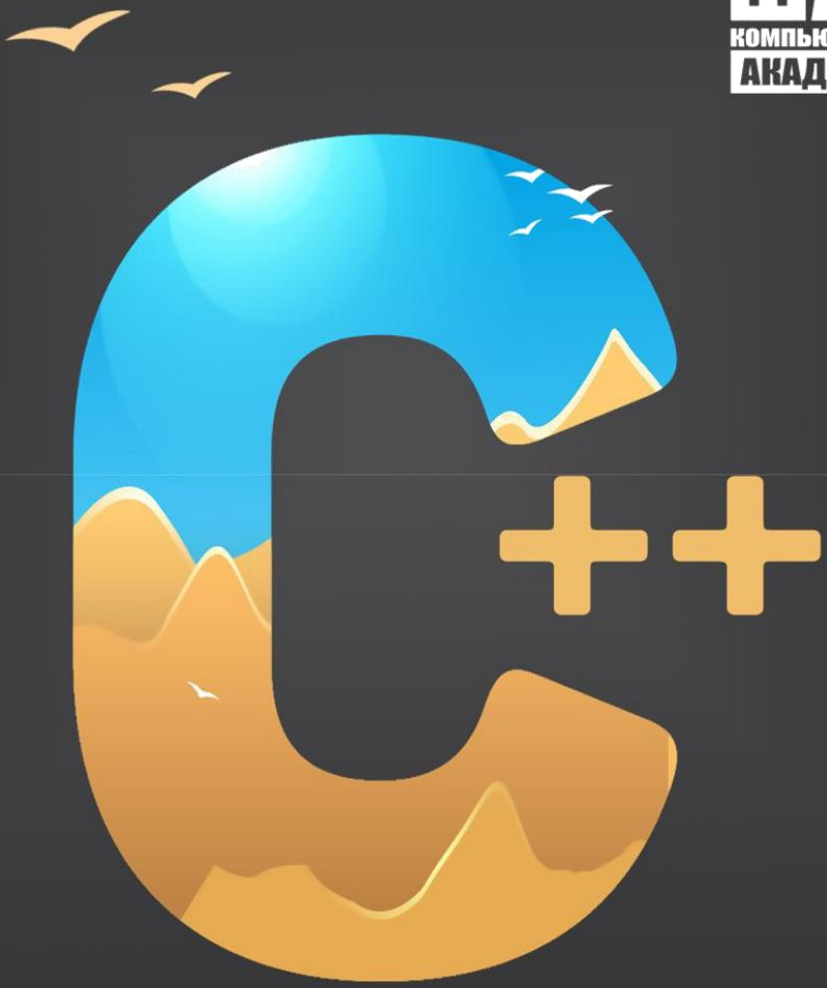
ШАГ
компьютерная
АКАДЕМИЯ



ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ C++

Dərs № 18

Preprosessor və onun istifadəsi

Mündəricat

Dərs materialları bu PDF faylına əlavə edilmişdir. Materiallara daxil olmaq üçün dərs, Adobe Acrobat Reader proqramında açılmalıdır.

1. Preprosessor

Preprosessor — orijinal proqram mətni kompilyasiya edilməzdən əvvəl, onun üzərində bəzi (bəzən olduqca əhəmiyyətli) manipulyasiyalar həyata keçirən proqramdır. İngilis dilindən hərfi tərcümədə **preprocessor** sözü **ilkin emal** deməkdir.

Preprosessorlar kompilyatorlar üçün giriş mətni yaradır və aşağıdakı funksiyaları yerinə yetirə bilirlər:

- makroların emal olunması;
- faylların daxil edilməsi;
- "rasional" ilkin emal;
- dil genişlənmələri.

Məsələn, proqramlarda çox vaxt "heç nə deməyən" ədədlərdən istifadə etmək lazım olur. Bunlar proqramda istifadə olunan bəzi riyazi sabitlər və ya massivlərin ölçüləri ola bilər və s. Ümumi olaraq qəbul edilmişdir ki, belə sabitlərin çoxluğu proqramların başa düşülməsini çətinləşdirir və pis proqramlaşdırma üslubunun əlaməti hesab olunur. Proqramçılar arasında bu sabitlər, rişxəndlə, "***magic numbers***" adlandırılmışdır. Proqramın onlarla zəngin olmaması üçün, proqramlaşdırma dilləri sabitə ad verməyə və sonra sabitin özü əvəzinə onu hər yerdə istifadə etməyə imkan verir.

Bu sehrli effektdən qaçınmaq üçün, siz artıq **const** açar sözü ilə sabitlər yaratmısınız .

Bu effektlə mübarizə aparmağın ikinci yolu preprosessor tərəfindən təmin edilir. Məsələn, təriflərdən istifadə edərək:

```
#define P1    3.14159
#define E     2.71284
```

preprosessor, proqramdakı bütün **P1** və **E** adlarını müvafiq ədədi sabitlərlə əvəz edəcəkdir. İndi, natural loqarifmlərin əsasının təxmini dəyərini səhv yazdığınızı öyrəndiyiniz zaman, bütün proqram üzərindən keçmək yerinə, yalnız sabitin elanı ilə olan tək bir sətri düzəltmək kifayətdir:

```
#define E 2.71828
```

C++ dilinin preprosessoru, yalnız sabitləri deyil, həm də bütün proqram konstruksiyalarını yenidən müəyyən etməyə imkan verir. Məsələn, belə elan yazmaq olar:

```
#define forever for(;;)
```

və sonra, bu formada hər yerdə sonsuz dövrlər yazmaq olar:

```
forever { <loop body> }
```

və əgər fiqurlu mötərizələri xoşunuza gəlmirsə, onda müəyyən edin

```
#define begin {
#define end }
```

və daha sonra, operator mötərizələri yerinə **begin** və **end** istifadə edin, Paskal dilində edildiyi kimi məsələn.

Makro insruksiyalar (makrolar) adlanan bu cür yazılar, parametrlərə malik ola bilərlər (və bununla daha güclü ola bilərlər), lakin bu barədə daha sonra danışacağıq.

Preprosessorun digər mühüm xüsusiyyəti — digər faylların məzmununun mənbə mətnə daxil edilməsidir. Bu xüsusiyyət, əsasən, proqramları, bütün fayllar üçün bəzi ümumi təriflərlə təmin etmək üçün istifadə olunur. Məsələn, C++ dilində proqramının əvvəlində sizə yaxşı tanış olan preprosessor konstruksiyasına rast gəlmək olar:

```
#include <iostream>
```

Proqramın mənbə kodu preprosessor tərəfindən emal edildikdə, bu instruksiya, I/O axınlarının işləməsi və verilənlərin elan edilməsi üçün lazım olan makro instruksiyaları özündə saxlayan **iostream** faylının məzmunu ilə əvəz olunur.

Preprosessor operatoru (direktiv) — "#" simvolu ilə başlayan, ardınca operator adı (**define**, **pragma**, **include**, **if**) və operandlar gələn mənbə kodun bir sətridir. Preprocessor operatorları, proqramın istənilən yerində rast gələ bilər və onlar bütün mənbə faylına tətbiq olunur.

Gəlin makroların istifadəsinə daha ətraflı nəzər salaq.

2. #define vasitəsi ilə sabitlərin təyini

#define operatoru daha çox simvolik sabitləri təyin etmək üçün istifadə olunur. O, mənbə faylın istənilən yerində rast gələ bilər və onun qüvvəsi, göründüyü yerdən faylın sonuna qədər etibarlıdır.

Qeyd: *Simvolik sabitin elanı sonunda (#define ifadəsinin sonunda) nöqtəli vergül qoyulmur!*

```
#define min 1
#define max 100
```

Proqram mətnində, 1 və 100 sabitlərinin yerinə, müvafiq olaraq, **min** və **max**-dan istifadə etmək olar.

```
#include <iostream>
using namespace std;

#define NAME "William Shakespeare"

int main ()
{
    cout << " My name is " << NAME;
    return 0;
}
```

Proqram nəticəsi:

```
My name is William Shakespeare
```

Qeyd: *Sətirlərin, simvol sabitlərinin və şərhlərin içindəki mətn dəyişdirilə bilməz, çünki sətirlər və simvol sabitləri C++ dilində bölünməz dil nişanələridir (tokenləridir).*

Beləliklə, aşağıdakı makro instruksiyasından sonra

```
#define YES 1
```

bu operatorda

```
cout << "YES";
```

heç bir makro əvəzetmə aparılmayacaq.

Proqram mətnindəki əvəzləmələr, aşağıdakı əmrdən istifadə edərək ləğv edilə bilər:

```
#undef <name>
```

Bu direktivin yerinə yetirilməsindən sonra, preprosessor üçün ad qeyri-müəyyən olur və onu yenidən təyin etmək olar. Məsələn, aşağıdakı direktivlər, xəbərdarlıq mesajlarına səbəb olmayacaqlar:

```
#define M 16
#undef M
#define M 'C'
#undef M
#define M "C"
```

#undef direktivi, müxtəlif vaxtlarda və ya müxtəlif proqramçılar tərəfindən yazılmış, ayrı-ayrı "mətn parçalarından" yığılan böyük proqramlar hazırlanan zaman faydalıdır. Bu halda, müxtəlif obyektlərin eyni nişanlamaları baş verə bilər. Mənbə fayllarını

dəyişdirməmək üçün, daxil edilmiş mətn, müvafiq **#define — #undef** direktivləri ilə "çərçivəyə salına" və bununla da, mümkün səhvlər aradan qaldırıla bilər.

Əgər token_string uzun olarsa, bu zaman, o, proqram mətninin növbəti sətrində davam etdirilə bilər. Bunun üçün, davam edən sətrin sonunda "\" simvolu qoyulur. Preprosessorun işləməsi mərhələlərindən birində, bu simvol sonrakı sətir sonu simvolu ilə birlikdə proqramdan silinəcəkdir. Məsələn:

```
#define LINE "First Line\n"\
             "Second Line\n"
…….
cout << LINE;
```

Ekrana çıxarılacaq:

```
First Line
Second Line
```

Xatırladaq ki, biz, 9-cu Dərsdə daxili yerləşdirmə haqqında öyrəndiyimiz zaman, makrolar yaratmaq üçün də **#define** direktivindən istifadə etmişik. Parametrləri olan makro nümunəsi:

```
#define Mult(a, b) (a)*(b)
```

Makro, iki arqumenti bir-birinə vurur. Makronun istifadəsi. Nəticədə, ekranda 6-dır.

```
cout << Mult(2, 3);
```

9-cu dərsə qayıtmağı və keçirilən materialı təkrarlamağı tövsiyə edirik.

3. Şərti kompilyasiya

Şərti kopmilyasiya direktivləri, proqram kodunu, müəyyən şərtlərin mümkünlüyündən asılı olaraq yaratmağa imkan verir. Şərti kompilyasiya C++ dilində əmrlər toplusu ilə təmin edilir, hansılar ki, mahiyyətcə, kompilyasiyaya deyil, preprosessor emalına nəzarət edir:

```
#if <constant_expression>
#ifdef <identifier>
#ifndef <identifier>
#else
#endif
#elif
```

İlk üç əmr şərtləri yoxlayır, sonrakı ikisi — yoxlanılan şərtin əhatə dairəsini müəyyən etməyə imkan verir. Sonuncu əmr, bir sıra şərtlərin yoxlanması təşkili üçün istifadə olunur. Şərti kompilyasiya direktivlərinin tətbiqi üçün ümumi struktur aşağıdakı kimidir:

```
#if/#ifdef/#ifndef <constant_expression or identifier>
<text_1>
#else // ixtiyari olan <text_2> direktividir
#endif
```

- **#else <text_2>** konstruksiyası ixtiyaridir.
- text_1 kompilyasiya edilən mətnə yalnız yoxlanılan şərt doğru olduqda daxil edilir.

- Əgər şərt yanlışdırsa, text_2, **#else** direktivi olduğu halda, kompilyasiyaya göndərilir.
- Əgər **#else** direktivi yoxdursa, şərt yanlış olduqda **#if**-dən **#endif**-ə qədər olan bütün mətn buraxılır.

#if əmrlərinin formaları arasındakı fərq ibarətdir:

1. **#if** direktivlərinin birincisində, sabit tam ədədi ifadənin qiyməti yoxlanılır. Əgər o, sıfırdan fərqlidirsə, yoxlanılan şərt doğru hesab olunur. Məsələn, direktivlərin icrası nəticəsində:

```
#if 5 + 12
    <text_1>
#endif
```

   text_1 həmişə kompilyasiya edilən proqrama daxil ediləcək.
2. **#ifdef** direktivində, **#ifdef**-dən sonra yerləşdirilmiş identifikatorun hazırda **#define** əmri ilə müəyyən edilib-edilmədiyini yoxlayır. Əgər identifikator müəyyən edilmişdirsə, text_1 kompilyator tərəfindən istifadə olunur.
3. **#ifndef** direktivində əks şərt yoxlanılır — identifikatorun müəyyən edilməmiş olması doğru hesab olunur, yəni, identifikatorun **#define** əmri ilə istifadə edilmədiyi və ya onun elanının **#undef** əmri ilə ləğv edildiyi hal.

Proqramın mənbə kodunun preprosessor tərəfindən emalı zamanı budaqlanmanı təşkil etmək üçün aşağıdakı direktiv daxil edilmişdir:

```
#elif <constant_expression>
```

#else#if konstruksiyasının qısaltmasıdır. Bu direktivdən istifadə edilən mənbə kodun strukturu aşağıdakı kimidir:

```
#if <constant_expression_1>
<text_1>
#elif <constant_expression_2>
<text_2>
#elif <constant_expression_3> <text_3>
.    .    .    .
#else
<text_N>
#endif
```

- Preprosessor əvvəlcə **#if** direktivindəki şərti yoxlayır, əgər şərt yalandırsa (0-a bərabərdirsə) — **constant_expression_2** hesablanır, əgər bu 0-a bərabərdirsə, **constant_expression_3** hesablanır və s.
- Əgər bütün ifadələr yalandırsa, bu zaman **#else** halının mətni kompilyasiya edilən mətnə daxil edilir.
- Əks halda, yəni, ən azı bir doğru ifadə olduqda (**#if** və ya **#elif**-də), bu direktivdən dərhal sonra gələn mətn emal edilir, bütün digər direktivlər isə nəzərə alınmır.
- Beləliklə, preprosessor həmişə, şərti kompilyasiya əmrləri ilə seçilmiş mətn bölmələrindən yalnız birini emal edir.

İndi isə, bir neçə nümunəyə nəzər salaq.

Nümunə 1. Sadə şərti daxiletmə direktivi

```
#ifdef ArrFlg
int arr[30];
#endif
```

Əgər direktivin interpretasiyası zamanı **ArrFlg** makro instruksiyası müəyyən edilərsə, yuxarıdakı qeyd, növbəti ifadəni yaradır

```
int arr[30];
```

Əks halda, heç bir ifadə yaradılmayacaq.

Nümunə 2

```
#include <iostream>
using namespace std;

#define ArrFlg 1

int main(){
    #ifdef ArrFlg
        int arr[30] = { 1, 21 };
        cout << arr[0] << " " << arr[1];

    #else
        cout << "Array is not defined!";

    #endif
    return 0;
}
```

Nümunə 3. Alternativli şərti daxiletmə direktivi

```
#if a + b == 5
cout << 5;
#else
cout << 13;
#endif
```

Əgər **a + b == 5** məntiqi ifadəsi doğrudursa, **cout << 5;** əmri yaradılacaq, əks halda, **cout << 13;** əmri yaradılacaq;.

Nümunə 4. Mürəkkəb şərti daxiletmə direktivi

```
#include <iostream>
using namespace std;

#define FirstVal 3
#define SecondVal 3

#if FirstVal*SecondVal == 9
    int main(){
        cout << "9\n";
        return 0;
    }

#elif FirstVal*SecondVal == 15
    int main() {
        cout << "15\n";
        return 0;
    }

#else
    int main() {
        cout << "Not 9 or 15\n";
```

```
        return 0;
    }
#endif
```

Yuxarıdakı yazının interpretasiyası nəticəsində yaradılacaq:

```
int main(){
        cout << "9\n";
        return 0;
}
```

4. Digər preprosessor direktivləri

Bizə məlum olan direktivlərdən əlavə, bir neçə daha direktivlər var, onlardan bəziləri bunlardır:

1. Sətirlərin nömrələnməsi üçün bu direktivdən istifadə etmək olar:

```
#line <constant>
```

hansı ki kompilyatora, aşağıdakı mətn sətri tam ədədli onluq sabiti ilə müəyyən edilən nömrəsi olduğunu bildirir. Əmr, yalnız sətir nömrəsini deyil, həm də fayl adını müəyyən edə bilər:

```
#line <constant> "<file_name>"
```

2. Direktiv

```
#error <token_sequence>
```

tokenlər ardıcıllığı şəklində diaqnostik mesajın çıxışa verilməsi ilə nəticələnir. **#error** direktivi, tez-tez şərti preprosessor əmrləri ilə istifadə olunur. Məsələn, bir **NAME** preprosessor dəyişənini təyin etməklə

```
#define NAME 5
```

daha sonra onun dəyərini yoxlaya və **NAME** dəyəri fərqli olarsa, çıxışa mesaj vermək olar:

```
#if (NAME != 5)
   #error "NAME must be equal to 5!"
```

Mesaj belə görünəcəkdir:

```
fatal: <file_name> <line_number>
#error: NAME must be equal to 5!
```

3. Əmr

```
#pragma <token_sequence>
```

kompilyatorun konkret icrasından asılı olan hərəkətləri müəyyənləşdirir və kompilyatora müxtəlif göstərişlər verməyə imkan verir.

4. C++ dilində # və ## operatorları ilə işləmək imkanı var. Bu operatorlar **#define** direktivi ilə ittifaqda istifadə olunur.

■■ # operatoru özündən əvvəl gələn arqumenti, dırnaqlar içərisinə alınmış sətirə çevirir.

```
#include <iostream>
using namespace std;

#define mkstr(s) #s
int main(){
    // Kompilyator üçün: cout << "I love C++";
    cout << mkstr(I love C++);
    return 0;
}
```

■■ ## operatoru iki tokenin konkatenasiyası (birləşdirilməsi) üçün istifadə olunur

```
#include <iostream>
using namespace std;
```

```
#define concat(a, b) a##b

int main(){
    int xy = 10;
    // Kompilyator üçün: cout << xy;
    // ekranda – 10
    cout << concat(x, y);
    return 0;
}
```

5. Layihənin fayl toplusuna bölünməsi

Uzun müddətdir bildiyiniz kimi, fayldan mətn daxil etmək üçün **#include** əmri istifadə olunur. Onunla daha yaxından tanış olmaq vaxtıdır. Bu əmr, preprosessor direktividir və iki formaya malikdir:

```
#include <file_name> // Ad bucaqlı mötərizələrdədir.
#include "file_name" // Ad dırnaqlar içərisindədir.
```

Əgər **file_name** — bucaqlı mötərizələr daxilindədirsə, onda preprosessor, faylı standart sistem qovluqlarında axtarır. Əgər **file_name** — dırnaqlar daxilindədirsə, onda preprosessor əvvəlcə istifadəçinin cari qovluğuna baxır və yalnız bundan sonra standart sistem qovluqlarına baxmağa müraciət edir.

C++ dili ilə işləməyə başlayarkən, biz dərhal proqramlarda giriş-çıxış vasitələrinin istifadəsi ehtiyacı ilə qarşılaşmışdıq. Bunun üçün proqram mətni əvvəlində direktivi yerləşdirirdik:

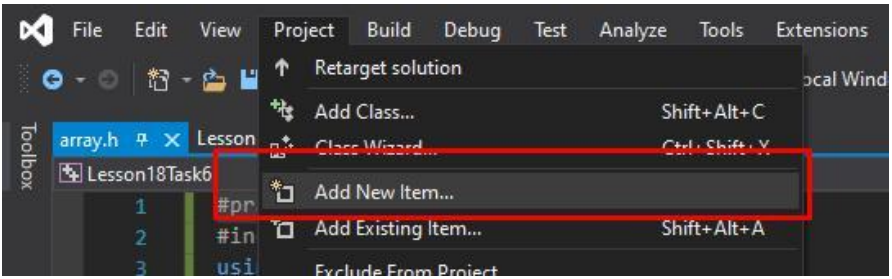
```
#include <iostream>
```

Bu direktivi yerinə yetirməklə, preprosessor proqrama Input/Output kitabxanası ilə əlaqəli vasitələri daxil edir. **iostream** faylının axtarışı, standart sistem qovluqlarında aparılır.

Başlıq faylları, böyük proqramların modullu tərtibatında çox effektiv vasitədir. Həmçinin, C++ dilində,

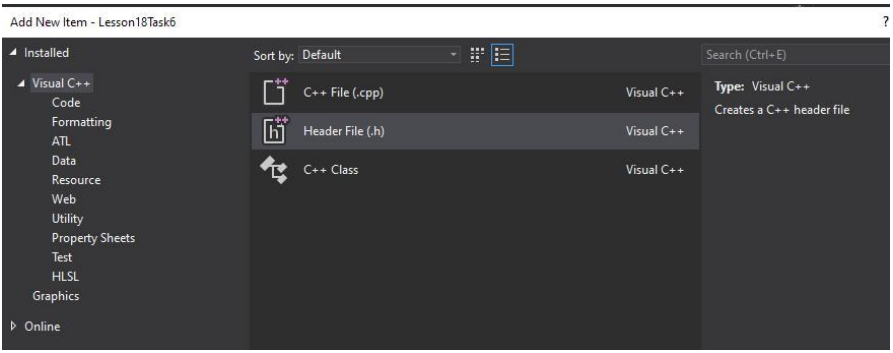
proqramlaşdırma praktikasında belə bir hal geniş yayılmışdır ki, proqramda bir neçə funksiyadan istifadə olunarsa, bu funksiyaların mətnlərini ayrıca faylda saxlamaq rahatdır. Proqram hazırlanarkən istifadəçi **#include** əmrləri vasitəsilə orada istifadə olunan funksiyaların mətnlərini daxil edir.

Layihəyə yeni başlıq faylının əlavə edilməsi üçün **Project -> Add new item** menyusunu seçmək lazımdır.



Şəkil 1

Açılan pəncərədə **Header File(.h)** seçin, fayl adını daxil edin və yadda saxlamaq üçün yolu seçin. Yaradılmış boş fayl layihənizə əlavə olunacaq.



Şəkil 2

Proqramın bir neçə fayla bölünməsinə nümunə olaraq, massivdə minimumun, massivdə maksimumun tapılması və massivin göstərilməsi məsələlərinə baxaq. Massiv ilə işləmək üçün funksiyalar, **array.h** adlanan ayrıca faylda yerləşir.

Əsas fayl:

```
#include <iostream>
/*
    Massiv ilə işləmək üçün funksiyaların
    olduğu fayl
*/

#include "array.h"
using namespace std;

int main(){
    const int size = 5;
    int arr[size] = { 8, 44, 67, 12, 13 };
    // Massivin göstərilməsi
    ShowArray(arr, size);
    cout << endl;
    // minimum
    cout << "Minimum: " << GetMin(arr, size)<<endl;
    // maksimum
    cout << "Maximum: " << GetMax(arr, size) << endl;
    return 0;
}
```

array.h başlıq faylı:

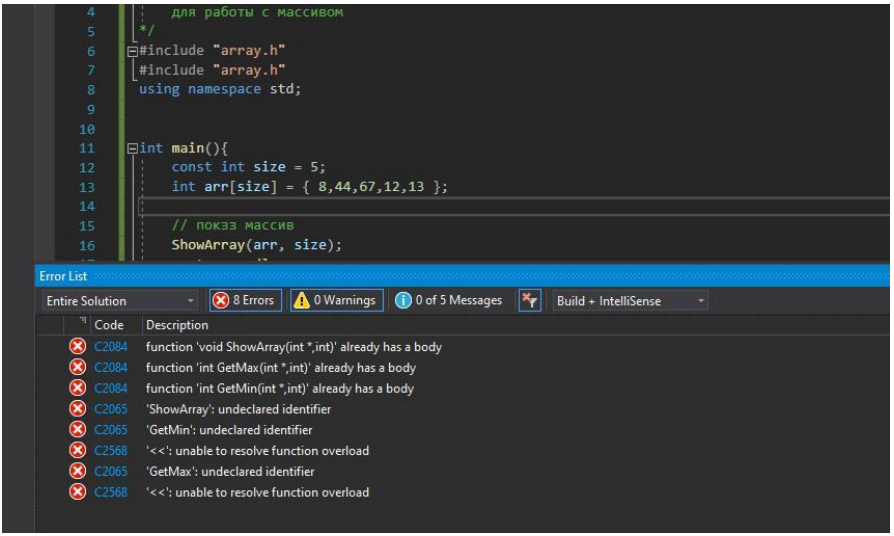
```
#pragma once
#include<iostream>
using namespace std;
```

```
void ShowArray(int* arr, int size) {
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
}

int GetMax(int* arr, int size) {
    int temp = arr[0];
    for (int i = 1; i < size; i++) {
        if (temp < arr[i])
            temp = arr[i];
    }

    return temp;
}

int GetMin(int* arr, int size) {
    int temp = arr[0];
    for (int i = 1; i < size; i++) {
        if (temp > arr[i])
            temp = arr[i];
    }

    return temp;

}
```

İkinci fayl ilə işləmək üçün, biz onu **include** ilə daxil etdik. Diqqət yetirin ki, başlıq faylımız **pragma** direktivinin istifadəsi ilə başlayır. **#pragma once** formatı, **array.h** faylının yenidən daxil edilməsinin qarşısını alır.

Bu direktiv qeyd edilmədikdə, faylın təkrar daxil edilməsi xətaya səbəb olacaq. Məsələn, belə

```
/*
  massiv ilə işləmək üçün
  funksiyaların olduğu fayl
*/
#include "array.h"
#include "array.h"
```



Şəkil 3

Bu xəta çox fayllı layihəniz olduqda və başlıq faylını müxtəlif yerlərdə bir neçə dəfə daxil etdikdə baş verə bilər.

Buna görə də, başlıq faylınızın həmişə **#pragma once** ilə başladığından əmin olun.

6. İmtahan tapşırıqları

1. Klaviaturadan daxil edilmiş mətni filtrdən keçirən proqram yaradın. Proqramın vəzifəsi verilmiş simvollar toplusunu boşluqlarla əvəz etməklə mətni oxumaq və onu ekranda göstərməkdir. Proqram, filtrləmə üçün aşağıdakı simvollar toplusunu təklif etməlidir:
   - Latın əlifbası simvolları;
   - Kiril simvolları;
   - durğu işarələri;
   - Rəqəmlər.

   Filtrlər ardıcıl olaraq tətbiq oluna bilər. Mövcud filtr yenidən tətbiq edildikdə, bu filtr ləğv olunmalıdır.
2. İnsan vs kompüter oyunu üçün "Battleship game" yazın. İnsan üçün gəmilərin avtomatik (gəmilər kompüter tərəfindən təsadüfi yerləşdirilir) və manual olaraq yerləşdirilməsi imkanını təmin edin. Atış zamanı kompüter bir məntiqə malikdirsə (yəni təsadüfi atəş etmirsə) tapşırığın dəyəri əhəmiyyətli dərəcədə artır.
3. Həqiqi ədədləri, həmçinin mötərizələri və növbəti əməliyyatları: +, -, *, /, ^ (qüvvətə yüksəltmə) daxil edə bilən arifmetik ifadənin qiymətinin hesablanması

üçün proqram yaradın. Hesablamalar, istifadə olunan əməliyyatların mötərizələri və prioritetləri nəzərə alınmaqla aparılmalıdır. Mümkün səhvlərin düzgün idarə olunmasını və onlar haqqında istifadəçinin məlumatlandırılmasını təmin edin.

STEP IT Academy, www.itstep.org

All rights to protected pictures, audio, and video belong to their authors or legal owners.
Fragments of works are used exclusively in illustration purposes to the extent justified by the purpose as part of an educational process and for educational purposes in accordance with Articles 21 and 23 of the Law of Ukraine “On Copyright and Related Rights”. The extent and method of cited works are in conformity with the standards, do not conflict with a normal exploitation of the work, and do not prejudice the legitimate interests of the authors and rightholders. Cited fragments of works can be replaced with alternative, non-protected analogs, and as such correspond the criteria of fair use.

All rights reserved. Any reproduction, in whole or in part, is prohibited. Agreement of the use of works and their fragments is carried out with the authors and other right owners. Materials from this document can be used only with resource link.

Liability for unauthorized copying and commercial use of materials is defined according to the current legislation of Ukraine.