به نام خدا

برنامه‌سازی پیشرفته

آرش شفیعی

مراجع

- برنامه‌سازی: اصول و شیوه‌ها با استفاده از سی‌پلاس‌پلاس، از بیارنه استراستروپ [1]
- سیاحتی در سی‌پلاس‌پلاس، از بیارنه استراستروپ [2]
- زبان برنامه‌سازی سی‌پلاس‌پلاس، از بیارنه استراستروپ [3]
- مرجع کامل سی‌پلاس‌پلاس [4]

[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

تاریخچه

- سیستم‌عامل یونیکس برای اولین بار بر روی یک کامپیوتر PDP۷ با استفاده از زبان اسمبلی توسط دنیس ریچی [1] و کن تامسون [2] در آزمایشگاه‌های بل [3] طراحی و پیاده‌سازی شد.
- یونیکس در نسخهٔ 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 [4] منتشر شد.

[1] Dennis Ritchie
[2] Ken Thompson
[3] AT&T Bell Laboratories
[4] The C Programming Language

مبنای اعداد

- تبدیل اعداد دهدهی [1] به دودویی [2]: $(x)_{۱۰} = (a_n a_{n-۱} \cdots a_۱ a_۰)_۲$ به طوری که $x = \Sigma_{i=۰}^{n} a_i \times ۲^i$ و $a_i \in \{۰, ۱\}$
- مثال: $(۴۲)_{۱۰} = ۱ \times ۲^۵ + ۰ \times ۲^۴ + ۱ \times ۲^۳ + ۰ \times ۲^۲ + ۱ \times ۲^۱ + ۰ \times ۲^۰$
  $(۴۲)_{۱۰} = ۳۲ + ۸ + ۲ = (۱۰۱۰۱۰)_۲$
- اعداد دهدهی می‌توانند علاوه بر قسمت صحیح [3] قسمت اعشاری [4] نیز داشته باشند.
- تبدیل اعداد دهدهی اعشاری به دودویی: $(۰.x)_{۱۰} = (۰.a_۱ a_۲ \cdots a_{n-۱} a_n)_۲$ به طوری که $۰.x = \Sigma_{i=۱}^{n} a_i \times ۲^{-i}$ و $a_i \in \{۰, ۱\}$
- مثال: $(۰.۷۵)_{۱۰} = ۱ \times ۲^{-۲} + ۱ \times ۲^{-۱}$
  $(۰.۷۵)_{۱۰} = ۰.۵ + ۰.۲۵ = (۰.۱۱)_۲$

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

مبنای اعداد

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

مبنای اعداد

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

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

مبنای اعداد

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

مبنای اعداد

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

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

مبنای اعداد

- برای تبدیل یک عدد دودویی علامتدار $(b)_{۲}$ با بیت علامت ۱ به مبنای دهدهی ابتدا مکمل دو b را محاسبه می‌کنیم. فرض کنیم عدد به دست آمده عدد c است. حال عدد c را به مبنای ده تبدیل می‌کنیم.
  $(c)_{۲} = (d)_{۱۰}$
- عدد دودویی b یک عدد منفی است که در مبنای ده برابر است با $-d$ بنابراین $(b)_{۲} = (-d)_{۱۰}$
- برای محاسبهٔ مکمل‌دو [1] یک عدد صفرها را به یک و یک‌ها را به صفر تبدیل کرده، سپس یک واحد به آن عدد می‌افزاییم.
- مثال: عدد بدون علامت $(۱۰۰۱)_{۲}$ در مبنای دو برابر است با ۹.
- اما عدد علامت‌دار $(۱۰۰۱)_{۲}$ در مبنای دو برابر است با -۷.

[1] two's complement

مبنای اعداد

- برای تبدیل یک عدد منفی دهدهی به یک عدد منفی دودویی، ابتدا آن عدد را به صورت مثبت در نظر گرفته، آن را به مبنای دو تبدیل کرده، سپس مکمل‌دو آن را محاسبه می‌کنیم.
- مثال: معادل عدد ۴۲− را در مبنای دو محاسبه کنید.
- $(۴۲)_{۱۰} = (۰۱۰۱۰۱۰)_۲$
  $(-۴۲)_{۱۰} = (۱۰۱۰۱۱۰)_۲$

مبنای اعداد

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

[1] hexadecimal

بدنهٔ برنامه

- یک متن که به زبان سی نوشته شده است را یک برنامهٔ سی [1] می‌نامیم.
- یک برنامهٔ سی در یک فایل سی ذخیره می‌شود و یک فایل سی توسط کامپایلر [2] سی به فایل آبجکت [3] تبدیل می‌شود. محتوای یک فایل آبجکت، برنامهٔ مورد نظر به زبان ماشین مقصد در قالب یک فایل دودویی است. فایل‌های آبجکت توسط یک پیونددهنده یا لینکر [4] 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

بدنهٔ برنامه

- اجرای برنامهٔ سی از تابع بدنه `main` آغاز می‌شود. قبل از تابع بدنه کتابخانه‌های مورد نیاز برای دسترسی به توابع کتابخانه‌ای معرفی می‌شوند.

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

ورودی و خروجی

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

```
#include <stdio.h>
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 [1]) خروجی را تعیین می‌کنند. این زیررشته‌ها را تعیین کنندهٔ فرمت [2] 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 تعیین‌کنندهٔ فرمت می‌تواند c% برای چاپ کاراکتر، d% برای چاپ اعداد صحیح دهدهی، f% برای چاپ اعداد اعشاری، x% برای چاپ اعداد در مبنای شانزده، و یا s% برای چاپ رشته‌ها باشد.
- این تابع تعداد کاراکترهای نوشته شده را در صورت موفقیت بازمی‌گرداند و درغیراینصورت یک عدد منفی بازمی‌گرداند.

[1] format

ورودی و خروجی

- تعیین‌کننده فرمت در حالت کلی به صورت
`%[flag][width][.precision][length] specifier`
است.

- برای مثال در `%010ld` مقدار پرچم [1] ۰ 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. `l` 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

```c
/* printf example */
#include <stdio.h>

int main()
{
   printf ("Characters: %c %c \n", 'a', 65);
   printf ("Decimals: %d %ld\n", 1977, 650000L);
   printf ("Preceding with blanks: %10d \n", 1977);
   printf ("Preceding with zeros: %010d \n", 1977);
   printf ("Some different radices: %d %x %#x \n", 100, 100, 100);
   printf ("floats: %4.2f %E \n", 3.1416, 3.1416);
   printf ("%s \n", "A string");
   return 0;
}
```

خروجی این برنامه به صورت زیر است:

```
Characters : a A
Decimals : 1977 650000
Preceding with blanks :       1977
Preceding with zeros : 0000001977
Some different radices : 100 64 0x64
floats : 3.14 3.141600E+00
A string
```

ورودی و خروجی

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

```
1 #include <stdio.h>
2 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 رشتهٔ `s` ذخیره می‌کند.

انواع داده

- هر متغیر در زبان سی دارای یک نوع داده [1] است که به کامپایلر اجازه می‌دهد دادهٔ قرارگرفته در آن متغیر را تفسیر کند. هر نوع داده اندازهٔ 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 دادهٔ اصلی [2]، انواع دادهٔ تعریف شده توسط کاربر [3]، و انواع دادهٔ مشتق شده [4].

[1] data type
[2] primitive
[3] user-defined
[4] derived

انواع دادهٔ اصلی

انواع دادهٔ 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
`duoble`	اعداد اعشاری (تا ۱۵ رقم اعشار)	۸

انواع دادهٔ اصلی

- هر یک از این انواع داده می‌توانند به صورت علامت‌دار (signed) یا بدون‌علامت (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 اندازهٔ متغیر می‌توانیم از تابع sizeof نیز استفاده کنیم. همچنین در کتابخانهٔ limit.h متغیرهای CHAR_MIN، CHAR_MAX، INT_MIN، INT_MAX، و غیره برای مقدار کمینه و بیشینهٔ هر نوع داده تعریف شده‌اند.

انواع دادهٔ اصلی

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

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

انواع دادهٔ اصلی

- بر روی متغیرها می‌توانیم انواع عملگرهای محاسباتی [1]، رابطه‌ای [2]، شرطی [3]، منطقی [4]، و بیتی [5] 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 دادهٔ اصلی

- در استفاده از عملگرها باید به اولویت یا تقدم [1] آنها توجه کرد.
- برای مثال اولویت ++ (به صورت پسوند) از * (مقدارگیری اشاره‌گر یا رفع ارجاع [2]) 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 -= c == a += ( b -= c)`

[1] precedence
[2] dereference

انواع دادهٔ اصلی

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 دادهٔ 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 دادهٔ تعریف‌شده توسط کاربر

- انواع دادهٔ تعریف‌شده توسط کاربر [1] عبارتند از تعریفی (`typedef`)، ساختمان (`struct`)، اجتماع (`union`)، شمارشی (`enum`).

[1] user-defined data types

انواع دادهٔ تعریف‌شده توسط کاربر

- از نوع تعریفی (typedef) برای تعریف یک نوع دادهٔ 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 دادهٔ تعریف‌شده توسط کاربر

- نوع دادهٔ ساختمان (struct) یک نوع دادهٔ مرکب [1] 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 دادهٔ تعریف‌شده توسط کاربر

- با تعریف یک متغیر نوع دادهٔ Student در حافظه بلوکی با ۴۰ بایت تخصیص داده می‌شود. 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 کارامدتر پردازنده به متغیرها، گاهی در ساختمان‌ها تعدادی بایت توسط کامپایلر اضافه شده، و همیشه طول ساختمان دقیقا 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 دادهٔ تعریف‌شده توسط کاربر

- نوع دادهٔ اجتماع (union) نیز شبیه ساختمان یک نوع دادهٔ 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.

```
union student {
    char name[32];
    int age;
};
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 دادهٔ تعریف‌شده توسط کاربر

- از نوع دادهٔ 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 کنندهٔ 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 دادهٔ تعریف‌شده توسط کاربر

- از نوع دادهٔ 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 دادهٔ مشتق‌شده

- انواع دادهٔ مشتق‌شده [1] عبارتند از آرایه [2] و اشاره‌گر [3].

[1] derived data types
[2] array
[3] pointer

انواع دادهٔ مشتق‌شده

- نوع دادهٔ آرایه مجموعه‌ای از مقادیر که همه از یک نوع داده (اصلی یا تعریف‌شده توسط کاربر) هستند را تعریف می‌کند.
- یک آرایه را چنین تعریف می‌کنیم: `type name[size];`
- بدین صورت از نوع دادهٔ `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 دادهٔ 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 دادهٔ مشتق‌شده

- کامپایلر زبان سی محدودهٔ دسترسی به یک آرایه را بررسی نمی‌کند. بنابراین دسترسی به خانه‌های حافظه‌ای که خارج از محدودهٔ 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 دادهٔ مشتق‌شده

- از آرایه‌ها می‌توانیم برای ذخیرهٔ 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 رشتهٔ 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 دادهٔ مشتق‌شده

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

انواع دادهٔ مشتق‌شده

- یک نوع دیگر از انواع دادهٔ مشتق‌شده اشاره‌گر [1] نام دارد.
- یک اشاره‌گر آدرس یک خانه در حافظه را نگهداری می‌کند. در سیستم‌های ۳۲ بیتی برای نگهداری یک آدرس به چهار بایت نیاز است و در سیستم‌های ۶۴ بیتی به هشت بایت.
- یک اشاره‌گر را به صورت `type * p;` تعریف می‌کنیم.
- `p` به آدرسی از حافظه اشاره می‌کند که در آن متغیری از نوع `type` نگهداری می‌شود. پس `p` یک متغیر هشت بایتی در حافظه است که یک آدرس هشت بایتی را نگهداری می‌کند.

[1] pointer

انواع دادهٔ مشتق‌شده

- برای دسترسی به محتوای حافظه از عملگر رفع ارجاع یا عملگر ستاره (`*p`) استفاده می‌کنیم.
- اگر یک متغیر از نوع دادهٔ اصلی یا تعریف‌شده 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 دادهٔ مشتق‌شده

- 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 خانهٔ حافظهٔ بعدی اشاره کند. نوع اشاره‌گر p مشخص می‌کند که عملگر ++ چند بایت باید اضافه شود. مثلا اگر اشاره‌گر p از نوع عدد صحیح باشد، p++ مقدار متغیر p را به اندازهٔ ۴ بایت افزایش می‌دهد.
- همچنین می‌توانیم از عملگرهای + و - استفاده کنیم و 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 اندازهٔ ۳ واحد اضافه می‌کند و اگر p از نوع عدد صحیح باشد هر واحد از این ۳ واحد ۴ بایت است.
- اشاره‌گر ها را همچنین می‌توانیم از هم کم کنیم ولی نمی‌توانیم با هم جمع، در هم ضرب یا بر هم تقسیم کنیم.
- تفاضل دو اشاره‌گر فاصلهٔ بین خانه‌های حافظه‌ای را مشخص می‌کند که آن دو اشاره‌گر به آنها اشاره می‌کنند.

انواع دادهٔ مشتق‌شده

- با استفاده از کلیدواژهٔ `const` می‌توانیم متغیری تعریف کنیم که مقدار آن تغییر نمی‌کند.
- همچنین در تعریف یک اشاره‌گر می‌توانیم از کلیدواژهٔ `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 `type * const 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 کد [1]، کد برنامه و بخش داده [2]، داده‌ها (مانند متغیرهای عمومی [3] و متغیرهای ایستا [4]) قرار می‌گیرند.
- پشتهٔ فراخوانی [5] 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 هیپ [6] 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

پشتهٔ فراخوانی

- هر گاه یک تابع فراخوانی می‌شود، همه متغیرهای تعریف شده در آن تابع در پشتهٔ فراخوانی [1] ذخیره می‌شوند، و پس از پایان اجرای تابع همهٔ 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 پشتهٔ فراخوانی برای تابع `f` تعریف شده است و هر گاه 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:

```
1  void swap(int x, int y) {
2      int z = x; x = y; y = z;
3  }
4  int main() {
5      int a=2, b=3;
6      swap(a,b);
7      return 0;
8  }
```

- از آنجایی که متغیرهای a و b در پشتهٔ 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 مقدار [1] 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 ارجاع [1] می‌نامیم.

[1] call by reference

فراخوانی با ارجاع

- از آنجایی که فراخوانی با مقدار هزینه دارد بدین معنی که کپی کردن یک متغیر در متغیر دیگر هم زمان‌بر است و هم مقدار حافظهٔ 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 هزینهٔ زمانی و هزینهٔ 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 کلیدواژهٔ `const` برای تعریف متغیرهای تابع استفاده کنیم.

فراخوانی با ارجاع

- کد زیر را در نظر بگیرید:

```
۱ void print(const char * str) {
۲     printf("%s \n", str);
۳ }
۴ int main() {
۵     char string[100];
۶     print(string);
۷     return 0;
۸ }
```

- با اینکه فراخوانی تابع print با ارجاع است، و مقدار string در متغیر str کپی نمی‌شود و str به string اشاره می‌کند (و در نتیجه هزینهٔ فراخوانی کاهش می‌یابد)، اما تابع print نمی‌تواند مقدار متغیر string را تغییر دهد.

تخصیص حافظهٔ پویا

- حافظه را می‌توان توسط کلیدواژهٔ `malloc` به طور پویا تخصیص داد [1].
- 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 حوزهٔ یکی از توابع تعریف شده است، ولی حافظهٔ 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 کلیدواژهٔ `free` می‌توان حافظهٔ تخصیص داده شده در هیپ را آزاد کرد.
- اگر فضاهای تخصیص داده شده آزاد نشوند حافظه رشد می‌کند و مقداری زیادی از حافظهٔ هیپ بلااستفاده می‌ماند.
- دقت کنید که از آنجایی که حافظهٔ هیپ از پشته بزرگتر است، لذا آرایه‌های بسیار بزرگ را بهتر است به طور پویا تخصیص داد.

[1] dynamically allocate

ساختارهای شرطی

- ساختارهای شرطی [1] 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.

```
1 if (<condition>) {
2   // code1
3 } else {
4   // code2
5 }
```

- در صورتی که مقدار `condition` درست باشد `code1` اجرا می‌شود و درغیراینصورت `code2` اجرا می‌شود.

[1] conditional structures

ساختارهای شرطی

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

```
switch (<expression>) {
  case <value1> :
    // code1
    break;
  case <value2> :
    // code2
    break;
  ...
  default :
    //codeD
}
```

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 شاخهٔ `default` اجرا می‌شود.

ساختارهای تکرار

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

```
// initialization
while (<condition>) {
    // code
    // step
}
```

- 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 مجددا سنجیده می‌شود. این حلقه تا زمانی ادامه پیدا می‌کند که شرط برقرار است.

ساختارهای تکرار

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

```
// initialization
do {
  // code
  // step
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.

ساختارهای تکرار

- ساختار `for` به صورت زیر است.

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

- 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 حلقهٔ `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.

ساختارهای تکرار

- 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 تابع [1] تعریف کنیم. برای این کار باید از امضای تابع [2] استفاده کنیم. امضای تابع مشخص می‌کند یک تابع چند ورودی از چه نوع‌های داده دارد و نوع دادهٔ 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 مقداردهی 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 a+b; }
۲  double sub(double x, double y) { return a-b; }
۳  double mul(double x, double y) { return a*b; }
۴  double div(double x, double y) { return a/b; }
۵
۶  int main() {
۷      double (*op[])(double, double) = { add, sub, mul, div };
۸      int i;
۹      double x,y;
۱۰     scanf("%d", i);
۱۱     scanf("%f", x); scanf("%f", y);
۱۲     if (i>=0 && i<4)
۱۳         op[i](x, y);
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 }
```

مثال

- با استفاده از زبان سی یک صف پیاده‌سازی کنید.

مثال

```
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;
```

مثال

```
int peek() {
   return intArray[front];
}

bool empty() {
   return itemCount == 0;
}

bool full() {
   return itemCount == MAX;
}

int size() {
   return itemCount;
}
```

مثال

```
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 }
```

مثال

```
int pop() {
    int data = 0;
    if (!empty()) {
        data = intArray[front++];
        if(front == MAX) {
            front = 0;
        }
        itemCount--;
    }
    return data;
}
```

مثال

```
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    }
```

مثال

```
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  : -1
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
```

مثال

```
    printf("Element at front: %d\n",peek());

    printf("index : 5 4 3 2  1  0\n");
    printf("Queue:  ");

    while(!empty()) {
       int n = pop();
       printf("%d ",n);
    }
}
```

مثال

- 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 اندازهٔ 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;
```

مثال

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

```
Queue construct_queue(int s) {
    Queue res;
    res.intArray =(int *)malloc(sizeof(int)*s);
    res.front = 0; res.rear = -1;
    res.itemCount = 0; res.size = s;
    return res;
}
```

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

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

مثال

- 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  ...
```

مثال

- 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);
```

مثال

- 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.
- 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.