

به نام خدا

برنامه‌سازی پیشرفته

آرش شفيعی



- برنامه‌سازی: اصول و شیوه‌ها با استفاده از سی‌پلاس‌پلاس، از بیارنه استراستروپ¹
- سیاحتی در سی‌پلاس‌پلاس، از بیارنه استراستروپ²
- زبان برنامه‌سازی سی‌پلاس‌پلاس، از بیارنه استراستروپ³
- مرجع کامل سی‌پلاس‌پلاس⁴

¹ Programming: Principles and Practice Using C++, by Bjarne Stroustrup

² A Tour of C++, by Bjarne Stroustrup

³ The C++ Programming Language, by Bjarne Stroustrup

⁴ www.cppreference.com

مروری بر مبانی برنامه‌سازی

- سیستم عامل یونیکس برای اولین بار بر روی یک کامپیوتر PDP۷ با استفاده از زبان اسمبلی توسط دنیس ریچی^۱ و کن تامسون^۲ در آزمایشگاه‌های بل^۳ طراحی و پیاده‌سازی شد.
- یونیکس در نسخه بعدی برای یک کامپیوتر PDP۱۱ پیاده‌سازی شد و از آنجایی که برای کامپیوتر جدید به تعدادی ابزار نیاز بود، طراحان آن تصمیم گرفتند کامپایلری برای یک زبان سطح بالا طراحی کنند تا ابزارها را بتوان با استفاده از آن زبان سطح بالا راحت‌تر پیاده‌سازی کرد. در آن زمان زبان BCPL طراحی شده بود. طراحان یونیکس با استفاده از ایده‌های این زبان، و همچنین زبان ALGOL کامپایلری برای یک زبان جدید طراحی و پیاده‌سازی کردند و زبان جدید را B نامیدند.
- بین سال‌های ۱۹۷۱ و ۱۹۷۲ به تدریج امکاناتی به زبان B اضافه شد و در نتیجه زبان جدیدی به وجود آمد که بعدها زبان C نامیده شد. در سال ۱۹۷۸ اولین نسخه از کتاب زبان برنامه‌سازی سی^۴ منتشر شد.

^۱ Dennis Ritchie

^۲ Ken Thompson

^۳ AT&T Bell Laboratories

^۴ The C Programming Language

مبنای اعداد

- تبدیل اعداد دهدهی¹ به دودویی²: $(x)_{10} = (a_n a_{n-1} \dots a_1 a_0)_2$ به طوری که $x = \sum_{i=0}^n a_i \times 2^i$ و $a_i \in \{0, 1\}$

- مثال: $(42)_{10} = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$
 $(42)_{10} = 32 + 8 + 2 = (101010)_2$

- اعداد دهدهی می‌توانند علاوه بر قسمت صحیح³ قسمت اعشاری⁴ نیز داشته باشند.

- تبدیل اعداد دهدهی اعشاری به دودویی: $(0.x)_{10} = (0.a_1 a_2 \dots a_{n-1} a_n)_2$ به طوری که $0.x = \sum_{i=1}^n a_i \times 2^{-i}$ و $a_i \in \{0, 1\}$

- مثال: $(0.75)_{10} = 1 \times 2^{-1} + 1 \times 2^{-2}$
 $(0.75)_{10} = 0.5 + 0.25 = (0.11)_2$

¹ decimal

² binary

³ integer part

⁴ fractional part

- روش تبدیل اعداد دهدهی اعشاری به دودویی: عدد دهدهی را n بار در ۲ ضرب می‌کنیم تا یا عدد به دست آمده قسمت اعشاری نداشته باشد و یا n از تعداد ارقام اعشاری مورد نیاز در عدد دودویی بیشتر شود. سپس عدد دهدهی بدون قسمت اعشاری را به دودویی تبدیل می‌کنیم و در عدد دودویی به دست آمده n رقم از سمت راست جدا می‌کنیم و ممیز اعشار را بعد از n رقم قرار می‌دهیم (در واقع عدد دودویی به دست آمده را n بار بر ۲ تقسیم می‌کنیم).

- مثال: معادل عدد دهدهی $(۴.۷۵)_{۱۰}$ را در مبنای دو را محاسبه کنید.

$$\begin{aligned} ۴.۷۵ \times ۲ \times ۲ &= ۱۹ \\ (۱۹)_{۱۰} &= (۱۰۰۱۱)_۲ \\ (۱۰۰۱۱)_۲ \div ۲ \div ۲ &= (۱۰۰.۱۱)_۲ \\ (۴.۷۵)_{۱۰} &= (۱۰۰.۱۱)_۲ \end{aligned}$$

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

$$- ۰.۳ \times ۲^{۱۴} = ۴۹۱۵.۲$$

$$(۴۹۱۵)_{۱۰} = (۱۰۰۱۱۰۰۱۱۰۰۱۱)_{۲}$$

$$(۱۰۰۱۱۰۰۱۱۰۰۱۱)_{۲} \div ۲^{۱۴} = (۰.۰۱۰۰۱۱۰۰۱۱۰۰۱۱)_{۲}$$

$$(۰.۳)_{۱۰} = (۰.۰۱۰۰۱۱۰۰۱۱۰۰۱۱)_{۲}$$

- روش تبدیل اعداد دودویی اعشاری به دهدهی: عدد دودویی را n بار در ۲ ضرب می‌کنیم تا عدد به دست آمده قسمت اعشاری نداشته باشد. سپس عدد دودویی بدون قسمت اعشاری را به دهدهی تبدیل می‌کنیم و عدد دهدهی به دست آمده را n بار بر ۲ تقسیم می‌کنیم.

- مثال: عدد دودویی $(100.11)_2$ را به دهدهی تبدیل کنید.

$$(100.11)_2 \times 2 \times 2 = (10011)_2$$

$$(10011)_2 = (19)_{10}$$

$$19 \div 2 \div 2 = 4.75$$

$$(100.11)_2 = (4.75)_{10}$$

- یک عدد دودویی را می‌توانیم به صورت یک عدد علامت‌دار¹ یا یک عدد بدون علامت² تعبیر کنیم.
- اولین بیت (رقم) از سمت چپ یک عدد را برای نشان دادن علامت آن عدد استفاده می‌کنیم و آن را بیت علامت³ می‌گوییم.
- اگر بیت علامت برابر با ۱ باشد عدد منفی است و اگر بیت علامت برابر با صفر باشد عدد مثبت است.

¹ signed

² unsigned

³ sign bit

- برای تبدیل یک عدد دودویی علامتدار $(b)_2$ با بیت علامت ۱ به مبنای دهدهی ابتدا مکمل دو b را محاسبه می‌کنیم. فرض کنیم عدد به دست آمده عدد c است. حال عدد c را به مبنای ده تبدیل می‌کنیم.
 $(c)_2 = (d)_{10}$
- عدد دودویی b یک عدد منفی است که در مبنای ده برابر است با $-d$ بنابراین $(b)_2 = (-d)_{10}$
- برای محاسبهٔ مکمل دو^۱ یک عدد صفرها را به یک و یک‌ها را به صفر تبدیل کرده، سپس یک واحد به آن عدد می‌افزاییم.
- مثال: عدد بدون علامت $(1001)_2$ در مبنای ده برابر است با ۹.
- اما عدد علامتدار $(1001)_2$ در مبنای ده برابر است با -۷ .

¹ two's complement

- برای تبدیل یک عدد منفی دهدهی به یک عدد منفی دودویی، ابتدا آن عدد را به صورت مثبت در نظر گرفته، آن را به مبنای دو تبدیل کرده، سپس مکمل دو آن را محاسبه می‌کنیم.
- مثال: معادل عدد $42 -$ را در مبنای دو محاسبه کنید.

$$- (42)_{10} = (0101010)_2$$

$$- (-42)_{10} = (1010110)_2$$

- اعداد دودویی را می‌توانیم با استفاده از روش زیر به اعداد پایه شانزده (شانزده شانزدهی یا هگزادسیمال¹) تبدیل کنیم.
- عدد دودویی را از سمت راست چهار بیت چهار بیت جدا می‌کنیم و معادل هگزادسیمال هر چهاربیت را از سمت راست می‌نویسیم. اعداد چهاربیتی می‌توانند بیت ۰ تا ۱۵ باشند. در مبنای شانزده، عدد ۱۰ را با A، ۱۱ را با B، ۱۲ را با C، ۱۳ را با D، ۱۴ را با E، و ۱۵ را با F نشان می‌دهیم.
- مثال: معادل عدد ۴۲ را در مبنای شانزده محاسبه کنید.
- $(42)_{10} = (101010)_2 = (2A)_{16}$

¹ hexadecimal

- یک متن که به زبان سی نوشته شده است را یک برنامه سی¹ می‌نامیم.
- یک برنامه سی در یک فایل سی ذخیره می‌شود و یک فایل سی توسط کامپایلر² سی به فایل آبجکت³ تبدیل می‌شود. محتوای یک فایل آبجکت، برنامه مورد نظر به زبان ماشین مقصد در قالب یک فایل دودویی است. فایل‌های آبجکت توسط یک پیونددهنده یا لینکر⁴ به یکدیگر پیوند داده می‌شوند و یک فایل اجرایی تولید می‌شود. فایل اجرایی، برنامه مورد نظر را اجرا می‌کند.

¹ C program

² compiler

³ object file

⁴ linker

- اجرای برنامه سی از تابع بدنه main آغاز می شود. قبل از تابع بدنه کتابخانه های مورد نیاز برای دسترسی به توابع کتابخانه ای معرفی می شوند.

```
۱ #include <stdio.h> // introduce library to use
۲ int main() {
۳     // code (program instructions)
۴     return 0;
۵ }
```

- برای چاپ کردن یک رشته بر روی خروجی استاندارد از تابع `printf` استفاده می‌کنیم.

```
۱ #include <stdio.h>
۲ int printf ( const char * format, ... );
```

- ورودی اول تابع، رشته‌ای است که در خروجی استاندارد چاپ می‌شود. این رشته می‌تواند شامل زیررشته‌هایی باشد که نحوه نمایش (فرمت^۱) خروجی را تعیین می‌کنند. این زیررشته‌ها را تعیین‌کننده فرمت^۲ می‌نامیم. تعیین‌کننده‌های فرمت با علامت % شروع می‌شوند. این تعیین‌کننده‌های فرمت با ورودی‌های بعدی تابع، که شامل اعداد و رشته‌ها هستند، جایگزین می‌شوند و اعداد و رشته‌ها را با فرمت تعیین شده در خروجی استاندارد چاپ می‌کنند. یک تعیین‌کننده فرمت می‌تواند %c برای چاپ کاراکتر، %d برای چاپ اعداد صحیح دهدهی، %f برای چاپ اعداد اعشاری، %x برای چاپ اعداد در مبنای شانزده، %p برای چاپ آدرس اشاره‌گر و یا %s برای چاپ رشته‌ها باشد.
- این تابع تعداد کاراکترهای نوشته شده را در صورت موفقیت بازمی‌گرداند و در غیراینصورت یک عدد منفی بازمی‌گرداند.

^۱ format

^۲ format specifier

- تعیین‌کننده فرمت در حالت کلی به صورت

`%[flag] [width] [.precision] [length] specifier`

است.

- برای مثال در `%010ld` مقدار پرچم ¹ است که بدین معنی است که جاهای خالی سمت چپ عدد صحیحی که برای چاپ شدن تعیین شده با صفر پر می‌شوند. مقدار عرض چاپ ² ۱۰ است، که بدین معنی است که عدد صحیح در یک فضای ۱۰ کاراکتری باید چاپ شود. 1 به معنی این است که عدد صحیح مورد نظر long است.
- برای اعداد اعشاری می‌توانیم داشته باشیم `%10.2f` که بدین معنی است که عدد اعشاری در یک فضای ۱۰ کاراکتری چاپ می‌شود و دقت ³ آن ۲ است، یعنی تنها دو رقم بعد از اعشار چاپ می‌شود.

¹ flag

² width

³ precision


```
۱  /* printf example */
۲  #include <stdio.h>
۳
۴  int main()
۵  {
۶      printf ("Characters: %c %c \n", 'a', 65);
۷      printf ("Decimals: %d %ld\n", 1977, 650000L);
۸      printf ("Preceding with blanks: %10d \n", 1977);
۹      printf ("Preceding with zeros: %010d \n", 1977);
۱۰     printf ("Some different radices: %d %x %#x \n", 100, 100, 100);
۱۱     printf ("floats: %4.2f %E \n", 3.1416, 3.1416);
۱۲     printf ("%s \n", "A string");
۱۳     return 0;
۱۴ }
```

خروجی این برنامه به صورت زیر است:

```
۱ Characters : a A
۲ Decimals : 1977 650000
۳ Preceding with blanks :      1977
۴ Preceding with zeros : 0000001977
۵ Some different radices : 100 64 0x64
۶ floats : 3.14 3.141600E+00
۷ A string
```

- برای دریافت ورودی از روی ورودی استاندارد از تابع `scanf` استفاده می‌کنیم.

```
۱ #include <stdio.h>  
۲ int scanf ( const char * format, ... );
```

- ورودی اول تابع، فرمت رشته‌ای است که از ورودی استاندارد دریافت می‌شود. ورودی‌های بعدی متغیرهایی هستند که اعداد و رشته‌های دریافت شده در آنها ذخیره می‌شوند.

- برای مثال `scanf("%2d / %4d %s", &m, &y, s)` یک عدد دو رقمی را دریافت کرده در متغیر `m` ذخیره می‌کند، سپس یک علامت `/` دریافت می‌کند، سپس یک عدد چهار رقمی دریافت کرده در متغیر `y` ذخیره می‌کند، و باقیمانده را در متغیر رشته `s` ذخیره می‌کند.

- هر متغیر در زبان سی دارای یک نوع داده¹ است که به کامپایلر اجازه می‌دهد داده قرار گرفته در آن متغیر را تفسیر کند. هر نوع داده اندازه معینی دارد که با بیت اندازه‌گیری می‌شود.
- یک متغیر را در زبان سی با نوع آن تعریف می‌کنیم: `type var;`
- انواع داده را می‌توان در سه دسته طبقه‌بندی کرد: انواع داده اصلی²، انواع داده تعریف شده توسط کاربر³، و انواع داده مشتق شده⁴.

¹ data type

² primitive

³ user-defined

⁴ derived

انواع داده اصلی

انواع داده اصلی عبارتند از:

نوع داده	کاربرد	اندازه (بایت)
char	حرف (کاراکتر)	۱
short int	اعداد صحیح کوچک	۲
int	اعداد صحیح	۴
long int	اعداد صحیح بزرگ	۴ یا ۸ (بسته به معماری)
float	اعداد اعشاری (تا ۷ رقم اعشار)	۴
double	اعداد اعشاری (تا ۱۵ رقم اعشار)	۸

- هر یک از این انواع داده می‌توانند به صورت علامت‌دار (signed) یا بدون علامت (unsigned) تعریف شوند. در صورتی که داده‌ای یک بایتی به صورت بدون علامت تعریف شود، در آن مقادیر ۰ تا ۲۵۵ قرار می‌گیرند و اگر داده‌ای یک بایتی علامت‌دار تعریف شود، در آن مقادیر ۱۲۸- تا ۱۲۷ قرار می‌گیرند.
- برای اندازه‌گیری اندازه متغیر می‌توانیم از تابع `sizeof` نیز استفاده کنیم. همچنین در کتابخانه `limit.h` متغیرهای `INT_MAX`، `INT_MIN`، `CHAR_MAX`، `CHAR_MIN`، و غیره برای مقدار کمینه و بیشینه هر نوع داده تعریف شده‌اند.

- برای مثال می‌خواهیم عددی صحیح را از ورودی دریافت کنیم و در صورتی که عدد مورد نظر در محدوده اعداد صحیح نبود پیام خطا صادر کنیم.

```
۱ long int x;  
۲ int y = 0;  
۳ scanf("%ld", &x);  
۴ if (x < INT_MAX && x > INT_MIN) {  
۵     y = x;  
۶ } else {  
۷     printf("Integer number is too large.\n");  
۸ }
```

انواع داده اصلی

- بر روی متغیرها می‌توانیم انواع عملگرهای محاسباتی¹، رابطه‌ای²، شرطی³، منطقی⁴، و بیتی⁵ را اعمال کنیم.
- عملگرهای محاسباتی +, -, *, /, ++, +=, --, -=
- عملگرهای رابطه‌ای <, >, ==, <=, >=
- عملگرهای شرطی : ?
- عملگرهای منطقی && (and), || (or), ! (not)
- عملگرهای بیتی
& (and), | (or), ^ (xor) ~(not), << (left shift), >> (right shift)

¹ mathematical

² relational

³ conditional

⁴ logical

⁵ bitwise

انواع داده اصلی

- در استفاده از عملگرها باید به اولویت یا تقدم¹ آنها توجه کرد.
- برای مثال اولویت ++ (به صورت پسوند) از * (مقدارگیری اشاره‌گر یا رفع ارجاع²) بیشتر است. بنابراین
$$*p++ == (*(p++))$$
- اگر دو عملگر اولویت یکسان داشته باشند، باید توجه کنیم آیا مقدار آنها از چپ به راست محاسبه می‌شود و یا از راست به چپ.
- برای مثال اولویت += با اولویت -= یکسان است. اما این دو عملگر از راست به چپ محاسبه می‌شوند.
بنابراین
$$a += b \quad -= \quad c \quad == \quad a \quad += \quad (\quad b \quad -= \quad c)$$

¹ precedence

² dereference

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	()	Function call	
	[]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(type){list}	Compound literal(C99)	
2	++ --	Prefix increment and decrement ^[note 1]	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(type)	Cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of ^[note 2]	
	_Alignof	Alignment requirement(C11)	

انواع داده اصلی

3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10		Bitwise OR (inclusive or)	
11	&&	Logical AND	
12		Logical OR	
13	?:	Ternary conditional ^[note 3]	Right-to-left
14 ^[note 4]	=	Simple assignment	
	+= -=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
	&= ^= =	Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

انواع داده تعریف شده توسط کاربر

– انواع داده تعریف شده توسط کاربر¹ عبارتند از تعریفی (typedef)، ساختمان (struct)، اجتماع (union)، شمارشی (enum).

¹ user-defined data types

انواع داده تعریف شده توسط کاربر

- از نوع تعریفی (typedef) برای تعریف یک نوع داده جدید بر اساس نوع داده‌های از پیش تعریف شده یا نوع داده‌های اصلی استفاده می‌کنیم. برای مثال می‌توانیم یک نام کوتاه برای یک نوع داده تعریف کنیم:

```
۱ typedef unsigned long long int ullint;  
۲ ullint i;  
۳ typedef ullint ull;  
۴ ull j;
```

همچنین می‌توانیم برای مثال یک رشته به طور ثابت تعریف کنیم:

```
۱ typedef char string[32];  
۲ string s;
```

انواع داده تعریف شده توسط کاربر

- نوع داده ساختمان (struct) یک نوع داده مرکب¹ است که برای تعریف مجموعه‌ای از متغیرها با انواع متفاوت در یک گروه با یک نام واحد در حافظه به کار می‌رود.

```
۱ struct student {  
۲     char name[32];  
۳     int age;  
۴     float average;  
۵ };  
۶ struct student st;  
۷ typedef struct student Student;  
۸ Student stu;  
۹ strcpy(stu.name, "Ali");  
۱۰ stu.age = 20; stu.average = 17.5;
```

¹ composite data type

انواع داده تعریف شده توسط کاربر

- با تعریف یک متغیر نوع داده Student در حافظه بلوکی با ۴۰ بایت تخصیص داده می شود. ۳۲ بایت برای نام دانشجو، ۴ بایت برای سن از نوع عدد صحیح، و ۴ بایت برای معدل از نوع عدد اعشاری. البته به دلیل دسترسی کارآمدتر پردازنده به متغیرها، گاهی در ساختمان ها تعدادی بایت توسط کامپایلر اضافه شده، و همیشه طول ساختمان دقیقاً برابر با مقدار محاسبه شده نیست.

```
۱ struct student {  
۲     char name[32];  
۳     int age;  
۴     float average;  
۵ };  
۶ struct student st;  
۷ typedef struct student Student;  
۸ Student stu;  
۹ strcpy(stu.name, "Ali");  
۱۰ stu.age = 20; stu.average = 17.5;
```

انواع داده تعریف شده توسط کاربر

- نوع داده اجتماع (union) نیز شبیه ساختمان یک نوع داده مرکب است. با این تفاوت که مقدار حافظه‌ای که برای یک متغیر از نوع اجتماع تخصیص داده می‌شود، برابر با متغیری از آن اجتماع است که بیشترین اندازه را دارد.

```
۱ union student {  
۲     char name[32];  
۳     int age;  
۴ };  
۵ union student st;
```

- برای مثال برای یک متغیر از نوع student در مثال بالا ۳۲ بایت در حافظه تخصیص داده می‌شود. با دسترسی به متغیر age تنها از ۴ بایت اول این ۳۲ بایت استفاده کرده‌ایم و با دسترسی به متغیر name از کل این ۳۲ بایت استفاده کرده‌ایم.

انواع داده تعریف شده توسط کاربر

- از نوع داده اجتماع (union) هنگامی استفاده می‌کنیم که می‌دانیم در طول یک برنامه یک برنامه‌نویس تنها به یکی از متغیرهای اجتماع نیازمند است.

```
۱ struct Connection {  
۲     int type;  
۳     union {  
۴         struct SSH ssh;  
۵         struct Telnet telnet;  
۶     };  
۷ };  
۸ struct Connection con;  
۹ con.type = 1; // con.type = 2;  
۱۰ con.ssh.sid = 20; // con.telnet.tid = 10;
```

- در مثال بالا، استفاده کننده یک اتصال شبکه‌ای Connection یا از پروتکل SSH استفاده می‌کند و یا از پروتکل Telnet اما هیچگاه از هر دو به طور همزمان استفاده نمی‌کند.

انواع داده تعریف شده توسط کاربر

- در مثال زیر، دانشگاه از تعدادی افراد تشکیل شده است. یک فرد می تواند یا دانشجو یا استاد باشد، اما نمی تواند در یک زمان هم دانشجو باشد و هم استاد. پس `per[i]` یا به متغیر `pr` نیاز دارد و یا `st`.

```
۱ union Person {
۲     struct Student st;
۳     struct Perofessor pr;
۴     // ...
۵ }
۶ struct University {
۷     union Person per[10000];
۸     // ...
۹ };
۱۰ per[0].pr.salary = 1400000;
۱۱ per[200].st.average = 17;
```

انواع داده تعریف شده توسط کاربر

- از نوع داده شمارشی (enum) برای تعریف تعدادی از مقادیر صحیح ثابت استفاده می‌کنیم به طوری که بتوان به آن مقادیر با استفاده از اسامی آنها دسترسی پیدا کرد.

```
۱ enum week { Sat, Sun, Mon, Tue, Wed, Thu, Fri };  
۲ enum week today = Sun; // Sun == 1  
۳  
۴ enum flags { italics = 1, bold = 2, underline = 4};  
۵  
۶ enum season { Spring = 1, Summer, Autumn, Winter };  
۷ enum season now = Winter; // Winter == 4
```

– انواع داده مشتق شده¹ عبارتند از آرایه² و اشاره گر³.

¹ derived data types

² array

³ pointer

انواع داده مشتق شده

- نوع داده آرایه مجموعه‌ای از مقادیر که همه از یک نوع داده (اصلی یا تعریف شده توسط کاربر) هستند را تعریف می‌کند.
- یک آرایه را چنین تعریف می‌کنیم: `type name[size];`
- بدین صورت از نوع داده `type` به تعداد `size` خانه در حافظه با نام `name` فضا تخصیص داده‌ایم.
- به هر کدام از اعضای آرایه می‌توان با اندیس ¹ آن دسترسی پیدا کرد: `name[index]` به طوری که $0 \leq \text{index} < \text{size}$. همچنین می‌توانیم بنویسیم `*(name+index)`
- برای مثال `int list[10];` تعداد ۱۰ خانه در حافظه از نوع داده عدد صحیح (هر خانه ۴ بایت) با نام `list` تخصیص می‌دهد.
- `list[0]` اولین عضو آرایه و `list[9]` آخرین عضو آرایه را مشخص می‌کند.

¹ index

انواع داده مشتق شده

- کامپایلر زبان سی محدوده دسترسی به یک آرایه را بررسی نمی‌کند. بنابراین دسترسی به خانه‌های حافظه‌ای که خارج از محدوده تعریف شده‌اند نیز امکان‌پذیر است. پس با استفاده از `name[size+1]` می‌توان به یک خانه از حافظه بعد از آرایه دسترسی پیدا کرد. برنامه‌نویس باید این محدوده‌ها را در هنگام نوشتن برنامه لحاظ کند.
- آرایه‌ها را می‌توان به صورت دوبعدی یا چندبعدی نیز تعریف کرد. برای مثال برای یک آرایه دوبعدی می‌توانیم تعریف کنیم: `type array[row][column]`؛ به طوری که `row` و `column` دو عدد صحیح هستند، `array` نام آرایه، و `type` نوع آرایه است.
- برای دسترسی به سطر `i` و ستون `j` در یک آرایه دوبعدی می‌توانیم از `array[i][j]` استفاده کنیم. همچنین می‌توانیم بنویسیم `(*(array + i) + j)`. در بحث اشاره‌گرها بدین موضوع خواهیم پرداخت.

انواع داده مشتق شده

- از آرایه‌ها می‌توانیم برای ذخیره رشته‌ها نیز استفاده کنیم.
- `char str[30];` یک رشته با ۳۰ کاراکتر تعریف می‌کند.
- می‌توانیم به صورت‌های مختلف به این رشته یک مقدار اولیه اختصاص دهیم.
- برای مثال `char str[30] = { 'h', 'e', 'l', 'l', 'o' };` یا `char str[30] = "hello";`
- بعد از آخرین در حرف در یک رشته کاراکتر `'\0'` قرار می‌گیرد که انتهای رشته را مشخص می‌کند. در صورتی که طول آرایه در مقداردهی اولیه برابر با طول رشته اولیه است، به طور دستی باید آخرین حرف را برابر با `'\0'` قرار داد.
- همچنین می‌توانیم آرایه‌ای از رشته‌ها به صورت `char str_array[num][len];` تعریف کنیم جایی که `num` یک عدد صحیح و تعداد رشته‌هاست و `len` یک عدد صحیح و طول هر یک از رشته‌های آرایه است.

انواع داده مشتق شده

- برای عملیات بر روی رشته‌ها می‌توانیم از توابعی که در کتابخانه `<string.h>` تعریف شده‌اند استفاده کنیم.
- تابع `strcat(str1, str2)` رشته دوم را به رشته اول الحاق می‌کند.
- تابع `strcpy(str1, str2)` رشته دوم را در رشته اول کپی می‌کند.
- تابع `strcmp(str1, str2)` رشته دوم را با رشته اول مقایسه می‌کند.
- تابع `strstr(str1, str2)` رشته دوم را در رشته اول جستجو می‌کند.

- برای مثال تابع `strstr` اشاره‌گری از نوع کاراکتر به ابتدای رشته یافته شده باز می‌گرداند و از آن می‌توان به صورت زیر استفاده کرد.

```
۱ char str[] = "This is a simple string";  
۲ char * pch;  
۳ pch = strstr (str, "simple");  
۴ if (pch != NULL)  
۵     strncpy (pch, "sample", 6);  
۶ puts (str);
```

انواع داده مشتق شده

- یک نوع دیگر از انواع داده مشتق شده اشاره گر¹ نام دارد.
- یک اشاره گر آدرس یک خانه در حافظه را نگهداری می کند. در سیستم های ۳۲ بیتی برای نگهداری یک آدرس به چهار بایت نیاز است و در سیستم های ۶۴ بیتی به هشت بایت.
- یک اشاره گر را به صورت `p * type` تعریف می کنیم.
- `p` به آدرسی از حافظه اشاره می کند که در آن متغیری از نوع `type` نگهداری می شود. پس `p` یک متغیر هشت بایتی در حافظه است که یک آدرس هشت بایتی را نگهداری می کند.

¹ pointer

انواع داده مشتق شده

- برای دسترسی به محتوای حافظه از عملگر رفع ارجاع یا عملگر ستاره ($*p$) استفاده می‌کنیم.
- اگر یک متغیر از نوع داده اصلی یا تعریف شده به صورت `type x` تعریف شده باشد می‌توان به آدرس آن متغیر با استفاده از عملگر ارجاع یا عملگر امپرسند ($\&x$) دسترسی پیدا کرد.
- برای مثال می‌توانیم داشته باشیم $p = \&x$ که بدین معنی است که p به آدرس x اشاره می‌کند و یا $x = *p$ برای اینکه مقداری که اشاره‌گر p به آن اشاره می‌کند در متغیر x کپی شود و یا $*p = x$ بدین معنی که مقدار متغیر x در خانه‌ای از حافظه که p بدان اشاره می‌کند کپی شود.

انواع داده مشتق شده

- مقداری که یک اشاره‌گر نگهداری می‌کند و یا به عبارتی آدرسی که نگهداری می‌کند قابل تغییر است، پس می‌توانیم داشته باشیم $p++$ بدین معنا که p به خانه حافظه بعدی اشاره کند. نوع اشاره‌گر p مشخص می‌کند که عملگر $++$ چند بایت باید اضافه شود. مثلاً اگر اشاره‌گر p از نوع عدد صحیح باشد، $p++$ مقدار متغیر p را به اندازه ۴ بایت افزایش می‌دهد.
- همچنین می‌توانیم از عملگرهای $+$ و $-$ استفاده کنیم و مقدار یک اشاره‌گر را با یک عدد صحیح جمع و یا یک عدد صحیح را از آن بکاهیم و بدین صورت آدرس اشاره‌گر را افزایش یا کاهش دهیم. برای مثال $p=p+3$; مقدار p را به اندازه ۳ واحد اضافه می‌کند و اگر p از نوع عدد صحیح باشد هر واحد از این ۳ واحد ۴ بایت است.
- اشاره‌گرها را همچنین می‌توانیم از هم کم کنیم ولی نمی‌توانیم با هم جمع، در هم ضرب یا بر هم تقسیم کنیم.
- تفاضل دو اشاره‌گر فاصله بین خانه‌های حافظه‌ای را مشخص می‌کند که آن دو اشاره‌گر به آنها اشاره می‌کنند.

انواع داده مشتق شده

- با استفاده از کلیدواژه `const` می‌توانیم متغیری تعریف کنیم که مقدار آن تغییر نمی‌کند.
- همچنین در تعریف یک اشاره‌گر می‌توانیم از کلیدواژه `const` استفاده کنیم.
- `const type * p;` بدین معناست که محتوای خانه‌ای از حافظه که `p` بدان اشاره می‌کند (از طریق دسترسی با عملگر ستاره) قابل تغییر نیست.
- `type * const p = &x;` بدین معناست که آدرسی که در `p` نگهداری می‌شود ثابت و غیرقابل تغییر است.
- همچنین اسامی آرایه‌ها اشاره‌گر هستند. البته این اشاره‌گرها ثابت هستند و مقدار آدرس آنها قابل تغییر نیست.
- پس `type a[size]` مانند `const type * a` عمل می‌کند.

انواع داده

- متغیرها همچنین می‌توانند به صورت ایستا (static) تعریف شوند. یک متغیر ایستا که در یک تابع تعریف شده است، مقدار خود را در فراخوانی‌های مختلف نگه می‌دارد.
- تابع زیر را در نظر بگیرید.

```
۱ void f() {  
۲     int a = 0; static int sa = 0; a += 1; sa += 1;  
۳     printf("a = %d, sa = %d\n", a, sa);  
۴ }  
۵ int main() {  
۶     for (int i = 0; i < 10; ++i) f();  
۷ }
```

- در هر بار فراخوانی تابع f مقدار متغیر sa یک واحد اضافه می‌شود و بعد از ۱۰ فراخوانی مقدار آن به ۱۰ می‌رسد، اما مقدار متغیر a در هر بار ورود به تابع صفر می‌شود و مقدار آن در هر بار فراخوانی ۱ است.

چیدمان حافظه هنگام اجرا

- در هنگام اجرای یک برنامه، حافظه به چند قسمت تقسیم می‌شود.
- در بخش کد¹، کد برنامه و بخش داده²، داده‌ها (مانند متغیرهای عمومی³ و متغیرهای ایستا⁴) قرار می‌گیرند.
- پشته⁵ فراخوانی در قسمتی دیگر از حافظه است که داده‌های مورد نیاز در فراخوانی توابع را نگهداری می‌کند و در نهایت قسمتی از حافظه که هرم یا هیپ⁶ نامیده می‌شود، متغیرهایی را نگهداری می‌کند که به طور پویا تخصیص داده می‌شوند.

¹ code segment

² data segment

³ global variable

⁴ static variables

⁵ call stack

⁶ heap

- هرگاه یک تابع فراخوانی می‌شود، همه متغیرهای تعریف شده در آن تابع در پشته فراخوانی¹ ذخیره می‌شوند، و پس از پایان اجرای تابع همه متغیرهای تعریف شده در آن تابع حذف می‌شوند.
- حال فرض کنید یک تابع بدین صورت تعریف شده باشد: `void f(int a);` متغیر `a` در پشته فراخوانی برای تابع `f` تعریف شده است و هرگاه اجرای تابع `f` به پایان برسد، متغیر `a` از حافظه پاک می‌شود.

¹ call stack

فراخوانی با مقدار

- حال کد زیر را در نظر بگیرید:

```
۱ void swap(int x, int y) {  
۲     int z = x; x = y; y = z;  
۳ }  
۴ int main() {  
۵     int a=2, b=3;  
۶     swap(a,b);  
۷     return 0;  
۸ }
```

- از آنجایی که متغیرهای a و b در پشتۀ فراخوانی برای تابع main تعریف شده‌اند، لذا در تابع swap قابل دسترسی نیستند. متغیر a در متغیر x کپی می‌شود و متغیر b در متغیر y. پس جابجا کردن محتوای متغیرهای x و y در محتوای a و b تأثیری ندارد و مقادیر a و b را تغییر نمی‌دهد.

- فراخوانی یک تابع با ارسال مقادیر به آن تابع را فراخوانی با مقدار¹ می‌نامیم.

¹ call by value

فراخوانی با ارجاع

- کد زیر را در نظر بگیرید:

```
۱ void swap(int * x, int * y) {  
۲     int z = *x; *x = *y; *y = z;  
۳ }  
۴ int main() {  
۵     int a=2, b=3;  
۶     swap(&a,&b);  
۷     return 0;  
۸ }
```

- در اینجا متغیرهای x و y به a و b اشاره می‌کنند. پس جابجا کردن محتوای متغیرهای x و y در محتوای a و b تأثیری دارد و مقادیر a و b را تغییر می‌دهد.

- فراخوانی یک تابع با ارسال اشاره‌گر به آن تابع را فراخوانی با ارجاع¹ می‌نامیم.

¹ call by reference

- از آنجایی که فراخوانی با مقدار هزینه دارد بدین معنی که کپی کردن یک متغیر در متغیر دیگر هم زمان بر است و هم مقدار حافظه بیشتری اشغال می‌کند، لذا در بسیاری موارد با اینکه نیازی به تغییر محتوای یک متغیر در یک تابع نداریم، اما از فراخوانی با ارجاع استفاده می‌کنیم.
- گرچه فراخوانی با ارجاع هزینه زمانی و هزینه استفاده از حافظه را کاهش می‌دهد، اما یک مشکل نیز دارد. مشکل این است که ممکن است تابع فراخوانی کننده نخواهد تابع فراخوانی شونده، مقادیری که به آن داده می‌شود را تغییر دهد.
- بدین منظور از کلیدواژه `const` برای تعریف متغیرهای تابع استفاده کنیم.

فراخوانی با ارجاع

- کد زیر را در نظر بگیرید. با اینکه فراخوانی تابع `print` با ارجاع است، و مقدار `stu` در متغیر `s` کپی نمی‌شود و `s` به `stu` اشاره می‌کند (و در نتیجه هزینه فراخوانی کاهش می‌یابد)، اما تابع `print` نمی‌تواند مقدار متغیر `stu` را تغییر دهد.

```
۱ struct student {
۲     int id;
۳     char name[1000];
۴ };
۵ void print(const struct student * s) {
۶     printf("%d %s \n", s->id, s->name);
۷ }
۸ int main() {
۹     struct student stu;
۱۰    strcpy(stu.name, "Ali");
۱۱    print(&stu);
۱۲    return 0;
۱۳ }
```

تخصیص حافظه پویا

- حافظه را می‌توان توسط کلیدواژه `malloc` به طور پویا تخصیص داد¹.
- برای مثال با دستور `int * p = (int*) malloc(100);` مقدار ۱۰۰ بایت از حافظه به طور پویا تخصیص داده می‌شود که اشاره‌گر `p` به آن مکان از حافظه اشاره می‌کند.
- دقت کنید که متغیر `p` در فضای پشته قرار می‌گیرد زیرا متغیری است که در حوزه یکی از توابع تعریف شده است، ولی حافظه پویا در هیپ تخصیص داده می‌شود.
- اگر مقدار اشاره‌گر `p` از دست برود دسترسی به فضایی که آن اشاره‌گر به آن اشاره می‌کند ناممکن می‌شود.
- با استفاده از کلیدواژه `free` می‌توان حافظه تخصیص داده شده در هیپ را آزاد کرد.
- اگر فضاهای تخصیص داده شده آزاد نشوند حافظه رشد می‌کند و مقداری زیادی از حافظه هیپ بلااستفاده می‌ماند.
- دقت کنید که از آنجایی که حافظه هیپ از پشته بزرگتر است، لذا آرایه‌های بسیار بزرگ را بهتر است به طور پویا تخصیص داد.

¹ dynamically allocate

تخصیص حافظه پویا

- همچنین می‌توانیم آرایه‌های دوبعدی (و یا چندبعدی) با استفاده از اشاره‌گر به اشاره‌گر در حافظه تخصیص دهیم.

- در مثال زیر یک آرایه ۲۰ در ۳۰ برای ذخیره اعداد صحیح در حافظه تخصیص می‌دهیم.

```
۱ int ** array = (int **)malloc(20*sizeof(int*));
۲ for (int i=0; i<20; i++) {
۳     array[i] = (int*)malloc(30*sizeof(int));
۴ }
۵ int i=2; int j=3
۶ array[i][j] = 7;
۷ *((array+i)+j) = 8;
```

تخصیص حافظه پویا

- برای دسترسی به عنصر i ام در یک آرایه یک بعدی می‌توانیم از عملگر رفع ارجاع به صورت $*(array+i)$ استفاده کنیم، بدین معنی که در حافظه به تعداد i خانه (طول هر خانه به نوع آرایه بستگی دارد) به جلو حرکت کرده و مقداری که در آن خانه وجود دارد را استخراج می‌کنیم. از آنجایی که آرایه دوبعدی در واقع اشاره‌گری به اشاره‌گرها است، ابتدا i خانه به جلو حرکت کرده، آدرس سطر i ام را استخراج کرده، سپس در فضای حافظه مربوط به سطر i ام، j خانه در حافظه حرکت و مقدار خانه حافظه مورد نظر را استخراج می‌کنیم.

```
۱ int ** array = (int **)malloc(20*sizeof(int*));
۲ for (int i=0; i<20; i++) {
۳     array[i] = (int*)malloc(30*sizeof(int));
۴ }
۵ int i=2; int j=3
۶ array[i][j] = 7;
۷ printf("array[%d][%d]=%d \n", i, j, (*(array+i)+j)); //array[2][3]=7
```

ساختارهای شرطی

- ساختارهای شرطی¹ در زبان سی برای انتخاب یک دسته از دستورات برای اجرا استفاده می‌شوند.
- دو دسته از ساختارهای شرطی وجود دارند که `if ... else` و `switch ... case` نامیده می‌شوند.
- ساختار `if ... else` به صورت زیر است.

```
۱ if (<condition>) {  
۲     // code1  
۳ } else {  
۴     // code2  
۵ }
```

- در صورتی که مقدار `condition` درست باشد `code1` اجرا می‌شود و در غیر این صورت `code2` اجرا می‌شود.

¹ conditional structures

- ساختار case ... switch به صورت زیر است.

```
۱ switch (<expression>) {  
۲     case <value1> :  
۳         // code1  
۴         break;  
۵     case <value2> :  
۶         // code2  
۷         break;  
۸     ...  
۹     default :  
۱۰         //codeD  
۱۱ }
```

در صورتی که مقدار عبارت expression برابر با value1 باشد، code1 اجرا می‌شود، در صورتی که مقدار آن برابر با value2 باشد، code2 اجرا می‌شود، الی آخر. در صورتی که مقدار عبارت expression برابر با هیچ یک از مقادیر تعیین شده در case ها نباشد، آنگاه codeD در شاخه default اجرا می‌شود.

ساختارهای تکرار

- ساختارهای تکرار شامل while، while ... do، و for می‌شوند.
- ساختار while به صورت زیر است.

```
۱ // initialization
۲ while (<condition>) {
۳     // code
۴     // step
۵ }
```

- ابتدا متغیرهای مورد نیاز برای شرط در قسمت initialization مقداردهی اولیه می‌شوند، سپس تا وقتی شرط condition برقرار است، code اجرا می‌شود، سپس متغیرهای مورد نیاز در شرط در قسمت step تغییر داده می‌شوند و شرط مجدداً سنجیده می‌شود. این حلقه تا زمانی ادامه پیدا می‌کند که شرط برقرار است.

- ساختار `while ... do` به صورت زیر است.

```
۱ // initialization
۲ do {
۳     // code
۴     // step
۵ while (<condition>);
```

- ابتدا متغیرهای مورد نیاز برای شرط در قسمت `initialization` مقداردهی اولیه می‌شوند، سپس تا وقتی شرط `condition` برقرار است، `code` اجرا می‌شود. متغیرهای مورد نیاز در شرط در قسمت `step` تغییر داده می‌شوند.

- ساختار for به صورت زیر است.

```
۱ for (<initialization> ; <condition> ; <step>) {  
۲     // code  
۳ }
```

- ابتدا متغیرهای مورد نیاز برای شرط در قسمت initialization مقداردهی اولیه می‌شوند، سپس تا وقتی شرط condition برقرار است، code اجرا می‌شود. سپس اجرا به ابتدای حلقه for باز می‌گردد، متغیرهای مورد نیاز در شرط در قسمت step تغییر داده می‌شوند، شرط بررسی می‌شود و این حلقه ادامه پیدا می‌کند تا وقتی که مقدار condition درست است.

- همچنین در ساختارهای تکرار (حلقه‌ها)، می‌توانیم از دستورات break و continue استفاده کنیم.
- دستور break باعث می‌شود اجرای برنامه از حلقه خارج شود.
- دستور continue باعث می‌شود اجرای برنامه به ابتدای حلقه بازگردد.

اشاره‌گر به تابع

- در زبان سی می‌توانیم اشاره‌گر به تابع¹ تعریف کنیم. برای این کار باید از امضای تابع² استفاده کنیم. امضای تابع مشخص می‌کند یک تابع چند ورودی از چه نوع‌های داده دارد و نوع داده خروجی آن چیست.
- برای مثال یک اشاره‌گر به تابع با دو ورودی عدد صحیح و اعشاری و یک خروجی بولی را به صورت زیر تعریف می‌کنیم.

```
۱ bool function(int i, double d) {  
۲     // ...  
۳ }  
۴  
۵ bool (*ptr) (int, double);  
۶ ptr = function;
```

- سپس این اشاره‌گر به تابع را می‌توانیم با نام یک تابع مقاردهی کنیم. پس نام توابع در واقع اشاره‌گر به تابع هستند.

¹ function pointer

² function signature

اشاره‌گر به تابع

- فرض کنید می‌خواهیم به چند تابع توسط آرایه‌ای از اشاره‌گرها به توابع دسترسی پیدا کنیم.

```
۱ double add(double x, double y) { return x+y; }
۲ double sub(double x, double y) { return x-y; }
۳ double mul(double x, double y) { return x*y; }
۴ double div(double x, double y) { return x/y; }
۵
۶ int main() {
۷     double (*op[])(double, double) = { add, sub, mul, div };
۸     int i;
۹     double a,b;
۱۰    scanf("%d", i);
۱۱    scanf("%f", a); scanf("%f", b);
۱۲    if (i>=0 && i<4)
۱۳        op[i](a, b);
۱۴    return 0;
۱۵ }
```

اشاره‌گر به تابع

- همچنین می‌توانیم تابعی تعریف کنیم که در ورودی یک تابع را دریافت می‌کند. برای این کار از اشاره‌گر به تابع استفاده می‌کنیم.

```
۱ double operation(double x, double y, double(*op)(double, double)) {  
۲     return op(x,y);  
۳ }  
۴  
۵ int main() {  
۶     double res;  
۷     res = operation(8, 3, div);  
۸  
۹     return 0;  
۱۰ }
```


– با استفاده از زبان سی یک صف پیاده‌سازی کنید.

```
۱ #include <stdio.h>
۲ #include <string.h>
۳ #include <stdlib.h>
۴ #include <stdbool.h>
۵ #define MAX 6
۶
۷ int intArray[MAX];
۸ int front = 0;
۹ int rear = -1;
۱۰ int itemCount = 0;
```

```
۱ int peek() {  
۲     return intArray[front];  
۳ }  
۴  
۵ bool empty() {  
۶     return itemCount == 0;  
۷ }  
۸  
۹ bool full() {  
۱۰    return itemCount == MAX;  
۱۱ }  
۱۲  
۱۳ int size() {  
۱۴     return itemCount;  
۱۵ }
```

```
۱ void push(int data) {  
۲     if(!full()) {  
۳         if(rear == MAX-1) {  
۴             rear = -1;  
۵         }  
۶         intArray[++rear] = data;  
۷         itemCount++;  
۸     }  
۹ }
```

```
۱ int pop() {  
۲     int data = 0;  
۳     if (!empty()) {  
۴         data = intArray[front++];  
۵         if(front == MAX) {  
۶             front = 0;  
۷         }  
۸         itemCount--;  
۹     }  
۱۰     return data;  
۱۱ }
```

```
۱ int main() {
۲     /* insert 5 items */
۳     push(3); push(5); push(9); push(1); push(12);
۴     // front : 0 , rear : 4
۵     // index : 0 1 2 3 4
۶     // queue : 3 5 9 1 12
۷
۸     push(15);
۹     // front : 0, rear : 5
۱۰    // index : 0 1 2 3 4 5
۱۱    // queue : 3 5 9 1 12 15
۱۲
۱۳    if(full()) {
۱۴        printf("Queue is full!\n");
۱۵    }
```

```
۱ // remove one item
۲ int num = pop();
۳ printf("Element removed: %d\n",num);
۴ // front : 1, rear : 5
۵ // index : 1 2 3 4 5
۶ // queue : 5 9 1 12 15
۷
۸ // insert more items
۹ push(16);
۱۰ // front : 1, rear : 0
۱۱ // index : 0 1 2 3 4 5
۱۲ // queue : 16 5 9 1 12 15
۱۳
۱۴ // As queue is full, elements will not be inserted.
۱۵ push(17); push(18);
۱۶ // index : 0 1 2 3 4 5
۱۷ // queue : 16 5 9 1 12 15
```

```
۱    printf("Element at front: %d\n",peek());
۲
۳    printf("index : 5 4 3 2 1 0\n");
۴    printf("Queue:  ");
۵
۶    while(!empty()) {
۷        int n = pop();
۸        printf("%d ",n);
۹    }
۱۰ }
```

- حال اگر بخواهیم به جای یک صف، همزمان از چند صف استفاده کنیم، نیاز به پیاده‌سازی صفی داریم که بتوان از آن چندین نمونه ساخت.
- در واقع متغیرهای `intArray`، `front`، `rear`، `itemCount` باید برای هر صف متمایز باشد.
- این متغیرها را می‌توانیم در یک ساختمان `struct` قرار دهیم و سپس برای هر نمونه از صف یک نمونه از ساختمان صف ساخت.
- از آنجایی که می‌خواهیم اندازه صف متغیر باشد، این بار آن را با استفاده از یک اشاره‌گر تعریف می‌کنیم.

```
۱ struct queue {  
۲     int * intArray;  
۳     int front; int rear;  
۴     int itemCount; int size;  
۵ };  
۶ typedef struct queue Queue;
```

- حال تابعی تعریف می‌کنیم که صف را بسازد و مقداردهی اولیه کند.

```
۱ Queue construct_queue(int s) {  
۲     Queue res;  
۳     res.intArray =(int *)malloc(sizeof(int)*s);  
۴     res.front = 0; res.rear = -1;  
۵     res.itemCount = 0; res.size = s;  
۶     return res;  
۷ }
```

- همچنین برای جلوگیری از نشت حافظه، باید در پایان صف را تخریب و حافظه اشغال شده توسط آن را آزاد کنیم.

```
۱ void destroy_queue(Queue * q) {  
۲     free(q->intArray);  
۳ }
```

- همهٔ توابع مشابه قبل پیاده‌سازی می‌شوند با این تفاوت که یک صف را نیز به عنوان ورودی دریافت می‌کنند.

```
۱ bool empty(Queue * q) {  
۲     return q->itemCount == 0;  
۳ }  
۴  
۵ bool full(Queue * q) {  
۶     return q->itemCount == q->size;  
۷ }  
۸ ...
```

- برای استفاده از صف باید توابع ساخت و تخریب صف فراخوانی شوند.

```
۱ Queue q;  
۲ q = construct_queue(10);  
۳ push(&q,3); push(&q,5);  
۴ ...  
۵ destroy_queue(&q);
```

- چندین مشکل در این پیاده‌سازی وجود دارد که با استفاده از زبان سی قابل رفع نیستند.
- ۱. متغیرهای درون `struct queue` قابل دسترسی و قابل تغییر هستند. چنانچه مقادیر این متغیرها تغییر کنند (یا دستکاری شوند)، برنامه به درستی کار نخواهد کرد.
- ۲. الزامی به فراخوانی تابع `construct_queue` در ابتدا و `destroy_queue` در انتها وجود ندارد، پس ممکن است صف به درستی ساخته نشود و یا به درستی تخریب نشود.
- ۳. صف پیاده‌سازی شده فقط برای اعداد صحیح قابل استفاده است و برای سایر انواع داده نمی‌تواند مورد استفاده قرار بگیرد. حتی اگر چندین صف برای چندین نوع داده پیاده‌سازی شوند، ممکن است انواع داده‌های تعریف شده توسط کاربر در آینده به وجود بیایند که در زمان پیاده‌سازی صف وجود نداشتند.
- ۴. چنانچه در آینده نیاز به پیاده‌سازی صفی خاص وجود داشته باشد، همه صف باید دوباره پیاده‌سازی شود و استفاده از بخشی از این صف در یک نوع صف دیگر امکان‌پذیر نیست.

مقایسهٔ سی و سی‌پلاس‌پلاس

- در زبان سی‌پلاس‌پلاس در کنار مفهوم struct مفهوم class نیز وجود دارد. یک نمونه از یک کلاس را یک شیء می‌نامیم. یک کلاس داده‌ها و توابع را در کنار هم قرار می‌دهد و برای داده‌ها و توابع سطح دسترسی تعیین می‌کند. پس اگر داده‌ای در یک کلاس دادهٔ خصوصی تعریف شود، دسترسی و تغییر آن امکان‌پذیر نخواهد بود.
- هرگاه یک شیء ساخته می‌شود سازندهٔ آن فراخوانی می‌شود و هرگاه تخریب می‌شود مخرب آن فراخوانی می‌شود.
- یک کلاس می‌تواند از یک کلاس دیگر داده‌ها و توابعی را به ارث ببرد. پس تعاریف درون کلاس‌ها در کلاس‌های دیگر می‌توانند مورد استفاده قرار بگیرند.
- یک کلاس می‌تواند به طور عمومی تعریف شود به طوری که برخی از متغیرهای آن با همهٔ انواع داده قابل استفاده باشند.
- یک عملگر می‌تواند برای کلاس‌هایی که توسط کاربر تعریف شده‌اند، بازتعریف شود. همچنین در زبان سی‌پلاس‌پلاس مفاهیم جدیدی مانند مدیریت استثناها وجود دارد که در آینده بررسی خواهیم.

مقدمه

- در سال ۱۹۷۹ یعنی حدود ۷ سال پس از تکمیل زبان سی، بیارنه استراستروپ^۱ کار بر روی زبانی جدید به نام زبان سی با کلاس‌ها^۲ را در آزمایشگاه‌های بل^۳ آغاز کرد. در آن زمان زبان Simula به عنوان زبانی که در آن مفاهیم شیء‌گرایی به کار می‌رفت استفاده میشد. برنامه‌سازان با استفاده از این زبان شیء‌گرا می‌توانستند برنامه‌های بسیار بزرگ را نظم دهند و خوانایی برنامه را افزایش دهند. از طرفی زبان C زبان بسیار پرکاربرد و کارآمدی برای سیستم‌عامل یونیکس به حساب می‌آمد. بنابراین انگیزه اصلی از طراحی زبان سی با کلاس‌ها ایجاد زبانی شیء‌گرا مانند سیمولا بود که به اندازه سی کارآمد باشد. در سال ۱۹۸۴ این زبان به ++C تغییر نام یافت. دلیل این نامگذاری این بود که سی++ همه ویژگی‌های سی را دارا بود بنابراین زبان جدیدی نبود، پس به جای تغییر نام آن به زبان دی، این زبان سی++ نامیده شد. در سال ۱۹۸۵ اولین نسخه از کتاب زبان برنامه‌سازی سی++^۵ منتشر شد.

^۱ Bjarne Stroustrup

^۲ C with Classes

^۳ AT&T Bell Labs

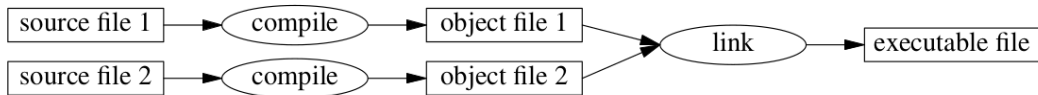
^۵ The C++ Programming Language

- در زبان‌های شیء‌گرا¹ مانند سی++ برخلاف زبان‌های رویه‌ای² یک برنامه از تعدادی شیء ساخته شده است که با یکدیگر در ارتباط هستند. پس زبان مدلسازی این زبان‌ها به زبان انسان و جهانی که انسان در آن زندگی می‌کند و توسط آن می‌اندیشد نزدیک‌تر است. جهان تشکیل شده است از اجسام و مفاهیم که با یکدیگر در ارتباط اند. همین‌طور یک برنامه در زبان‌های شیء‌گرا تشکیل شده است از تعدادی شیء که با یکدیگر در ارتباط اند. یک شیء که متعلق به یک کلاس یا یک خانواده است در واقع اجسام و مفاهیم را مدلسازی و پیاده‌سازی می‌کند. هر شیء تعدادی ویژگی و تعدادی رفتار دارد. ویژگی‌ها خصوصیات و ماهیت یک شیء را تعیین می‌کنند و رفتارهای عملیاتی که آن شیء می‌تواند انجام دهد.
- به طور مثال یک دانشجوی معین در یک برنامه سامانه دانشگاهی نوشته شده توسط یک زبان شیء‌گرا در واقع شیئی است از کلاس یا خانواده دانشجو. این دانشجو ویژگی‌هایی دارد مانند نام و شماره دانشجویی و رفتارهایی دارد مانند ورود به سامانه و یا اخذ یا حذف درس.
- در زبان‌های رویه‌ای مانند سی، یک برنامه تشکیل شده است از تعدادی تابع که هر کدام عملیات معینی را انجام می‌دهند. پس زبان‌های شیء‌گرا به زبان‌های مدل‌سازی ما در جهان واقعی نزدیک‌تر اند.

¹ object-oriented programming languages

² procedural programming languages

- کامپایلر سی++ متن سورس برنامه را که در یک فایل سورس¹ نگهداری می‌شود به زبان ماشین ترجمه می‌کند و فایل‌هایی به نام آبجکت² می‌سازد که حاوی برنامه به زبان ماشین مقصد برای اجرا است.
- سپس فایل‌های آبجکت باید توسط لینکر³ (یا پیونددهنده) به یکدیگر پیوند داده‌شوند و یک فایل اجرایی برای اجرا تهیه شود. معمولاً یک برنامه از تعداد زیادی فایل سورس تشکیل شده است.



¹ source file

² object file

³ linker

- در نهایت یک برنامه اجرایی¹ در قالب یک فایل اجرایی² برای یک سخت افزار مقصد تهیه می شود. این فایل قابل انتقال³ نیست، بدین معنی که نمی توان آن را از یک سیستم عامل به یک سیستم عامل دیگر یا از یک سخت افزار به سخت افزار دیگر انتقال داد و اجرا کرد.
- زبان C++ تشکیل شده است از یک هسته زبان⁴ و یک کتابخانه استاندارد⁵ که در آن بسیاری از ابزارهای مورد نیاز برنامه نویسان (توسط خود زبان سی++) پیاده سازی شده اند.

¹ executable program

² executable file

³ portable

⁴ core language

⁵ standard library

- یک برنامه کوتاه برای چاپ یک عبارت بر روی خروجی استاندارد در زبان سی++ به صورت زیر نوشته می‌شود.

```

۱ #include <iostream>
۲ int main(){
۳     std::cout << "Hello, World!\n";
۴ }

```

- در اینجا cout یک شیء است (مانند یک متغیر از یک نوع معین) که عملگر << برای آن تعریف شده است. با اعمال عملگر << بر روی شیء cout رشته‌ای که در طرف دیگر عملگر نوشته شده است بر روی خروجی استاندارد چاپ می‌شود. در واقع جمله Hello, World! بر روی استریم خروجی استاندارد¹ std::cout نوشته می‌شود.

- #include <iostream> امکانات استاندارد ورودی خروجی زبان را به متن برنامه اضافه می‌کند.

¹ standard output stream

```
۱ #include <iostream>
۲ // include (import) the declarations for the I/O stream library
۳ using namespace std;
۴ // make names from std visible without std::
۵
۶ double square(double x) {
۷     return x*x;
۸ } // square a double precision floating-point number
۹
۱۰ void print_square(double x) {
۱۱     cout << "the square of " << x << " is " << square(x) << "\n";
۱۲ }
۱۳
۱۴ int main() {
۱۵     print_square(1.234);
۱۶ } // print: the square of 1.234 is 1.52276
```

- معمولا برای انجام محاسبات طولانی، برنامه را به تعداد زیادی تابع تقسیم می‌کنیم. هر تابع وظیفه انجام قسمتی از محاسبات را دارد.
- برای تعریف توابع، نوع داده خروجی، نام تابع و نوع داده‌های ورودی را مشخص می‌کنیم. در فراخوانی توابع نوع داده‌های ورودی و خروجی باید با نوع تعریف شده مطابقت داشته باشد.

```

۱ Elem * next_elem();
۲ // no argument; return a pointer to Elem (an Elem*)
۳ void exit(int);
۴ // int argument; return nothing
۵ double sqrt(double);
۶ // double argument; return a double
۷
۸ double s2 = sqrt(2);
۹ // call sqrt() with the argument double{2}
۱۰ double s3 = sqrt("three");
۱۱ // error: sqrt() requires an argument of type double

```

- وقتی چند تابع با نام یکسان تعریف شده باشند، ولی ورودی‌ها و خروجی‌های آنها از نوع‌های متفاوت تعریف شده باشد، کامپایلر در هنگام فراخوانی، از تعریف تابعی استفاده می‌کند که ورودی و خروجی‌های مناسب داشته باشد.

```

۱ void print(int); // takes an integer argument
۲ void print(double); // takes a floating-point argument
۳ void print(string); // takes a string argument
۴ void user() {
۵     print(42); // calls print(int)
۶     print(9.65); // calls print(double)
۷     print("Hello"); // calls print(string)
۸ }
```

- تعریف چند تابع با یک نام را سربارگذاری تابع¹ می‌نامیم.

¹ function overloading

- اگر در هنگام فراخوانی دو تابع با نام یکسان ابهامی وجود داشته باشد، کامپایلر پیام خطا صادر می‌کند.

```
۱ void print(int, double);  
۲ void print(double, int);  
۳ void user2() {  
۴     print(0,0); // error : ambiguous  
۵     print(0.0,0); //calls print(double,int)  
۶ }
```

- همانند سی، در زبان C++ انواع داده اصلی، تعریف شده توسط کاربر، و مشتق شده وجود دارد.
- یکی از انواع داده که در زبان سی وجود ندارد، نوع bool است که یک مقدار منطقی درست یا نادرست را نگهداری می کند. متغیرهایی از این نوع یک بایت را در حافظه اشغال می کنند.
- عملگرهای C++ مانند عملگرهای زبان سی هستند.

مقداردهی اولیه

- متغیرها را به سه شکل می‌توان مقداردهی اولیه¹ کرد:

```
۱ double d1 = 2.3;  
۲ double d2 {2.3};  
۳ double d3 = {2.3};
```

- برای اطمینان از صحت مقداردهی اولیه معمولاً از شکل دوم یا سوم استفاده می‌کنیم:

```
۱ int i1 = 7.8; // it becomes 7  
۲ int i2 {7.8}; // error: floating-point to integer conversion  
۳ int i3 = {7.8}; // error: floating-point to integer conversion
```

- مقادیر ثابت باید همیشه مقداردهی اولیه شوند.

```
۱ const int i4 {7};
```

¹ initialize

- می‌توانیم از کلیدواژه `auto` برای تعریف یک متغیر استفاده کنیم. نوع چنین متغیری با توجه به محتوای کد تعیین می‌شود. با استفاده از این کلیدواژه می‌توان برنامه‌های کوتاه‌تری نوشت. متغیری که با `auto` تعریف می‌شود، باید حتما مقداردهی اولیه شود.

```
۱ auto b = true; // a bool
۲ auto ch = 'x'; // a char
۳ auto i = 123; // an int
۴ auto d = 1.2; // a double
۵ auto z = sqrt(y); // z has the type of whatever sqrt(y) returns
۶ auto bb {true}; // bb is a bool
```

محدودهٔ تعریف

- یک نام (نام متغیر، نام تابع، ...) در یک محدوده ¹ تعریف می‌شود.
- نام سراسری ² نامی است که در همه جای برنامه تعریف شده است.
- نام محلی ³ نامی است که در یک بلوک از کد تعریف شده و قابل دسترسی است. یک بلوک از کد بین دو علامت آکولاد { } قرار دارد.
- نام اعضای کلاس ⁴ نامی است که در یک کلاس تعریف شده است (در مورد کلاس در آینده بیشتر صحبت خواهیم کرد).
- نام اعضای فضای نام ⁵ نامی است که در یک فضای نام تعریف شده است (در مورد فضای نام در آینده بیشتر صحبت خواهیم کرد)

¹ scope

² global name

³ local name

⁴ class member name

⁵ namespace member name

```
۱ vector<int> vec;  
۲ // vec is global (a global vector of integers)  
۳ struct Record { string name; // ... };  
۴ // name is a member of Record (a string member)  
۵ void fct(int arg) // fct is global (a global function)  
۶ // arg is a local variable for fct (an integer argument)  
۷ {  
۸     string motto {"Truth shall set you free"};  
۹     // motto is local for fct  
۱۰    auto p = new Record{"Ali"};  
۱۱    // p points to an unnamed Record (created by new)  
۱۲ }
```

- به جای استفاده از malloc و free در سی++ از دو کلیدواژه new و delete استفاده می‌کنیم.

```
۱ Record * tp = new Record;  
۲ int * p = new int;  
۳ int * array = new int[20];  
۴ delete p;  
۵ delete tp;  
۶ delete[] array;
```

- تا زمانی که برای یک مکان حافظه که توسط new تخصیص داده شده است، delete فراخوانی نشده، آن مکان در حافظه باقی می‌ماند، حتی اگر از حوزه تعریف آن خارج شویم.

- در سی++ دو نوع ثابت وجود دارد.
- `const` به معنی مقدار ثابتی است که می‌تواند در زمان اجرا مقداردهی اولیه شود. به طور مثال برای اشاره‌گرها در توابع وقتی می‌خواهیم مقدار آنها تغییر نکند از `const` استفاده می‌کنیم.
- `constexpr` ثابتی است که در زمان کامپایل باید مقدار اولیه آن تعریف شده باشد.

```

۱ constexpr int dmV = 17; // dmV is a named constant
۲ int var = 17; // var is not a constant
۳ const double sqv = sqrt(var);
۴ // sqv is a named constant, possibly computed at run time
۵ double sum(const vector<double> * v);
۶ // sum will not modify its argument v, since it is constant
۷ const double s1 = sum(v);
۸ // OK: sum(v) is evaluated at run time
۹ constexpr double s2 = sum(v);
۱۰ // error : sum(v) is not a constant expression

```

- اگر یک تابع توسط constexpr تعریف شود، مقدار آن در زمان کامپایل محاسبه می‌شود.

```
۱ constexpr double square(double x) { return x*x; }
۲ constexpr double max1 = 1.4*square(17);
۳ // OK 1.4*square(17) is a constant expression
۴ constexpr double max2 = 1.4*square(var);
۵ // error : var is not a constant expression
۶ const double max3 = 1.4*square(var);
۷ // OK, may be evaluated at run time
```

- برای بهبود سرعت برنامه می‌توان توابع ساده را به صورت constexpr تعریف کرد.

آرایه، اشاره‌گر، و مرجع

- دنباله‌ای از داده‌ها در حافظه که همگی از یک نوع هستند را آرایه می‌نامیم.
- اشاره‌گر متغیری است که به یک خانه از حافظه اشاره می‌کند.
- با استفاده از عملگر ارجاع، می‌توان آدرس یک خانه از حافظه را استخراج و اشاره‌گری به یکی از اعضای آرایه تعریف کرد: `char * p = &v[3];`

آرایه، اشاره‌گر، و مرجع

- علاوه بر تعریف یک اشاره‌گر، در سی++ می‌توانیم یک متغیر مرجع نیز تعریف کنیم. یک متغیر مرجع را به صورت `type & var;` تعریف می‌کنیم. امکان تعریف آرایه‌های از متغیرهای مرجع وجود ندارد.
- پس از اینکه یک متغیر مرجع مقداردهی اولیه شد و به یک خانه از حافظه اشاره کرد، نمی‌توان مکانی در حافظه که آن متغیر به آن اشاره می‌کند را تغییر داد. پس یک مرجع برخلاف اشاره‌گر یک خانه در حافظه نیست که یک آدرس را نگهداری کند، بلکه نامی مستعار¹ است برای یک متغیر دیگر.
- همچنین برای دسترسی به مقدار یک متغیر مرجع به عملگر `*` نیازی نداریم.

```
۱ int v[] = {0,1,2,3,4,5,6,7,8,9};  
۲ int & ref1;  
۳ // error : declaration of reference ref1 required initializer  
۴ int & ref2 = v[2];  
۵ ref2 = 5; // here we change the value of v[2]  
۶ cout << "ref2: " << ref2 << " v[2]: " << v[2] << endl;
```

¹ alias

آرایه، اشاره‌گر، و مرجع

- استفاده اصلی متغیر مرجع برای فراخوانی با ارجاع است. بدین صورت دیگر نیازی به اشاره‌گر و اشغال فضای حافظه برای ذخیره‌سازی آدرس‌ها نخواهیم داشت.

```
۱ void swap(int & x, int & y) {  
۲     int tmp = x; x = y; y = tmp;  
۳ }
```

- هنگامی که می‌خواهیم از مرجع جهت کاهش سربار کپی استفاده کنیم ولی نمی‌خواهیم مقادیری که مرجع به آن اشاره می‌کند تغییر کنند، از کلیدواژه `const` استفاده می‌کنیم.

```
۱ double sum(const vector<double>&);
```

- همچنین مقدار بازگشت یک تابع می‌تواند یک متغیر مرجع باشد.

```
۱ int vals[] = { 2, 6, 1, 3, 5 , 4};  
۲  
۳ int & value(int i) {  
۴     if (i >= 0 && i <= 5)  
۵         return vals[i];  
۶ }  
۷  
۸ int a = value(3);  
۹ value(1) = 7; // we set vals[1] = 7  
۱۰ cout << value(4);  
۱۱ cin >> value(0);
```

- در سی++ می‌توانیم با استفاده از یک حلقه بر روی دامنه¹ توسط کلیدواژه for به هر یک از اعضای یک آرایه به ترتیب از اولین عضو تا آخرین عضو دسترسی پیدا کنیم.

```
۱ void print()  
۲ {  
۳     int v[] = {0,1,2,3,4,5,6,7,8,9};  
۴     for (auto x : v)  
۵         cout << x << '\n'; // for each x in v  
۶  
۷     for (auto x : {10,21,32,43,54,65})  
۸         cout << x << '\n';  
۹ }
```

¹ range for statement

آرایه، اشاره‌گر، و مرجع

- توجه کنید که در عبارت `auto x : v` هر یک از اعضای `v` در متغیر `x` کپی می‌شوند. اگر بخواهیم از سربار این کپی بکاهیم، می‌توانیم از مرجع استفاده کنیم: `auto & x : v`
- همچنین با استفاده از مرجع می‌توانیم متغیر `x` را تغییر داده و در نتیجه اعضای آرایه `v` را تغییر دهیم.

```
۱ void print() {  
۲     int v[] = {0,1,2,3,4,5,6,7,8,9};  
۳     for (auto & x : v) cout << x++ << " ";  
۴     cout << endl;  
۵     for (auto x : v)  
۶         cout << x << " ";  
۷     cout << endl;  
۸     for (const int & x : v)  
۹         cout << x << '\n';  
۱۰ }
```

آرایه، اشاره‌گر، و مرجع

- وقتی یک اشاره‌گر به هیچ مکانی در حافظه اشاره نمی‌کند از کلیدواژه `nullptr` برای مقداردهی اولیه آن استفاده می‌کنیم.

```
۱ double * pd = nullptr;
۲ int x = nullptr; // error: nullptr is a pointer not an integer
۳
۴ int count_x(const char * p, char x) {
۵     // count the number of occurrences of x in p[]
۶     // p is assumed to point to
۷     // a zero-terminated array of char (or to nothing)
۸     if (p==nullptr) return 0;
۹     int count = 0;
۱۰    for (; *p != 0; ++p)
۱۱        if (*p == x)
۱۲            ++count;
۱۳    return count;
۱۴ }
```

انواع داده تعریف شده توسط کاربر

- یک ساختمان یا struct یک نوع داده تعریف شده توسط کاربر است که برای تعریف یک نوع داده ترکیب شده از داده‌هایی از انواع مختلف در حافظه تحت یک نام واحد به کار می‌رود.

```
۱ struct Vector {  
۲     int sz; // number of elements  
۳     double* elem; // pointer to elements  
۴ };  
۵ void vector_init(Vector& v, int s) {  
۶     v.elem = new double[s]; // allocate an array of s doubles  
۷     v.sz = s;  
۸ }  
۹ Vector v;  
۱۰ vector_init(v,10);
```

- بر خلاف زبان سی در تعریف یک متغیر از یک ساختمان نیازی به واژه struct نیست.

انواع داده تعریف شده توسط کاربر

- برای متغیرهایی از نوع ساختمان، اگر به صورت مرجع تعریف شده باشند توسط عملگر نقطه، و اگر به صورت اشاره گر تعریف شده باشند، توسط عملگر \rightarrow می توانیم به اعضای آنها دسترسی پیدا کنیم.

```
۱ void f(Vector v, Vector& rv, Vector* pv) {  
۲     int i1 = v.sz; // access through name  
۳     int i2 = rv.sz; // access through reference  
۴     int i3 = pv->sz; // access through pointer  
۵ }
```

انواع داده تعریف شده توسط کاربر

- یک اجتماع یا union شبیه یک ساختمان است با این تفاوت که union تنها به اندازه بزرگترین عضو خود در حافظه فضا اشغال می کند.
- از یک اجتماع زمانی استفاده می کنیم که از بین چندین متغیر در هر زمان فقط به یکی از آنها نیاز داشته باشیم.
- برای مثال فرض کنید یک ورودی Entry همیشه یا یک شماره دارد و یا یک اسم. پس این ورودی را بدین صورت تعریف می کنیم.

```
۱ enum Type { str, num }; // a Type can hold values str and num
۲ struct Entry {
۳     Type t;
۴     string s; // use s if t==str
۵     int i; // use i if t==num
۶ };
```

- در اینجا همیشه یکی از متغیرهای s یا i بلا استفاده می ماند.

انواع داده تعریف شده توسط کاربر

- توسط یک اجتماع می توانیم ورودی Entry را چنین تعریف کنیم.

```
۱ union Value {  
۲     string s;  
۳     int i;  
۴ };  
۵ struct Entry {  
۶     Type t;  
۷     Value v; // use v.s if t==str; use v.i if t==num  
۸     // there is no extra space for v.i in the memory  
۹ };
```

انواع دادهٔ تعریف‌شده توسط کاربر

- یک کلاس class اساسی‌ترین مفهوم در زبان‌های شیء‌گرا است.
- یک کلاس یک نوع داده را تعریف می‌کند. این نوع داده تعدادی متغیر به همراه تعدادی تابع که بر روی آن متغیرها تغییر اعمال می‌کنند را کپسوله‌سازی¹ یا لفافه‌بندی می‌کند.
- یک کلاس تعدادی اعضا² دارد که این اعضا می‌توانند داده، یا تابع باشند.
- یک نمونه از یک کلاس را یک شیء می‌نامیم.

¹ encapsulate

² member

انواع داده تعریف شده توسط کاربر

- یک کلاس دارای یک سازنده و یک مخرب است. یک سازنده تابعی است که در هنگام ساخته شدن یک شیء به صورت خودکار فراخوانی می شود و یک مخرب تابعی است که در هنگام تخریب یک شیء به طور خودکار فراخوانی می شود.
- در یک ساختمان، دسترسی به همه اعضای ساختمان ممکن است. در یک کلاس برای داده ها سطح دسترسی تعریف می شود.
- اگر سطح دسترسی یک عضو کلاس public باشد، می توان به آن عضو از طریق شیء ساخته شده از کلاس دسترسی پیدا کرد. در صورتی که سطح دسترسی یک عضو private باشد، آن عضو توسط شیء ساخته شده از آن قابل دسترسی نیست.

انواع داده تعریف شده توسط کاربر

- کلاس Vector را می توان به صورت زیر به صورت یک کلاس تعریف کرد.

```
1 class Vector {
2 public:
3     Vector(int s) { elem = new double[s]; sz = s; }
4     double& value(int i) { return elem[i]; }
5     int size() { return sz; }
6     ~Vector() { delete[] elem; }
7 private:
8     double* elem; // pointer to the elements
9     int sz; // the number of elements
10 };
11 Vector v(6); // a Vector with 6 elements
12 cout << v.value(2);
```

- تابع سازنده Vector همیشه به محض ساختن یک شیء از کلاس Vector فراخوانی می شود. همچنین تابع مخرب ~Vector همیشه به محض تخریب یک شیء کلاس فراخوانی می شود.

انواع داده تعریف شده توسط کاربر

- حال می‌توانیم بدین صورت از Vector استفاده کنیم.

```
1 double read_and_sum(int s) {  
2     Vector v(s); // make a vector of s elements  
3     // we cannot access v.elem and v.sz  
4     // and we cannot change their values directly  
5     for (int i=0; i!=v.size(); ++i)  
6         cin>>v.value(i); // read into elements  
7     double sum = 0;  
8     for (int i=0; i!=v.size(); ++i)  
9         sum+=v.value(i); // take the sum of the elements  
10    return sum;  
11 }
```

انواع داده تعریف شده توسط کاربر

- در سی++ می توان از استراکت هم برای تعریف کلاس استفاده کرد ولی این کار پیشنهاد نمی شود و معمولا استراکت به طور سنتی شبیه استراکت های زبان سی استفاده می شود.
- فرق کلاس و استراکت در این است که اعضا در کلاس به طور پیش فرض خصوصی هستند در حالی که سطح دسترسی پیش فرض در استراکت عمومی است.

انواع داده تعریف شده توسط کاربر

- علاوه بر enum که در سی برای نامگذاری تعدادی مقدار صحیح به کار می رود، در سی++ نوع داده enum class نیز وجود دارد.
- با استفاده از enum class مقدار دو enum متفاوت را نمی توان به متغیرهایی از نوع متفاوت انتساب کرد.

```
۱ enum class Color { red, blue, green };
۲ enum class Traffic_light { green, yellow, red };
۳ Color col = Color::red;
۴ Traffic_light light = Traffic_light::red;
۵
۶ Color x = red; // error : which red?
۷ Color y = Traffic_light::red; // error: that red is not a Color
۸ Color z = Color::red; // OK
۹
۱۰ int i = Color::red; // error: Color::red is not an int
۱۱ Color c = 2; // initialization error: 2 is not a Color
```

انواع داده تعریف شده توسط کاربر

- عملگرهای مقایسه برای `enum class` تعریف شده اند.

```
۱ if (x > Color::blue) {  
۲     // do something  
۳ }
```

انواع داده تعریف شده توسط کاربر

- همچنین می توان برای یک `enum class` عملگرهای جدید تعریف کرد.

```
۱ // prefix increment: ++
۲ Traffic_light& operator++(Traffic_light& t) {
۳     switch (t) {
۴         case Traffic_light::green:
۵             return t=Traffic_light::yellow;
۶         case Traffic_light::yellow:
۷             return t=Traffic_light::red;
۸         case Traffic_light::red:
۹             return t=Traffic_light::green;
۱۰     }
۱۱ }
۱۲ Traffic_light light = Traffic_light::red;
۱۳ Traffic_light next = ++light;
۱۴ // next becomes Traffic_light::green
```

- در زبان سی++ از کتابخانه استاندارد <string> برای عملیات بر روی رشته‌ها استفاده می‌کنیم. در این کتابخانه کلاس string تعریف شده است.
- برای این کلاس عملگرها و توابع مورد نیاز برای کار بر روی رشته‌ها تعریف شده‌اند.
- برای مثال، عملگر + برای الحاق رشته‌ها به یکدیگر تعریف شده است.

```
۱ string compose(const string& name, const string& domain) {  
۲     return name + '@' + domain;  
۳ }  
۴ auto addr = compose("user", "computer");
```

- عملگر += دو رشته سمت چپ و راست عملگر را به یکدیگر الحاق و در رشته سمت چپ عملگر ذخیره می‌کند.

```
۱ string s1, s2;  
۲ s1 = s1 + '\n'; // append newline  
۳ s2 += '\n';
```

- تابع substr برای استخراج یک زیررشته از یک رشته، و تابع replace برای جایگزین کردن یک زیررشته استفاده می‌شود.

```
۱ name = "C++ language";  
۲ string s = name.substr(4,8); // s = "language"  
۳ name.replace(4,8,"programming"); // name becomes "C++ programming"
```

- عملگرهای دیگر از جمله = برای انتساب رشته‌ها، [] برای استخراج یک حرف از رشته، ==، != برای مقایسه تساوی رشته‌ها، <، <=، >، >= برای مقایسه رشته‌ها بر اساس ترتیب الفبایی برای کلاس رشته سربارگذاری شده‌اند.
- برای تبدیل یک رشته به یک `char *` جهت استفاده از توابعی که در کتابخانه‌های سی پیاده‌سازی شده‌اند، می‌توان از تابع `c_str()` استفاده کرد.

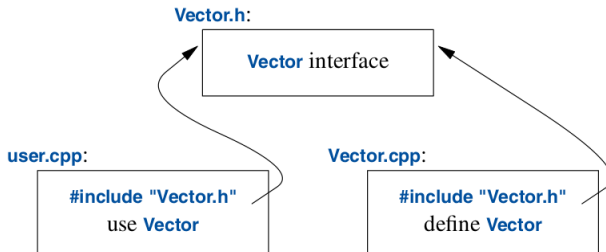
```

۱ string s = "hello";
۲ // s.c_str() returns a pointer to 's' characters
۳ printf("For people who like printf: %s\n",s.c_str());
۴ cout << "For people who like streams: " << s << '\n';

```

تقسیم‌بندی برنامه

- در یک برنامه سی++ معمولاً تعاریف توابع و کلاس‌ها در یک فایل به نام فایل سریتیر یا هدر¹ و پیاده‌سازی تعاریف در یک فایل جداگانه به نام فایل کدمنبع یا فایل سورس² قرار گرفته می‌شوند.
- هر فایل سورس به طور جداگانه کامپایل می‌شود و در نهایت لینکر فایل‌های آبجکت تولید شده پس از کامپایل فایل‌های سورس را به هم پیوند می‌دهد و فایل اجرایی می‌سازد.



¹ header file

² source file

- ممکن است فایل‌های آجکت توسط افراد دیگر به صورت کتابخانه‌هایی تولید شده و در دسترس برنامه‌نویسان قرار داده شده باشد.
- در این صورت، در هنگام پیوند دادن فایل‌های اجرایی، لینکر باید فایل‌های آجکت مربوط به آن کتابخانه‌ها را در اختیار داشته باشد. این کتابخانه‌ها را کتابخانه‌های ایستا می‌نامیم و فایل‌های آنها که مجموعه‌ای از فایل‌های آجکت است در لینوکس با پسوند `a` و در ویندوز با پسوند `lib` ذخیره می‌شوند.

- در مقابل کتابخانه‌های ایستا¹، کتابخانه‌های پویا² وجود دارند، که در زمان اجرا به توسط بارگذاری کننده یا لودر³ به فایل اجرایی متصل می‌شوند. فایل‌های این کتابخانه‌ها معمولاً در لینوکس با پسوند `.so` و در ویندوز با پسوند `.dll` ذخیره می‌شوند.
- برخلاف لینکر که فایل‌های اجرایی را با هم ادغام می‌کند، لودر فایل‌ها را ادغام نمی‌کند، بلکه آدرس توابع مورد نیاز در کتابخانه پویا را برای استفاده در فایل اجرایی استخراج و بارگذاری می‌کند.
- مزیت اصلی کتابخانه ایستا در این است که فایل اجرایی یکپارچه است و در نتیجه سرعت بیشتری دارد. مزیت اصلی کتابخانه پویا این است که فایل کتابخانه جدا است و در نتیجه می‌توان هر بار فایل کتابخانه جدیدی را بدون نیاز به کامپایل مجدد برنامه جایگزین کرد. همچنین حجم برنامه اجرایی با استفاده از کتابخانه پویا کوچک‌تر خواهد بود.

¹ static libraries

² dynamic libraries

³ loader

تقسیم‌بندی برنامه

- در یک برنامه سی++ معمولا تعاریف توابع و کلاس‌ها در یک فایل به نام فایل سریتیر یا هدر و پیاده‌سازی تعاریف در یک فایل جداگانه به نام فایل کدمنبع یا فایل سورس قرار گرفته می‌شوند.

```
۱ // Vector.h:
۲ class Vector {
۳ public:
۴     Vector(int s);
۵     double& value(int i);
۶     int size();
۷ private:
۸     double* elem;
۹     // elem points to an array of sz doubles
۱۰    int sz;
۱۱ };
```

- وقتی پیاده‌سازی را جداگانه در فایل کدمنبع قرار می‌دهیم، برای تعریف هر تابع، کلاس آن را نیز با استفاده از عملگر تفکیک حوزه (::) تعیین می‌کنیم.

```
۱ // Vector.cpp:
۲ #include "Vector.h" // get 'Vectors interface
۳
۴ // initialize members
۵ Vector::Vector(int s) { elem = new double[s]; sz = s;}
۶
۷ double& Vector::value(int i) { return elem[i]; }
۸
۹ int Vector::size() { return sz; }
```

تقسیم‌بندی برنامه

- یک برنامه، برای استفاده از کلاس‌ها و توابعی که در یک فایل هدر جداگانه تعریف شده‌اند، تنها نیاز به افزودن فایل هدر دارد.

```
۱ // user.cpp:
۲ #include "Vector.h"
۳ #include <cmath>
۴ // get 'Vectors interface
۵ // get the standard-library math function interface
۶ // including sqrt()
۷ double sqrt_sum(Vector& v) {
۸     double sum = 0;
۹     for (int i=0; i!=v.size(); ++i)
۱۰         sum+=std::sqrt(v.value(i));
۱۱     return sum;
۱۲ }
```

- برای اینکه در یک برنامه بسیار بزرگ تعداد نام‌ها زیاد است و نام‌ها ممکن است با یکدیگر مشابه باشند، در سی++ می‌توان فضای نام¹ تعریف کرد. بدین صورت نام‌ها با یکدیگر تداخل پیدا نمی‌کنند. در دو فضای نام می‌توانند فضای نام‌های مشابه وجود داشته باشند ولی در یک فضای نام، نام‌ها نمی‌توانند مشابه باشند.

```
۱ namespace List {  
۲     class Vector { };  
۳ };  
۴  
۵ namespace Euclidean {  
۶     class Vector { };  
۷ };  
۸  
۹ List::Vector lv;  
۱۰ Euclidean::Vector ev;
```

¹ namespace

فضای نام

- همچنین می‌توان یک فضای نام را با استفاده از کلیدواژه `using` به برنامه افزود. در اینصورت همه نام‌ها در آن فضای نام را می‌توان بدون استفاده از نام فضای نام استفاده کرد.

```
۱ std::string str1;  
۲  
۳ using namespace std;  
۴  
۵ string str;  
۶ cout << str;
```

- در صورتی که بخواهیم تنها از یکی از نام‌ها در یک فضای نام استفاده کنیم، می‌توانیم با استفاده از کلیدواژه `using` تنها آن نام را به برنامه بیفزاییم.

```
۱ using std::string;  
۲ string str; // OK  
۳ cout << str; // error : cout is undeclared
```

– فضاهای نام می‌توانند همچنین تودرتو باشند. بدین ترتیب می‌توانیم در یک فضای نام یک فضای نام دیگر تعریف کنیم.

```
۱ namespace N1 {  
۲     int i;  
۳     namespace N2 {  
۴         int i;  
۵         int j;  
۶     }  
۷ }  
۸  
۹ N1::i = 2;  
۱۰ N1::N2::i = 3  
۱۱ N1::N2::j = 4;
```

- همچنین فضاهاى نام مى‌توانند به صورت گسسته در فایل‌ها مختلف تعريف شوند.

```
۱ // file1.h
۲ namespace N1 {
۳     int i;
۴ }
۵
۶ // file2.h
۷ namespace N1 {
۸     int j;
۹ }
۱۰
۱۱ // main.cpp
۱۲ N1::i = 2;
۱۳ N1::j = 3;
```

برنامه سازی شیء گرا

- یک کلاس یک نوع داده¹ تعریف شده توسط کاربر است که یک مفهوم را پیاده‌سازی می‌کند. معمولاً در پیاده‌سازی یک سیستم، نیاز به تعدادی مفهوم انتزاعی داریم که هر کدام از این مفاهیم ویژگی‌هایی دارند. این مفاهیم را توسط یک کلاس در زبان سی++ نشان می‌دهیم و پیاده‌سازی می‌کنیم.
- برنامه‌ای که از تعدادی مفهوم تشکیل شده باشد که با هم در ارتباط هستند، قابل فهم‌تر است و راحت‌تر می‌توان آن را عیب‌یابی کرد. همچنین در چنین برنامه‌ای منطق برنامه را می‌توان بهتر دنبال کرد و بهره‌وری بهتری خواهد داشت.

¹ user-defined type

- یک نمونه از یک کلاس را شیء می‌نامیم. پس برای استفاده از یک کلاس باید یک شیء از آن بسازیم و عملیات تعریف شده برای کلاس را بر روی شیء مورد نظر انجام دهیم.
- یک کلاس تعدادی عضو داده ¹ و تعدادی تابع عضو ² دارد. اعضای داده‌ای می‌توانند از انواع اصلی، تعریف شده توسط کاربر (شامل کلاس‌ها)، و انواع مشتق شده باشند.
- به داده‌های یک کلاس ویژگی‌ها ³ و به توابع کلاس رفتارها ⁴ نیز می‌گوییم.
- هر عضو کلاس یک سطح دسترسی دارد. این سطح دسترسی می‌تواند عمومی (public)، خصوصی (private)، یا محافظت شده (protected) باشد.
- سطوح دسترسی عمومی و خصوصی را توضیح می‌دهیم و سطح دسترسی محافظت شده را در آینده بررسی خواهیم کرد.

¹ data member

² member function

³ attributes

⁴ behaviours

- سطح دسترسی عمومی بدین معناست که داده تعریف شده توسط این سطح دسترسی برای همه قابل استفاده است. پس اگر یک شیء از کلاسی تعریف شود، می‌توان به اعضای عمومی آن دسترسی پیدا کرد.

```
۱ class student {  
۲ public:  
۳     string name;  
۴ };  
۵ student st; //st is an object of student class  
۶ st.name = "ali";
```

- سطح دسترسی خصوصی بدین معناست که داده تعریف شده توسط این سطح دسترسی فقط برای توابع عضو قابل استفاده است. پس اگر یک شیء از کلاسی تعریف شود، نمی‌توان به اعضای خصوصی آن دسترسی پیدا کرد. اگر سطح دسترسی برای یک عضو تعریف نشود، سطح دسترسی پیش فرض خصوصی است.

```

۱ class student {
۲     private:
۳         int average;
۴     public:
۵         string name;
۶         int getAverage() { return average ; }
۷ };
۸ student st; //st is an object of student class
۹ st.name = "ali";
۱۰ st.average = 17; // error : average is a private member

```

- معمولا بهتر است همه داده‌ها با سطح دسترسی خصوصی تعریف شوند. سپس توابعی با سطح دسترسی عمومی برای تغییر داده‌های عضو تعریف کنیم. بدین ترتیب از دستکاری شدن و تغییرات غیرمنتظره داده‌های عضو جلوگیری می‌کنیم.
- تنها توابعی را عمومی تعریف می‌کنیم که شیء نیاز به دسترسی به آنها را دارد. باقی توابع نیز با سطح دسترسی خصوصی تعریف می‌شوند.

```

۱ class student {
۲     private:
۳         string name;
۴         int average;
۵     public:
۶         void setName(string n) { name = n; }
۷         int getAverage() { return average ; }
۸ };
۹ student st;
۱۰ st.setName("ali");

```

- همانطور که برای انواع داده اصلی می‌توانستیم اشاره‌گر و آرایه تعریف کنیم، برای کلاس‌ها نیز می‌توانیم اشاره‌گرهایی به اشیاء و یا آرایه‌هایی از اشیاء بسازیم.

```
۱ student st;  
۲ student * stptr;  
۳ stptr = &st;  
۴ int s = stptr->getAverage();  
۵ student stgroup[40];  
۶ stgroup[0].setName("ali");
```

- هر کلاس می‌تواند یک سازنده¹ داشته باشد. وقتی شیئی از یک کلاس ساخته می‌شود، به طور خودکار سازنده فراخوانی می‌شود. نام سازنده با نام کلاس یکسان است و سازنده هیچ مقداری باز نمی‌گرداند. سازنده‌ای که مقدار ورودی ندارد، سازندهٔ پیش فرض² نامیده می‌شود.

```

۱ class student {
۲     private:
۳         string name;
۴         int age;
۵         int average;
۶     public:
۷         // default constructor
۸         student() { name = " "; age = 0; }
۹         // constructor with name and age as arguments
۱۰        student(string n, int a) : name(n), age(a) {}
۱۱ };

```

¹ constructor

² default constructor

- بنابراین در مثال قبل تابع سازنده در کلاس student سربارگذاری¹ شده است.
- با استفاده از سربارگذاری تابع²، توابعی که از نظر منطقی کار یکسانی را انجام می‌دهند و تنها تفاوت آنها در نوع ورودی‌های آنهاست را توسط یک نام واحد نامگذاری می‌کنیم.
- دو تابع که ورودی‌های آنها یکسان است و نوع خروجی آنها متفاوت است نمی‌توانند سربارگذاری شوند.

```

۱ bool print(int i);
۲ bool print(float f); // print is overloaded
۳ int print(int i); //error : functions that differ
۴     // only in their return type cannot be overloaded.

```

¹ overloaded

² function overloading

- سازنده پیش فرض را می‌توانیم همچنین با مقداره‌ی ورودی‌های یک سازنده غیرپیش فرض بسازیم.

```

۱ class student {
۲ private:
۳     string name;
۴     int age;
۵     int average;
۶ public:
۷     // this is both a default constructor and
۸     // a constructor with name and age as arguments
۹     student(string n=" ", int a=0) : name(n), age(a) {}
۱۰ };

```

- برای تخصیص حافظه به یک اشاره‌گر می‌توانیم از عملگر `new` استفاده کنیم. وقتی از عملگر `new` استفاده می‌کنیم، سازنده نیز فراخوانی می‌شود.
- همچنین از عملگر `delete` برای آزادسازی فضای حافظه استفاده می‌کنیم.
- در صورتی که یک اشاره‌گر به یک آرایه اشاره کند از `delete[]` برای آزادسازی حافظه استفاده می‌کنیم.

```

۱ // constructor is not called when malloc is used
۲ student *st1 = (student*)malloc(sizeof(student));
۳ // constructor is called when new is used
۴ student *st2 = new student;
۵ student *st3 = new student("ali", 21);
۶ student *st4 = new student[40];
۷ free(st1);
۸ delete st2;
۹ delete st3;
۱۰ delete[] st4;
```

- هر کلاس همچنین یک مخرب¹ دارد. وقتی شیئی از کلاس تخریب می‌شود، تابع مخرب فراخوانی می‌شود.
- ممکن است در سازنده یک کلاس حافظه‌ای را به طور پویا تخصیص دهیم. وقتی که شیئی از کلاس ساخته شود حافظه تخصیص داده می‌شود، ولی وقتی شیء از بین می‌رود، حافظه آزاد نمی‌شود. همچنین ممکن است در یکی از توابع کلاس حافظه‌ای به طور پویا تخصیص داده شود، که در آن صورت نیز، در صورتی که آن تابع فراخوانی شده باشد، باید حافظه را در مخرب آزاد کرد.
- بنابراین باید در مخرب یک کلاس همه حافظه‌هایی که تخصیص داده‌ایم را آزاد کنیم.

```

۱ class student {
۲     private:
۳         int * courses;
۴     public:
۵         student() { courses = new int[100]; }
۶         ~student() { delete[] courses; }
۷ };

```

¹ destructor

- در ابتدای برنامه همه متغیرهای عمومی در حافظه ساخته می‌شوند، بنابراین اگر شیئی از یک کلاس به طور عمومی تعریف شده باشد، در ابتدای برنامه سازنده آن فراخوانی می‌شود و در پایان برنامه مخرب آن فراخوانی می‌شود.
- همچنین هرگاه وارد تابعی می‌شویم، سازنده اشیای درون تابع فراخوانی می‌شوند و در پایان اجرای تابع مخرب آن اشیا (از آخر به اول) فراخوانی می‌شوند.
- برای اشیایی که با کلیدواژه static در یک تابع تعریف شده‌اند، در اولین فراخوانی تابع سازنده آنها فراخوانی می‌شود و مخرب آنها در پایان برنامه فراخوانی می‌شود.

- می‌توانیم یک شیء را با استفاده از کلیدواژه `const` به صورت ثابت تعریف کنیم.
- از آنجایی که یک شیء ثابت نمی‌تواند هیچ‌یک از ویژگی‌هایش را تغییر دهد، لذا هیچ تابعی هم بر روی آن قابل فراخوانی نیست، چرا که یک تابع ممکن است یکی از ویژگی‌های آن را تغییر دهد.
- برای اینکه یک شیء ثابت بتواند یک تابع را فراخوانی کند، در تعریف آن تابع باید از کلیدواژه `const` استفاده کرد.
- یک تابع ثابت نمی‌تواند ویژگی‌های کلاس را تغییر دهد.

```
۱ class student {
۲ private:
۳     string name;
۴     int birth_year;
۵ public:
۶     student(string n, int b) { name = n; bith_year = b; }
۷     string getName() const { return name; }
۸     int getBirthYear() { return birth_year; }
۹ };
۱۰ const student st("ali", 2000);
۱۱ cout << st.getName();
۱۲ cout << st.birthYear(); // error: constant object
۱۳                          // cannot call non-constant function
```

- داده‌های عضو یک کلاس نیز می‌توانند ثابت باشند. در اینصورت امکان تغییر آنها بعد از ساخته شدن شیء وجود نخواهد داشت.
- داده‌های ثابت یک کلاس را باید قبل از ورود به بدنهٔ سازنده در لیست مقداردهی به صورت `member1(var1), member2(var2), ...` مقداردهی اولیه کرد.

```
۱ class student {  
۲ private:  
۳     const int national_id;  
۴ public:  
۵     student(int id) : national_id(id) { }  
۶ };
```

- لیست مقداردهی در موارد دیگری نیز استفاده می‌شود. برای مثال وقتی یک داده عضو یک متغیر مرجع باشد که مقداردهی اولیه نیاز دارد و یا وقتی یکی از اعضای کلاس شیئی از کلاسی باشد که آن کلاس سازنده پیش فرض ندارد، در چنین مواردی نیز از لیست مقداردهی استفاده می‌کنیم.
- به طور کلی هر متغیر یا شیئی که مقدار دهی اولیه نیاز داشته باشد، باید در لیست مقداردهی قبل از ورود به بدنه سازنده مقدار اولیه آن تعیین شود.

- اعضای یک کلاس می‌توانند همچنین ایستا (static) باشند.
- ویژگی یک متغیر ایستا این است که یک بار مقداردهی اولیه می‌شود و حتی اگر از حوزه تعریف آن متغیر خارج شویم، متغیر مقدار خود را نگه می‌دارد.
- هر متغیر عضو یک کلاس برای یک شیء معین از آن کلاس در حافظه در فضای پشته¹ ساخته می‌شود. اما اگر متغیری ایستا باشد، آن متغیر در فضای داده‌ای¹ ساخته می‌شود و بنابراین مقدار آن برای همه اشیای ساخته شده از کلاس یکسان است.

¹ stack

¹ data segment

- فرض کنید می‌خواهیم متغیری در یک کلاس تعریف کنیم که تعداد اشیای ساخته شده از آن کلاس را بشمارد.
- اگر این متغیر به طور عادی یک عضو داده‌ای باشد، به ازای ساخته شدن هر شیء، متغیر برای آن شیء ساخته شده و مقداردهی می‌شود. برای حل این مشکل از متغیر ایستا استفاده می‌کنیم.

```
۱ // student.h
۲ class student {
۳ private:
۴     static int counter;
۵ public:
۶     student() { counter++; }
۷     static int getCounter() { return counter; }
۸     ~student() { counter--; }
۹ };
۱۰ // student.cpp
۱۱ int student::counter = 0;
```

- در اینجا متغیر ایستای counter تنها یک بار در یک فضای حافظه برای کلاس student ساخته می‌شود و حتی اگر هیچ شیئی از این کلاس وجود نداشته باشد، این متغیر ساخته شده است.
- حال می‌توانیم از این متغیر ایستا به صورت زیر استفاده کنیم.

```
۱ student::getCounter(); // counter = 0
۲ student st1;
۳ st1.getCounter(); // counter = 1
۴ student st2;
۵ st2.getCounter(); // counter = 2
۶ student st3;
۷ student::getCounter(); // counter = 3
```

- به هر یک از توابع کلاس می‌توانیم شیئی از همان کلاس به عنوان ورودی بدهیم و یا از یک تابع یک کلاس شیئی از همان کلاس را بازگردانیم.

```

۱ class student {
۲ private:
۳     string name;
۴     int age;
۵     int average;
۶ public:
۷     student(string n, int a) : name(n), age(a) {}
۸     bool compareAge(student s) { return (age >= s.age); }
۹     student * copy() { return new student(name, age); }
۱۰ };

```

- به هر یک از توابع کلاس می‌توانیم شیئی از همان کلاس به عنوان ورودی بدهیم و یا از یک تابع یک کلاس شیئی از همان کلاس را بازگردانیم.

```
۱ student st1, st2, *s3;  
۲ if (st1.compareAge(st2)) {  
۳     cout << "st1 is older than st2\n";  
۴ }  
۵ s3 = s1.copy();
```

- وقتی یک تابع از یک کلاس فراخوانی می‌شود، در واقع شیئی از آن کلاس ساخته شده، و شیء مورد نظر تابع کلاس را فراخوانی کرده است. در درون تعریف تابع نمی‌دانیم چه شیئی تابع را فراخوانی کرده است، اما زبان سی++ اشاره‌گری تعریف کرده و در اختیار برنامه‌نویس قرار داده است که با استفاده از آن اشاره‌گر به شیئی که تابع برای آن فراخوانی شده، دسترسی پیدا می‌کنیم. این اشاره‌گر `this` نام دارد.

```

۱ class student {
۲ public:
۳     student * compareAge(const student * st) {
۴         if (this->age >= st->age)
۵             return this;
۶         else
۷             return st;
۸     }
۹ };

```

- استفاده از `this->age` به جای `age` در اینجا به جهت خوانایی بهتر برنامه است.

- کلاس وکتور را در نظر بگیرید.

```
۱ class Vector {  
۲ public:  
۳     Vector(int s) :elem{new double[s]}, sz{s} {  
۴         for (int i=0; i!=s; ++i)  
۵             elem[i]=0;  
۶     }  
۷     void setElement(int i, double d) { elem[i] = d; }  
۸     ~Vector() { delete[] elem; }  
۹ private:  
۱۰     double* elem;  
۱۱     int sz;  
۱۲ };
```


- حال فرض کنید می‌خواهیم یک شیء از این کلاس بسازیم. پس از مقداردهی تعدادی از عناصر این وکتور می‌خواهیم وکتور دیگری بسازیم که کپی وکتور اولیه است.

```
۱
۲ Vector v1(10);
۳ v1.setElement(0,1.6);
۴ v1.setElement(1,3.14);
۵ Vector v2 = v1;
```

- در اینجا چه اتفاقی می‌افتد؟ آیا کامپایلر به طور خودکار اعضای v1 را در v2 کپی می‌کند؟ در اینصورت کامپایلر اشاره‌گر elem را چگونه کپی می‌کند؟

- وقتی از عملگر تساوی برای کپی کردن یک شیء در شیء دیگر استفاده می‌کنیم، کامپایلر تابعی به نام سازنده^۱ کپی^۱ را فراخوانی می‌کند.
- در صورتی که تابع سازنده^۲ کپی توسط کاربر تعریف نشده باشد، مانند توابع سازنده و مخرب، یک سازنده^۲ کپی پیش فرض^۲ توسط کامپایلر تعریف می‌شود.
- در تابع سازنده^۲ کپی پیش فرض، کامپایلر اعضای کلاس را یک به یک کپی می‌کند. پس سازنده^۲ کپی پیش فرض برای کلاس وکتور به صورت زیر خواهد بود.

```

۱ class Vector {
۲ public:
۳     Vector(Vector & v) { elem = v.elem; sz = v.sz; }
۴ private:
۵     double* elem;    int sz;
۶ };

```

^۱ copy constructor

^۲ default copy constructor

- اما منظور استفاده‌کننده کلاس وکتور کپی کردن تمام عناصر وکتور است و نه کپی کردن مقدار اشاره‌گر.

```
۱ Vector v2 = v1;
۲ // v1 and v2 both use the same pointer elem
۳ // if v2 changes its elements, the elements of v1 also changes.
```

- در کد بالا با استفاده از سازنده کپی پیش‌فرض v1 و v2 هر دو اشاره‌گری به نام elem دارند که به یک مکان واحد در حافظه اشاره می‌کند. همچنین با تخریب v1 حافظه تخصیص داده شده برای elem آزاد می‌شود و در هنگام تخریب v2 مکانی در حافظه که قبلاً آزاد شده باید دوباره آزاد شود که به یک خطای حین اجرا¹ برمی‌خوریم.

¹ run-time fault

- بنابراین برای استفاده از عملگر تساوی و کپی اشیاء، سازندهٔ کپی باید توسط برنامه‌نویس پیاده‌سازی شود.

```

۱  class Vector {
۲  public:
۳      Vector(Vector & v) {
۴          sz = v.sz;
۵          elem = new double[sz];
۶          for (int i=0; i<v.sz; i++)
۷              elem[i] = v.elem[i];
۸      }
۹  private:
۱۰     double* elem;   int sz;
۱۱ };
۱۲ Vector v2 = v1;
۱۳ // v1 and v2 have two different locations on memory
۱۴ // allocated for their elements

```

- به طور کلی سازنده کپی در سه موقعیت فراخوانی می‌شود.

۱. وقتی از عملگر تساوی برای کپی یک شیء در یک شیء دیگر استفاده می‌کنیم.

```
۱ Vector v1;  
۲ Vector v2 = v1;
```

۲. وقتی یک تابع فراخوانی با مقدار می‌شود و مقادیر ورودی تابع اشیایی از یک کلاس هستند.

```
۱ void print(Vector v) {  
۲     for (int i=0; i<v.size(); i++)  
۳         cout <<v.getElement(i);  
۴ }  
۵ Vector v1;  
۶ print(v1); // v = v1
```

۳. وقتی یک تابع شیئی از یک کلاس را بازمی‌گرداند.

```

۱ Vector larger(Vector &v1, Vector &v2) {
۲     if (v1.size() > v2.size()) return v1;
۳     else return v2;
۴ }
۵ Vector v1 {1,2}, v2 {3};
۶ Vector v3 = larger(v1,v2); // tmp = v1; v3 = tmp;

```

- فرض کنید می‌خواهیم یک نوع داده جدید برای ذخیره و محاسبات بر روی اعداد مختلط تعریف کنیم.

```
۱ class complex {  
۲     double re, im; // representation: two doubles  
۳ public:  
۴     // construct complex from two scalars  
۵     complex(double r, double i) :re{r}, im{i} {}  
۶     // construct complex from another complex  
۷     complex(complex& c) :re{c.real()}, im{c.imag()} {}  
۸     // construct complex from one scalar  
۹     complex(double r) :re{r}, im{0} {}  
۱۰    // default complex: {0,0}  
۱۱    complex() :re{0}, im{0} {}  
۱۲  
۱۳    ...
```

```
۱    ...
۲    double real() const { return re; }
۳    void real(double d) { re=d; }
۴    double imag() const { return im; }
۵    void imag(double d) { im=d; }
۶    ...
```

- در اینجا سه سازنده¹ برای کلاس complex تعریف کردیم.
- وقتی یک شیء از یک کلاس ساخته می‌شود، سازنده آن فراخوانی می‌شود.
- اگر در هنگام ساختن شیء سازنده آن مشخص نباشد، سازنده پیش فرض فراخوانی می‌شود. سازنده پیش فرض، سازنده‌ای است که هیچ مقدار ورودی ندارد.

```
۱ complex c1;  
۲ complex c2(3);  
۳ complex c3(2,4);  
۴ complex c4 {3};  
۵ complex c5 {2,4};  
۶ complex c6(c5);
```

¹ constructor

سربارگذاری عملگرها

- حال می‌خواهیم با استفاده از عملگرهای رایج در زبان سی++ دو عدد مختلط را با هم جمع یا از هم تفریق کنیم.
- برای این کار نیاز داریم عملگرها را برای اعمال بر روی اشیای کلاس تعریف کنیم.
- به تعریف یک عملگر برای یک کلاس سربارگذاری عملگر می‌گوییم. در سربارگذاری یک عملگر (تعریف مجدد یک عملگر برای یک کلاس) در واقع تابعی برای کلاس تعریف می‌کنیم که نام آن یک عملگر است.
- عملگرها می‌توانند یگانی¹ یا دوتایی² باشند.
- عملگرهای یگانی مانند !، --، ++ و عملگرهای دوتایی مانند &، ||، &&، *، /، -، +، !=، ==، | هستند.

¹ unary operator

² binary operator

- می‌توانیم عملگر + را نیز به صورت زیر تعریف کنیم.

```
۱    ...  
۲    complex operator+(const complex& b) {  
۳        return complex(re+b.re, im+b.im);  
۴    }  
۵    };
```

- سپس می‌توانیم از این عملگر به صورت زیر استفاده کنیم.

```
۱    complex c1 {1,2};  
۲    complex c2(3);  
۳    complex c3 = c1 + c2;  
۴    complex c4 {c2+complex{1,2.3}};
```

سربارگذاری عملگرها

- در تعریف عملگر جمع به صورت زیر، اگر عملگر بر روی یک متغیر ثابت فراخوانی شود، کامپایلر خطای کامپایل صادر خواهد کرد، زیرا یک تابع غیر ثابت بر روی یک شیء ثابت فراخوانی شده است.

```
۱    ...
۲    complex operator+(const complex& b) {
۳        return complex(re+b.re, im+b.im);
۴    }
۵ };
۶ const complex c1; complex c2;
۷ complex c3 = c1 + c2; // error : non-constant function is called
۸    // on constant object.
```

- در صورتی که می‌دانیم تابع `operator+` مقدار اعضای کلاس را تغییر نمی‌دهد. پس می‌توانیم آن را به صورت ثابت تعریف کنیم.

```
۱ complex operator+(const complex& b) const;
```

- وقتی یک عملگر بر روی یک شیء اعمال می‌شود، در واقع تابع تعریف شده برای آن عملگر فراخوانی می‌شود.
- پس کامپایلر به طور خودکار در هنگام کامپایل صورت نحوی یک عبارت حاوی عملگر را تشخیص داده، آن را ترجمه می‌کند.
- در مثال فوق `c1+c2` را کامپایلر به صورت `c1.operator+=(c2)` ترجمه می‌کند.

```
۱ c3 = c1+c2; // is equivalent to:  
۲ c3= c1.operator+=(c2);
```

- عملگرهای == و != را نیز برای این کلاس تعریف می‌کنیم.

```
۱    ...
۲    bool operator==(const complex& z) {
۳        return (re == z.re && im == z.im);
۴    }
۵    bool operator!=(const complex& z) {
۶        return (re != z.re || im != z.im);
۷    }
۸    };
```

- می‌توانیم عملگر + و == را به گونه‌ای تعریف کنیم که یک عدد مختلط را با یک عدد صحیح جمع کند یا با یک عدد صحیح مقایسه کند.

```
۱ complex operator+(const int& i) {  
۲     return complex(re+i, im);  
۳ }  
۴ bool operator==(const int& i) {  
۵     return (re == i && im == 0);  
۶ }
```

- سپس می‌توانیم از این عملگر به صورت زیر استفاده کنیم.

```
۱ complex c2 = c1 + 5;  
۲ if (c3==4) { ... }
```

- همچنین می‌توانیم یک عملگر را خارج از کلاس تعریف کنیم.

```
۱ complex operator-(complex a, complex b) { return a-=b; }
۲ bool operator==(complex a, complex b) { // equal
۳     return a.real()==b.real() && a.imag()==b.imag();
۴ }
۵ bool operator!=(complex a, complex b) {
۶     return !(a==b);
۷ }
```

- برای جمع یک عدد صحیح و یک عدد مختلط وقتی که عدد صحیح اولین عملوند و عدد مختلط دومین عملوند باشد، راهی جز تعریف تابع خارج از کلاس نداریم.

```
۱ complex operator+(const int& a, const complex& b) {  
۲     return complex(a+b.real(), b.image());  
۳ }  
۴  
۵ c3 = 1 + c2 // c3 = operator+(1, c2);
```

سربارگذاری عملگرها

- برای اینکه بتوانیم در تابعی که خارج از کلاس تعریف می‌شود، به اعضای خصوصی کلاس دسترسی پیدا کنیم، آن تابع را باید به صورت friend تعریف کنیم.

```
۱ class complex {
۲     ...
۳ public:
۴     friend complex operator+(const int& a, const complex& b);
۵ };
۶
۷ complex operator+(const int& a, const complex& b) {
۸     // we can access b.re and b.im, because
۹     // operator+(int, complex) is a friend function.
۱۰    return complex(a+b.re, b.im);
۱۱ }
۱۲
۱۳ c3 = 1 + c2 // c3 = operator+(1, c2);
```

سربارگذاری عملگرها

- توصیه می‌شود، توابع حتی الامکان به صورت friend تعریف نشوند و تنها در مواقع ضروری مانند مثال قبل توابع را به صورت friend تعریف کنیم.
- همچنین می‌توان یک کلاس را دوست یک کلاس تعریف کرد. بدین ترتیب وقتی کلاس F دوست کلاس A باشد همهٔ توابع کلاس F به اعضای خصوصی کلاس A دسترسی خواهند داشت.

```
۱ class A {  
۲     ...  
۳ public:  
۴     friend class F;  
۵ };  
۶ // now all functions of F can access  
۷ // private members of A.
```

- توصیه می‌شود کلاس‌ها نیز دوست تعریف نشوند، مگر اینکه واقعا نیازی به تعریف آن باشد و دلیلی برای آن وجود داشته باشد.

- می‌توانیم عملگرهای دیگری مانند += و -= را نیز برای این کلاس تعریف می‌کنیم.

```
۱    ...
۲    complex& operator+=(complex z) {
۳        re+=z.re; im+=z.im;
۴        return *this; // and return the result
۵    }
۶    complex& operator-=(complex z) {
۷        re-=z.re; im-=z.im;
۸        return *this;
۹    }
۱۰ };
```

- دلیل استفاده از `& complex` در مقدار خروجی تابع این است که اگر بخواهیم مقدار `c1 += c2` را محاسبه کنیم، باید پس از محاسبه مقدار `c2 += c1` متغیر `c1` را بازگردانیم تا با مقدار `c3` جمع شود.
- اگر نوع خروجی تابع به جای `& complex` نوع `complex` باشد، آنگاه پس از محاسبه مقدار `c1 += c2` یک متغیر موقت بازگردانده می‌شود و مقدار آن متغیر موقت با `c3` جمع می‌شود.

```
۱ complex operator+=(complex z) {  
۲     re+=z.re; im+=z.im;  
۳     return *this;  
۴ }  
۵ (c1 += c2) += c3 // this is equal to:  
۶ // (tmp=c1.operator+=(c2)).operator+=(c3), so tmp is updated.
```

- اما اگر نوع خروجی تابع `& complex` باشد، آنگاه پس از محاسبه مقدار `c2 += c1` متغیر `c1` بازگردانده می‌شود و مقدار آن با `c3` جمع می‌شود.

```
1 complex& operator+=(complex z) {  
2     re+=z.re; im+=z.im;  
3     return *this;  
4 }  
5 (c1 += c2) += c3 // this is equal to:  
6 // (c1=c1.operator+=(c2)).operator+=(c3), so c1 is updated.  
7  
8 c1 += c2 += c3 // this is equal to:  
9 // (c1.operator+=(c2.operator+=(c3))  
10 // because += associativity is right to left
```

سربارگذاری عملگرها

- در سربارگذاری عملگرها، تنها می‌توان عملگرهای موجود در زبان سی++ را سربارگذاری کرد، و نمی‌توان اولویت این عملگرها را تغییر داد.
- همچنین عملگرهای زیر را نمی‌توان سربارگذاری کرد:
- عملگر . که برای دسترسی به اعضای کلاس به کار می‌رود.
- : ؟ که برای انشعاب شرطی به کار می‌رود.
- عملگر :: که برای تفکیک حوزه اشیا و کلاس‌ها به کار می‌رود.
- و عملگر * که برای دسترسی به اعضای کلاس به کار می‌رود.

```
۱ class Vector { public : int size; ... };  
۲ int Vector::* s = &Vector::size;  
۳ Vector v;  
۴ cout << (v.*s);
```

- عملگریگانی ! را برای کلاس complex اینگونه تعریف می‌کنیم.

```
۱ class complex {  
۲ public :  
۳     complex operator!() {  
۴         return complex(-re, -im);  
۵     }  
۶ };
```

- عملگرهای یگانی ++ و -- می‌توانند به دو صورت پیشوند¹ و پسوند² استفاده شود.
- برای سربارگذاری چنین عملگرهایی که هم به صورت پیشوند و هم به صورت پسوند مورد استفاده قرار می‌گیرند، کامپایلر قرارداد کرده است که در حالت پیشوند تابع سربارگذاری عملگر بدون ورودی است:
`operator++()` , `operator--()`
- و در حالت پسوند تابع سربارگذاری عملگر یک ورودی عدد صحیح می‌گیرد: `operator++(int)` , `operator--()`

¹ prefix

² postfix

```
1 class complex {
2 public :
3     // prefix ++
4     complex& operator++() {
5         re++; // first increment and then return
6         return *this;
7     }
8     // postfix ++
9     complex operator++(int) {
10         complex res(*this);
11         re++;
12         return res; // return the current result and increment
13     }
14 };
```

- فرض کنید می‌خواهیم عملگر << را برای کلاس اعداد مختلط سربارگذاری کنیم تا اعداد مختلط را توسط این عملگر بر روی خروجی استاندارد چاپ کنیم.
- این عملگر یک عملگر دوتایی است که ورودی اول آن یک شیء از کلاس ostream و ورودی دوم آن یک شیء است که در اینجا می‌خواهیم آن را از کلاس complex تعریف کنیم.
- از آنجایی که ورودی اول این عملگر از کلاس ostream است و ما به این کلاس دسترسی نداریم، بنابراین این تابع را باید خارج از کلاس complex تعریف کنیم، زیرا اگر آن را در کلاس complex تعریف کنیم، ورودی آن از نوع اعداد مختلط خواهد بود.

```
۱ ostream& operator<<(ostream &out, const complex& c) {  
۲     // we are modifying out, so it cannot be constant  
۳     out << c.re << showpos << c.im << "i";  
۴     return out;  
۵ }
```

- همچنین برای دسترسی به اعضای خصوصی کلاس `complex` در تابع `<<operator`، آن را به عنوان یک تابع دوست در کلاس `complex` تعریف می‌کنیم.

```
۱ class complex {  
۲ public:  
۳     ...  
۴     friend ostream& operator<<(ostream &, complex&);  
۵ }
```

سربارگذاری عملگرها

- به طور مشابه عملگر >> را نیز برای کلاس اعداد مختلط سربارگذاری می‌کنیم تا اعداد مختلط را توسط این عملگر از روی ورودی استاندارد دریافت کنیم.
- این عملگر یک عملگر دوتایی است که ورودی اول آن یک شیء از کلاس istream و ورودی دوم آن یک شیء است که در اینجا می‌خواهیم آن را از کلاس complex تعریف کنیم.

```
۱ class complex {
۲ public:
۳     ...
۴     friend istream& operator>>(istream &, complex&);
۵ }
۶ istream& operator>>(istream &in, complex& c) {
۷     in >> c.re;
۸     in >> c.im
۹     return in;
۱۰ }
```

- عملگرهای تبدیل نوع¹ برای تبدیل یک نوع به نوع دیگر استفاده می‌شوند.
- حال فرض کنید می‌خواهیم نوع عدد مختلط را با استفاده از عملگر تبدیل نوع `double()` به یک عدد اعشاری تبدیل کنیم و منظور ما از این تبدیل استخراج قسمت حقیقی عدد مختلط است.

¹ type casting

- عملگر `double()` را باید برای کلاس اعداد مختلط به صورت زیر تعریف کنیم.

```
۱ class complex {  
۲ public:  
۳     ...  
۴     operator double() {  
۵         return re;  
۶     }  
۷ }  
۸ complex c(1,2);  
۹ double d1 = double(c);  
۱۰ double d2 = (double)c;  
۱۱ double d3 = c;
```

- در صورتی که بخواهیم کاربر را مجبور کنیم که از عملگر تبدیل نوع استفاده کند و کاربر این تبدیل را به کامپایلر واگذار نکند، از کلیدواژه `explicit` استفاده می‌کنیم.

```
۱ class complex {
۲ public:
۳     ...
۴     explicit operator double() {
۵         return re;
۶     }
۷ }
۸ complex c(1,2);
۹ double d1 = double(c);
۱۰ double d2 = (double)c;
۱۱ double d3 = c; // error
```

- به طریق مشابه می‌توانیم عملگر تبدیل نوع string را برای نوع مختلط تعریف کنیم.

```
۱ class complex {  
۲ public:  
۳     ...  
۴     operator string() {  
۵         string res = to_string(re);  
۶         if (im > 0) res+= "+";  
۷         res += (to_string(im) + "i");  
۸         return res;  
۹     }  
۱۰ }  
۱۱ complex c(1,2);  
۱۲ string s = (string)c;
```

- در صورتی که بخواهیم با استفاده از عملگر تبدیل نوع، یک نوع را به نوع کلاس خود تبدیل کنیم، باید از سازنده استفاده کنیم.
- برای مثال فرض کنید می‌خواهیم یک عدد اعشاری را با استفاده از عملگر `complex()` به یک عدد مختلط تبدیل کنیم.
- در این صورت می‌توانیم بنویسیم `complex(5.6)` و یا `complex(5.6)`.

- در این موارد سازنده کلاس complex اگر با ورودی عدد اعشاری تعریف شده باشد، فراخوانی می‌شود.

```
۱ class complex {
۲ public:
۳     ...
۴     complex(double r) :re{r}, im{0} {}
۵ };
۶ complex c1(1,2);
۷ // using the constructor,
۸ // 0 is implicitly type-casted to complex
۹ if (c1 == 0) { ... }
۱۰ complex c2;
۱۱ // using the constructor,
۱۲ // 5.6 is explicitly type-casted to complex
۱۳ c2 = c1 + (complex)5.6;
```

- اگر بخواهیم تبدیل یک نوع به نوع کلاس مورد نظر ما به طور خودکار توسط کامپایلر انجام نشود، از کلیدواژه `explicit` استفاده می‌کنیم.

```
۱ class complex {  
۲ public:  
۳     ...  
۴     explicit complex(double r) :re{r}, im{0} {}  
۵ };  
۶ complex c1 = 10; // error
```

- پس برای جمع دو عدد اعشاری و مختلط اکنون دو راه وجود دارد. اول اینکه از عملگر جمع برای جمع دو عدد اعشاری و مختلط استفاده کنیم و دوم اینکه عدد اعشاری را با استفاده از سازنده به مختلط تبدیل کنیم و با استفاده از عملگر جمع برای دو عدد مختلط آن دو عدد را با هم جمع کنیم.

```
۱ // 1. use operator+(double, complex)
۲ // complex = double + complex
۳ c2 = 1.6 + c1;
۴ // 2. use constructor to cast double to complex
۵ // complex = (complex)double + complex
۶ c2 = (complex)1.6 + c1;
```

- همچنین برای برای جمع دو عدد اعشاری و مختلط و بازگرداندن یک عدد اعشاری سه راه وجود دارد. اول اینکه از عملگر جمع برای جمع دو عدد اعشاری و مختلط استفاده کنیم و دوم اینکه عدد اعشاری را با استفاده از سازنده به مختلط تبدیل کنیم و با استفاده از عملگر جمع برای دو عدد مختلط آن دو عدد را با هم جمع کنیم.

```
۱ // 1. use operator+(double, complex) and operator double()
۲ // double = (double)(double + complex)
۳ d = (double)(1.6 + c1);
۴ // 2. use operator double()
۵ // double = (double + (double)complex)
۶ d = 1.6 + (double)c1;
۷ // 3. use constructor to cast double to complex and operator double()
۸ // double = (double)((complex)double + complex)
۹ d = (double)((complex)1.6 + c1);
```

- حال فرض کنید می‌خواهیم یک عدد اعشاری بزرگ (long double) را با یک عدد مختلط جمع کنیم. در این صورت کامپایلر دو راه پیش رو دارد. می‌تواند عدد اعشاری بزرگ را به اعشاری تبدیل کند و سپس آن را با عدد مختلط جمع کند. و یا می‌تواند عدد مختلط را به اعشاری تبدیل کند و سپس آن را با عدد اعشاری بزرگ جمع کند. کامپایلر در این مورد نمی‌تواند تصمیم بگیرد و بنابراین پیام خطا ارسال می‌کند.

```
۱ d = 1.6L + c1; // error : use of operator + is ambiguous
۲ d = (double)1.6L + c1; // operator+(double,complex) is defined
۳ d = 1.6L + (double)c1; // operator+(long double, double) is defined
```

- برای چاپ یک عدد مختلط نیز دو راه حل وجود دارد. یا از عملگر درج برای چاپ یک عدد مختلط استفاده کنیم و یا یک عدد مختلط را به یک رشته تبدیل و سپس رشته را چاپ کنیم.

```
۱ // 1. use operator<<(ostream, complex)
۲ cout << c1;
۳ // 2. use operator string()
۴ cout << (string)c1;
```

سربارگذاری عملگرها

- حال فرض کنید می‌خواهیم عملگر زیرنویس ¹ [] را برای کلاس وکتور سربارگذاری کنیم، به گونه‌ای که با فراخوانی این عملگر بر روی شیء یک کلاس، یکی از اعضای وکتور متناسب با مقدار درون عملگر بازگردانده شود.
- بنابراین می‌خواهیم از این عملگر به صورت زیر استفاده کنیم.

```
1 class Vector {  
2     double* elem;  
3     int sz;  
4 public:  
5     Vector(int s) :elem{new double[s]}, sz{s} { }  
6 };  
7 Vector v(10);  
8 v[0] = 6;  
9 cout << v[0];
```

¹ subscript

- عملگر زیرنویس [] را به صورت زیر سربارگذاری می‌کنیم.

```
۱ class Vector {  
۲ public:  
۳     double& operator[](int i) { return elem[i]; }  
۴     ...  
۵ };  
۶ Vector v(10);  
۷ v[0] = 6;  
۸ cout << v[0];
```

سربارگذاری عملگرها

- حال فرض کنید می‌خواهیم از عملگر زیرنویس برای یک شیء ثابت استفاده کنیم.

```
۱ class Vector {  
۲ public:  
۳     double& operator[](int i) { return elem[i]; }  
۴     ...  
۵ };  
۶ const Vector v(10);  
۷ cout << v[0]; // error
```

- از آنجایی که شیء v یک شیء ثابت است، نمی‌توانیم یک تابع غیرثابت (در اینجا عملگر زیرنویس) را برای آن فراخوانی کنیم.

- از طرفی اگر تابع سربارگذاری عملگر زیرنویس را ثابت تعریف کنیم، امکان مقدار دهی عناصر وکتور به صورت $v[0] = 6$ را نخواهیم داشت.

- بنابراین باید تابع عملگر زیرنویس را به صورت ثابت و غیرثابت سربارگذاری کنیم و کامپایلر نیز این اجازه را به ما می‌دهد، گرچه ورودی هر دو تابع یکسان است.

```
۱ class Vector {  
۲ public:  
۳     double& operator[](int i) { return elem[i]; }  
۴     double operator[](int i) const { return elem[i]; }  
۵     ...  
۶ };  
۷ const Vector v1(10);  
۸ cout << v1[0];  
۹ Vector v2(10);  
۱۰ v2[0] = 6;  
۱۱ cout << v2[0]
```

- می‌توانیم عملگر -> را نیز برای این کلاس تعریف می‌کنیم.

```
۱ class Element {  
۲ public :  
۳     double e[100];  
۴ };  
۵ class Vector {  
۶ private:  
۷     Element * element;  
۸ public:  
۹     Vector() : element{new Element()} {}  
۱۰     Element * operator->() { return element; }  
۱۱ };  
۱۲ Vector v;  
۱۳ cout << v->e[2];
```

- عملگر = مانند توابع سازنده و سازنده‌کپی به طور پیش فرض برای همه کلاس‌ها تعریف شده است.
- در تعریف پیش فرض این تابع همه مقادیر اعضای کلاس از شیء مبدأ به شیء مقصد کپی می‌شوند.
- در مواردی که اعضای کلاس شامل اشاره‌گر هستند و نمی‌خواهیم اعضای کلاس عیناً کپی شوند، بلکه می‌خواهیم برای اشاره‌گرها حافظه تخصیص بدهیم، عملگر مساوی را تعریف می‌کنیم.

```
1 class Vector {
2 public:
3     Vector& operator=(const Vector & v) {
4         sz = v.sz;
5         elem = new double[sz];
6         for (int i=0; i<v.sz; i++)
7             elem[i] = v.elem[i];
8         return *this;
9     }
10 private:
11     double* elem;   int sz;
12 };
13 Vector v1;
14 Vector v2 = v1; // the copy constructor is called.
15 Vector v3;
16 v3 = v2; // v3.operator=(v2) is called.
```


- عملگر فراخوانی تابع () را نیز می‌توانیم سربارگذاری کنیم.
- با شیئی که از این کلاس ساخته می‌شود، می‌توان مانند یک تابع رفتار کرده و آن را فراخوانی کرد یا آن را مانند اشاره‌گر به تابع به عنوان ورودی به توابع دیگر وارد کرد.
- به چنین اشیایی اشیای تابعی¹ یا فانکتور² نیز می‌گوییم.

¹ function object

² functor

```

1 class Linear {
2 private:
3     double a, b;
4 public:
5     Linear(double _a, double _b) : a(_a), b(_b) {}
6     double operator()(double x) const {
7         return a * x + b;
8     }
9 };
10 Linear f{2, 1}; // Represents function 2x + 1.
11 Linear g{-1, 0}; // Represents function -x.
12 // f and g are objects that can be used like a function.
13 double f_0 = f(0);
14 double f_1 = f(1);
15 double g_0 = g(0);

```

- در کلاس صف، برای کار راحت تر با صف می توانیم عملگرهای زیر را سربارگذاری کنیم.

```
۱ class Queue {  
۲ public:  
۳     Queue& operator,(const int& data) { push(data); return *this; }  
۴     int operator()() { return pop(); }  
۵     bool operator!() { return !empty(); }  
۶ }  
۷ Queue q1(100);  
۸ q1,2,3,7,8,9; // push 2,3,7,8,9 into the queue  
۹ while(!q1) { cout << q1() << " "; } // pop from the queue
```

وراثت و چندریختی

- وقتی چند کلاس ویژگی‌ها و رفتارهای مشترک دارند، باید آن ویژگی‌ها و رفتارها را برای همه آن کلاس‌ها تعریف کرد.
- تعریف ویژگی‌ها و رفتارهای واحد در چند کلاس متفاوت معایبی دارد، از جمله اینکه هزینه پیاده‌سازی افزایش می‌یابد، و همچنین اعمال تغییرات در پیاده‌سازی سخت‌تر می‌شود، چرا که اگر یکی از رفتارهای مشترک تغییر کند، همه 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 وراثت¹ 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 بقیه کلاس‌ها می‌توانند آن ویژگی‌ها و رفتارها را از آن کلاس به ارث ببرند.

¹ inheritance

- کلاسی که ویژگی‌ها و رفتارهای مشترک را تعریف می‌کند کلاس پدر¹ یا کلاس مافوق یا کلاس پایه² و کلاسی که ویژگی‌ها و رفتارهای مشترک را به ارث می‌برد، کلاس فرزند³ یا زیرکلاس یا کلاس مشتق شده⁴ نامیده می‌شود. چندین کلاس می‌توانند از یک کلاس پایه مشتق شوند.
- برای مثال در یک سامانه 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

- 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 فعلاً 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 };
```

- 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، همه فرزندان به آن اعضا دسترسی مستقیم دارند، اما استفاده‌کنندگان دیگر کلاس پدر به این اعضا دسترسی مستقیم ندارند.

- در مثال قبل گفتیم کلاس دانشجو از کلاس شخص به ارث می برد. پس می توانیم 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 };
```

- در این مثال کلاس دانشجو همه ویژگی‌ها و رفتارهای کلاس شخص را دارد، بنابراین وقتی یک شیء از کلاس دانشجو ساخته می‌شود، این شیء در حافظه همه 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 };
```

- وقتی یک شیء از کلاس دانشجو ساخته شود، این شیء به همه اعضای عمومی کلاس دانشجو و کلاس شخص دسترسی دارد.
- وقتی یک شیء فرزند ساخته می‌شود، ابتدا سازنده پیش فرض کلاس پدر، و سپس سازنده پیش فرض کلاس فرزند فراخوانی می‌شود. در صورتی که سازنده پیش فرض وجود نداشته باشد، سازنده غیرپیش فرض در کلاس فرزند باید مقادیر اولیه برای سازنده غیرپیش فرض در کلاس پدر را (در لیست مقادیردهی اولیه) تعیین کند تا سازنده غیرپیش فرض 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 سازنده پدر مقادیردهی اولیه شد، سازنده 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 سازنده پیش فرض داشته باشند، ابتدا سازنده پیش فرض کلاس پدر و سپس کلاس فرزند، فراخوانی می‌شوند. اما اگر سازنده پیش فرض 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 };
```

- کلاس فرزند می‌تواند رفتار کلاس پدر خود را لغو¹ 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 چندگانه¹ در زبان سی++ امکان پذیر است.
- یک کلاس می تواند به طور همزمان از دو کلاس به ارث ببرد. به طور مثال یک تدریس یار هم یک دانشجوی تحصیلات تکمیلی است و هم یک محقق. پس کلاس تدریس یار (assistant) هم از کلاس دانشجو (student) و هم از کلاس محقق (researcher) به ارث می برد.

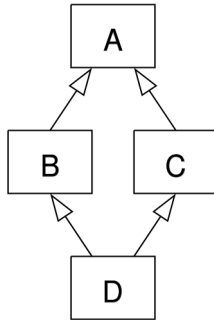
```
۱ class student {  
۲     protected : int average;  
۳ };  
۴ class researcher {  
۵     protected: string subject;  
۶ };  
۷ class assistant : public student, public researcher {  
۸     // assistant has access to both  
۹     // student and researcher class members  
۱۰ };
```

¹ multiple inheritance

- وراثت 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.
```

- به این مشکل، مشکل لوزی¹ 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

- یکی از راه‌های حل مشکل لوزی این است که استفاده‌کننده 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 کلیدواژه virtual از کلاس A به ارث ببرند. به این وراثت، وراثت مجازی¹ 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 attribute

```

¹ virtual inheritance

- قوانین دسترسی در انواع وراثت عمومی، خصوصی، و حفاظت شده 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 };

```

- 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 };

```

- از وراثت خصوصی در کلاس A وقتی استفاده می‌کنیم که به همه 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 رابطه ترکیب (یعنی A در ترکیب خود دارای B است)، از رابطه وراثت (یعنی A از B به ارث برده است) استفاده می‌کنیم.

- 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 حفاظت‌شده، فرزندان A نیز می‌توانند به اعضای B به طور مستقیم دسترسی پیدا کنند. این نوع وراثت نیز به ندرت استفاده می‌شود.

- کلاس فرزند می‌تواند سطح دسترسی به اعضای کلاس پدر خود را با استفاده از کلیدواژه `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 };
```

- 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.

- در نظریهٔ زبان‌های برنامه‌نویسی، چندریختی¹ به معنای فراهم کردن یک رابط² برای موجوداتی از نوع‌های متفاوت است.
- در زبان سی++ نیز چندریختی قابلیت است برای فراهم کردن امکان یک فراخوانی واحد برای اشیایی از کلاس‌های متفاوت.

¹ polymorphism

² interface

- 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 همه اعضای کلاس پدر مقادیر مورد نیاز خود را دریافت می‌کنند هیچ مشکلی به وجود نخواهد آمد.

- 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.

- 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.

- 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.

```
\ 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.

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

```
\ 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.

- 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:

```
۱ circle& operator=(const shape& s) {  
۲ // copy members of shape s into members of this circle  
۳ // and initialize other members of this circle with 0 and ""
```

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

- پس اگر کلاس پدر تابعی را تعریف کرده باشد، اشیایی از کلاس فرزند را در یک شیء از کلاس پدرکی 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.

```

- 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);

```

- 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؟

- 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.

- حال سناریوی زیر را در نظر بگیرید.
- می خواهیم آرایه‌ای از اشیایی تشکیل دهیم که همه فرزندان 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 نحوه محاسبه مساحت برای هر کدام از آنها متفاوت است. حال آرایه‌ای از اشکال متفاوت داریم و می‌خواهیم تابع محاسبه مساحت را برای همه اعضای این لیست محاسبه کنیم.

- برای همه 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.

```

- 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 قابلیت 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 فرزندان 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 کلیدواژه 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.

- تابعی که توسط کلمهٔ virtual تعریف شده است، و می تواند با یک نام واحد بسته به کلاسی که آن را فراخوانی می کند شکل ها یا ریخت های متفاوت داشته باشد، یک تابع چندریخت¹ نامیده می شود.
- کلاسی که یک تابع چندریخت را تعریف کند یا به ارث ببرد، یک کلاس چندریخت² نامیده می شود.

¹ polymorphic function

² polymorphic class

- برای همه 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.

```


- 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.

```
۱ shape shp; circle crc;  
۲ shp = crc; // here we copy members of crc into members of shp  
۳ shp.calcArea(); // here we call calcArea of  
۴ // an object of shape class,  
۵ // so shape::calcArea() is called,  
۶ // even if shape::calcArea is virtual.
```

- وقتی شیئی از کلاس پدر را با شیئی از کلاس فرزند مقاردهی 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.

- 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 همه 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 مجموعه اشکال اضافه شود، تابع info باید برای شکل جدید پیاده‌سازی شود.

- 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.

– 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 }
```

- 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.

- یک تابع را می‌توان به صورت مجازی خالص¹ 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 کلیدواژه `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

- 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 دسته کلاس‌های انضمامی¹، و کلاس‌های انتزاعی²، 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 داده 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.

- 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.

```
۱ class shape {  
۲ public: virtual ~shape() { }  
۳ };  
۴ shape* shp;  
۵ circle* crc = new circle;  
۶ shp = crc;  
۷ delete shp;  
۸ // first circle::~~circle and then shape::~~shape is called.
```

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

- 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 }
```

- 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 }
```

- در صورتی که کلاس پدر توابع مجازی نداشته باشد، و در نتیجه کلاس پدر چندریخت¹ نباشد، نمی‌توان از تابع `dynamic_cast` استفاده 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 * s = new circle;  
۸ circle * c = dynamic_cast<circle*>(s);  
۹ // error : shape is not polymorphic
```

¹ polymorphic

- انتخاب توابع چندریخت به طور پویا در زمان اجرا صورت می‌گیرد. به عبارت دیگر تنها در زمان اجرا¹ 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 کامپایل² 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

- 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 پویا¹ استفاده می‌کند.
- در مقابل سازوکار پیوستن پویا، پیوستن ایستا² 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 شماره دانشجویی، اطلاعات دانشجو را بیابد، تابع سربارگذاری شده `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 دقیقاً 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.

- 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.
- نحوه 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 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.

- 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.

- حال برنامه زیر را در نظر بگیرید.

```

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

```

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

```

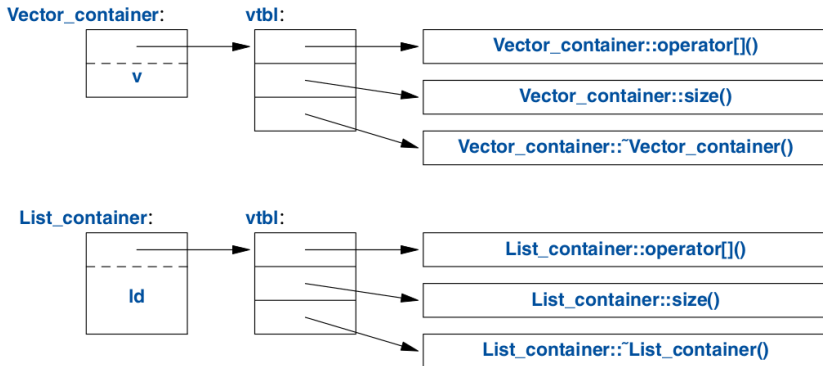
۱ A aobj; B bobj; A * aptr;
۲ // aobj.vptr and bobj.vptr are added.
۳ // aobj.vtable contains addresses of f and g implemented in A.
۴ // bobj.vtable contains the address of f implemented in B
۵ // and the address of g implemented in A.
۶ if (/*user chooses A*/) aptr = &aobj;
۷ else if (/*user chooses B*/) aptr = &bobj;
۸ aptr->f(); // => aptr->vptr->vtable[f]();
۹ aptr->g(); //=> aptr->vptr->vtable[g]();

```

```
۱ A aobj; B bobj; A * aptr;  
۲ // aobj.vptr and bobj.vptr are added.  
۳ // aobj.vtable contains addresses of f and g implemented in A.  
۴ // bobj.vtable contains the address of f implemented in B  
۵ // and the address of g implemented in A.  
۶ if (/*user chooses A*/) aptr = &aobj;  
۷ else if (/*user chooses B*/) aptr = &bobj;  
۸ aptr->f(); // => aptr->vptr->vtable[f]();  
۹ 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.

- پس اشیای چندریخت که از کلاس‌های انتزاعی به ارث برده‌اند، جدولی به نام جدول تابع مجازی¹ یا vtbl در حافظه نگهداری می‌کنند که در آن جدول آدرس توابعی که باید فراخوانی شوند، یادداشت شده است.



¹ virtual function table

- در دیباگر gdb می‌توان جدول vtable برای اشاره‌گر ptr را با دستور `-exec info vtbl ptr` مشاهده کرد.

```
-exec info vtbl aptr
```

```
vtable for 'A' @ 0x55555556b8f8 (subobject @ 0x7fffffffdf0f0):
```

```
[0]: 0x55555556370c <A::f(>
```

```
[1]: 0x55555556371c <A::g(>
```

```
-exec info vtbl aobj
```

```
vtable for 'A' @ 0x55555556b8f8 (subobject @ 0x7fffffffdf0f0):
```

```
[0]: 0x55555556370c <A::f(>
```

```
[1]: 0x55555556371c <A::g(>
```

```
-exec info vtbl bobj
```

```
vtable for 'B' @ 0x55555556b8d8 (subobject @ 0x7fffffffdf100):
```

```
[0]: 0x55555556372c <B::f(>
```

```
[1]: 0x55555556371c <A::g(>
```

- رابط کلاس¹ در واقع تعریف کلاس است که در فایل سریتیر یا فایل هدر² آمده است.
- اما کلاس رابط³ 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

- فرض کنید می‌خواهیم از مفهوم وکتور یا حامل¹ در برنامه‌های خود استفاده کنیم. یک حامل در واقع ظرفی² است برای نگهداری تعدادی عنصر³. 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

```
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 }
```


- تعریف تابع `size()` با کلیدواژه `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

```

- 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 اندازه فضای تخصیص داده شده را افزایش می‌دهد.

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

```
۱ // initialize with a list
۲ Vector(std::initializer_list<double> lst) {
۳     elem = new double[lst.size()];
۴     sz = lst.size();
۵     cur = sz;
۶     std::copy(lst.begin(), lst.end(), elem);
۷ }
```

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

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

- 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 ظرف¹ یک مفهوم انتزاعی است. می‌دانیم یک وکتور یک ظرف است و همچنین یک صف² هم یک ظرف است. ظرف خصوصیات 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

- پس یک ظرف چنین تعریف می‌شود.

```

۱ class Container {
۲ public:
۳     virtual double& operator[] (int) = 0;
۴     virtual int size() const = 0;
۵     virtual ~Container() {}
۶ };

```

- واژه 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 خالص¹ است و حتما باید توسط کلاس‌هایی که از این کلاس ارث برده‌اند پیاده‌سازی شود.

¹ pure virtual

- 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.

```

۱ Container c;
۲ // error : there can be no objects of an abstract class
۳ Container* p = new Vector_container(10);
۴ // 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.

```

۱ void use(Container& c) {
۲     const int sz = c.size();
۳     for (int i=0; i!=sz; ++i)
۴         cout << c[i] << '\n';
۵ }

```

- 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 };
```

- کلیدواژه `public`: بدین معناست که `Vector_container` از `Container` به ارث می‌برد و به عبارت دیگر وکتور یکی از انواع ظرف است.
- ظرف وکتور یک زیرکلاس¹ از کلاس ظرف است و کلاس ظرف، کلاس مافوق یا ابرکلاس² 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

- 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 نقطه مرکز خواهد داشت که توسط تابع center به دست می‌آید و هر شکل را می‌توان جابجا کرد و نقطه 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 درجه معین دوران داد.

- 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 v's elements by angle degrees
۲ void rotate_all(Shape* v[], int size, int angle) {
۳     for (int i=0; i<size; i++)
۴         v[i]->rotate(angle);
۵ }
```

- 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 کلیدواژه `override` متمایز می‌کنیم.
- این کلیدواژه برای خوانایی بهتر کد به کار می‌رود.
- همچنین با استفاده از این کلیدواژه کامپایلر بررسی می‌کند که تابع مورد نظر در کلاس پدر به صورت مجازی تعریف شده باشد. در غیراینصورت کامپایلر پیام خطا ارسال می‌کند.

- معمولا قبل از پیاده‌سازی یک سیستم نرم‌افزاری طراحی برای آن آماده می‌کنیم. این طرح می‌تواند شامل مستندات و اشکال و فلوچارت‌هایی باشد که نحوه 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 همه خواسته‌ها بر طبق نیازها برآورده شده‌اند یا خیر.
- زبان طراحی یکپارچه¹ یا یوام‌ال زبانی است استاندارد که برای مستند کردن و توصیف کردن یک سیستم نرم‌افزاری استفاده می‌شود. گروه مدیریت اشیا² این زبان را در سال ۱۹۹۷ برای طراحی سیستم‌های پیچیده نرم‌افزاری ارائه کرده است.

¹ Unified Modeling Language (UML)

² Object Management Group (OMG)

- زبان مدلسازی یکپارچه تعدادی نمودار استاندارد برای طراحی سیستم ارائه می‌کند. این نمودارها شامل نمودارهای ساختاری¹ و نمودارهای رفتاری² می‌شوند.
- نمودارهای ساختاری شامل نمودار کلاس³، نمودار اشیا⁴، نمودار اجزا⁵، و نمودار استقرار⁶ می‌شوند.
- نمودارهای رفتاری شامل نمودار فعالیت⁷، نمودار توالی⁸، نمودار کارخواست⁹، و نمودار حالت¹⁰ 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

- در مبحث طراحی شیءگرای سیستم‌ها همه این نمودارها شرح داده می‌شوند. در اینجا تنها به نمودار کلاس می‌پردازیم.
- وقتی یک سیستم را توصیف می‌کنیم، معمولاً 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 عمومی با علامت +، اعضای خصوصی با علامت - و اعضای حفاظت‌شده با علامت # مشخص می‌شوند.

class name
- private_attribute : type
protected_attribute : type
+ public_attribute : type
- private_function() : type
protected_function() : type
+ public_function() : type

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

person
name : string
id : int
birth_year : int
+ setName(name : string)
+ getName() : string

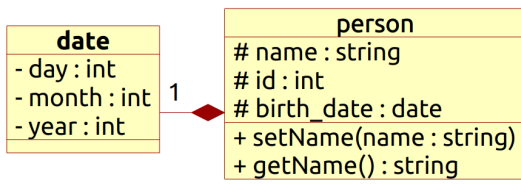
- چندین نوع رابطه برای توصیف رابطه بین کلاس‌ها وجود دارد. یکی از این رابطه‌ها رابطه ترکیب¹ و یکی دیگر رابطه تجمیع² می‌باشند.
- در هر دو رابطه ترکیب و تجمیع می‌گوییم یک کلاس دارای عضوی یا اعضای از کلاس دیگر است. به طور مثال یک شخص یک تاریخ تولد از کلاس تاریخ دارد یا یک شخص یک مسکن از کلاس مسکن دارد.
- تفاوت ترکیب و تجمیع در این است که در رابطه 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

- رابطه ترکیب به صورت زیر نمایش داده می‌شود. وقتی کلاس A یک عضو از کلاس B دارد، در طرف کلاس A از یک لوزی توپر¹ استفاده می‌کنیم.
- همچنین کثرت² یک رابطه را می‌توانیم در یک طرف یا در دو طرف آن نشان دهیم، یعنی بگوییم کلاس A چند عضو از کلاس B دارد.
- مثلاً کلاس شخص یک عضو از کلاس تاریخ دارد که تاریخ تولد او را نشان می‌دهد.

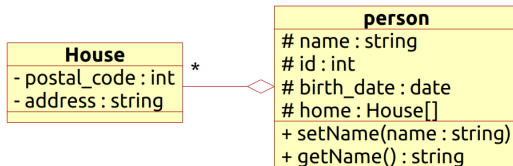


¹ filled diamond

² multiplicity

طراحی شیء‌گرا

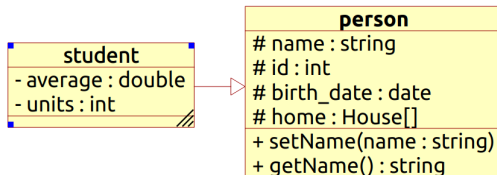
- رابطهٔ تجمیع به صورت زیر نمایش داده می‌شود. وقتی کلاس A عضوی دارد که به یک شیء از کلاس B اشاره می‌کند، در طرف کلاس A از یک لوزی توخالی¹ استفاده می‌کنیم. در برنامه‌سازی در این موارد از اشاره‌گر استفاده می‌کنیم. همچنین کثرت² یک رابطه را می‌توانیم در یک طرف یا در دو طرف آن نشان دهیم، یعنی بگوییم کلاس 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

- یکی دیگر از رابطه‌های بین دو کلاس، رابطهٔ تعمیم یا رابطهٔ وراثت¹ است.
- رابطهٔ وراثت را توسط یک خط بین پدر و فرزند و یک مثلث توخالی² در سمت پدر نشان می‌دهیم.



¹ generalization relation or inheritance relation

² empty triangle






- کثرت یک رابطه می‌تواند همچنین n به m یا چند به چند¹ باشد، بدین معنی که تعداد n شیء از کلاس A می‌تواند m شیء از کلاس B داشته باشند. مثلاً تعداد n نویسند می‌توانند m کتاب داشته باشند (چند نویسنده همکار می‌توانند یک یا چند کتاب داشته باشند و همچنین چند کتاب می‌توانند متعلق به یک یا چند نویسنده باشند).

¹ many-to-many

- یکی دیگر از رابطه‌ها رابطهٔ مشارکت² است، بدین معنی که شیئی از یک کلاس به نحوی در یک کلاس دیگر شرکت کرده است. مثلاً یک کلاس می‌تواند در پارامترهای وروی توابع خود از یک کلاس دیگر استفاده کند بدون اینکه عضوی از آن کلاس داشته باشد.
- رابطهٔ پیاده‌سازی³ بدین معنی است که یک کلاس از کلاس دیگر به ارث برده و توابع مجازی خالص آن کلاس دیگر را پیاده‌سازی کرده است.

² association relation

³ realization relation

Type of relationship	UML syntax source target	Brief semantics
Association		The description of a set of links between objects
Aggregation		The target element is a part of the source element
Composition		A strong (more constrained) form of aggregation
Generalization		The source element is a specialization of the more general target element and may be substituted for it
Realization		The source element guarantees to carry out the contract specified by the target element

برنامه‌سازی عمومی

برنامه‌سازی عمومی و قالب‌ها

- یکی از قابلیت‌هایی که در زبان سی++ وجود دارد، برنامه‌سازی عمومی² است که توسط قالب‌ها³ فراهم شده است.
- با استفاده از این قابلیت، یک تابع یا یک کلاس می‌تواند به جای اینکه برای یک نوع خاص داده تعریف شود، برای همهٔ انواع داده تعریف شود.
- در صورتی که یک تابع به صورت عمومی با استفاده از یک قالب تعریف شود، یک یا چند ورودی مورد نظر و/یا خروجی تابع می‌توانند از هر نوع داده‌ای باشند.
- همچنین در صورتی که یک کلاس به صورت عمومی تعریف شود، یک یا چند عضو کلاس یا ورودی و خروجی برخی از توابع آن می‌توانند از هر نوع داده‌ای باشند.
- وقتی از برنامه‌سازی عمومی استفاده می‌کنیم، یک متغیر مانند T یا Y به عنوان یک متغیر عمومی تعریف می‌شود و این متغیر در زمان کامپایل می‌تواند با هر نوع داده‌ای مانند int، double، string جایگزین شود.

² generic programming

³ templates

- برای مثال فرض کنید می‌خواهیم تابعی داشته باشیم که یک عنصر را در یک آرایه جستجو کند. تابعی به صورت زیر برای آرایه‌هایی از نوع عدد صحیح می‌نویسیم.

```
۱ int search(int * list, int length, int key) {  
۲     for (int i=0; i< length; i++) {  
۳         if (list[i] == key)  
۴             return i;  
۵     }  
۶     return -1;  
۷ }
```

- حال فرض کنید می‌خواهیم تابعی مشابه برای اعداد اعشاری داشته باشیم. باید این تابع را برای اعداد اعشاری نیز تعریف کنیم. با این که هر دوی این توابع کاری مشابه انجام می‌دهند، اما نیاز به تعریف مجدد آن وجود دارد.
- همچنین اگر بخواهیم این تابع را برای نوع‌های دیگری که توسط کاربر تعریف شده‌اند استفاده کنیم، باید توابعی جدید برای آنها نیز تعریف کنیم. فرض کنید نویسنده این تابع در یک کتابخانه تابع خود را تعریف کرده و در اختیار استفاده‌کنندگان قرار داده است. در اینصورت الزاماً نویسنده تابع از نوع داده‌هایی که توسط آنها تابع فراخوانی خواهد شد اطلاع نخواهد داشت.

برنامه‌سازی عمومی و قالب‌ها

- پس نیاز داریم تابع جستجو را برای یک نوع عمومی تعریف کنیم و نمی‌دانیم در زمان فراخوانی این تابع توسط چه نوعی فراخوانی خواهد شد.
- برای برنامه‌سازی عمومی در چنین مواردی زبان سی++ قالب‌ها را عرضه کرده است.
- با استفاده از یک قالب می‌توانیم تابع جستجو را به صورت زیر تعریف کنیم.

```
۱ template<typename T>
۲ int search(T * list, int size, T key) {
۳     for (int i=0; i< size; i++) {
۴         if (list[i] == key)
۵             return i;
۶     }
۷     return -1;
۸ }
```

- سپس تابع را با استفاده از نوع مورد نظر فراخوانی می‌کنیم.

```
۱ int iarr[100]; double darr[100];  
۲ search<int>(iarr, 100, 5);  
۳ search<double>(darr, 100, 5.0);  
۴ search(iarr, 100, 5); //ok  
۵ search(darr, 100, 5.0); //ok
```

- در زمان کامپایل، کامپایلر دو تابع با ورودی int و double از تابع عمومی می‌سازد.

برنامه‌سازی عمومی و قالب‌ها

- در حالت کلی قبل از تابع از کلیدواژه `template` استفاده می‌کنیم و نوع‌های عمومی که در تابع مورد استفاده قرار می‌گیرند را معرفی می‌کنیم.

```
۱ template<typename T1, typename T2, typename T3 ...>
۲ T1 function(T2 x, T3 y, ...) { ... }
```

- سپس در استفاده از تابع آن را با استفاده از نوع‌های داده‌ای مورد نیاز فراخوانی می‌کنیم.

```
۱ int res; double x; float y;
۲ res = function<int, double, float, ...>(x, y, ...);
۳ res = function(x, y, ...); // also ok
```

- به ازای هر فراخوانی از یک تابع عمومی، کامپایلر تابعی با نوع‌های داده‌ای مورد نظر را می‌سازد. تعداد توابعی که کامپایلر از یک تابع عمومی می‌سازد وابسته به نوع داده‌ها در فراخوانی‌های آن تابع است.

- یک تابع که به صورت عمومی تعریف شده است، می‌تواند برای یک حالت خاص نیز تعریف شود.
- برای مثال تابع جستجو را می‌توان در حالت خاص برای نوع رشته به گونه‌ای دیگر تعریف کرد.

```
۱ template<typename T>
۲ int search(T * list, int size, T key) {
۳     ...
۴ }
۵ int search(string * list, int size, string key) {
۶     ...
۷ }
```

- مشابه توابع، یک کلاس را نیز می‌توان به صورت عمومی با استفاده از قالب تعریف کرد.
- برای تعریف یک کلاس از نوع عمومی، قبل از تعریف تابع، از کلیدواژه `template` استفاده می‌کنیم.

```
۱ template<typename T1, typename T2, typename T3 ...>  
۲ class className {  
۳     ...  
۴     T1 variable;  
۵     T2 function(T3 x, ...) { ... }  
۶ };
```

- برای مثال فرض کنید کلاسی برای پیاده‌سازی یک وکتور تعریف کرده‌ایم که تنها می‌تواند شامل اعداد صحیح باشد.

```
۱ class Vector {  
۲ private:  
۳     int list[100];  
۴ public:  
۵     push_back(int x) { ... }  
۶ };
```

برنامه‌سازی عمومی و قالب‌ها

- برای اینکه این وکتور بتواند همهٔ نوع‌های داده‌ای را شامل شود، از قالب استفاده می‌کنیم.

```
۱ template <typename T>
۲ class Vector {
۳ private:
۴     T list[100];
۵ public:
۶     void push_back(T x) { ... }
۷ };
```

- در صورتی که بخواهیم تابع `push_back` را خارج از تعریف کلاس تعریف کنیم، باید مجدداً قالب را تعریف کنیم.

```
۱ template <typename T>
۲ void Vector<T>::push_back(T x) { ... }
```

- حال می‌توانیم شیئی از کلاس مورد نظر بسازیم.

```
۱ Vector<int> v1;  
۲ v1.push_back(4);  
۳ Vector<double> v2;  
۴ v2.push_back(5.5);
```

برنامه‌سازی عمومی و قالب‌ها

- همچنین می‌توانیم یک کلاس عمومی را برای یک نوع خاص پیاده‌سازی کنیم. برای چنین کاری باید نام تابع را با آن نوع خاص تعیین و کلاس را مجدداً جداگانه پیاده‌سازی کنیم.

```
۱ template < >  
۲ class Vector<string> { ...  
۳ };
```

- وقتی شیئی از کلاس `Vector<string>` ساخته می‌شود، کامپایلر پیاده‌سازی رشته‌ای وکتور را که توسط کاربر تعیین شده به کار می‌برد.

برنامه‌سازی عمومی و قالب‌ها

- برای یک نوع عمومی می‌توان یک مقدار پیش فرض نیز تعیین کرد. بدین منظور، در تعریف قالب مقداری برای متغیر عمومی قرار می‌دهیم.

```
۱ template <typename T=int>  
۲ class Vector { ...  
۳ };
```

- حال در استفاده از کلاس وکتور می‌توانیم متغیر عمومی را مقدار دهی نکنیم.

```
۱ Vector< > v; // this is a vector of default type "int"
```

- علاوه بر نوع‌های داده‌ای، می‌توان یک مقدار را نیز به صورت عمومی تعریف کرد. پس ورودی یک قالب علاوه بر اینکه می‌تواند یک نوع عمومی باشد، می‌تواند یک متغیر نیز باشد.
- در مثال زیر، یک وکتور از یک نوع عمومی تعریف شده است و اندازه وکتور هم به عنوان ورودی به قالب باید تعیین شود.

```
۱ template <typename T, int SIZE>
۲ class Vector { ...
۳     T data[SIZE];
۴ };
```

- حال در استفاده از کلاس برای ساختن شیء باید علاوه بر نوع داده‌های وکتور، اندازه آن را نیز تعیین کنیم.

```
۱ Vector<int, 100> v;  
۲ // this is a vector of data type int and of size 100
```

- دقت کنید برای مقادیر ورودی قالب نیز همانند نوع‌های داده‌ای، کامپایلر به ازای هر مقدار جدید یک نسخه جدید از کلاس را در زمان کامپایل می‌سازد که سرباری در زمان کامپایل ایجاد می‌کند.

- دقت کنید وقتی یک تابع از یک کلاس به صورت عمومی تعریف می‌شود، در زمان کامپایل، کامپایلر هیچ اطلاعی از نوع داده‌ای که در برنامه استفاده خواهد شد ندارد، بنابراین نمی‌تواند آن تابع را با نوع مورد نظر کامپایل کند و فایل آبجکت بسازد.
- بنابراین همه توابع عمومی باید در فایل سریتتر یا هدر تعریف شوند.

- پس به طور کلی از قالب برای دریافت یک نوع داده به عنوان پارامتر استفاده می‌کنیم.

```
۱ template<typename T>
۲ class Vector {
۳ private:
۴     T* elem; // elem points to an array of sz elements of type T
۵     int sz;
۶ public:
۷     explicit Vector(int s); // constructor: acquire resources
۸     ~Vector() { delete[] elem; } // destructor: release resources
۹     T& operator[](int i); // for non-const Vectors
۱۰    const T& operator[](int i) const; // for const Vectors
۱۱    int size() const { return sz; }
۱۲ };
```

- عبارت `template<typename T>` بدین معنی است که برای همهٔ نوع‌های داده‌ای `T` تابع یا کلاس را تعریف کن.
- سپس وکتور را به صورت‌های زیر می‌توانیم تعریف کنیم.

```
۱ Vector<char> vc(200); // vector of 200 characters
۲ Vector<string> vs(17); // vector of 17 strings
۳ Vector<list<int>> vli(45); // vector of 45 lists of integers
```

- می‌توانیم از این وکتور به صورت زیر استفاده کنیم.

```
۱ void write(const Vector<string>& vs) {  
۲     for (int i = 0; i!=vs.size(); ++i)  
۳         cout << vs[i] << '\n';  
۴ }
```

برنامه‌سازی عمومی و قالب‌ها

- اگر دو تابع `begin` و `end` برای کلاس وکتور تعریف شده باشند، می‌توانیم از ساختار حلقه بر روی دامنه استفاده کنیم.

```
۱  template<typename T>
۲  T* begin(Vector<T>& x) {
۳      // pointer to first element or nullptr
۴      return x.size() ? &x[0] : nullptr;
۵  }
۶  template<typename T>
۷  T* end(Vector<T>& x) {
۸      // pointer to one-past-last element
۹      return x.size() ? &x[0]+x.size() : nullptr;
۱۰ }
۱۱ void write2(Vector<string>& vs) {
۱۲     for (auto& s : vs)
۱۳         cout << s << '\n';
۱۴ }
```

- یکی از مفاهیمی که در سی++ استفاده می‌شود، شیء تابع¹ یا فانکتور² است با کلمهٔ تابعگون نیز ترجمه شده است. با استفاده از فانکتور می‌توانیم شیئی بسازیم و از آن شیء مانند یک تابع استفاده کنیم.
- کلاس زیر را در نظر بگیرید. عملگر () برای این کلاس تعریف شده است، بنابراین اگر شیئی از این کلاس ساخته شود، می‌توان آن را با استفاده از این عملگر فراخوانی کرد.

```
1 template<typename T>
2 class Less_than {
3     const T val; // value to compare against
4 public:
5     Less_than(const T& v) :val{v} { }
6     // call operator
7     bool operator()(const T& x) const { return x<val; }
8 };
```

¹ function object

² functor

- حال می‌توانیم شیئی از این کلاس به صورت زیر بسازیم.

```

۱ // lti(i) will compare i to 42 using < (i<42)
۲ Less_than lti {42};
۳ // lts(s) will compare s to "Hello" using < (s<"Hello")
۴ Less_than lts {"Hello"s};
۵ // "Hello" is a C-style string,
۶ // so we need <string> to get the right <
۷ Less_than<string> lts2 {"Hello"};

```

- از این اشیاء می‌توانیم به صورت زیر استفاده کنیم.

```

۱ void fct(int n, const string& s) {
۲     bool b1 = lti(n); // true if n<42
۳     bool b2 = lts(s); // true if s<"Backus"
۴     // ...
۵ }

```

- از فانکتور می‌توانیم به صورت زیر استفاده کنیم. می‌خواهیم بر روی اعضای یک لیست دلخواه (که به صورت پارامتری تعیین می‌شود و می‌تواند هر نوع لیستی باشد)، یک تابع دلخواه (که آن نیز به صورت پارامتری تعیین می‌شود و می‌تواند هر نوع تابعی باشد) را فراخوانی کنیم.

```

۱ template<typename C, typename P>
۲ int count(const C& c, P func) {
۳     int cnt = 0;
۴     for (const auto& x : c)
۵         if (func(x)) cnt++;
۶     return cnt;
۷ }
```

- حال می‌توانیم از این تابع به صورت زیر استفاده کنیم.

```

۱ Vector<int> vec {12, 24, 43};
۲ Less_than lti {42};
۳ int c = count(vec, lti);
```

- به جای استفاده از فانکتور، می‌توانیم از توابع لامبدا¹ استفاده کنیم.

- یک عبارت لامبدا² که به صورت زیر تعریف می‌شود، در واقع یک شیء تابع یا فانکتور باز می‌گرداند. به عبارت دیگر عبارت لامبدا شیئی باز می‌گرداند که می‌توان از آن به عنوان یک تابع استفاده کرد و آن تابع، تابع لامبدا نامیده می‌شود.

```
\ [ <variables to capture> ] ( <input variables> ) { <function body> }
```

- در قسمت [] مشخص می‌کنیم چه متغیرهایی که در بیرون عبارت لامبدا تعریف شده‌اند را می‌خواهیم استفاده کنیم. در قسمت () لیست ورودی‌های تابع را ذکر می‌کنیم. و در قسمت { } بدنهٔ تابع را می‌نویسیم.

¹ lambda function

² lambda expression

- برای مثال می‌خواهیم یک تابع لامبدا تعریف کنیم که عدد ورودی به تابع را با یک عدد معین مقایسه کند.
- این تابع را به صورت زیر می‌نویسیم.

```
\ auto lti = [&](int a){ return a<x; }
```

- عبارت [&] بدین معنی است که می‌خواهیم به همه متغیرهای بیرون عبارت لامبدا دسترسی با ارجاع داشته باشیم. بنابراین متغیر x بیرون از عبارت تعریف شده است.
- همچنین می‌توانیم بنویسیم [=] بدین معنا که می‌خواهیم به متغیرهای بیرون از عبارت لامبدا دسترسی با کپی داشته باشیم.
- حال می‌توانیم تابع لامبدا به صورت `lti(n)` استفاده کنیم.
- این تابع را قبلاً به صورت فانکتور با استفاده از کلاس `Less_than` تعریف کرده بودیم.

– از آنجایی که در این تابع فقط به متغیر x نیاز داریم، بنابراین عبارت لامبدا را می‌توانیم به صورت زیر نیز تعریف کنیم.

```
\ auto lti = [&x](int a){ return a<x; }
```

- فرض کنید می‌خواهیم تابعی بنویسیم که یک لیست دلخواه (که می‌تواند هر نوع لیستی باشد) دریافت کند و یک تابع دلخواه را (که می‌تواند هر نوع تابعی باشد) را دریافت کرده و تابع را بر روی همهٔ اعضای لیست فراخوانی کند.

```
۱ // C is a container, and O is an operation
۲ template<typename C, typename O>
۳ void for_all(C& c, O op) {
۴     for (auto& x : c)
۵         op(x);
۶ }
```

- حال این تابع را به صورت زیر استفاده می‌کنیم.

```
۱ std::vector<int> v {20, 30, 40, 50};
۲ for_all(v, [](int& n){ n *= 2; });
```

مدیریت استثنا

- مدیریت استثنا¹ به فرایند واکنش دادن به استثناها در حین اجرای برنامه گفته می شود.
- یک استثنا، یک شرایط غیرعادی است که به واکنش ویژه ای نیازمند است.
- یک استثنا در جریان عادی اجرای برنامه وقفه ایجاد کرده و اجرای برنامه را به قسمتی جهت مدیریت استثنا منتقل می کند.
- زبان های برنامه سازی معمولاً سازوکارهایی برای مدیریت استثنا فراهم می کنند.

¹ exception handling

مدیریت استثنا

- معمولا در زبان سی یا زبان‌هایی که سازوکاری برای مدیریت استثنا ندارند، دو راه حل در برخورد با شرایط غیرعادی وجود دارد.
- راه اول این است که وقتی تابعی با مقادیری روبرو می‌شود که آن مقادیر در حوزه مقادیر عادی نیستند، پیام خطایی صادر کند و برنامه را با استفاده از دستوراتی مانند `exit()` یا `abort()` خاتمه دهد.
- برای مثال در دسترسی به عناصر یک وکتور، اگر دسترسی در محدوده عناصر نباشد، می‌توان پیام خطا صادر کرده و از برنامه خارج شد.

```
۱ double& Vector::operator[](int i) {  
۲     if (i<0 || size()<=i) {  
۳         cout << "error: out of range access\n" ;  
۴         exit();  
۵     }  
۶     return elem[i];  
۷ }
```

- مشکل این راه حل این است که برنامه را خاتمه می‌دهد، در صورتی که در بیشتر مواقع انتظار داریم برنامه به حیات خود ادامه داده و فرصتی دوباره برای اصلاح خطا به کاربر داده شود.

```
۱ double& Vector::operator[](int i) {  
۲     if (i<0 || size()<=i) {  
۳         cout << "error: out of range access\n" ;  
۴         exit();  
۵     }  
۶     return elem[i];  
۷ }
```

- راه دوم این است که تابعی که با مقادیر غیرعادی روبرو می‌شود، با استفاده از مقداردهی یک متغیر عمومی وجود خطا را به فراخوانی‌کننده تابع اعلام کند.
- برای مثال در دسترسی به عناصر یک وکتور، در صورتی که دسترسی در محدوده صحیح نباشد، تابع می‌تواند یک متغیر عمومی را مقداردهی کند.

```
۱ double& Vector::operator[](int i) {  
۲     if (i<0 || size()<=i) {  
۳         out_of_range = -1; // what about data hiding?  
۴         return out_of_range;  
۵     }  
۶     return elem[i];  
۷ }
```

- در راه حل دوم، اگر بخواهیم یک متغیر عمومی را مقداردهی کنیم، قوانین کپسوله‌سازی و پنهان سازی داده‌ها در برنامه‌سازی شیء‌گرا را نقض کرده‌ایم. در برنامه‌سازی شیء‌گرا داده‌ها معمولاً توسط اشیا کپسوله‌سازی و پنهان‌سازی شده‌اند و نمی‌توان به آنها به طور مستقیم دسترسی پیدا کرد. پس کسی نمی‌تواند با تغییر دادن یک متغیر عمومی منطق برنامه را به هم بزند. در حالی که در اینجا یک متغیر عمومی تعریف کرده‌ایم که می‌تواند توسط دیگر توابع نیز دستکاری و تغییر داده شود.

```
1 double& Vector::operator[](int i) {  
2     if (i<0 || size()<=i) {  
3         out_of_range = -1; // what about data hiding?  
4         return out_of_range;  
5     }  
6     return elem[i];  
7 }
```

- راه سوم این است که تابعی که با مقادیر غیرعادی روبرو می‌شود، با استفاده از مقدار خروجی تابع وجود خطا را به فراخوانی‌کنندهٔ تابع اعلام کند.
- برای مثال در دسترسی به عناصر یک وکتور، در صورتی که دسترسی در محدودهٔ صحیح نباشد، تابع می‌تواند مقداری به عنوان کد خطا بازگرداند.

```
۱ double& Vector::operator[](int i) {  
۲     if (i<0 || size()<=i) {  
۳         return -1; // what if elem[i] = -1?  
۴     }  
۵     return elem[i];  
۶ }
```

- مشکل راه حل سوم این است که ممکن است همه مقادیر خروجی یک تابع مقادیر مورد نیاز فراخوانی کننده تابع باشند و هیچ مقداری را نتوان به عنوان کد خطا اعلام کرد.
- همچنین ممکن است در تابع سازنده یا مخرب با خطایی روبرو شویم، در حالی که تابع سازنده و مخرب مقدار خروجی ندارند.
- از طرف دیگر ممکن است تابع f تابع g و تابع g تابع h را فراخوانی کند و به همین ترتیب یک سلسله فراخوانی توابع اتفاق بیافتد و خطا در آخرین تابع در این سلسله شناسایی شود در حالی که اولین تابع در این سلسله نیاز به مطلع شدن از خطا داشته باشد. به عبارت دیگر ممکن است استفاده کننده خطا فراخوانی کننده بلاواسطه تابع دارای خطا نباشد و خطا با چندین واسطه نیاز به انتشار داشته باشد.

```
۱ double& Vector::operator[](int i) {  
۲     if (i<0 || size()<=i) {  
۳         return -1; // what if elem[i] = -1?  
۴     }  
۵     return elem[i];  
۶ }
```

- در زبان سی++ سازوکاری برای حل این مشکلات فراهم شده است که مدیریت استثنا نامیده می‌شود.
- هر تابع جدا از مقداری که به عنوان خروجی باز می‌گرداند می‌تواند یک مقدار به عنوان مقدار خطا نیز باز گرداند.
- این مقدار با کلمه کلیدی throw به فراخوانی کننده تابع بازگردانده (یا پرتاب) می‌شود. فراخوانی کننده تابع با استفاده از خطای دریافت شده می‌تواند تصمیم بگیرد چگونه جریان اجرای برنامه را تغییر دهد.

```
۱ double& Vector::operator[](int i) {  
۲     if (i<0 || size()<=i) {  
۳         throw 1;  
۴     }  
۵     return elem[i];  
۶ }
```

- پس یک تابع علاوه بر بازگرداندن مقادیر می‌تواند مقادیری را نیز توسط کلیدواژه `throw` ارسال کند. این مقادیر می‌توانند از هر نوعی (مانند عدد صحیح، عدد اعشاری، رشته، اشیایی از کلاس‌های تعریف‌شده توسط کاربر، و غیره) باشند.

```
۱ throw 1;
۲ throw -1;
۳ int a=0; string s="error";
۴ char message[50] = "error";
۵ complex c;
۶ throw a; throw s;
۷ throw message;
۸ throw c;
```

- پس ارسال کننده استثنا در یک تابع دلخواه f استثنایی را با استفاده از کلیدواژه throw پرتاب می‌کند.

```
۱ <return type> f(<arguments>) {  
۲     // ...  
۳     // x is a variable of any type defined somewhere  
۴     throw x;  
۵     // ...  
۶ }
```

مدیریت استثنا

- حال استفاده کننده تابعی که استثنایی را پرتاب می‌کند، می‌تواند استثنا را دریافت کند. دریافت استثنا جهت مدیریت توسط دو کلیدواژه try و catch صورت می‌گیرد.
- پس از اینکه یک استثنا در یک بلوک try دریافت شد، دستورات بعدی در بلوک اجرا نمی‌شوند و اجرا به اولین خط از بلوک catch منتقل می‌شود. در بلوک catch مقداری که توسط throw پرتاب شده است، دریافت می‌شود.

```
۱ try {  
۲     // ...  
۳     f();  
۴     // if an exception is caught,  
۵     // next command in try block are not executed.  
۶     // ...  
۷ } catch(<type> e) {  
۸     // variable x thrown in f() is caught here in variable e.  
۹     // how to handle exception  
۱۰ }
```


- همچنین می‌توان بلوک `catch` را به صورت `catch(...)` نوشت بدین معنا که نوع استثنایی که فرستاده می‌شود بی‌اهمیت است و تنها گیرنده استثنا باید استثنایی از هر نوعی را مدیریت کند.

```
۱ try {  
۲     // ...  
۳ } catch(...) {  
۴     // handle exception of any type  
۵ }
```

- فرض کنید شیئی از کلاس Vector می‌خواهد توسط عملگر زیرنویس به اعضای وکتور دسترسی پیدا کند و عملگر زیرنویس در کلاس وکتور سربارگذاری شده و در حالات غیرعادی استثنایی پرتاب می‌کند. بدین ترتیب استفاده کننده می‌تواند استثنا را به صورت زیر دریافت و مدیریت کند.

```
۱ Vector v {1, 2, 4};  
۲ try {  
۳     v[5] = 7;  
۴ }  
۵ catch (int e) {  
۶     if (e==1)  
۷         cout << "out of range access\n";  
۸ }
```

- دقت کنید هرگاه استثنایی پرتاب می‌شود، کنترل برنامه به نزدیک‌ترین نقطه‌ای می‌رود که در آن بلوک catch قرار دارد.
- برای مثال اگر تابع `f1()` تابع `f2()` را فراخوانی کند و `f2()` تابع `f3()` و به همین ترتیب الی آخر و استثنایی در تابع `fn()` پرتاب شود، و بلوک catch در کنار فراخوانی تابع `f1()` قرار داشته باشد، کنترل برنامه به اولین خط از دستورات بلوک catch در کنار تابع `f1()` منتقل می‌شود.

```
۱ try {  
۲     // ...  
۳     f1(); // f1() calls f2(), f2() calls f3(), and so on.  
۴     // finally an exception is thrown in fn()  
۵     // ...  
۶ } catch(...) {  
۷     // once fn() throws an exception,  
۸     // program control is directed here.  
۹ }
```

- همچنین بعد از اجرای دستورات درون بلوک catch، دستورات بعد از بلوک اجرا می‌شوند.

```
۱ try {  
۲     // ...  
۳     f1(); // f1() calls f2(), f2() calls f3(), and so on.  
۴     // finally an exception is thrown in fn()  
۵     // ...  
۶ } catch(...) {  
۷     // once fn() throws an exception,  
۸     // program control is directed here.  
۹ }  
۱۰ // once commands in catch are executed,  
۱۱ // program control is directed here
```

- این امکان وجود دارد که بعد از پرتاب استثنا در `fn()` یکی از توابعی که منجر به فراخوانی تابع `fn()` شده است، (مثلا `f3()`) استثنا را تا حدی مدیریت کند، و اگر مدیریت در سطح تابع `f3()` به درستی صورت نگرفت، استثنایی پرتاب کند که در سطح `f1()` مدیریت شود.

```
۱ void f3() {  
۲     try {  
۳         // ...  
۴     } catch(...) {  
۵         // handle if possible, and if not re-throw an exception  
۶         // throw ...  
۷     }  
۸ }
```

```
۱ void f3() {
۲     try {
۳         // ...
۴     } catch(...) {
۵         // handle if possible, and if not re-throw an exception
۶         // throw ...
۷     }
۸ }
۹ void main() {
۱۰ try {
۱۱     // ...
۱۲     f1(); // f1() calls f2(), f2() calls f3(), and so on.
۱۳     // finally an exception is thrown in fn()
۱۴     // f3() may catch or may re-throw an exception
۱۵ } catch(...) {
۱۶     // ...
۱۷ }
```

- این امکان وجود دارد که یک شیء از یک کلاس در زمان رخداد استثنا پرتاب شود.

```
۱ class vector_exception {  
۲ private:  
۳     int error_code;  
۴     int var;  
۵     string message;  
۶ public:  
۷     vector_exception(int e, int v, string s) :  
۸         var(v), error_code(e), message(s) { }  
۹     string what() {  
۱۰         if (error_code==1)  
۱۱             return message + ": access to element " + to_string(v);  
۱۲     }  
۱۳ };
```

- با پرتاب یک شیء از کلاس مدیریت استثنا می‌تواند کارآمدتر صورت بگیرد، بدین دلیل که اطلاعات بیشتری را می‌توان توسط ارسال کنندهٔ استثنا به دریافت کنندهٔ استثنا انتقال داد.

```
۱ double& Vector::operator[](int i) {  
۲     if (i<0 || size()<=i) {  
۳         throw vector_exception(1, i, "out_of_range");  
۴     }  
۵     return elem[i];  
۶ }  
۷ try {  
۸     vector v {1,2,3};  
۹     cout << v[5] << endl;  
۱۰ } catch(vector_exception e) {  
۱۱     cout << e.what() << endl();  
۱۲ }
```


- برخی از انواع استثناها در کتابخانه استاندارد <stdexcept> تعریف شده‌اند.
- به طور مثال استثنای خارج از محدوده عناصر یک ظرف با عنوان استثنای out_of_range تعریف شده است که می‌توانیم از آن استفاده کنیم.

```
۱ double& Vector::operator[](int i) {  
۲     if (i<0 || size()<=i)  
۳         throw out_of_range{"Vector::operator[]"};  
۴     return elem[i];  
۵ }
```

- پس می‌توانیم از کلاس استثناهای از پیش تعریف شده استفاده کنیم.

```
۱ void f(Vector& v) {  
۲     // ...  
۳     try { // exceptions here are handled by the handler defined below  
۴         v[v.size()] = 7; // try to access beyond the end of v  
۵     } catch (out_of_range& err) {  
۶         // out_of_range error  
۷         cerr << err.what() << '\n';  
۸     }  
۹     // ...  
۱۰ }
```

- کلاس‌های تعریف شده در کتابخانه استاندارد `stdexcept` دارای یک سلسله مراتب هستند. به طور مثال کلاس `out_of_range` زیرکلاس `logic_error` می‌باشد و `logic_error` زیرکلاس `exception` است.
- کاربر می‌تواند از این کلاس‌های از پیش تعریف شده به ارث ببرد و کلاس جدیدی به عنوان زیرکلاس یکی از این کلاس‌ها تعریف کند.

- در صورتی که در تعریف یک تابع از کلیدواژه `noexcept` استفاده کنیم، تابع استثنایی ارسال نخواهد کرد.

```
۱ void user(int sz) noexcept {  
۲     Vector v(sz);  
۳     iota(&v[0], &v[sz], 1);  
۴     // ...  
۵ }
```

- در سازنده کلاس وکتور در صورتی که وکتور با یک عدد منفی ساخته شود و یا تخصیص حافظه به درستی صورت نگیرد نیز می‌توان یک استثنا ارسال کرد.

```
۱ Vector::Vector(int s) {  
۲     if (s < 0)  
۳         throw length_error{"Vector constructor: negative size"};  
۴     elem = new double[s]; // new may also throw an exception  
۵     sz = s;  
۶ }  
۷ void test() {  
۸     try { Vector v(-27); } catch (std::length_error& err) {  
۹         // handle negative size  
۱۰    } catch (std::bad_alloc& err) {  
۱۱        // handle memory exhaustion  
۱۲    }  
۱۳ }
```

مدیریت استثنا

- تابع `test` می‌تواند برخی از استثناها را خود مدیریت کند و مابقی استثناها را ارسال کند تا توابع فراخوانی‌کننده تابع `test` آنها را مدیریت کنند.
- در اینجا در صورتی که وکتور به درستی مقداردهی اولیه نشود، یک پیام خطا چاپ می‌شود و یک استثنا به تابع فراخوانی‌کننده `test` ارسال می‌شود، اما در صورتی که حافظه به درستی تخصیص داده نشود، برنامه متوقف می‌شود، چرا که این برنامه برای چنین استثناهایی هیچ تدارکی ندیده است.

```
1 void test() {  
2     try { Vector v(-27); } catch (std::length_error&) {  
3         cerr << "test failed: length error\n"; // print the error  
4         throw; // rethrow  
5     } catch (std::bad_alloc&) {  
6         // this program is not designed to handle memory exhaustion  
7         std::terminate(); // terminate the program  
8     }  
9 }
```

- استثناها در زمان اجرای برنامه تشخیص داده می‌شوند. برخی از خطاها را می‌توان در زمان کامپایل تشخیص داد. برای این کار از `static_assert` استفاده می‌کنیم.
- `static_assert` یک شرط را در اولین ورودی خود می‌گیرد. در صورتی که شرط برقرار نبود، پیام خطای دومین ورودی خود را صادر می‌کند.
- برای مثال در زمان کامپایل می‌توان تشخیص داده اگر نوع داده عدد صحیح در سیستمی که کد بر روی آن اجرا می‌شود ۲ بایتی است و یا ۴ بایتی. می‌خواهیم در صورتی که نوع عدد صحیح ۲ بایتی است، پیام خطا صادر کنیم که به صورت زیر عمل می‌کنیم. `assert` استفاده می‌کنیم.

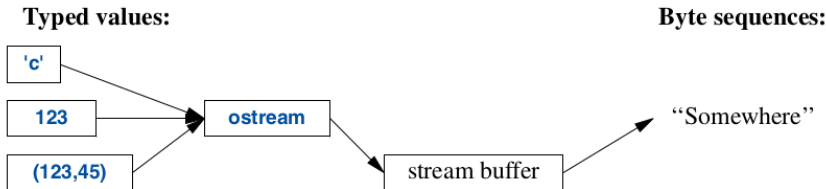
```
\ static_assert(4<=sizeof(int), "integers are too small");
```

- همچنین مقادیر ثابت را می‌توان در زمان کامپایل بررسی کرده، بر روی آنها قید گذاشت و در صورتی که قید برقرار نبود، پیام خطا صادر کرد.

```
۱ constexpr double C = 299792.458; // km/s
۲ void f(double speed) {
۳     constexpr double local_max = 160.0/(60*60);
۴     // 160 km/h == 160.0/(60*60) km/s
۵     static_assert(speed < C, "can't go that fast");
۶     // error : speed must be a constant
۷     static_assert(local_max < C, "can't go that fast"); // OK
۸     // ...
۹ }
```

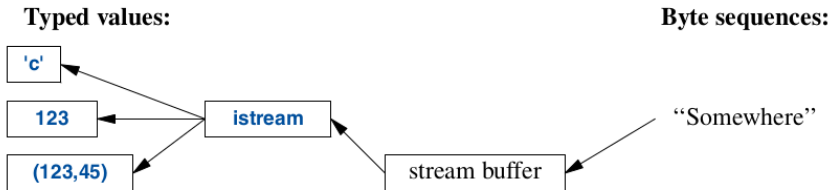

ورودی و خروجی

- کتابخانه جریان ورودی و خروجی¹ امکاناتی برای خواندن یک ورودی و چاپ یک خروجی فراهم می‌کند.
- کلاس ostream اشیایی با نوع‌های داده‌ای متفاوت را به جریانی (استریمی) از کاراکترها تبدیل می‌کند.



¹ I/O stream library

- همچنین کلاس `istream` جریانی (استریمی) از کاراکترها را به اشیایی با نوع‌های داده‌ای متفاوت تبدیل می‌کند.



- شیء `cout` از کلاس `ostream` در کتابخانه `<ostream>` تعریف شده است. عملگر درج خروجی ¹ `<<` برای این کلاس سربارگذاری شده است.
- این عملگر یک عملگر دوتایی است که یک شیء از کلاس `ostream` را در طرف چپ خود و یک داده در طرف راست خود دریافت کرده و آن داده را به خروجی استاندارد برای چاپ می‌فرستد.
- همچنین `cerr` خروجی استاندارد برای ارسال پیام‌های خطاهاست.

```
۱ i = 42;  
۲ cout << "the value of i is " << i << '\n';
```

¹ insert output

- به طور مشابه شیء cin از کلاس istream در کتابخانه <istream> تعریف شده است. عملگر استخراج ورودی¹ >> برای این کلاس سربارگذاری شده است.
- این عملگر یک عملگر دوتایی است که یک شیء از کلاس istream را در طرف چپ خود و یک متغیر در طرف راست خود دریافت کرده و مقدار آن متغیر را از ورودی استاندارد می‌خواند.
- حرف c در cin و cout مخفف کلمه کاراکتر است.

```
۱ int i;  
۲ double d;  
۳ cin >> i >> d; // read into i and d
```

¹ extract input

- cin با استفاده از جداکننده خط فاصله تشخیص می‌دهد که ورودی‌ها را از هم جدا کرده، در متغیرهای مختلف قرار دهد.

- برای خواندن یک خط از ورودی که با کاراکتر '\n' پایان می‌یابد، از تابع getline استفاده می‌کنیم.

```
۱ cout << "Please enter your name\n";  
۲ string str;  
۳ getline(cin, str);  
۴ cout << "Hello, " << str << "!\n";
```

ورودی و خروجی

- فرض کنید می‌خواهیم از یک جریان (استریم) ورودی، تعدادی عدد صحیح بخوانیم و به یک وکتور اضافه کنیم. این جریان ورودی می‌تواند یک فایل ورودی یا ورودی استاندارد باشد. بعدها خواهیم دید که علاوه بر cin یک فایل ورودی نیز شیئی از یک زیرکلاس istream است.

- با استفاده از عملگر استخراج ورودی در یک حلقهٔ تکرار به صورت زیر عمل می‌کنیم.

```
۱ vector<int> read_ints(istream& is) {  
۲     vector<int> res;  
۳     for (int i; is>>i; )  
۴         res.push_back(i);  
۵     return res;  
۶ }
```

- عملگر استخراج بر روی یک استریم ورودی، یک استریم ورودی باز می‌گرداند که در صورتی که دادهٔ درستی نخواند مقدار پرچم داخلی¹ خود را برابر با failbit قرار می‌دهد که معادل صفر یا نادرست است، پس از حلقه خارج می‌شویم.

¹ internal flag

- همچنین می‌توانیم به صورت زیر با استفاده از تابع `good()` تا وقتی که جریان ورودی با اشکالی روبرو نشده است، ورودی را از `cin` بخوانیم.

```
۱ cin >> x;  
۲ while(cin.good()) {  
۳     cout << "x: " << x << endl;  
۴     cin >> x;  
۵ }
```

- سپس برای استفاده مجدد از `cin` آن را توسط `cin.clear()` پاکسازی می‌کنیم و مقدار پرچم داخلی را برابر با `goodbit` قرار می‌دهیم و آنچه در بافر قرار دارد توسط `cin.ignore()` خالی می‌کنیم.

- عملگرهای درج و استخراج برای همه نوع‌های داده‌ای اصلی سربارگذاری شده‌اند.
- همانطور که در سربارگذاری توابع گفتیم، دو عملگر استخراج و درج را می‌توانیم برای انواع داده‌ای تعریف‌شده توسط کاربر نیز تعریف کنیم.

```
۱ struct Entry {  
۲     string name;  
۳     int number;  
۴ };  
۵  
۶ ostream& operator<<(ostream& os, const Entry& e) {  
۷     return os << "{\n" << e.name << "\", " << e.number << "}";  
۸ }
```

- برای سربارگذاری عملگر استخراج به صورت زیر عمل می‌کنیم.

```
۱ istream& operator>>(istream& is, Entry& e)
۲ // read { "name" , number } pair. Note: formatted with { " " , and }
۳ {
۴     char c, c2;
۵     if (is>>c && c=='{' && is>>c2 && c2=='"') { // start with a { "
۶
۷         string name;
۸         // the default value of a string is the empty string: ""
۹         while (is.get(c) && c!='"')
۱۰             // anything before a " is part of the name
۱۱             name+=c;
۱۲
۱۳         // (to be continued)
```

- برای سربارگذاری عملگر استخراج به صورت زیر عمل می‌کنیم.

```
۱
۲     if (is>>c && c=='(',')') {
۳         int number = 0;
۴         if (is>>number>>c && c=='}')) { // read the number and a }
۵             e = {name,number};
۶             // assign to the entry
۷             return is;
۸         }
۹     }
۱۰ }
۱۱ is.setstate(ios_base::failbit);
۱۲ return is;
۱۳ // register the failure in the stream
۱۴ }
```

- بعد از سربارگذاری این دو عملگر می‌توانیم به صورت زیر مقادیری را در ساختمان Entry بخوانیم و چاپ کنیم.

```
۱ for (Entry ee; cin>>ee; ) // read from cin into ee
۲     cout << ee << '\n'; // write ee to cout
```

- کتابخانه `iostream` تعدادی عملگر برای کنترل کردن فرمت خروجی و ورودی نیز ارائه می‌دهد.

- برای مثال می‌توانیم یک خروجی را در مبناهای مختلف چاپ کنیم.

```
۱ cout << 1234 << ',' << hex << 1234 << ',' << oct << 1234 << '\n';  
۲ // print 1234,4d2,2322
```

- همچنین می‌توانیم اعدادی اعشاری به اشکال مختلف نمایش دهیم.

```
۱ constexpr double d = 123.456;  
۲ cout << d << "; " // use the default format for d  
۳ << scientific << d << "; " // use 1.123e2 style format for d  
۴ << hexfloat << d << "; " // use hexadecimal notation for d  
۵ << fixed << d << "; " // use 123.456 style format for d  
۶ << defaultfloat << d << '\n'; // use the default format for d  
۷  
۸ // output :123.456; 1.234560e+002;  
۹ // 0x1.edd2f2p+6; 123.456000; 123.456
```

- اعداد اعشاری را می‌توان توسط تابع `precision` گرد کرد.

```
۱ cout.precision(8);  
۲ cout << 1234.56789 << '\n'; // output : 1234.5679  
۳ cout.precision(4);  
۴ cout << 1234.56789 << '\n'; // output : 1235
```

- از کلاس `iostream` که برای ورودی و خروجی استاندارد طراحی شده است، کلاس `fstream` ارث‌بری می‌کند که برای خواندن از فایل‌ها و نوشتن بر روی آنهاست.
- به طور خاص، کلاس `ifstream` برای خواندن از فایل‌ها و کلاس `ofstream` برای نوشتن بر روی فایل‌هاست.
- همچنین در کتابخانه `sstream` کلاس `stringstream` برای خواندن از یک رشته و نوشتن بر روی یک رشته طراحی شده است.
- به طور خاص `istringstream` برای خواندن از روی یک رشته و `ostringstream` برای نوشتن بر روی یک رشته طراحی شده است.

- برای مثال می‌توانیم یک رشته را به صورت زیر با استفاده از `ostringstream` تولید کنیم.

```
۱ ostringstream oss;  
۲ oss << "{temperature," << scientific << 123.4567890 << "}";  
۳ cout << oss.str() << '\n';
```

- کتابخانه <filesystem> دارای کلاس‌هایی است که می‌توان برای استفاده با فایل‌ها از آنها استفاده کرد.
- برای مثال می‌توانیم با استفاده از کلاس path آدرس یک فایل را دریافت کنیم.

```

۱ int main(int argc, char* argv[]) {
۲     if (argc < 2) {
۳         cerr << "arguments expected\n";
۴         return 1;
۵     }
۶
۷     path p {argv[1]}; // create a path from the command line
۸     cout << p << " " << exists(p) << '\n';
۹     // note: a path can be printed like a string
۱۰    // ...
۱۱ }

```

– با استفاده از کلاس ofstream می‌توانیم یک فایل را برای نوشتن و با استفاده از ifstream می‌توانیم یک فایل را برای خواندن باز کنیم.

```
۱ ofstream file;  
۲ file.open ("example.txt");  
۳ if (file.is_open()) {  
۴     file << "Writing this to a file.\n";  
۵     file.close();  
۶ }
```

- با استفاده از کلاس ofstream می‌توانیم یک فایل را برای نوشتن و با استفاده از ifstream می‌توانیم یک فایل را برای خواندن باز کنیم.

```

۱ string line;
۲ ifstream file ("example.txt");
۳ if (file.is_open()) {
۴     while ( getline (file,line) ) {
۵         cout << line << '\n';
۶     }
۷     file.close();
۸ }

```

- بعد از خواندن ورودی یا نوشتن خروجی می‌توانیم بررسی کنیم که آیا خواندن و نوشتن موفقیت آمیز بوده یا با خطا برخورد کرده‌ایم و یا به انتهای فایل رسیده‌ایم. بدین منظور توابعی مانند eof، fail، bad، good در نظر گرفته شده است.

- با استفاده از تابع `seekg` می‌توانیم مکانی در فایل که خواندن از آنجا صورت می‌گیرد را تعیین کنیم. همچنین با استفاده از تابع `seekp` می‌توانیم مکانی در فایل که نوشتن در آنجا صورت می‌گیرد را تعیین کنیم.
- با استفاده از تابع `tellg` و `tellp` نیز مکانی که خواندن و نوشتن در آنجا صورت می‌گیرد را دریافت کنیم.

```
۱ seekg ( position );  
۲ seekp ( position );  
۳ seekg ( offset, direction );  
۴ seekp ( offset, direction );  
۵ tellg();  
۶ tellp();
```

- برای مثال با استفاده از این توابع می‌توانیم اندازه یک فایل را تعیین کنیم.

```

۱ streampos begin,end;
۲ ifstream myfile ("example.bin", ios::binary);
۳ begin = myfile.tellg();
۴ myfile.seekg (0, ios::end);
۵ end = myfile.tellg();
۶ myfile.close();
۷ cout << "size is: " << (end-begin) << " bytes.\n";

```

- برای فایل‌های دودویی که متنی نیستند، می‌توانیم از توابع `read(memory_block, size)` و `write(memory_block, size)` برای خواندن و نوشتن استفاده کنیم.

```
۱ streampos size;
۲ char * memblock;
۳ ifstream file ("example.bin", ios::in|ios::binary|ios::ate);
۴ if (file.is_open()) {
۵     size = file.tellg();
۶     memblock = new char [size];
۷     file.seekg (0, ios::beg);
۸     file.read (memblock, size);
۹     file.close();
۱۰     cout << "the entire file content is in memory";
۱۱     delete[] memblock;
۱۲ } else
۱۳     cout << "Unable to open file";
```

- کلاس path توابع کاربردی زیادی دارد که می‌توان از آنها استفاده کرد.

```

۱ void test(path p) {
۲     if (is_directory(p)) {
۳         cout << p << ":\n";
۴         for (const directory_entry& x : directory_iterator(p)) {
۵             const path& f = x;
۶             string n = f.extension().string();
۷             if (n == ".cpp" || n == ".C" || n == ".cxx")
۸                 cout << f.stem() << " is a C++ source file\n";
۹         }
۱۰     }
۱۱ }

```

- توابع دیگری مانند copy برای کپی کردن فایل‌ها، copy_file برای کپی کردن محتوای فایل، create_directory برای ساختن پوشه، و remove برای حذف فایل وجود دارند که در صورت نیاز می‌توان از آنها استفاده کرد.

ساختار داده‌ها و الگوریتم‌های استاندارد

- در پیاده‌سازی سیستم‌های نرم‌افزاری در بیشتر مواقع به مجموعه‌ای از داده‌ها نیاز داریم که باید به نحوی ذخیره‌سازی کرده و بر روی آنها عملیات انجام دهیم.
- کلاسی را که یک مجموعه از داده‌ها را نگهداری می‌کند، معمولاً یک ظرف¹ می‌نامیم.
- معمولاً در یک برنامه باید ظرف‌هایی که نیاز داریم را با انواع توابع مورد نیازها پیاده‌سازی کنیم. در کتابخانه استاندارد سی++ بسیاری از ساختار داده‌های معمول پیاده‌سازی شده‌اند.

¹ container

- ظرف‌های استاندارد را می‌توان به چهار دسته ظرف‌های ترتیبی¹، ظرف‌های مبدل²، ظرف‌های رابطه‌ای³ تقسیم کرد.

¹ sequence containers

² container adaptors

³ associative containers

ساختار داده‌های استاندارد

- ظرف‌های ترتیبی به ظرف‌هایی گفته می‌شود که دسترسی به عناصر آنها به صورت ترتیبی صورت می‌گیرد.
- این ظرف‌ها شامل آرایه (array)، وکتور یا حامل (vector)، صف دوطرفه (deque)، و لیست یا لیست پیوندی (list) می‌شوند.
- آرایه ظرفی است که اندازه آن در زمان اجرای برنامه غیرقابل تغییر است.
- وکتور آرایه‌ای است که اندازه آن در زمان اجرا قابل افزایش و کاهش است و علاوه بر آن پیاده‌سازی آن برای دسترسی تصادفی به عناصر وکتور بهینه‌سازی شده است.
- صف دوطرفه یک صف است که از ابتدا و انتها می‌توان به عناصر آن دسترسی پیدا کرد. به علاوه دسترسی تصادفی به عناصر آن به صورت بهینه انجام می‌شود.
- لیست یک لیست پیوندی دوطرفه است.

- ظرف‌های مبدل به ظرف‌هایی گفته می‌شود که یک سازوکار جدید برای پیاده‌سازی ظرف‌ها ارائه نمی‌کنند، بلکه تنها از ظرف‌های ترتیبی به عنوان زیرساخت استفاده کرده و امکانات و قابلیت‌های جدید به آنها اضافه می‌کنند.
- این ظرف‌ها شامل صف (queue)، پشته (stack)، و صف اولویت (priority_queue) می‌شوند.

- پشته ساختار داده‌ای است که در آن آخرین عنصری که وارد می‌شود، اولین عنصری است که خارج می‌شود. به عبارت دیگر برای اضافه کردن یک عنصر به پشته به بالای آن عنصری را اضافه و برای حذف از پشته آخرین عنصر اضافه شده به پشته اولین عنصری است که حذف می‌شود. پشته مفهوم آخرین-ورودی، اولین-خروجی¹ را پیاده‌سازی می‌کند.
- صف ساختار داده‌ای است که در آن اولین عنصری که وارد می‌شود، اولین عنصری است که خارج می‌شود. پس وقتی یک عنصر به صف وارد می‌شود در آخر صف قرار می‌گیرد و اولین عنصری که به صف وارد شده اولین عنصری است که خارج می‌شود. پشته مفهوم اولین-ورودی، اولین-خروجی² را پیاده‌سازی می‌کند.
- در یک صف اولویت، هر عنصر دارای یک اولویت است و عناصری که اولویت بیشتری دارند زودتر از صف خارج می‌شوند.

¹ last in, first out (LIFO)

² first in, first out (FIFO)

- ظرف‌های رابطه‌ای به ظرف‌هایی گفته می‌شود دسترسی به عناصر آنها بر اساس یک رابطه یا یک نگاشت است.
- ظرف‌های رابطه‌ای شامل نگاشت (map)، مجموعه (set)، چندنگاشت (multimap)، چندمجموعه (multiset)، نگاشت نامرتب (unordered_map) و مجموعه نامرتب (unordered_set) می‌شوند.
- نگاشت ظرفی است که توسط آن یک کلید به یک مقدار نسبت داده می‌شود، پس دسترسی به هر عنصر توسط کلید یکتای آن عنصر است. کلیدهای یک نگاشت به صورت مرتب در درون آن قرار گرفته‌اند.
- در یک مجموعه هر عنصر تنها یک بار تکرار می‌شود و دسترسی به هر عنصر مجموعه توسط یک کلید یکتا صورت می‌گیرد. عناصر یک مجموعه به صورت مرتب در آن قرار گرفته‌اند.

- چندنگاشت، نگاشتی است که در آن کلید یکتا نیست و چند عنصر می‌توانند کلید یکسان داشته باشند. همچنین چندمجموعه، مجموعه‌ای است که در عناصر می‌توانند تکرار شوند.
- نگاشت و مجموعه نامرتب شبیه نگاشت و مجموعه هستند با این تفاوت که عناصر درونی آنها مرتب نشده است.

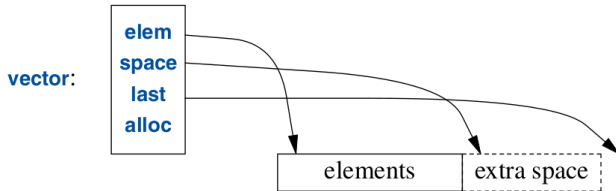
- هر ظرف شامل یک پیمایشگر¹ است. یک پیمایشگر شبیه یک اشاره‌گر است که به یک عنصر از یک ظرف اشاره می‌کند. پیمایشگرهای ظرف‌های مختلف به صورت‌های مختلف تعریف شده‌اند.
- همچنین برای هر ظرف الگوریتم‌های متفاوتی تعریف شده است که می‌توان از این الگوریتم‌ها به صورت بهینه برای عملیات متفاوت بر روی ظرف‌ها از جمله مرتب‌سازی و جستجو استفاده کرد.

¹ iterator

- وکتور یا حامل یکی از پر استفاده ترین ظرفهاست. یک وکتور دنباله ای از عناصر با یک نوع معین است. عناصر وکتور به صورت به هم پیوسته ¹ مجاور یکدیگر در حافظه قرار گرفته اند.
- در یک وکتور تعدادی عنصر قرار گرفته و تعدادی فضای خالی برای عناصری که در آینده وارد وکتور خواهند شد وجود دارد. وقتی فضای خالی پر شد، وکتور مجددا فضای جدیدی در حافظه تخصیص می دهد.

¹ contiguous

- به عبارت دیگر یک وکتور شامل اندازه‌ای ¹ است که تعداد عناصر درون آن را تعیین می‌کند و دارای ظرفیتی ² است که حداکثر تعداد عناصری که می‌توانند درون آن قرار بگیرند را تعیین می‌کند. وقتی اندازه بیشتر به ظرفیت برسد، وکتور باید برای عناصر جدید فضای بیشتر تخصیص دهد و ظرفیت وکتور را افزایش دهد.



¹ size

² capacity

- وکتور توسط قالب پیاده‌سازی شده است و آن را می‌توان با استفاده از عناصری از هر نوع داده‌ای ساخت. در مقداردهی اولیه می‌توان اندازه وکتور و مقدار پیش‌فرض عناصر آن را تعیین کرد.

```
۱ vector<int> v1 = {1, 2, 3, 4}; // size is 4
۲ vector<string> v2; // size is 0
۳ vector<Shape*> v3(23); // size is 23; initial element value: nullptr
۴ vector<double> v4(32,9.9); // size is 32; initial element value: 9.9
```

- عملگر زیرنویس برای وکتور سربارگذاری شده است، بنابراین می‌توانیم به عناصر آن به صورت تصادفی دسترسی پیدا کنیم.

```
۱ vector<int> myvector (10); // 10 zero-initialized elements
۲ vector<int>::size_type sz = myvector.size();
۳ // assign some values:
۴ for (unsigned i=0; i<sz; i++) myvector[i]=i;
```

- می‌توانیم وکتوری به صورت زیر بسازیم:

```
۱ struct Entry { string name; int number; };  
۲  
۳ vector<Entry> phone_book = {  
۴     {"Mr. X", 123},  
۵     {"Mrs. Y", 456}  
۶ };
```

- از آنجایی که عملگر درج را برای Entry تعریف کرده‌ایم می‌توانیم به صورت زیر عناصر وکتور را چاپ کنیم.

```
۱ void print_book(const vector<Entry>& book) {  
۲     for (int i = 0; i!=book.size(); ++i)  
۳         cout << book[i] << '\n';  
۴ }
```

- همچنین توسط حلقهٔ تکرار بر روی دامنه می‌توانیم به صورت زیر به عناصر وکتور دسترسی پیدا کنیم.

```
۱ void print_book(const vector<Entry>& book) {  
۲     for (const auto& x : book)  
۳         cout << x << '\n';  
۴ }
```

- یکی از توابع وکتور تابع `push_back` است که توسط آن می‌توان یک عنصر به وکتور افزود. با فرض اینکه عملگر استخراج برای نوع داده `Entry` تعریف شده باشد، می‌توانیم به صورت زیر عمل کنیم.

```

۱ void input() {
۲     for (Entry e; cin>>e; )
۳         phone_book.push_back(e);
۴ }
```

- تابع `push_back` به گونه‌ای طراحی شده است که تا وقتی که اندازه وکتور به ظرفیت آن نرسیده است عنصر را به وکتور اضافه می‌کند و پس از اینکه اندازه به ظرفیت رسید، ظرفیت وکتور را می‌افزاید و همچنین ممکن است فضایی جدید در حافظه برای عناصر خود تخصیص دهد و عناصر در حافظه قبلی را در حافظه جدید کپی کند.

- با استفاده از تابع `reserve()` می‌توان فضایی را در حافظه تخصیص داد و بدین‌گونه از تخصیص مجدد حافظه جلوگیری کرد، اما باید دانست که وکتور از راه‌های اکتشافی یا هیوریستیک‌هایی استفاده می‌کند که توسط آن مقدار بهینه ظرفیت را پیش‌بینی می‌کند.

- با استفاده از تابع `pop_back()` می‌توانیم عنصری را از یک وکتور حذف کنیم.

```
۱ vector<int> myvector;  
۲ int sum (0);  
۳ myvector.push_back (100);  
۴ myvector.push_back (200);  
۵ myvector.push_back (300);  
۶  
۷ while (!myvector.empty()) {  
۸     sum+=myvector.back();  
۹     myvector.pop_back();  
۱۰ }
```

- دسترسی به عناصر خارج از محدوده در وکتور توسط عملگر زیرنویس [] خطایی تولید نمی‌کند.

```
۱ vector<Entry> book;  
۲ int i = book[book.size()].number;  
۳ // i gets a random value  
۴ }
```

- وکتور در دسترسی خارج از محدوده استثنایی ارسال نمی‌کند، چرا که با پیاده‌سازی استثنا برای آن سربار اضافی تحمیل شده و از بهره‌وری وکتور کاسته می‌شود.

- دسترسی به عناصر خارج از محدوده در وکتور توسط عملگر زیرنویس [] خطایی تولید نمی‌کند.

```
۱ vector<Entry> book;  
۲ int i = book[book.size()].number;  
۳ // i gets a random value  
۴ }
```

- از طرف دیگر می‌توان با استفاده از تابع at به اعضای یک وکتور دسترسی پیدا کرد که در صورت دسترسی خارج از محدوده این تابع یک استثنا ارسال می‌کند.

- می‌توانیم کلاسی تعریف کنیم که از کلاس وکتور ارث‌بری می‌کند و بدین ترتیب عملگر زیرنویس را سربارگذاری کرده به نحوی که دسترسی به عناصر استثنا ارسال کند.

```
۱ template<typename T>
۲ class Vec : public std::vector<T> {
۳ public:
۴     // use the constructors from vector (under the name Vec)
۵     using vector<T>::vector;
۶     T& operator[](int i) { return vector<T>::at(i); } // range check
۷     const T& operator[](int i) const {
۸         return vector<T>::at(i); // range check const objects
۹     }
۱۰ };
```

- می‌توانیم کلاسی تعریف کنیم که از کلاس وکتور ارث‌بری می‌کند و بدین ترتیب عملگر زیرنویس را سربارگذاری کرده به نحوی که دسترسی به عناصر استثنا ارسال کند.

```
۱ void checked(Vec<Entry>& book) {  
۲     try {  
۳         book[book.size()] = {"Joe", 999999};  
۴         // ...  
۵     } catch (out_of_range&) {  
۶         cerr << "range error\n";  
۷     }  
۸ }
```

- بهتر است همیشه در بدنه اصلی برنامه همه استثناها را مدیریت کنیم تا اگر تابعی یک استثنا را مدیریت نکرد با توقف برنامه روبرو نشویم.

```
۱ int main() {  
۲     // ...  
۳     try {  
۴         // code using Vec  
۵     } catch (out_of_range&) {  
۶         cerr << "range error\n";  
۷     } catch (...) {  
۸         cerr << "unknown exception thrown\n";  
۹     }  
۱۰    // ...  
۱۱ }
```

- معمولا در استفاده از ظرف‌ها می‌خواهیم عناصر ظرف را پیمایش کنیم. برای چنین کاری معمولا از یک پیمایشگر¹ استفاده می‌کنیم.
- یک پیمایشگر شبیه یک اشاره‌گر است که به یکی از عناصر یک ظرف اشاره می‌کند و می‌توان با استفاده از آن عناصر را پیمایش کرد. هر ظرف برای خود به نحوی متفاوت پیمایشگری پیاده‌سازی کرده است ولی نحوه استفاده از آنها یکسان است.

¹ iterator

- هر ظرف در کتابخانه استاندارد دو تابع `begin()` و `end()` فراهم کرده است. تابع `begin()` پیمایشگری به اولین عنصر ظرف و تابع `end()` پیمایشگری به عنصر ماقبل آخر ظرف باز می گرداند.
- اگر `p` یک پیمایشگر باشد، `*p` مقدار عنصری است که آن پیمایشگر به آن اشاره می کند، و `++p` پیمایشگر را یک عنصر به جلو حرکت می دهد. همچنین اگر `p` به کلاسی اشاره کند که یکی از اعضای آن `m` است، آنگاه `p->m` نشان دهنده آن عضو از کلاس است که معادل با `m`. (`*p`) است.

```

۱ vector<int> vec(10,100);
۲ for (vector<int>::iterator it = vec.begin(); it != vec.end(); ++it)
۳     cout << ' ' << *it;

```

- در کلاس وکتور می‌توان پیمایشگر را به صورت زیر تعریف کرد.

```
۱ template <typename T>
۲ class Vector {
۳ public:
۴     typedef T * iterator;
۵ };
۶ auto Vector<int>::iterator iter;
```

- همچنین با استفاده از پیمایشگرها می‌توانیم وکتور را مقداردهی اولیه کنیم.

```
۱ // empty vector of ints
۲ vector<int> first;
۳ // four ints with value 100
۴ vector<int> second (4,100);
۵ // iterating through second
۶ vector<int> third (second.begin(),second.end());
۷ // a copy of third
۸ vector<int> fourth (third);
۹ // the iterator constructor can also be used to construct from arrays:
۱۰ int myints[] = {16,2,77,29};
۱۱ vector<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );
```


- از تابع insert می‌توانیم برای درج عناصری در میانهٔ وکتور استفاده کنیم.

```
۱ vector<int> myvector (3,100);
۲ vector<int>::iterator it;
۳ it = myvector.begin();
۴ it = myvector.insert ( it , 200 );
۵ myvector.insert (it,2,300);
۶ // "it" no longer valid, get a new one:
۷ it = myvector.begin();
۸ vector<int> anothervector (2,400);
۹ myvector.insert (it+2,anothervector.begin(),anothervector.end());
۱۰ int myarray [] = { 501,502,503 };
۱۱ myvector.insert (myvector.begin(), myarray, myarray+3);
```

- تابع reserve برای اختصاص دادن حافظه و تغییر ظرفیت وکتور به کار برده می‌شود.

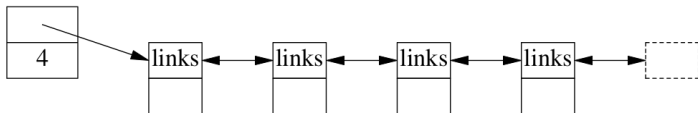
```
۱ vector<int> bar;  
۲ sz = bar.capacity();  
۳ bar.reserve(100);    // change the capacity to 100  
۴ cout << "making bar grow:\n";  
۵ for (int i=0; i<200; ++i) {  
۶     bar.push_back(i);  
۷     if (sz!=bar.capacity()) {  
۸         sz = bar.capacity();  
۹         cout << "capacity changed: " << sz << '\n';  
۱۰     }  
۱۱ }
```

- یکی از ساختار داده‌های ترتیبی صف دوطرفه ¹ است که شبیه وکتور است با این تفاوت که پیاده‌سازی آن به نحوی است که درج و حذف از ابتدای وکتور را نیز فراهم می‌کند.
- اضافه کردن این رفتار سرباری نیز دارد به طوری که بهره‌وری آن از وکتور کمتر است ولی در جایی که به اضافه و درج در ابتدای وکتور نیاز است می‌تواند این کار را بهینه‌تر انجام دهد.

¹ double ended queue

- یکی از ساختار داده‌های ترتیبی لیست است که در واقع پیاده‌سازی یک لیست پیوندی دوطرفه¹ است.

list:



- معمولاً در جایی که به ظرفی از عناصر نیاز داریم از وکتور استفاده می‌کنیم. وکتور همچنین الگوریتم‌های جستجو و مرتب‌سازی بهینه‌تری دارد. اما وقتی نیاز به درج و حذف تعداد زیادی از عناصر در میانه ظرف داریم از لیست استفاده می‌کنیم.

¹ doubly-linked list

- هنگامی از لیست استفاده می‌کنیم که می‌خواهیم در لیست درج و حذف کنیم، بدون اینکه عناصر دیگر لیست را جابجا کنیم.

- یک لیست را می‌توانیم شبیه یک وکتور مقداردهی اولیه و از آن استفاده کنیم.

```
۱ list<Entry> phone_book = { {"Mr. X",123}, {"Mrs. Y", 456} };
۲ int get_number(const string& s) {
۳     for (const auto& x : phone_book)
۴         if (x.name==s)
۵             return x.number;
۶     return 0; // use 0 to represent "number not found"
۷ }
```

- همچنین با استفاده از یک پیمایشگر می‌توانیم عناصر یک لیست را پیمایش کنیم.

```
۱ int get_number(const string& s) {  
۲     for (auto p = phone_book.begin(); p!=phone_book.end(); ++p)  
۳         if (p->name==s)  
۴             return p->number;  
۵     return 0; // use 0 to represent "number not found"  
۶ }
```

- پیمایشگر لیست از نوع پیمایشگر دوطرفه (bidirectional_iterator) است، یعنی تنها به جلو و عقب حرکت می‌کند. دسترسی به یک عنصر در میانهٔ لیست به صورت تصادفی توسط عملگر زیرنویس امکان پذیر نیست.

- با استفاده از توابع insert و erase می‌توانیم عنصری به لیست بیافزاییم و عنصری از لیست حذف کنیم.

```
۱ void f(const Entry& ee,  
۲         list<Entry>::iterator p, list<Entry>::iterator q) {  
۳     // add ee before the element referred to by p  
۴     phone_book.insert(p, ee);  
۵     // remove the element referred to by q  
۶     phone_book.erase(q);  
۷ }
```

- تابع insert(p, elem) عنصری که یک کپی از elem است را به لیست، قبل از عنصری که p به آن اشاره می‌کند می‌افزاید.

- همچنین erase(p) عنصری را که p به آن اشاره می‌کند، از لیست حذف می‌کند.

- برای لیست دو تابع `push_front` و `pop_front` نیز تعریف شده‌اند که می‌توان با استفاده از آنها عنصری را به ابتدای لیست افزود یا از ابتدای لیست عنصری را حذف کرد.
- دلیل این که این عملیات در لیست وجود دارند ولی در وکتور وجود ندارند این است که برای اضافه کردن یک عنصر در ابتدای لیست تنها لازم است عنصری را در حافظه ساخته و اشاره‌گری به اولین عنصر لیست فعلی در آن قرار داد. اما برای اضافه کردن یک عنصر به ابتدای یک وکتور باید همهٔ عناصر وکتور را جابجا کرد.
- تابع `sort` در لیست عناصر آن را مرتب‌سازی می‌کند.
- با استفاده از تابع `remove` می‌توان یک عنصر را با استفاده از مقدار آن از لیست حذف کرد.

```
۱ int myints[] = {17,89,7,14};  
۲ list<int> mylist (myints,myints+4);  
۳ mylist.remove(89);
```

- دو لیست مرتب شده را می توان با استفاده از تابع merge ادغام کرد.

```
۱ int first[] = {5,10,15,20,25};  
۲ int second[] = {50,40,30,20,10};  
۳ vector<int> v(10);  
۴ sort (first,first+5);  
۵ sort (second,second+5);  
۶ merge (first,first+5,second,second+5,v.begin());
```

- پیچیدگی درج در یک وکتور بسته به اینکه وکتور ظرفیت داشته باشد، یا ظرفیت آن تمام شده باشد و نیاز به تخصیص حافظه جدید داشته باشد $O(1)$ یا $O(n)$ است، اما پیچیدگی درج یک عنصر در یک لیست $O(1)$ است.
- البته باید توجه داشت در حالتی که وکتور ظرفیت داشته باشد، برای درج در آن نیاز به تخصیص حافظه نیست، اما در لیست در هر بار اضافه کردن یک عنصر باید یک فضای به اندازه همان یک عنصر در حافظه تخصیص داد.

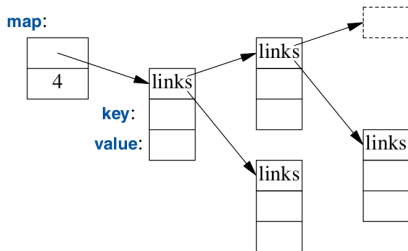
- نگاشت¹ یکی از ساختارهای داده است که توسط آن می‌توان تعدادی زوج را ذخیره سازی کرد. هر زوج شامل یک کلید و یک مقدار است. می‌گوییم یک ظرفِ نگاشت هر کلید را به یک مقدار نگاشت می‌کند و ارتباطی بین یک مجموعه از کلیدها و یک مجموعه از مقادیر را نشان می‌دهد.
- یک نگاشت را آرایهٔ رابطه‌ای یا لغت‌نامه یا دیکشنری² نیز می‌نامیم.
- در یک نگاشت هر کلید یکتاست و دو کلید یکسان را نمی‌توان در یک نگاشت درج کرد. از طرفی در ساختار دادهٔ چندنگاشت² می‌توان دو کلید یکسان درج کرد.

¹ map

² dictionary

² multi-map

- علاوه بر نگهداری زوج‌های کلید و مقدار، یک نگاشت، کلیدها را توسط یک درخت جستجوی دودویی¹ ذخیره‌سازی می‌کند. در درخت جستجوی دودویی، مقادیر همهٔ رئوس زیردرخت سمت چپ در یک رأس پدر، کوچکتر از مقدار رأس پدر است و همچنین مقادیر همهٔ رئوس زیردرخت سمت راست بزرگتر از رأس پدر است. بنابراین جستجو در کلیدهای یک نگاشت با پیچیدگی زمانی کمتر و در نتیجه بهینه‌تر انجام می‌شود.



- پیچیدگی جستجو برای یک نگاشت با n عضو $O(\log n)$ است.

¹ binary search tree

– می‌توانیم از یک نگاشت بدین صورت استفاده کنیم.

```
۱ map<string,int> phone_book { {"Mr. X",123}, {"Mrs. Y", 456} };  
۲  
۳ int get_number(const string& s) {  
۴     return phone_book[s];  
۵ }
```

– یک نگاشت از رشته‌ها و به اعداد صحیح تعریف می‌کنیم. پس در اینجا کلیدها رشته‌ها و مقادیر اعداد صحیح هستند.

- پس یک نگاشت از کلاس map مجموعه‌ای است از زوج‌ها که توسط کلاس زوج pair تعریف شده‌اند.
- یک زوج توسط یک قالب با دو متغیر تعریف شده است. متغیر اول نوع داده‌ای کلید و متغیر دوم نوع داده‌ای مقدار را تعیین می‌کند.

```
۱ pair <string,double> product1; // default constructor
۲ pair <string,double> product2 ("tomatoes",15.5); // value init
۳ pair <std::string,double> product3 (product2); // copy constructor
۴
۵ map<string,double> mymap;
۶ mymap.insert (product1);
۷ mymap.insert (product2);
۸ mymap.insert (product3);
۹ mymap.insert (pair <std::string,double> ("potatoes", 10.5));
```

- عملگر زیرنویس¹ برای نگاشت تعریف شده است، بنابراین می‌توانیم به صورت زیر نیز کلیدها و مقادیر را در آن درج کنیم.

```

۱ map<char, string> mymap;
۲
۳ mymap['a']="an element";
۴ mymap['b']="another element";
۵ mymap['c']=mymap['b'];

```

- پس برای یک کلید در یک نگاشت توسط عملگر زیرنویس، نیازی نیست آن کلید در نگاشت وجود داشته باشد. با استفاده از عملگر زیرنویس، در صورتی که یک کلید در نگاشت وجود نداشته باشد آن کلید در نگاشت درج می‌شود و در صورتی که آن کلید در نگاشت وجود داشته باشد، مقدار آن تغییر می‌کند.

¹ subscript operator

- سازنده نگاشت می‌تواند توسط پیمایشگر یک نگاشت دیگر نیز نگاشت را بسازد.

```
۱ map<char, int> first;  
۲ first['a']=10; first['b']=30;  
۳ first['c']=50; first['d']=70;  
۴ map<char, int> second (first.begin(), first.end());  
۵ map<char, int> third (second);
```

- همانند وکتور، عملگر زیرنویس در نگاشت نیز استثنا ارسال نمی‌کند. می‌توان از تابع `at` برای دسترسی به عناصر نگاشت استفاده کرد. این تابع در صورت خطا استثنا ارسال می‌کند.

- با استفاده از تابع `find` می‌توان در یک نگاشت جستجو کرد. اگر جستجو موفقیت‌آمیز بود، این تابع یک پیمایشگر به کلید یافته شده باز می‌گرداند. اگر کلید مورد نظر یافته نشود، تابع مقدار پیمایشگر `end()` را باز می‌گرداند که در واقع به بعد از آخرین عنصر در نگاشت اشاره می‌کند.
- پیمایشگر در یک نگاشت دو عضو `first` و `second` دارد که به کلید و مقدار در یک زوج اشاره می‌کنند.

- در مثال زیر با دریافت یک حرف الفبای انگلیسی معادل عددی آن را در یک نگاشت جستجو می‌کنیم.

```
۱ map<char, int> m;  
۲ for(int i = 0; i < 26; i++) {  
۳     m.insert(pair<char, int>('A'+i, 65+i));  
۴ }  
۵ char ch;  
۶ cout << "Enter key: ";  
۷ cin >> ch;  
۸ map<char, int>::iterator p;  
۹ p = m.find(ch);  
۱۰ if(p != m.end())  
۱۱     cout << "The ASCII value of" << p->first << " is " << p->second;  
۱۲ else  
۱۳     cout << "Key not in map.\n";
```

- تابع درج (insert) یک زوج بازمی‌گرداند. متغیر دوم در این زوج یک متغیر بولی است. در صورتی که تابع insert موفقیت‌آمیز انجام شود، متغیر دوم در زوج بازگردانده شده، مقدار درست را بازمی‌گرداند و در غیراینصورت مقدار نادرست را بازمی‌گرداند.
- همچنین متغیر اول در زوج بازگردانده شده یک پیمایشگر است.
- در صورتی که کلید مورد نظر برای درج در نگاشت وجود داشت، تابع درج پیمایشگری به کلید یافته شده بازمی‌گرداند و در صورتی که کلید مورد نظر در نگاشت وجود نداشت، کلید و مقدار را درج کرده و پیمایشگری به کلید تازه درج شده بازمی‌گرداند.

```
۱ map<char,int> mymap;  
۲  
۳ // first insert function version (single parameter):  
۴ mymap.insert ( std::pair<char,int>('a',100) );  
۵ mymap.insert ( std::pair<char,int>('z',200) );  
۶  
۷ pair<std::map<char,int>::iterator,bool> ret;  
۸ ret = mymap.insert ( std::pair<char,int>('z',500) );  
۹ if (ret.second==false) {  
۱۰     cout << "element 'z' already existed";  
۱۱     cout << " with a value of " << ret.first->second << '\n';  
۱۲ }
```

- همچنین با استفاده از تابع `erase` می‌توان تعدادی از زوج‌ها را در یک نگاشت حذف کرد.

```

۱ map<char,int> mymap;
۲ map<char,int>::iterator it;
۳
۴ // insert some values:
۵ mymap['a']=10; mymap['b']=20; mymap['c']=30;
۶ mymap['d']=40; mymap['e']=50; mymap['f']=60;
۷
۸ it=mymap.find('b');
۹ mymap.erase(it); // erasing by iterator
۱۰ mymap.erase('c'); // erasing by key
۱۱ it=mymap.find('e');
۱۲ mymap.erase(it, mymap.end()); // erasing by range

```

- حذف می‌تواند توسط یک مقدار یا توسط یک پیمایشگر برای یک عضو یا بازه‌ای از اعضا انجام شود.

- پیمایشگر نگاشت را شبیه به پیمایشگر وکتور و لیست می‌توان استفاده کرد و اعضای نگاشت را پیمایش کرد.

```
۱ // show content:  
۲ for (it=mymap.begin(); it!=mymap.end(); ++it)  
۳     cout << it->first << " => " << it->second << '\n';
```

- چندنگاشت شبیه یک نگاشت است با این تفاوت که کلیدها می‌توانند تکرار شوند.
- تابع `درج` پیمایشگری به کلید `درج` شده بازمی‌گرداند.
- تابع `erase` نیز همهٔ زوج‌ها با یک کلید معین را حذف می‌کند.
- تابع `شمارش` (`count`) تعداد کلیدها را در نگاشت می‌شمارد.
- تابع `equal_range` یک کلید را دریافت کرده و بازه‌ای با شروع از اولین تکرار کلید و پایان با آخرین تکرار کلید مورد نظر را بازمی‌گرداند.

- مقدار بازگردانده شده توسط تابع `equal_range` یک دوتایی است که مقدار اول آن پیمایشگری به ابتدای محدوده یافت شده و مقدار دوم آن پیمایشگری به بعد از انتهای محدوده یافت شده است.

```

۱ std::multimap<char,int> mm;
۲ mm.insert(pair<char,int>('a',10)); mm.insert(make_pair('b',20));
۳ mm['b'] = 30; mm['b'] = 40; mm['c'] = 50; mm['c'] = 60; mm['d'] = 60;
۴ cout << "mm contains:\n";
۵ for (char ch='a'; ch<='d'; ch++) {
۶     pair <multimap<char,int>::iterator,
۷         multimap<char,int>::iterator> ret;
۸     ret = mm.equal_range(ch);
۹     cout << ch << " =>";
۱۰    for (multimap<char,int>::iterator it=ret.first;
۱۱        it!=ret.second; ++it)
۱۲        cout << ' ' << it->second;
۱۳    cout << '\n';
۱۴ }
```



```
۱ multimap<char,int> mm;
۲ mm.insert(make_pair('x',50));
۳ mm.insert(make_pair('y',100));
۴ mm['y']=150; mm['y']=200; mm['z']=250; mm['z']=300;
۵
۶ for (char c='x'; c<='z'; c++) {
۷     cout << "There are " << mm.count(c)
۸         << " elements with key " << c << ":";
۹     multimap<char,int>::iterator it;
۱۰    for (it=mm.equal_range(c).first;
۱۱        it!=mm.equal_range(c).second; ++it)
۱۲        cout << ' ' << (*it).second;
۱۳    cout << '\n';
۱۴ }
```

- تابع `lower_bound` پیمایشگری بازمی‌گرداند به اولین کلیدی که از کلید داده شده توسط تابع بزرگتر یا مساوی است. تابع `upper_bound` پیمایشگری بازمی‌گرداند به اولین کلیدی که از کلید داده شده توسط تابع بزرگتر است.

```

۱ multimap<char,int> mm;
۲ multimap<char,int>::iterator it,itlow,itup;
۳ mm.insert(std::make_pair('a',10));
۴ mm.insert(std::make_pair('b',121));
۵ mm['c']=1001; mm['c']=2002;
۶ mm['d']=11011; mm['e']=44;
۷
۸ itlow = mm.lower_bound ('b');    // itlow points to b
۹ itup = mm.upper_bound ('d');    // itup points to e (not d)
۱۰
۱۱ // print range [itlow,itup):
۱۲ for (it=itlow; it!=itup; ++it)
۱۳     cout << (*it).first << " => " << (*it).second << '\n';

```

از آنجایی که کلیدها در یک نگاشت مرتب شده‌اند، بنابراین اگر بخواهیم از یک کلاس دلخواه به عنوان کلید برای یک نگاشت استفاده کنیم، عملگر کوچکتر < باید برای آن کلاس سربارگذاری شده باشد.

```
۱ class student {  
۲     int id;  
۳     string name;  
۴ public:  
۵     bool operator<(const student& s) { return (id < s.id); }  
۶     // ...  
۷ };  
۸ map<student, double> grades;
```

- به جز کلید، مقدار در یک نگاشت نیز می‌تواند از یک کلاس دلخواه و تعریف‌شده توسط کاربر باشد. اما نیازی نیست عملگر کوچکتر بر روی آن کلاس تعریف شده باشد.

- مجموعه (set) شبیه نگاشت (map) است با این تفاوت که به جای نگهداری یک زوج کلید و مقدار، فقط یک کلید را نگهداری می‌کند.
- پس یک مجموعه ساختار داده‌ای است برای نگهداری کلیدهای یکتا که در یک درخت جستجوی دودویی ذخیره شده‌اند.
- چندمجموعه نیز مانند مجموعه است با این تفاوت که کلیدها می‌توانند تکرار شوند.

- یک پشته ساختار داده‌ای است که داده‌ها را می‌توان به انتهای آن اضافه کرد یا از انتهای آن حذف کرد.
- پشته را می‌توان توسط یک وکتور یا صف دوطرفه ساخت. پس زیرساخت پشته ساختار داده جدیدی نیست بلکه تنها عملگرهای مورد نیاز برای آن تعریف شده‌اند.
- با استفاده از تابع push در پشته درج و با استفاده از تابع pop از پشته مقداری را حذف می‌کنیم.

```

۱ deque<int> mydeque (3,100); // deque with 3 elements
۲ vector<int> myvector (2,200); // vector with 2 elements
۳ stack<int> first; // empty stack
۴ stack<int> second (mydeque); // stack initialized to copy of deque
۵ stack<int, vector<int> > third; // empty stack using vector
۶ stack<int, vector<int> > fourth (myvector);

```

- یک صف ساختار داده‌ای است که داده‌ها را می‌توان به انتهای آن اضافه کرد یا از ابتدای آن حذف کرد. اولین داده‌ای که وارد صف می‌شود، اولین داده‌ای است که از صف خارج می‌شود.
- صف را می‌توان توسط یک لیست یا صف دوطرفه ساخت. پس زیرساخت صف نیز ساختار داده‌ی جدیدی نیست بلکه تنها تابع `push` برای درج و تابع `pop` برای حذف برای آن تعریف شده‌اند. به طور پیش فرض صف با استفاده از صف دوطرفه ساخته می‌شود.

```

۱ deque<int> mydeck (3,100); // deque with 3 elements
۲ list<int> mylist (2,200); // list with 2 elements
۳ queue<int> first; // empty queue
۴ queue<int> second (mydeck); // queue initialized to copy of deque
۵ queue<int,list<int> > third; // empty queue with
۶ // list as underlying container
۷ queue<int,list<int> > fourth (mylist);

```

- نوع خاصی از صف، صف اولویت است که توسط کلاس `priority_queue` پیاده‌سازی شده است. داده‌ها در صف اولویت با استفاده از اولویتشان خارج می‌شود. عنصری که بیشترین اولویت را دارد، به عنوان اولین عنصر از صف خارج می‌شود. بنابراین عملگر مقایسه‌ای کوچکتر باید برای آنها تعریف شده باشند.
- برای مثال اگر صفی با مقادیر صحیح داشته باشیم، بزرگترین عدد اول از همه از صف خارج می‌شود.

```

۱ priority_queue<int> mypq;
۲ mypq.push(30); mypq.push(100); mypq.push(25); mypq.push(40);
۳
۴ cout << "Popping out elements...";
۵ while (!mypq.empty()) {
۶     cout << ' ' << mypq.top();
۷     mypq.pop();
۸ } // Popping out elements... 100 40 30 25

```

- در کتابخانه `<algorithm>` الگوریتم‌هایی تعریف شده‌اند که از آنها می‌توان برای همهٔ ساختار داده‌های کتابخانهٔ استاندارد استفاده کرد.
- برای مثال `all_of` بررسی می‌کند آیا برای همهٔ عناصر یک محدوده از یک ظرف (که با دو پیمایشگر ابتدا و انتها تعیین شده است) شرطی برقرار است یا خیر. تابع `any_of` بررسی می‌کند آیا برای یکی از عناصر یک محدوده شرطی برقرار است یا خیر.
- تابع `for_each` تابعی را بر روی همهٔ عناصر یک محدوده اعمال می‌کند.
- تابع `find` عنصری را در یک محدوده جستجو می‌کند و تابع `find_if` عنصری را جستجو می‌کند در صورتی که برای آن عنصر شرطی برقرار باشد.
- توابع `count` برای شمارش، `search` برای جستجوی یک زیر دنباله، `copy` برای کپی یک محدوده `replace` برای جایگزین کردن یک محدوده، `remove` برای حذف مقادیری در یک محدوده، `remove_if` برای حذف مقادیری در یک محدوده در صورت وجود یک شرط، `sort` برای مرتب سازی، و `merge` برای الحاق دو محدوده از مقادیر به کار می‌روند.

- در بسیاری از توابع کتابخانه الگوریتم از اشاره گر به تابع استفاده شده است.
- می توان به جای استفاده از اشاره گر به تابع از فانکتور یا تابع لامبدا نیز استفاده کرد.

- برای مثال برای شمردن عناصری از یک مجموعه که همگی فرد هستند به صورت زیر عمل می‌کنیم. تابعی که فرد بودن یک عنصر را بررسی می‌کند باید به عنوان یک ورودی به تابع `count_if` به عنوان یک اشاره‌گر به تابع وارد شود.

```

۱ bool IsOdd (int i) { return ((i%2)==1); }
۲
۳ vector<int> myvector;
۴ for (int i=1; i<10; i++) myvector.push_back(i);
۵ // myvector: 1 2 3 4 5 6 7 8 9
۶ int mycount = count_if (myvector.begin(), myvector.end(), IsOdd);
۷ cout << "myvector contains " << mycount << " odd values.\n";
۸ // myvector contains 5 odd values.

```

- می‌توانیم از توابع لامبدا نیز برای وارد کردن یک تابع به تابع دیگر استفاده کنیم.

```
۱ int main () {  
۲     array<int,7> foo = {0,1,-1,3,-3,5,-5};  
۳  
۴     if ( any_of(foo.begin(), foo.end(), [](int i){return i<0;}) )  
۵         cout << "There are negative elements in the range.\n";  
۶  
۷     return 0;  
۸ }
```

- همچنین به جای استفاده از اشاره‌گر به تابع یا توابع لامبدا می‌توانیم از فانکتورها استفاده کنیم.

```

۱ class myclass {                      // function object type:
۲ public:
۳     void operator() (int i) {std::cout << ' ' << i;}
۴ };
۵ vector<int> myvector;
۶ myvector.push_back(10); myvector.push_back(20);
۷ myvector.push_back(30);
۸ myclass op;
۹ cout << "myvector contains:";
۱۰ for_each (myvector.begin(), myvector.end(), op);

```

- رشته‌ها در زبان سی++ با استفاده از وکتور پیاده‌سازی شده‌اند.
- بنابراین همهٔ توابعی که برای وکتور وجود دارند را می‌توان برای رشته‌ها نیز استفاده کرد.
- از پیمایشگرها برای پیمایش رشته‌ها می‌توان استفاده کرد و همچنین از همهٔ توابع کتابخانهٔ الگوریتم نیز می‌توان برای کار با رشته‌ها استفاده کرد.

- در کتابخانه استاندارد بسیاری از توابع مورد نیاز برای کار با ظرفها از قبیل مرتب سازی و جستجو و کپی عناصر وجود دارد.

- برای مثال می توانیم به سادگی عناصر یک وکتور را مرتب و سپس عناصر آن را در یک لیست کپی کنیم.

```
۱ sort(vec.begin(), vec.end()); // use < for order
۲ unique_copy(vec.begin(), vec.end(), lst.begin()); // don't copy adjacent
```

- برای مرتب سازی یک وکتور شامل عناصری از کلاس Entry عملگر < باید برای کلاس تعریف شده باشد.

```
۱ bool operator<(const Entry& x, const Entry& y) { // less than
۲     return x.name<y.name; // order Entries by their names
۳ }
```

- در صورتی که بخواهیم عناصری را به یک وکتور اضافه کنیم از تابع `back_inserter` استفاده می‌کنیم.

```
۱ sort(vec.begin(), vec.end());  
۲ unique_copy(vec.begin(), vec.end(), back_inserter(lst));
```

- با فراخوانی `back_inserter` فضایی در پایان یک ظرف افزوده می‌شود.

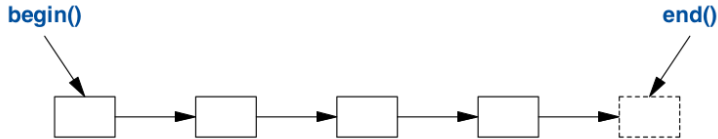
- از تابع find می‌توان برای جستجو در یک ظرف استفاده کرد. در صورتی که عنصر مورد نظر در یک ظرف یافته نشود، پیمایشگر end() بازگردانده می‌شود.

```
۱ auto p = find(s.begin(), s.end(), c);  
۲ if (p != s.end()) return true;  
۳ else return false;
```

- کد بالا را به صورت زیر نیز می‌توانیم بنویسیم.

```
۱ return find(s.begin(), s.end(), c) != s.end();
```

- از آنجایی که همه الگوریتم‌ها در کتابخانه استاندارد با بازه‌های نیمه باز (بازه‌های بسته-باز) کار می‌کنند، پیمایشگر `end()` نیز به بعد از عنصر پایانی در یک ظرف اشاره می‌کند.



- یک پیمایشگر به یک عنصر در یک ظرف اشاره می‌کند. برای وکتور پیمایشگر می‌تواند توسط یک اشاره‌گر تعریف شود، ولی برای ظرف‌های پیچیده‌تر پیمایشگر متناسب با کلاس ظرف مربوطه پیاده‌سازی می‌شود. برای پیمایشگرها عملگرهای ++ و * تعریف شده‌اند.

iterator:

p

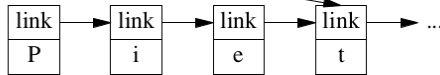
vector:



iterator:

p

list:
elements:



- خلاصه‌ای از الگوریتم‌های کتابخانه الگوریتم به صورت زیر است.

```

۱ // For each element x in [b:e) do f(x)
۲ f=for_each(b,e,f)
۳ // p is the first p in [b:e) so that *p==x
۴ p=find(b,e,x)
۵ // p is the first p in [b:e) so that f(*p)
۶ p=find_if(b,e,f)
۷ // n is the number of elements *q in [b:e) so that *q==x
۸ n=count(b,e,x)
۹ // n is the number of elements *q in [b:e) so that f(*q)
۱۰ n=count_if(b,e,f)
۱۱ // Replace elements *q in [b:e) so that *q==v with v2
۱۲ replace(b,e,v,v2)
۱۳ // Replace elements *q in [b:e) so that f(*q) with v2
۱۴ replace_if(b,e,f,v2)

```

- خلاصه‌ای از الگوریتم‌های کتابخانه الگوریتم به صورت زیر است.

```

۱ // Copy [b:e) to [out:p)
۲ p=copy(b,e,out)
۳ // Copy elements *q from [b:e) so that f(*q) to [out:p)
۴ p=copy_if(b,e,out,f)
۵ // Move [b:e) to [out:p)
۶ p=unique_copy(b,e,out)
۷ // Copy [b:e) to [out:p); 'dont copy adjacent duplicates
۸ p=move(b,e,out)
۹ // Sort elements of [b:e) using < as the sorting criterion
۱۰ sort(b,e)
۱۱ // Sort elements of [b:e) using f as the sorting criterion
۱۲ sort(b,e,f)

```

- خلاصه‌ای از الگوریتم‌های کتابخانه الگوریتم به صورت زیر است.

```
۱ // [p1:p2) is the subsequence of the sorted sequence [b:e)
۲ // with the value v; basically a binary search for v
۳ (p1,p2)=equal_range(b,e,v)
۴ // Merge two sorted sequences [b:e) and [b2:e2) into [out:p)
۵ p=merge(b,e,b2,e2,out)
۶ // Merge two sorted sequences [b:e) and [b2:e2) into [out:p)
۷ // using f as the comparison
۸ p=merge(b,e,b2,e2,out,f)
```

- با استفاده از کتابخانه استاندارد امکان اجرای الگوریتم‌ها به صورت موازی نیز وجود دارد.

```
۱ sort(v.begin(),v.end()); // sequential
۲ sort(seq,v.begin(),v.end()); // sequential (same as the default)
۳ sort(par,v.begin(),v.end()); // parallel
```

برنامه‌سازی همروند

- برنامه‌سازی همروند یا همزمان¹ به معنای اجرای برنامه‌هاست به گونه‌ای که تعدادی از عملیات به طور همزمان اجرا شوند.
- اگر قسمت‌هایی از برنامه به صورت همزمان اجرا شوند، زمان پاسخ (یا تاخیر)² و توان عملیاتی³ برنامه بهبود می‌یابد.

¹ concurrent programming

² response time (or delay)

³ throughput

- به طور مثال محاسبه ضرب دو ماتریس را در نظر بگیرید. از آنجایی که درایه‌های ماتریس حاصلضرب مستقل از یکدیگر قابل محاسبه هستند بنابراین هر یک از درایه‌ها می‌توانند به طور مستقل و موازی با درایه‌های محاسبه شوند (البته در صورتی که تعداد پردازنده‌ها به تعداد کافی باشد) و در این صورت ضرب دو ماتریس با تاخیر کمتری محاسبه خواهد شد.
- حال یک سیستم پردازش تصویر را در نظر بگیرید که در آن تصاویر به ترتیب از ورودی خوانده می‌شوند و پس از چند مرحله پردازش در خروجی نمایش داده می‌شوند. پس از این که اولین مرحله پردازش توسط یک پردازنده انجام شد، در صورتی که تعداد پردازنده‌ها به تعداد کافی باشد، پردازنده اول می‌تواند تصویر دوم را پردازش کند و تصویر اول برای پردازش به پردازنده دوم برود. بدین ترتیب در یک فاصله زمانی معین تعداد بیشتری تصویر پردازش می‌شوند. در این حالت می‌گوییم توان عملیاتی افزایش سیستم افزایش یافته است.

- در برخی مواقع نیاز به همزمانی دو قسمت از برنامه پیدا می‌کنیم، زیرا هر قسمت وظیفه‌ای معین و متفاوت را انجام می‌دهند در حالی که آن دو قسمت باید با هم به طور همزمان اجرا شوند و در ارتباط باشند.
- برای مثال برنامه‌ای را در نظر بگیرید که مقادیری را در خروجی چاپ می‌کند و در صورت دریافت مقداری معین از ورودی برنامه را خاتمه می‌دهد. در این صورت قسمتی از برنامه که مسئولیت چاپ را بر عهده دارد باید به طور موازی با قسمتی از برنامه که مسئولیت دریافت ورودی را بر عهده دارد اجرا شود.

- همهٔ زبان‌های برنامه‌سازی قابلیت‌هایی برای اجرای برنامه‌های موازی دارند. در زبان سی++ نیز در کتابخانهٔ استاندارد، کلاس‌ها و توابعی برای برنامه‌سازی همروند فراهم شده است.
- برنامه‌سازی همروند به دو صورت می‌تواند انجام شود: چند پردازهای¹ و چندریسه‌ای (چندنخی یا چند ریسمانی)².
- یک پروسه یا پردازش³ نمونه‌ای از برنامه است که بر روی فضایی متمایز در حافظه اجرا می‌شود، بنابراین پروسه‌ها با یکدیگر حافظه را به اشتراک نمی‌گذارند. اگر قسمت‌هایی از برنامه به طور کاملاً مستقل بر روی پروسه‌های متمایز اجرا شوند، می‌گوییم برنامه به صورت چندپردازهای اجرا می‌شود.
- یک ریشه (نخ یا ریسمان)⁴ واحدی از دستورات برنامه است که به طور مستقل اجرا می‌شود و حافظه را با دیگر ریشه‌ها در پروسهٔ خود به اشتراک می‌گذارد. اگر قسمت‌های مختلف برنامه بر روی ریشه‌های متمایز اجرا شوند، می‌گوییم برنامه به صورت چندریسه‌ای اجرا می‌شود.

¹ multiprocessing

² multithreading

³ process

⁴ thread

- هر واحد محاسباتی را که می‌تواند به طور همزمان با واحدهای محاسباتی دیگر انجام شود، یک وظیفه¹ می‌نامیم.
- یک وظیفه یا task را یک ریسه یا thread اجرا می‌کند.
- در کتابخانه استاندارد کلاس thread پیاده‌سازی شده است. یک ریسه تابعی را برای اجرا دریافت و اجرا می‌کند. برای همگام‌سازی² ریسه‌ها گاه لازم است برای به پایان رسیدن یک ریسه صبر کنیم تا به پایان برسد. توسط تابع join بر روی یک ریسه، اجرای برنامه متوقف می‌شود تا ریسه مورد نظر به پایان برسد.

¹ task

² synchronization

- در کتابخانه استاندارد کلاس thread پیاده سازی شده است.

```

۱ void hello() { for(int i=0; i<1000; i++) cout << "Hello" << endl; }
۲ void world() { for(int i=0; i<1000; i++) cout << "World" << endl; }
۳ void execute() {
۴     thread t1 {hello}; // hello() executes on one thread
۵     thread t2 {world}; // world() executes on another separate thread
۶     t1.join(); // wait for t1
۷     t2.join(); // wait for t2
۸ }
```

- پس وقتی اجرای برنامه به توابع join می‌رسد، اجرا متوقف می‌شود تا اجرای ریسه‌ها به پایان برسند و پس از آن اجرای برنامه ادامه پیدا کرده و تابع execute به اتمام می‌رسد.

- همچنین می‌توانیم مقدار ورودی توابع را به ریسه‌ها ارسال کنیم.

```
۱ void print(const string & s) { for(int i=0; i<1000; i++) cout << s << endl; }
۲ void execute() {
۳     thread t1 {print, "Hello"}; // print("Hello") executes on one thread
۴     thread t2 {print, "World"}; // print("World") executes on another thread
۵     t1.join(); // wait for t1
۶     t2.join(); // wait for t2
۷ }
```

- مقدار خروجی یک تابع را می‌توانیم توسط یک متغیر مرجع در ورودی تابع دریافت کنیم.

```
۱ void multiply(int x, int y, int & res) { res = x * y; }
۲ void execute() {
۳     int res1, res2;
۴     thread t1 {multiply, 2, 3, res1};
۵     thread t2 {multiply, 4, 5, res2};
۶     t1.join(); // wait for t1
۷     t2.join(); // wait for t2
۸ }
```

- ریسه‌ها می‌توانند با یکدیگر حافظه را به اشتراک بگذارند. در کد زیر ریسه‌ها چه قسمتی را به اشتراک گذاشته‌اند؟

```
۱ void hello() { for(int i=0; i<1000; i++) cout << "Hello" << endl; }
۲ void world() { for(int i=0; i<1000; i++) cout << "World" << endl; }
۳ void execute() {
۴     thread t1 {hello}; // hello() executes on one thread
۵     thread t2 {world}; // world() executes on another separate thread
۶     t1.join(); // wait for t1
۷     t2.join(); // wait for t2
۸ }
```


- وقتی قسمتی از حافظه بین دو ریشه به اشتراک گذاشته می‌شود، ممکن است در هنگام اجرا نتیجهٔ مطلوب برنامه نویس به دست نیاید.
- به طور مثال هر دو تابع hello و world در کد زیر از شیء cout برای چاپ در خروجی استاندارد استفاده می‌کنند. بنابراین ممکن است قبل از اینکه تابع اول چاپ را به پایان برساند، تابع دوم اجرا شود و تابع دوم کنترل خروجی استاندارد را به دست بگیرد و در نتیجه قبل از اتمام چاپ یک رشته در تابع اول، یک رشته در تابع دوم چاپ شود.

```
۱ void hello() { for(int i=0; i<1000; i++) cout << "Hello" << endl; }  
۲ void world() { for(int i=0; i<1000; i++) cout << "World" << endl; }
```

حافظه مشترک

- به عنوان یک مثال دیگر، فرض کنید دو تابع به طور همزمان مقدار یک متغیر counter را افزایش می دهند.
- بنابراین هر ریشه در کد خود counter++ را اجرا می کند.
- در زبان اسمبلی این کد به سه قسمت تقسیم می شود: (۱) خواندن متغیر از حافظه و ذخیره آن بر روی رجیستر پردازنده، (۲) افزایش مقدار، (۳) ذخیره مقدار رجیستر بر روی حافظه.
- حال دو ریشه ممکن است به طور همزمان این عملیات را انجام داده و در نتیجه هر دو همزمان این عملیات را انجام داده و به جای اینکه دو واحد به مقدار متغیر افزوده شود، تنها یک واحد به مقدار آن افزوده شود.
- بنابراین به سازوکاری نیاز داریم که از حافظه مشترک محافظت کنیم و اجازه ندهیم ریشه های مختلف به طور همزمان بر روی حافظه های مشترک عملیات انجام دهد.
- چنین سازوکاری فراهم شده است به گونه ای که هر ریشه قبل از شروع به کار با یک متغیر مشترک، قفلی را در دست گرفته و به دیگران اجازه نمی دهد تا هنگام رهاسازی آن قفل، از متغیر مشترک استفاده کنند.

- فرض کنید چندین ریسه تابع زیر را اجرا کنند. به دلیلی که ذکر شد، پس از اتمام اجرای همه ریسه‌ها، مقدار متغیر counter به مقدار مورد نظر افزایش نمی‌یابد.

```
۱ int counter = 0;
۲ void count() {
۳     for(int i=0; i<1000; i++)
۴         counter++;
۵ }
۶ int main() {
۷     thread t1 {count};
۸     thread t2 {count};
۹     t1.join();
۱۰    t2.join();
۱۱    cout << counter << endl; // counter is less than 2000
۱۲    return 0;
۱۳ }
```

- در زبان سی++ کلاسی به نام mutex به معنی انحصار متقابل¹ سازوکاری را فراهم می‌کند که هر ریسسه بتواند دسترسی به یک دادهٔ مشترک را قفل کرده و منحصرًا بر روی داده کار کند و پس از اتمام کار بر روی داده مشترک قفل را آزاد کرده تا ریسسه‌های دیگر بتوانند از آن داده استفاده کنند.
- بدین ترتیب قبل از دسترسی به حافظه مشترک باید متغیری از کلاس mutex را توسط تابع lock() قفل و پس از دسترسی باید آن را توسط تابع unlock() آزاد کرد.

¹ mutual exclusion

- در مثال زیر هر ریسه در هنگام استفاده از شیء `cout` می‌تواند اطمینان حاصل کند که ریسهٔ دیگری به آن دسترسی ندارد.

```
۱ mutex m;  
۲ void print(string s) {  
۳     for(int i=0; i<1000; i++) {  
۴         m.lock();  
۵         cout << s << endl;  
۶         m.unlock();  
۷     }  
۸ }
```

- در مثال شمارنده، می‌تواند متغیر counter در هر ریسسه با استفاده از یک mutex قفل کرد. بدین ترتیب متغیر به مقدار مورد نظر افزایش پیدا می‌کند.

```
۱ int counter = 0;
۲ mutex m;
۳ void count() {
۴     for(int i=0; i<1000; i++) {
۵         m.lock(); counter++; m.unlock();
۶     }
۷ }
۸ int main() {
۹     thread t1 {count}; thread t2 {count};
۱۰    t1.join(); t2.join();
۱۱    cout << counter << endl; // counter is equal to 2000
۱۲    return 0;
۱۳ }
```

حافظهٔ مشترک

- یک راه حل دیگر این است که متغیر counter با استفاده از کلاس atomic به عنوان یک متغیر تجزیه‌ناپذیر در سطح کد اسمبلی تعریف شود و بدین ترتیب کد اسمبلی مربوط به محاسبات بر روی متغیر به طور تجزیه‌ناپذیر اجرا شده و مشکل مذکور مرتفع می‌شود. در هر ریشه با استفاده از یک mutex قفل کرد. بدین ترتیب متغیر به مقدار مورد نظر افزایش پیدا می‌کند.

```
۱ atomic<int> counter = 0;
۲ mutex m;
۳ void count() {
۴     for(int i=0; i<1000; i++) {
۵         counter++;
۶     }
۷ }
۸ int main() {
۹     thread t1 {count}; thread t2 {count};
۱۰    t1.join(); t2.join();
۱۱    cout << counter << endl; // counter is equal to 2000
۱۲    return 0;
۱۳ }
```

- می‌توان از کلاس `scoped_lock` برای قفل کردن `mutex` استفاده کرد. بدین ترتیب هنگامی که حوزه تعریف `scoped_lock` خارج می‌شویم، قفل به طور خودکار آزاد می‌شود.

```
۱ mutex m; // controlling mutex
۲ void print(string s) {
۳     for(int i=0; i<1000; i++) {
۴         scoped_lock lck {m}; // acquire mutex
۵         cout << s << endl; // manipulate shared data
۶     } // release mutex implicitly after each iteration
۷ }
```

- برای ارسال یک متغیر با ارجاع به تابع یک ریشه از `ref()` استفاده می‌کنیم.

```
۱ mutex m; // controlling mutex
۲ void print(const string & s) {
۳     for(int i=0; i<1000; i++) {
۴         scoped_lock lck {m}; // acquire mutex
۵         cout << s << endl; // manipulate shared data
۶     } // release mutex implicitly after each iteration
۷ }
۸ void execute() {
۹     string s1 = "Hello";
۱۰    string s2 = "World";
۱۱    thread t1 {print, ref(s1)};
۱۲    thread t2 {print, ref(s2)};
۱۳    t1.join();
۱۴    t2.join();
۱۵ }
```

- گاهی داده‌ای بین چندین ریس به اشتراک گذاشته شده است در حالی که تعدادی از ریس‌ها فقط متغیر را می‌خوانند و یک ریس بر روی متغیر می‌نویسد. در چنین مواقعی ریس‌هایی که فقط متغیر را می‌خوانند می‌توانند همگی قفل را دریافت کرده و متغیر را بخوانند. نویسنده تنها در صورتی می‌تواند بر روی متغیر بنویسد که هیچ خواننده‌ای مشغول خواندن آن نباشد. در چنین سناریویی می‌توانیم از `shared_lock` و `unique_lock` استفاده کنیم.

```
۱ shared_mutex mx; // a mutex that can be shared
۲ void reader() {
۳     shared_lock lck {mx}; // willing to share access with other readers
۴     // ... read ...
۵ }
۶ void writer() {
۷     unique_lock lck {mx}; // needs exclusive (unique) access
۸     // ... write ...
۹ }
```

- گاهی یک ریسه نیاز دارد قبل از ادامه کار برای یک رویداد خارجی صبر کند. این رویداد خارجی می‌تواند پیام یا سیگنالی باشد که از یک ریسه دیگر ارسال می‌شود و یا صرفاً این رویداد یک زمان خاص باشد.
- در صورتی که رویداد زمان باشد می‌توان با استفاده از کتابخانه chrono وقفه در اجرای ریسه ایجاد کرد.

```

۱ using namespace std::chrono;
۲ auto t0 = high_resolution_clock::now();
۳ this_thread::sleep_for(milliseconds{20});
۴ auto t1 = high_resolution_clock::now();
۵ duration<double> diff = t1 - t0;
۶ cout << diff.count() << " seconds passed\n";
۷ cout << duration_cast<nanoseconds>(diff).count()
۸     << " nanoseconds passed\n";

```

- برای ارتباط دو ریسه و ارسال سیگنال از یک ریسه به ریسۀ دیگر کتابخانه `condition_variable` طراحی شده است.
- یک ریسه می‌تواند بر روی شیئی از کلاس `condition_variable` منتظر یک رویداد بماند. این رویداد سیگنالی است که از یک ریسۀ دیگر ارسال می‌شود. با دریافت سیگنال و اطلاع از رویداد، ریسۀ در حال انتظار، فعالیت مورد نظر را ادامه می‌دهد.

- برای مثال دو ریشه را در نظر بگیرید که به طور همزمان بر روی یک صف عملیات انجام می‌دهند. یکی از ریشه‌ها اطلاعات را از روی ریشه می‌خواند و ریشه دیگر اطلاعات را بر روی صف می‌نویسد. پس یک ریشه خواننده و یک ریشه نویسنده بر روی این صف عملیات انجام می‌دهند.

```

۱ class Message { // object to be communicated
۲ // ...
۳ };
۴
۵ queue<Message> mqueue; // the queue of messages
۶ condition_variable mcond; // the variable communicating events
۷ mutex mmutex; // for synchronizing access to mcond

```

- ریسۀ خواننده برای اینکه بتواند پیامی را از روی صف بخواند نیاز دارد از خالی نبودن صف اطمینان پیدا کند.
- برای این کار بر روی condition_variable تابع wait را فراخوانی می‌کند.

```

۱ void consumer() {
۲     while(true) {
۳         unique_lock lck {mutex}; // acquire mutex
۴         mcond.wait(lck,[] { return !mqueue.empty(); });
۵         // release lck and wait;
۶         // re-acquire lck upon wakeup
۷         // 'dont wake up unless mqueue is non-empty
۸         auto m = mqueue.front(); // get the message
۹         mqueue.pop();
۱۰        lck.unlock(); // release lck
۱۱        // ... process m ...
۱۲    }
۱۳ }
```

- همچنین ریسۀ نویسنده هر بار پیامی را در صف وارد می‌کند سیگنالی را به خواننده ارسال می‌کند.
- برای این کار بر روی condition_variable تابع notify_one را فراخوانی می‌کند.

```

۱ void producer() {
۲     while(true) {
۳         Message m;
۴         // ... fill the message ...
۵         scoped_lock lck {mutex}; // protect operations
۶         mqueue.push(m); // notify
۷         mcond.notify_one(); // release lock (at end of scope)
۸     }
۹ }
```

- می‌خواهیم برنامه‌ای بنویسیم که مقداری را در خروجی چاپ کند و به طور همزمان مقداری را از ورودی دریافت کرده و در صورتی که مقدار ورودی برابر با حرف q باشد از برنامه خارج شود.


```
1 bool active = true;
2 void input() {
3     char ch;
4     while(active) {
5         ch=getchar();
6         if (ch == 'q') active = false;
7     }
8 }
9 int main() {
10     thread t {input};
11     while(active) {
12         cout << "Hello World!" << endl;
13     }
14     t.join();
15     return 0;
16 }
```

- برنامه‌ای بنویسید که ضرب دو ماتریس را به طور موازی بر روی چند ریشه انجام دهد.