ХИМИКОТЕХНОЛОГИЧЕН И МЕТАЛУРГИЧЕН УНИВЕРСИТЕТ – СОФИЯ

ИНФОРМАТИКА

част втора

Основи на програмирането на C++

лектор: гл. ас. д-р Стефан М. Панов

Катедра “Информатика”

Лекция 6 (14)

C++ предпроцесор

Клас памет на променливите

Предпроцесорът, (наричан често и препроцесор) както показва и самото име, преобразува сорс-кода преди неговата компилация до обектен код. Казано иначе, той подготвя началния код до състояние, когато компилацията може да започне. **Предпроцесорните команди се наричат директиви.** Задават се от програмиста в кода на програмата **и се разпознават по това, че винаги започват със символа #**.

#define	**#error**	**#include**
#if	**#else**	**#elif**
#endif	**#ifdef**	**#ifndef**
#undef	**#line**	**#pragma**

Таблица 14.1 – директивите на C++ предпроцесора

Заб. С развитието на езика C++ някои от директивите станаха излишни, но ние трябва да познаваме и тях, защото една част от програмистите продължават да ги ползват (въпрос на навици -;) в техните програми.

Директива #define

Синтаксисът ѝ е: **#define име_на_макрос заместващ_текст**

Между името и текста трябва да има поне един интервал. Името на макроса е специален идентификатор, защото **предпроцесорът ще замени всяко срещане на име_на_макрос със заместващ_текст**. Действието се нарича **макроразширение**. **заместващ_текст** може да бъде произволен текст, стига след замяната в кода да се получава синтактически правилна конструкция. При задаване на **име_на_макрос** са в сила същите правила както при задаване на име на променлива. **Общоприето е** имената на макросите да се изписват само с главни букви. Това ги отличава от имена на променливи. Примерно след изпълнението на следващия **cout** ще видим:**1 2**

```
#define NALIAVO 1 // ! забележете, няма точка и запетая ;
#define NADIASNO 2 // но може да има коментари
// .....................
cout << NALIAVO << ' ' << NADIASNO;
```

След дефиницията на един макрос той може да бъде използван като част от определението на други макроси. Например, следният код също определя NADIASNO и би довел с горния **cout** до същият резултат:

```
#define NALIAVO 1
#define NADIASNO NALIAVO+1
```

Навсякъде, където използваме NADIASNO предпроцесорът ще го замени със **1+1**, а при изпълнение на горния **cout** изразът **1+1** ще се изчисли и стойността ще се изведе на екран.

Заб. **Никаква** замяна няма да се извърши, ако името на макроса се намира във символен низ. Например при изпълнение на следната инструкция:

```
cout << "NADIASNO – tova dali e ime na makros?\n";
```

на екрана ще се изведе NADIASNO – tova dali e ime na makros?

а не:2 – tova dali e ime na makros?

Когато **заместващ_текст** не се събира на един ред можем да го продължим на няколко реда, като поставим обратна наклонена черта **** в края на всеки ред без последния. Пример:

```
#define DYLYG_NIZ "Tova e edna dosta dylga \
posledovatelnost ot simvoli, koiato shte izpolzvame za da \
onagledim kazanoto."
```

Прието е всички директиви #define да ги разположим в началото на файла или да включим в отделен файл, за да не ги търсим после по цялата програма. (виж по-надолу директивата #include).

Не е трудно да се забележи, че и с #define, и посредством спецификатора const можем да дефинираме константи, но вторият начин не може да се използва в чистото C. Примерно размерът на един масив може да зададем или с **#define SIZE 100** или чрез **const int SIZE = 100;**

Макроопределения, действащи като функции

Директивата **#define има и втора форма** при която **име_на_макрос** може да се използва с аргументи. Такива макроопределения действат подобно на функции и понякога се наричат макро-функции. Синтаксисът на втората форма е: **#define име_на_макрос(списък-параметри) заместващ_текст**

където **(списък-параметри)** се състои от имена на параметри, разделени със запетая. При извикване на макроса в кода, в скобите след името се подават аргументи. **Всяко срещане на параметър в заместващия текст се заменя със съответния аргумент**. Аргументите на макроса, както и при истинските функции, могат да бъдат и изрази. Да разгледаме като пример програма_68. Ако изпълним програмата ще видим, че стойността на променливата е 18, а не очакваното 30. Това е така, защото предпроцесорът заменя макроса с израза (3+3*5), което е равно на 18. Решение на проблема е всички параметри в заместващия текст да се ограждат с малки скоби,

```
using namespace std; // nashata 68 programa
#include <iostream> // primer za makrosi
#define NALIAVO 1 // ! zabelejete, niama tochka i zapetaia ;
#define NADIASNO NALIAVO+1
#define DYLYG_NIZ "Tova e edna dosta dylga \
posledovatelnost ot simvoli, koiato shte izpolzvame da \
onagledim kazanoto."
#define MULTI(a,b) (a*b)  // zabelejete razlikata v rezultata
//#define MULTI(a,b) ((a)*(b)) // pri dvata makrosa
int main( ){
      cout << NALIAVO << ' ' << NADIASNO<<endl<<DYLYG_NIZ<<endl;
// izvikvane na makro-funkciata MULTI s argumenti 3+3 i 5
      int pr = MULTI(3+3,5); // makrosyt MULTI(3+3,5) se zamenia s izraza (3+3*5)
      cout << "product = "<< pr <<endl; // ili s izraza ((3+3)*(5))
      return 0;
}
```

включително и целия заместващ текст. Вторият вариант на макроса ще се замести с израза ((3+3)*(5)) което е равно на 30.

Използването на макроопределения вместо истински функции има едно съществено достойнство: тъй като **заместващ_текст** на макроса се вмъква директно в кода на програмата няма никакви разходи на системни ресурси, както е при извикването на функции. Оттук и скоростта на работа на програмата ще бъде по-висока в сравнение с употребата на обикновени функции. Все пак повишената скорост е за сметка на увеличението на размера на програмата поради дублиране на кода на макро-функцията при всяко нейно използване.

Заб. Вместо макро-функциите в C++ могат да се използват т.н. **inline-функции** (ще бъдат разгледани по-късно), които вършат същата работа, но по-безопасно, защото не изискват допълнителни малки скоби. Но в чистото C inline-функции не се използват, затова там се прилагат само макро-функции.

Директива #error

Използва се рядко, обикновено при дебъгване. Директивата #error дава указание на компилатора да преустанови компилацията след извеждане на съобщението, за да може да се отстрани грешката. Синтаксисът ѝ е следният:

#error съобщение

Елементът съобщение не е затворен в двойни кавички. Освен съобщението, в някои работни среди се извежда и друга информация. Всеки може да провери дали е налична такава информация и евентуално каква е тя с експеримент.

Директива #include

Директивата на предпроцесора #include задължава компилаторът да включи или стандартен заглавен файл, или друг файл (заглавен или сорс), името на който се посочва. Имената на стандартните заглавни файлове се затварят в ъглови скобки, както е направено в разгледаните досега програми. Например

```
#include <cstring>
```

включва стандартния заглавен (хедър) файл за работа с низове.

При включване на друг файл името му може да се зададе или в двойни кавички или в ъглови скобки. Например, и двете следващи директиви задължават C++ компилатора да прочете и компилира файл с име prog69.h

```
#include <prog69.h>
```

```
#include "prog69.h"
```

Ако името на файла е затворено в ъглови скобки, то търсенето на файла ще се извърши в специалните папки, определени от конкретната реализация. Ако обаче името на файла е затворено в кавички, търсенето на файла започва в текущата папка (която най-често е директорията на проекта). Само ако файлът не е намерен, търсенето продължава в специалните папки, все едно името на файла е било зададено в ъглови скобки.

Самите вложени файлове също могат да съдържат директивата #include за да включат в себе си други вложени файлове.

Директива #undef

Премахва определение на макрос, ако то съществува. Иначе директивата се пренебрегва. След премахване може да се предефинира макрос със същото име. С директивата се постига локалност на действието на макросите, ако има такава необходимост. Пример:

```
#undef MULTI
```

Директиви за условна компилация

Позволяват избирателно да се компилират части от сорс-кода. Именно такава компилация се нарича **„условна“**. Използва се за разработка на различни версии на една и съща програма. Примерно за код, който зависи от операционната система или от архитектурата на компютъра.

Директиви #if, #else, #elif и #endif

Основният синтаксис е:

#if управляващ-израз
последователност от инструкции
#else
последователност от инструкции
#endif

Действие: Изчислява се управляващ-израз. Ако резултатът е ненулева стойност (ИСТИНА), инструкциите, заключени между #if и #else, се подават на компилатора. Инструкциите, заключени между #else и #endif, не се

компилират. И обратно: ако резултатът е нула тогава инструкците, заключени между #if и #else, не се компилират, а към компилатора се подават инструкциите, заключени между #else и #endif. Директивата #else не е задължителна и може да се пропусне. Следователно, следната конструкция е напълно валидна:

```
#if управляващ-израз
последователност от инструкции
#endif
```

Изразът, който стои след директивата **#if**, се изчислява по време на компилация. Следователно, той трябва да съдържа или константи, или идентификатори, които са били предварително определени. **Използването на променливи тук е изключено**.

Директивата #elif е еквивалентна на връзката инструкции else-if и се използва за формиране на многостепенна схема if-else-if, позволяваща

няколко варианта на компилация. (Напомня на if-else-if стълбицата от лекция 5.)

След #elif трябва да има константен израз. Ако изразът е ИСТИНА (има ненулева стойност), следващата последователност от инструкции се компилира и никакви други #elif-изрази не се проверяват или компилират. В противен случай се проверява аналогично следващият по ред #elif-израз. Общият формат на директивата #elif е следният:

```
# if израз
последователност от инструкции
#elif израз 1
последователност от инструкции
#elif израз 2
последователност от инструкции
// . . . . . .
#elif израз N
последователност от инструкции
#endif
```

Директиви #ifdef и #ifndef

Предлагат още два варианта на условна компилация. Синтаксис:

#ifdef име_на_макрос
последователност от инструкции
#else
последователност от инструкции
#endif

Ако име_на_макрос е определено с директивата #define, инструкциите, заключени между #ifdef и #else, се подават на компилатора. Инструкциите,

заключени между #else и #endif, не се компилират. И обратно: ако име_на_макрос не е определено с директивата #define или макросът е бил премахнат с #undef тогава инструкциите, заключени между #ifdef и #else, не се компилират, а към компилатора се подават инструкциите, заключени

между #else и #endif. Директивата #else не е задължителна и може да се пропусне. Следователно, следната конструкция е напълно валидна:

```
#ifdef име_на_макрос
последователност от инструкции
#endif
```

Директивата #ifndef се ползва вместо #ifdef със същия синтаксис, но с противоположна логика, затова няма да се разглежда отделно. (#ifdef като смисъл е „ако е дефинирано“, докато #ifndef е „ако не е дефинирано“.)

И двете директиви позволяват употребата на #elif вместо #else.

Предпроцесорен оператор defined

Явява се алтернатива на директивата #ifdef. Тоест,

следните двойки конструкции са еквивалентни:

#if defined (име_на_макрос) ≈ #ifdef име_на_макрос

#if !defined (име_на_макрос) ≈ #ifndef име_на_макрос

Предимството на оператора defined пред директивата #ifdef е, че с негова помощ могат да се проверят няколко макро-имена наведнъж. За целта резултатите от няколко оператора defined могат да се комбинират с логическите операции **&&** и **||**. Пример:

```
#define MACRO1

#define MACRO2

#if defined (MACRO1) && defined (MACRO2)
```

Директива #pragma

Работата на директивата #pragma зависи от конкретната реализация компилатора. Тя позволява да се задават на компилатора различни инструкции, предвидени от създателя му. Общият формат е такъв:

#pragma име

Тук елементът име представлява желаната #pragma-инструкция. Ако указаното име не се разпознае от компилатора, директивата #pragma просто го игнорира без да се издаде съобщения за грешка.

Най-простата #pragma-инструкция е Pragma-съобщението. То има следния синтаксис: #pragma message ("Текст на съобщението")

Следващата програма_70 е съставена от 2 файла, като всички директиви на предпроцесора се в отделен файл prog70.h. За да се видят тънкостите при употребата на отделните директиви програмата трябва да се пусне няколко пъти като се следват препоръките в коментарите. Следва файлът prog70.h

```
#define NALIAVO 1 // ! zabelejete, niama tochka i zapetaia ;
#define NADIASNO NALIAVO+1 //zamenete + s - za da sraboti #error
#define DYLYG_NIZ "Tova e edna dosta dylga \
posledovatelnost ot simvoli, koiato shte izpolzvame da \
onagledim kazanoto."
//#define MULTI(a,b) (a*b)  // zabelejete razlikata v rezultata
#define MULTI(a,b) ((a)*(b)) // pri uj dvata ednakvi makrosa
#if NADIASNO < NALIAVO
#error zabraneno e  NADIASNO < NALIAVO da e TRUE
#else
#pragma message ("Vinagi   spazvaite NADIASNO > NALIAVO da e TRUE!")
#endif
#define DEBUG 0 //probvaite s #define DEBUG 1  i #define DEBUG
```

Файлът prog70.h в включен в следващият prog70.cpp

```
using namespace std; // nashata 70 programa
#include <iostream>
#include "prog70.h" // sydyrja makrosi
int main( ){
	cout << NALIAVO << ' ' << NADIASNO<<endl<<DYLYG_NIZ<<endl;
// izvikvane na makro-funkciata MULTI s argumenti 3+3 i 5
	int pr = MULTI(3+3,5); // makrosyt MULTI(3+3,5) se zamenia s izraza (3+3*5)
	cout << "product = "<< pr<<endl; // ili s izraza ((3+3)*(5))
#ifdef DEBUG
// #if defined DEBUG  // moje vmesto #ifdef DEBUG
	cout << "Programata e v Debug rejim.\n";
#else
	cout << "Programata e v Normlalen rejim.\n";
#endif
// zabelejete razlikite mejdu #ifdef DEBUG i #if DEBUG
#if DEBUG // syshtoto kato #if DEBUG != 0
	cout << "PAK. Programata e v Debug rejim.\n";
#else
```

```
        cout << "Pak. Programata e v Normlalen rejim.\n";
#endif
        return 0;
}
```

Зарезервирани макроимена

В езика C++ са дефинирани шест вградени макроимена:

__LINE__ __FILE__ __DATE__ __TIME__ __STDC__ __cplusplus

(__ са две долни черти)

Макросите __LINE__ и __FILE__ съдържат номера на текущия ред и името на файла на компилираната програма.

Макросът __DATE__ е низ във формат месец/ден/година, който означава датата на транслация (компилация) на сорс-файла в обектен код.

Аналогично, макросът __TIME__ е низ във формат час/минути/секунди, който означава времето на транслация на сорс-файла в обектен код.

Точното назначение на макроса __STDC__ зависи от конкретната реализация на компилатора. Като правило, ако __STDC__ е определен, то компилаторът ще приема само стандартният C/C++ код, (който не съдържа никакви нестандартни разширения.)

Компилатор, съответстващ на ANSI/ISO-стандарта C++, определя макроса __cplusplus като стойност, съдържаща най-малко шест цифри. “Нестандартните” компилатори са длъжни да използват стойност, съдържаща пет или по-малко цифри.

Директива #line

Директивата #line се използва за изменение на стойностите на запазените макроимена __LINE__ и __FILE__

Нейният синтаксис е: #line номер_на_ред "име_на_файла"

Стойността на елемента номер_на_ред става номер на текущия ред в сорс-файла, а стойността на елемента име_на_файла — име на сорс-файла (от гледна точка на компилация и изпълнение на програмата.)

За да видите работата на директивата #line и стойностите на запазените макроимена добавете в края на prog70.cpp следните редове

```
#if NADIASNO > NALIAVO
// ! sledva vremeto kogato faila e kompiliran, a ne izpylnen
      cout << "Big error in " << __FILE__ << " at line " << __LINE__ << "\n";
      cout  << __DATE__ << "  " << __TIME__ << " "<< __cplusplus << "\n";
#endif
```

Изпълнете програмата, а след под реда #include "prog70.h" вмъкнете следния ред:

#line 104 "prog69.cpp"

Компилирайте, пуснете програмата още един път и отчетете разликите.

Заб. Директивата #line и #error се използват рядко, най-вече в специални случаи при дебъгване.

Спецификатори за клас-памет на променливите

C++ поддържа пет спецификатора за клас памет:

auto extern register static mutable

Те определят, как трябва да се съхранява една променлива, което включва:

къде променливата да бъде съхранена в паметта, областта на видимост и времето ѝ на живот.

Заб. Спецификаторът **mutable** се прилага само при работа с обекти на класовете (свързан е с ООП) и тук няма да бъде разглеждан.

Класът памет се задава в определението на променливата, т.е.

клас-памет тип име_на_променлива;

като е допустимо използването на само един спецификатор (за клас-памет).

Клас памет auto

Може да се прилага само към локални променливи (без параметри на функции). **Но тъй-като локалните променливи са клас auto по подразбиране, не е необходимо спецификаторът да се указва явно**.

Локалните променливи се съхраняват в област от паметта, наречена стек (stack). Когато при изпълнение програмата влезе в код с локална област на действие (каквато е тялото на функция и блок), компилаторът заделя памет за тях в стека. При излизане от функцията (или блока с код) **паметта се освобождава** и може да се използва от компилатора за заделяне на памет за други автоматични локални променливи.

Локалните auto-променливи не се инициализират автоматично (по подразбиране) от компилатора. При липса на явен инициализатор те съдържат произволни стойности.

Но ако има явна инициализация, то променливата се инициализира всеки път при влизане във функцията или блока с код. Инициализаторът на **auto**-променливите може да бъде всякакъв израз, чийто тип е съвместим с типа на променливата. Програма_71 илюстрира инициализацията:

```
using namespace std; // nashata 71-va programa
#include <iostream> // inicializacia na lokalna promenliva
void fun1(void);
int main(void) {
      fun1(); fun1(); fun1();
      return 0;
}
void fun1(void) { //  zamenete auto sys static i vijte razlikite pri pechat na a
      auto int a = 10; // syshtoto kato int a = 10;
      cout << "Vyv fun1:    a = "<< a << '\n';
      a++; // s auto promianata niama efekt, bezsmislena e
}
```

Паметта, заделена за a се освобождава при излизане от функцията и всяка съхранена в нея стойност се губи. Затова увеличението на a с 1 е безсмислено. При всяко влизане в fun1() се заделя памет за a в стека и a се инициализира с 10.

Клас памет register

Може да се прилага само към локални променливи и параметри на функции. Наличието на тоя спецификатор води до заявка за компилатора да задели памет за дефинираната променлива така, че достъпът до нея да бъде максимално бърз, т.е. във вътрешните регистри на процесора или кеша, вместо в RAM паметта. Ако компилаторът не може да удовлетвори заявката, променливата се съхранява като обикновена auto-променлива. Правилата за инициализация са същите като при auto променливите.

Заб. Ключовата дума **register** на практика може и да не се използва. Повечето съвременни компилатори са достатъчно интелигентни сами да преценят дали да оптимизират достъпа до дадена променлива или не. Има и компилатори, които игнорират ключовата дума **register.**

Следващата програма използва функцията clock(), чийто прототип:

clock_t clock ();

както и типът **clock_t** са определени в заглавния файл <ctime>. Функцията (опростено казано) връща броя на „цъканията“ (ticks) на часовника, изминали от момента на началото на изпълнение на програмата. Единицата време отговаряща на едно цъкане се определя от макроса **CLOCKS_PER_SEC, който задава броя цъкания за една секунда.** За нашият компилатор в devC++ средата е в сила **#define CLOCKS_PER_SEC 1000**

което означава, че едно цъкане е равно на една милисекунда. Типът **clock_t** е някой от основните аритметични типове, който е в състояние да представи броя на цъканията (примерно long int). Програма_72 измерва времето за изпълнение на два вложени цикъла, когато изпълняващата променлива е обикновена и регистърна.

```
using namespace std; // nashata 72-a programa za
#include <iostream> // izmervane na vremena pri izpolzvane na registerni i
#include <ctime>  // obiknoveni promenlivi za deistvia v cikyl.
void fun1(void);
unsigned int i; //obiknovena promenliva
int main(void) {
      register unsigned int j; // register-promenliva
      long start, end;
      start = clock();
      for(int delay=0; delay<50; delay++)
            for(i=0; i < 64000000; i++);
      end = clock();
      cout << "start=" << start <<"\tend=" << end <<"\n";
      cout << "Broj cykania za ne register-cikyl: "<< end-start << '\n';
      start = clock();
      for(int delay=0; delay<50; delay++) for(j=0; j < 64000000; j++) ;
      end = clock();
      cout << "Broj cykania za register-cikyl: " <<  end-start << '\n';
      return 0; }
```

Клас памет extern

Когато една програма е много голяма е неудобно да се състои от един файл, защото времето за компилация става дразнещо дълго. По-удобно е програмата да се раздели на няколко файла – тогава бъдещи изменения на един файл няма да налагат прекомпилация на цялата програма. Спецификаторът extern позволява реализацията на такъв подход. С негова помощ във всеки сорс файл могат да бъдат известни имената и типовете на глобални променливи, използвани от програмата като цяло и определени в друг файл.

Спецификаторът extern обявява променлива, но не заделя за нея памет.

Да се спрем на разликата между обявяване (деклариране) на променлива – на променливата се присвоява име и тип – и определяне (дефиниране) на променлива – за променливата се заделя памет. Обикновено обявяването се явява и нейното дефиниране. Но именно употребата на ключовата дума

extern обявява променливата без да я дефинира. Всяка променлива може да има няколко обявления, но само едно определение.

Следващата програма ще бъде реализирана като проект, в който ще влизат два сорс файла – prog73b.cpp и prog73a.cpp със следния код:

```
#include <iostream> // 73b programa, vtori file za specifikatora extern
extern  int y; // globalna obiava na vynshna globalna promenliva
void fun1(void) {
	extern int x; // lokalna obiava na vynshna globalna promenliva
	std::cout <<"in fun1:   x="<<x<<" \ty="<<y<<'\n';
	x= 200;
	y= 300;
}
void fun2(void) {
	y= 400;
//	x= 200; // shte dade greshka!
}

using namespace std; // nashata 73a programa, pyrvi file za
```

```
#include <iostream> // specifikatora extern
extern void fun1(void); // moje i extern void fun1(void);
void fun2(void); // no toia extern e za funkciata fun2
int x; //globalna promenliva
int main(void) {
	x= 10;
	extern int y;// zashtoto e definirana po-nadolu v syshtia file
	cout << "in main:   y=" << y << endl;// inicializirana e s 0;
	fun2();
	cout << "in main:   y=" << y << endl;
	y = 20;
	fun1();
	cout << "in main:   x=" << x << "\ty=" << y << endl;
	return 0;
}
int y; //globalna promenliva, definirana nakraia
```

Забележете, че **using namespace std;** може да бъде пропуснато, както е направено в prog73b.cpp, но в такъв случай трябва явно да се укаже чрез **std::cout**, че обектът **cout** принадлежи към пространството на имената **std**.

Когато една глобална променлива се използва преди нейното дефиниране, тя трябва да се обяви. Така е постъпено с променливата **y** във файла prog73a.cpp.

За да може prog73b.cpp да използва променливите **x** и **y**, определени в prog73a.cpp, е необходимо преди това те да бъдат обявени в prog73b.cpp. **Обявяването може да се направи глобално**, в началото на файла, както е направено с променливата **y**, **или локално в блока, в който тази променлива се използва**, както е направено с променливата **x**. Променливата **y** ще бъде достъпна в целия prog73b.cpp, докато **x** е достъпна във **fun1()**, но не и във **fun2()**. **Причината е, че обявяването на x не се вижда извън fun1()**.

Глобалните променливи се инициализират по подразбиране с 0 (напр. **y**).

За да използваме повече от един cорс-файл в една програма е необходимо да създадем проект. За нашата **dev-C++** среда това става така:

File-New-Project с мишката се избира **Console Application** след което се задава името на проекта, примерно **Proj_73** и накрая **OK**. В името на проекта може да се посочи и пътят към папката, която предварително сме създали, примерно: **D:\Stefan\HTMU\Study\Lekcii_srs\Lesson_14_preprocess\Proj_73**

След това може да премахнем създадения по подразбиране файл main.cpp, като една от възможностите е да извършим следните действия:

1. Избираме **Project** в малкия ляв прозорец. Само ако вляво от името на проекта (**Proj_73**) стои знака **+** щракаме върху него.

2. Избираме с мишката main.cpp, избор на десен бутон на мишката и от падащото меню се избира "**Remove File**".

Следваща стъпка е да добавим двата предварително създадени сорс файла prog73b.cpp и prog73a.cpp в папката на проекта **Lesson_14_preprocess**. Стъпките са:

1. Избираме **Project** в малкия ляв прозорец. Избор на десен бутон на мишката и от падащото меню се избира "**Add to Project**".

2. При натиснат **CTRL** клавиш се избират с мишката файловете prog73b.cpp и prog73a.cpp и накрая се щрака върху **Open**.

След това двата файла могат да се компилират наведнъж с **Execute-Rebuild All**

При промени на някой от файловете се компилира само той. (Някои от детайлите при компилация след промяна на сорс файл ще бъдат показани на упражнение).

Клас памет static

Клас памет **static** може да се прилага и към глобални, и към локални променливи (без параметри на функции). **Статичните променливи се съхраняват извън стека и съществуват през цялото време на изпълнение на програмата. Те се създават при пускане на програмата**.

Локални статични променливи

Когато спецификаторът **static** е приложен към определението за локална променлива, примерно: **static unsigned int age;**

то за нея се заделя памет извън стека по същия начин, както за глобална променлива. Именно по такъв начин, че паметта да се запазва при изпълнението на цялата програма. А това позволява статичната локална променлива да съхрани стойността си между извикванията на функцията, в която тя е определена (за разлика от обикновената локална променлива).

Ключовата разлика между статична локална променлива и глобална променлива се състои в това, че локалната е известна само на блока, в който тя е обявена. (Защото е локална!)

Статичната променлива също може да се инициализира, примерно: **static unsigned int age = 72;**

Но тая инициализация се прави само веднъж, в началото на изпълнение на програмата. Обикновените локални (auto) променливи се инициализират при всяко влизане във функцията, в която те са дефинирани.

Благодарение на статичните локални променливи (СЛП) може да се създават функции, които да съхраняват стойностите си между извикванията на функцията. Ако ги нямаше СЛП, щеше да се наложи употребата на глобални променливи, но това крие рискове от всевъзможни странични ефекти. Следващата програма_74 използва СЛП за да съхранява **текущата** средна стойност на целите числа, въведени от потребителя.

```
using namespace std; // nashata 74-ta programa za izchisliavane na
#include <iostream> // tekushtata sredna stoinost na celi chisla
float sredno(int i) ; // vavedeni ot potrebitelia
int main() { // vmesto -1 nai-dobre e da se izpolzwa cialo chislo, za koeto
      int num; // znaem, che niama da uchastva v izchislenieto
      do {
            cout << "Vavedi cialo chislo (-1 za izhod): ";  cin >> num;
            if(num != -1) cout<<"Tekushoto sredno = "<< sredno(num) <<'\n';
      } while(num != -1);
      return 0;
}
float sredno(int i){ // namira tekushtata sredna stoinost
      static int sum=0, count=0;
      sum += i;
      return (float)sum/(++count);
}
```

Друг пример за СЛП е програма_71 когато спецификаторът static се замени с auto. Забележете разликите в резултата при двата варианта!

Глобални статични променливи

Когато спецификаторът static е приложен към определението за глобална променлива, тя ще бъде видима само за файла, където е определена. Т.е. въпреки, че е глобална, функциите, дефинирани в другите файлове нито имат представа за тая променлива, нито ще могат да променят стойността ѝ. Като следствие можем да имаме статични глобални променливи (СГП) в отделните сорс-файлове с еднакви имена. **Но по-важното е, че за големи програми (с много файлове) една СГП е много по-добре защитена от несанкциониран достъп в сравнение с обикновената глобална променлива**.

Следва подобрен вариант със СГП на програмата за изчисляване на текуща средна стойност, реализиран като проект с два сорс файла, позволяващ нулиране на средното във всеки момент без повторно пускане на програмата.

```
using namespace std; // nashata 75-ta programa, pyrvi file prog75a.cpp,
#include <iostream> // nov  variant za izchisliavane na tekushtata sredna
float sredno(int i) ;// stoinost na celi chisla vavedeni ot potrebitelia
void reset(); // funkcia za nulirane na tekushtoto sredno
int main() { // vmesto -1 i -2 nai-dobre e da se izpolzwat celi chisla, za
    int num; // koito znaem, che niama da uchastvat v izchislenieto
    do {
        cout << "Vavedi cialo chislo (-1 za izhod) (-2 za nulirane): ";
        cin >> num;
        if(num==-2) { reset (); continue; }
        if(num != -1) cout<<"Tekushoto sredno = "<< sredno(num) <<'\n';
    } while(num != -1);
    return 0;
}
```

Във вторият файл на Програма_75 променливите са статични глобални, за да подсигурим, че ще бъдат променяни само от функциите във файла.

```
// vtori fail 75b kym programa za izchisliavane na sredna stoinost
static int sum=0, count=0; // s globalni statichni promenlivi
float sredno(int i){ // namira tekushtata sredna stoinost
      sum += i;
      return (float)sum/(++count);
}
void reset() {//nulira tekushtata sredna stoinost
      sum = 0;
      count = 0;
}
```

В заключение ще споменем, че освен със спецификатора **static** достъпът до глобални променливи **при големи проекти може да бъде осъществен и с използване на пространствата на имената**.