریخ تولد نمی‌تواند بدون وجود یک شخص وجود داشته باشد. اما یک مسکن می‌تواند بدون وجود یک شخص وجود داشته باشد و همچنین از شخصی به شخص دیگر منتقل شود.

[1] composition relation
[2] aggregation relation

طراحی شیءگرا

- رابطهٔ ترکیب به صورت زیر نمایش داده می‌شود. وقتی کلاس A یک عضو از کلاس B دارد، در طرف کلاس A از یک لوزی توپر [1] استفاده می‌کنیم.
- همچنین کثرت [2] یک رابطه را می‌توانیم در یک طرف یا در دو طرف آن نشان دهیم، یعنی بگوییم کلاس A چند عضو از کلاس B دارد.
- مثلا کلاس شخص یک عضو از کلاس تاریخ دارد که تاریخ تولد او را نشان می‌دهد.

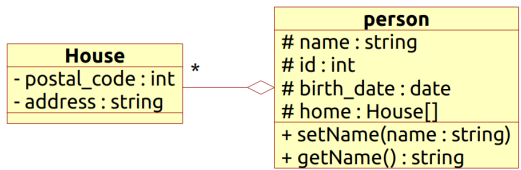


[1] filled diamond
[2] multiplicity

طراحی شیءگرا

- رابطهٔ تجمیع به صورت زیر نمایش داده می‌شود. وقتی کلاس A عضوی دارد که به یک شیء از کلاس B اشاره می‌کند، در طرف کلاس A از یک لوزی توخالی [1] استفاده می‌کنیم. در برنامه‌سازی در این موارد از اشاره‌گر استفاده می‌کنیم. همچنین کثرت [2] یک رابطه را می‌توانیم در یک طرف یا در دو طرف آن نشان دهیم، یعنی بگوییم کلاس A چند عضو از کلاس B دارد.

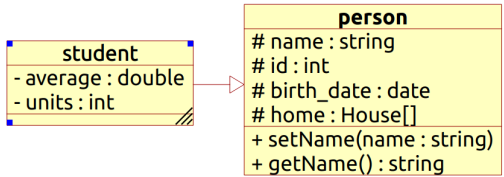
- برای مثال یک شخص می‌تواند یک منزل از کلاس خانه داشته باشد ولی منزل یک شخص قابل انتقال است و بدون وجود شخص نیز وجود دارد، پس این رابطه یک رابطهٔ تجمیع است. همچنین یک شخص می‌تواند چند خانه داشته باشد.



[1] empty diamond
[2] multiplicity

طراحی شیءگرا

- یکی دیگر از رابطه‌های بین دو کلاس، رابطهٔ تعمیم یا رابطهٔ وراثت [1] است.
- رابطهٔ وراثت را توسط یک خط بین پدر و فرزند و یک مثلث توخالی [2] در سمت پدر نشان می‌دهیم.



[1] generalization relation or inheritance relation
[2] empty triangle