از سی به سی++

مقدمه

شما در درس مبانی کامپیوتر با زبان سی آشنا شدید. زبان سی++ دارای اکثر توابع و فیچرهای زبان سی بوده و در نگاه اول، شباهت زیادی بین این دو دیده می‌شود. این تفکر موجب کثیف شدن کد نوشته شده در سی++ می‌شود. در این مطلب به برخی از تفاوت‌های بین این دو زبان و اشتباهات رایج کسانی که تازه آن را یاد می‌گیرند می‌پردازیم

Namespaces

namespace-ها از اساسی‌ترین مفاهیم سی++ هستند. می‌دانیم که در سی++، همه identifier-ها باید یکتا بوده و در صورتی که ابهامی در تشخیص متغیر یا تابع باشد، به خاطر تداخل نام‌ها ارور کامپایل می‌گیریم. به طور مثال وقتی در دو فایل تابعی با نام یکسان func تعریف شده باشد، هنگام لینک شدن به مشکل برمی‌خوریم.

در سی++ با استفاده از namespace-ها می‌توانیم یک scope جدید تعریف کرده و با تعریف تابع، متغیر و کلاس در آنها، از تداخل نام‌ها جلوگیری کنیم.

چیزی که در هیچ namespace-ای قرار ندارد، به اصطلاح در global namespace قرار دارد:

```
#include <iostream>
float func() {...}
int main() {
  std::cout << func();
  std::cout << ::func();
  return 0;
}
```

در این مثال تابع func در global namespace قرار دارد و می‌توان به صورت func و یا ::func به آن دست یافت. اپراتور :: (با نام Scope Resolution Operator) برای دسترسی به namespace-ها استفاده می‌شود.

یک namespace از قبل تعریف شده به نام std وجود دارد که تمامی توابع کتابخانه استاندارد سی++ در آن قرار دارند. به طور مثال، متغیر cout که از آن برای نمایش خروجی استفاده می‌کنید در این namespace تعریف شده است.

برای تعریف namespace-ها از کلیدواژه namespace استفاده می‌کنیم:

```
namespace example {
  void func() {...}
  int test = 0;
}
int main() {
  return test; // Error
  return example::test; // Correct
}
```

همانطور که می‌بینید، کلیدواژه namespace، scope جدیدی ساخته و identifier-های داخلش را از بقیه کد جدا می‌کند. در این مثال بدون استفاده از example:: نمی‌توان به func یا test دست یافت. namespace-ها می‌توانند به صورت تو در تو (nested) هم باشند:

```
#include <iostream>
float func() {...}
int main() {
  std::cout << func();
```

```
  std::cout << ::func();
  return 0;
}
```

معمولا کتابخانه‌ها کدشان را در namespace-ای قرار می‌دهند تا با کد کاربر تداخلی پیدا نکند. کتابخانه استاندارد سی++ نیز از این قاعده مستثنی نیست و تمامی توابع و کلاس‌های آن در یک namespace به نام std قرار دارد.

قطعا تا الآن using namespace std را استفاده کرده‌اید. دستور using namespace تمامی identifier-های داخل namespace نوشته شده را به scope جایی که دستور زده شده وارد می‌کند. با این کار انگار آن namespace وجود نداشته و مستقیم به داخل آن دست می‌یابیم.

```
namespace example {
  int test = 0;
}
int func() {
  using namespace example;
  return test;
}
int main() {
  return test; // Error
}
```

در این مثال، چون داخل تابع func از using namespace example استفاده شده است، در scope آن (که کل تابع func است) تمامی اجزای namespace-ای به نام example قابل دسترسی اند. اگر using namespace example در اول کد (خارج از همه توابع یا در همان global namespace) زده شده بود، تابع main نیز می‌توانست به آن دسترسی یابد.

برای مثال در مورد std، در صورت زدن using namespace std می‌توانیم به طور مستقیم از string و vector و cout و غیره استفاده کنیم. در صورت نزدن آن، باید std:: را قبل آنها اضافه کنیم: std::string و std::vector و...

استفاده از using namespace std یک bad practice محسوب می‌شود و در کد تمیز نباید از using namespace در global namespace استفاده کرد. این کار احتمال تداخل نام‌ها را بالا می‌برد (مثلا تداخل متغیری به نام max با std::max). تداخل‌ها با استفاده از کتابخانه‌های بیشتر، بسیار زیاد می‌شوند و در کل هدف namespace-ها زیر سوال می‌رود.

با این حال، استفاده از آن در scope-ها ممکن است کاربردی باشد. به طور مثال می‌توانید درباره std::literals::string_literals تحقیق کنید.

Function Overloading

در سی++ قابلیت overload کردن توابع را داریم. این یعنی دو تابع در صورتی که پارامترهای ورودی متفاوتی داشته باشند، می‌توانند نام یکسانی داشته باشند. توجه کنید که بر اساس تایپ ریترن نمی‌توان overload کرد.

```
int func(int a) {...}
int func(int b) {...} // Error
float func(int a) {...} // Error
int func(float a) {...} // Correct
int func(int a, int b) {...} //
Correct
```

Default Arguments

در سی++ می‌توان در تعریف توابع، از مقادیر پیش‌فرض برای آرگومان‌ها استفاده کرد. با این کار اگر در صدا زدن تابع پارامتری مقدار دهی نشده باشد و مقدار پیش‌فرض داشته باشد، مقدار پارامتر برابر مقدار پیش‌فرض می‌شود. توجه کنید که اگر در صدا زدن تابع مقداری به پارامتر داده شده باشد تابع از مقدار پیش‌فرض استفاده نمی‌کند:

```
void func(int x, int y = 1) {...}
func(1, 2); // func(1, 2)
func(1); // func(1, 1)
func(); // Error
```

آرگومان‌های پیش‌فرض باید الزاما در انتهای تعریف تابع باشند. در مورد دلیل این موضوع فکر کنید.

```
void func(int a, int b = 1, int c) {...} // Error
void func(int a, int b = 1, int c = 2) {...} // Correct
```

توجه کنید که اگر تابع را دیکلر می‌کنید، در تعریف آن مقادیر پیش‌فرض را دوباره ننویسید:

```
void func(int a = 2);
void func(int a = 2) {...} // Error
void func(int a) {...} // Correct
```

NULL vs nullptr

در زبان سی از ماکرو NULL استفاده می‌کردیم. این ماکرو معمولا در سی به صورت زیر تعریف می‌شود:

```
#define NULL (void*)0
```

در سی *void به صورت implicit (خودکار) به هر تایپ پوینتری تبدیل می‌شود و برای همین وقتی از malloc استفاده می‌کردیم، می‌شد به صورت زیر نوشت:

```
int* test = malloc(sizeof(int));
```

این کد در سی++ ارور داده و باید به صورت explicit (دستی) کست کرد:

```
int* test = (int*)malloc(sizeof(int));
```

به این خاطر در سی++ ماکرو NULL معمولا به صورت زیر تعریف می‌شود:

```
#define NULL 0
```

توجه کنید که اگر تابع را دیکلر می‌کنید، در تعریف آن مقادیر پیش‌فرض را دوباره ننویسید:

```
void func(int a = 2);
void func(int a = 2) {...} // Error
void func(int a) {...} // Correct
```

در سی++ باید به جای NULL از nullptr استفاده کنیم که مقدار پوینتر نال واقعی دارد و نه مقدار عددی.

یعنی در سی++ از int* a = nullptr و مثلا از time(nullptr) به جای time(NULL) استفاده می‌کنیم.

NULL می‌تواند مشکل‌زا بوده و مثلا در مورد زیر موجب اشتباه می‌شود:

```
void func(int a) {...}
void func(int* a) {...}
func(NULL); // calls the first function
func(nullptr); // calls the second function
```

Type & Namespace Alias

در سی برای ساخت Type Alias-ها از typedef استفاده می‌کردیم:

```
typedef int MyInteger;
```

در سی++ از using استفاده می‌کنیم:

```
using MyInteger = int
```

یا برای اشاره‌گر توابع:

```
typedef void (*MyFunc) (int, float);
using MyFunc = void (*) (int, float);
```

(اینجا MyFunc اشاره‌گر به تابعی است که دو پارامتر int و float گرفته و تایپ ریترن آن void است)

در سی++ می‌توانیم namespace-ها هم alias کنیم:

```
namespace first {
  namespace second {
    void func();
  }
}
namespace test = first::second;
```

```
// alias
test::func();
```

Struct Without Keyword

در سی++ دیگر نیاز به نوشتن کلیدواژه struct قبل از استفاده از تایپ آن نیست. پس نیازی به typedef کردن آن نیز نداریم.

```
struct MyStruct { int value; }
MyStruct instance; // C++
struct MyStruct instance; // C (do not)
typedef struct { int value; } Mystruct; // do not
```

Working With the Heap

در زبان سی برای تخصیص حافظه پویا از malloc استفاده می‌کردیم:

```
struct MyStruct* heapInstance = malloc(sizeof(struct MyStruct));
```

و وقتی کار ما با آن تمام شد، مطمئن بودیم که free می‌کنیم چون در غیر این صورت memory leak خواهیم داشت:

```
free(heapInstance);
```

در سی++ با وجود داده‌ساختارهای std::vector و std::string و غیره، کمتر به تخصیص دستی حافظه heap نیاز پیدا می‌کنیم و خوب است تا جای ممکن از آن اجتناب کنیم. در صورت نیاز، به جای malloc از new استفاده می‌کنیم:

```
MyStruct* heapInstance = new MyStruct();
```

که باید در نهایت توسط delete حافظه آن را برگردانیم:

```
delete heapInstance;
```

برای ساخت آرایه، می‌توان از syntax زیر استفاده کرد:

```
int* array = new int[num]();
delete[] array;
```

توجه کنید که جلوتر می‌خوانید که خوب است منابع (مثلا همین حافظه پویا) را توسط مکانیزم‌های RAII سی++ در کلاس‌ها هندل کنیم و با وجود وکتور نیازی به ساخت آرایه پویا نداریم.

The Boolean Type

سی++ تایپ bool را در اختیار ما می‌گذارد و دیگر نیاز نیست برای متغیرهایی که در سی مقدار 0 یا 1 به آن می‌دادیم، از int استفاده کنیم. تایپ bool مقادیر true و false می‌گیرد و یک بایت فضا اشغال می‌کند.

```
bool a = true;
if (a) { a = false; }
```

Casting

در سی یک نوع cast داریم که به صورت زیر بوده و نباید در سی++ از آن استفاده کرد:

```
char a = 'A';
int b = (int)a;
```

در سی++ چهار نوع cast داریم که برای cast-های معمولی از static_cast استفاده می‌شود:

```
int b = static_cast<int>(a);
```

این cast سخت‌گیرتر بوده و سازگاری تایپ‌ها را بررسی می‌کند و مشکلات را در زمان کامپایل نشان می‌دهد.

به طور مثال کد زیر با C-style cast کامپایل شده ولی با static_cast ارور می‌دهد:

```
int* ptr = (int*)a; // undefined behavior
int* ptr = static_cast<int*>(a); // compile error
```

سه نوع cast دیگر سی++، dynamic_cast و const_cast و reinterpret_cast می‌باشند که جلوتر با dynamic_cast در محیط polymorphic آشنا می‌شوید. می‌توانید درباره دو نوع دیگر تحقیق کنید.

C Standard Library

با اینکه اکثراً جایگزین سی++ برای کتابخانه و فیچرهای زبان سی داریم، گاهی نیاز است که از کتابخانه‌های استاندارد سی استفاده کنیم. به طور مثال، math.h در سی++ هم استفاده می‌شود ولی باید توجه کنیم که برای استفاده از آنها، پسوند h را حذف کرده و به اول آن c اضافه کنیم. یعنی برای استفاده از کتابخانه math.h باید به صورت زیر اینکلود کنیم:

```
#include <cmath>
```

Naming Rules

طبق استاندارد سی++، سه نوع نامگذاری متغیر و توابع برای استفاده کامپایلر و کتابخانه‌های استاندارد رزرو شده اند و ما نباید از آنها استفاده کنیم.

تمام نام‌هایی که از دو underscore پشت سر هم استفاده می‌کنند. (__name یا name__ یا test__name)

تمام نام‌هایی که با یک underscore شروع شده و در ادامه حرف uppercase-ای دارند. (_Name)

تمام نام‌هایی که با یک underscore شروع شده، در ادامه حرف lowercase-ای آمده و در global namespace قرار دارند. (_name)

بنابراین خوب است که کلا از دو underscore پشت سر هم استفاده نکرده و نامی را با یک underscore هم شروع نکنیم.

توجه کنید که استفاده از underscore در انتهای نام موردی ندارد و معمولا متغیرهای private کلاسها را چنین نامگذاری می‌کنند. (name_)

خیلی وقت‌ها header guard به صورت زیر نوشته می‌شود که طبق مطلب گفته شده، درست نمی‌باشد:

```
#ifndef __FILE_NAME_HPP__
#define __FILE_NAME_HPP__
#endif
```

خوب است که نام هدرهای متناظر با فایل‌های cpp.، پسوند hpp. داشته باشند و در هدر گارد هم مانند بقیه جاها از دو underscore استفاده نکنیم.

Pointer vs Reference

با پوینترها آشنایی داریم. می‌دانیم که یک پوینتر صرفا به خانه‌ای در حافظه اشاره می‌کند و پوینتر به هر تایپی، معمولا در سیستم 32 بیتی 4 بایت و در سیستم 64 بیتی 8 بایت فضا می‌گیرد تا آدرس خانه مقصد را ذخیره کند.

در سی++ رفرنس‌ها هم داریم که در واقع همان پوینتر اند ولی سینتکس استفاده از آنها راحت‌تر می‌باشد. خوب است که تا جای ممکن از رفرنس‌ها استفاده کرده و فقط در صورت لزوم از پوینترها استفاده کنیم.

```
// pointer
void func(int* a) { *a = 2; }
int a = 1;
func(&a);
// reference
void func(int& a) {  a = 2; }
int a = 1;
func(a);
```

رفرنس‌ها تفاوت‌های دیگری با پوینتر نیز دارند. رفرنس‌ها نمی‌توانند initialize نشده باشند:

```
int a = 1;
int& b; // Error
int& c = a; // Correct
```

این مقدار اولیه برای رفرنس‌ها برخلاف پوینترها قابل تغییر نیست:

```
int a = 1, b = 2;
int* ptr = &a;
ptr = &b; // changes the pointer to b
int& ref = a;
ref = b; // assigns 2 to a
```

برخلاف پوینتر به پوینتر، رفرنس به رفرنس یا پوینتر به رفرنس نداریم. و در نهایت پوینتر می‌تواند nullptr بشود ولی رفرنس از اول مقدار گرفته و نمی‌تواند نال باشد.

در داده‌ساختارهایی مانند std::vector نیز نمی‌توان از رفرنس‌ها استفاده کرد:

```
std::vector<int&> // Error
std::vector<int*> // Correct
```

Pass by Value

در اینجا صرفا به باطن پاس دادن متغیر می‌پردازیم و صرفا کافیست بدانیم که پاس دادن متغیر به صورت reference به تابع، pass by reference محسوب می‌شود.

در عمل، همه متغیرها در سی و سی++، pass by value می‌شوند و مقدار آنها به تابع کپی می‌شود. مفهوم pass by reference در واقع کپی کردن متغیری از نوع پوینتر است:

```
void func(int* p) { p = &globalX; }
int a = 1;
int* ptr = &a;
func(ptr);
```

پس از اجرای تابع، متغیر ptr همچنان به a اشاره می‌کند. چرا که هنگام پاس دادن ptr به تابع، یک کپی از آن گرفته شده است. از آنجا که این کپی نیز به a اشاره می‌کند، در صورت نوشتن *p = 2 محتوای a تغییر می‌کند ولی خود پوینتر را نمی‌توانیم تغییر دهیم. (احتمالا در مبانی برای این کار آدرس پوینتر (&ptr) را به فرم int** به تابع می‌دادید)

رفرنس‌ها هم در باطن یک پوینتر هستند که محدودیت‌ها و syntax متفاوتی دارند. از آنجا که به طور کلی نمی‌توان رفرنس را re-assign کرد، کد بالا برای رفرنس‌ها معنایی ندارد.

Const Reference

گاهی متغیرها را برای تغییر یافتن به عنوان رفرنس پاس می‌دهیم. ولی گاهی هم از آنجا که آبجکتی که پاس می‌دهیم سنگین محسوب می‌شود، از رفرنس استفاده می‌کنیم که هزینه کپی شدن آن را ندهیم. ولی با این کار امکان تغییر آبجکت در داخل تابع وجود دارد که مطلوب ما نیست. در این حالات از const reference استفاده می‌کنیم:

```
void func(const std::vector<int>&
vec) {...}
```

کلیدواژه const جلوی تغییرات احتمالی رفرنس را می‌گیرد. به رعایت این موضوع const correctness می‌گویند یعنی همه متغیرهایی که قرار نیست تغییر کنند را const می‌کنیم تا خوانایی کد بیشتر شده و احتمال خطا کاهش یابد.

آبجکت سنگین معمولا آبجکتی است که منبعی را مدیریت می‌کند (مثلا هیپ) و از این رو کپی کردن آنها هزینه زیادی دارد. std::vector و std::string از این نوع محسوب می‌شوند.

توجه کنید که کانست رفرنس کردن fundamental type-ها (مانند int, float, char, bool) نه تنها هزینه کپی را کم نمی‌کند (چون پوینتر هم مانند آنها باید کپی شود)، بلکه هزینه دسترسی به متغیر را بیشتر می‌کند (یک indirection اضافی بین متغیر و مقدار آن در حافظه قرار می‌گیرد).

همچنین در مورد const correctness، معمولا متغیرهای pass by value توابع را const نمی‌کنیم. این به این دلیل است که آنها به تابع کپی می‌شوند و اینکه const هستند یا خیر، تأثیری در بیرون تابع نمی‌گذارد و اطلاعات مفیدی به ما نمی‌دهد.

```
void func(const int& a); // bad
void func(const int a); // not useful
void func(int a); // good
void func(std::string s); // bad
void func(std::string& s); // not
good if we don't want to change the
string
void func(const std::string& s); //
good
```

Const & Constexpr

در زبان سی، برای تعریف constant-ها از define استفاده می‌کردیم:

```
#define CONSTANT 10
```

در زمان کامپایل، بخش pre-processor هر جای کد که از CONSTANT استفاده شده را با 10 جایگزین می‌کند.

در سی++، از const و constexpr استفاده می‌کنیم.

```
constexpr int CONSTANT = 10;
```

یکی از مزایای این روش نسبت به روش define، این است که تایپ متغیر مشخص می‌باشد. از آنجا که همه چیز را نمی‌توان به صورت constexpr (که باید در زمان کامپایل مشخص باشد) نوشت، می‌توانیم از const هم استفاده کنیم:

```
const std::string TITLE = "test";
```

دقت کنید که با گذاشتن خط بالا در فایل هدر و اینکلود کردن آن در چند فایل، هر فایل کپی خود را از این متغیر ثابت دریافت می‌کند. جهت اشتراک یک متغیر بین چند فایل، می‌توانید درباره extern تحقیق کنید.

همچنین می‌توانید در مورد محدودیت‌های استفاده از constexpr مطالعه کنید.

Variable Length Array

یکی از فیچرهای زبان سی، VLA-ها هستند که به صورت زیر تعریف می‌شوند:

```
int n = 12;
int arr[n];
```

همانطور که می‌دانیم، اندازه آرایه باید در زمان کامپایل مشخص باشد ولی اینجا از یک متغیر برای آن استفاده شده است و به نوعی انگار آرایه دینامیک ساخته می‌شود.

از آنجا که آرایه بر روی استک قرار دارد، این کار در صورت بزرگ بودن n می‌تواند منجر به stack overflow بشود. این فیچر به طور کلی منجر به نوشتن کد ناامن شده و استفاده از آن توصیه نمی‌شود؛ چرا که اگر می‌دانیم مقدار n کم است، پس می‌توانیم به آرایه اندازه ثابتی بدهیم و اگر مقدار n را نمی‌دانیم، باید از heap استفاده کنیم.

این فیچر در استاندارد سی++ وجود ندارد ولی کامپایلرها معمولا به صورت extension آن را ساپورت می‌کنند. در سی++ نباید هیچ وقت از VLA-ها استفاده کنیم و می‌توانیم std::vector را یک جایگزین امن برای آن به شمار بیاوریم.

Data Structures

سی++ داده‌ساختارهای متعددی دارد که جلوتر با آنها آشنا خواهید شد. در اینجا به سه نکته اشاره می‌کنیم.

در زبان سی برای تعریف آرایه با اندازه ثابت بر روی استک، به صورت زیر می‌نوشتیم:

```
int arr[10];
```

همانطور که احتمالا در مبانی به آن برخوردید، پاس دادن این آرایه به توابع و کپی گرفتن از آن کار راحتی نیست. از این رو در سی++ می‌توانیم از معادل آن یعنی std::array استفاده کنیم:

```
#include <array>
std::array<int, 10> arr;
```

کار کردن با این آرایه همانند کار کردن با وکتور بوده و با توابع <algorithm> نیز همخوانی دارد.

در هدر <utility>، std::pair تعریف شده که مانند یک struct با دو متغیر است:

```
#include <utility>
std::pair<int, char> a = {10, 'b'};
std::cout << a.first << a.second;
```

همچنین در هدر <tuple>، std::tuple تعریف شده که می‌تواند تعداد ثابتی از متغیرها را نگه دارد:

```
#include <tuple>
std::tuple<int, int, bool> a = {1, 2,
true};
std::cout << std::get<0>(a); //
prints 1
```

استفاده از tuple-ها توصیه نمی‌شود و هرجا که مقادیر بیشتری را می‌خواهیم با هم داشته باشیم، خوب است که از یک struct استفاده کنیم. با این کار چون متغیرها نام دارند، خوانایی کد بیشتر می‌شود.