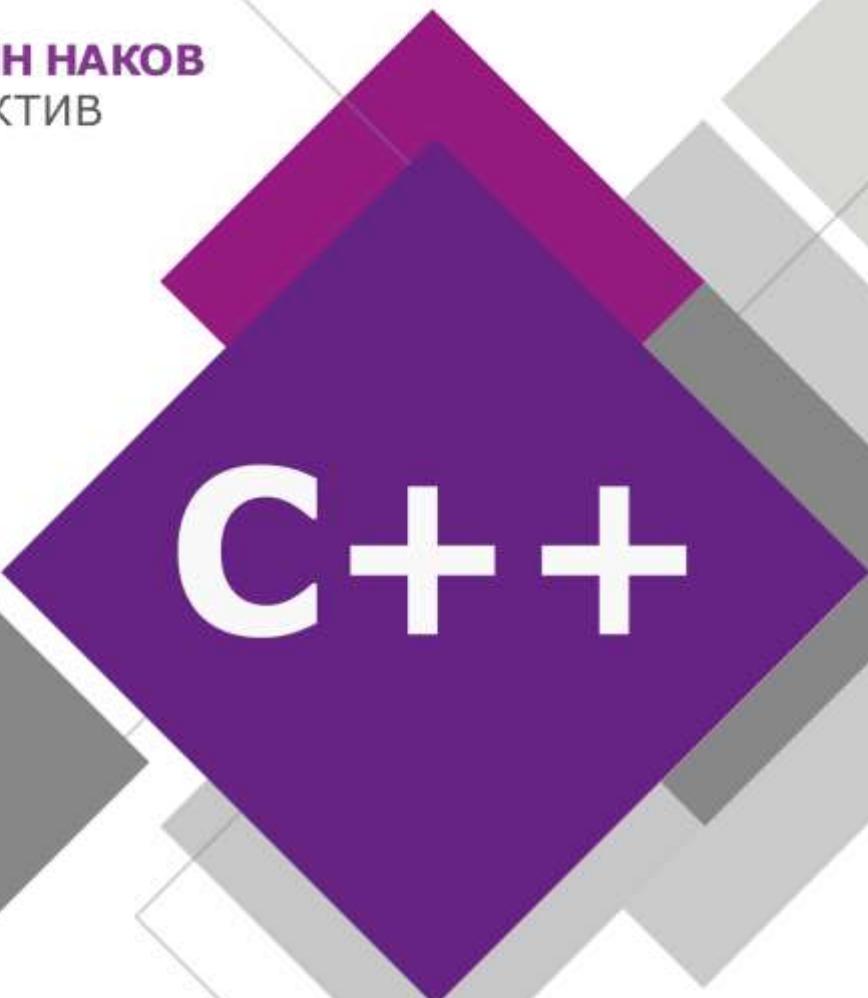


СВЕТЛИН НАКОВ
& КОЛЕКТИВ



C++

**ОСНОВИ НА
ПРОГРАМИРАНЕТО
СЪС **C++****



SoftUni
Foundation



Software
University

Кратко съдържание

Кратко съдържание	3
Основи на програмирането със C++	5
Съдържание	7
Предговор	13
Глава 1. Първи стъпки в програмирането.....	27
Глава 2.1. Прости пресмятания с числа.....	55
Глава 2.2. Прости пресмятания с числа – изпитни задачи.....	87
Глава 3.1. Прости проверки.....	105
Глава 3.2. Прости проверки – изпитни задачи.....	133
Глава 4.1. По-сложни проверки	147
Глава 4.2. По-сложни проверки – изпитни задачи.....	169
Глава 5.1. Повторения (цикли)	189
Глава 5.2. Повторения (цикли) – изпитни задачи	203
Глава 6.1. Вложени цикли.....	221
Глава 6.2. Вложени цикли – изпитни задачи	237
Глава 7.1. По-сложни цикли.....	257
Глава 7.2. По-сложни цикли – изпитни задачи	277
Глава 8.1. Подготовка за практически изпит – част I	289
Глава 8.2. Подготовка за практически изпит – част II	315
Глава 9.1. Задачи за шампиони – част I.....	331
Глава 9.2. Задачи за шампиони – част II.....	347
Глава 10. Функции.....	363
Глава 11. Хитрости и хакове	399
Заключение.....	413

Качествено образование,
професия и работа за

Софтуерни инженери

- ✓ Безплатен старт за начинаещи - присъствено и онлайн
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



Софтуни предоставя съвременно иновативно образование за програмиране; ИТ и дигитални умения на хиляди млади хора. Програмата "**Софтуерен университет**" изгражда истински **профессионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на Софтуни, за да усвояте **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

Софтуни работи пряко с **компаниите от софтуерната индустрия**, съдействайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ



1 - 1.5 години



1.5 - 2 години



Programming Basics

Кариерен Старт

Дипломиране

70/100 credits

80/100/150 credits

Кандидатствай

softuni.bg/apply

Основи на програмирането със C++

Светлин Наков и колектив

Андрей Царев	Зорница Йоханова
Божидар Колев	Игнес Симеонова
Борис Горанов	Йордан Добрев
Венцислав Петров	Любомир Господинов
Георги Лацев	Мирела Дамянова
Георги Шавов	Николай Костов
Денис Миланов	Пламен Динев
Димитър Кацарски	Светлин Наков
Димо Димов	Цветан Иванов
Димо Чанев	

ISBN: 978-619-00-0951-1

София, 2019

Основи на програмирането със C++

© Светлин Наков и колектив, 2019 г.

Първо издание, последна редакция: март 2019 г.

Настоящата книга се разпространява **свободно** под **отворен лиценз CC-BY-SA**, който определя следните права и задължения:

- **Споделяне** – можете да копирате и разпространявате книгата свободно във всякакви формати и медиии.
- **Адаптиране** – можете да копирате, миксирате и променяте части от книгата и да създавате нови материали на базата на извадки от нея.
- **Признание** – при използване на извадки от книгата трябва да цитирате оригиналния източник, настоящия лиценз и да описвате направените промени, без да въвеждате потребителя в заблуда, че оригиналните автори подкрепят вашата работа.
- **Некомерсиална употреба** – нямате право да използвате книгата за комерсиални цели.
- **Подобно споделяне** – ако създавате материали чрез миксиране, промяна и копиране на извадки от книгата, трябва да споделите резултата под същия или подобен лиценз.

Всички запазени марки, използвани в тази книга, са собственост на техните притежатели.

Издателство: Фабер, гр. Велико Търново

ISBN: 978-619-00-0951-1

Корица: Марина Шидерова – <https://shideroff.com>

Официален уеб сайт: <https://cpp-book.softuni.bg>

Официална Facebook страница: <https://fb.com/IntroProgrammingBooks>

Сурс код: <https://github.com/SoftUni/Programming-Basics-Book-CPP-BG>

Съдържание

Кратко съдържание	3
Основи на програмирането със C++	5
Съдържание	7
Предговор	13
За кого е тази книга?	13
Защо избрахме езика C++?	14
Книгата на други програмни езици: C#, Java, JavaScript, Python, C++, PHP	14
Програмиране се учи с много писане, не с четене!.....	15
За Софтуерния университет (Софтууни).....	15
Как се става програмист?	17
Книгата в помощ на учителите.....	23
Историята на тази книга.....	23
Официален сайт на книгата	24
Форум за вашите въпроси.....	25
Официална Facebook страница на книгата.....	25
Лиценз и разпространение.....	25
Докладване на грешки	26
Приятно четене!	26
Глава 1. Първи стъпки в програмирането.....	27
Видео	27
Какво означава "да програмираме"?	27
Как да напишем конзолна програма?	31
Среда за разработка (IDE)	31
Пример: създаване на конзолна програма "Hello C++"	36
Тествайте примерните програми.....	44
Типични грешки в C++ програмите	45
Какво научихме от тази глава?	48
Упражнения: първи стъпки в коденето	49
Конзолни, графични и уеб приложения	54
Глава 2.1. Прости пресмятания с числа.....	55
Видео	55

Системна конзола	55
Пресмятания в програмирането	56
Типове данни и променливи.....	56
Четене на числа от конзолата	57
Четене на дробно число от конзолата.....	58
Четене на текст	59
Печатане на текст и числа	60
Аритметични операции.....	61
Съединяване на текст и число	63
Числени изрази	64
Какво научихме от тази глава?.....	67
Упражнения: прости пресмятания	67
Глава 2.2. Прости пресмятания с числа – изпитни задачи.....	87
Четене на числа от конзолата	87
Извеждане на текст по шаблон (placeholder)	87
Аритметични оператори.....	88
Конкатенация.....	88
Изпитни задачи.....	89
Задача: учебна зала	89
Задача: зеленчукова борса.....	92
Задача: ремонт на плочки	95
Задача: парички.....	99
Задача: дневна печалба.....	102
Глава 3.1. Прости проверки.....	105
Видео	105
Сравняване на числа.....	105
Прости проверки	106
Проверки с if-else конструкция	107
За къдравите скоби { } след if / else	108
Живот на променлива	110
Серии от проверки.....	110
Упражнения: прости проверки	112
Дебъгване - прости операции с дебъгер	116

Упражнения: прости проверки	119
Глава 3.2. Прости проверки – изпитни задачи.....	133
Изпитни задачи	133
Задача: цена за транспорт	133
Задача: тръби в басейн	136
Задача: поспаливата котка Том.....	139
Задача: реколта	141
Задача: фирма	144
Глава 4.1. По-сложни проверки	147
Видео	147
Вложени проверки.....	147
По-сложни проверки.....	150
Логическо "И"	151
Логическо "ИЛИ"	153
Логическо отрицание	155
Операторът скоби ()	155
По-сложни логически условия	155
Условна конструкция switch-case	160
Какво научихме от тази глава?	162
Упражнения: по-сложни проверки	163
Глава 4.2. По-сложни проверки – изпитни задачи.....	169
Вложени проверки.....	169
Switch-case проверки	169
Изпитни задачи	170
Задача: навреме за изпит	170
Задача: пътешествие.....	174
Задача: операции между числа.....	178
Задача: билети за мач	181
Задача: хотелска стая.....	185
Глава 5.1. Повторения (цикли).....	189
Видео	189
Повторения на блокове код (for цикъл).....	189
Code Snippet за for цикъл във Visual Studio	190

Какво научихме от тази глава?	199
Упражнения: повторения (цикли)	199
Глава 5.2. Повторения (цикли) – изпитни задачи	203
Изпитни задачи	203
Задача: хистограма	203
Задача: умната Лили	208
Задача: завръщане в миналото	211
Задача: болница	213
Задача: деление без остатък	216
Задача: логистика	218
Глава 6.1. Вложени цикли	221
Видео	221
Вложени цикли	222
Чертане на по-сложни фигури	228
Какво научихме от тази глава?	236
Глава 6.2. Вложени цикли – изпитни задачи	237
Изпитни задачи	237
Задача: чертане на крепост	237
Задача: пеперуда	241
Задача: знак "Стоп"	244
Задача: стрелка	247
Задача: брадва	251
Глава 7.1. По-сложни цикли	257
Видео	257
Цикли със стъпка	257
While цикъл	260
Най-голям общ делител (НОД)	261
Алгоритъм на Евклид	262
Do-while цикъл	263
Безкрайни цикли и операторът break	265
Вложени цикли и операторът break	269
Задачи с цикли	270
Какво научихме от тази глава?	275

Глава 7.2. По-сложни цикли – изпитни задачи	277
Изпитни задачи	277
Задача: генератор за тъпи пароли	277
Задача: магически числа	279
Задача: спиращо число	282
Задача: специални числа	284
Задача: цифри	286
Глава 8.1. Подготовка за практически изпит – част I	289
Видео	289
Практически изпит по "Основи на програмирането"	289
Система за онлайн оценяване (Judge)	289
Задачи с прости пресмятания	289
Задачи с единична проверка	293
Задачи с по-сложни проверки	297
Задачи с единичен цикъл	301
Задачи за чертане на фигурки на конзолата	306
Задачи с вложени цикли с по-сложна логика	310
Глава 8.2. Подготовка за практически изпит – част II	315
Изпитни задачи	315
Задача: разстояние	315
Задача: смяна на плочки	318
Задача: магазин за цветя	321
Задача: оценки	324
Задача: коледна шапка	326
Задача: комбинации от букви	328
Глава 9.1. Задачи за шампиони – част I	331
По-сложни задачи върху изучавания материал	331
Задача: пресичащи се редици	331
Задача: магически дати	336
Задача: пет специални букви	341
Глава 9.2. Задачи за шампиони – част II	347
По-сложни задачи върху изучавания материал	347
Задача: дни за страстно пазаруване	347

Задача: числен израз	352
Задача: бикове и крави	357
Глава 10. Функции.....	363
Какво е "функция"?	363
Функции с параметри	371
Връщане на резултат от функция	376
Варианти на функции.....	382
Вложени функции (локални функции).....	387
Именуване на функции. Добри практики при работа с функции	387
Какво научихме от тази глава?	390
Упражнения	390
Глава 11. Хитрости и хакове	399
Форматиране на кода	399
Именуване на елементите на кода.....	401
Бързи клавиши във Visual Studio.....	402
Шаблони с код (code snippets).....	403
Техники за дебъгване на кода	405
Справочник с хитрости	408
Заключение.....	413
Тази книга е само първа стъпка!	413
Накъде да продължим след тази книга?	414
Онлайн общности за стартиращите в програмирането	417
Успех на всички!	418

Предговор

Книгата "Основи на програмирането" е официален учебник за курса "Programming Basics" за начинаещи в Софтуерния университет (Софтуни):

<https://softuni.bg/courses/programming-basics>

Тя запознава читателите с писането на **програмен код** на начално ниво (basic coding skills), работа със **среда за разработка** (IDE), използване на **променливи** и **данни**, **оператори и изрази**, работа с **конзолата** (четене на входни данни и печатане на резултати), използване на **условни конструкции** (**if, if-else, switch-case**), **цикли** (**for, while, do-while**) и работа с **функции** (деклариране и извикване на функции, подаване на параметри и връщане на стойност). Използват се езикът за програмиране **C++** и средата за разработка **Visual Studio**. Обхванатият учебен материал дава базова подготовка за по-задълбочено изучаване на програмирането и подготвя читателите за приемния изпит в Софтуни.



Тази книга ви дава само **първите стъпки към програмирането**. Тя обхваща съвсем начални умения, които предстои да развивате години наред, докато достигнете до ниво, достатъчно за започване на работа като програмист.

Книгата се използва и като неофициален учебник за училищните курсове по програмиране в професионалните гимназии, изучаващи професиите "Програмист", "Приложен програмист" и "Системен програмист", както и като допълнително учебно пособие в началните курсове по програмиране в **средните училища, профилираните и математическите гимназии**, за паралелките с профил "информатика и информационни технологии".

За кого е тази книга?

Тази книга е подходяща за **напълно начинаещи в програмирането**, които искат да опитат какво е да програмираш и да научат основните конструкции за създаване на програмен код, които се използват в софтуерната разработка, независимо от езиците за програмиране и използваните технологии. Книгата дава една **солидна основа** от практически умения, които се използват за по-нататъшно обучение в програмирането и разработката на софтуер.

За всички, които не са преминали бесплатния курс по основи на програмирането за напълно начинаещи в Софтуни, специално препоръчваме да го запишат **напълно бесплатно**, защото програмиране се учи с правене, не с четене! На курса ще получите бесплатно достъп до учебни занятия, обяснения и демонстрации на живо или онлайн (като видео уроци), **много практика и писане на код**, помощ при решаване на задачите след всяка тема, достъп до преподаватели, асистенти и ментори, както и форум и дискусионни групи за въпроси, достъп до общност от хиляди навлизящи в програмирането и всякаква друга помощ за начинаещия.

Бесплатният курс в Софтуни за напълно начинаещи е подходящ за **ученици** (от 5

клас нагоре), **студенти и работещи** други професии, които искат да натрупат технически знания и да разберат дали им харесва да програмират и дали биха се занимавали сериозно с разработка на софтуер за напред.

Нова група започва всеки месец. Курсът "Programming Basics" в СофтУни се организира регулярно с няколко различни езика за програмиране, така че опитайте. Обучението е **безплатно** и може да се откажете по всяко време, ако не ви допадне. **Записването** за бесплатно присъствено или онлайн обучение за стаптиращи в програмирането е достъпно през **формата за кандидатстване** в СофтУни: <https://softuni.bg/apply>.

Защо избрахме езика C++?

За настоящата книга избрахме езика **C++**, макар и да не е най-подходящият за начинаещи, защото е **мощен и ефективен** език за програмиране от високо ниво с отворен код и се използва масово, когато се търси ефективност. В много школи стартът в програмирането започва от C++ (вместо от някой по-лесен език), за да се усвои работата на ниско ниво (по-близо до хардуера), управлението на паметта и писането на ефективен код. Към 2019 г. езикът C++ е **основният**, на който се пише по олимпиади и състезания по програмиране, включително и на международната олимпиада по информатика за ученици ([IOI](#)).

Езикът C++ е **широкоразпространен** и поддържан на всички платформи, с добре развита екосистема и съответно дава силна основа за развитие в програмирането и софтуерните технологии. C++ е статично типизиран, компилируем език, който комбинира парадигмите на процедурното, обектно-ориентираното и generic програмирането.

В книгата ще използваме **езика C++** и средата за разработка **Visual Studio**, които са достъпни бесплатно от Microsoft.

Както ще обясним по-късно, **езикът за програмиране, с който стаптираме, няма съществено значение**, но все пак трябва да ползваме някакъв програмен език, и в тази книга сме избрали именно C++. Книгата може да се намери преведена огледално и на други езици за програмиране като C#, Java, Python и JavaScript (вж. <https://csharp-book.softuni.bg>).

Книгата на други програмни езици: C#, Java, JavaScript, Python, C++, PHP

Настоящата книга по програмиране за напълно начинаещи е достъпна на няколко езика за програмиране (или е в процес на адаптация за тях):

- [Основи на програмирането със C#](#)
- [Основи на програмирането с Java](#)
- [Основи на програмирането с JavaScript](#)

- [Основи на програмирането с Python](#)
- [Основи на програмирането със C++](#)
- [Основи на програмирането с PHP](#)

Ако предпочитате друг език, изберете си от списъка по-горе.

Програмиране се учи с много писане, не с четене!

Ако някой си мисли, че ще прочете една книга и ще се научи да програмира без да пише код и да решава здраво задачи, определено е в заблуда. Програмирането се учи с **много, много практика**, с писане на код всеки ден и решаване на стотици, дори хиляди задачи, сериозно и с постоянство, в продължение на години.

Трябва **да решавате здраво задачи**, да бъркате, да се поправяте, да търсите решения и информация в Интернет, да пробвате, да експериментирате, да намирате по-добри решения, да свиквате с кода, със синтаксиса, с езика за програмиране, със средата за разработка, с търсенето на грешки и дебъгването на неработещ код, с разсъжденията над задачите, с алгоритмичното мислене, с разбиването на проблемите на стъпки и имплементацията на всяка стъпка, да трупате опит и да вдигате уменията си всеки ден, защото да се научиш да пишеш код е само **първата стъпка към професията "софтуерен инженер"**. Имате да учите много, наистина много!

Съветваме читателя като минимум **да пробва всички примери от книгата**, да си поиграе с тях, да ги променя и тества. Още по-важни от примерите, обаче, са **задачите за упражнения**, защото те развиват практическите умения на програмиста.

Решавайте всички задачи от книгата, защото програмиране се учи с практика! Задачите след всяка тема са внимателно подбрани, така че да покриват в дълбочина обхванатия учебен материал, а целта на решаването на всички задачи от всички обхванати теми е да дадат **цялостни умения за писане на програмен код** на начално ниво (каквато е целта и на тази книга). На курсовете в СофтУни не случайно **наблягаме на практиката** и решаването на задачи, и в повечето курсове писането на код в клас е над 70% от целия курс.



Решавайте всички задачи за упражнения от книгата. Иначе нищо няма да научите! Програмиране се учи с писане на много код и решаване на хиляди задачи!

За Софтуерния университет (Софтуни)

[Софтуерният университет \(Софтуни\)](#) е **най-мащабният** учебен център за софтуерни инженери в България. През него преминават десетки хиляди студенти всяка година. СофтУни отваря врати през 2014 г. като продължение на усилията на [д-р Светлин Наков](#) масирано да изгражда **кадърни софтуерни специалисти** чрез

истинско, съвременно и качествено образование, което комбинира фундаментални знания със съвременни софтуерни технологии и много практика.

Софтуерният университет предоставя **качествено образование, професия, работа и възможност за придобиване на бакалавърска степен** за програмисти, софтуерни инженери и ИТ специалисти. СофтУни изгражда изключително успешно трайна връзка между **образование и индустрия**, като си сътрудничи със стотици софтуерни фирми, осигурява работа и стажове на своите студенти, предоставя качествени специалисти за софтуерната индустрия и директно отговаря на нуждите на работодателите чрез учебния процес.

Безплатните курсове по програмиране в СофтУни

Софтуерни организира **безплатни курсове по програмиране** за напълно начинаещи в цяла България - присъствено и онлайн. Целта е **всеки, който има интерес към програмиране и технологии, да опита програмирането** и да се увери сам дали то е интересно за него и дали иска да се занимава сериозно с разработка на софтуер. Можете да се запишете за **безплатния курс по основи на програмирането** от страницата за кандидатстване в СофтУни: <https://softuni.bg/apply>.

Безплатните курсове по основи на програмирането в СофтУни имат за цел да ви запознят с **основните програмни конструкции** от света на софтуерната разработка, които ще можете да приложите при всеки един език за програмиране. Те включват работа с **данни, променливи и изрази**, използване на **проверки**, конструиране на **цикли** и дефиниране и извикване на **методи** и други похвати за изграждане на програмна логика. Обученията са **изключително практически насочени**, което означава, че **силно се набляга на упражнения**, а вие получавате възможността да приложите знанията си още докато ги усвоявате.

Настоящият **учебник по програмиране** съпътства безплатните курсове по програмиране за начинаещи в СофтУни и служи като допълнително учебно помагало, в помощ на учебния процес.

Judge системата за проверка на задачите

Софтуери Judge системата (<https://judge.softuni.bg>) представлява автоматизирана система в Интернет за проверка на решения на задачи по програмиране чрез **поредица от тестове**. Предаването и проверката на задачите се извършва в **реално време**: пращате решение и след секунди получавате отговор дали е вярно. Всеки **успешно** преминат тест дава предвидените за него точки. При вярно решение получавате всички точки за задачата. При частично вярно решение получавате част от точките за дадената задача. При напълно грешно решение, получавате 0 точки.

Всички задачи от настоящата книга са достъпни за тестване в СофтУни Judge и силно препоръчваме да ги тествате след като ги решите, за да знаете дали не изпускате нещо и дали наистина решението ви работи правилно, според изискванията на задачата.

Имайте предвид и някои особености на SoftUni Judge системата:

- За всяка задача Judge системата пази най-високия постигнат резултат. Ако качите решение с грешен код или по-слаб резултат от предишното ви изпратено, системата няма да ви отнеме точки.
- Изходните резултати на вашата програма се сравняват от системата стриктно с очаквания резултат. Всеки излишен символ, липсваща запетайка или интервал може доведе до 0 точки на съответния тест. **Изходът**, който Judge системата очаква, е описан в условието на всяка задача и към него не трябва да се добавя нищо повече.

Пример: ако в изхода се изисква да се отпечата число (напр. **25**), не извеждайте описателни съобщения като **The result is: 25**, а отпечатайте точно каквото се изисква, т.е. само числото.

Софтуни Judge системата е достъпна по всяко време от нейния сайт: <https://judge.softuni.bg>.

- За вход използвайте автентификацията си от сайта на Софтуни: <https://softuni.bg>.
- Използването на системата е **бесплатно** и не е обвързано с участието в курсовете на Софтуни.

Убедени сме, че след няколко изпратени задачи, ще ви хареса да получавате моментална обратна връзка дали написаното от вас решение е вярно, и Judge системата ще ви стане най-любимия помощник при учене на програмирането.

Как се става програмист?

Драги читатели, сигурно много от вас имат амбицията да стават програмисти, да си изкарват прехраната с разработка на софтуер или да работят в ИТ сектора. Затова сме приготвили за вас **кратко ръководство "Как се става програмист"**, за да ви ориентираме за стъпките към тази така желана професия.

Програмист (на ниво започване на работа в софтуерна фирма) се става за **най-малко 1-2 години здраво учене и писане на код всеки ден**, решаване на няколко хиляди задачи по програмиране, разработка на няколко по-серииозни практически проекта и трупане на много опит с писането на код и разработката на софтуер. Не става за един месец, нито за два! Професията на софтуерния инженер изисква голям обем познания, покрити с много, много практика.

Има **4 основни групи умения**, които всички програмисти трябва да притежават. Повечето от тези умения са устойчиви във времето и не се влияят съществено от развитието на конкретните технологии (които се променят постоянно). Това са уменията, които **всеки добър програмист притежава** и към които всеки новобранец трябва да се стреми:

- писане на код (20%)

- алгоритмично мислене (30%)
- фундаментални знания за професията (25%)
- езици и технологии за разработка (25%)

Умение #1 – кодене (20%)

Да се научите **да пишете код** формира около 20% от минималните умения на програмиста, необходими за започване на работа в софтуерна фирма. Умението да кодиш включва следните компоненти:

- работа с променливи, проверки, цикли
- ползване на функции, методи, класове и обекти
- работа с данни: масиви, списъци, хеш-таблици, стрингове

Умението да кодиш **може да се усвои за няколко месеца** усилено учене и здраво решаване на практически задачи с писане на код всеки ден. Настоящата книга покрива само първата точка от умението да кодиш: **работка с променливи, проверки и цикли**. Останалото остава да се научи в последващи обучения, курсове и книги.

Книгата (и курсовете, базирани на нея) дават само началото от едно дълго и сериозно учене, по пътя на професионалното програмиране. Ако не усвоите до съвършенство учебния материал от настоящата книга, няма как да станете програмист. Ще ви липсват фундаментални основи и ще ви става все по-трудно напред. Затова **отделете достатъчно внимание на основите на програмирането**: решавайте здраво задачи и пишете много код месеци наред, докато се научите **да решавате с лекота всички задачи от тази книга**. Тогава продължете напред.

Специално обръщаме внимание, че **езикът за програмиране няма съществено значение** за умението да кодиш. Или можеш да кодиш или не. Ако можеш да кодиш на C++, лесно ще се научиш да кодиш и на Java, и на C++, и на друг език. Затова **уменията да кодираш** се изучават доста сериозно в началните курсове за софтуерни инженери в СофтУни (вж. [учебния план](#)) и с тях стартира всяка книга за програмиране за напълно начинаещи, включително нашата.

Умение #2 – алгоритмично мислене (30%)

Алгоритмичното (логическо, инженерно, математическо, абстрактно) мислене формира около 30% от минималните умения на програмиста за старт в професията. **Алгоритмичното мислене** е умението да разбивате една задача на логическа последователност от стъпки (алгоритъм) и да намирате решение за всяка отделна стъпка, след което да сглобявате стъпките в работещо решение на първоначалната задача. Това е най-важното умение на програмиста.

Как да си изградим алгоритмично мислене?

- Алгоритмичното мислене се развива се чрез решаване на **много (1000+)** задачи по програмиране, възможно най-разнообразни. Това е рецептата:

решаване на хиляди практически задачи, измисляне на алгоритъм за тях и имплементиране на алгоритъма, заедно с дебъгване на грешките по пътя.

- Помагат физика, математика и/или подобни науки, но не са задължителни! Хората с **инженерни и технически наклонности** обикновено по-лесно се научават да мислят логически, защото имат вече изградени умения за решаване на проблеми, макар и не алгоритмични.
- Способността **да решавате задачи по програмиране** (за която е нужно алгоритмично мислене) е изключително важна за програмиста. Много фирми изпитват единствено това умение при интервюта за работа.

Настоящата книга развива **начално ниво на алгоритмично мислене**, но съвсем не е достатъчна, за да ви направи добър програмист. За да станете кадърни в професията, ще трябва да добавите **умения за логическо мислене и решаване на задачи** отвъд обхвата на тази книга, например работа със **структурите от данни** (масиви, списъци, матрици, хеш-таблици, дърворидни структури) и базови **алгоритми** (търсене, сортиране, обхождане на дърворидни структури, рекурсия и други).

Умения за алгоритмично мислене се развиват сериозно в началните курсове за софтуерни инженери в СофтУни (вж. [учебния план](#)), както и в специализираните курсове по [структурите от данни](#) и [алгоритми](#).

Както може би се досещате, **езикът за програмиране няма значение** за разяването на алгоритмичното мислене. Да мислиш логически е универсално, дори не е свързано само с програмирането. Именно заради силно развитото логическото мислене се счита, че **програмистите са доста умни** и че прост човек не може да стане програмист.

Умение #3 – фундаментални знания за професията (25%)

Фундаменталните знания и умения за програмирането, разработката на софтуер, софтуерното инженерство и компютърните науки формират около 25% от минималните умения на програмиста за започване на работа. Ето по-важните от тези знания и умения:

- **базови математически концепции**, свързани с програмирането: координатни системи, вектори и матрици, дискретни и недискретни математически функции, крайни автомати и state machines, понятия от комбинаториката и статистика, сложност на алгоритъм, математическо моделиране и други
- **умения да програмираш** - писане на код, работа с данни, ползване на условни конструкции и цикли, работа с масиви, списъци и асоциативни масиви, стрингове и текстообработка, работа с потоци и файлове, ползване на програмни интерфейси (APIs), работа с дебъгер и други
- **структурите от данни и алгоритми** - списъци, дървета, хеш-таблици, графи, търсене, сортиране, рекурсия, обхождане на дърворидни структури и други

- **обектно-ориентирано програмиране** (ООП) – работа с класове, обекти, наследяване, полиморфизъм, абстракция, интерфейси, капсуляция на данни, управление на изключения, шаблони за дизайн
- **функционално програмиране** (ФП) - работа с ламбда функции, функции от по-висок ред, функции, които връщат като резултат функция, затваряне на състояние във функция (closure) и други
- **бази данни** - релационни и нерелационни бази данни, моделиране на бази данни (таблици и връзки между тях), език за заявки SQL, технологии за обектно-релационен достъп до данни (ORM), транзакционност и управление на транзакции
- **мрежово програмиране** - мрежови протоколи, мрежова комуникация, TCP/IP, понятия, инструменти и технологии от компютърните мрежи
- взаимодействие **клиент-сървър**, комуникация между системи, back-end технологии, front-end технологии, MVC архитектури
- **технологии за сървърна (back-end) разработка** - архитектура на уеб сървър, HTTPS протокол, MVC архитектура, REST архитектура, frameworks за уеб разработка, templating engines
- **уеб front-end технологии (клиентска разработка)** - HTML, CSS, JS, HTTPS, DOM, AJAX, комуникация с back-end, извикване на REST API, front-end frameworks, базови дизайн и UX (user experience) концепции
- **мобилни технологии** - мобилни приложения, Android и iOS разработка, мобилен потребителски интерфейс (UI), извикване на сървърна логика
- **вградени системи** - микроконтролери, управление на цифров и аналогов вход и изход, достъп до сензори, управление на периферия
- **операционни системи** - работа с операционни системи (Linux, Windows и други), инсталация, конфигурация и базова системна администрация, работа с процеси, памет, файлова система, потребители, многозадачност, виртуализация и контейнери
- **паралелно програмиране и асинхронност** - управление на нишки, асинхронни задачи, promises, общи ресурси и синхронизация на достъпа
- **софтуерно инженерство** - сорс контрол системи, управление на разработката, планиране и управление на задачи, методологии за софтуерна разработка, софтуерни изисквания и прототипи, софтуерен дизайн, софтуерни архитектури, софтуерна документация
- **софтуерно тестване** - компонентно тестване (unit testing), test-driven development, QA инженерство, докладване на грешки и трекери за грешки, автоматизация на тестването, билд процеси и непрекъсната интеграция

Трябва да поясним и този път, че **езикът за програмиране няма значение** за усвояването на всички тези умения. Те се натрупват бавно, в течение на много години практика в професията. Някои знания са фундаментални и могат да се

усвояват теоретично, но за пълното им разбиране и осъзнаването им дълбочина, са необходими години практика.

Фундаментални знания и умения за програмирането, разработката на софтуер, софтуерното инженерство и компютърните науки се учат по време на [ялостната програма за софтуерни инженери в СофтУни](#), както и с редица [изборни курсове](#). Работата с разнообразни софтуерни библиотеки, програмни интерфейси (APIs), технологични рамки (frameworks) и софтуерни технологии и тяхното взаимодействие, постепенно изграждат тези знания и умения, така че не очаквайте да ги добиете от единичен курс, книга или проект.

За започване на работа като програмист обикновено са достатъчни само **начални познания в изброените по-горе области**, а задълбоването става на работното място според използваните технологии и инструменти за разработка в съответната фирма и екип.

Умение #4 - езици за програмиране и софтуерни технологии (25%)

Езиците за програмиране и технологиите за софтуерна разработка формират около 25% от минималните умения на програмиста. Те са най-обемни за научаване, но най-бързо се променят с времето. Ако погледнем **обявите за работа** от софтуерната индустрия, там често се споменават всякачки думички (като изброените по-долу), но всъщност в обявите мълчаливо **се подразбираят първите три умения**: да кодиш, да мислиш алгоритично и да владееш фундамента на компютърните науки и софтуерното инженерство.

За тези чисто технологични умения вече **езикът за програмиране има значение**.

- **Обърнете внимание:** само за тези 25% от професията има значение езикът за програмиране!
- **За останалите 75% от уменията няма значение езикът** и тези умения са устойчиви във времето и преносими между различните езици и технологии.

Ето и някои често използвани езици и технологии (software development stacks), които се търсят от софтуерните фирми (актуални към февруари 2019 г.):

- C# + ООП + ФП + класовете от .NET + база данни SQL Server + Entity Framework (EF) + ASP.NET MVC + HTTPS + HTML + CSS + JS + DOM + jQuery
- Java + Java API classes + ООП + ФП + бази данни + MySQL + HTTPS + уеб програмиране + HTML + CSS + JS + DOM + jQuery + JSP/Servlets + Spring MVC или Java EE / JSF
- PHP + ООП + бази данни + MySQL + HTTPS + уеб програмиране + HTML + CSS + JS + DOM + jQuery + Laravel / Symfony / друг MVC framework за PHP
- JavaScript (JS) + ООП + ФП + бази данни + MongoDB или MySQL + HTTPS + уеб програмиране + HTML + CSS + JS + DOM + jQuery + Node.js + Express +

Angular или React

- Python + ООП + ФП + бази данни + MongoDB или MySQL + HTTPS + уеб програмиране + HTML + CSS + JS + DOM + jQuery + Django
- C++ + ООП + STL + Boost + native development + бази данни + HTTPS + други езици
- Swift + MacOS + iOS + Cocoa + Cocoa Touch + XCode + HTTPS + REST + други езици

Ако изброените по-горе думички ви изглеждат страшни и абсолютно непонятни, значи сте съвсем в началото на кариерата си и имате **да учите още години** докато достигнете професията "софтуерен инженер". Не се притеснявайте, всеки програмист преминава през един или няколко технологични стека и се налага да изучи **съвкупност от взаимосвързани технологии**, но в основата на всичко това стои **умението да пишеш програмна логика (да кодиш)**, което се развива в тази книга, и умението **да мислиш алгоритично** (да решаваш задачи по програмиране). Без тях не може!

Езикът за програмиране няма значение!

Както вече стана ясно, **разликата между езиците за програмиране** и по-точно между уменията на програмистите на различните езици и технологии, е в около **10-20% от уменията**.

- Всички програмисти имат около **80-90% еднакви умения**, които не зависят от езика! Това са уменията да програмираш и да разработваш софтуер, които са много подобни в различните езици за програмиране и технологии за разработка.
- Колкото повече езици и технологии владеете, толкова по-бързо ще учите нови и толкова по-малко ще усещате разлика между тях.

Наистина, **езикът за програмиране почти няма съществено значение**, просто трябва да се научите да програмирате, а това започва с **коденето** (настоящата книга), продължава в по-сложните **концепции от програмирането** (като структури от данни, алгоритми, ООП и ФП) и включва усвояването на **фундаментални знания и умения за разработката на софтуер, софтуерното инженерство и компютърните науки**.

Едва накрая, когато захванете конкретни технологии в даден софтуерен проект, ще ви трябват **конкретен език за програмиране**, познания за конкретни програмни библиотеки (APIs), технологични рамки (frameworks) и софтуерни технологии (front-end UI технологии, back-end технологии, ORM технологии и други). Спокойно, ще ги научите, всички програмисти ги научават, но първо се научават на фундамента: **да програмират и то добре**.

Настоящата книга използва езика C++, но той не е съществен и може да се замени със C#, JavaScript, Python, PHP, Java, Ruby, Swift, Go, Kotlin или друг език. За овладяване на **професията "софтуерен разработчик"** е необходимо да се научите

да кодите (20%), да се научите да мислите алгоритмично и да решавате проблеми (30%), да натрупате **фундаментални знания по програмиране и компютърни науки** (25%) и да владеете **конкретен език за програмиране и технологиите около него** (25%). Имайте търпение, за година-две всичко това може да се овладее на добро начално ниво, стига да сте сериозни и усърдни.

Книгата в помощ на учителите

Ако сте учител по програмиране, информатика или информационни технологии или искате да преподавате програмиране, тази книга ви дава нещо повече от добре структуриран учебен материал с много примери и задачи. **Бесплатно** към книгата получавате **качествено учебно съдържание** за преподаване в училище, на **български език**, съобразено с училищните изисквания:

- Учебни презентации (PowerPoint слайдове) за всяка една учебна тема, съобразени с 45-минутните часове в училищата – бесплатно.
- Добре разработени **задачи за упражнения** в клас и за домашно, с детайлно описани условия и примерен вход и изход – бесплатно.
- Система за автоматизирана проверка на задачите и домашните (online judge system), която да се използва от учениците, също бесплатно.
- **Видео-уроци** с методически указания от **бесплатния курс за учители по програмиране**, който се провежда регулярно от СофтУни фондацията.

Всички тези **бесплатни преподавателски ресурси** можете да намерите на сайта на СофтУни фондацията, заедно с учебно съдържание за цяла поредица от курсове по програмиране и софтуерни технологии. Изтеглете ги свободно от тук: <https://softuni.foundation/projects/applied-software-developer-profession/>.

Историята на тази книга

Главен двигател и ръководител на проекта за създаване на настоящата **свободна книга по програмиране за начинаещи** с отворен код е [д-р Светлин Наков](#). Той е основен идеолог и създател на учебното съдържание от [курса "Основи на програмирането" в СофтУни](#), който е използван за основа на книгата.

Всичко започва с масовите **бесплатни курсове по основи на програмирането**, провеждани в цялата страна от 2014 г. насам, когато стартира инициативата "Софтуни". В началото тези курсове имат по-голям обхват и включват повече теория, но през 2016 г. д-р Светлин Наков изцяло ги преработва, обновява, опростява и насочва много силно към практиката. Така е създадено ядрото на **учебното съдържание от тази книга**.

Бесплатните обучения на СофтУни за старт в програмирането са може би най-мащабните, провеждани някога в България. До 2017 г. курсът на СофтУни по основи на програмирането **се провежда над 100 пъти в близо 30 български града** присъствено и многократно онлайн, с над 40 000 участника. Съвсем естествено

възниква и нуждата да се напише **учебник** за десетките хиляди участници в курсовете на СофтУни по програмиране за начинаещи. На принципа на свободния софтуер и свободното знание, Светлин Наков повежда **екип от доброволци** и задвижва този open-source проект, първоначално за създаване на книга по основи на програмирането с езика C#, а по-късно и с други езици за програмиране. Настоящата книга е почти огледален превод към C++ от нейната оригинална C# версия.

Проектът е част от усилията на [Фондация "Софтуерен университет"](#) да създава и разпространява отворено учебно съдържание за обучение на софтуерни инженери и ИТ специалисти.

Авторски колектив

Настоящата книга “Основи на програмирането със C++” е разработена от широк авторски колектив от **доброволци**, които отделиха от своето време, за да ви подарат тези систематизирани знания и насоки при старта в програмирането. Списък на всички автори и редактори (по алфобетен ред):

Андрей Царев, Божидар Колев, Борис Горанов, Венцислав Петров, Георги Лацов, Георги Шавов, Денис Миланов, Димитър Кацарски, Димо Димов, Димо Чанев, Зорница Йоханова, Игнеш Симеонова, Йордан Добрев, Любомир Господинов, Мирела Дамянова, Николай Костов, Пламен Динев, Цветан Иванов

Книгата е базирана на нейния първоначален C# вариант ([Въведение в програмирането със C#](#)), който е разработен от широк авторски колектив и който има принос и към настоящата книга:

Александър Кръстев, Александър Лазаров, Ангел Димитриев, Васко Викторов, Венцислав Петров, Даниел Цветков, Димитър Татарски, Димо Димов, Диян Тончев, Елена Роглева, Живко Недялков, Жулиета Атанасова, Захария Пехливанова, Ивелин Кирилов, Искра Николова, Калин Примов, Кристиян Памидов, Любослав Любенов, Николай Банкин, Николай Димов, Павлин Петков, Петър Иванов, Росица Ненова, Руслан Филипов, Светлин Наков, Стефка Василева, Теодор Куртев, Тоньо Желев, Християн Христов, Христо Христов, Цветан Илиев, Юlian Линев, Яница Вълева

Дизайн на корица: Марина Шидерова – <https://linkedin.com/in/marina-shideroff>.

Книгата е написана в периода декември 2018 - март 2019.

Официален сайт на книгата

Настоящата книга по **основи на програмирането със C++** за начинаещи е достъпна за свободно ползване в Интернет от адрес:

<https://cpp-book.softuni.bg>

Това е **официалният сайт на книгата** и там ще бъде качвана нейната последна версия. Книгата е преведена огледално и на други езици за програмиране, посочени на нейния сайт.

Форум за вашите въпроси

Задавайте вашите въпроси към настоящата книга по основи на програмирането във форума на СофтУни:

<https://softuni.bg/forum>

В този дискусионен форум ще получите безплатно **адекватен отговор по всякаакви въпроси от учебното съдържание на настоящия учебник**, както и по други въпроси от програмирането. Общността на СофтУни за навлизящи в програмирането е толкова голяма, че обикновено отговор на зададен въпрос се получава **до няколко минути**. Преподавателите, асистентите и менторите от СофтУни също отговарят постоянно на вашите въпроси.

Поради големия брой учащи по настоящия учебник, във форума можете да намерите **решение на практически всяка задача от него**, споделено от ваш колега. Хиляди студенти преди вас вече са решавали същите задачи, така че ако закъснате, потърсете из форума. Макар и задачите в курса "Основи на програмирането" да се сменят от време на време, споделянето е винаги настърчавано в СофтУни и затова лесно ще намерите решения и насоки за всички задачи.

Ако все пак имате конкретен въпрос, например защо не тръгва дадена програма, над която умувате от няколко часа, **задайте го във форума** и ще получите отговор. Ще се учудите колко добронамерени и отзивчиви са обитателите на СофтУни форума.

Официална Facebook страница на книгата

Книгата си има и **официална Facebook страница**, от която може да следите за новини около книгите от поредицата "Основи на програмирането", нови издания, събития и инициативи:

fb.com/IntroProgrammingBooks

Лиценз и разпространение

Книгата се разпространява **бесплатно** в електронен формат под отворен лиценз [CC-BY-SA](#).

Книгата се издава и разпространява **на хартия** от СофтУни и хартиено копие може да се закупи от рецепцията на СофтУни (вж. <https://softuni.bg/contacts>).

Сурс кодът на книгата може да се намери в GitHub:

<https://github.com/SoftUni/Programming-Basics-Book-CPP-BG>.

Международен стандартен номер на книга ISBN: 978-619-00-0951-1.

Докладване на грешки

Ако откриете **грешки**, неточности или дефекти в книгата, можете да ги докладвате в официалния тракер на проекта:

<https://github.com/SoftUni/Programming-Basics-Book-CPP-BG/issues>

Не обещаваме, че ще поправим всичко, което ни изпратите, но пък имаме желание **постоянно да подобряваме качеството** на настоящата книга, така че докладваните безспорни грешки и всички разумни предложения ще бъдат разгледани.

Приятно четене!

И не забравяйте **да пишете код** в големи количества, да **пробвате примерите** от всяка тема и най-вече да **решавате задачите от упражненията**. Само с четене няма да се научите да програмирате, така че решавайте задачи здраво!

Глава 1. Първи стъпки в програмирането

В тази глава ще разберем **какво е програмирането** в неговата същина. Ще се запознаем с идеята за **програмни езици** и ще разгледаме **средите за разработка на софтуер** (IDE) и как да работим с тях, в частност с **Visual Studio**. Ще напишем и изпълним **първата си програма** на програмния език **C++**, а след това ще се упражним с няколко задачи: ще създадем конзолни програми. Ще се научим как да проверяваме за коректност решенията на задачите от тази книга в **Judge системата на СофтУни** и накрая ще се запознаем с типичните грешки, които често се допускат при писането на код и как да се предпазим от тях.

Видео

Гледайте видео урок по учебния материал от настоящата глава от книгата: <https://youtube.com/watch?v=CYPURYobOT8>.

Какво означава "да програмираме"?

Да програмираме означава да даваме команди на компютъра какво да прави, например "да иззвири някакъв звук", "да отпечата нещо на екрана" или "да умножи две числа". Когато командите са няколко една след друга, те се наричат **компютърна програма**. Текстът на компютърните програми се нарича **програмен код** (или **сурс код** или за по-кратко **код**).

Компютърни програми

Компютърните програми представляват **поредица от команди**, които се изписват на предварително избран **език за програмиране**, например C++, Java, JavaScript, Python, Ruby, PHP, C, C#, Swift, Go или друг. За да пишем команди, трябва да знаем **синтаксиса и семантиката на езика**, с който ще работим, в нашия случай **C++**. Затова ще се запознаем със синтаксиса и семантиката на езика C++ и с програмирането като цяло в настоящата книга, изучавайки стъпка по стъпка писането на код, от по-простите към по-сложните програмни конструкции.

Алгоритми

Компютърните програми обикновено изпълняват някакъв алгоритъм. **Алгоритмите** са последователност от стъпки, необходими, за да се свърши определена работа и да се постигне някакъв очакван резултат, нещо като "рецепта". Например, ако пържим яйца, ние изпълняваме някаква рецепта (алгоритъм): загряваме мазнина в някакъв съд, чупим яйцата, изчакваме докато се изпържат, отместваме от огъня. Аналогично, в програмирането **компютърните програми изпълняват алгоритми**: поредица от команди, необходими, за да се свърши определена работа. Например, за да се подредят поредица от числа в

нарастващ ред, е необходим алгоритъм, примерно да се намери най-малкото число и да се отпечата, от останалите числа да се намери отново най-малкото число и да се отпечата и това се повтаря, докато числата свършат.

За удобство при създаването на програми, за писане на програмен код (команди), за изпълнение на програмите и за други операции, свързани с програмирането, ни е необходима и среда за разработка, например Visual Studio.

Езици за програмиране, компилатори, интерпретатори и среди за разработка

Езикът за програмиране е изкуствен език (синтаксис за изразяване), предназначен за **задаване на команди**, които искаме компютъра да прочете, обработи и изпълни. Чрез езиците за програмиране пишем поредици от команди (**програми**), които **задават какво да прави компютъра**. Изпълнението на компютърните програми може да се реализира с **компилатор** или с **интерпретатор**.

Компилаторът превежда кода от програмен език на **машинен код**, като за всяка от конструкциите (командите) в кода избира подходящ, предварително подготвен фрагмент от машинен код, като междувременно **роверява за грешки текста на програмата**. Заедно компилираните фрагменти съставят програмата в машинен код, както я очаква микропроцесорът на компютъра. След като е компилирана програмата, тя може да бъде директно изпълнена от микропроцесора в кооперация с операционната система. При компилируемите езици за програмиране **компилирането на програмата** се извършва задължително преди нейното изпълнение и по време на компилация се откриват синтактичните грешки (грешно зададени команди). С компилатор работят езици като C++, C#, Java, Swift и Go.

Някои езици за програмиране не използват компилатор, а се **интерпретират директно** от специализиран софтуер, наречен "интерпретатор". **Интерпретаторът** е "програма за изпълняване на програми", написани на някакъв програмен език. Той изпълнява командите на програмата една след друга, като разбира не само от единични команди и поредици от команди, но и от другите езикови конструкции (роверки, повторения, функции и т.н.). Езици като PHP, Python и JavaScript работят с интерпретатор и се изпълняват без да се компилират. Поради липса на предварителна компилация, при интерпретирамите езици **грешките се откриват по време на изпълнение**, след като програмата започне да работи, а не предварително.

Средата за програмиране (Integrated Development Environment - IDE, интегрирана среда за разработка) е съвкупност от традиционни инструменти за разработване на софтуерни приложения. В средата за разработка пишем код, компилираме и изпълняваме програмите. Средите за разработка интегрират в себе си **текстов редактор** за писане на кода, **език за програмиране**, **компилатор** или **интерпретатор** и **среда за изпълнение** на програмите, **дебъгер** за проследяване на програмата и

търсене на грешки, инструменти за дизайн на потребителски интерфейс и други инструменти и добавки.

Средите за програмиране са удобни, защото интегрират всичко необходимо за разработката на програмата, без да се напуска средата. Ако не ползваме среда за разработка, ще трябва да пишем кода в текстов редактор, да го компилираме с команда от конзолата, да го изпълняваме с друга команда от конзолата и да пишем още допълнителни команди, когато се налага, и това ще ни губи време. Затова повечето програмисти ползват IDE в ежедневната си работа.

За програмиране на **езика C++** най-често се ползва средата за разработка **Visual Studio**, която се разработва и разпространява бесплатно от Microsoft и може да се изтегли от: <https://www.visualstudio.com/downloads>. Друга разпространена среда за разработка е **Code::Blocks** (<https://www.codeblocks.org/downloads>). В настоящата книга ще използваме средата за разработка **Visual Studio**.

Езици от ниско и високо ниво, среди за изпълнение (Runtime Environments)

Програмата в своята същност е **набор от инструкции**, които карат компютъра да свърши определена задача. Те се въвеждат от програмиста и се **изпълняват безусловно от машината**.

Съществуват различни видове **езици за програмиране**. С езиците от най-ниско ниво могат да бъдат написани **самите инструкции**, които **управляват процесора**, например с езика "assembler". С езици от малко по-високо ниво като **C** и **C++** могат да бъдат създадени операционна система, драйвери за управление на хардуера (например драйвер за видеокарта), уеб браузъри, компилатори, двигатели за графика и игри (game engines) и други системни компоненти и програми. С езици от още по-високо ниво като **C#, Python** и **JavaScript** създават приложни програми, например програма за четене на поща или чат програма.

Езиците от ниско ниво управляват директно хардуера и изискват много усилия и огромен брой команди, за да свършат единица работа. **Езиците от по-високо ниво** изискват по-малко код за единица работа, но нямат директен достъп до хардуера. На тях се разработва приложен софтуер, например уеб приложения и мобилни приложения.

Болшинството софтуер, който използваме ежедневно, като музикален плейър, видеоплейър, GPS програма и т.н., се пише на **езици за приложно програмиране**, които са от високо ниво, като **C++, Java, Python, C#, JavaScript, PHP** и др.

C++ е компилируем език, а това означава, че пишем команди, които се компилират до машинен код преди да се изпълнят. Именно тези команди, чрез помощна програма (компилатор), се преобразуват във файл, който може да се изпълнява (executable). За да пишем на език като **C++** ни трябва текстов редактор или среда за разработка.

Компютърни програми - компилация и изпълнение

Както вече споменахме, програмата е **последователност от команди**, иначе казано тя описва поредица от пресмятания, проверки, повторения и всякакви подобни операции, които целят постигане на някакъв резултат.

Програмата се пише в текстов формат, а самият текст на програмата се нарича **сорс код** (source code). Той се компилира до **изпълним файл** (например **Program.cpp** се компилира до **Program.exe**).

Процесът на **компилация** на кода преди изпълнение е присъщ само за компилируеми езици като C++, Java, C# и др. При **скриптови и интерпретеруеми езици**, като JavaScript, Python и PHP, сорс кодът се изпълнява **последователно** от интерпретатор.

Компютърни програми – примери

Да започнем с много прост пример за кратка C++ програма.

Пример: програма, която отпечатва даден текст

Нашата първа програма ще е единична C++ команда, която отпечатва текста "Hello":

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

int main() {
    cout << "Hello";

    return 0;
}
```

Може да тествате примера онлайн: <https://repl.it/@nakov/CPlusPlus-Hello>.

След малко ще разберем как можем да изпълним тази команда и да видим отпечатания текст, но засега нека само разгледаме какво представляват командите в програмирането. Да се запознаем с още няколко примера.

Пример: програма, която отпечатва английската азбука

Можем да усложним предходната програма, като зададем за изпълнение повтарящи се в цикъл команди за извеждане на текст:

```
for (char ch = 'A'; ch <= 'Z'; ch++) {
    cout << ch << endl;
}
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/EnglishAlphabet>.

В горния пример караме компютъра да отпечатва един след друг, на нови редове, много символи, започвайки от A и приключвайки с Z. Резултатът от програмата е отпечатване на цялата английска азбука.

Как работят повторенията (циклите) в програмирането ще научим в [глава "Повторения \(цикли\)"](#), но засега нека приемем, че просто повтаряме някаква команда много пъти.

Пример: програма, която конвертира от левове в евро

Да разгледаме още една проста програма, която прочита от потребителя някаква сума в лева (цяло число), конвертира я в евро (като я разделя на курса на еврото) и отпечатва получения резултат. Това е програма от 4 поредни команди:

```
double leva;
cin >> leva;
double euro = leva / 1.95583;
cout << euro << endl;
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/LevaToEuro>.

Разгледахме **три примера за компютърни програми**: единична команда, серия команди в цикъл и поредица от 4 команди. Нека сега преминем към по-интересното: как можем да пишем собствени програми на C++ и как можем да ги компилираме и изпълняваме?

Как да напишем конзолна програма?

Нека преминем през **стъпките за създаване и изпълнение на компютърна програма**, която чете и пише своите данни от и на текстова конзола (прозорец за въвеждане и извеждане на текст). Такива програми се наричат "**конзолни**". Преди това, обаче, трябва първо да си **инсталдраме и подгответим** средата за разработка, в която ще пишем и изпълняваме C++ програмите от тази книга и упражненията към нея.

Среда за разработка (IDE)

Както вече стана дума, за да програмираме ни е нужна **среда за разработка - Integrated Development Environment (IDE)**. Това всъщност е редактор за програми, в който пишем програмния код и можем да го изпълняваме, да виждаме грешките, да ги поправяме и да стартираме програмата отново.

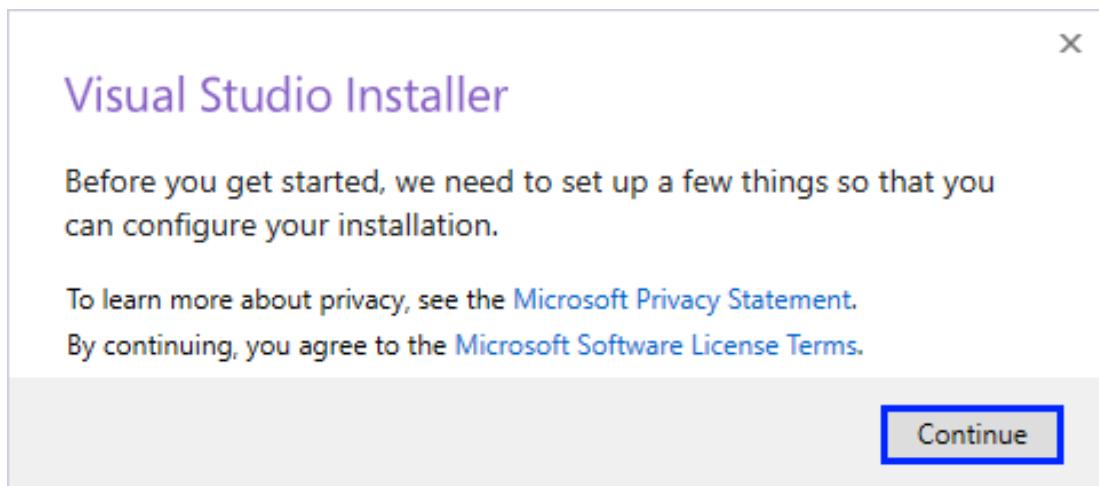
- За програмиране на C++ използваме средата **Visual Studio** за операционната система Windows и **Eclipse** за Linux или Mac OS X.
- Ако програмираме на Java, подходящи са средите **IntelliJ IDEA**, **Eclipse** или **NetBeans**.
- Ако ще пишем на Python, можем да използваме средата **PyCharm**.

Инсталация на Visual Studio Community

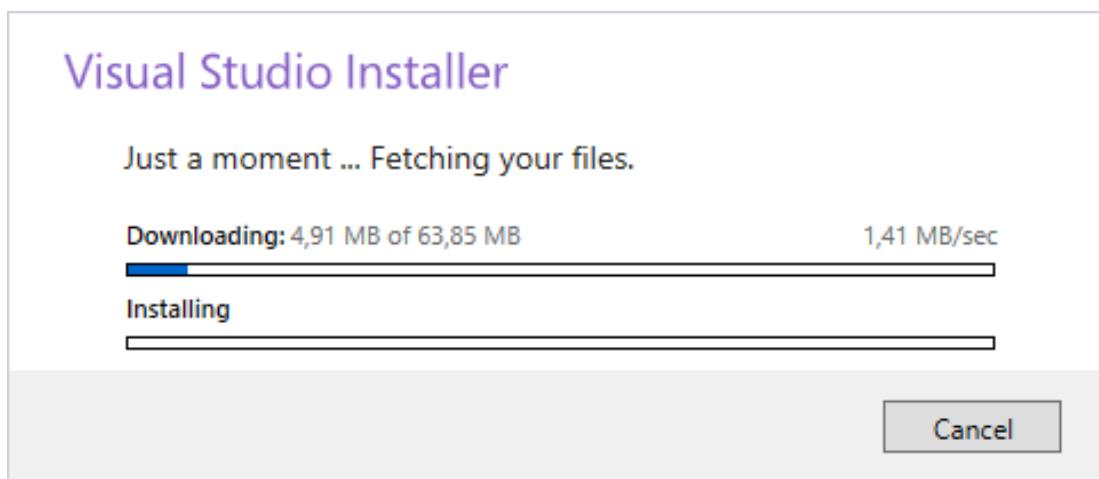
Започваме с инсталацията на интегрираната среда Microsoft Visual Studio Community (версия 2017, актуална към януари 2019 г.).

Community версията на Visual Studio (VS) се разпространява бесплатно от Microsoft и може да бъде изтеглена от: <https://www.visualstudio.com/vs/community>. Инсталацията е типичната за Windows с [Next], [Next] и [Finish], но е важно да включим компонентът за "Desktop development with C++". Не е необходимо да променяме останалите настройки за инсталация.

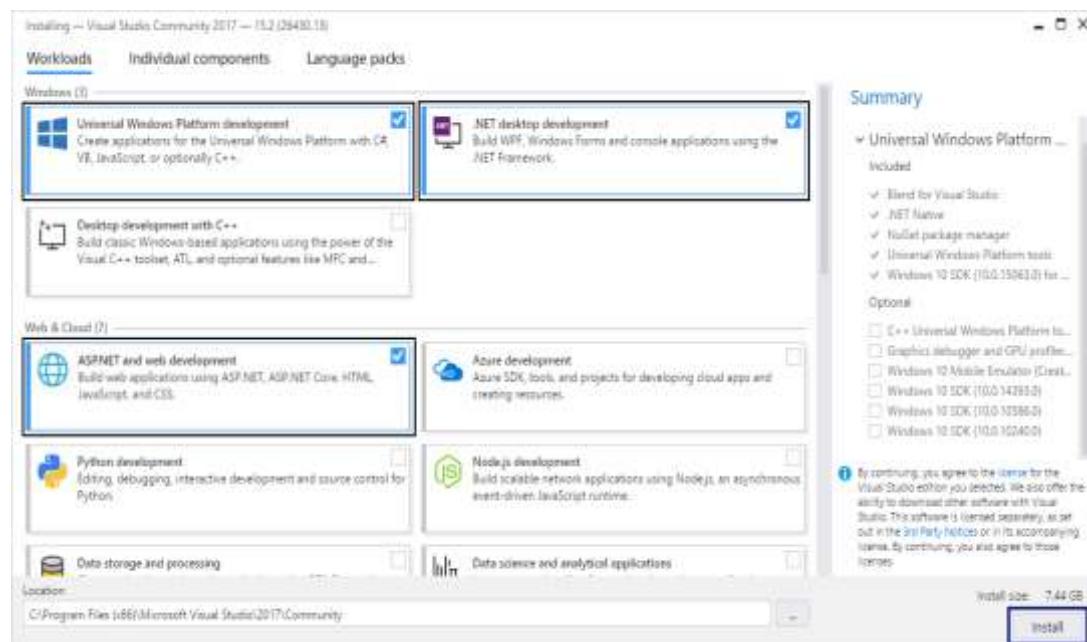
В следващите редове са описани подробно **стъпките за инсталация на Visual Studio** (версия Community 2017). След като свалим инсталационния файл и го стартираме, се появява следният экран:



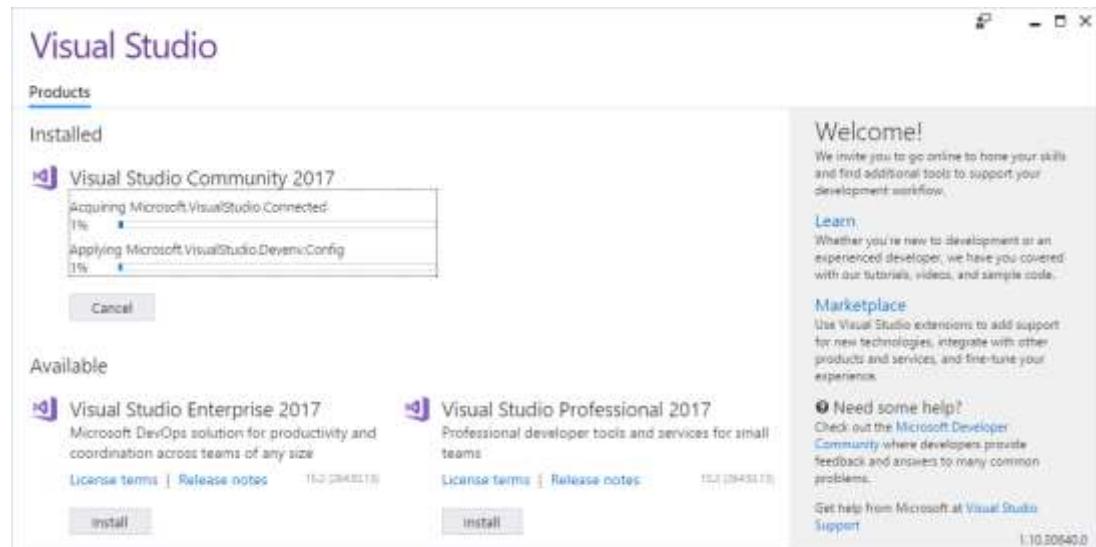
Натискаме бутона [Continue], след което ще видим прозореца долу:



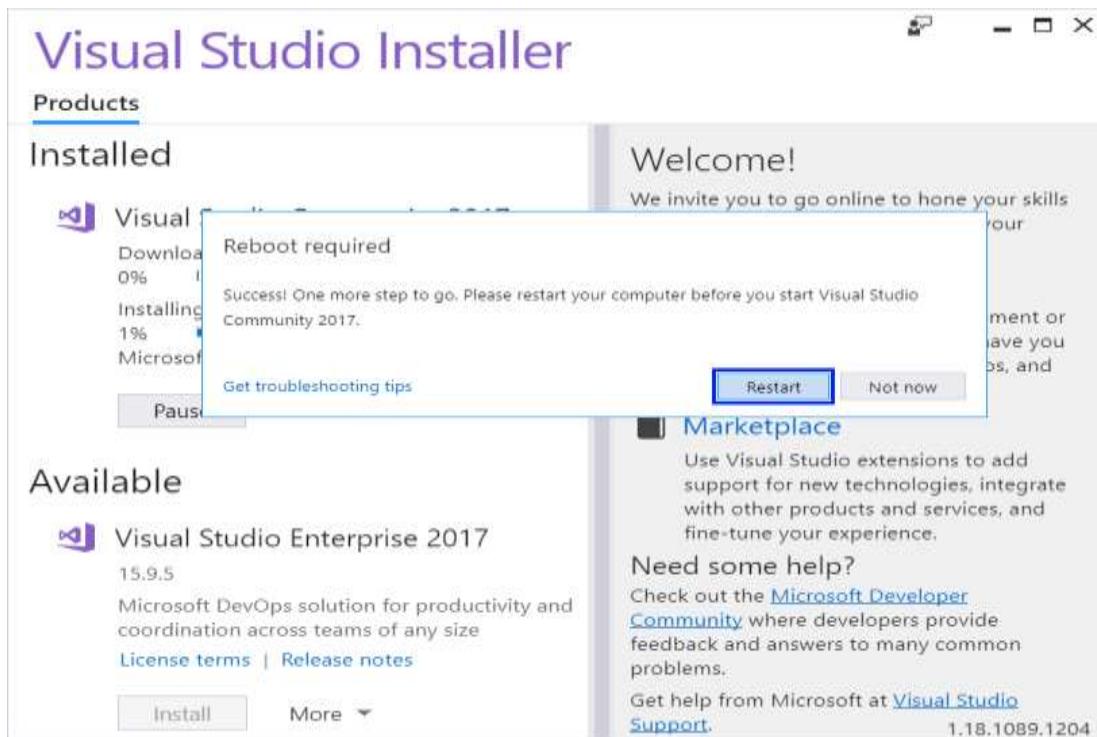
Зарежда се прозорец с инсталационния панел на Visual Studio. Слагаме отметка на [Desktop development with C++], след което натискаме бутона [Install]:



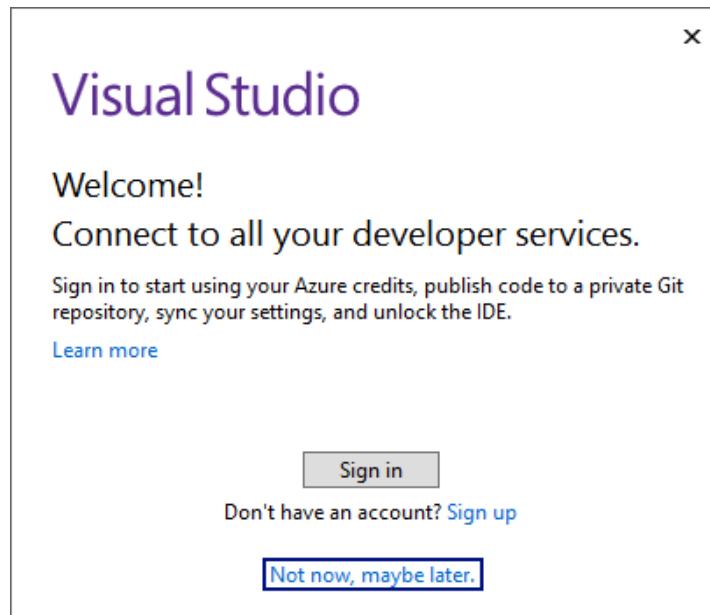
Започва инсталацията на Visual Studio и ще се появи еcran като показания по-долу:



След като Visual Studio се инсталира, ще иска компютърът да бъде рестартиран, преди да започнем работа. Можем да натиснем бутона [Not now], ако не искаме да рестартираме веднага компютъра си, но трябва задължително да го направим преди да ползваме **Visual Studio**. Ако сме готови веднага да рестартираме, натискаме бутона [Restart]. След това компютърът ще се рестартира. Когато се включи отново, пускаме **Visual Studio**. Ако инсталираме по-стара или различна версия е възможно да не се изисква рестартиране. Тогава само трябва да натиснем бутона [Launch], за да стартираме **Visual Studio**:

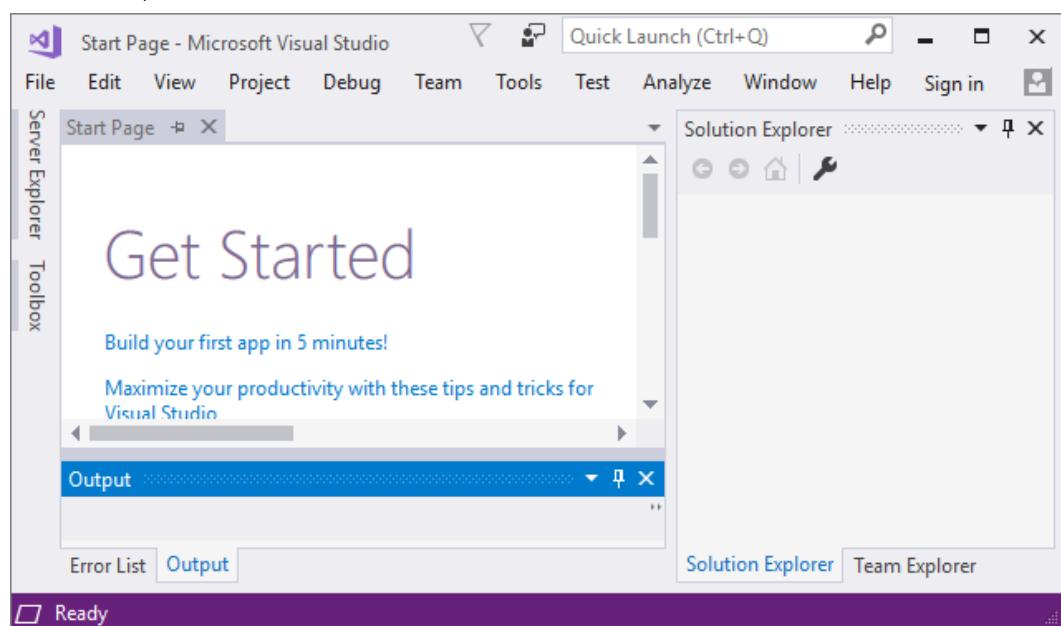


След **старта на VS** излиза екран като този по-долу. От него можем да изберем дали да влезем с Microsoft профила си. За момента избираме да работим без да сме се логнали с Microsoft акаунта си, затова избираме опцията [**Not now, maybe later.**]. На по-късен етап, ако имате такъв акаунт, можете да се логнете, а ако нямате и срещате затруднения със създаването му, винаги можете да пишете във форума на SoftUni: <https://softuni.bg/forum>:



Следващата стъпка е да изберем **цветовата тема**, с която да се визуализира Visual Studio. Тук изборът е изцяло според предпочтенията на потребителя, като няма значение коя опция ще бъде избрана. Темите са съответно Blue, Dark и Light.

Последното, което трябва да направим, е да натиснем бутона **[Start Visual Studio]** и се зарежда началния изглед на Visual Studio Community:



По-стари версии на Visual Studio

Можем да използваме и по-стари версии на Visual Studio (например версия 2015 или 2013 или дори 2010 или 2005), но **не е препоръчително**, тъй като в тях не се съдържат някои от по-новите възможности за разработка и не е сигурно дали всички примери от книгата ще тръгнат.

Онлайн среди за разработка

Съществуват и **алтернативни среди за разработка онлайн**, директно във вашия уеб браузър. Тези среди не са много удобни, но ако нямаете друга възможност, може

да стаптирате обучението си с тях и да си инсталирате Visual Studio на по-късен етап. Ето някои линкове:

- За езика C++ можем да използваме: <https://cpp.sh>
- За езика C# сайтът .NET Fiddle позволява писане на код и изпълнението му онлайн: <https://dotnetfiddle.net>.
- За Java можем да използваме следното онлайн Java IDE: <https://www.compilejava.net>.
- За JavaScript можем да пишем JS код директно в конзолата на даден браузър с натискане на [F12].

Проектни решения и проекти във Visual Studio

Преди да започнем да работим с Visual Studio е нужно да се запознаем с понятията **Visual Studio Solution** и **Visual Studio Project**, които са неизменна част от него.

Visual Studio Project представлява "проектът", върху който работим. В началото това ще са нашите конзолни програми, които ще се научим да пишем с помощта на настоящата книга, ресурсите към нея и в курса Programming Basics в SoftUni. При по-задълбочено изучаване и с времето и практиката, тези проекти ще преминат в приложения, игри и други разработки. Проектът във VS **логически групира множество файлове**, изграждащи дадено приложение или компонент. Един C++ проект съдържа един или няколко C++ сорс файла, конфигурационни файлове и други ресурси. Във всеки C++ сорс файл има една или повече **дефиниции на типове** (класове или други дефиниции). В **класовете** има **методи** (действия), а те се състоят от **поредици от команди**. Изглежда сложно, но при големи проекти такава структура е много удобна и позволява добра организация на работните файлове.

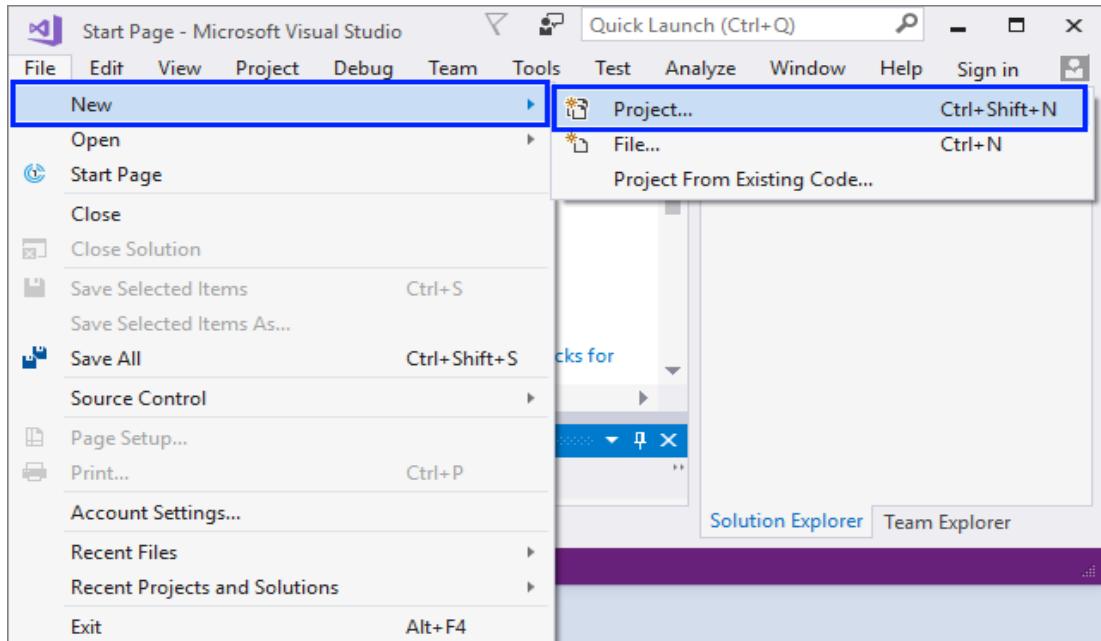
Visual Studio Solution представлява контейнер (работно решение), в който **логически са обединени няколко проекта**. Целта на обединението на тези VS Projects е да има възможност кодът от който и да е от проектите да си взаимодейства с кода на останалите VS проекти, за да може приложението да работи коректно. Когато софтуерният продукт или услуга, който разработваме, е голям, той се изгражда като **VS Solution**, а този Solution се разделя на **проекти** (VS Projects) и във всеки проект има **папки със сорс файлове**. Такава йерархична организация е много удобна при по-сериозни проекти (да кажем над 50 000 реда код).

За **малки проекти** VS Solutions и VS Projects повече усложняват работата, отколкото помагат, но се свиква бързо.

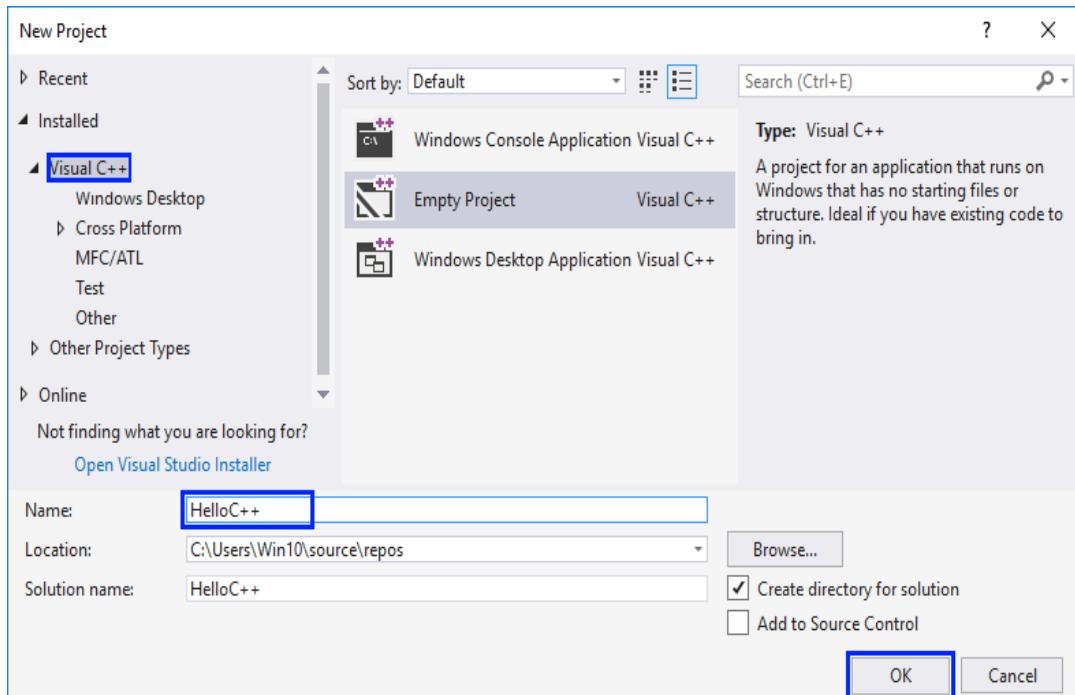
Пример: създаване на конзолна програма "Hello C++"

Да се върнем на нашата конзолна програма. Вече имаме Visual Studio и можем да

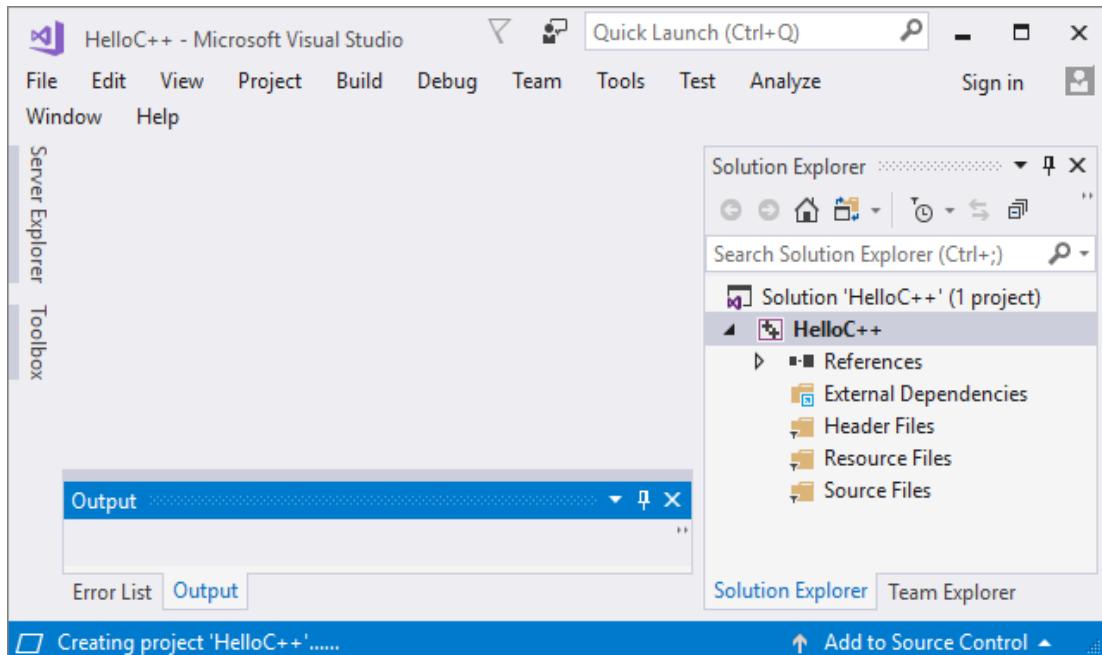
го стартираме. След това създаваме нов конзолен проект: [File] → [New] → [Project...]:



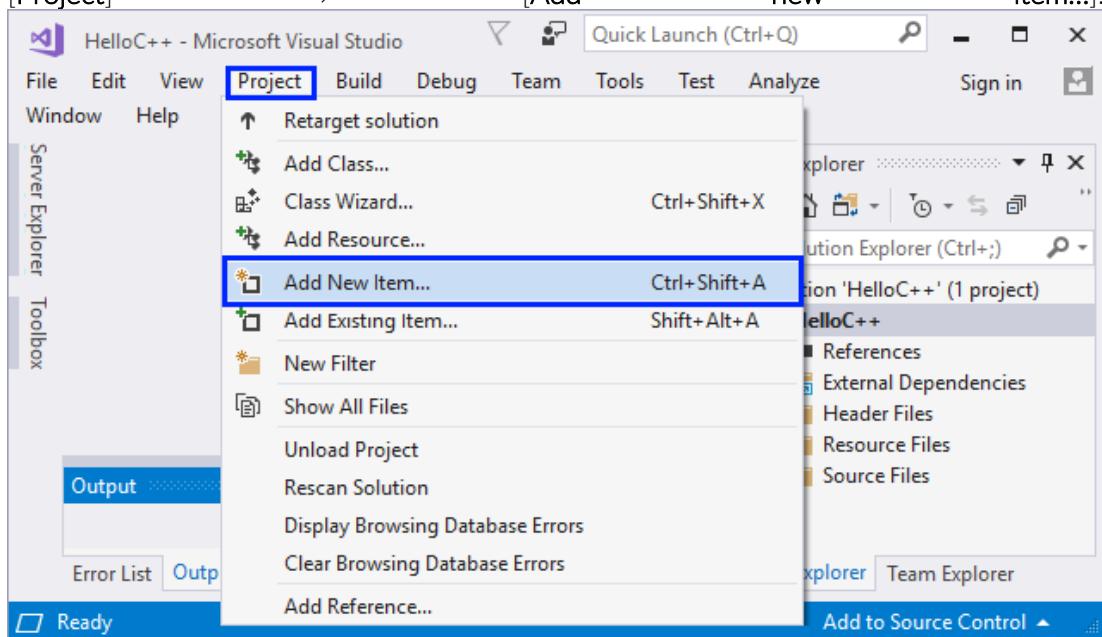
Ще се отвори прозорец, който ще изглежда като картичка по-долу. Трябва да изберем [Visual C++] → [Empty Project]. След което задаваме **смислено име** на нашия проект, например "**HelloC++**".



Когато сме готови, натискаме бутона [OK] и Visual Studio ще създаде за нас VS Solution с един празен C++ проект в него:

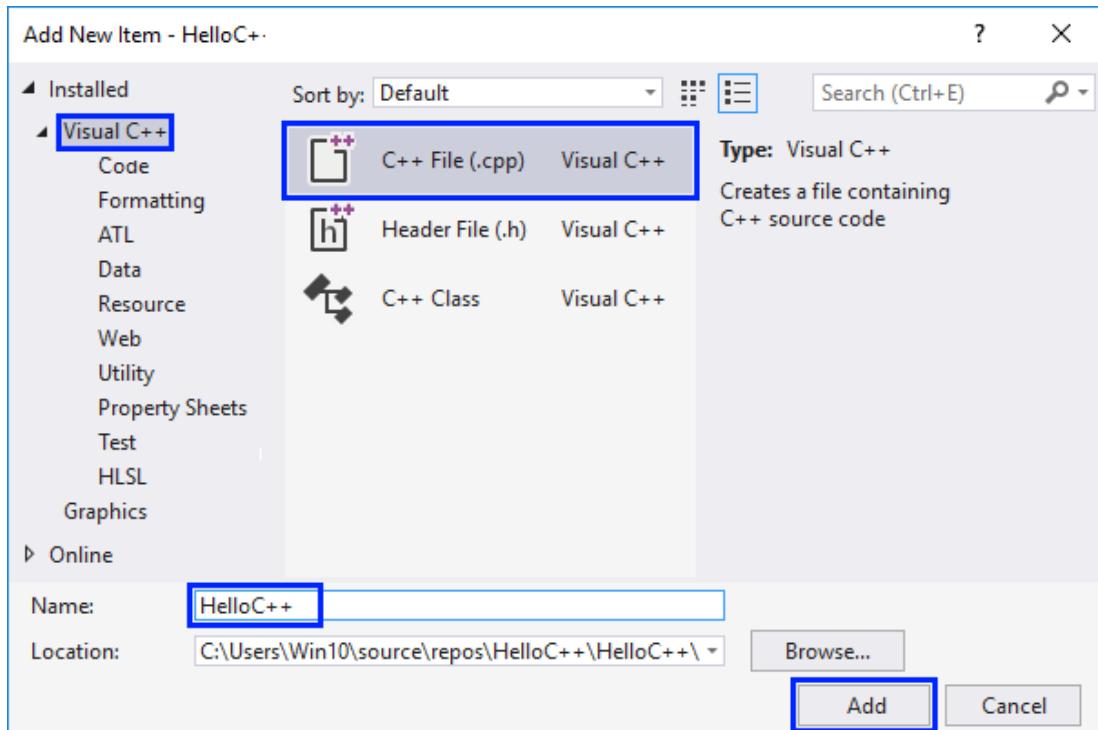


За да добавим **празна C++ програма** в новосъздадения проект, трябва да изберем:
 [Project] → [Add] new item...:

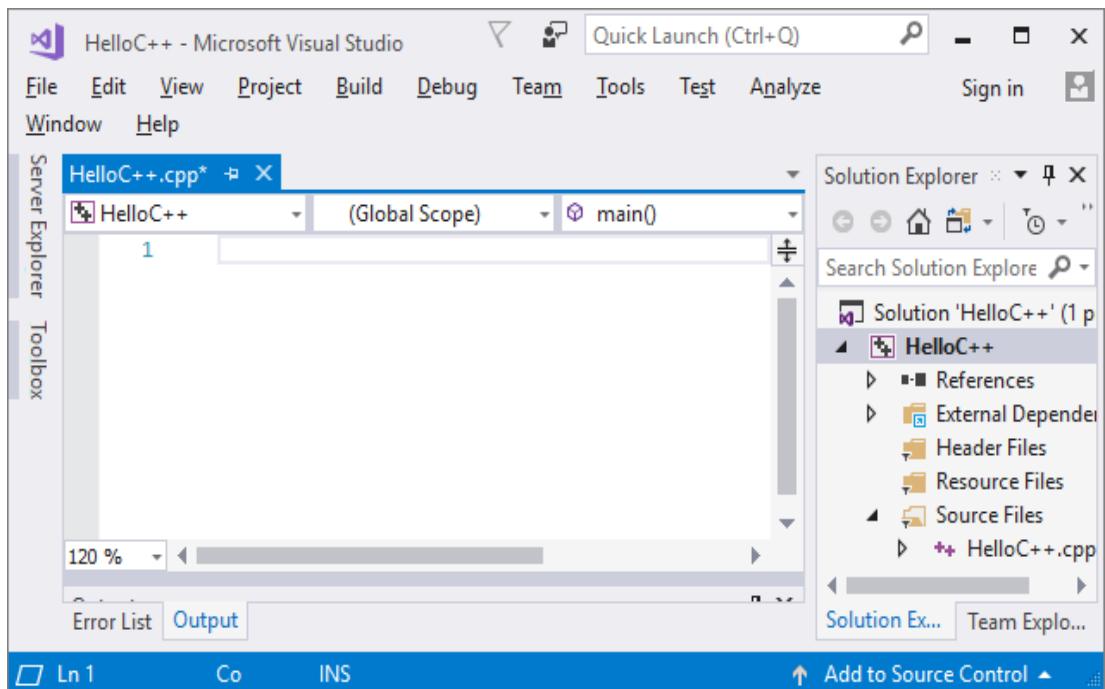


След което ще се отвори прозорец, в който трябва да изберем [Visual C++] → [C++ file (.cpp)]. Задаваме **смислено име** на нашата програма, например **HelloC++.cpp**.

Ако забравим да напишем **.cpp** разширението, Visual Studio ще го добави:

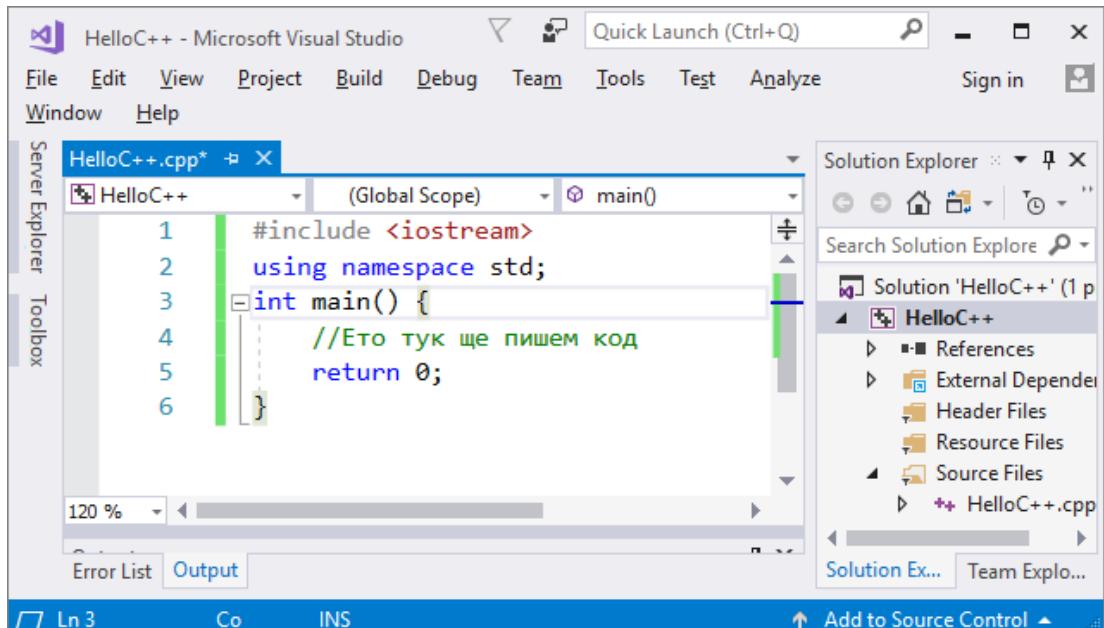


Натискаме бутона [Add] и ще се отвори празна C++ програма:



Писане на програмен код

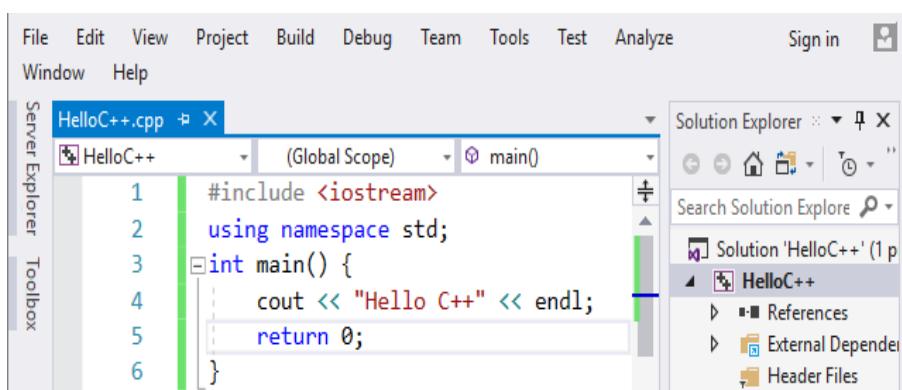
За сега всяка наша програма трябва да започваме със следния код:



Също така за сега ще пишем командите само във функцията **int main()**, между отварящата и затварящата скоба **{ }**. Това е главната функция (действие), което се изпълнява при стартиране на една конзолна C++ програма. По-нататък в книгата, в глава [Функции](#), ще се запознаем по-подборно какво представляват функциите и как можем да ги използваме.

Натискаме [Enter] след **отварящата скоба {** и **започваме да пишем**. Кодът на програмата се пише **отмествен навътре**, като това е част от оформянето на текста, за по-голямо удобство при повторен преглед и/или дебъгване. Пишем следната команда:

```
cout << "Hello C++" << endl;
```



Може да пуснете в действие и тествате кода от горния пример онлайн:

[Ето как трябва да изглежда нашата програма във Visual Studio:](https://repl.it/@vncpetrov>HelloCPP1.</p></div><div data-bbox=)

Командата `cout << "Hello C++" << endl;` означава да изпълним отпечатване върху конзолата и да отпечатаме текстовото съобщение **Hello C++**, което трябва да оградим с кавички, за да поясним, че това е текст. `endl` означава нов ред, а `cout << endl;` означава отпечатване на нов ред. Тези команди могат да се напишат и на два реда:

```
cout << "Hello C++";
cout << endl;
```

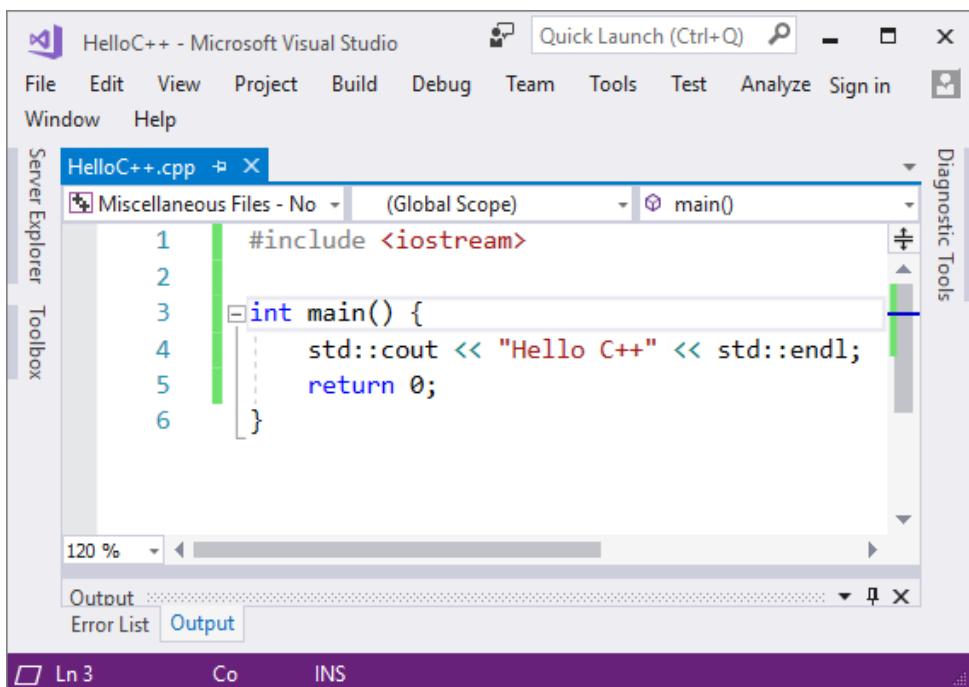
Може да тествате примера онлайн: [В края на всяка команда на езика C++ се слага символът `;`. Той указва, че командата свършва на това място \(т.е. не продължава на следващия ред\).](https://repl.it/@vncpetrov>HelloCPP2.</p></div><div data-bbox=)

Редът `#include<iostream>` означава включи библиотеката `iostream`. Библиотеките са код, който е написан от някой друг и можем да използваме наготово. Например от библиотеката `iostream` използваме `cin` и `cout`. Има най-различни други библиотеки, също така можем и сами да си пишем библиотеки, но това не е предмет на настоящата книга.

Редът `using namespace std;` означава използвай namespace `std`. Какво е namespace? Представете си, че по даден проект работят много хора, сред които Пешо и Гошо. Пешо иска да си кръсти някоя променлива `count`, но и Гошо иска да използва променлива с име `count`. За компилатора тези две променливи ще означават едно и също, независимо дали Пешо или Гошо я използва. За да се реши този проблем, се използва namespace. Тогава Пешо ще си има namespace `Peter`, а Гошо ще си има namespace `Geore`. За да използва Пешо своята променлива `count`, ще напише `Peter::count`, а Гошо, за да използва своята променлива `count`, ще напише съответно `Geore::count`. Тогава вече компилаторът ще прави разлика между променливата `count` на Гошо и съответно на Пешо. Символите `::` означават да се вземе променливата, написана отляво на `::`, от namespace-а, написан отляво на `::`.

Има и готови namespaces, като например `std`. Съкращението `std` идва от `Standard Template Library` и конкретно от думата `standard`. Няколко библиотеки са обединени в namespace `std`. Т.е. за да използваме нещата от тях, трябва да пишем навсякъде `std::....`. Както вече казахме, редът `using namespace std;` означава използвай namespace `std`. Т.е. ако го напишем в програмата си, включваме всички неща от съответния namespace и няма нужда навсякъде да пишем `std::....`. Горният пример без този ред ще изглежда така:

За улеснение, в книгата ще използваме първия вариант с `using namespace std;`.



Стартиране на програмата

За стартиране на програмата натискаме **[Ctrl + F5]**. Ако няма грешки, програмата ще се изпълни. Резултатът ще се изпише на конзолата (в черния прозорец).

Забележете, че стартираме с **[Ctrl + F5]**, а не само с **[F5]** или с бутона за стартиране във Visual Studio. Ако използваме **[F5]**, програмата ще се изпълни за кратко и веднага след това конзолата ще изчезне и няма да видим резултата.

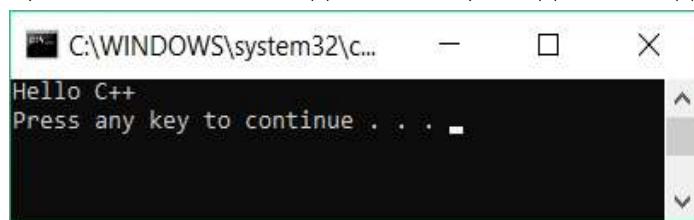
Въщност, изходът от програмата е следното текстово съобщение:

```
Hello C++
```

Съобщението "Press any key to continue ..." се изписва допълнително на най-долния ред на конзолата от Visual Studio след като програмата завърши изпълнението си, за да ни подкани да натиснем клавиш, за да затворим конзолата.

Възможно е **[Ctrl + F5]** да не работи и конзолата да се затваря веднага. За да оправим проблема, трябва да добавим следния ред код на реда **преди return 0;:**

```
system("pause");
```



Кодът ни ще изглежда така:

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello C++" << endl;
    system("pause");
    return 0;
}
```

Ако имаме този проблем, няма значение дали стартираме с [Ctrl + F5] или с [F5].

Тестване на програмата в Judge системата

Тестването на задачите от тази книга е автоматизирано и се осъществява през Интернет, от сайта на Judge системата: <https://judge.softuni.bg>. Оценяването на задачите се извършва на момента от системата. Всяка задача минава поредица от тестове, като всеки успешно преминат тест дава предвидените за него точки. Тестовете, които се подават на задачите, са скрити.

```
#include<iostream>
using namespace std;
int main()
{
    cout << "Hello C++" << endl;
    return 0;
}
```

Allowed working time: 0.100 sec.
 Allowed memory: 16.00 MB
 Size limit: 16.00 KB
 Checker: Trim

C++ code Submit

Горната програма може да се тества тук:

<https://judge.softuni.bg/Contests/Practice/Index/1357#0>.

Поставяме целия сорс код на програмата в черното поле и избираме **C++ code**, както е показано на изображението на предходната страница.

Изпращаме решението за оценяване с бутона [**Изпрати**] (или [**Submit**]). Системата връща резултат след няколко секунди в таблицата с изпратени решения. При необходимост може да натиснем бутона за обновяване на резултатите [**Refresh**], който се намира в горната дясна част на таблицата с изпратени за проверка решения:

Submissions		
1	Refresh	
Points	Time and memory used	Submission date
✓ 100 / 100	Memory: 1.75 MB Time: 0.015 s	16:02:28 21.01.2019
✗ 0 / 100	Memory: 1.75 MB Time: 0.000 s	16:00:21 21.01.2019

В таблицата с изпратените решения Judge система ще покаже един от следните **възможни резултати**:

- Брой точки (между 0 и 100), когато предаденият код се компилира успешно (няма синтактични грешки) и може да бъде тестван.
 - При **вярно решение** всички тестове са маркирани в зелено и получаваме **100 точки**.
 - При **грешно решение** някои от тестовете са маркирани в червено и получаваме непълен брой точки или 0 точки.
- При грешна програма ще получим **съобщение за грешка** по време на компилация.

Как да се регистрирам в SoftUni Judge?

Използваме идентификацията си (Username + Password) от сайта softuni.bg. Ако нямате СофтУни регистрация, направете си. Отнема само минутка - стандартна регистрация в Интернет сайт.

Тествайте примерните програми

Сега, след като вече знаете как да изпълнявате програми, можете да тествате примерните програми по-горе. Позабавявайте се, пробвайте да ги промените и да си поиграете с тях.

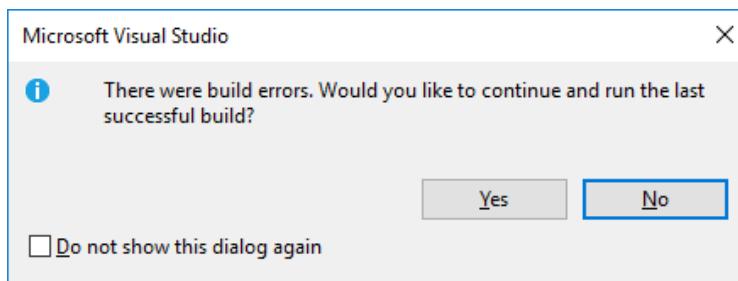
Типични грешки в C++ програмите

Липсата на точка и запетая (;) в края на командите е един от вечните проблеми на начинаещия програмист. Пропускането на този знак води до **неправилно функциониране на програмата** или до **нейното неизпълнение** и често проблемът остава незабелязан. Ето примерен грешен код:

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

int main() {
    int x;
    cin >> x
    cout << x;
}
```

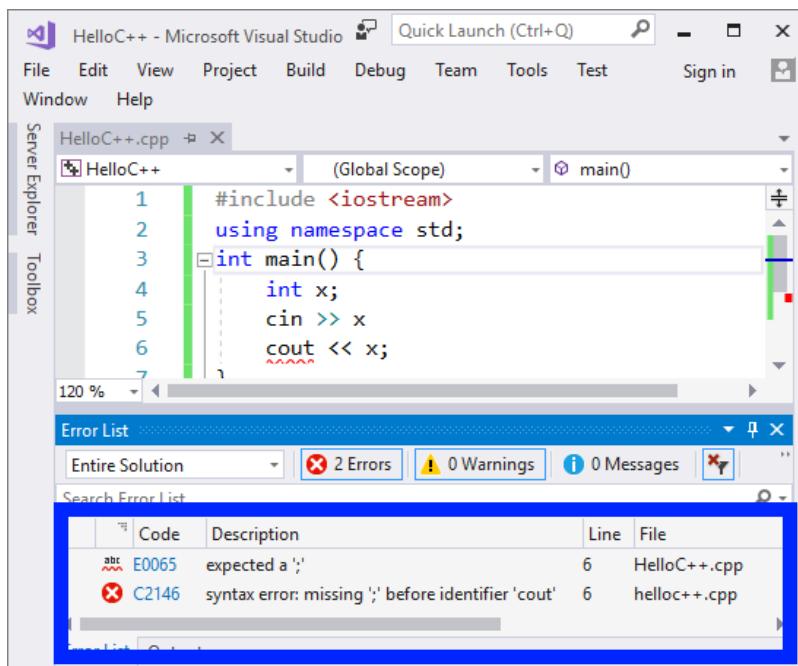
При опит за компилиация, във Visual Studio ще се появи прозорец, който ни казва,



че има проблем.

Ако натиснем [Yes], ще се изпълни последната версия на програмата, която се е компилирала успешно. Ако обаче изберем [No], ще можем да видим

списъка с грешки, който се намира в прозорец **Error List**, под кода на програмата:

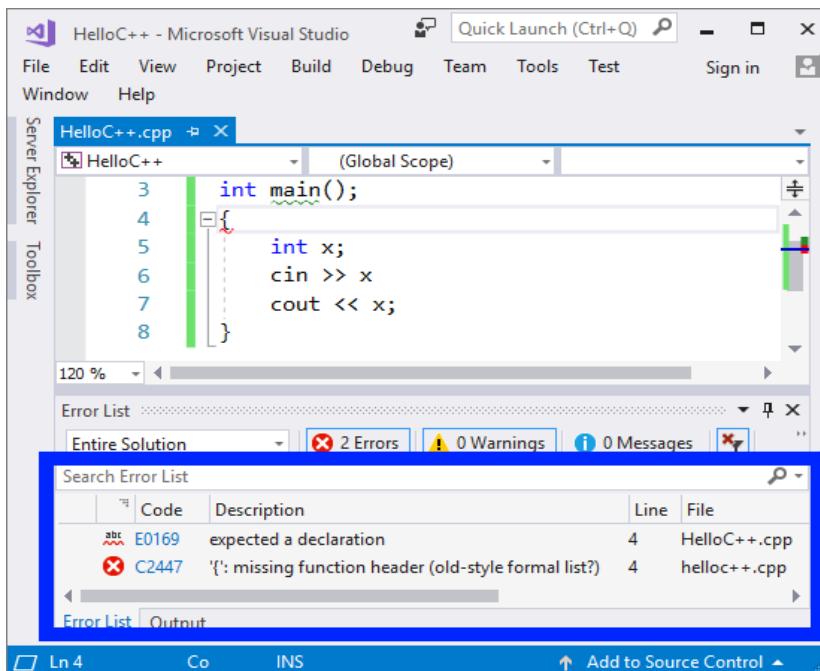


Допълнителна точка и запетая (;) също е проблем. Това също води до **неправилно функциониране на програмата** или до **нейното неизпълнение**. Пример за грешен код:

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

int main() ;
{
    int x;
    cin >> x
    cout << x;
}
```

Във Visual Studio ще се появи следната грешка:

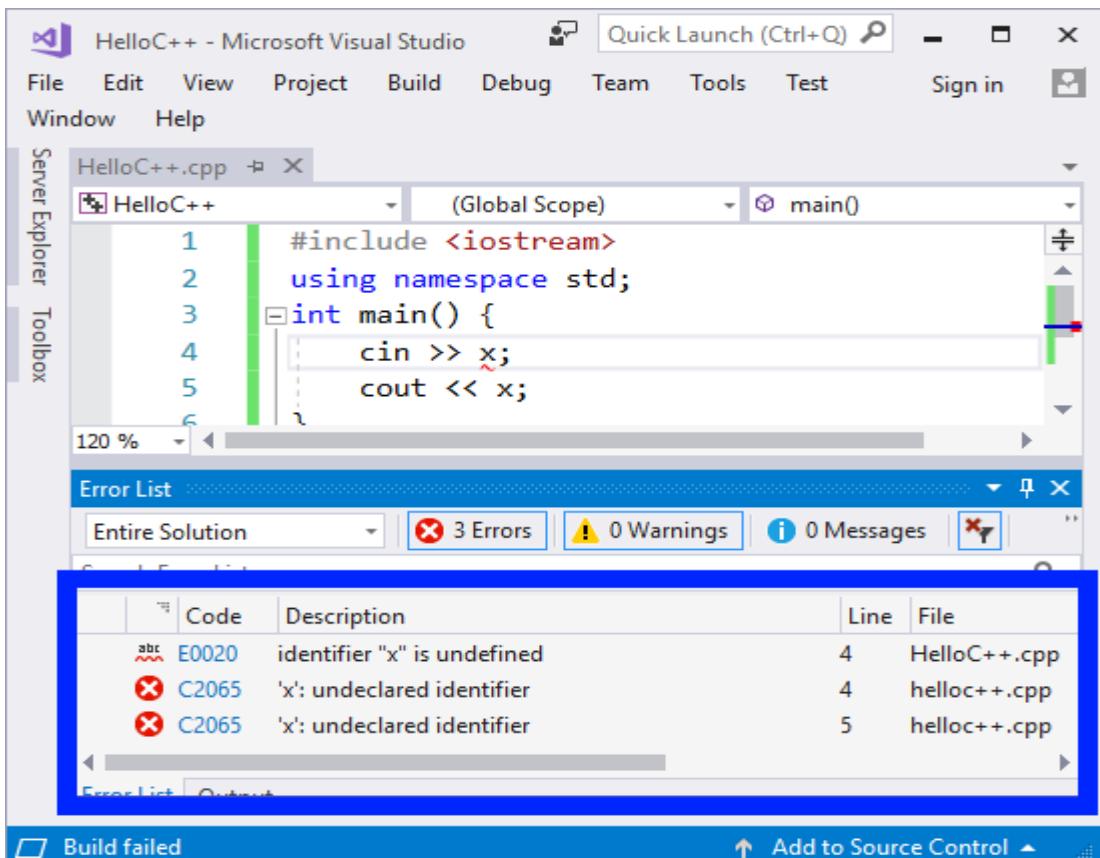


Недекларирани променливи. В следващия пример компилаторът не знае какво е **x**, защото първо трябва да се зададе тип на променливата. Програмата отново **няма да се изпълни**:

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

int main() {
    cin >> x;
    cout << x;
}
```

Във Visual Studio ще се появи следната грешка:



Липсваща кавичка или липса на отваряща или затваряща скоба (неправилен брой скоби) също може да се окажат проблеми. Както и при точката и запетаята, така и тук проблемът води до **неправилно функциониране на програмата** или въобще до **нейното неизпълнение**. Този пропуск трудно се забелязва при по-обемен код.

Ето пример за **грешна** програма:

```

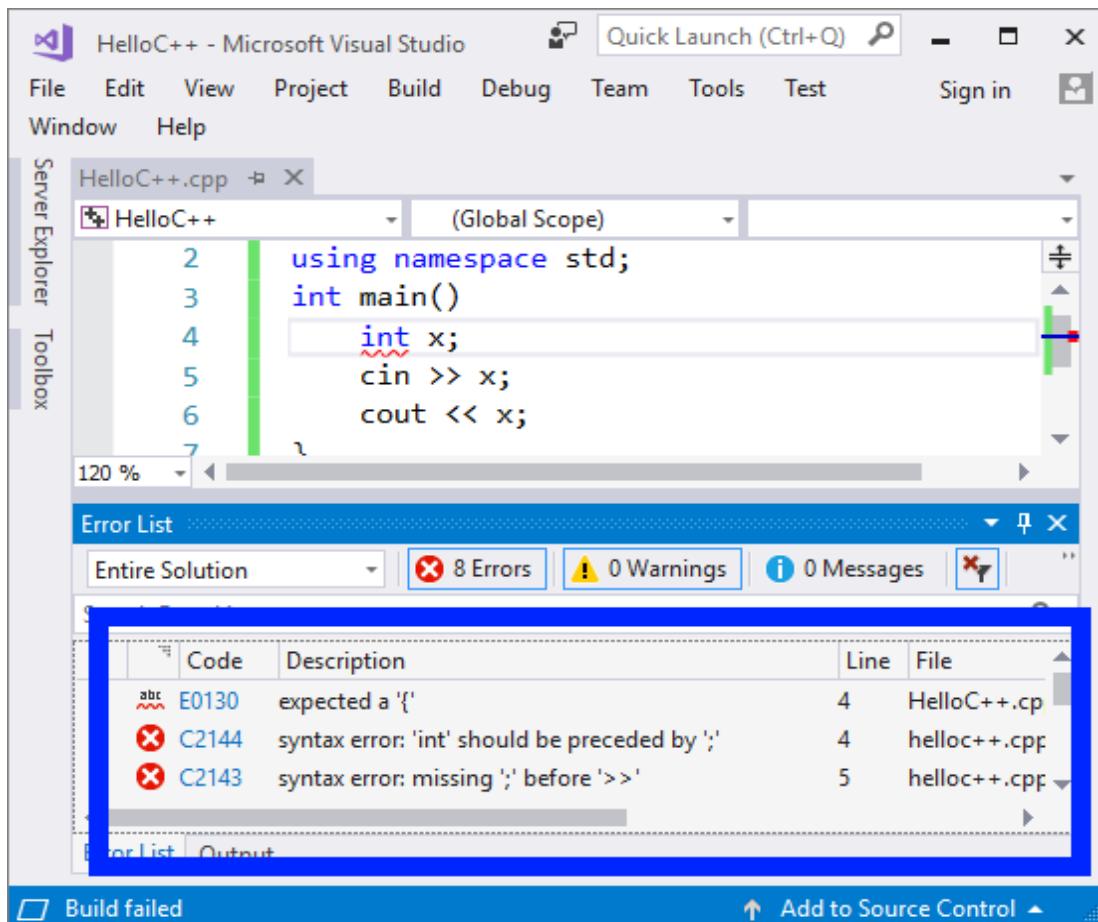
#include <iostream>
using namespace std;

int main()
    int x;
    cin >> x;
    cout << x;
}

```

Тази програма ще даде **грешка при опит за компилиация** и стартиране и дори още преди това кодът ще бъде подчертан, за да се насочи вниманието на програмиста

към грешката, която е допуснал (пропуснатата отваряща скоба). Тук списъкът от грешки ще е по-дълъг, което може да е малко объркващо в началото:



Какво научихме от тази глава?

На първо място научихме какво е програмирането - задаване на команди, изписани на компютърен език, които машината разбира и може да изпълни. Разбрахме още какво е **компютърна програма** - тя представлява **поредица от команди**, подредени една след друга. Запознахме се с **езика за програмиране C++** на базисно ниво и как да създаваме прости конзолни програми с Visual Studio. Проследихме и структурата на програмния код в езика C++, като например, че за сега ще задаваме командите в секцията **int main()** между **отварящата и затварящата къдрава скоба**. Видяхме как да печатаме с **cout << ... ;** и как да стартираме програмата си с **[Ctrl + F5]**. Научихме се да тестваме кода си в **SoftUni Judge** системата и се запознахме с типични грешки в C++ програмите.

Добра работа! Да се захващаме с **упражненията**. Нали не сте забравили, че програмиране се учи с много писане на код и решаване на задачи? Да решим няколко задачи, за да затвърдим наученото.

Упражнения: първи стъпки в коденето

Добре дошли в упражненията. Сега ще напишем няколко конзолни програми, с които ще направим още няколко първи стъпки в програмирането.

Задача: програма "Expression"

Да се напише C++ програма, която **пресмята и отпечатва** стойността на следния числен израз:

$$(3522 + 52353) * 23 - (2336 * 501 + 23432 - 6743) * 3$$

Забележка: **не е разрешено да се пресметне стойността предварително** (например с Windows Calculator).

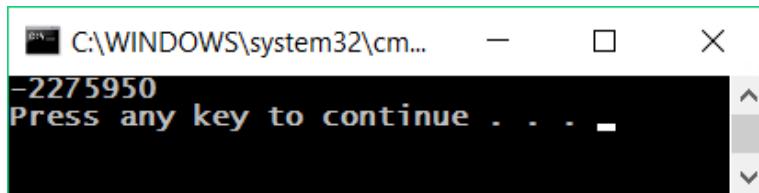
Насоки и подсказки

Правим нов C++ празен проект с име "Expression". Написваме си първоначалния код, **влизаме в тялото на int main()** между **{** и **}**. След това трябва да **напишем кода**, който да изчисли горния числен израз и да отпечатана конзолата стойността му. Пишем горния числов израз на мястото на многоточието в следната команда:

cout << ... ;

Резултатът, който програмата трябва да изведе програмата:

Стартираме предложената програма с [Ctrl+F5] и проверяваме дали резултатът е същият като на картинката:



Задача: числата от 1 до 20

Да се напише C++ програма, която **отпечатва** числата от 1 до 20 на отделни редове на конзолата.

Насоки и подсказки

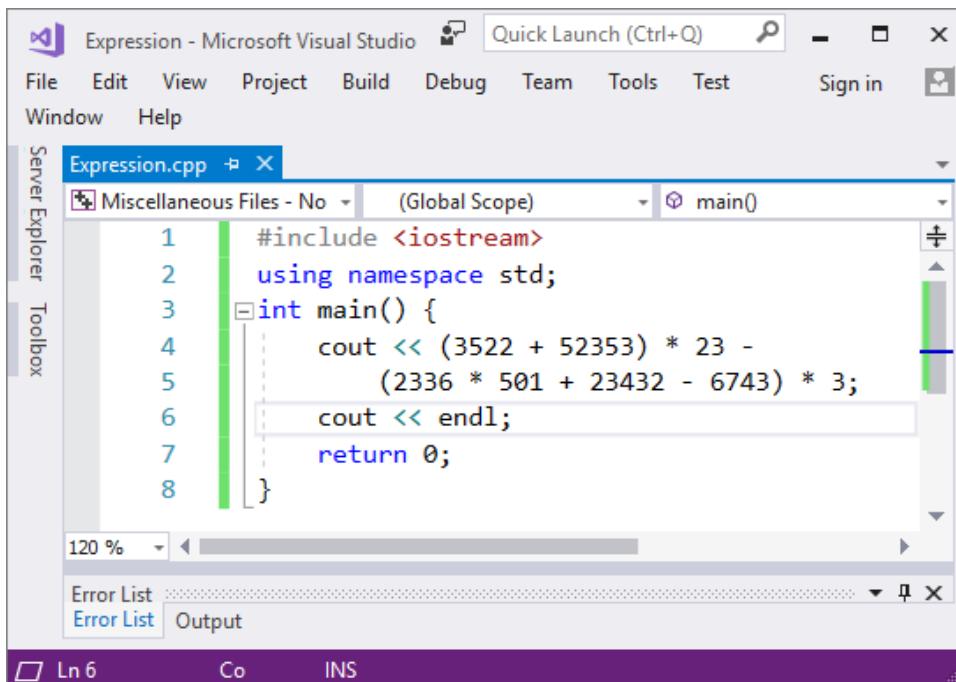
Създаваме нов C++ празен проект с име "Nums1To20". В **int main()** функцията пишем 20 команди **cout << ... << endl;**, всяка на отделен ред, за да отпечатаме числата от 1 до 20 едно след друго:

```
cout << 1 << endl;
cout << 2 << endl;
...

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1357#2>.



Сега **стартираме програмата** и поверьваме дали резултатът е какъвто се очаква да бъде:

```

1
2
...
20
```

02. Expression

```

1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     cout << (3522 + 52353) * 23 -
6         (2336 * 501 + 23432 - 6743) * 3;
7     cout << endl;
8     return 0;
9 }
```

Allowed working time: 0.100 sec.
 Allowed memory: 16.00 MB
 Size limit: 16.00 KB
 Checker: Numbers Checker

[C++ code](#) [Submit](#)

Submissions		
1		
Points	Time and memory used	Submission date
✓ 100 / 100	Memory: 1.77 MB Time: 0.000 s	16:55:47 21.01.2019 Details

Сега помислете дали може да напишем програмата по **по-умен начин**, така че да не повтаряме 20 пъти една и съща команда. Потърсете в Интернет информация за "[for loop C++](#)".

Задача: триъгълник от 55 звездички

Да се напише C++ програма, която отпечатва триъгълник от 55 звездички, разположени на 10 реда:

```
*  
**  
***  
****  
*****  
*****  
*****  
*****  
*****  
*****
```

Насоки и подсказки

Създаваме нов C++ празен проект с име "TriangleOf55Stars". В него трябва да напишем код, който печата триъгълника от звездички, например чрез 10 команди, като посочените по-долу:

```
cout << "*" << endl;  
cout << "**" << endl;  
...  
...
```

Може да тествате примера онлайн:<https://repl.it/@vncpetrov/TriangleOf55Stars>.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1357#3>.

Опитайте да подобрите решението, така че да няма много повтарящи се команди. Може ли това да стане с **for** цикъл? Успяхте ли да намерите умно решение (например с цикъл) на предната задача? При тази задача може да се ползва нещо подобно, но малко по-сложно (два цикъла един в друг). Ако не успеете, няма проблем, ще учим цикли след няколко глави и ще си спомните за тази задача тогава.

Задача: лице на правоъгълник

Да се напише C++ програма, която прочита от конзолата **две числа a и b**, пресмята и отпечатва лицето на правоъгълник със страни **a** и **b**.

Примерен вход и изход

a	b	area
2	7	14

a	b	area
12	5	60

a	b	area
7	8	56

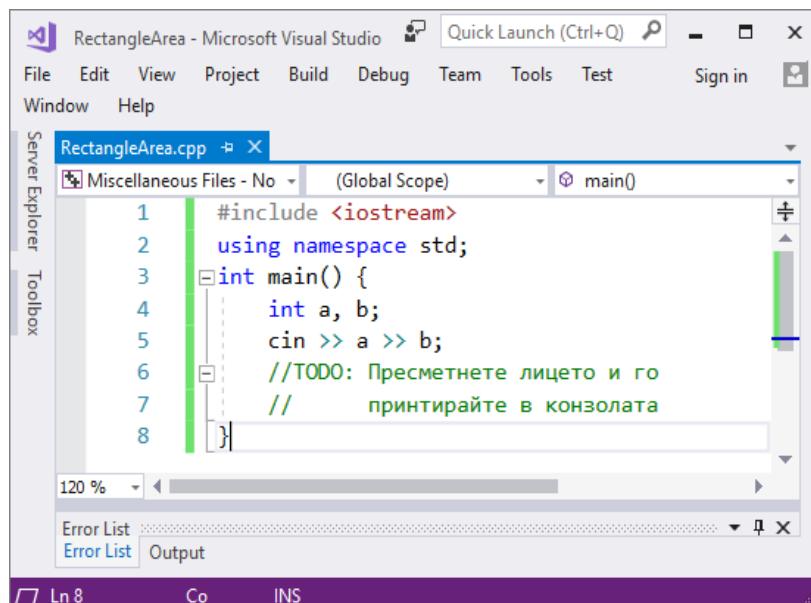
Насоки и подсказки

Правим нов C++ празен проект. За да прочетем двете числа, използваме следната команда команди:

```
int a, b;
cin >> a >> b;
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/RectangleArea>.

Остава да се допише програмата по-горе, за да пресмята лицето на правоъгълника и да го отпечатат. Използвайте познатата ни вече команда **cout << ...**; и на мястото на многоточието напишете произведението на числата **a** и **b**. В програмирането умножението се извършва с оператора *****.



Тествайте решението си

Тествайте решението си с няколко примера. Трябва да получите резултат, подобен на този (въвеждаме 2 и 7 като вход и програмата отпечатва като резултат 14 - тяхното произведение):

```
2
7
14
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1357#4>.

* Задача: квадрат от звездички

Да се напише C++ програма, която прочита от конзолата **цяло положително число N** и отпечатва на конзолата **квадрат от N звездички**, като в примерите по-долу.

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
3	*** * * ***	4	***** * * * * *****	5	***** * * * * * * *****

Насоки и подсказки

Правим нов C++ празен проект. За да прочетем числото **N** ($2 \leq N \leq 100$), използваме следния код:

```
int n;
cin >> n;
```

Може да пуснете в действие и тествате кода от горния пример онлайн:

<https://repl.it/@vncpetrov/SquareOfStars>.

Да се допише програмата по-горе, за да отпечатва квадрат, съставен от звездички. Може да се наложи да се използват **for** цикли. Потърсете информация в Интернет.

Внимание: тази задача е по-трудна от останалите и нарочно е дадена сега и е обозначена със звездичка, за да ви провокира да потърсите информация в Интернет. Това е едно от най-важните умения, което трябва да развивате докато учите програмирането: **да търсите информация в Интернет**. Това ще правите

всеки ден, ако работите като програмисти, така че не се плашете, а се опитайте. Ако имате трудности, можете да потърсите помощ и в СофтУни форума: <https://softuni.bg/forum>.

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1357#5>.

Конзолни, графични и уеб приложения

При конзолните приложения (Console Applications), както и сами можете да се доссетите, всички операции за четене на вход и печтане на изход се извършват през конзолата. Там се въвеждат входните данни, които се прочитат от приложението, там се отпечатват и изходните данни след или по време на изпълнение на програмата.

Докато конзолните приложения ползват текстовата конзола, уеб приложенията (Web Applications) използват уеб-базиран потребителски интерфейс. За да се постигне тяхното изпълнение са необходими две неща - уеб сървър и уеб браузър, като браузърът играе главната роля по визуализация на данните и взаимодействието с потребителя. Уеб приложенията са много по-приятни за потребителя, изглеждат визуално много по-добре, използват се мишка и докосване с пръст (при таблети и телефони), но зад всичко това стои програмирането. И затова трябва да се научим да програмираме и вече направихме първите си съвсем малки стъпки.

Графичните (GUI) приложения имат визуален потребителски интерфейс, директно върху вашия компютър или мобилно устройство, без да е необходим уеб браузър. Графичните приложения (настолни приложения или, иначе казано, desktop apps) се състоят от един или повече графични прозореца, в които се намират определени контроли (текстови полета, бутони, картички, таблици и други), служещи за диалог с потребителя по по-интуитивен начин. Подобни са и мобилните приложения във вашия телефон и таблет: ползваме форми, текстови полета, бутони и други контроли и ги управляваме чрез програмен код. Нали затова се учим сега да пишем код: **кодът е навсякъде в разработката на софтуер**.

Страшно ли изглежда? **Не се плашете!** Имаме да извървим дълъг път, за да достигнем ниво на знания и умения, за да пишем свободно код на C++ в много по-големи и по-сложни програми. Ако не успеете да се справите, няма страшно, продължете спокойно напред. След време ще си спомняте с усмивка колко непонятен и вълнуващ е бил първият ви сблъсък с програмирането. Ако имате проблеми с примерите по-горе, **гледайте видеото** в началото на тази глава или питайте във форума на СофтУни: <https://softuni.bg/forum>. Имате да учене още много.

Глава 2.1. Прости пресмятания с числа

В настоящата глава ще се запознаем със следните концепции и програмни техники:

- Какво представлява **системната конзола**?
- Как да **прочитаме числа** от системната конзола?
- Как да работим с **типове данни и променливи**, които са ни необходими при обработка на числа и операциите между тях?
- Как да **изведем** резултат (число) на системната конзола?
- Как да извършваме прости **аритметични операции**: събиране, изваждане, умножение, деление, съединяване на низ?

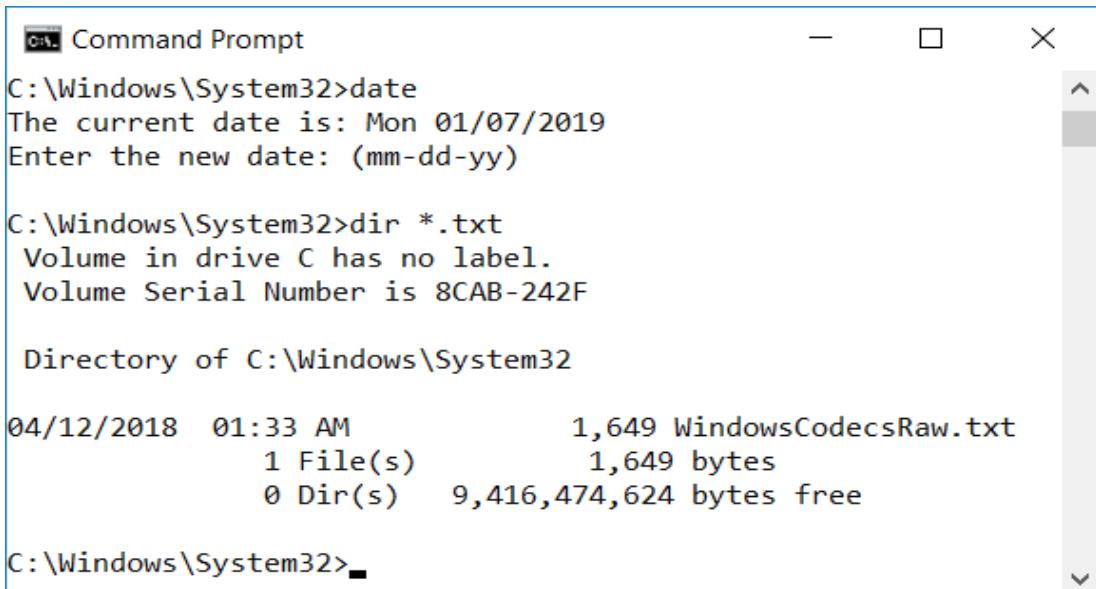
Видео

Гледайте видео урок по учебния материал от настоящата глава от книгата: <https://www.youtube.com/watch?v=QCqD7btQMbl>.

Системна конзола

Обикновено наричана само "конзола", системната или още компютърната конзола, представлява устройството, чрез което подаваме команди на компютъра в текстов вид и получаваме резултатите от тяхното изпълнение отново като текст.

В повечето случаи системната конзола представлява текстов терминал, т.е. приема и визуализира само **текст**, без графични елементи като например бутони, менюта и т.н. Обикновено изглежда като прозорец с черен/бял цвят като този:



```
Command Prompt
C:\Windows\System32>date
The current date is: Mon 01/07/2019
Enter the new date: (mm-dd-yy)

C:\Windows\System32>dir *.txt
Volume in drive C has no label.
Volume Serial Number is 8CAB-242F

Directory of C:\Windows\System32

04/12/2018  01:33 AM           1,649 WindowsCodecsRaw.txt
               1 File(s)        1,649 bytes
               0 Dir(s)  9,416,474,624 bytes free

C:\Windows\System32>
```

В повечето операционни системи **конзолата** е достъпна като самостоятелно приложение на което пишем конзолни команди. В Windows се нарича **Command Prompt**, а в Linux и Mac се нарича **Terminal**. В конзолата се изпълняват конзолни приложения. Те четат текстов вход от командния ред и печатат изхода си като текстов изход на конзолата. В настоящата книга ще се учат на програмиране като създаваме предимно **конзолни приложения**. Не забравяйте да настроите конзолата (от [Project] -> [Properties] -> [Linker] -> [System] -> Console(/SUBSYSTEM:CONSOLE)).

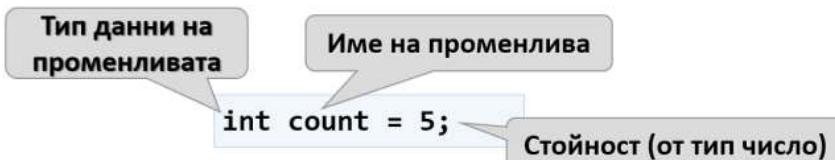
В C++, за въвеждане на данни се използва **cin**, а за отпечатването им - **cout**. Това са обекти, дефинирани в **namespace std** (пространство от имена). За улеснение, пространството може да бъде добавено към нашия проект. В противен случай, както вече научихме в предходната глава, всеки път преди използването на **cin** и **cout** (както и други обекти, дефинирани в **std**) е нужно да се изписва **std::**, т.е.:

```
std::cin >> name;  
std::cout << name;
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/CinCout>.

Пресмятания в програмирането

За компютрите знаем, че са машини, които обработват данни. Всички **данни** се записват в компютърната памет (RAM памет) в **променливи**. Променливите са именувани области от паметта, които пазят данни от определен тип, например число или текст. Всяка една **променлива** в C++ има **име**, **тип** и **стойност**. Ето как бихме дефинирали една променлива, като едновременно с декларацията ѝ, ѝ присвояваме и стойност:



След тяхната обработка, данните се записват отново в променливи (т.е. някъде в паметта, заделена от нашата програма).

Типове данни и променливи

В програмирането всяка една променлива съхранява определена **стойност** от даден **тип**. Типовете данни могат да бъдат например: **число**, **буква**, **текст** (стринг), **дата**, **цвят**, **картинка**, **списък** и др.

Ето няколко примера за типове данни:

- тип **цяло число**: 1, 2, 3, 4, 5, ...
- тип **дробно число**: 0.5, 3.14, -1.5, ...

- тип **буква от азбуката** (символ): 'a', 'b', 'c', ...
- тип **текст** (стринг): "Здрави", "Hi", "Beer", ...

C++ е **силно типизиран език**, тоест типът на стойността винаги трябва да бъде обозначен (2 е цяло число, 0.5 е дробно число, "Здрави" е стринг).

В следващите примери и задачи ще разгледаме следните типове данни:

- **int** - за цели числа,
- **double** - за дробни числа,
- **string** - за текст.

Четене на числа от конзолата

За да прочетем **цяло** (не дробно) **число** от конзолата е необходимо да **декларираме променлива**, да посочим **типа на числото**, както и да използваме **>>** (оператор за побитово изместване надясно) за прочитане на информация **от** системната конзола:

```
int num;
cin >> num;
```

Може да пуснете в действие и тествате кода от горния пример онлайн:

<https://repl.it/@vncpetrov/ReadIntNumber>.

Пример: пресмятане на лице на квадрат със страна a

За пример да вземем следната програма, която прочита цяло число от конзолата, умножава го по него самото (вдига го на квадрат) и отпечатва резултата от умножението. Така можем да пресметнем лицето на квадрат по дадена дължина на страната:

```
cout << "a = ";
int a;
cin >> a;

int area = a * a;
cout << "Square = ";
cout << area << endl;
```

Може да пуснете в действие и тествате кода от горния пример онлайн:

<https://repl.it/@vncpetrov/SquareArea>.

Следващата картичка показва как би работила програмата при квадрат с размер на страната 3:

```
C:\WINDOWS\system32\cmd...
a = 3
Square area = 9
Press any key to continue . . .
```

Опитайте да въведете грешно число, например "hello".

Как работи примерът?

На първия ред, с **int a**, е декларирана променливата от **тип int** (т.е. цяло число).

На втория ред, чрез **cout << "a = "**; се печата информативно съобщение, което подканва потребителя да въведе страната на квадрата **a**. За печатането на информация **на** системната конзола се използва **<<** (побитово изместване наляво). След отпечатването курсорът остава на същия ред. Оставането на същия ред е по-удобно за потребителя, чисто визуално. Използва се **cout << "...";**, а не **cout << "..." << endl;** и така курсорът остава на същия ред (**endl** също е част от **namespace std** за преминаване на нов ред може да се използва и "**\n**"): **cout << area << "\n";**.

На третия ред **cin >> a;** се прочита цяло число от конзолата и запазва във вече дефинираната променлива **a**.

На четвъртия ред **int area = a * a;** се създава нова променлива **area**, в която резултатът от умножението на **a** по **a** се записва.

На петия ред **cout << "Square area = "**; се отпечатва посочения текст, без да се преминава на нов ред. Отново се използва **cout << "...";**, а не **cout << "..." << endl;** и така курсорът остава на същия ред, за да може след това да се отпечата и изчисленото лице на квадрата.

На шестия ред, последната команда **cout << area;** отпечатва стойността на променливата **area**.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1358#0>.

Четене на дробно число от конзолата

За да прочетем **дробно число** от конзолата е необходимо отново да **декларираме променлива**, да посочим **типа на числото**, както и да използваме **>>** (оператор за побитово изместване надясно):

```
double num;
cin >> num;
```

Може да пуснете в действие и тествате кода от горния пример онлайн:

<https://repl.it/@vncpetrov/ReadDoubleNumber>.

Пример: прехвърляне от инчове в сантиметри

Да напишем програма, която чете дробно число в инчове и го обръща в сантиметри:

```
cout << "Inches = ";
double inches;
cin >> inches;
double centimeters = inches * 2.54;
cout << "Centimeters = ";
cout << centimeters << endl;
```

Може да пуснете в действие и тествате кода от горния пример онлайн:

<https://repl.it/@vncpetrov/InchesToCentimeters>.

Да стартираме програмата и да се уверим, че при подаване на стойност в инчове, получаваме коректен резултат в сантиметри:

```
C:\WINDOWS\system32\c...
a = 5
Square area = 25
Press any key to continue . . .
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1358#1>.

Четене на текст

При работа с текст, библиотеката **<string>** е нужно да бъде включена. За прочитане на текст от системната конзола си **декларираме нова променлива** (от тип стринг) и използваме **>>** (оператор за побитово изместване надясно):

```
#include <iostream>
#include <string>

using namespace std;

int main {
    string str;
    cin >> str;

    return 0;
}
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/ReadText>.

Пример: поздрав по име

Да напишем програма, която въвежда името на потребителя и го поздравява с текста "Hello, *име*".

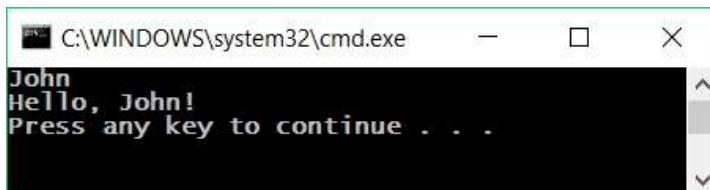
```
string name;
getline(cin, name);
cout << "Hello, " << name << "!\n";
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/GreetingByName>.

В този пример е използван друг начин за прочитане на входните данни, тъй като не се знае колко думи ще бъдат въведени за име. Поради тази причина, в C++ съществува функцията **getline(...)**, с чиято помощ се извличат всички символи, написани на един ред (включително и интервалите). В глава "[Функции](#)"** ще се запознаем по-подробно с функциите, но сега нека разгледаме какво трябва да запишем в дадената.

На първо място в скобите поставяме обекта **cin**, който "взима" входните данни и ги запаметява в променлива. За да укажем точно в коя променлива, на второ място изписваме нейното име (в този случай - **name**).

Както се забелязва, на последния ред стрингът "Hello, " е преди променливата **name**, а другият стринг "!" - след нея. Всички те ще бъдат отпечатани в **тази последователност**, благодарение на **<<** (операторът за побитово изместване наляво), който ги **свързва** в единен текст:



Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1358#2>.

Печатане на текст и числа

```
string firstName, lastName, town;
int age;
cin >> firstName;
cin >> lastName;
cin >> age;
cin >> town;
cout << "You are " << firstName << " " << lastName <<
    ", a " << age << "-years old person from " << town << "." <<
    endl;
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/ConcatenateData>.

Ето резултатът, който ще получим, след изпълнение на този пример:

```
C:\WINDOWS\system32\cmd.exe
Ivan
Ivanov
30
Plovdiv
You are Ivan Ivanov, a 30-years old person from Plovdiv.
Press any key to continue . . .
```

Може да ви направи впечатление, че **firstName**, **lastName**, **town** са написани на един ред. Това е така, защото **когато променливите са от еднакъв тип, те могат се декларират на един ред**.

И в този пример обърнете внимание как всяка една променлива трябва да бъде подадена в **реда, в който искаме да се печата**. Това се отнася и за местата, където се налага да сложим интервал (трябва изрично да укажем кои са те - в случая е използван празен стринг " ", между **firstName** и **lastName**, но съществуват и други варианти).

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1358#3>.

Аритметични операции

Да разгледаме базовите аритметични операции в програмирането.

Събиране на числа (оператор +)

Можем да събираме числа с оператора **+**:

```
int a = 5;
int b = 7;
int sum = a + b; // 12
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/SumOperator>.

Изваждане на числа (оператор -)

Изваждането на числа се извършва с оператора **-**.

Пример:

```
int a, b;
cin >> a;
cin >> b;
```

```
int result = a - b;
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/SubtractionOperator>.

Ето резултатът от изпълнението на програмата (при числа 10 и 3):

```
C:\WINDOWS\system32\cmd...
10
3
7
Press any key to continue . . .
```

Умножение на числа (оператор *)

За умножение на числа използваме оператора *****:

```
int a = 5;
int b = 7;
int product = a * b; // 35
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/MultiplyOperator>.

Деление на числа (оператор /)

Делението на числа се извършва с оператора **/**. Той работи различно при цели и при дробни числа.

- Когато делим две цели числа, се извършва **целочислено деление** и полученият резултат е цяло число с отрязана дробна част. Например $11 / 3 = 3$.
- Когато делим две числа, от които поне едното е дробно, се извършва **дробно деление** и полученият резултат е дробно число, както в математиката. Например $11 / 4.0 = 2.75$. При невъзможност за точно разделяне, резултатът се закръгля, например $11.0 / 3 = 3.66666666666667$.
- Целочисленото **деление на 0** предизвиква **грешка** по време на изпълнение (runtime exception).
- Дробното **деление на 0** не предизвиква грешка, а резултатът е **+/- безкрайност** или специалната стойност **inf**. Например $5 / 0.0 = \infty$.

Ето няколко примера за използване на оператора за делене:

```
int a = 25;
cout << a / 4 << endl;      // 6 - целочислено деление, което дава
// резултат цяло число.
// При делението на цели числа, дробната част се отрязва.
```

```
cout << a / 4.0 << endl; // 6.25 – дробно деление, което дава
// разултат дробно число. Указали сме числото 4 да се интерпретира като
// дробно, като сме добавили десетичната точка, следвана от нула.
// (ако f беше от тип int, дробната част пак щеше да бъде отрязана, а
// резултатът отново 6)
cout << a / 0 << endl; // Грешка: целочислено деление на 0
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/DivisionOperator1>.

Да разгледаме и няколко примера за **целочислено деление** (запомнете, че при **деление на цели числа** в езика C++ резултатът е **цяло число**):

```
int a = 25;
cout << a / 7 << endl; // Целочислен резултат: 3
cout << a / 9 << endl; // Целочислен резултат: 2
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/DivisionOperator2>.

Да разгледаме няколко примера за **деление на дробни числа**. При дробно делене резултатът винаги е **дробно число** и деленето никога не дава грешка и работи коректно със специалните стойности **+∞** и **-∞**:

```
int a = 15;
cout << a / 2.0 << endl; // Дробен резултат: 7.5
cout << a / 0.0 << endl; // Резултат: inf
cout << -a / 0.0 << endl; // Резултат: -inf
cout << 0.0 / 0.0 << endl; // Резултат: -nan(Not a Number), т.е. //
результатът от операцията не е валидна числена стойност
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/DivisionOperator3>.

Съединяване на текст и число

Операторът **+** освен за събиране на числа служи и за съединяване на текст (долепяне на два символни низа един след друг). В програмирането съединяване на текст с текст или с число наричаме **"конкатенация"**. Ето как можем да съединяваме текст и число с оператора **+**. Не забравяйте да добавите библиотеката **<string>**:

```
string firstName = "Maria";
string lastName = "Ivanova";
string age = "19";
string str = firstName + " " + lastName + " @ " + age;
cout << str << endl; // Maria Ivanova @ 19
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/Concatenation>.

Числата също могат да бъдат представяни като текст. Следователно, **конкатенацията** е възможна. Ето още един пример:

```
string a = "1.5";
string b = "2.5";
string concatenation = "The sum is: " + a + b;
cout << concatenation << endl; // The sum is: 1.52.5
```

Може да пуснете в действие и тествате кода от горния пример онлайн:

<https://repl.it/@vncpetrov/Concatenation2>.

Числени изрази

В програмирането можем да пресмятаме и **числови изрази**, например:

```
int expr = (3 + 5) * (4 - 2);
```

Може да пуснете в действие и тествате кода от горния пример онлайн:

<https://repl.it/@vncpetrov/Expr>.

В сила е стандартното правило за приоритетите на аритметичните операции: **умножение и деление се извършват винаги преди събиране и изваждане**. При наличие на **израз** в скоби, той се изчислява пръв, но ние знаем всичко това от училищната математика.

Пример: изчисляване на лице на трапец

Да напишем програма, която въвежда дълчините на двете основи на трапец и неговата височина (по едно дробно число на ред) и пресмята **лицето на трапеца** по стандартната математическа формула:



Ако $(b_1 + b_2) * h$ се дели без остатък на 2.0, десетичната точка и нула ще бъдат пренебрегнати и няма да бъдат изписани на конзолата.

```
#include <iostream>
#include <string>

using namespace std;

int main() {
    string firstName, lastName, town;
    double b1, b2, h;
```

```

    cin >> b1;
    cin >> b2;
    cin >> h;
    double area = (b1 + b2) * h / 2.0;
    cout << "Trapezoid area = " << endl;

    return 0;
}

```

Ако стартираме програмата и въведем за страните съответно **3, 4** и **5**, ще получим следния резултат:

```

3
4
5
Trapezoid area = 17.5

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1358#4>.

Пример: периметър и лице на кръг

Нека напишем програма, която при въвеждане **радиуса r** на кръг **изчислява лицето и периметъра** на кръга / окръжността.

Формули:

$$\text{Лице} = \pi * r * r \quad \text{Периметър} = 2 * \pi * r \quad \pi \approx 3.14159265358979323846\ldots$$

```

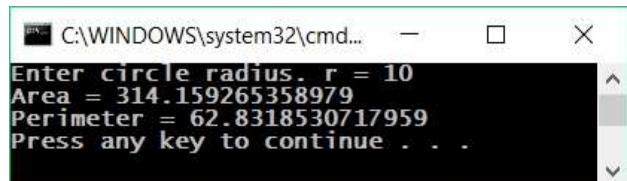
#include <iostream>
using namespace std;

int main() {
    cout << "Enter circle radius. r = ";
    double r;
    cin >> r;
    double pi = 3.14159265359;
    cout << "Area = " << pi * r * r << endl;
    cout << "Perimeter = " << 2 * pi * r << endl;

    return 0;
}

```

Нека изprobваме програмата с радиус **r = 10**:

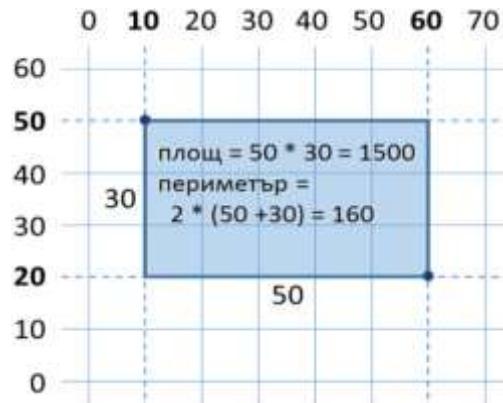


Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1358#5>.

Пример: лице на правоъгълник в равнината

Правоъгълник е зададен с координатите на два от своите два срещуположни ъгла. Да се пресметнат площта и периметъра му:



В тази задача трябва да съобразим, че ако от по-големия **x** извадим по-малкия **x**, ще получим дължината на правоъгълника. Аналогично, ако от по-големия **y** извадим по-малкия **y**, ще получим височината на правоъгълника. Остава да умножим двете страни. Ето примерна имплементация на описаната логика:

```
#include <iostream>
#include <cmath>

using namespace std;

int main() {
    double x1, x2, y1, y2;
    cin >> x1;
    cin >> y1;
    cin >> x2;
    cin >> y2;

    double width = fabs(x2 - x1);
    double height = fabs(y2 - y1);

    cout << "Area = " << width * height << endl;
    cout << "Perimeter = " << 2 * (width + height) << endl;

    return 0;
}
```

Използваме **`fabs(x2 - x1)`** и **`fabs(y2 - y1)`**, за да намерим абсолютната стойност на получената разлика, тъй като не знаем кое измежду двете числа е по-голямо. Тази функция се съдържа в библиотеката **<cmath>**.

При стартиране на програмата със стойностите от координатната система в условието, получаваме следния резултат:

```

60
20
10
50
Area = 1500
Perimeter = 160
Press any key to continue . . .

```

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1358#6>.

Какво научихме от тази глава?

Да резюмираме какво научихме от тази глава на книгата:

- Въвеждане на текст, цяло число и дробно число, чрез `cin: cin >> age;`.
- Извършване на пресмятания с числа и използване на съответните аритметични оператори [+,-,*,/,(]): `int sum = 5 + 3;`.
- Извеждане на единен текст, чрез `cout: cout << "My names is" << firstName << " " << lastName << endl;`.

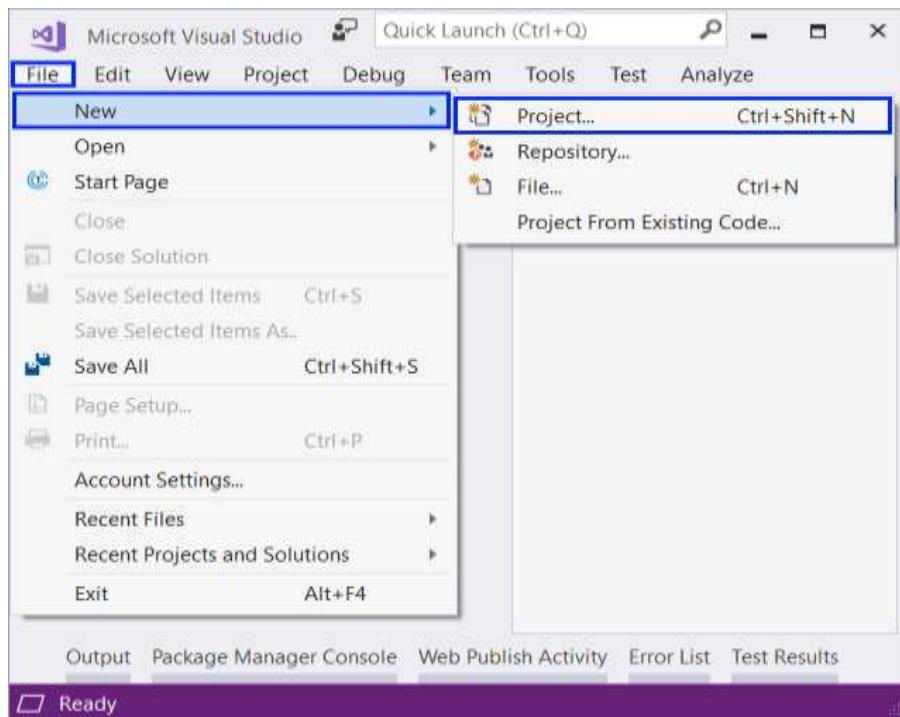
Упражнения: прости пресмятания

Нека затвърдим наученото в тази глава с няколко задачи.

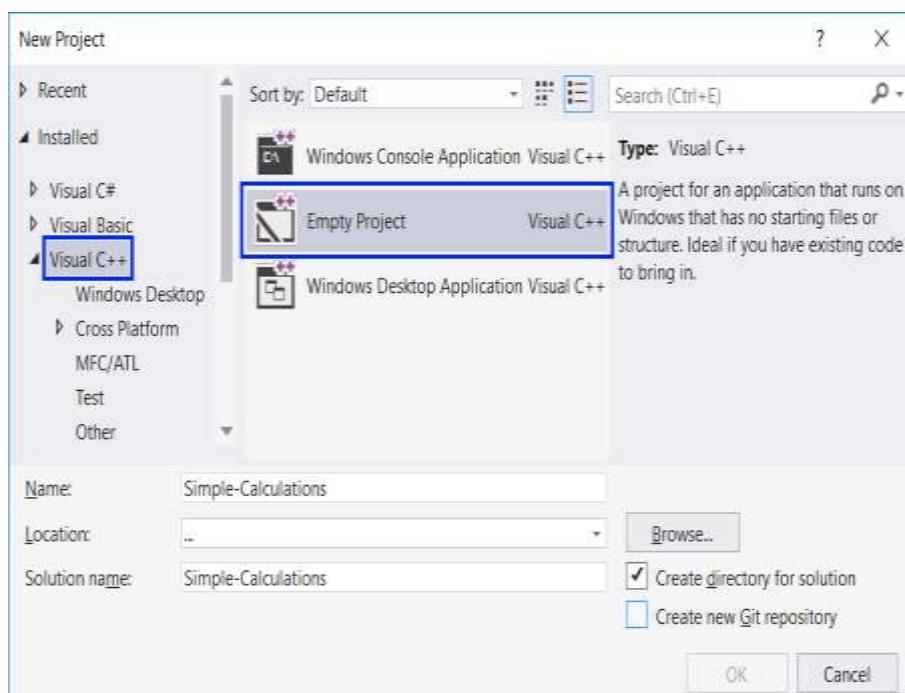
Празно Visual Studio решение (Empty Project)

В настоящото практическо занимание ще използваме **Solution** с няколко проекта, за да организираме решенията на задачите от упражненията, т.е. всяка задача ще бъде в отделен проект, а всички проекти в общ VS Solution. Започваме, като създадем празен проект (**Empty Project**) във Visual Studio:

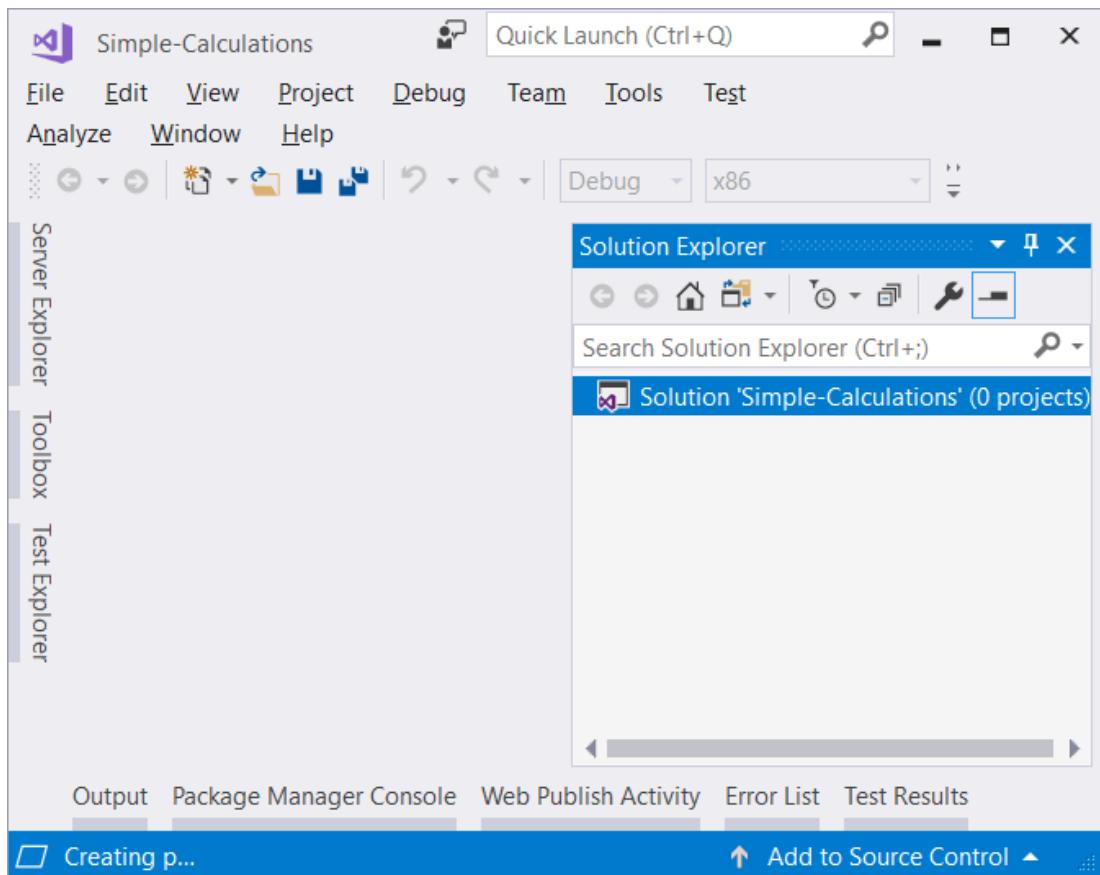
- Стартираме Visual Studio.
- Създаваме нов Empty Project: [File] -> [New] -> [Project...].



Избираме от диалоговия прозорец [Visual C++] -> [Empty Project] и даваме подходящо име на проекта, например "Simple-Calculations":



Сега имаме създаден празен Visual Studio Solution (с 0 проекта в него):



Задача: пресмятане на лице на квадрат

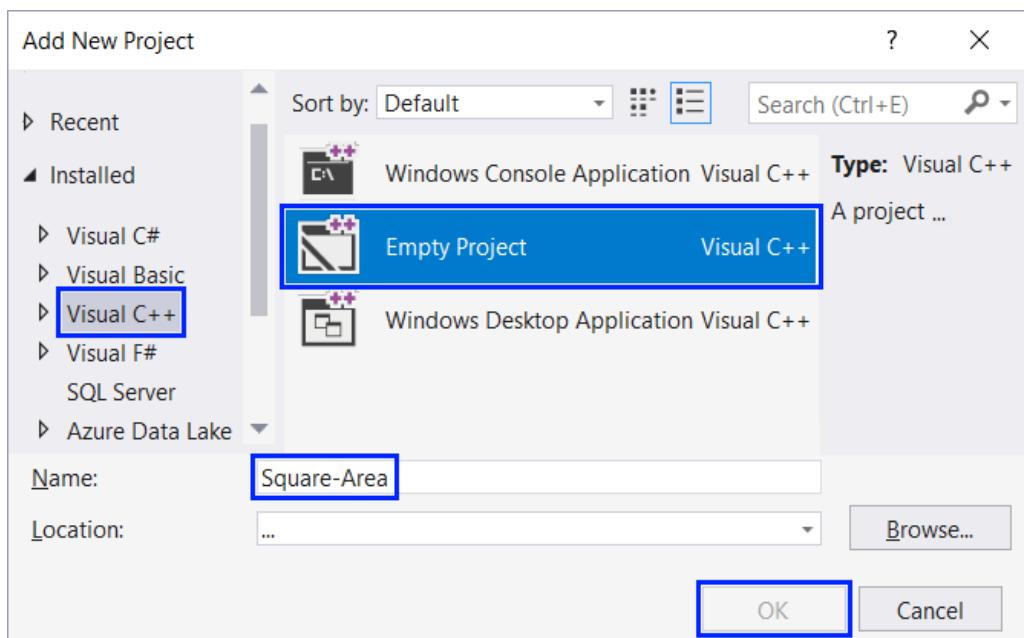
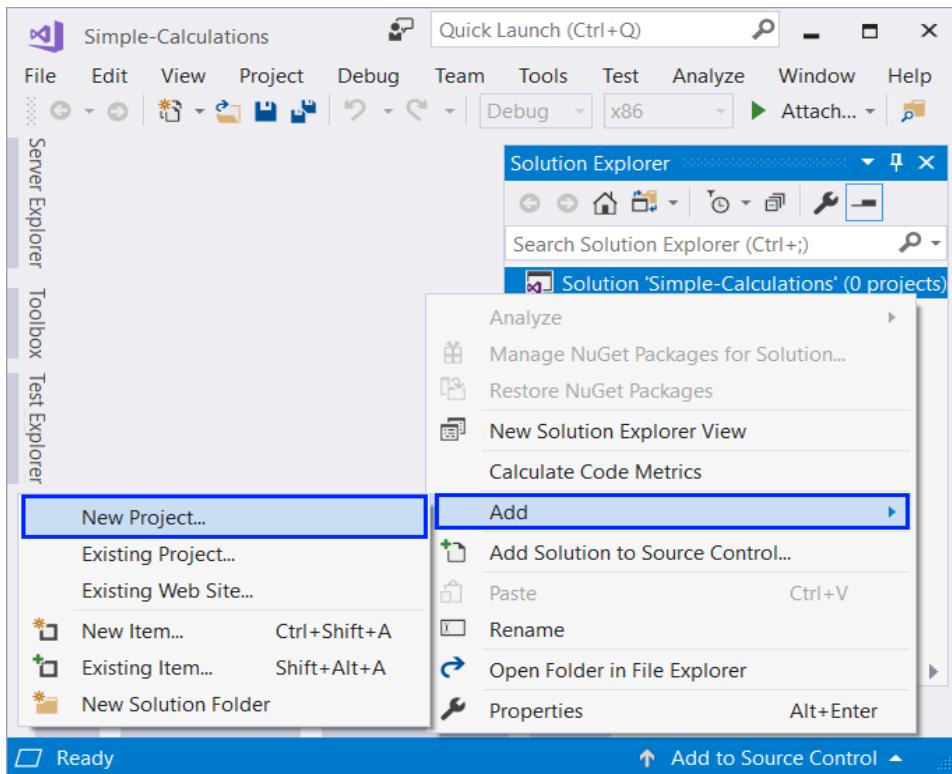
Първата задача от тази тема е следната: да се напише конзолна програма, която **въвежда цяло число а и пресмята лицето на квадрат със страна а**. Задачата е тривиална: прочитаме число от конзолата, умножаваме го само по себе си и отпечатваме получения резултат на конзолата.

Насоки и подсказки

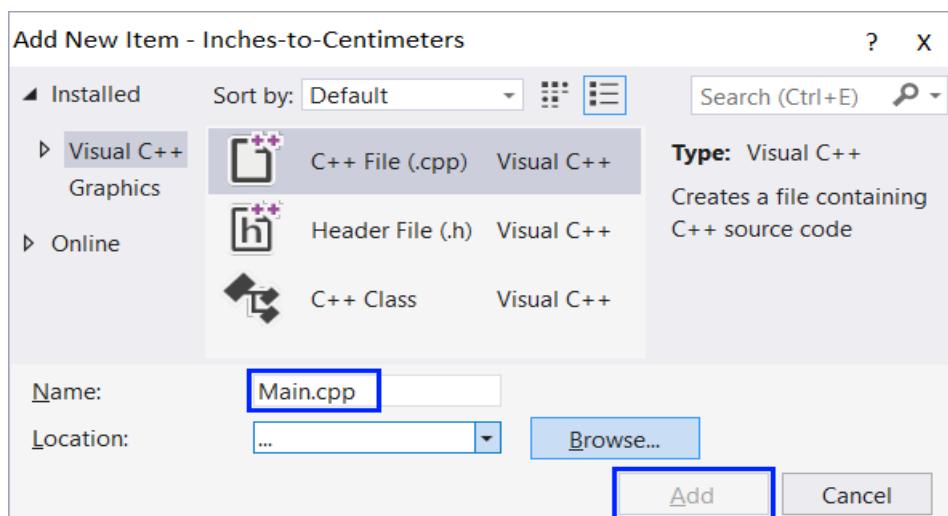
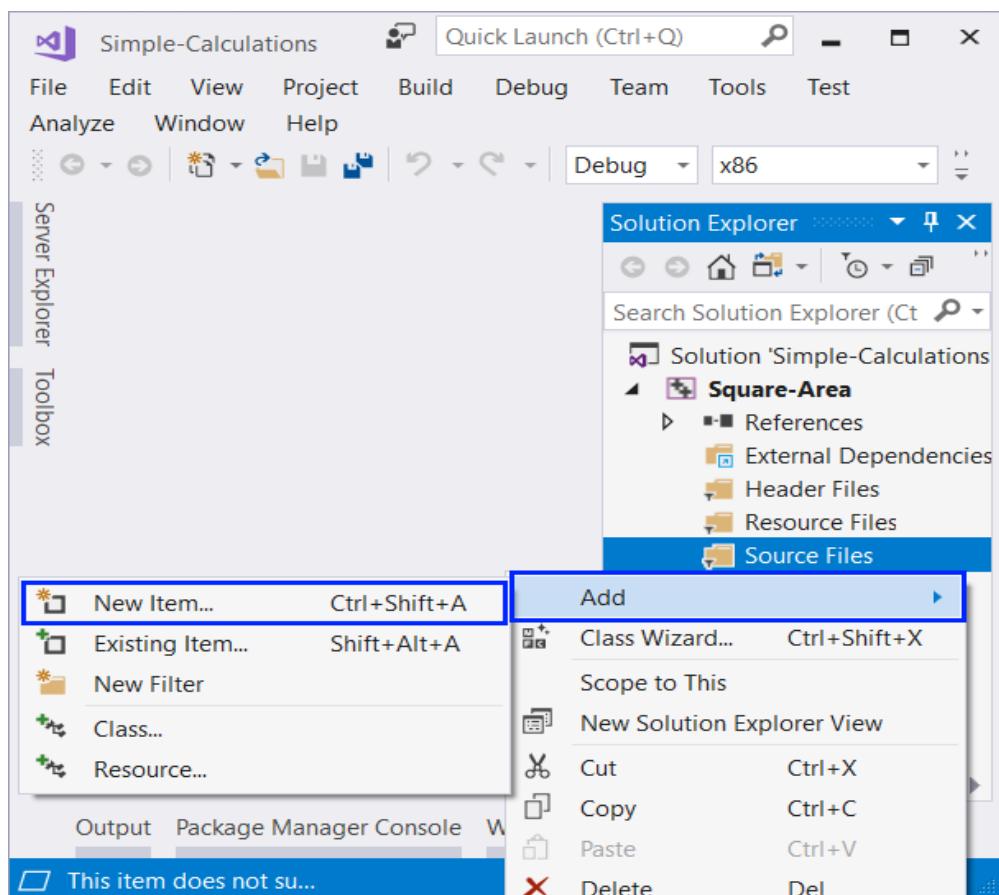
Създаваме **нов проект** в съществуващото Visual Studio решение. В Solution Explorer кликваме с десен бутон на мишката върху **Solution 'Simple-Calculations'** и избираме **[Add] -> [New Project...]**.

Тогава ще се отвори **диалогов прозорец** за избор на **тип проект**, който да бъде създаден. Избираме **[Visual C++] -> [Empty Project]** и му задаваме смислено име, например "**Square-Area**".

Следващите две картички илюстрират описаното.

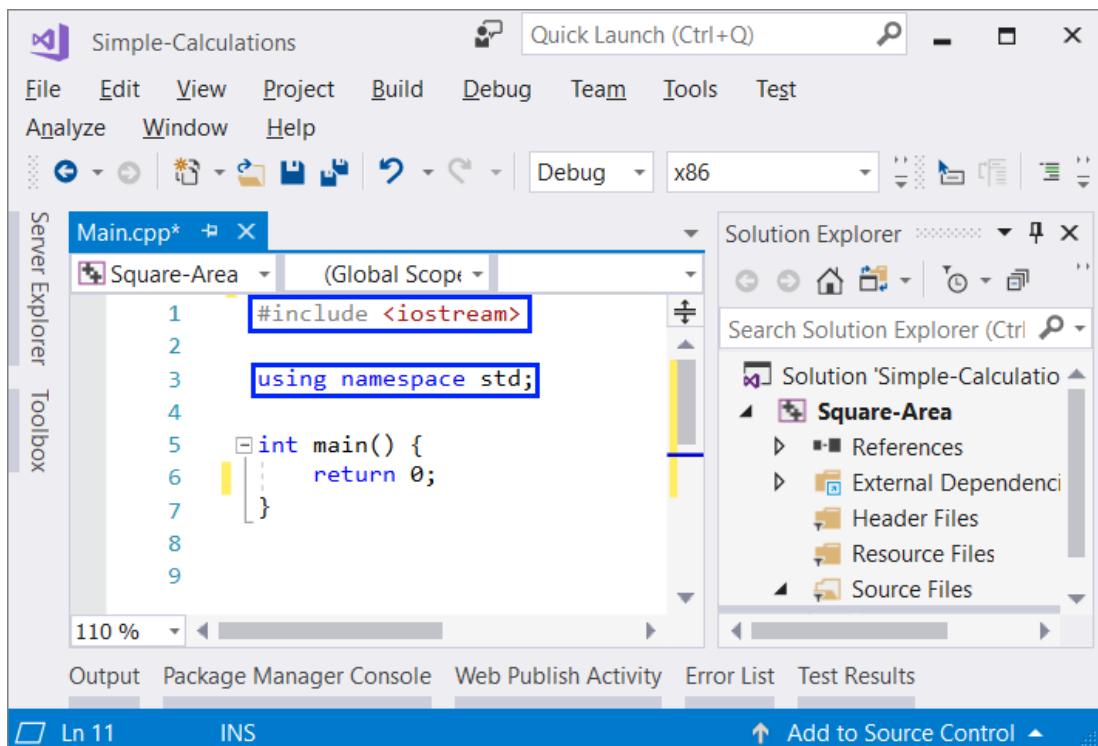


След това добавяме нов C++ файл (**.cpp**), който трябва да се назовава **Main.cpp** (от [Source Files] -> [Add] -> [New Item...]):



Във файла няма нищо написано. Както знаем, на първия ред е необходимо да добавим библиотеката `<iostream>`, а после включваме и именуваното

пространство **std**. След това пишем **main()** функцията, чрез която работата на нашата програма е възможна:



Остава да напишем кода за решаване на задачата. За целта отиваме в тялото на **main()** функцията и въвеждаме следния код:

```

cout << "a = ";
int a;
cin >> a;
int area = a * a;
cout << "Square = ";
cout << area << endl;

```

Кодът въвежда цяло число със **cin >> a;**, след това изчислява лицето на квадрата (**area = a * a;**) и накрая печата стойността на променливата **area** на конзолата. Стартираме програмата с [Ctrl + F5] и я тестваме дали работи коректно с различни входни стойности:



Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1358#0>

Трябва да получите 100 точки (напълно коректно решение):

01. Square Area

```

2
3 using namespace std;
4
5 int main()
6 {
7     cout << "a = ";
8     int a;
9     cin >> a;
10    int area = a * a;
11    cout << "Square = ";
12    cout << area << endl;
13
14    return 0;
15 }
16

```

Allowed working time: 0.100 sec.

Allowed memory: 16.00 MB

Size limit: 16.00 KB

Checker: Numbers Checker 

C++ code ▾

Submit

Submissions



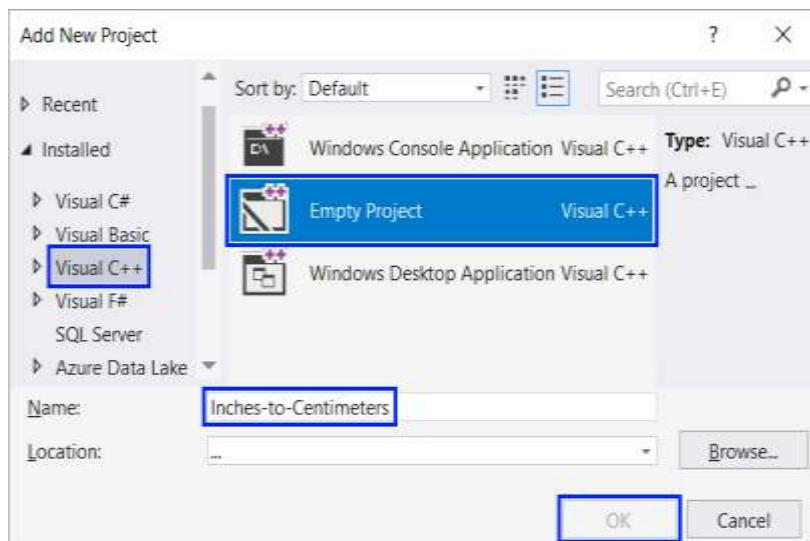
Points	Time and memory used	Submission date
4 ✓ ✓ ✓ ✓ 100 / 100	Memory: 1.82 MB Time: 0.000 s	13:27:27 08.01.2019 Details

Задача: от инчове към сантиметри

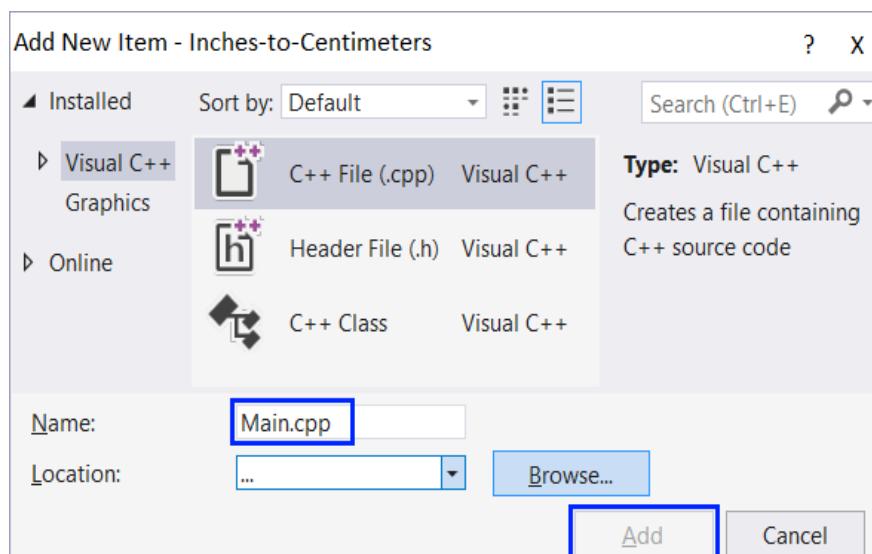
Да се напише програма, която чете от конзолата число (не непременно цяло) и преобразува числото от **инчове в сантиметри**. За целта умножава инчовете по 2.54 (защото 1 инч = 2.54 сантиметра).

Насоки и подсказки

Първо създаваме нов C++ проект в решението "Simple-Calculations". За целта кликваме с десен бутон на мишката върху решението в Solution Explorer и избираме [Add] -> [New Project...]. Избираме [Visual C++] -> [Empty Project] и задаваме име "Inches-to-Centimeters":



След това добавяме Main.cpp файла (от [Source Files] -> [Add] -> [New Item...]):



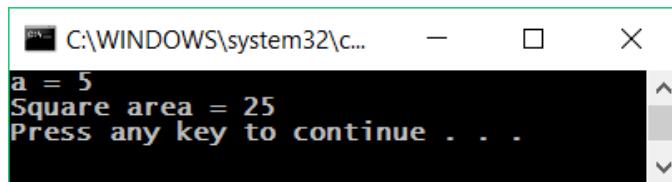
Следва да напишем кода на програмата:

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

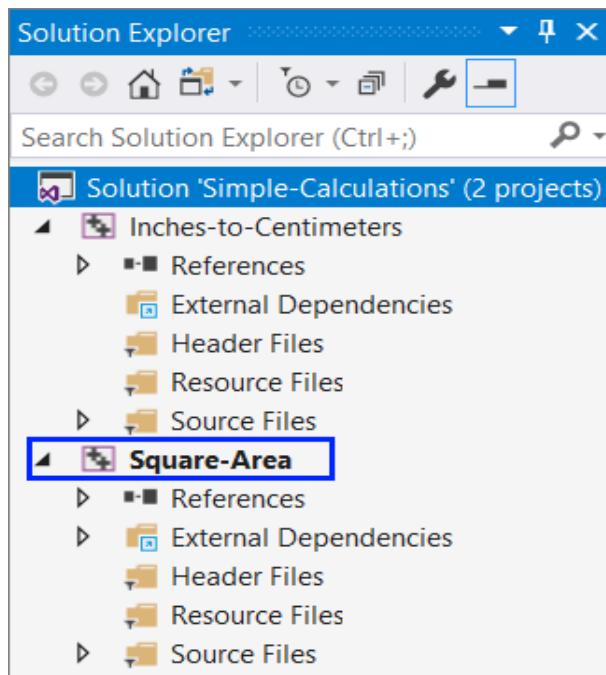
int main() {
    cout << "Inches = ";
    double inches;
    cin >> inches;
    double centimeters = inches * 2.54;
    cout << "Centimeters = ";
    cout << centimeters << endl;

    return 0;
}
```

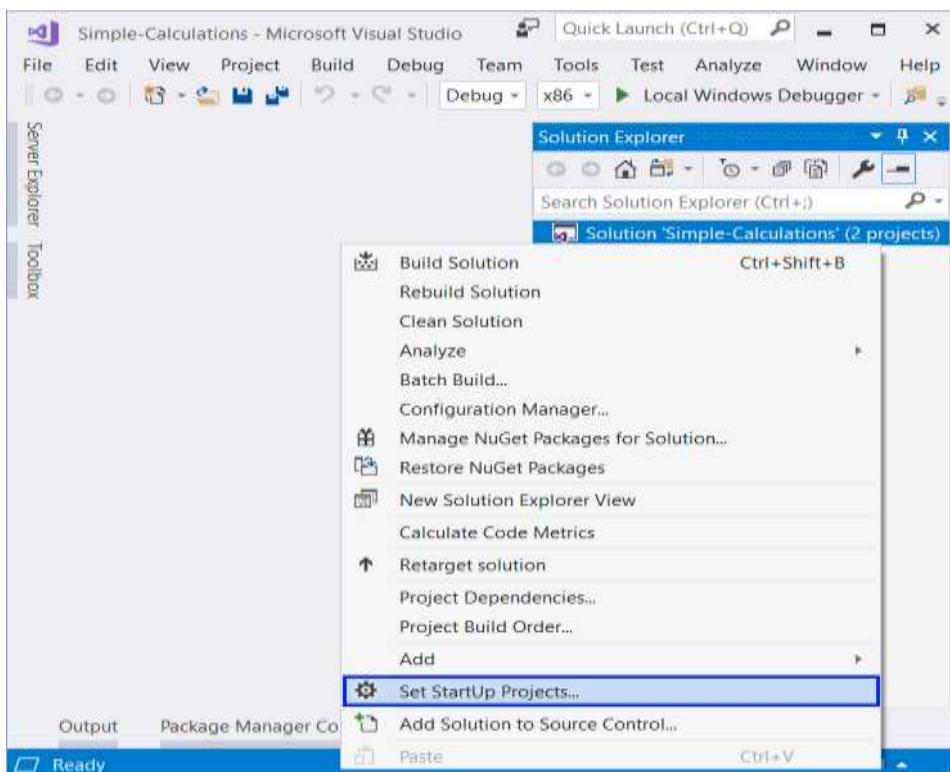
Стартираме програмата с [Ctrl + F5]:



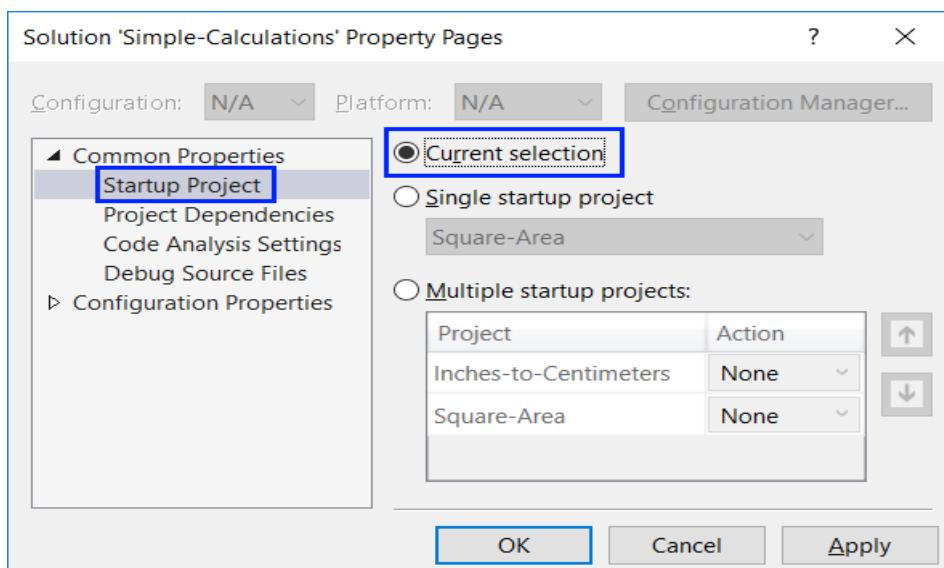
Изненада! Какво става? Програмата не работи правилно... Въсъщност това не е ли предходната програма? Във Visual Studio **текущият активен проект** в един Solution е маркиран в получерно и може да се сменя. Както виждаме, маркираният проект е "Square-Area":



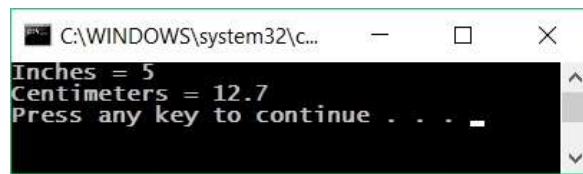
За да включим режим на **автоматично преминаване към текущия проект**, кликваме върху решението с десния бутон на мишката и избираме [Set StartUp Projects...]:



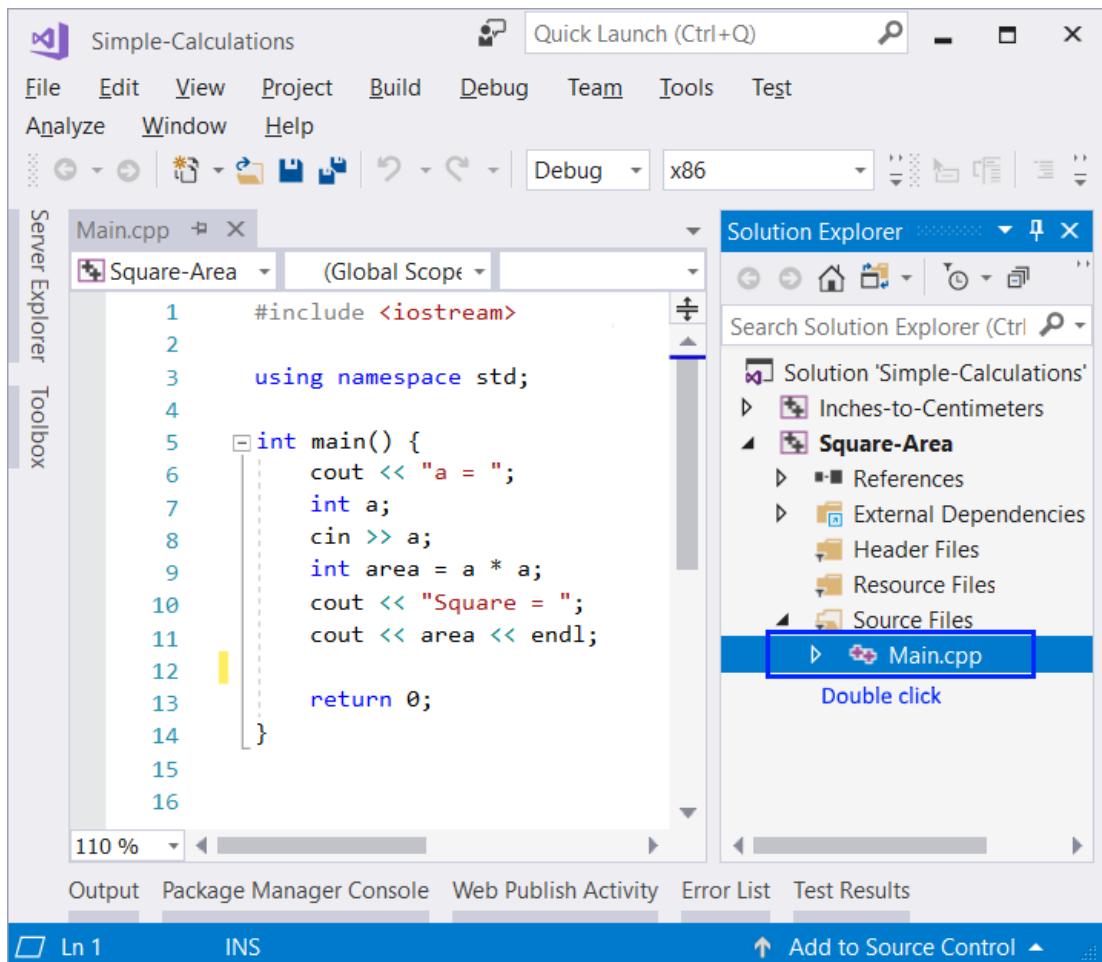
Ще се появи диалогов прозорец, от който трябва да се избере [Startup Project] -> [Current Selection]:



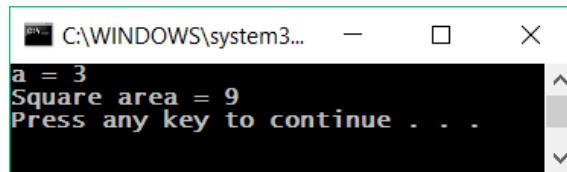
Сега отново **стартираме програмата**, както обикновено с [Ctrl + F5]. Този път ще се стартира **текущата отворена програма**, която преобразува инчове в сантиметри. Изглежда работи коректно:



Сега **да превключим към преходната програма** (лице на квадрат). Това става с двоен клик на мишката върху файла **Main.cpp** от предходния проект "Square-Area" в панела **[Solution Explorer]** на Visual Studio:

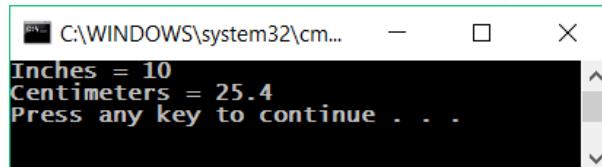


Натискаме пак [Ctrl + F5]. Този път трябва да се стартира другият проект, както е показано:



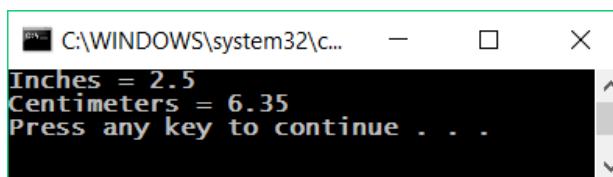
```
C:\WINDOWS\system32>
a = 3
Square area = 9
Press any key to continue . . .
```

Обратно към проекта "Inches-to-Centimeters" и го стартираме с [Ctrl + F5]:



```
C:\WINDOWS\system32>
Inches = 10
Centimeters = 25.4
Press any key to continue . . .
```

Превключването между проектите е много лесно, нали? Просто избираме файла със сорс кода на програмата, кликваме го два пъти с мишката и при стартиране тръгва програмата от този файл. Да тестваме с дробни числа, например с 2.5:

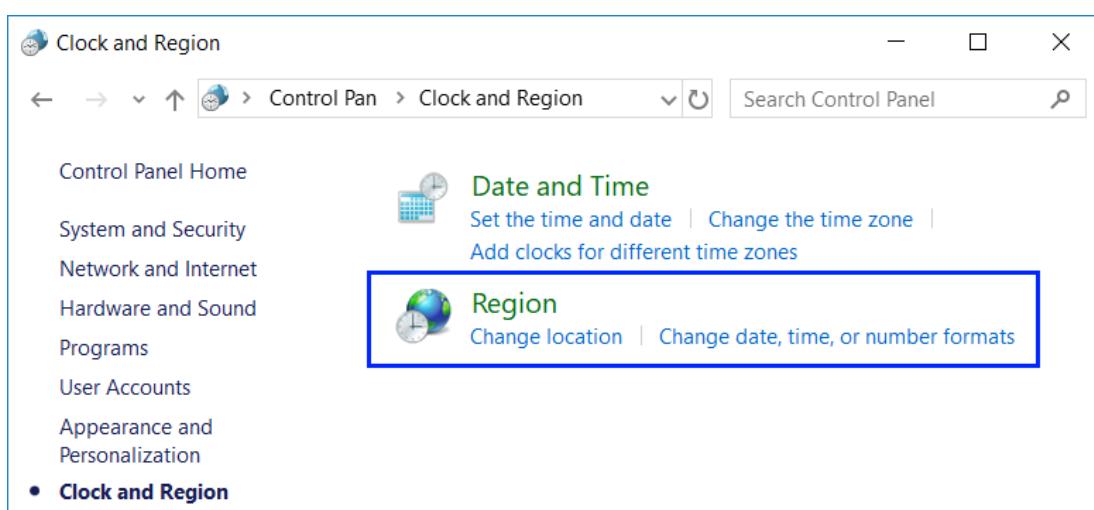


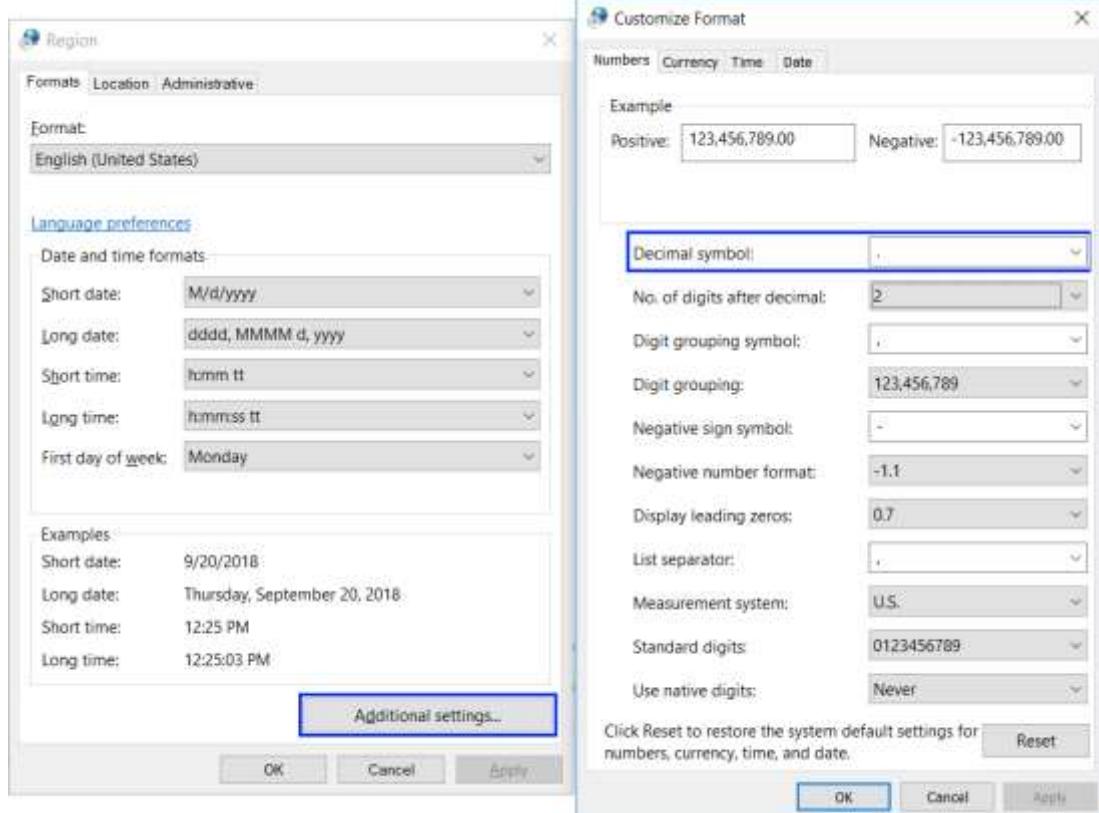
```
C:\WINDOWS\system32>
Inches = 2.5
Centimeters = 6.35
Press any key to continue . . .
```



В зависимост от регионалните настройки на операционната система, е възможно вместо **десетична точка** (US настройки) да се използва **десетична запетая** (BG настройки).

Ако програмата очаква десетична точка и бъде въведено число с десетична запетая или обратното (бъде въведена десетична точка, когато се очаква десетична запетая) е възможно да се получи грешка и тя да бъде изписана на конзолата. Затова е препоръчително да променим настройките на компютъра си, така че да се използва **десетична точка**:





Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1358#1>.

Решението би трябвало да бъде прието като напълно коректно:

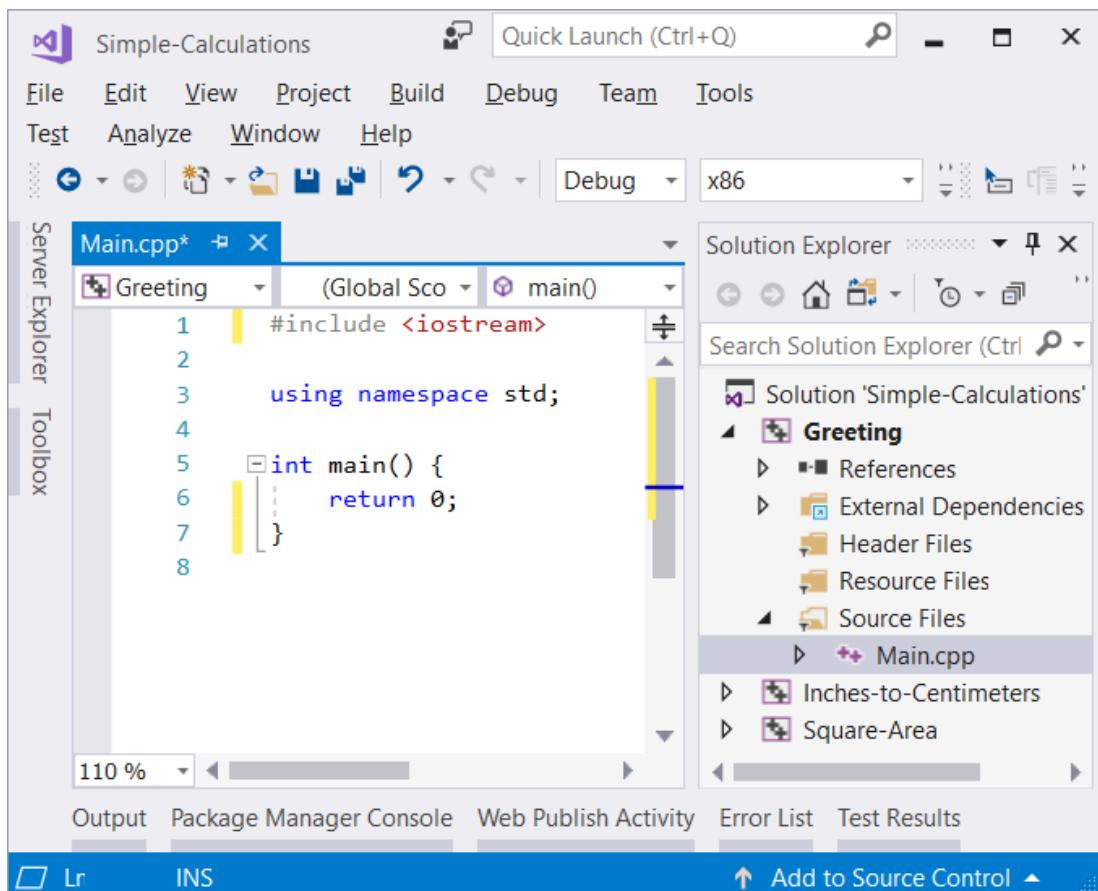
Submissions			
		1	
Points	Time and memory used	Submission date	
4/4 100 / 100	Memory: 1.86 MB Time: 0.015 s	11:07:00 09.01.2019	Details

Задача: поздрав по име

Да се напише програма, която чете от конзолата име на човек и отпечатва **Hello, <име>!**, където **<име>** е въведеното преди това име.

Насоки и подсказки

Първо създаваме нов C++ проект с име "Greeting" в решението "Simple-Calculations":



Следва да напишем кода на програмата. Може да ползвате примерния код:

```
#include <iostream>
#include <iostream>
#include <string>

using namespace std;

int main() {
    string name;
    getline(cin, name);
    cout << "Hello, " << name << endl;
    return 0;
}
```

Стартираме програмата с [Ctrl+F5] и я тестваме дали работи:

```
C:\WINDOWS\system32\cmd.exe
Svetlin Nakov
Hello, Svetlin Nakov!
Press any key to continue . . .
```

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1358#2>.

Задача: съединяване на текст и числа

Напишете C++ програма, която прочита от конзолата име, фамилия, възраст и град и печата съобщение от следния вид: **You are <firstName> <lastName>, a <age>-years old person from <town>**.

Насоки и подсказки

Добавяме към текущото Visual Studio решение още един C++ проект с име "Concatenate-Data". Пишем кода, който чете входните данни от конзолата:

```
string firstName, lastName, town;
int age;
cin >> firstName;
cin >> lastName;
cin >> age;
cin >> town;
```

Кодът, който отпечатва описаното в условието на задачата съобщение, е целенасочено замъглен и трябва да се допише от читателя:



Следва да тестваме решението локално с [Ctrl + F5] и примерни входни данни.

Тестване в Judge системата

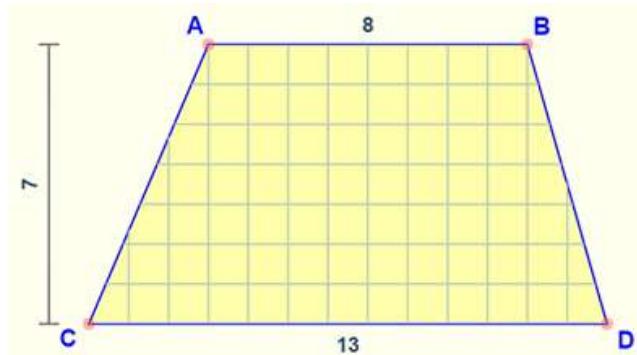
Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1358#3>.

Задача: лице на трапец

Напишете програма, която чете от конзолата три числа b_1 , b_2 и h и пресмята лицето на трапец с основи b_1 и b_2 и височина h . Формулата за лице на трапец е $(b_1 + b_2) * h / 2$.

На фигурата по-долу е показан трапец със страни 8 и 13 и височина 7. Той има лице $(8 + 13) * 7 / 2 = 73.5$.



Насоки и подсказки

Отново трябва да добавим към текущото Visual Studio решение още един C++ проект с име "Trapezoid-Area" и да напишем кода, който чете входните данни от конзолата, пресмята лицето на трапеца и го отпечатва:

```
double b1, b2, h;
```

```
cout << "Enter b1: ";
cin >> b1;
cout << "Enter b2: ";
cin >> b2;
cout << "Enter h: ";
cin >> h;
```

```
cout << "Trapezoid area = " << area << endl;
```

```
return 0;
```

Кодът на картинката е нарочно размазан, за да помислите върху него и да го допишете сами.

Тествайте решението локално с [Ctrl + F5] и въвеждане на примерни данни.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1358#4>.

Задача: периметър и лице на кръг

Напишете програма, която чете от конзолата число r и пресмята и отпечатва лицето и периметъра на кръг/окръжност с радиус r .

Примерен вход и изход

Вход	Изход	Вход	Изход
3	Area = 28.2743 Perimeter = 18.8496	4	Area = 63.6173 Perimeter = 28.2743

Насоки и подсказки

За изчисленията можете да използвате следните формули:

- **Area = pi * r * r.**
- **Perimeter = 2 * pi * r.**

Тестване в Judge системата

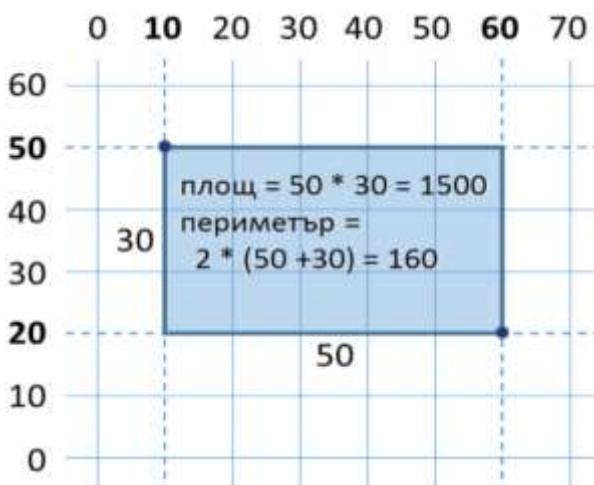
Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1358#5>.

Задача: лице на правоъгълник в равнината

Правоъгълник е зададен с координатите на два от своите срещуположни ъгъла $(x_1, y_1) - (x_2, y_2)$. Да се пресметнат **площта и периметъра** му. **Входът** се чете от конзолата. Числата **x1, y1, x2 и y2** са дадени по едно на ред. **Изходът** се извежда на конзолата и трябва да съдържа два реда с по една число на всеки от тях – лицето и периметъра.

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
60		30		600.25	350449.6875
20	1500	40	2000	500.75	2402
10		70	180	100.50	
50		-10		-200.5	



Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1358#6>.

Задача: лице на триъгълник

Напишете програма, която чете от конзолата **страна и височина на триъгълник** и пресмята неговото лице. Използвайте **формулата** за лице на триъгълник: $\text{area} = \frac{a * h}{2}$. Закръглете резултата до **2 цифри след десетичния знак** използвайки **fixed << setprecision(2)**, като включите библиотеката **<iomanip>**.

Примерен вход и изход

Вход	Изход
20 30	Triangle area = 300
7.75 8.45	Triangle area = 32.74

Вход	Изход
15 35	Triangle area = 262.50
1.23456 4.56789	Triangle area = 2.82

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1358#7>.

Задача: конзолен конвертор - от градуси °C към градуси °F

Напишете програма, която чете **градуси по скалата на Целзий** ($^{\circ}\text{C}$) и ги преобразува до **градуси по скалата на Фаренхайт** ($^{\circ}\text{F}$). Потърсете в Интернет подходяща **формула**, с която да извършите изчисленията. Закръглете резултата до **2 символа след десетичния знак**. Ето няколко примера:

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
25	77	0	32	-5.5	22.1	32.3	90.14

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1358#8>.

Задача: конзолен конвертор - от радиани в градуси

Напишете програма, която чете **ъгъл в радиани** (**rad**) и го преобразува в **градуси** (**deg**). Потърсете в Интернет подходяща формула. Закръглете резултата до най-

близкото цяло число използвайки функцията **round(degree)**, която се намира в библиотеката **<cmath>**.

Примерен вход и изход

Вход	Изход
3.1416	180
6.2832	360

Вход	Изход
0.7854	45
0.5236	30

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1358#9>.

Задача: конзолен конвертор - USD към BGN

Напишете програма за конвертиране на щатски долари (USD) в български лева (BGN). Закръглете резултата до 2 цифри след десетичния знак. Използвайте фиксиран курс между долар и лев: 1 USD = 1.79549 BGN.

Примерен вход и изход

Вход	Изход
20	35.91 BGN

Вход	Изход
100	179.55 BGN

Вход	Изход
12.5	22.44 BGN

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1358#10>.

Задача: * конзолен междувалутен конвертор

Напишете програма за конвертиране на парична сума от една валута в друга. Трябва да се поддържат следните валути: BGN, USD, EUR, GBP. Използвайте следните фиксираны валутни курсове:

Курс	USD	EUR	GBP
1 BGN	1.79549	1.95583	2.53405

Входът е сума за конвертиране, входна валута и изходна валута. **Изходът** е едно число – преобразуваната сума по посочените по-горе курсове, закръглен до 2 цифри след десетичната точка.

Примерен вход и изход

Вход	Изход
20 USD BGN	35.91 BGN
100 BGN EUR	51.13 EUR

Вход	Изход
12.35 EUR GBP	9.53 GBP
150.35 USD EUR	138.02 EUR

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/504#11>.

Глава 2.2. Прости пресмятания с числа

– ИЗПИТНИ задачи

В предходната глава се запознахме със системната конзола и как да работим с нея – как да прочетем число от конзолата и как да отпечатаме резултат на конзолата. Разгледахме основните аритметични операции и накратко споменахме типовете данни. В настоящата глава ще упражним и затвърдим наученото досега, като разгледаме няколко **по-сложни задачи**, давани на изпити.

Четене на числа от конзолата

Преди да преминем към задачите, да си припомним най-важното от изучавания материал от предходната тема. Ще започнем с четенето на числа от конзолата.

Четене на цяло число

Необходима ни е променлива, в която да запазим числото (напр. **num**), и да използваме стандартната команда за четене на данни от конзолата **cin**:

```
int num;  
cin >> num;
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/ReadingNumber>.

Четене на дробно число

По същия начин, както четем цяло число, но този път трябва да използваме променлива от тип **double**:

```
double num;  
cin >> num;
```

Може да пуснете в действие и тествате кода от горния пример онлайн: <https://repl.it/@vncpetrov/ReadingFloatingPointNumber>.

Извеждане на текст по шаблон (placeholder)

Placeholder представлява израз, който ще бъде заменен с конкретна стойност при отпечатване. Функцията **printf(...)** поддържа печатане на текст по шаблон, като първият аргумент, който трябва да подадем, е форматиращият низ, следван от броя аргументи, равен на броя на плейсхолдърите. Функцията се намира в библиотеката **stdio.h** и е необходимо да я реферираме в началото на програмата:

```
printf("You are %s %s, a %d-years old person from %s",  
       firstName.c_str(), secondName.c_str(), age, town.c_str());
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/Placeholders>

Аритметични оператори

Да си припомним основните аритметични оператори за пресмятания с числа.

Оператор +

```
int result = 3 + 5; // резултатът е 8
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/Sum>.

Оператор -

```
int result = 3 - 5; // резултатът е -2
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/Difference>.

Оператор *

```
int result = 3 * 5; // резултатът е 15
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/Multiplication>.

Оператор /

```
int result = 7 / 3; // резултатът е 2 (целочислено деление)
double result2 = 5 / 2.0; // резултатът е 2.5 (дробно деление)
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/Division>.

Конкатенация

При използване на оператора `+` между променливи от тип текст се извършва т.нар. конкатенация (слепване на низове).

```
string firstName = "Ivan";
string lastName = "Ivanov";
int age = 19;
string str = firstName + " " + lastName + " is " + to_string(age)
+ " years old";
// Ivan Ivanov is 19 years old
```

Може да пуснете в действие и тествате кода от горния пример онлайн:

<https://repl.it/@vncpetrov/ConcatenationExample>.

Функцията **to_string(...)** конвертира годините в текст, за да може да се извърши конкатенацията.

Изпитни задачи

Сега, след като си припомнихме как се извършват пресмятания с числа и как се четат и печатат числа на конзолата, да минем към задачите. Ще решим няколко задачи от приемен изпит за кандидатстване в СофтУни.

Задача: учебна зала

Учебна зала има правоъгълен размер l на w метра, без колони във вътрешността си. Залата е разделена на две части – лява и дясна, с коридор - приблизително по средата. В лявата и в дясната част има **редици с бюра**. В задната част на залата има голяма **входна врата**. В предната част на залата има **катедра** с подиум за преподавателя. Едно **работно място** заема **70 на 120 см** (маса с размер 70 на 40 см + място за стол и преминаване с размер 70 на 80 см). **Коридорът** е широк поне **100 см**. Изчислено е, че заради **входната врата** (която е с отвор 160 см) **се губи точно 1 работно място**, а заради **катедрата** (която е с размер 160 на 120 см) **се губят точно 2 работни места**. Напишете програма, която въвежда размери на учебната зала и изчислява **броя работни места в нея** при описаното разположение (вж. фигурата).

Входни данни

От конзолата се четат **2 числа**, по едно на ред: l (дължина в метри) и w (широкина в метри).

Ограничения: $3 \leq w \leq l \leq 100$.

Изходни данни

Да се отпечата на конзолата едно цяло число: **броят места** в учебната зала.

Пояснения към примерите на стр. 92

В първия пример залата е дълга 1500 см. В нея могат да бъдат разположени **12 реда** ($12 * 120 \text{ cm} = 1440 + 60 \text{ см остатък}$). Залата е широка 890 см. От тях 100 см отиват за коридора в средата. В останалите 790 см могат да се разположат по **11 бюра на ред** ($11 * 70 \text{ cm} = 770 \text{ cm} + 20 \text{ см остатък}$). **Брой места = 12 * 11 - 3 = 132 - 3 = 129** (имаме 12 реда по 11 места = 132 минус 3 места за катедра и входна врата).

Във втория пример залата е дълга 840 см. В нея могат да бъдат разположени **7 реда** ($7 * 120 \text{ cm} = 840$, без остатък). Залата е широка 520 см. От тях 100 см отиват за коридора в средата. В останалите 420 см могат да се разположат по **6 бюра на ред** ($6 * 70 \text{ cm} = 420 \text{ cm}$, без остатък). **Брой места = 7 * 6 - 3 = 42 - 3 = 39** (имаме 7 реда по 6 места = 42 минус 3 места за катедра и входна врата).

Примерен вход и изход

Вход	Изход	Чертеж
15 8.9	129	 <p>коридор: поне 1 m</p> <p>врата</p> <p>катедра</p>
8.4 5.2	39	 <p>коридор: 1 m</p> <p>врата</p> <p>катедра</p>

Насоки и подсказки

Опитайте първо сами да решите задачата. Ако не успеете, разгледайте насоките и подсказките.

Идея за решение

Както при всяка една задача по програмиране, е **важно да си изградим идея за решението ѝ**, преди да започнем да пишем код. Да разгледаме внимателно зададеното ни условие. Изисква се да напишем програма, която да изчислява броя работни места в една зала, като този брой е зависим от дължината и височината ѝ. Забелязваме, че те ще ни бъдат подадени като входни данни **в метри**, а информацията за това колко пространство заемат работните места и коридорът, ни е дадена **в сантиметри**. За да извършим изчисленията, ще трябва

да използваме еднакви мерни единици, няма значение дали ще изберем да превърнем височината и дължината в сантиметри, или останалите данни в метри. За представеното тук решение е избрана първата опция.

Следва да изчислим **колко колони и колко редици** с бюра ще се съберат. Колоните можем да пресметнем като **от широчината извадим необходимото място за коридора (100 см) и разделим остатъка на 70 см** (колкото е дължината на едно работно място). Редиците ще намерим като разделим **дължината на 120 см**. И при двете операции може да се получи реално число с цяла и дробна част, **в променлива трябва да запазим обаче само цялата част**. Накрая умножаваме броя на редиците по този на колоните и от него изваждаме 3 (местата, които се губят заради входната врата и катедрата). Така ще получим исканата стойност.

Избор на типове данни

От примерните входни данни виждаме, че за вход може да ни бъде подадено реално число с цяла и дробна част, затова не е подходящо да избираме тип **int**, нека за тях използваме **double**. Изборът на тип за следващите променливи зависи от метода за решение, който изберем. Както всяка задача по програмиране, тази също има **повече от един начин на решение**. Тук ще бъдат показани два такива.

Решение

Време е да пристъпим към решението. Мислено можем да го разделим на три подзадачи:

- **Прочитане** на входните данни.
- **Извършване** на изчисленията.
- **Извеждане** на изход на конзолата.

Първото, което трябва да направим, е да прочетем входните данни от конзолата. С функцията **cin** четем стойностите от конзолата, а типът данни ще бъде **double**:

```
double length;
cin >> length;
```

```
double width;
cin >> width;
```

Нека пристъпим към изчисленията. Особеното тук е, че след като извършим делението, трябва да запазим в променлива само цялата част от резултата.



Търсете в Google! Винаги, когато имаме идея как да решим даден проблем, но не знаем как да го изпишем на C++, или когато се сблъскаме с такъв, за който предполагаме, че много други хора са имали, най-лесно е да се справим като потърсим информация в Интернет.

В случая може да пробваме със следното търсене: "[C++ get whole number part of double](#)". Откриваме, че едната възможност е да използваме функцията **trunc(...)**. Тъй като тя работи с променливи от тип **double**, за броя редици и колони създаваме променливи също от този тип. Кодът по-долу е целенасочено замъглен и трябва да бъде довършен от читателя:

Втори вариант: както вече знаем, операторът за деление **/** има различно действие върху цели и реални числа. При деление на целочислен с целочислен тип (напр. **int**), върнатият резултат е отново целочислен. Следователно можем да потърсим как да преобразуваме реалните числа, които имаме като стойности за височината и широчината, в цели числа и след това да извършим делението.

Тъй като в този случай може да се получи загуба на данни, идваща от премахването на дробната част, е необходимо преобразуването да стане изрично (explicit typecasting). Използваме оператора за преобразуване на данни (**type**), като заменяме думата **type** с необходимия тип данни и го поставяме преди променливата:

```
int cols = ((int)width - 100) / 70;
int rows = (int)length / 120;
int seats = rows * cols - 3;
```

Със **cout** отпечатваме резултата на конзолата:

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1359#0>.

Задача: зеленчукова борса

Градинар продава реколтата от градината си на зеленчуковата борса. Продава зеленчуци за **N** лева на килограм и плодове за **M** лева за килограм. Напишете програма, която да пресмята приходите от реколтата в евро (ако приемем, че **едно евро** е равно на **1.94 лв.**).

Входни данни

От конзолата се четат **4 числа**, по едно на ред:

- Първи ред – Цена за килограм зеленчуци – число с плаваща запетая.
- Втори ред – Цена за килограм плодове – число с плаваща запетая.

- Трети ред – Общо килограми на зеленчуците – цяло число.
- Четвърти ред – Общо килограми на плодовете – цяло число.

Ограничения: Всички числа ще са в интервала от 0.00 до 1000.00

Изходни данни

Да се отпечата на конзолата **едно число с плаваща запетая**: приходите от всички плодове и зеленчуци в евро.

Примерен вход и изход

Вход	Изход	Вход	Изход
0.194		1.5	
19.4	101	2.5	
10		10	20.6185567010309
10		10	

Пояснения към първия пример:

- Зеленчуците струват: 0.194 лв. * 10 кг. = **1.94** лв.
- Плодовете струват: 19.4 лв. * 10 кг. = **194** лв.
- Общо: **195.94** лв. = **101** евро.

Насоки и подсказки

Първо ще дадем няколко разъждения, а след това и конкретни насоки за решаване на задачата, както и съществената част от кода.

Идея за решение

Нека първо разгледаме зададеното ни условие. В случая, от нас се иска да пресметнем колко е **общият приход** от реколтата. Той е равен на **сбора от печалбата от плодовете и зеленчуците**, а тях можем да изчислим като умножим **цената на килограм по количеството им**. Входните данни са дадени в лева, а за изхода се изисква да бъде в евро. По условие 1 евро е равно на 1,94 лева, следователно, за да получим исканата **изходна стойност**, трябва да разделим **сбора на 1,94**.

Избор на типове данни

След като сме изяснили идеята си за решаването на задачата, можем да пристъпим към избора на подходящи типове данни. Да разгледаме **входа**: дадени са **две цели числа** за общия брой килограми на зеленчуците и плодовете, съответно променливите, които декларираме, за да пазим техните стойности, ще бъдат от тип **int**. За цените на плодовете и зеленчуците е указано, че ще бъдат подадени **две числа с плаваща запетая**, т.е. променливите ще бъдат от тип **double**.

Може да декларираме също две променливи, в които да пазим стойността на печалбата от плодовете и зеленчуците поотделно. Тъй като умножаваме променлива от тип **int** (общо килограми) с такава от тип **double** (цена), резултатът също трябва да бъде от тип **double**. Нека поясним това: по принцип операторите работят с аргументи от един и същи тип. Следователно, за да извършим операция като умножение върху два различни типа данни, ни се налага да ги преобразуваме към един и същ такъв. Когато в един израз има типове с различен обхват, преобразуването винаги се извършва към този с най-голям обхват, в този случай това е **double**. Тъй като няма опасност от загуба на данни, преобразуването е неявно (implicit) и става автоматично от компилатора.

Като **изход** се изиска също **число с плаваща запетая**, т.е. резултата ще пазим в променлива от тип **double**.

Решение

Време е да пристъпим към решението. Мислено можем да го разделим на три подзадачи:

- **Прочитане** на входните данни.
- **Извършване** на изчисленията.
- **Извеждане** на изход на конзолата.

За да прочетем входните данни декларираме променливи, като внимаваме да използваме правилният тип данни и да ги именуваме по такъв начин, който да ни подсказва какви стойности съдържат променливите. С функцията **cin** прочитаме стойностите от конзолата като типовете данни, които ще използваме ще бъдат **int** и **double**:

```
double pricePerKilogramVegetables;
cin >> pricePerKilogramVegetables;
```



След което, извършваме необходимите изчисления, както е показано тук:

```

double priceVegetables =
    pricePerKilogramVegetables * totalKilogramVegetables;
double priceFruits = pricePerKilogramFruits * totalKilogramFruits;
double totalPrice = priceVegetables + priceFruits;

```

В условието на задачата не е зададено специално форматиране на изхода, но за да може Judge системата да приеме решението ни, трябва да използваме **setprecision** заради неточностите, които се получават в сметките. Както в математиката, така и в програмирането делението има приоритет пред събирането. За задачата обаче трябва първо да **сметнем** **сбора** на двете получени стойности и след това да **разделим** на 1.94. За да дадем предимство на събирането, може да използваме скоби. Със **cout** отпечатваме изхода на конзолата.

```
cout << setprecision(15) << totalPrice << endl;
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1359#1>.

Задача: ремонт на плоочки

На площадката пред жилищен блок трябва да се поставят плоочки. Площадката е с форма на квадрат със страна **N** метра. Плочките са широки „**W**“ метра и дълги „**L**“ метра. На площадката има една пейка с ширина **M** метра и дължина **O** метра. Под нея не е нужно да се слагат плоочки. Всяка плоочка се поставя за **0.2** минути.

Напишете програма, която чете от конзолата размерите на площадката, плоchkите и пейката и пресмята **колко** плоочки са **необходими** да се покрие площадката и пресмята **времето** за поставяне на всички плоочки.

Пример: площадка с размер 20 м. има площ 400 кв.м.. Пейка, широка 1 м. и дълга 2 м., заема площ 2 кв.м. Една плоочка е широка 5 м. и дълга 4 м. и има площ = 20 кв.м. Площта, която трябва да се покрие, е $400 - 2 = 398$ кв.м. Необходими са $398 / 20 = 19.90$ плоочки. Необходимото време е $19.90 * 0.2 = 3.98$ минути.

Входни данни

От конзолата се четат 5 числа:

- **N** – дълчината на страна от площадката в интервала [1 ... 100].
- **W** – широчината на една плоочка в интервала [0.1 ... 10.00].
- **L** – дълчината на една плоочка в интервала [0.1 ... 10.00].
- **M** – широчината на пейката в интервала [0 ... 10].
- **O** – дълчината на пейката в интервала [0 ... 10].

Изходни данни

Да се отпечатат на конзолата **две числа**: броя **плочки**, необходим за ремонта и времето за поставяне, всяко на нов ред.

Примерен вход и изход

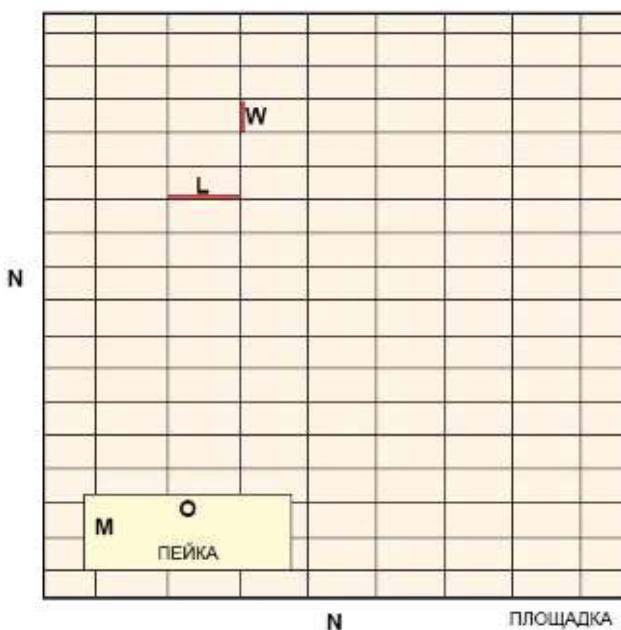
Вход	Изход	Вход	Изход
20		40	
5		0.8	
4	19.9	0.6	3302.08333333333
1	3.98	3	660.416666666667
2		5	

Обяснение към първия пример:

- Обща площ = $20 * 20 = 400$.
- Площ на пейката = $1 * 2 = 2$.
- Площ за покриване = $400 - 2 = 398$.
- Площ на плочки = $5 * 4 = 20$.
- Необходими плочки = $398 \sqrt{20} = 19.9$.
- Необходимо време = $19.9 * 0.2 = 3.98$.

Насоки и подсказки

Нека да си направим чертеж, за да поясним условието на задачата.



Идея за решение

Изисква се да пресметнем **броя** **плочки**, който трябва да се постави, както и **времето**, за което това ще се извърши. За да **изчислим** **броя**, е необходимо да сметнем **площта**, която трябва да се покрие, и да я **разделим на лицето на една** **плочка**. По условие площадката е квадратна, следователно общата площ ще намерим, като умножим страната ѝ по стойността ѝ **N * N**. След това пресмятаме **площта, която заема пейката**, също като умножим двете ѝ страни **M * O**. Като извадим площта на пейката от тази на цялата площадка, получаваме площта, която трябва да се ремонтира.

Лицето на единична плочка изчисляваме като **умножим** **едната** ѝ **страница** по **другата W * L**. Както вече отбелязахме, сега трябва да **разделим** **площта за покриване на площта на една плочка**. По този начин ще разберем какъв е необходимият брой плочки. Него умножаваме по **0.2** (времето, за което по условие се поставя една плочка). Така вече ще имаме исканите изходни стойности.

Избор на типове данни

Дължината на страна от площадката, широчината и дължината на пейката ще бъдат дадени като **цели числа**, следователно, за да запазим техните стойности може да декларираме **променливи от тип int**. За широчината и дължината на плочките ще ни бъдат подадени реални числа (с цяла и дробна част), затова за тях ще използваме **double**. Изходът на задачата отново ще е реално число, т.е. променливите ще бъдат също от тип **double**.

Решение

Както и в предходните задачи, можем мислено да разделим решението на три части:

- **Прочитане** на входните данни.
- **Извършване** на изчисленията.
- **Извеждане** на изход на конзолата.

Първото, което трябва да направим, е да разгледаме **входните данни** на задачата. Важно е да внимаваме за последователността, в която са дадени. Със **cin** прочитаме стойностите от конзолата:

```
int groundLength;
cin >> groundLength;
```



След като сме инициализирали променливите и сме запазили съответните стойности в тях, пристъпваме към **изчисленията**. Тъй като стойностите на променливите **groundLength**, **benchWidth** и **benchLength**, с които работим, са запазени в променливи от тип **int**, за резултатите от изчисленията може да дефинираме **променливи също от този тип**. Кодът по-долу е нарочно даден замъглен, за да може читателят да помисли самостоятелно над него:



Променливите **tileWidth** и **tileLength** са от тип **double**, т.е. за лицето на една **плочка** създаваме променлива от същия тип. За финал **изчисляваме** стойностите, които трябва да **отпечатаме** на конзолата. **Броят** на необходимите **плочки** получаваме като **разделим площта**, която трябва да се покрие, на **площта на единична плочка**. При деление на две числа, от които **едното е реално**, резултатът е **реално число** с цяла и дробна част. Следователно, за да са коректни изчисленията ни, запазваме резултата в променлива от тип **double**. В условието на задачата не е зададено специално форматиране или закръгляне на изхода, затова просто отпечатваме стойностите със **cout**:

```
double tiles = areaToRepair / (tileWidth * tileLength);
double time = tiles * 0.2;

cout << tiles << endl;
cout << time << endl;
```

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1359#2>.

Задача: парички

Преди време Пешо си е купил биткойни. Сега ще ходи на екскурзия из Европа и ще му трябва евро. Освен биткойни има и китайски юани. Пешо иска да обмени парите си в евро за екскурзията. Напишете програма, която да пресмята колко евро може да купи спрямо следните валутни курсове:

- 1 биткойн = 1168 лева.
- 1 китайски юан = 0.15 долара.
- 1 доллар = 1.76 лева.
- 1 евро = 1.95 лева.

Обменното бюро има комисионна от 0 до 5 процента от крайната сума в евро.

Входни данни

От конзолата се четат 3 числа:

- На първия ред – броят биткойни. Цяло число в интервала [0 ... 20].
- На втория ред – броят китайски юани. Реално число в интервала [0.00 ... 50 000.00].
- На третия ред – комисионната. Реално число в интервала [0.00 ... 5.00].

Изходни данни

На конзолата да се отпечата 1 число - резултатът от обмяната на валутите. Не е нужно резултатът да се закръгля.

Примерен вход и изход

Вход	Изход
1	
5	
5	569.668717948718

Обяснение:

- 1 биткойн = 1168 лева
- 5 юана = 0.75 долара
- 0.75 долара = 1.32 лева
- $1168 + 1.32 = 1169.32$ лева = 599.651282051282 евро
- Комисионна: 5% от 599.651282051282 = 29.9825641025641
- Резултат: $599.651282051282 - 29.9825641025641 = 569.668717948718$ евро

Вход	Изход
20	
5678	12442.2442010256
2.4	

Вход	Изход
7	
50200.12	10659.4701177436
3	

Насоки и подсказки

Нека отново помислим първо за начина, по който можем да решим задачата, преди да започнем да пишем код.

Идея за решение

Виждаме, че ще ни бъдат подадени **броят биткойни** и **броят китайски юани**. За **изходната стойност** е указано да бъде **евро**. В условието са посочени и валутните курсове, с които трябва да работим. Забелязваме, че към евро можем да преобразуваме само сума в лева, следователно трябва **първо да пресметнем цялата сума**, която Пешо притежава в лева, и след това да **изчислим изходната стойност**.

Тъй като ни е дадена информация за валутния курс на биткойни срещу лева, можем директно да направим това преобразуване. От друга страна, за да получим стойността на **китайските юани в лева**, трябва първо да ги **конвертираме в долари**, а след това **доларите - в лева**. Накрая ще **съберем двете получени стойности** и ще пресметнем на колко евро съответстват.

Остава последната стъпка: да **пресметнем колко ще бъде комисионната** и да извадим получената сума от общата. Като комисионна ще ни бъде подадено **реално число**, което ще представлява определен **процент от общата сума**. Нека още в началото разделим подаденото число на 100, за да изчислим **процентната му стойност**. Няя ще умножим по сумата в евро, а резултатът ще извадим от същата тази сума. Получената сума ще отпечатаме на конзолата.

Избор на типове данни

Биткойните са дадени като **цяло число**, следователно за тяхната стойност може да използваме **променлива от тип int**. Като брой **китайски юани** и **комисионна** ще получим **реално число**, следователно за тях използваме **double**. Тъй като **double** е типът данни с по-голям обхват, а **изходът** също ще бъде **реално число**, ще използваме него и за останалите променливи, които създаваме.

Решение

След като сме си изградили идея за решението на задачата и сме избрали структурите от данни, с които ще работим, е време да пристъпим към **писането на код**. Както и в предните задачи, можем да разделим решението на три подзадачи:

- Прочитане на входните данни.

- Извършване на изчисленията.
 - Извеждане на изход на конзолата.

Декларираме променливите, които ще използваме, като отново внимаваме да изберем **смислени имена**, които подсказват какво съдържат те. Инициализираме техните стойности: със **cin** четем подадените числа на конзолата:

```
double bitcoins;  
cin >> bitcoins;
```

Извършваме необходимите изчисления. Кодът по-долу е целенасочено замъглен и трябва да бъде довършен от читателя:

```
double bitcoinsToLeva = bitcoins * 1168;  
double yuansToDollars = yuans * 0.15;  
double dollarsToLeva = yuansToDollars * 1.76;
```

Накрая пресмятаме стойността на комисионната и я изваждаме от сумата в евро. Нека обърнем внимание на начина, по който можем да изпишем това: **euro - commission * euro** е съкратен начин за изписване на **euro = euro - (commission * euro)**. В случая използваме комбиниран оператор за присвояване (**=**), който изважда стойността от операнда вдясно от този вляво. Операторът за умножение (*****) има по-висок приоритет от комбинириания оператор, затова изразът **commission * euro** се изпълнява първи, след което неговата стойност се изважда.

В условието на задачата не е зададено специално форматиране или закръгляне на резултата, следователно трябва просто да изчислим изхода и да го отпечатаме на конзолата:

```
euro -= euro * commission;  
cout << euro << endl;
```

Нека обърнем внимание на нещо, което важи за всички задачи от този тип: разписано по този начин, решението на задачата е доста подробно. Тъй като условието като цяло не е сложно, бихме могли на теория да напишем един голям

израз, в който директно след получаване на входните данни да сметнем изходната стойност. Например, такъв израз би изглеждал ето така:

```
double euro = (bitcoins * 1168 + yuans * 0.15 * 1.76) / 1.95
- ((bitcoins * 1168 + yuans * 0.15 * 1.76) / 1.95 * commission);
```

Този код би дал правилен резултат, но се чете трудно. Няма да ни е лесно да разберем какво прави, дали съдържа грешки и ако има такива - как да ги поправим. По-добра практика е вместо един сложен израз да напишем няколко прости и да запишем резултатите от тях в променливи със подходящи имена.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1359#3>.

Задача: дневна печалба

Иван работи средно N дни в месеца, като изкарва средно по M долара на ден. В края на годината получава бонус, който е равен на 2.5 месечни заплати. От спечеленото през годината му се удържат 25% данъци. Напишете програма, която да пресмята колко е чистата средна печалба на Иван на ден в лева. Годината има 365 дни. Курсът на долара спрямо лева ще се подава на функцията.

Входни данни

От конзолата се четат 3 числа:

- На първия ред – работни дни в месеца. Цяло число в интервала [5 ... 30].
- На втория ред – изкарани пари на ден. Реално число в интервала [10.00 ... 2000.00].
- На третия ред – курсът на долара спрямо лева /1 доллар = X лева/. Реално число в интервала [0.99 ... 1.99].

Изходни данни

На конзолата да се отпечата едно число – средната печалба на ден в лева. Резултатът да се форматира до втората цифра след десетичния знак.

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
21 75.00 1.59	74.61	15 105 1.71	80.24	22 199.99 1.50	196.63

Обяснение на първия вход и изход

- 1 месечна заплата = $21 * 75 = 1575$ долара.
- Годишен доход = $1575 * 12 + 1575 * 2.5 = 22837.5$ долара.
- Данък = 25% от 22837.5 = 5709.375 лева.
- Чист годишен доход = 17128.125 долара = 27233.71875 лева.
- Средна печалба на ден = $27233.71875 / 365 = 74.61$ лева.

Насоки и подсказки

Първо да анализираме задачата и да измислим как да я решим. След това ще изберем типовете данни и накрая ще напишем кода на решението.

Идея за решение

Нека първо пресметнем колко е **месечната заплата** на Иван. Това ще направим като **умножим работните дни в месеца по парите**, които той печели на ден. Умножаваме **получения резултат** първо по 12, за да изчислим колко е заплатата му за 12 месеца, а след това и **по 2.5**, за да пресметнем бонуса. Като съберем двете получени стойности, ще изчислим **общия му годишен доход**. От него трябва **да извадим 25%**. Това може да направим като умножим общия доход по 0.25 и извадим резултата от него. Спрямо дадения ни курс **преобразуваме долларите в лева**, след което **разделяме резултата на дните в годината**, за които приемаме, че са 365.

Избор на типове данни

Работните дни за месец са дадени като **цяло число**, следователно за тяхната стойност може да декларираме променлива от **тип int**. За **изкараните пари**, както и за **курса на долара спрямо лева**, ще получим **реално число**, следователно за тях използваме **double**. Тъй като **double** е типът данни с **по-голям обхват**, а за изходната стойност също се изисква **реално число** (с цяла и дробна част), ще използваме него и за останалите променливи, които създаваме.

Решение

Отново, след като имаме идея как да решим задачата и сме помислили за типовете данни, с които ще работим, пристъпваме към **писането на програмата**. Както и в предходните задачи, можем да разделим решението на три подзадачи:

- Прочитане на входните данни.
- Извършване на изчисленията.
- Извеждане на изход на конзолата.

Декларираме променливите, които ще използваме, като отново се стараем да изберем **подходящи имена**. С функцията **cin** прочитаме подадените числа на конзолата. Типовете данни, които ще използваме са **int** и **double**:



Извършваме необходимите изчисления. Кодът по-долу е целенасочено замъглен и трябва да бъде довършен от читателя:

```
double monthSalary = workDays * moneyPerDay;
double moneyPerYear = (monthSalary * 12) + (monthSalary * 2.5);
double taxes = moneyPerYear * 0.25;
double netSalary = moneyPerYear - taxes;
double salaryInLeva = netSalary * currencyRate;
```

Бихме могли да напишем израза, с който пресмятаме общия годишен доход, и без скоби. Тъй като умножението е операция с по-висок приоритет от събирането, то ще се извърши първо. Въпреки това **писането на скоби се препоръчва, когато използваме повече оператори**, защото така кодът става по-лесно четим и възможността да се допусне грешка е по-малка.

Накрая остава да изведем резултата на конзолата. Забелязваме, че се **изисква форматиране на числената стойност до втория знак след десетичната точка**. За целта ще използваме функцията **setprecision(...)**, която се намира в библиотеката **iomanip**:

```
double average = salaryInLeva / 365;
cout << fixed << setprecision(2) << average << endl;
```

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1359#4>.

Глава 3.1. Прости проверки

В настоящата глава ще разгледаме **условните конструкции в езика C++**, чрез които нашата програма може да има различно поведение, в зависимост от дадено условие. Ще обясним синтаксиса на условните оператори за проверки (**if**, **if-else**) с подходящи примери и ще видим в какъв диапазон живее една променлива (нейният **обхват**). Накрая ще разгледаме техники за **дебъгване**, чрез които последователно да проследяваме пътя, който извървява нашата програма по време на своето изпълнение.

Видео

Гледайте видео урок по учебния материал от настоящата глава от книгата: <https://www.youtube.com/watch?v=mCacBknnYKQ>.

Сравняване на числа

В програмирането можем да сравняваме стойности чрез следните **оператори**:

- Оператор **<** (по-малко)
- Оператор **>** (по-голямо)
- Оператор **<=** (по-малко или равно)
- Оператор **>=** (по-голямо или равно)
- Оператор **==** (равно)
- Оператор **!=** (различно)

При сравнение резултатът се свежда до булева стойност – **true (1)** или **false (0)**, в зависимост от това дали резултатът от сравнението е истина или лъжа.

Примери за сравнение на числа

```
int a = 5;
int b = 10;

cout << boolalpha << (a < b);      // true
cout << boolalpha << (a > 0);      // true
cout << boolalpha << (a > 100);    // false
cout << boolalpha << (a < a);      // false
cout << boolalpha << (a <= 5);     // true
cout << boolalpha << (b == 2 * a);  // true
```

Обърнете внимание, че за да могат да се отпечатат с оператора **cout** булевите стойности **true** и **false**, трябва да се извика функцията **boolalpha**. В противен случай ще се отпечата **1** за истина или **0** за лъжа.

Сравнение на числа от типове float и double

Когато сравняваме числа от целочислени типове, нещата са малко или много еднопосочни. Трябва да се има в предвид, обаче, една особеност при сравнението на числа от типове с плаваща запетая. Нека да вземем следния пример, който ще доведе до неочекван резултат:

```
double d1 = 100 - 99.99;
double d2 = 10 - 9.99;

cout << boolalpha << (d1 == d2) << endl;
```

Може да пуснете в действие и тествате кода от горния пример онлайн:

<https://repl.it/@vncpetrov/ComparingFloatsAndDoubles>.

При изпълнението на горната програма бихме очаквали да се отпечата **true**, защото очакваме стойностите на променливите **d1** и **d2** да равни (0.01), но вместо това тя отпечатва **false**. Това е така, заради това, че тези числа се изчисляват до стойност, достатъчно близка до 0.01, но **не са точно** 0.01 - **d1** е равно на 0.010000000000005116, а **d2** - 0.009999999999997868. Може да установим това, ако отпечатаме числата с достатъчно голяма точност след десетичната точка:

```
double d1 = 100 - 99.99;
double d2 = 10 - 9.99;

cout << setprecision(17) << d1 << endl;
cout << setprecision(17) << d2 << endl;
```

Може да пуснете в действие и тествате кода от горния пример онлайн:

<https://repl.it/@vncpetrov/ComparingFloatsAndDoubles2>.

Прости проверки

В програмирането често проверяваме **дадени условия** и извършваме различни действия, спрямо резултата от тези проверки. Проверките извършваме посредством **if** клаузи, които имат следната конструкция:

```
if (булев израз) {
    // тяло на условната конструкция;
}
```

Пример: отлична оценка

Въвеждаме оценка (като десетична дроб) в конзолата и проверяваме дали тя е **отлична (≥ 5.50)**.

```

double grade;
cin >> grade;

if (grade >= 5.50) {
    cout << "Excellent!" << endl;
}

```

Тествайте кода от примера локално. Опитайте да въведете различни оценки, например 4.75, 5.49, 5.50 и 6.00. При оценки по-малки от 5.50 програмата няма да изведе нищо, а при оценка 5.50 или по-голяма, ще изведе "Excellent!".

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни: <https://judge.softuni.bg/Contests/Practice/Index/1360#0>.

Проверки с if-else конструкция

Конструкцията **if** може да съдържа и **else** клауза, с която да окажем конкретно действие в случай, че булевият израз (който е зададен в началото **if (булев израз)**) върне отрицателен резултат (**false**). Така построена, **условната конструкция** наричаме **if-else** и поведението ѝ е следното: ако резултатът от условието е **позитивен (true)** - извършваме едни действия, а когато е **негативен (false)** - други. Форматът на конструкцията е:

```

if (булево условие) {
    // тяло на условната конструкция;
}
else {
    // тяло на else-конструкция;
}

```

Пример: отлична оценка или не

Подобно на горния пример, въвеждаме оценка, проверяваме дали е отлична, но изписваме резултат и в двета случая.

```

double grade;
cin >> grade;

if (grade >= 5.50) {
    cout << "Excellent!" << endl;
}
else {
    cout << "Not excellent." << endl;
}

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1360#1>.

За къдравите скоби { } след if / else

Когато имаме само една команда в тялото на **if** конструкцията, можем да пропуснем къдравите скоби, обозначаващи тялото на условния оператор. Когато искаме да изпълним блок от код (група команди), къдравите скоби са **задължителни**. В случай че ги изпуснем, ще се изпълни само първият ред след **if** клаузата.



Добра практика е, **винаги да слагаме къдрави скоби**, понеже това прави кода ни по-четим и по-подреден.

Ето един пример, в който изпускането на къдравите скоби води до объркване:

```
string color = "red";
if (color == "red")
    cout << "tomato" << endl;
else if (color == "yellow")
    cout << "banana" << endl;
cout << "lemon" << endl;
```

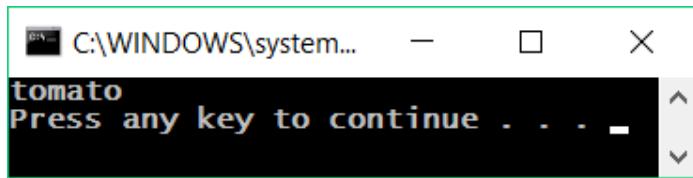
Изпълнението на горния код ще изведе следния резултат на конзолата:

```
C:\WINDOWS\system3...
tomato
lemon
Press any key to continue . . .
```

С къдрави скоби:

```
string color = "red";
if (color == "red") {
    cout << "tomato" << endl;
}
else if (color == "yellow") {
    cout << "banana" << endl;
    cout << "lemon" << endl;
}
```

На конзолата ще бъде отпечатано следното:



Обърнете внимание, че за да може да се декларират и да се работи с променливи от тип **string**, трябва в началото на програмата да се добави и следният ред: **#include <string>**.



Абсолютно и двата начина на използване са **правилни** и може да се използват в зависимост от конкретния случай, изискванията в документацията и/или други изисквания, но абсолютно винаги трябва да се внимава и да се съобразим с резултатите.

Пример: четно или нечетно

Да се напише програма, която проверява, дали дадено цяло число е **четно** (even) или **нечетно** (odd).

Задачата можем да решим с помощта на една **if-else** конструкция и оператора **%**, който връща **остатък при деление** на две числа:

```
int num;
cin >> num;

if (num % 2 == 0) {
    cout << "Even" << endl;
}
else {
    cout << "Odd" << endl;
}
```

Тестване в Judge системата

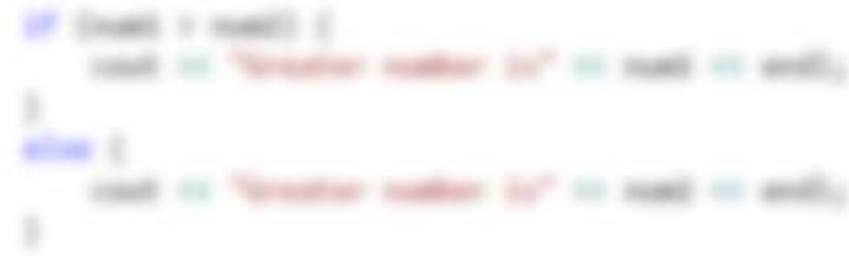
Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1360#2>.

Пример: по-голямото число

Да се напише програма, която чете две цели числа и извежда по-голямото от тях.

Първата ни подзадача е да **прочетем** двете числа. След което, чрез проста **if-else** конструкция, в съчетание с **оператора за по-голямо (>)**, да направим проверка. Кодът е замъглен умислено и трябва да бъде довършен от читателя:

```
int num1, num2;
cin >> num1;
cin >> num2;
```



Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1360#3>.

Живот на променлива

Всяка една променлива си има обхват, в който съществува, наречен **variable scope**. Този обхват уточнява къде една променлива може да бъде използвана. В езика C++ областта, в която една променлива съществува, започва от реда, на който сме я **дeфинириали** и завършва до първата затваряща къдрава скоба **}** (на метода, на **if** конструкцията и т.н.). За това е важно да знаем, че **всяка променлива, дефинирана вътре в тялото на if, няма да бъде достъпна извън него**, освен ако не сме я дефинирали по-нагоре в кода.

В примера по-долу, на последния ред, на който се опитваме да отпечатаме променливата **salary**, която е дефинирана в **if** конструкцията, ще получим грешка, защото нямаме достъп до нея (в случай и самото IDE ни предупреждава за **variable scope**):

```
int myMoney = 500;
int payDayDate = 7;
int todayDate = 10;

if (todayDate > payDayDate) {
    int salary = 5000;
    myMoney = myMoney + salary;
}

cout << myMoney << endl;
cout << salary << endl; // Error!
```

Серии от проверки

Понякога се налага да извършим серия от проверки, преди да решим какви действия ще изпълнява нашата програма. В такива случаи, можем да приложим конструкцията **if-else if...-else** в серия. За целта използваме следния формат:

```

if (условие) {
    // тяло на условната конструкция;
}
else if (условие2) {
    // тяло на условната конструкция;
}
else if (условие3) {
    // тяло на условната конструкция;
}
...
else {
    // тяло на else-конструкция;
}

```

Пример: число от 1 до 9 на английски

Да се изпише число в интервала от 1 до 9 с текст на английски език (числото се чете от конзолата). Можем да прочетем числото и след това чрез **серия от проверки** отпечатваме съответстващата му английска дума:

```

int num;
cin >> num;

if (num == 1) {
    cout << "one" << endl;
}
else if (num == 2) {
    cout << "two" << endl;
}
else if (...) {
    ...
}
else if (num == 9) {
    cout << "nine" << endl;
}
else {
    cout << "number too big" << endl;
}

```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/NumberToText>.

Програмната логика от примера по-горе **последователно сравнява** входното число от конзолата с цифрите от 1 до 9, като **всяко следващо сравнение се извършва, само в случай че предходното сравнение не е било истина**. В крайна сметка, ако никое от **if** условията не е изпълнено, се изпълнява последната **else** **клause**.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1360#4>.

Упражнения: прости проверки

За да затвърдим знанията си за условните конструкции **if** и **if-else**, ще решим няколко практически задачи.

Задача: бонус точки

Дадено е **цяло число** – брой точки. Върху него се начисляват **бонус точки** по правилата, описани по-долу. Да се напише програма, която пресмята **бонус точките** за това число и **общия брой точки** с бонусите.

- Ако числото е **до 100** включително, бонус точките са 5.
- Ако числото е **по-голямо от 100**, бонус точките са **20%** от числото.
- Ако числото е **по-голямо от 1000**, бонус точките са **10%** от числото.
- Допълнителни бонус точки (начисляват се отделно от предходните):
 - За **четно** число -> + 1 т.
 - За число, което **завършва на 5** -> + 2 т.

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
20	6	175	37	2703	270.3 2973.3	15875	1589.5 17464.5
	26		212				

Насоки и подсказки

Основните и допълнителните бонус точки можем да изчислим с поредица от няколко **if-else-if-else** проверки. Като за **основните бонус точки имаме 3 случая** (когато въведеното число е до 100, между 100 и 1000 и по-голямо от 1000), а за **допълнителните бонус точки - още 2 случая** (когато числото е четно и нечетно):

```

int num;
cin >> num;
float bonusScore = 0.0f;

if (num > 1000) {
    bonusScore = num * 0.10f;
}
else // TODO: Write more logic here...
  
```

```

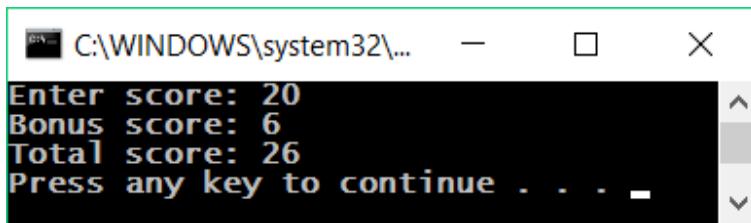
if (num % 10 == 5) {
    bonusScore += 2;
}
else // TODO: Write more logic here...

// bonus score
cout << bonusScore << endl;

// total score
cout << num + bonusScore << endl;

```

Ето как би могло да изглежда решението на задачата в действие:



Обърнете внимание, че за тази задача Judge е настроен да игнорира всичко, което не е число, така че можем да печатаме не само числата, но и уточняващ текст.

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1360#5>.

Задача: сумиране на секунди

Трима спортни състезатели финишират за някакъв **брой секунди** (между 1 и 50). Да се напише програма, която въвежда времената на състезателите и пресмята **сумарното им време** във формат "минути:секунди". Секундите да се изведат с водеща нула (2 -> "02", 7 -> "07", 35 -> "35").

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
35		22		50		14	
45	2:04	7	1:03	50	2:29	12	0:36
44		34		49		10	

Насоки и подсказки

Първо сумираме трите числа, за да получим общия резултат в секунди. Понеже **1 минута = 60** секунди, ще трябва да изчислим броя минути и броя секунди в диапазона от 0 до 59:

- Ако резултатът е между 0 и 59, отпечатваме 0 минути + изчислените секунди.
- Ако резултатът е между 60 и 119, отпечатваме 1 минута + изчислените секунди минус 60.
- Ако резултатът е между 120 и 179, отпечатваме 2 минути + изчислените секунди минус 120.
- Ако секундите са по-малко от 10, извеждаме водеща нула преди тях.

```
int firstCompetitor;
cin >> firstCompetitor;
// TODO: Read also second and third competitors' time

int seconds = firstCompetitor + secondCompetitor + thirdCompetitor;
int minutes = 0;

if (seconds > 59) {
    minutes++;
    seconds = seconds - 60;
}
// TODO: Write more logic here...

if (seconds < 10) {
    cout << minutes << ":" << "0" << seconds << endl;
}
else {
    cout << minutes << ":" << seconds << endl;
}
```

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1360#6>.

Задача: конвертор за мерни единици

Да се напише програма, която преобразува **разстояние** между следните **8 мерни единици**: **m, mm, cm, mi, in, km, ft, yd**. Използвайте съответствията от таблицата по-долу:

Входна единица	Изходна единица
1 meter (m)	1000 millimeters (mm)
1 meter (m)	100 centimeters (cm)
1 meter (m)	0.000621371192 miles (mi)
1 meter (m)	39.3700787 inches (in)
1 meter (m)	0.001 kilometers (km)
1 meter (m)	3.2808399 feet (ft)
1 meter (m)	1.0936133 yards (yd)

Входните данни се състоят от три реда:

- Първи ред: число за преобразуване.
- Втори ред: входна мерна единица.
- Трети ред: изходна мерна единица (за резултата).

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
12 km ft	39370.0788	150 mi in	9503999.99393599	450 yd km	0.41147999937455

Насоки и подсказки

Ако желаем програмата да може да разпознава както главни, така и малки букви при прочитането на мерните единици, то може да се използва функцията **transform(...)** (от namespace std), която да трансформира всички букви в долн регистър:

```
transform(data.begin(), data.end(), data.begin(), ::tolower);
```

За да можете да използвате тази функция, трябва да включим библиотеката **algorithm** в началото на програмата:

```
#include <algorithm>
```

Нека не забравяме също да включим и **string** файла, за да може да декларираме и използваме променливи от тип **string**.

Както виждаме от таблицата в условието, можем да конвертираме само **между**

метри и някаква друга мерна единица. Следователно трябва първо да изчислим числото за преобразуване в метри. Затова трябва да направим набор от проверки, за да определим каква е входната мерна единица, а след това и за изходната мерна единица:

```
double size;
cin >> size;

string sourceMetric;
getline(cin, sourceMetric);
transform(sourceMetric.begin(), sourceMetric.end(),
         sourceMetric.begin(), ::tolower);

string destMetric;
getline(cin, destMetric);
transform(destMetric.begin(), destMetric.end(),
         destMetric.begin(), ::tolower);

if (sourceMetric == "km") {
    size = size / 0.001;
}
// TODO: Check the other metrics: mm, cm, ft, yd

if (destMetric == "ft") {
    size = size * 3.2808399;
}
// TODO: Check the other metrics: mm, cm, ft, yd

cout << size << " " << destMetric << endl;
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1360#7>.

Дебъгване - прости операции с дебъгер

До момента писахме доста код и често пъти в него имаше грешки, нали? Сега ще покажем един инструмент, с който можем да намираме грешките по-лесно.

Какво е "дебъгване"?

Дебъгване е процесът на "закачане" към изпълнението на програмата, който ни позволява да проследим поетапно процеса на изпълнение. Можем да следим **ред по ред** какво се случва с нашата програма, какъв път следва, какви стойности имат

дeфинираните променливи на всяка стъпка от изпълнението на програмата и много други неща, които ни позволяват да откриваме грешки (**бъгове**):

```

1 #include "pch.h"
2 #include <iostream>
3
4 using namespace std;
5
6 int main() {
7     int number;
8     cin >> number;
9
10    number += number;
11    cout << number << endl;
12
13    return 0;
14 }

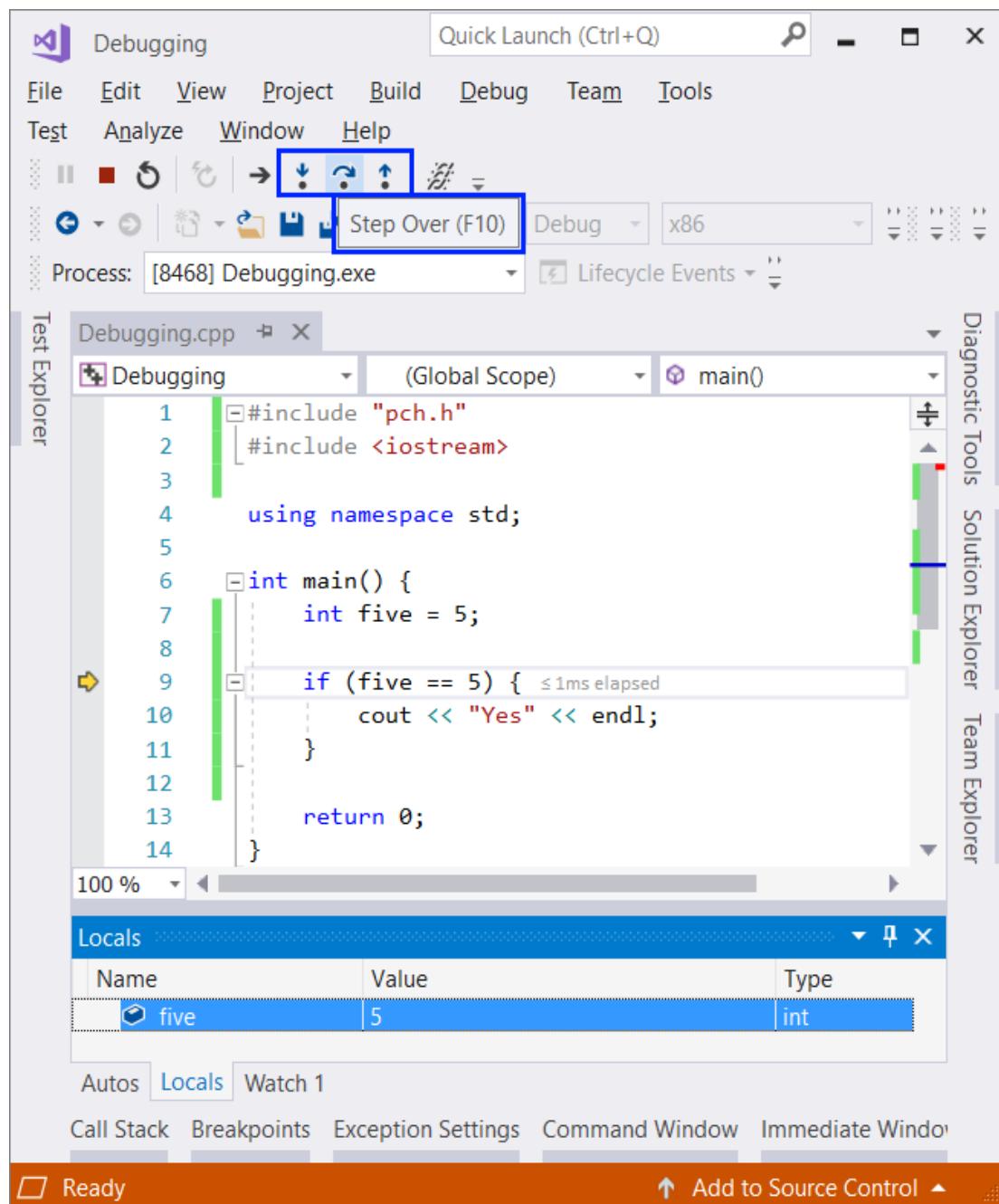
```

Name	Value	Type
number	5	int

Дебъгване във Visual Studio

Чрез натискане на бутона [F10], стартираме програмата в Debug режим. Преминаваме към **следващия ред** отново с [F10]. (вж. стр. 118).

Чрез [F9] създаваме стопери – така наречените **breakpoints**, до които можем да стигнем директно, използвайки [F5] при стартирането на програмата. Тествайте описаните действия, за да придобиете пълна представа, както и да си изградите един полезен навик.



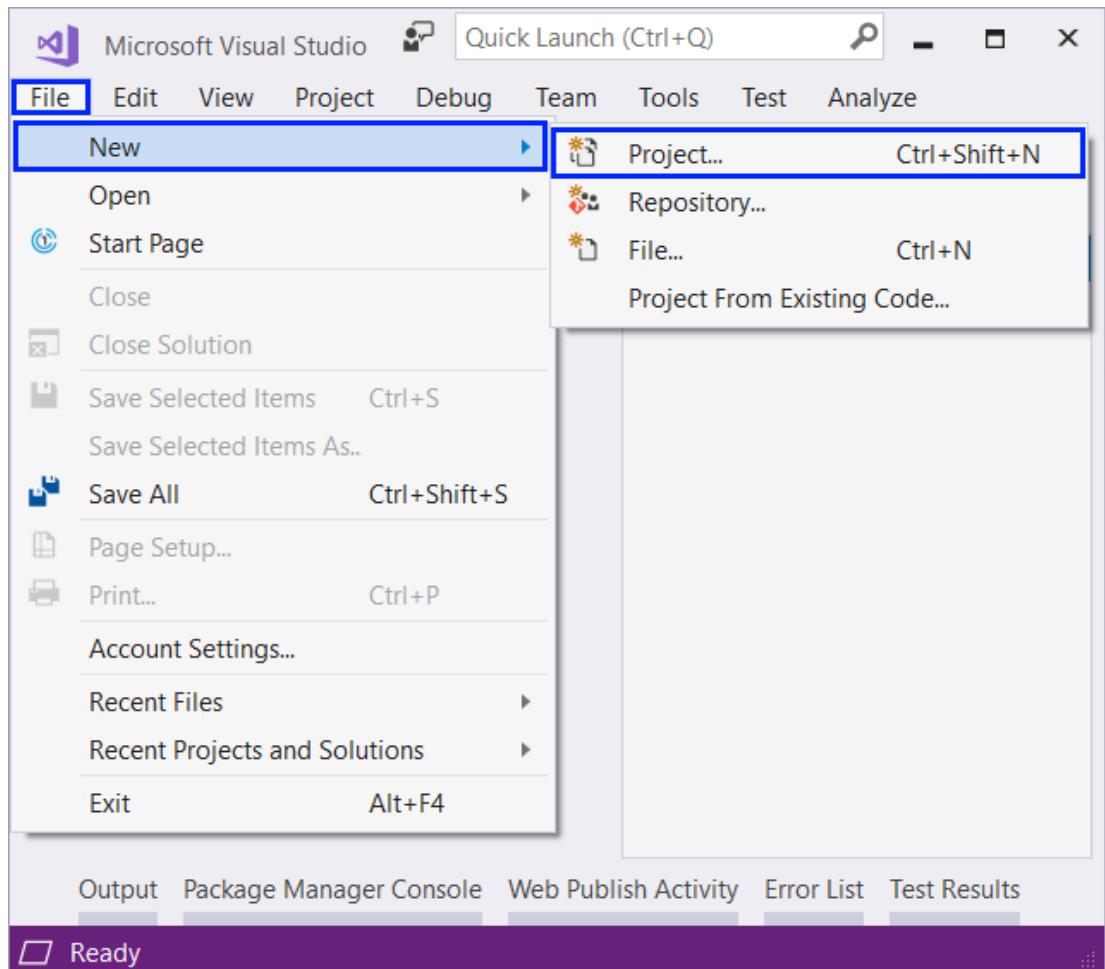
Упражнения: прости проверки

Нека затвърдим наученото в тази глава с няколко задачи.

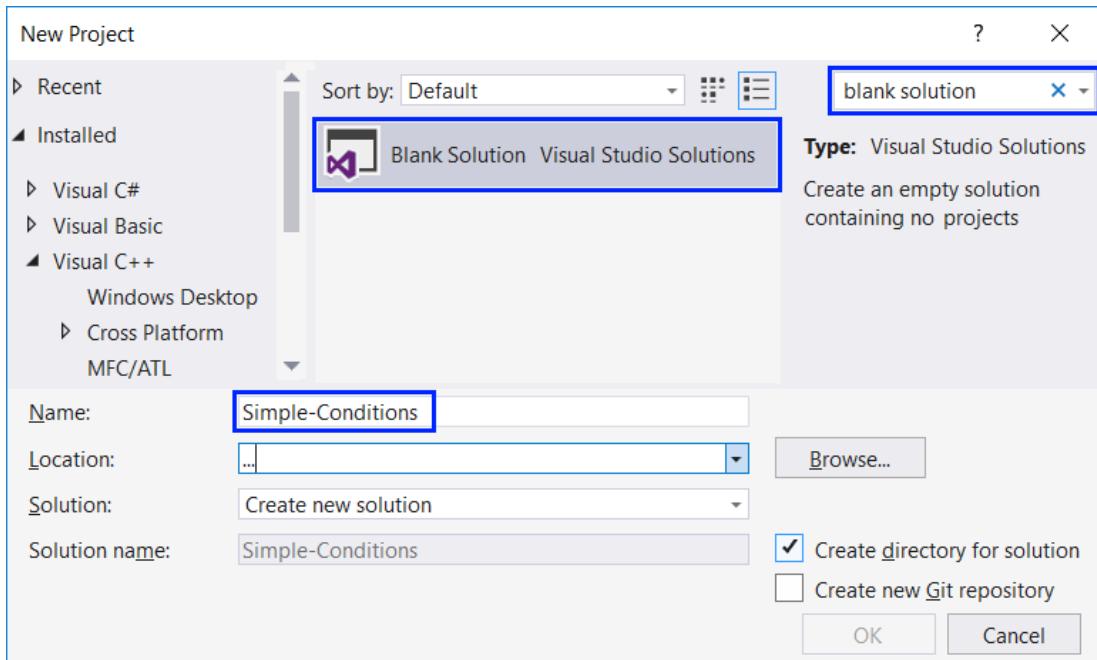
Празно Visual Studio решение (Blank Solution)

Нека си припомним как се създава празно решение (Blank Solution) във Visual Studio, за да организираме по-добре решенията на задачите от упражненията – всяка задача ще бъде в отделен проект и всички проекти ще бъдат в общ Solution.

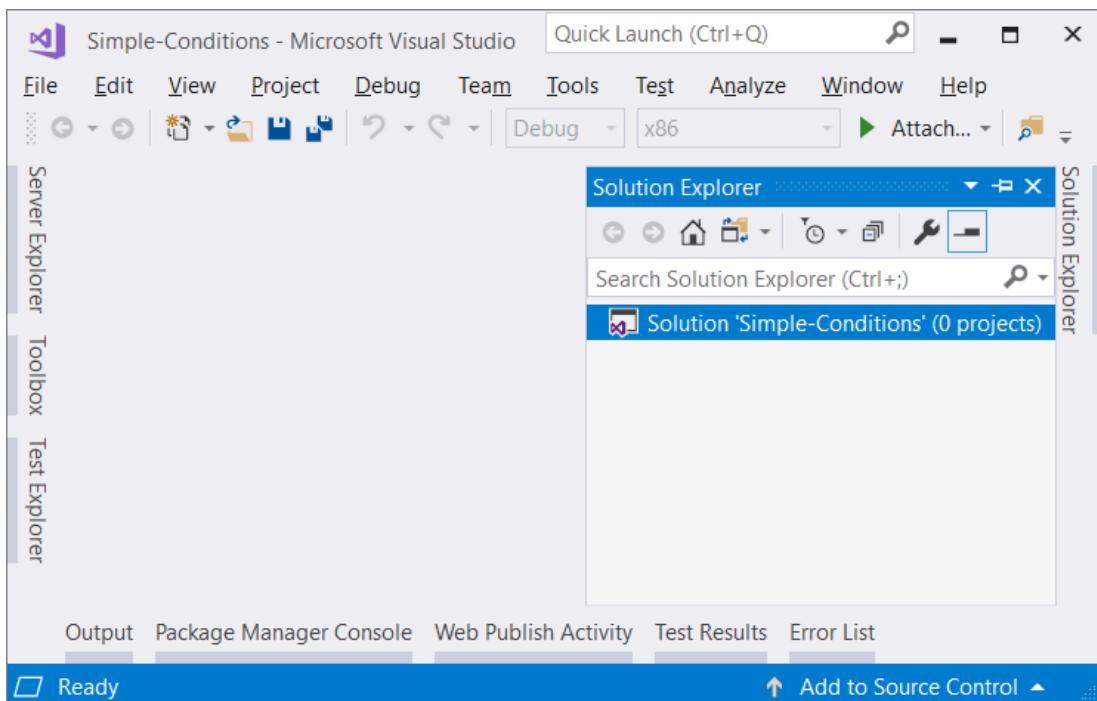
Стартираме Visual Studio. Създаваме нов Blank Solution от [File] -> [New] -> [Project...]:



Избираме от диалоговия прозорец [Templates] -> [Other Project Types] -> [Visual Studio Solutions] -> [Blank Solution] или използваме търсачката и даваме подходящо име на проекта, например "Simple-Conditions":



Сега имаме създаден празен Visual Studio Solution (без проекти в него):



Задача: проверка за отлична оценка

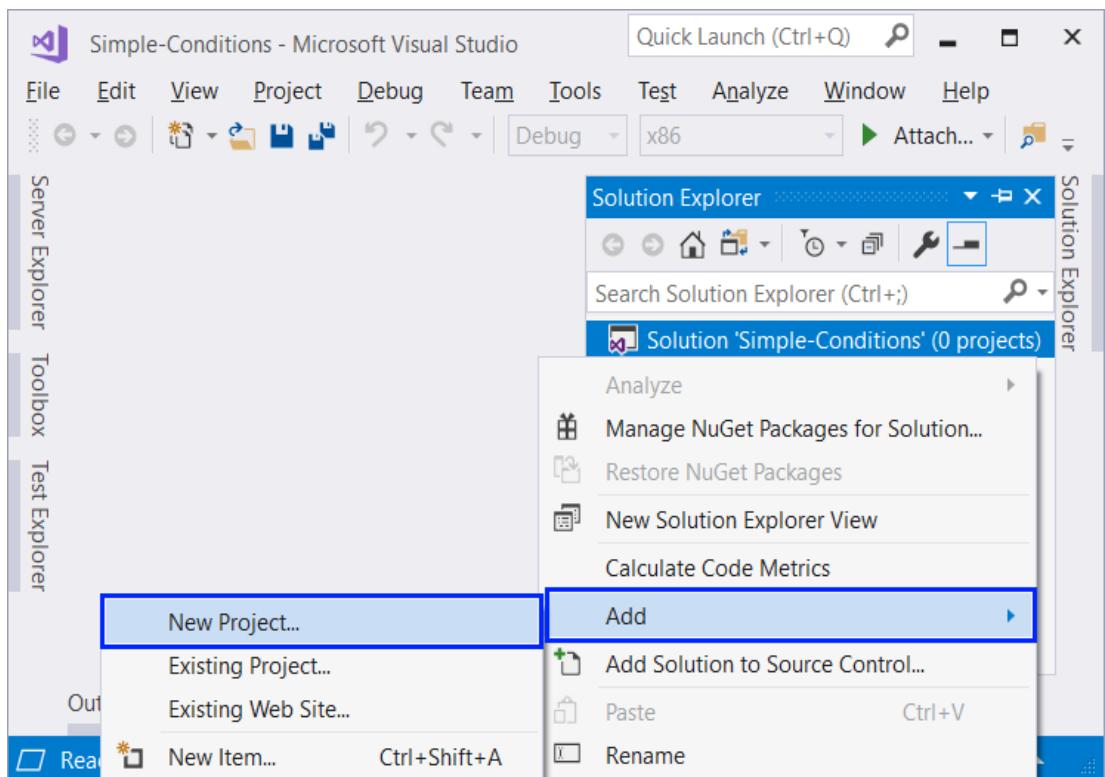
Първата задача от упражненията за тази тема е да се напише конзолна програма, която въвежда оценка (десетично число) и отпечатва "Excellent!", ако оценката е 5.50 или по-висока.

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
6	Excellent!	5.5	Excellent!	5.49	(няма изход)

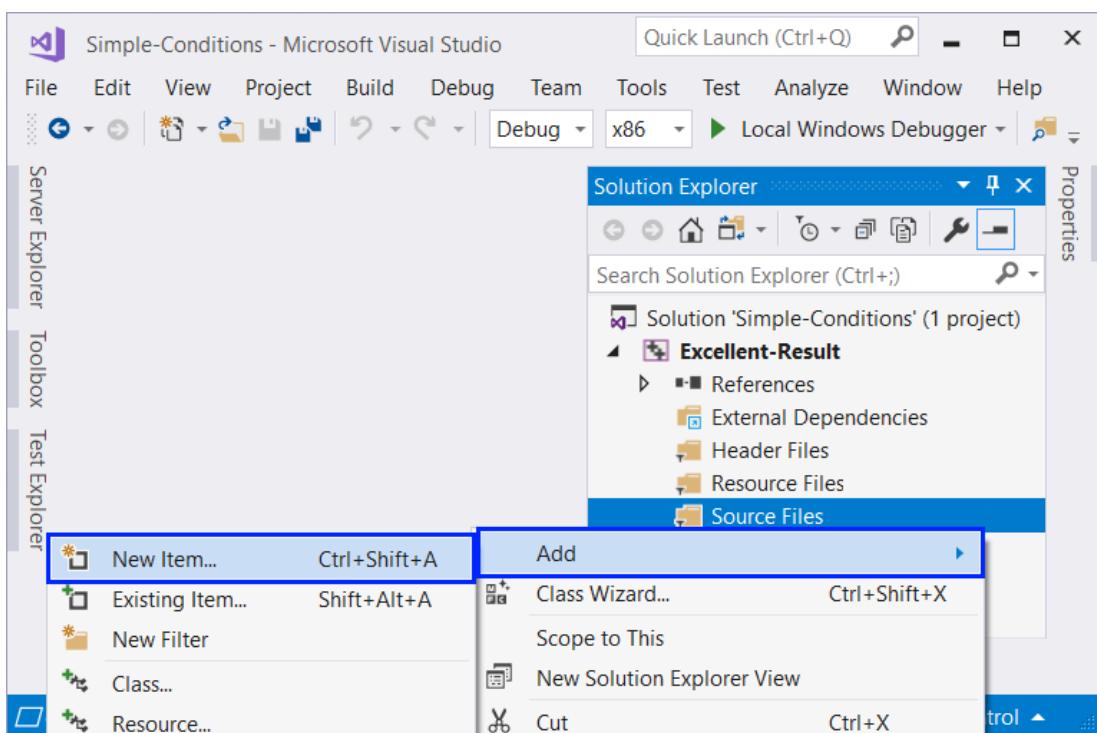
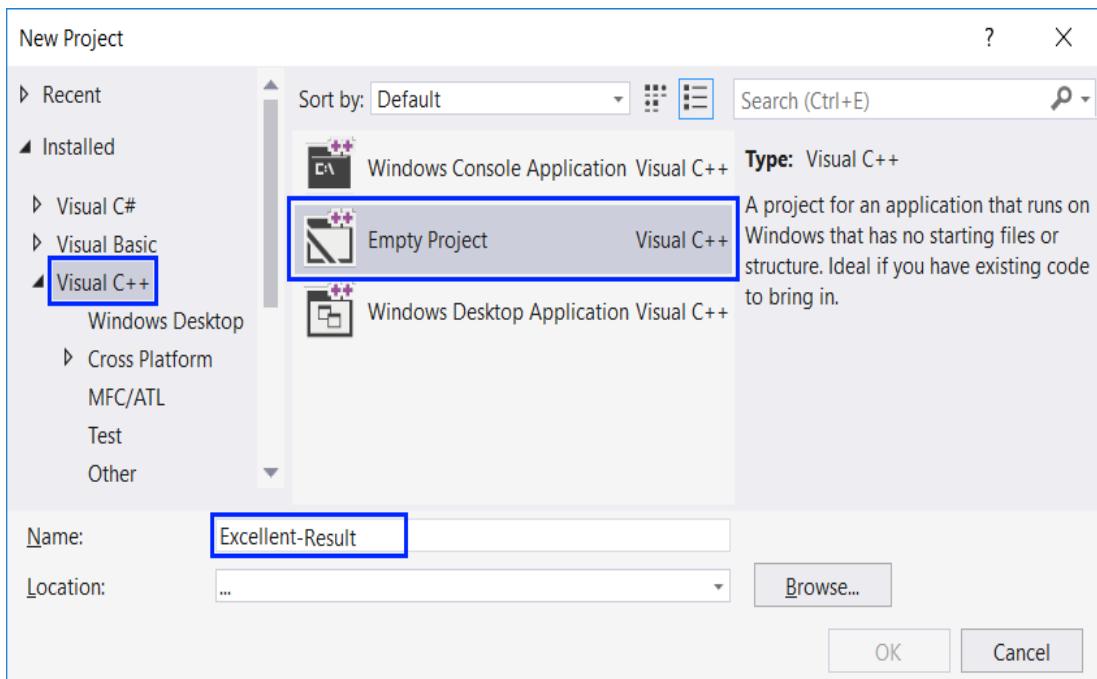
Насоки и подсказки

Създаваме нов проект в съществуващото Visual Studio решение. В Solution Explorer кликваме с десен бутон на мишката върху Solution 'Simple-Conditions'. Избираме [Add] -> [New Project...]:



Ще се отвори диалогов прозорец за избор на тип проект за създаване. Избираме от дървото в ляво [Visual C++], а от централния списък избираме [Empty Project] и задаваме име, например "Excellent-Result".

След това добавяме нов C++ файл (.cpp), който трябва да се казва Main.cpp (от [Source Files] -> [Add] -> [New Item...]).



Вече имаме Solution с едно конзолно приложение в него. Остава да напишем кода за решаване на задачата:

```

double grade;
cin >> grade;

if (grade >= 5.50) {
    cout << "Excellent!" << endl;
}

```

Стартираме програмата с [Ctrl + F5], за да я тестваме с различни входни стойности:

```

C:\WINDOWS\system32\cmd.exe - X
5.25
Press any key to continue . . .

C:\WINDOWS\system32\cmd.e... - X
5.6
Excellent!
Press any key to continue . . .

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1360#0>.

Задача: отлична оценка или не

Следващата задача от тази тема е да се напише конзолна програма, която въвежда оценка (десетично число) и отпечатва "Excellent!", ако оценката е 5.50 или по-висока, или "Not excellent." в противен случай.

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
6	Excellent!	5	Not Excellent!	5.49	Not excellent.

Насоки и подсказки

Първо създаваме нов C++ конзолен проект в решението "Simple-Conditions" с име "Excellent-or-Not".

Следва да напишем кода на програмата. Може да си помогнем с примерния код от картинката:

```

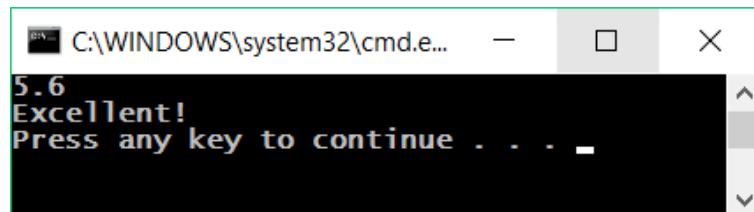
double grade;
cin >> grade;

if (grade >= 5.50) {
    cout << "Excellent!" << endl;
}
else {
    cout << "Not excellent." << endl;
}

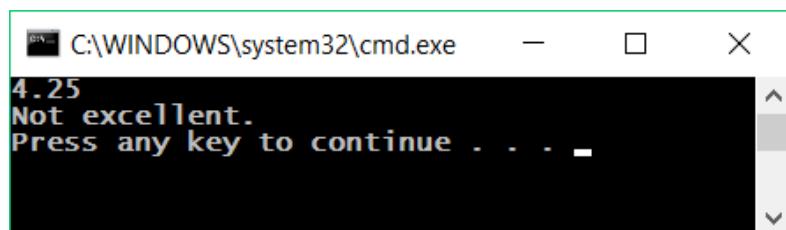
```

Спомняме си, че трябва да включим автоматичното превключване към текущия проект, което обяснихме подробно в глава [2.1 Прости пресмятания](#). За целта трябва да кликнем върху главния Solution с десния бутон на мишката и изберем [Set StartUp Projects...]. Ще се появи диалогов прозорец, от който трябва да се избере [Startup Project] -> [Current selection].

Сега **стартираме програмата**, както обикновено с [Ctrl + F5] и я тестваме дали работи коректно:



```
C:\WINDOWS\system32\cmd.exe
5.6
Excellent!
Press any key to continue . . .
```



```
C:\WINDOWS\system32\cmd.exe
4.25
Not excellent.
Press any key to continue . . .
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1360#1>.

Submissions		
Points	Time and memory used	Submission date
4/4 100 / 100	Memory: 1.82 MB Time: 0.015 s	12:11:03 05.02.2019
		Details

Задача: четно или нечетно

Да се напише програма, която въвежда **цяло число** и печата дали е **четно** или **нечетно**.

Примерен вход и изход

Вход	Изход
2	even

Вход	Изход
3	odd

Вход	Изход
25	odd

Вход	Изход
25	odd

Насоки и подсказки

Отново, първо добавяме **нов C++ конзолен проект** в съществуващия Solution. Проверката дали дадено число е четно, може да се реализира с оператора **%**, който ще ни върне **остатъка при целочислено деление на 2** по следния начин: **int isEven = (num % 2 == 0)**.

Остава да **стартираме** програмата с **[Ctrl + F5]** и да я тестваме:

```
C:\WINDOWS\system32...
42
even
Press any key to continue . . .
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1360#2>.

Задача: намиране на по-голямото число

Да се напише програма, която въвежда **две цели числа** и отпечатва по-голямото от двете.

Примерен вход и изход

Вход	Изход
5 3	5

Вход	Изход
3 5	5

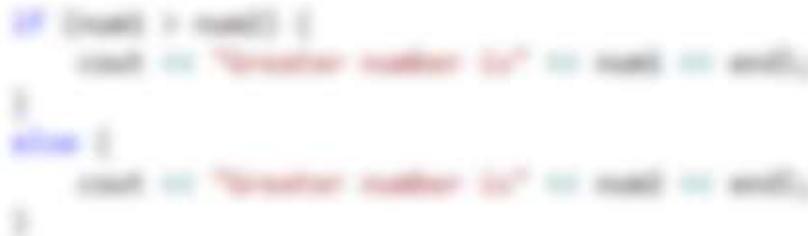
Вход	Изход
10 10	10

Вход	Изход
-5 5	5

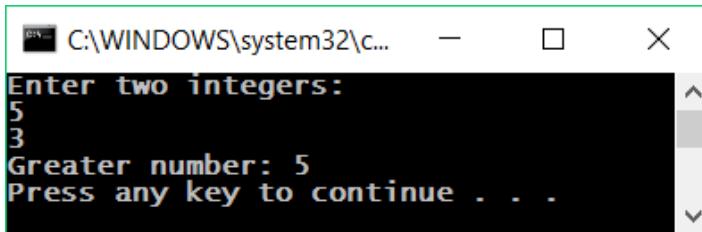
Насоки и подсказки

Както обикновено, първо трябва да добавим **нов C++ конзолен проект** в съществуващия Solution. За кода на програмата ни е необходима единична **if-else** конструкция. Може да си помогнете частично с кода от картинката, който е умислено замъглен, за да помислите как да допишете сами:

```
int num1, num2;
cin >> num1;
cin >> num2;
```



След като сме готови с имплементацията на решението, **стартираме** програмата с [Ctrl + F5] и я тестваме:



```
C:\WINDOWS\system32\c...
Enter two integers:
5
3
Greater number: 5
Press any key to continue . . .
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1360#3>.

Задача: изписване на число до 9 с думи

Да се напише програма, която въвежда **цяло** число в диапазона [0 ... 9] и го **изписва с думи** на английски език. Ако числото е извън диапазона, изписва "number too big".

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
5	five	1	one	9	nine	10	number too big

Насоки и подсказки

Може да използваме поредица от **if-else** конструкции, с които да разгледаме възможните **11** случая.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1360#4>.

Задача: познай паролата

Да се напише програма, която **въвежда парола** (един ред с произволен текст) и проверява дали въведеното **съвпада** с фразата "s3cr3t!P@sswOrd". При съответствие да се изведе "Welcome", а при несъответствие - "Wrong password!".

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
qwerty	Wrong password!	s3cr3t!P@sswOrd	Welcome	s3cr3t!p@ss	Wrong password!

Насоки и подсказки

Може да използваме **if-else** конструкцията.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1360#8>.

Задача: число от 100 до 200

Да се напише програма, която **въвежда цяло число** и проверява дали е **под 100**, **между 100 и 200** или **над 200**. Да се отпечатат съответно съобщения, като в примерите по-долу.

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
95	Less than 100	120	Between 100 and 200	210	Greater than 200

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1360#9>.

Задача: еднакви думи

Да се напише програма, която **въвежда две думи** и проверява дали са еднакви. Да не се прави разлика между главни и малки букви. Да се изведе "yes" или "no".

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
Hello Hello	yes	SoftUni softuni	yes	Soft Uni	no	beer vodka	no

Насоки и подсказки

Преди сравняване на думите, трябва да ги обърнем в долен регистър, за да не оказва влияние размера на буквите (главни/малки). За целта, ще използваме функцията **transform(...)**, която да направи всички букви малки. Функцията има следния формат:

```
transform(data.begin(), data.end(), data.begin(), ::tolower);
```

За да можем да използваме тази функция, трябва да включим **algorithm** файла в началото на програмата: **#include <algorithm>**.

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1360#10>.

Задача: информация за скоростта

Да се напише програма, която **въвежда** скорост (десетично число) и отпечатва **информация** за скоростта:

- При скорост **до 10** (включително), отпечатайте "slow".
- При скорост **над 10 и до 50**, отпечатайте "average".
- При скорост **над 50 и до 150**, отпечатайте "fast".
- При скорост **над 150 и до 1000**, отпечатайте "ultra fast".
- При по-висока скорост, отпечатайте "extremely fast".

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
8	slow	126	fast	3500	extremely fast
49.5	average	160	ultra fast		

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1360#11>.

Задача: лица на фигури

Да се напише програма, която **въвежда** размерите на геометрична фигура и **пресмята лицето** ѝ. Фигурите са четири вида: квадрат (**square**), правоъгълник (**rectangle**), кръг (**circle**) и триъгълник (**triangle**).

На първия ред на входа се чете вида на фигурата (**square**, **rectangle**, **circle**, **triangle**).

- Ако фигурата е **квадрат**, на следващия ред се чете едно число – дължина на страната му.
- Ако фигурата е **правоъгълник**, на следващите два реда се четат две числа – дължините на страните му.
- Ако фигурата е **кръг**, на следващия ред се чете едно число – радиуса на кръга.
- Ако фигурата е **триъгълник**, на следващите два реда се четат две числа – дължината на страната му и дължината на височината към нея.

Резултатът да се закръгли до 3 цифри след десетичния знак.

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
square 5	25	rectangle 7 2.5	17.5	circle 6	113.097	triangle 4.5 20	45

Насоки и подсказки

Закръглянето на изходната стойност може да извършим, подавайки последователно към **cout** функциите **fixed** и **setprecision(n)**, където с **n** указваме до кой знак след десетичната точка да се извърши закръглянето. Включваме и следната библиотека в началото на нашата програма: **#include <iomanip>**.

Ето примерна употреба, при което като изход ще бъде изведенено 3.142:

```
double f = 3.14159;
cout << fixed << setprecision(3) << f << endl;
```

Може да пуснете в действие и тествате примера онлайн:

<https://repl.it/@vncpetrov/SampleRounding>.

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1360#12>

Задача: време + 15 минути

Да се напише програма, която **въвежда час и минути** от 24-часово денонощие и изчислява колко ще е **часът след 15 минути**. Резултатът да се отпечата във

формат **hh:mm**. Часовете винаги са между 0 и 23, а минутите винаги са между 0 и 59. Часовете се изписват с една или две цифри. Минутите се изписват винаги с по две цифри и с **водеща нула**, когато е необходимо.

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
1 46	2:01	0 01	0:16	23 59	0:14	11 08	11:23

Насоки и подсказки

Добавете 15 минути и направете няколко проверки. Ако минутите надвишат 59, **увеличете часовете** с 1 и **намалете минутите** с 60. По аналогичен начин разгледайте случая, когато часовете надвишат 23. При печатането на минутите, проверете за водеща нула.

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1360#13>.

Задача: еднакви 3 числа

Да се напише програма, в която се въвеждат **3 числа** и се отпечатва дали те са **еднакви** (yes / no).

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
5 5 5	yes	5 4 5	no	1 2 3	no

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1360#14>

Задача: * изписване на число от 0 до 100 с думи

Да се напише програма, която превръща число в диапазона [0 ... 100] в текст. (на английски език).

Примерен вход и изход

Вход	Изход
25	twenty five

Вход	Изход
42	forty two

Вход	Изход
6	six

Насоки и подсказки

Проверете първо за **едноцифриeni числа** и ако числото е едноцифрено, отпечатайте съответната дума за него. След това проверете за **двуцифриени числа**. Тях отпечатавайте на две части: лява част (**десетици** = числото / 10) и дясна част (**единици** = числото % 10). Ако числото има 3 цифри, трябва да е 100 и може да се разгледа като специален случай.

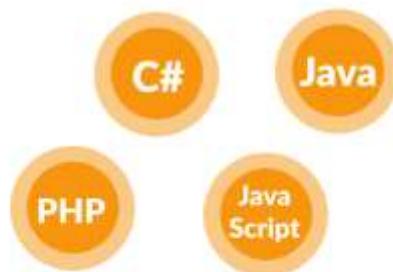
Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1360#15>.

Качествено образование,
професия и работа за
Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



СофТУни предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата **"Софтуерен университет"** изгражда истински **профессионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на СофТУни, за да усвоите **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

СофТУни работи пряко с компаниите от софтуерната индустрия, съдействайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ



1 - 1.5 години



1.5 - 2 години



Programming Basics

Кариерен Старт

Дипломиране

70/100 credits

80/100/150 credits

Глава 3.2. Прости проверки – изпитни задачи

В предходната глава разгледахме условните конструкции в езика C++, чрез които можем да изпълняваме различни действия в зависимост от някакво условие. Споменахме още какъв е обхватът на една променлива (нейният **scope**), както и как последователно да проследяваме изпълнението на нашата програма (т.нар. **дебъгване**). В настоящата глава ще упражним работата с логически проверки, като разгледаме някои задачи, давани на изпити. За целта нека първо си припомним конструкцията на логическата проверка:

```
if (булев израз) {  
    // тяло на условната конструкция;  
}  
else {  
    // тяло на else-конструкция;  
}
```

if проверките се състоят от:

- **if** клауза
- булев израз - променлива от булев тип (**bool**) или булев логически израз (израз, който връща резултат **true/false**)
- тяло на конструкцията - съдържа произволен блок със сорс код
- **else** клауза и нейният блок със сорс код (**незадължително**)

Изпитни задачи

След като си припомнихме как се пишат условни конструкции, да решим няколко задачи, за да получим практически опит с **if-else** конструкцията.

Задача: цена за транспорт

Студент трябва да пропътува **n** километра. Той има избор между **три вида транспорт**:

- **Такси.** Начална такса: **0.70** лв. Дневна тарифа: **0.79** лв./км. Нощна тарифа: **0.90** лв./км.
- **Автобус.** Дневна / нощна тарифа: **0.09** лв./км. Може да се използва за разстояния минимум **20** км.
- **Влак.** Дневна / нощна тарифа: **0.06** лв./км. Може да се използва за разстояния минимум **100** км.

Напишете програма, която въвежда броя **километри n** и период от **дения**

(ден или нощ) и изчислява цената на най-евтиния транспорт.

Входни данни

От конзолата се четат **два реда**:

- Първият ред съдържа числото **n** – брой километри – цяло число в интервала [1 ... 5000].
- Вторият ред съдържа дума “**day**” или “**night**” – пътуване през деня или през нощта.

Изходни данни

Да се отпечата на конзолата **най-ниската цена** за посочения брой километри.

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
5 day	4.65	7 night	7	25 day	2.25	180 night	10.8

Насоки и подсказки

Ще прочетем входните данни и в зависимост от разстоянието ще изберем най-евтиния транспорт. За целта ще използваме няколко проверки.

Обработка на входните данни

В условието на задачата е дадена **информация за входа и изхода**. Съответно, първите **три реда** от решението ще съдържат декларирането и инициализирането на двете **променливи**, в които ще пазим **стойностите на входните данни**. За **първия ред е упоменато**, че съдържа **цяло число**, затова и променливата, която ще бъде декларирана, е от тип **int**. За **втория ред** указанietо е, че съдържа **дума**, съответно променливата е от тип **string**:

```
int distance;
string dayOrNight;
cin >> distance >> dayOrNight;
```

Преди да започнем проверките е нужно да **декларираме** и една **променлива**, в която ще пазим стойността на **цената за транспорт**:

```
double price = 0;
```

Извършване на проверки и съответните изчисления

След като вече сме декларирали и инициализирали входните данни и променливата, в която ще пазим стойността на цената, трябва да преценим кои

Условия от задачата да бъдат първи променливи.

От условието е видно, че тарифите на две от превозните средства **не зависят** от това дали е **ден или нощ**, но тарифата на единия превоз (такси) **зависи**. По тази причина **първата проверка** ще е именно дали е **ден или нощ**, за да стане ясно коя тарифа на таксито ще се **използва**. За целта **декларираме още една променлива**, в която ще пазим стойността на **тарифата на таксито**:

```
double taxiRate = 0;
```

За да определим **тарифата на таксито**, ще използваме проверка от типа **if-else**:

```
if (dayOrNight == "day") {
    taxiRate = 0.79;
}
else {
    taxiRate = 0.90;
}
```

След като е направено и това, вече може да пристъпим към изчислението на самата **цена за транспорта**. Ограниченията, които присъстват в условието на задачата, са относно **разстоянието**, което студента иска да пропътува. По тази причина, ще построим още една **if-else** конструкция, с чиято помощ ще определим **цената** за транспорта в зависимост от подадените километри:

```
if (distance < 20) {
    price = 0.70 + distance * taxiRate;
}
else if (distance < 100) {
    price = distance * 0.09;
}
else {
    price = distance * 0.06;
}
```

Първо правим проверка дали километрите са **под 20**, тъй като от условието е видно, че **под 20** километра студента би могъл да използва само **такси**. Ако условието на проверката е **вярно** (връща **true**), на променливата, която пази стойността на цената на транспорта (**price**), ще присвоим съответната стойност. Тази стойност е равна на **първоначалната такса**, която **събираме** с неговата **тарифа**, **умножена по разстоянието**, което студента трябва да измине.

Ако условието на променливата **не е вярно** (т.е. връща **false**), следващата стъпка е програмата ни да провери дали километрите са **под 100**. Правим това, защото от условието е видно, че в този диапазон може да се използва и **автобус** като транспортно средство. **Цената** за километър на автобуса е **по-ниска** от тази на таксито. Следователно, ако резултата от проверката е **верен**, то в блок тялото на

else if, на променливата за цената на транспорта (**price**) трябва да присвоим стойност, равна на резултата от умножението на **тарифата** на автобуса по **разстоянието**.

Ако и тази проверка **не върне true** като резултат, остава в тялото на **else** конструкцията, на променливата за цена да присвоим **стойност**, равна на **резултата от умножението на разстоянието по тарифата** на влака. Това се прави, тъй като влакът е **най-евтиния** вариант за транспорт при даденото разстояние.

Отпечатване на изходните данни

След като сме направили **проверките** за разстоянието и сме **изчислили** цената на **най-евтиния транспорт**, следва да я **отпечатаме**. В условието на задачата **няма** изисквания как да бъде форматиран резултата и по тази причина ще отпечатаме само **променливата**:

```
cout << price << endl;
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1361#0>.

Задача: тръби в басейн

Басейн с **обем V** има **две тръби**, от които се пълни. **Всяка тръба има определен дебит** (литрите вода, минаващи през една тръба за един час). Работникът пуска тръбите едновременно и излиза за **N часа**. Напишете програма, която изкарва състоянието на басейна, **в момента**, когато работникът се върне.

Входни данни

От конзолата се четат **четири реда**:

- Първият ред съдържа числото **V** – обем на басейна в литри – цяло число в интервала [1 ... 10000].
- Вторият ред съдържа числото **P1** – дебит на първата тръба за час – цяло число в интервала [1 ... 5000].
- Третият ред съдържа числото **P2** – дебит на втората тръба за час – цяло число в интервала [1 ... 5000].
- Четвъртият ред съдържа числото **H** – часовете, в които работникът **отсъства** – число с плаваща запетая в интервала [1.0 ... 24.00].

Изходни данни

Да се отпечата на конзолата **едно от двете възможни състояния**:

- До колко се е запълнил басейнът и коя тръба с колко процента е допринесла. Всички проценти да се форматират до цяло число

(без закръгляне)

- До колко се е запълнил басейнът и коя тръба с колко процента е допринесла. Всички проценти да се форматират до цяло число
 - "The pool is [x]% full. Pipe 1: [y]%. Pipe 2: [z]%"
- Ако басейнът се е препълнил – с колко литра е прелял за даденото време, число с плаваща запетая.
 - "For [x] hours the pool overflows with [y] liters."

Имайте предвид, че поради закръглянето до цяло число се губят данни и е нормално сборът на процентите да е 99%, а не 100%.

Примерен вход и изход

Вход	Изход
1000	
100	
120	The pool is 66% full. Pipe 1: 45%. Pipe2: 54%.
3	

Вход	Изход
100	
100	
100	
2.5	For 2.5 hours the pool overflows with 400 liters.

Насоки и подсказки

За да решим задачата, ще прочетем входа, ще извършим няколко проверки и изчисления и ще отпечатаме резултата.

Обработка на входните данни

От условието на задачата виждаме, че в програмата ни трябва да има **четири реда**, от които четем **входните данни**. Чрез първите **три** ще въвеждаме **цели числа** и по тази причина **променливите**, в които ще запазваме стойностите, ще бъдат от тип **int**. За **четвъртия** ред ни е казано, че ще бъде **число**, което е **с плаваща запетая**, затова и **променливата**, която ще използваме, ще е от тип **double**:

```
int volume;
int pipe1;
int pipe2;
double hours;
```

Следващата ни стъпка е да **декларираме и инициализираме** променлива, в която ще изчислим с колко **литра** се е **напълнил** басейна за **времето**, в което работникът е **отсъствал**. Изчисленията ще направим като **съберем** стойностите на дебита на **двете тръби** и ги **умножим** по **часовете**, които са ни зададени като вход:

```
int pipes =
double water =
```

Извършване на проверки и обработка на изходните данни

След като вече имаме и **стойността на количеството** вода, което е минало през **тръбите**, следва стъпката, в която трябва да **сравним** това количество с обема на самия басейн.

Това ще направим с прости **if-else** конструкции, в която условието ще проверява дали **количеството вода е по-малко от обема на басейна**. Ако проверката върне **true**, то трябва да разпечатаме един **ред**, който да съдържа **съотношението** между количеството **вода**, минало през **тръбите**, и **обема на басейна**, както и **съотношението на количеството вода от всяка една тръба спрямо обема на басейна**.

Съотношението е нужно да бъде изразено в **проценти**, затова и всички изчисления до момента ще бъдат **умножени по 100**. Тъй като има условие **результатът в проценти** да се форматира до **две цифри** след **десетичния** знак без закръгляне, то за целта ще използваме функцията **trunc(...)**:

```
if (water <= volume) {
    double sumPercent = trunc(
        (water / volume) * 100
    );
    double pipe1Percent = trunc(
        ((volume - water) / volume) * 100
    );
    double pipe2Percent = trunc(
        ((volume - water) / volume) * 100
    );

    cout << "The pool is " << sumPercent << "% full. Pipe 1: " <<
        pipe1Percent << "%." Pipe 2: " << pipe2Percent << "%" << endl;
}
else {
    double difference = water - volume;
    cout << "For " << hours << " hours the pool overflows with " <<
        difference << " liters" << endl;
}
```

Ако проверката обаче върне резултат **false**, то това означава, че **количеството вода е по-голямо от обема на басейна**, съответно той е **прелял**. Отново изхода трябва да е на **един ред**, но този път съдържа само две стойности - тази на **часовете**, в които работникът е отъствал, и **количеството вода**, което е разлика между влязлата вода и обема на басейна.

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1361#1>

Задача: поспаливата котка Том

Котката Том обича по цял ден да спи, за негово съжаление стопанинът му си играе с него винаги когато има свободно време. За да се наспи добре, **нормата за игра** на Том е **30 000 минути в година**. Времето за игра на Том **зависи от почивните дни на стопанина му**:

- Когато е на **работа**, стопанинът му си играе с него **по 63 минути на ден**.
- Когато **почива**, стопанинът му си играе с него **по 127 минути на ден**.

Напишете програма, която въвежда **броя почивни дни** и отпечатва дали **Том може да се наспи добре** и колко е **разликата от нормата** за текущата година, като приемем че **годината има 365 дни**.

Пример: 20 почивни дни -> работните дни са 345 ($365 - 20 = 345$). Реалното време за игра е 24 275 минути ($345 * 63 + 20 * 127$). Разликата от нормата е 5 725 минути ($30\ 000 - 24\ 275 = 5\ 725$) или 95 часа и 25 минути.

Входни данни

Входът се чете от конзолата и се състои от едно цяло число - **броят почивни дни** в интервала **[0 ... 365]**.

Изходни данни

На конзолата трябва да се отпечатат **два реда**.

- Ако времето за игра на Том **е над нормата** за текущата година:
 - На **първия ред** отпечатайте: "Tom will run away".
 - На **втория ред** отпечатайте разликата от нормата във формат: "**{H}** hours and **{M}** minutes more for play".
- Ако времето за игра на Том **е под нормата** за текущата година:
 - На **първия ред** отпечатайте: "Tom sleeps well".
 - На **втория ред** отпечатайте разликата от нормата във формат: "**{H}** hours and **{M}** minutes less for play".

Примерен вход и изход

Вход	Изход
20	Tom sleeps well 95 hours and 25 minutes less for play

Вход	Изход
113	Tom will run away 3 hours and 47 minutes for play

Насоки и подсказки

За да решим задачата, ще прочетем входа, ще извършим няколко проверки и изчисления и ще отпечатаме резултата.

Обработка на входните данни и прилежащи изчисления

От условието на задачата виждаме, че **входните данни** ще бъдат прочетени само от **един ред**, който ще съдържа в себе си **едно цяло число** в интервала [0 ... 365]. По тази причина ще използваме променлива от тип **int**:

```
int holidays;
```

За да решим задачата, **първо** трябва да изчислим колко **общо минути** стопанинът на Том си играе с него. От условието виждаме, че освен в **почивните дни**, поспаливата котка трябва да си играе и в **работните** за стопанина му. Числото, което прочитаме от конзолата, е това на **почивните дни**.

Следващата ни стъпка е с помощта на това число да **изчислим** колко са **работните дни** на стопанина, тъй като без тях не можем да стигнем до **общото количество минути за игра**. Щом общият брой на дните в годината е **365**, а броят на почивните дни е **X**, то това означава, че броят на работните дни е **365 - X**. **Разликата** ще запазим в нова променлива, която ще използваме **само** за тази стойност:

```
int workingDays =
```

След като вече имаме броя дни за игра, то вече можем да изчислим **времето за игра** на Том **в минути**. Неговата **стойност е равна** на **результатата от умножението на работните дни по 63** минути (в условието е зададено, че в работни дни, времето за игра е 63 минути на ден) **събран с резултата от умножението на почивните дни по 127** минути (в условието е зададено, че в почивните дни, времето за игра е 127 минути на ден):

```
int totalPlayMinutes =
```

В условието на задачата за изхода виждаме, че ще трябва да **разпечатаме разликата** между двете стойности в **часове и минути**. За тази цел от **общото време за игра** ще **извадим** нормата от **30 000** минути и получената разлика ще **запишем в нова** променлива. След това тази променлива ще **разделим** **целочислено** на 60, за да получим **часовете**, а след това, за да открием колко са **минутите**, ще използваме **модулно деление** (**оператора %**) с 60.

Тук трябва да отбележим, че ако полученото количество **време игра** на Том е **помалко** от **30 000**, при **изваждането** на нормата от него ще получим **число с отрицателен знак**. За да **неутрализираме** знака в двете деления по-късно, ще използваме **функцията abs(...)** при намирането на разликата:

```
int difference = abs( );  
int hours = ;  
int minutes = ;
```

Извършване на проверки

Времето за игра вече е изчислено, което ни води до **следващата** стъпка – **сравняване** на времето за игра на Том с нормата, от която зависи дали котката ще се наспива добре. За целта ще използваме **if-else** проверка, като в **if** клаузата ще проверим дали **времето за игра** е по-голямо от 30 000 (нормата).

Обработка на изходните данни

Каквъто и **результат** да ни върне проверката, то трябва да разпечатаме колко е **разликата в часове и минути**. Това ще направим с променливите, в които изчислихме стойностите на часовете и минутите, като форматирането ще е според условието за изход:

```
if (totalPlayMinutes > 30000) {  
    cout << "Tom will run away" << endl;  
    cout << hours << " hours and "  
        minutes << " minutes more for play" << endl;  
}  
else {  
    cout << "Tom sleeps well" << endl;  
    cout << hours << " hours and "  
        minutes << " minutes less for play" << endl;  
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1361#2>.

Задача: реколта

От лозе с площ X квадратни метри се заделя 40% от реколтата за производство на вино. От 1 кв. м. лозе се изкарват Y килограма грозде. За 1 литър вино са нужни 2,5 кг. грозде. Желаното количество вино за продан е Z литра.

Напишете програма, която пресмята колко вино може да се произведе и дали това количество е достатъчно. Ако е достатъчно, остатъкът се разделя по равно между работниците на лозето.

Входни данни

Входът се чете от конзолата и се състои от **точно 4 реда**:

- 1-ви ред: **X** кв.м е лозето – цяло число в интервала [10 ... 5000].
- 2-ри ред: **Y** грозде за един кв.м. – реално число в интервала [0.00 ... 10.00].
- 3-ти ред: **Z** нужни литри вино – цяло число в интервала [10 ... 600].
- 4-ти ред: брой работници – цяло число в интервала [1 ... 20].

Изходни данни

На конзолата трябва да се отпечата следното:

- Ако произведеното вино е **по-малко** от нужното:
 - “It will be a tough winter! More {недостигащо вино} liters wine needed.”
* Резултатът трябва да е закръглен към по-ниско цяло число.
- Ако произведеното вино е **повече** от нужното:
 - “Good harvest this year! Total wine: {общо вино} liters.”
* Резултатът трябва да е закръглен към по-ниско цяло число.
 - “[Оставашо вино] liters left -> {вино за 1 работник} liters per person.”
* И двата резултата трябва да са закръглени към по-високото цяло число.

Примерен вход и изход

Вход	Изход	Вход	Изход
650	Good harvest this year! Total wine: 208 liters.	1020	
2	33 liters left -> 11 liters per person.	1.5	It will be a tough winter! More
175		425	180 liters wine needed.
3		4	

Насоки и подсказки

За да решим задачата, ще прочетем входа, ще извършим няколко проверки и изчисления и ще отпечатаме резултата.

Обработка на входните данни и прилежащи изчисления

Първо трябва да проверим какви ще са **входните данни** и да изберем какви **променливи** ще използваме. Кодът по-долу е целенасочено замъглен и трябва да бъде довършен от читателя:

```
double vineyardArea;
double grapePerSquare;
double neededLiters;
int workers;
```

За да решим задачата е нужно да **изчислим** колко **литра вино** ще получим на база **входните данни**. От условието на задачата виждаме, че за да **пресметнем** количеството **вино в литри**, трябва първо да разберем какво е **количеството грозде в килограми**, което ще се получи от тази реколта. За тази цел ще **декларираме** една **променлива**, на която ще присвоим **стойност**, равна на **40%** от резултата от **умножението** на площта на лозето и количеството грозде, което се получава от 1 кв. м.

След като сме извършили тези пресмятания, сме готови да **пресметнем** и количеството **вино в литри**, което ще се получи от тази реколта. За тази цел **декларираме** още една **променлива**, в която ще пазим това **количество**, а от условието стигаме до извода, че за да го пресметнем, е нужно да **разделим** количеството грозде в **кг** на **2.5**:

```
double harvestPerWine =
double wine =
```

Извършване на проверки и обработка на изходните данни

Вече сме направили нужните пресмятания и следващата стъпка е да **роверим** дали получените литри вино са **достатъчни**. За целта ще използваме пристапа **if-else**, като в условието ще **роверим** дали **литрите вино** от реколтата са **повече от** или **равни** на **нужните литри**.

Ако проверката върне резултат **true**, от условието на задачата виждаме, че на **първия ред** трябва да разпечатаме **виното**, което сме **получили от** реколтата. За да спазим условието тази стойност да бъде **закръглена до по-ниското цяло число**, ще използваме функцията **floor(...)**.

На **втория ред** има изискване да разпечатаме резултатите, като ги **закръглим към по-високото цяло число**, което ще направим с функцията **ceil(...)**. Стойностите, които трябва да разпечатаме, са на **оставащото количество вино** и **количеството вино**, което се пада на **един работник**. Оставащото количество вино е равно на разликата между получените литри вино и нужните литри вино. Стойността на това количество ще изчислим в нова променлива, която ще декларираме и инициализираме в **блок тялото** на **if** клаузата, **преди** разпечатването на първия ред. Количество вино, което се полага на **един работник**, ще изчислим като оставащото вино го разделим на броя на работниците:

```
if ( ) {
    double totalWine = floor( );
    double litersLeft = ceil( );
    double litersPerPerson = ceil( );
}
```

```

    cout << "Good harvest this year! Total wine: " <<
        totalWine << " liters." << endl;
    cout << litersLeft << " liters left -> " <<
        litersPerPerson << " liters per person." << endl;
}

```

Ако проверката ни върне резултат **false** от условието на задачата виждаме, че трябва да разпечатаме разликата от нужните литри и получените от тази реколта литри вино. Има условие резултата да е закръглен към по-ниското цяло число, което ще направим с функцията **floor(...)**:

```

else {
    double litersNeeded = floor(
        litersLeft / litersPerPerson);
    cout << "It will be a tough winter! More " <<
        litersNeeded << " liters wine needed." << endl;
}

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1361#3>.

Задача: фирма

Фирма получава заявка за изработването на проект, за който са необходими определен брой часове. Фирмата разполага с **определен брой дни**. През 10% от дните служителите са на **обучение** и не могат да работят по проекта. Един нормален **работен ден във фирмата е 8 часа**. Проектът е важен за фирмата и всеки служител задължително работи по проекта в **извънработно време по 2 часа на ден**.

Часовете трябва да са **закръглени към по-ниско цяло число** (например → 6.98 часа се закръглат на **6 часа**).

Напишете програма, която изчислява дали фирмата може да завърши проекта навреме и колко часа не достигат или остават.

Входни данни

Входът се чете от конзолата и съдържа **точно 3 реда**:

- На **първия** ред са **необходимите часове** – цяло число в интервала [0 ... 200 000].
- На **втория** ред са **дните**, с които фирмата разполага – цяло число в интервала [0 ... 20 000].
- На **третия** ред е **броят на всички служители** – цяло число в интервала [0 ... 200].

Изходни данни

Да се отпечатат на конзолата един ред:

- Ако времето е достатъчно: "Yes!{оставащите часове} hours left.".
- Ако времето НЕ Е достатъчно: "Not enough time!{недостигащите часове} hours needed.".

Примерен вход и изход

Вход	Изход	Вход	Изход
90		99	
7	Yes!2 hours left.	3	Not enough time!72 hours needed.
3		1	

Насоки и подсказки

За да решим задачата, ще прочетем входа, ще извършим няколко проверки и изчисления и ще отпечатаме резултата.

Обработка на входните данни

За решението на задачата е нужно първо да преценим какви **типове променливи** ще използваме за **входните данни**. Кодът по-долу е целенасочено замъглен и трябва да бъде довършен от читателя:

```
int projectHours;
int availableDays;
int overtimeWorkers;
```



Помощни изчисления

Следващата стъпка е да изчислим **количество**та на **работните часове** като умножим работните дни по 8 (всеки ден се работи по 8 часа) с броя на работниците и ги съберем с извънработното време. **Работните дни** са равни на **90% от дните**, с които фирмата разполага. **Извънработното време** е равно на резултата от умножението на броя на служителите с 2 (възможните часове извънработно време), като това също се умножава по броя на дните, с които фирмата разполага. От условието на задачата виждаме, че има условие **часовете** да са **закръглени към по-ниско цяло число**, което ще направим с функцията **floor(...)**:

```
double workDays =  
double overtimeHours =  
double workHours =  
double totalHours =
```

Извършване на проверки

След като сме направили изчисленията, които са ни нужни, за да разберем стойността на **работните часове**, следва да направим проверка дали тези часове **достигат или остават допълнителни** такива.

Ако **времето е достатъчно**, разпечатваме резултата, който се изисква в условието на задачата, а именно разликата между **работните часове и необходимите часове** за завършване на проекта.

Ако **времето не е достатъчно**, разпечатваме допълнителните часове, които са нужни за завършване на проекта и са равни на разликата между **часовете за проекта и работните часове**:

```
if ( ) {  
    int result =  
    cout << "Yes!" << result << " hours left." << endl;  
}  
else {  
    int result =  
    cout << "Not enough time!" << result << " hours needed." << endl;  
}
```

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1361#4>.

Глава 4.1. По-сложни проверки

В настоящата глава ще разгледаме **вложените проверки** в езика C++, чрез които нашата програма може да съдържа **условни конструкции**, в които има **вложени други условни конструкции**. Наричаме ги "вложени", защото поставяме **if** конструкция в друга **if** конструкция. Ще разгледаме и **по-сложни логически условия** с подходящи примери.

Видео

Гледайте видео урок по учебния материал от настоящата глава от книгата: <https://www.youtube.com/watch?v=XK8W2DP-oDM>.

Вложени проверки

Доста често програмната логика налага използването на **if** или **if-else** конструкции, които се съдържат една в друга. Те биват наричани **вложени if** или **if-else** конструкции. Както се подразбира от названието "вложени", това са **if** или **if-else** конструкции, които са поставени в други **if** или **else** конструкции.

```
if (условие1) {  
    if (условие2) {  
        // тяло  
    }  
    else {  
        // тяло  
    }  
}
```



Влагането на повече от три условни конструкции една в друга не се счита за добра практика и трябва да се избягва, най-вече чрез оптимизиране на структурата/алгоритъма на кода и/или чрез използването на друг вид условна конструкция, който ще разгледаме по-надолу в тази глава.

Пример: обръщение според възраст и пол

Според въведени **възраст** (десетично число) и **пол** (**m** / **f**) да се отпечатва обръщение:

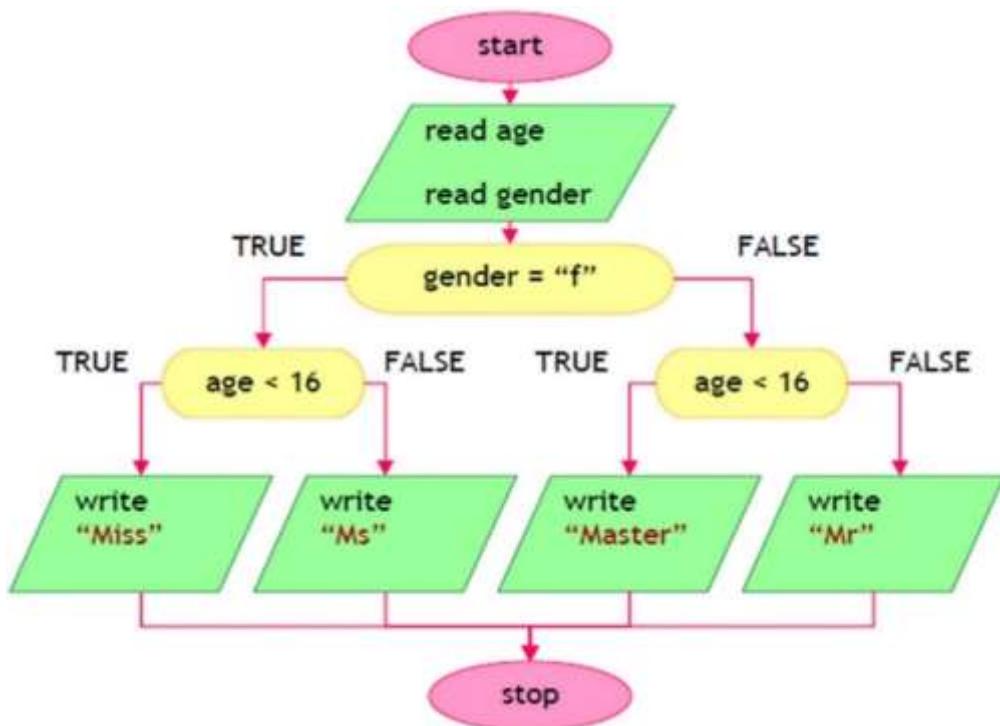
- "Mr." – мъж (пол "m") на 16 или повече години.
- "Master" – момче (пол "m") под 16 години.
- "Ms." – жена (пол "f") на 16 или повече години.
- "Miss" – момиче (пол "f") под 16 години.

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
12 f	Miss	17 m	Mr.	25 f	Ms.	13.5 m	Master

Решение

Можем да забележим, че **изходът** на програмата зависи от **няколко неща**. Първо трябва да проверим какъв **пол** е въведен и **после** да проверим **възрастта**. Съответно ще използваме няколко **if-else** блока. Тези блокове ще бъдат вложени, т.е. от **результатата** на първия ще се определи кои от **другите** да се изпълни:



След прочитане на входните данни от конзолата ще трябва да се изпълни следната примерна програмна логика:

```

double age;
string gender;
cin >> age >> gender;
  
```

```

if (gender == "m") {
    if (age < 16) {
        cout << "Master" << endl;
    }
    else {
        cout << "Mr." << endl;
    }
}
else if (gender == "f") {
    if (age < 16) {
        cout << "Miss" << endl;
    }
    else {
        cout << "Ms." << endl;
    }
}

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1362#0>.

Пример: квартално магазинче

Предприемчив българин отваря по едно квартално магазинче в няколко града с различни цени за следните продукти:

продукт	Sofia	Plovdiv	Varna
coffee	0.50	0.40	0.45
water	0.80	0.70	0.70
beer	1.20	1.15	1.10
sweets	1.45	1.30	1.35
peanuts	1.60	1.50	1.55

По дадени град (стринг), продукт (стринг) и количество (десетично число), да се пресметне цената ѝ.

Примерен вход и изход

Вход	Изход
coffee	
Varna	0.9
2	

Вход	Изход
peanuts	
Plovdiv	1.5
1	

Вход	Изход
beer	
Sofia	7.2
6	

Вход	Изход
water	
Plovdiv	2.1
3	

Решение

Примерен алгоритъм за решение:

```
string product;
string town;
double quantity;

cin >> product >> town >> quantity;

if (town == "Sofia") {
    if (product == "coffee") {
        cout << 0.50 * quantity << endl;
    }
    // TODO: Finish this...
}
if (town == "Plovdiv") {
    // TODO: Finish this...
}
if (town == "Varna") {
    // TODO: Finish this...
}
```

Кодът е недовършен с цел да се стимулира самостоятелно мислене и решение.

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1362#1>.

По-сложни проверки

Нека разгледаме как можем да правим по-сложни логически проверки. Може да използваме логическо "И" (**&&**), логическо "ИЛИ" (**||**), логическо **отрицание** (**!**) и скоби (**()**).

Логическо "И"

Както видяхме, в някои задачи се налага да правим **много проверки наведнъж**. Но какво става, когато за да изпълним някакъв код, трябва да бъдат изпълнени **няколко условия едновременно** и **не искаме** да правим **отрицание (else)** за всяко едно от тях? Вариантът с вложените **if блокове** е валиден, но кодът би изглеждал много **неподреден** и със сигурност - **труден** за четене и поддръжка.

Логическо "И" (оператор **&&**) означава **няколко условия да са изпълнени едновременно**. В сила е следната таблица на истинност:

a	b	$a \parallel b$	a	b	$a \parallel b$
true	true	true	false	true	false
true	false	false	false	false	false

Как работи операторът **&&** ?

Операторът **&&** приема **няколко булеви** (условни) израза, които имат стойност **true** или **false**, и ни връща **един** булев израз като **резултат** (0 за **false** и 1 за **true**). Използването му **вместо** редица вложени **if** блокове прави кода **по-четлив**, **подреден** и **лесен** за поддръжка. Но как **работи**, когато поставим **няколко** условия едно след друго? Както видяхме по-горе, логическото "И" връща **true**, само когато приема като **аргументи изрази** със стойност **true**. Съответно, когато имаме **последователност** от аргументи, логическото "И" **проверява** или докато **свършат** аргументите, или докато не **срещне** аргумент със стойност **false**.

Пример:

```
bool a = true;
bool b = true;
bool c = false;
bool d = true;

bool result = a && b && c && d;
// 0 - false (като d не се проверява)
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/OperatorAND>.

Програмата ще се изпълни по **следния** начин: **започва** проверката от **a**, прочита я и отчита, че има стойност **true**, след което **проверява b**. След като е **отчела**, че **a** и **b** връщат стойност **true**, **проверява следващия** аргумент. Стига до **c** и отчита, че променливата има стойност **false**. След като програмата отчете, че аргументът **c** има стойност **false**, тя изчислява израза **до c, независимо** каква е стойността на **d**. За това проверката на **d** се прескача и целият израз бива изчислен като **false**.

```
if (x >= x1 && x <= x2 && y >= y1 && y <= y2)
```

Пример: точка в правоъгълник

Проверка дали точка $\{x, y\}$ се намира вътре в правоъгълника $\{x1, y1\} - \{x2, y2\}$. Входните данни се четат от конзолата и се състоят от 6 реда: десетичните числа $x1, y1, x2, y2, x$ и y (като се гарантира, че $x1 < x2$ и $y1 < y2$).

Примерен вход и изход

Вход	Изход	Визуализация
2 -3 12 3 8 -1	Inside	

Решение

Една точка е вътрешна за даден многоъгълник, ако **едновременно** са изпълнени следните четири условия:

- Точката е вдясно от лявата страна на правоъгълника.
- Точката е вляво от дясната страна на правоъгълника.
- Точката е под горната страна на правоъгълника.
- Точката е над долната страна на правоъгълника.

```
double x1, y1, x2, y2, x, y;
cin >> x1 >> y1
    >> x2 >> y2
    >> x >> y;

if (x >= x1 && x <= x2
    && y >= y1 && y <= y2) {
    cout << "Inside" << endl;
}
```

```
else {
    cout << "Outside" << endl;
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1362#2>.

Логическо "ИЛИ"

Логическо "ИЛИ" (оператор `||`) означава да е **изпълнено** поне **едно** измежду няколко условия. Подобно на оператора `&&`, логическото "ИЛИ" приема няколко аргумента от **булев** (условен) тип и връща **true** или **false**. Лесно можем да се досетим, че **получаваме** като стойност **true**, винаги когато поне **един** от аргументите има стойност **true**. Типичен пример за логиката на този оператор е следният:

В училище учителят казва: "Иван или Петър да измият дъската". За да бъде изпълнено това условие (дъската да бъде измита), е възможно само Иван да я измие, само Петър да я измие или и двамата да го направят.

a	b	a b	a	b	a b
true	true	true	false	true	true
true	false	true	false	false	false

Как работи операторът || ?

Вече научихме какво **представлява** логическото "ИЛИ". Но как всъщност се реализира? Както при логическото "И", програмата **роверява** от ляво на дясно **аргументите**, които са зададени. За да получим **true** от израза, е необходимо **само един** аргумент да има стойност **true**, съответно проверката **продължава** докато се срещне **аргумент с такава** стойност или докато **не свършат** аргументите.

Ето един **пример** за оператора `||` в действие:

```
bool a = false;
bool b = false;
bool c = true;
bool d = true;

bool result = a || b || c || d;
// 1 - true (като с и d не се проверяват)
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/OperatorOr>.

Програмата проверява **a**, отчита, че има стойност **false** и продължава. Стигайки до **b**, отчита, че има стойност **true** и целият израз получава стойност **true**, без да се проверява **c** и **d**, защото техните стойности не биха променили резултата на израза.

Пример: плод или зеленчук

Нека проверим дали даден продукт е **плод** или **зеленчук**. Плодовете "fruit" са **banana**, **apple**, **kiwi**, **cherry**, **lemon** и **grapes**. Зеленчуците "vegetable" са **tomato**, **cucumber**, **pepper** и **carrot**. Всички останали са "**unknown**".

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
banana	fruit	tomato	vegetable	java	unknown

Решение

Трябва да използваме няколко условни проверки с логическо "ИЛИ" (**||**):

```
string input;
cin >> input;

if (input == "banana" || input == "apple" ||
    input == "kiwi" || input == "cherry" ||
    input == "lemon" || input == "grapes") {
    cout << "fruit" << endl;
}
else if (input == "tomato" || input == "cucumber" ||
          input == "pepper" || input == "carrot") {
    cout << "vegetable" << endl;
}
else {
    cout << "unknown" << endl;
}
```

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1362#3>.

Логическо отрицание

Логическо отрицание (оператор `!`) означава да не е изпълнено дадено условие.

a	!a	a	!a
true	false	false	true

Операторът `!` приема като аргумент булева променлива и обръща стойността ѝ (истината стажа лъжа, а лъжата става истина).

Пример: невалидно число

Дадено число е валидно, ако е в диапазона `[100 ... 200]` или е `0`. Да се направи проверка за невалидно число.

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
75	invalid	150	(няма изход)	220	invalid

Решение

```
bool inRange = (number >= 100 && number <= 200) || number == 0;
if (!inRange) {
    cout << "invalid" << endl;
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1362#4>.

Операторът скоби `()`

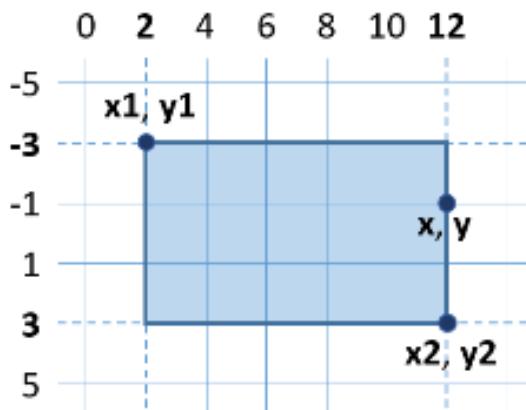
Както останалите оператори в програмирането, така и операторите `&&` и `||` имат приоритет, като в случая `&&` е с по-голям приоритет от `||`. Операторът `()` служи за промяна на приоритета на операторите и се изчислява пръв, също както в математиката. Използването на скоби също така придава по-добра четливост на кода и се счита за добра практика.

По-сложни логически условия

Понякога условията може да са доста сложни, така че да изискват дълъг булев израз или поредица от проверки. Да разгледаме няколко такива примера.

Пример: точка върху страна на правоъгълник

Да се напише програма, която проверява дали точка $\{x, y\}$ се намира върху някоя от страните на правоъгълник $\{x_1, y_1\} - \{x_2, y_2\}$. Входните данни се четат от конзолата и се състоят от 6 реда: десетичните числа x_1, y_1, x_2, y_2, x и y (като се гарантира, че $x_1 < x_2$ и $y_1 < y_2$). Да се отпечата "Border" (точката лежи на някоя от страните) или "Inside / Outside" (в противен случай).



Примерен вход и изход

Вход	Изход	Вход	Изход
2		2	
-3		-3	
12	Border	12	
3		3	Inside / Outside
12		8	
-1		-1	

Решение

Точка лежи върху някоя от страните на правоъгълник, ако:

- x съвпада с x_1 или x_2 и същевременно y е между y_1 и y_2 или
- y съвпада с y_1 или y_2 и същевременно x е между x_1 и x_2 .

```
if (((x == x1 || x == x2) && (y >= y1) && (y <= y2)) ||
    ((y == y1 || y == y2) && (x >= x1) && (x <= x2))) {
    cout << "Border" << endl;
}
```

Предходната проверка може да се опрости по предложения на идната страница начин:

```

int onLeftSide = (x == x1) && (y >= y1) && (y <= y2);
int onRightSide = (x == x2) && (y >= y1) && (y <= y2);
int onUpSide = (y == y1) && (x >= x1) && (x <= x2);
int onDownSide = (y == y2) && (x >= x1) && (x <= x2);

if (onLeftSide || onRightSide || onUpSide || onDownSide) {
    cout << "Border" << endl;
}

```

Вторият начин с допълнителните булеви променливи е по-дълъг, но е много по-разбираем от първия, нали? Препоръчваме ви когато пищете булеви условия, да ги правите **лесни за четене и разбиране**, а не кратки. Ако се налага, ползвайте допълнителни променливи със смислени имена. Имената на булевите променливи трябва да подсказват каква стойност се съхранява в тях.

Остава да се допише кода, за да отпечатва "Inside / Outside", ако точката не е върху някоя от страните на правоъгълника.

Тестване в Judge системата

След като допишете решението, може да го тествате тук: <https://judge.softuni.bg/Contests/Practice/Index/1362#5>.

Пример: магазин за плодове

Магазин за плодове в **работни дни** продава на **цени**, показани в лявата таблица, а през почивните дни – на цените от дясната таблица:

Плод	Цена
banana	2.50
apple	1.20
orange	0.85
grapefruit	1.45
kiwi	2.70
pineapple	5.50
grapes	3.85

Плод	Цена
banana	2.70
apple	1.25
orange	0.90
grapefruit	1.60
kiwi	3.00
pineapple	5.60
grapes	4.20

Напишете програма, която чете от конзолата **плод** (banana / apple / ...), **ден от седмицата** (Monday / Tuesday / ...) и **количество (десетично число)** и пресмята **цената** според цените от таблиците по-горе. Резултатът да се отпечата **закръглен**

с 2 цифри след десетичния знак. При невалиден ден от седмицата или невалидно име на плод да се отпечата "error".

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
orange Sunday 3	2.70	kiwi Monday 2.5	6.75	grapes Saturday 0.5	2.10	tomato Monday 0.5	error

Решение

```
if (day == "Saturday" || day == "Sunday") {
    if (fruit == "banana") {
        price = 2.70;
    }
    else if (fruit == "apple") {
        price = 1.25;
    }
    // TODO: More fruits come here...
}
else if (day == "Monday" || day == "Tuesday" ||
          day == "Wednesday" || day == "Thursday" || day == "Friday") {
    if (fruit == "banana") {
        price = 2.50;
    }
    // TODO: More fruits come here...
```

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1362#6>.

Пример: търговски комисионни

Фирма дава следните **комисионни** на търговците си според **града**, в който работят и обема на продажбите s:

Град	0 <= s <= 500	500 < s <= 1000	1000 < s <= 10000	s > 10000
Sofia	5%	7%	8%	12%
Varna	4.5%	7.5%	10%	13%
Plovdiv	5.5%	8%	12%	14.5%

Напишете програма, която чете име на **град** (стринг) и обем на **продажбите** (десетично число) и изчислява размера на комисионната. Резултатът да се изведе закръглен с **2 десетични цифри** след десетичния знак. При **невалиден град** или **обем на продажбите**(отрицателно число) да се отпечата "error".

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
Sofia 1500	120.00	Plovdiv 499.99	27.50	Kaspichan -50	error

Решение

Първоначално задаваме комисионната да е **-1**. Тя ще бъде променена, ако градът и ценовият диапазон бъдат намерени в таблицата с комисионните. За да изчислим комисионната според града и обема на продажбите се нуждаем от няколко вложени **if проверки**, както е в примерния код по-долу:

```
double comission = -1.0;

if (town == "Sofia") {
    if (0 <= sales && sales <= 500) {
        comission = 0.05;
    }
    else if (500 < sales && sales <= 1000) {
        comission = 0.07;
    }

    // TODO: Check the other price ranges...
}

else if (town == "Varna") {
    // TODO: Check the other price ranges...
}

else if (town == "Plovdiv") {
    // TODO: Check the other price ranges...
}

if (comission >= 0) {
    cout << setprecision(2) << fixed << comission;
}
else {
    cout << "error";
}
```



Добра практика е да използваме **блокове**, които **заграждаме** с къдрави скоби `{ }` след **if** и **else**. Също така, препоръчително е при писане да **отместваме** кода след **if** и **else** с една табулация **навътре**, за да направим кода по-лесно четим.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1362#7>.

Условна конструкция switch-case

Конструкцията **switch-case** работи като поредица **if-else** блокове. Когато работата на програмата ни зависи от стойността на **една променлива**, вместо да правим последователни проверки с **if-else** блокове, можем да използваме условната конструкция **switch**. Тя се използва за **избор измежду списък с възможности**. Конструкцията сравнява дадена стойност с определени константи и в зависимост от резултата предприема действие.

Променливата, която искаме да сравняваме, поставяме в скобите след оператора **switch** и се нарича "селектор". Тук типът трябва да е сравним. Последователно започва **сравняването** с всяка една **стойност**, която се намира след **case** етикетите. При съвпадение започва изпълнението на кода от съответното място и продължава, докато стигне оператора **break**. В C++ и някои други програмни езици **break** може да се изпуска, за да се изпълнява код от друга **case** конструкция, докато не стигне до въпросния оператор. При **липса** на **съвпадение**, се изпълнява **default** конструкцията, ако такава **съществува**.

```
switch (селектор) {
    case стойност1:
        конструкция;
        break;
    case стойност2:
        конструкция;
        break;
    case стойност3:
        конструкция;
        break;
    ...
    default:
        конструкция;
        break;
}
```

В C++ **селекторът** и **стойностите** в **case** етикетите трябва да са **цели числа (int)**. За разлика от други езици за програмиране, C++ **не позволява** използването на конструкция **switch-case** с низове (**string**), или дробни числа (**double**).

Пример: ден от седмицата

Нека напишем програма, която принтира **дения от седмицата** (на английски) според въведеното число (1 ... 7) или "Error!", ако е въведено невалидно число.

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
1	Monday	7	Sunday	-1	Error!

Решение

```
int day;
cin >> day;

switch (day)
{
case 1:
    cout << "Monday" << endl;
    break;
case 2:
    cout << "Tuesday" << endl;
    break;
...
default:
    cout << "Error" << endl;
    break;
}
```



Добра практика е на първо място да поставяме онези **case** случаи, които обработват **най-често случилите се ситуации**, а **case** конструкциите, обработващи **по-рядко възникващи ситуации**, да оставим **в края**, преди **default** конструкцията. Друга добра практика е да подреждаме **case** етикетите в **нарастващ ред**, без значение дали са целочислени или символни.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1362#8>.

Пример: вид животно

Напишете програма, която принтира вида на животно според името му:

- dog -> mammal
- crocodile, tortoise, snake -> reptile
- others -> unknown

Примерен вход и изход

Вход	Изход
tortoise	reptile

Вход	Изход
dog	mammal

Вход	Изход
elephant	unknown

Решение

В тази задача не можем да използваме **switch-case**, тъй като трябва да сравняваме низове, а не цели числа. Можем да я решим чрез няколко **if-else** проверки:

```
if (animal == "dog") {
    cout << "mammal" << endl;
}
else if (animal == "crocodile" ||
          animal == "tortoise" ||
          animal == "snake") {
    cout << "reptile" << endl;
}
else {
    cout << "unknown" << endl;
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1362#9>.

Какво научихме от тази глава?

Да си припомним новите конструкции и програмни техники, с които се запознахме в тази глава:

Вложени проверки

```
if (условие1) {
    if (условие2) {
        // тяло
    }
    else {
```

```
// тяло
}
}
```

По-сложни проверки с &&, ||, ! и ()

```
if (((x == left) || (x == right))
    && (y >= top) && (y <= bottom)) {
    // тяло
}
```

Switch-case проверки

```
switch (селектор)
{
    case стойност1:
        конструкция;
        break;
    case стойност2:
        конструкция;
        break;
    case стойност3:
        конструкция;
        break;
    ...
    default:
        конструкция;
        break;
}
```

Упражнения: по-сложни проверки

Нека сега да упражним работата с по-сложни проверки. Да решим няколко практически задачи.

Задача: кино

В една кинозала столовете са наредени в **правоъгълна** форма в **rows** реда и **columns** колони. Има три вида прожекции с билети на **различни** цени:

- **Premiere** – премиерна прожекция, на цена **12.00** лева.
- **Normal** – стандартна прожекция, на цена **7.50** лева.
- **Discount** – прожекция за деца, ученици и студенти на намалена цена от **5.00** лева.

Напишете програма, която въвежда **тип прожекция** (стринг), брой **редове** и брой **колони** в залата (цели числа) и изчислява **общите приходи** от билети при **пълна зала**. Резултатът да се отпечата във формат като в примерите по-долу - с 2 цифри след десетичния знак.

Примерен вход и изход

Вход	Изход	Вход	Изход
Premiere 10 12	1440.00 leva	Normal 21 13	2047.50 leva

Насоки и подсказки

Прочитаме входните данни. След което, създаваме и инициализираме променлива, която ще ни съхранява изчислените приходи. В друга променлива пресмятаме пълния капацитет на залата. Тъй като тук не можем да използваме условната конструкция **switch-case**, ще използваме **if-else**, за да изчислим прихода в зависимост от вида на прожекцията и отпечатваме резултата на конзолата в зададения формат (потърсете нужната **C++** функционалност в интернет).

При прочитането на входа можем да обърнем типа на прожекцията в малки букви (с функцията **.lower()**). Създаваме и променлива, която ще ни съхранява изчислените приходи. В друга променлива пресмятаме пълния капацитет на залата. Използваме **if-elif** условна конструкция, за да изчислим прихода в зависимост от вида на прожекцията и отпечатваме резултата на конзолата в зададения формат (потърсете нужната **Python** функционалност в интернет).

Примерен код на описаната идея (част от кода са замъглени с цел да се стимулира самостоятелно мислене и решение):

```
string type;
int rows;
int columns;

int full =
double income;

if (type == "Premiere") {
    income =
}
```



Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:
<https://judge.softuni.bg/Contests/Practice/Index/1362#10>.

Задача: волейбол

Влади е студент, живее в София и си ходи от време на време до родния град. Той е много запален по волейбала, но е зает през работните дни и играе **волейбол** само през **уикендите** и в **празничните дни**. Влади играе в **София** всяка събота, когато **не е на работа** и не си пътува до родния град, както и в **2/3 от празничните дни**. Той пътува до **родния си град h** пъти в годината, където играе волейбол със старите си приятели в **неделя**. Влади **не е на работа 3/4 от уикендите**, в които е в София. Отделно, през **високосните години** Влади играе с **15% повече** волейбол от нормалното. Приемаме, че годината има точно **48 уикенда**, подходящи за волейбол. Напишете програма, която изчислява **колко пъти Влади е играл волейбол** през годината. **Закръглете резултата** надолу до най-близкото цяло число (напр. 2.15 -> 2; 9.95 -> 9).

Входните данни се четат от конзолата:

- Първият ред съдържа думата "leap" (високосна година) или "normal" (нормална година с 365 дни).
- Вторият ред съдържа цялото число **p** – брой празници в годината (които не са събота или неделя).
- Третият ред съдържа цялото число **h** – брой уикенди, в които Влади си пътува до родния град.

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
leap 5 2	45	normal 3 2	38	normal 11 6	44	leap 0 1	41

Насоки и подсказки

Стандартно прочитаме входните данни от конзолата. Последователно пресмятаме **уикендите прекарани в София**, **времето за игра в София** и **общото време за игра**. Накрая проверяваме дали годината е **високосна**, правим допълнителни изчисления при необходимост и извеждаме резултата на конзолата, **закръглен надолу** до най-близкото **цяло число** (потърсете **C++** функция с такава функционалност в интернет).

Примерен код на описаната идея (части от кода са замъглени с цел да се стимулира самостоятелно мислене и решение):

```
string year;
int holidays;
int weekendsHome;

int sofiaWeekends =
double playSofia =

double playTotal =

if (      ) {
    playTotal =
}

else if (      ) {

}

cout << playTotal;
```

Тестване в Judge системата

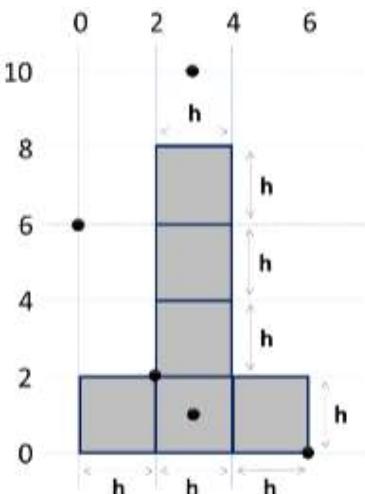
Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1362#11>.

Задача: * точка във фигурата

Фигура се състои от 6 блокчета с размер $h \times h$, разположени като на фигурата. Долният ляв ъгъл на сградата е на позиция $\{0, 0\}$. Горният десен ъгъл на фигурата е на позиция $\{2 \cdot h, 4 \cdot h\}$. На фигурата координатите са дадени при $h = 2$:

Да се напише програма, която въвежда цяло число h и координатите на дадена **точка** $\{x, y\}$ (цели числа) и отпечатва дали точката е вътре във фигурата (**inside**), вън от фигурата (**outside**) или на някоя от стените на фигурата (**border**).



Примерен вход и изход

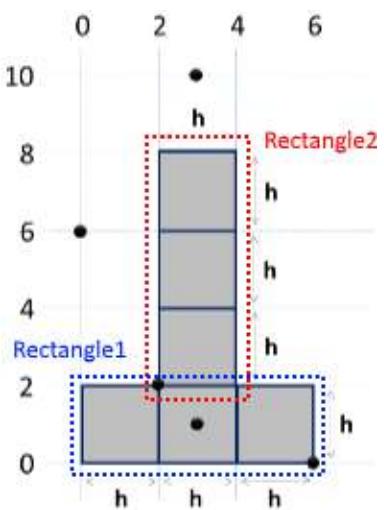
Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
2		2		2		2	
3	outside	3	inside	2	border	6	border
10		1		2		0	

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
2		15		15		15	
0	outside	13	outside	29	inside	37	outside
6		55		37		18	

Насоки и подсказки

Примерна логика за решаване на задачата (не е единствената правилна):

- Може да разделим фигурата на **два правоъгълника** с обща стена:
- Една точка е **външна (outside)** за фигурата, когато е едновременно **извън** двета правоъгълника.
- Една точка е **вътрешна (inside)** за фигурата, ако е вътре в някой от правоъгълниците (изключвайки стените им) или лежи върху общата им стена.



- В **противен случай** точката лежи на стената на правоъгълника (**border**).

Примерен код (части от кода са замъglени с цел да се стимулира самостоятелно мислене и решение):

```
int h, x, y;

bool outRectangle1 = ...;
bool outRectangle2 = ...;

bool inRectangle1 = ...;
bool inRectangle2 = ...;

bool commonBorder = ...;

if (...) {
    cout << "outside" << endl;
}
else if (...) {
    cout << "inside" << endl;
}
else {
    cout << "border" << endl;
}
```

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1362#12>

Глава 4.2. По-сложни проверки – изпитни задачи

В предходната глава се запознахме с **вложените условни конструкции** в езика C++. Чрез тях програмната логика в дадена програма може да бъде представена посредством **if конструкции**, които се съдържат една в друга. Разглеждахме и условната конструкция **switch-case**, която позволява избор измежду списък от възможности. Следва да упражним и затвърдим наученото досега, като разгледаме няколко по-сложни задачи, давани на изпити. Преди да преминем към задачите, ще си припомним условните конструкции:

Вложени проверки

```
if (условие1) {  
    if (условие2) {  
        // тяло;  
    } else {  
        // тяло;  
    }  
}
```



Запомнете, че не е добра практика писането на **дълбоко вложени условни конструкции** (с ниво на влагане повече от три). Избягвайте влагане на повече от три условни конструкции една в друга. Това усложнява кода и затруднява неговото четене и разбиране.

Switch-case проверки

Когато работата на програмата зависи от стойността на една променлива, вместо да правим последователни проверки с множество **if-else** блокове, можем да използваме условната конструкция **switch-case**:

```
switch (селектор)  
{  
    case стойност1:  
        конструкция;  
        break;  
    case стойност2:  
        конструкция;  
        break;  
    default:  
        конструкция;  
        break;  
}
```

Конструкцията се състои от:

- **Селектор** - израз, който се изчислява до някаква конкретна стойност. Типът на селектора може да бъде **цяло число** или **enum**.
- Множество **case етикети** с команди след тях, които могат да завършат с оператора **break**, но както вече знаем, той може да бъде изпуснат.

Изпитни задачи

Сега, след като си припомнихме как се използват условни конструкции и как се влагат една в друга условни конструкции, за реализиране на по-сложни проверки и програмна логика, нека решим няколко изпитни задачи.

Задача: навреме за изпит

Студент трябва да отиде **на изпит в определен час** (например в 9:30 часа). Той идва в изпитната зала в даден **час на пристигане** (например 9:40). Счита се, че студентът е дошъл **навреме**, ако е пристигнал в часа на изпита или до половин час **преди това**. Ако е пристигнал **по-рано**, повече от 30 минути, той е **подранил**. Ако е дошъл **след часа на изпита**, той е **закъснял**.

Напишете програма, която въвежда време на изпит и време на пристигане и отпечатва дали студентът е дошъл **навреме**, дали е **подранил** или е **закъснял**, както и **с колко часа или минути** е подранил или закъснял.

Входни данни

От конзолата се четат **четири цели числа** (по едно на ред):

- Първият ред съдържа **час на изпита** – цяло число от 0 до 23.
- Вторият ред съдържа **минута на изпита** – цяло число от 0 до 59.
- Третият ред съдържа **час на пристигане** – цяло число от 0 до 23.
- Четвъртият ред съдържа **минута на пристигане** – цяло число от 0 до 59.

Изходни данни

На първия ред отпечатайте:

- "Late", ако студентът пристига **по-късно** от часа на изпита.
- "On time", ако студентът пристига **точно** в часа на изпита или до 30 минути по-рано.
- "Early", ако студентът пристига повече от 30 минути **преди** часа на изпита.

Ако студентът пристига с поне минута разлика от часа на изпита, отпечатайте на следващия ред:

- "**mm minutes before the start**" за идване по-рано с по-малко от час.

- "hh:mm hours before the start" за подраняване с 1 час или повече. Минутите винаги печатайте с 2 цифри, например "1:05".
- "mm minutes after the start" за закъснение под час.
- "hh:mm hours after the start" за закъснение от 1 час или повече. Минутите винаги печатайте с 2 цифри, например "1:03".

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
9 30 9 50	Late 20 minutes after the start	16 00 15 00	Early 1:00 hours before the start	9 00 8 30	On time 30 minutes before the start
9 00 10 30	Late 1:30 hours after the start	14 00 13 55	On time 5 minutes before the start	10 00 10 00	On time

Насоки и подсказки



Препоръчително е да прочетете няколко пъти заданието на дадена задача, като си водите записи и си скицирате примерите, докато разсъждавате над тях, преди да започнете писането на код.

Обработка на входните данни

Съгласно заданието очакваме да ни бъдат подадени **четири** поредни реда с различни **цели числа**. Разглеждайки дадените входни параметри можем да се спрем на типа **int**, тъй като той удовлетворява очакваните ни стойности. След като сме избрали типа на променливите, трябва да ги **декларираме** и да **прочетем стойностите им** от конзолата:

```
int examHours, examMinutes, arrivalHours, arrivalMinutes;

cin >> examHours;
cin >> examMinutes;
cin >> arrivalHours;
cin >> arrivalMinutes;
```

Разглеждайки очаквания изход, можем да създадем променливи, които да съдържат различните видове изходни данни, с цел да избегнем използването на т.нар. "**magic strings**" в кода:

```
string late = "Late";
string onTime = "On time";
string early = "Early";
```

Изчисления

След като прочетохме входа, можем да започнем да разписваме логиката за изчисление на резултата. Нека първо да изчислим **началния час** на изпита **в минути** за по-лесно и точно сравнение:

```
int examTime = (examHours * 60) + examMinutes;
```

Нека изчислим по същата логика и времето на пристигане на студента:

```
int arrivalTime = (arrivalHours * 60) + arrivalMinutes;
```

Остава ни да пресметнем разликата в двете времена, за да можем да определим кога и с какво време спрямо изпита е пристигнал студентът:

```
int totalMinutesDifference = arrivalTime - examTime;
```

Следващата ни стъпка е да направим необходимите **проверки и изчисления**, като накрая ще изведем резултата от тях. Нека разделим изхода на **две** части:

- Първо ще покажем **кога е пристигнал** студентът - дали е подранил, закъснял или е пристигнал навреме. За целта ще се спрем на **if-else** конструкция.
- След това ще покажем **времевата разлика**, ако студентът пристигне в **различно време** от началния час на изпита.

С цел да спестим една допълнителна проверка (**else**), можем по подразбиране да приемем, че студентът е закъснял.

След което, съгласно условието, проверяваме дали разликата във времената е **повече от 30 минути**, което означава, че е дошъл повече от 30 минути по-рано. Ако това е така, приемаме, че е **подранил**. Ако не влезем в първото условие, то следва да проверим само дали **разликата е по-малка или равна на нула** (**≤ 0**), с което проверяваме условието, студентът да е дошъл в рамките на от **0 до 30 минути** преди изпита.

При всички останали случаи приемаме, че студентът е **закъснял**, което сме направили **по подразбиране**, и не е нужна допълнителна проверка:

```
string studentArrival = late;
if (totalMinutesDifference < -30) {
    studentArrival = early;
}
else if (totalMinutesDifference <= 0) {
    studentArrival = onTime;
}
```

За финал ни остава да разберем и покажем с **каква разлика от времето на изпита е пристигнал**, както и дали тази разлика показва време на пристигане **преди или след изпита**.

Правим проверка дали разликата ни е **над** един час, за да изпишем съответно часовете и минутите в желания **формат**, или е **под** един час, за да изпишем **само минутите**. Остава да направим още една проверка - дали времето на пристигане на студента е **преди** или **след** началото на изпита:

```
string result;
if (totalMinutesDifference != 0) {
    int hoursDifference =
        abs(totalMinutesDifference / 60);
    int minutesDifference =
        abs(totalMinutesDifference % 60);

    if (hoursDifference > 0) {
        //result = "{hoursDifference}:" +
        result = to_string(hoursDifference) + ":";

        //result = "{hoursDifference}:0" / "{hoursDifference}:" +
        if (minutesDifference < 10) {
            result += "0";
        }

        //result = "{hoursDifference}:{minutesDifference} hours"
        result += to_string(minutesDifference) + " hours";
    }
    else {
        result = to_string(minutesDifference) + " minutes";
    }

    if (totalMinutesDifference < 0) {
        result += " before the start";
    }
    else {
        result += " after the start";
    }
}
```

Отпечатване на резултата

Накрая остава да изведем резултата на конзолата. По задание, ако студентът е

дошъл точно на време (**без нито една минута разлика**), не трябва да изваждаме втори резултат. Затова правим следната проверка:

```
cout << studentArrival << endl;
if (result != "") {
    cout << result << endl;
}
```

Не забравяйте да добавите библиотеката **<string>**, когато искате да работите с текстови данни.

Реално за целите на задачата извеждането на резултата **на конзолата** може да бъде направен и в по-ранен етап - още при самите изчисления. Това като цяло не е много добра практика. **Защо?**

Нека разгледаме идеята, че кодът ни не е 10 реда, а 100 или 1000! Някой ден ще се наложи извеждането на резултата да не бъде в конзолата, а да бъде записан във **файл** или показан на **уеб приложение**. Тогава на колко места в кода ще трябва да бъдат нанесени корекции поради тази смяна? И дали няма да пропуснем някое място?



Винаги си мислете за кода с логическите изчисления, като за отделна част, различна от обработката на входните и изходните данни. Той трябва да може да работи без значение как му се подават данните и къде ще трябва да бъде показан резултатът.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1363#0>.

Задача: пътешествие

Странно, но повечето хора си планират от рано почивката. Млад програмист разполага с **определен бюджет** и свободно време в даден **сезон**.

Напишете програма, която да приема **на входа бюджета и сезона**, а **на изхода** да изкарва **къде ще почива** програмистът и **колко ще похарчи**.

Бюджетът определя дестинацията, а сезонът определя колко от бюджета ще бъде изхарчен. Ако е лято, ще почива на къмпинг, а зимата - в хотел. Ако е в Европа, независимо от сезона, ще почива в хотел. Всеки къмпинг или хотел, според дестинацията, има собствена цена, която отговаря на даден **процент от бюджета**:

- При **100 лв.** или **по-малко** – някъде в **България**.
 - Лято – **30%** от бюджета.
 - Зима – **70%** от бюджета.
- При **1000 лв.** или **по малко** – някъде на **Балканите**.

- Лято – 40% от бюджета.
- Зима – 80% от бюджета.
- При повече от 1000 лв. – някъде из Европа.
 - При пътуване из Европа, независимо от сезона, ще похарчи 90% от бюджетата.

Входни данни

Входът се чете от конзолата и се състои от **два реда**:

- На първия ред получаваме **бюджета** – реално число в интервал [10.00 ... 5000.00].
- На втория ред – **един** от двета възможни сезона: "summer" или "winter".

Изходни данни

На конзолата трябва да се отпечатат **два реда**:

- На първи ред – "Somewhere in {дестинация}" измежду "Bulgaria", "Balkans" и "Europe".
- На втори ред – "{Вид почивка} – {Похарчена сума}".
 - Потребителят може да е между "Camp" и "Hotel".
 - Сумата трябва да е закръглена с точност до втория символ след десетичния знак.

Примерен вход и изход

Вход	Изход
50 summer	Somewhere in Bulgaria Camp – 15.00
312 summer	Somewhere in Balkans Camp – 124.80

Вход	Изход
75 winter	Somewhere in Bulgaria Hotel – 52.50
1500 summer	Somewhere in Europe Hotel – 1350.00

Насоки и подсказки

Типично, както и при другите задачи, можем да разделим решението на няколко части:

- Четене на входните данни
- Изчисления
- Отпечатване на резултата

Обработка на входните данни

Прочитайки внимателно условието, разбираме, че очакваме **два** реда с входни данни. Първият параметър е **реално число**, за което е добре да изберем подходящ тип на променливата. Ще се спрем на **double** като тип за бюджета, а за сезона - **string**:

```
double budget;
```

```
cin >> budget;
```

```
string season;
```

```
cin >> season;
```



Винаги преценявайте какъв **тип стойност** се подава при входните данни, за да работят правилно създадените от вас програмни конструкции!

Изчисления

Нека си създадем и инициализираме нужните за логиката и изчисленията променливи:

```
string destinationResult, holidayInformation, restingPlace;
double moneySpent = 0;
```

Разглеждайки отново условието на задачата, забелязваме, че основното разпределение за това къде ще почиваме се определя от **стойността на подадения бюджет**, т.е. основната ни логика се разделя на два случая:

- Ако бюджетът е **по-малък** от дадена стойност.
- Ако е **по-малък** от друга стойност, или е **повече** от дадена гранична стойност.

Спрямо това как си подредим логическата схема (в какъв ред ще обхождаме граничните стойности), ще имаме повече или по-малко проверки в условията. **Помислете защо!**

След това е необходимо да направим проверка за стойността на **подадения сезон**. Спрямо нея ще определим какъв процент от бюджета ще бъде похарчен, както и къде ще почива програмистът - в **хотел** или на **къмпинг**. Пример за един от възможните подходи за решение е:

```
if (budget <= 100) {
    destinationResult = "Bulgaria";
    if (season == "summer") {
        moneySpent = 0.3 * budget;
        restingPlace = "Camp";
    }
}
```

```

    else {
        moneySpent = 0.7 * budget;
        restingPlace = "Hotel";
    }
}

else if (budget <= 1000) {
    destinationResult = "Balkans";
    if (season == "summer") {
        moneySpent = 0.4 * budget;
        restingPlace = "Camp";
    }
    else {
        moneySpent = 0.8 * budget;
        restingPlace = "Hotel";
    }
}

else {
    destinationResult = "Europe";
    moneySpent = 0.9 * budget;
    restingPlace = "Hotel";
}
}

```

Винаги можем да инициализираме дадена стойност на параметъра и след това да направим само една проверка. Това ни спестява една логическа стъпка.

Например следният блок:

```

if (budget <= 100) {
    destinationResult = "Bulgaria";
    if (season == "summer") {
        moneySpent = 0.3 * budget;
        restingPlace = "Camp";
    }
    else {
        moneySpent = 0.7 * budget;
        restingPlace = "Hotel";
    }
}

```

може да бъде съкратен до следния вид:

```

destinationResult = "Bulgaria";
moneySpent = 0.7 * budget;
restingPlace = "Hotel";

if (season == "summer") {
    moneySpent = 0.3 * budget;
    restingPlace = "Camp";
}

```

Отпечатване на резултата

Остава да покажем изчисления резултат на конзолата:

```

cout << "Somewhere in " << destinationResult << endl;
cout << restingPlace << " - " <<
    fixed << setprecision(2) << moneySpent << endl;

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1363#1>.

Задача: операции между числа

Напишете програма, която чете **две цели числа (n1 и n2)** и **оператор**, с който да се извърши дадена математическа операция с тях. Възможните операции са: **събиране (+)**, **изважддане (-)**, **умножение (*)**, **деление (/)** и **модулно деление (%)**. При събиране, изважддане и умножение на конзолата трябва да се отпечата резултата и дали той е **четен** или **нечетен**. При обикновено деление – **единствено резултата**, а при модулно деление – **остатъка**. Трябва да се има предвид, че **делителят може да е равен на нула (= 0)**, а на нула не се дели. В този случай трябва да се отпечата специално съобщение.

Входни данни

От конзолата се прочитат **3 реда**:

- N1 – цяло число в интервала [0 ... 40 000].
- N2 – цяло число в интервала [0 ... 40 000].
- Оператор – един символ измежду: "+", "-", "*", "/", "%".

Изходни данни

Да се отпечата на конзолата **един ред**:

- Ако операцията е **събиране, изважддане или умножение**:
 - "{N1} {оператор} {N2} = {резултат} – {even/odd}".

- Ако операцията е **деление**:
 - " $\{N1\} / \{N2\} = \{\text{результат}\}$ " – резултатът е форматиран до втория символ след десетичния знак.
- Ако операцията е **модулно деление**:
 - " $\{N1\} \% \{N2\} = \{\text{остатък}\}$ ".
- В случай на **деление на 0** (нула):
 - "Cannot divide $\{N1\}$ by zero".

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
123 12 /	123 / 12 = 10.25	112 0 /	Cannot divide 112 by zero	10 12 +	10 + 12 = 22 - even
10 3 %	10 % 3 = 1	10 0 %	Cannot divide 10 by zero	10 1 -	10 - 1 = 9 - odd

Насоки и подсказки

Задачата не е сложна, но има доста редове код за писане. Както вече разбрахме, е **добра практика да разделяме кода си на части**, но при тази задача това само би я **усложнено**.

Обработка на входните данни

След прочитане на условието разбираме, че очакваме **три** реда с входни данни. На първите **два** реда ни се подават **цели числа** (в указания от заданието диапазон), а на третия - **аритметичен символ**:

```
double n1, n2;
cin >> n1;
cin >> n2;

char nOperator;
cin >> nOperator;
```

Изчисления и отпечатване на резултата

Нека си създадем и инициализираме променлива, в която ще пазим **резултата от изчисленията**:

```
double result = 0;
```

Прочитайки внимателно условието, разбираме, че има случаи, в които не трябва да правим **ниакви** изчисления, а просто да изведем резултат. Следователно първо може да проверим дали второто число е **0** (нула), както и дали операцията е **деление** или **модулно деление**, след което да принтираме необходимото съобщение:

```
if (n2 == 0 && (nOperator == '/' || nOperator == '%')) {
    cout << "Cannot divide " << n1 << " by zero" << endl;
}
```

От условието можем да видим, че за **събиране (+)**, **изваждане (-)** или **умножение (*)** очакваният резултат има еднаква структура: "**{n1} {оператор} {n2} = {резултат} - {even/odd}**", докато за **деление (/)** и за **модулно деление (%)** резултатът има различна структура.

В C++ не можем да използваме оператора за **модулно деление (%)** с реални числа, затова ще трябва да използваме функцията за модулно деление **fmod(...)**, а за нея ще ни е необходима и библиотеката **<cmath>**:

```
else if (nOperator == '/') {
    result = n1 / n2;
    cout << n1 << " " << nOperator << " " << n2 <<
        " = " << fixed << setprecision(2) << result << endl;
}
else if (nOperator == '%') {
    result = fmod(n1, n2);
    cout << n1 << " " << nOperator << " " << n2 <<
        " = " << result << endl;
}
```

Завършваме с проверките за събиране, изваждане и умножение:

```
else {
    if (nOperator == '+') {
        result = n1 + n2;
    }
    else if (nOperator == '-') {
        result = n1 - n2;
    }
    else if (nOperator == '*') {
        result = n1 * n2;
    }
}
```

```

cout << n1 << " " << nOperator << " " << n2 <<
    " = " << result << " - " <<
    (fmod(result, 2) == 0 ? "even" : "odd") << endl;
}

```

При кратки и ясни проверки, както в горния пример за четно и нечетно число, е възможно да се използва **тернарен оператор**. Нека разгледаме възможната проверка с и без тернарен оператор.

Без използване на тернарен оператор кодът е по-дълъг, но се чете лесно:

```

string numberIs;
if (fmod(result, 2) == 0) {
    numberIs = "even";
}
else {
    numberIs = "odd";
}

```

С използване на тернарен оператор кодът е много по-кратък, но може да изиска допълнителни усилия, за да бъде прочетен и разбран като логика:

```
string numberIs = fmod(result, 2) == 0 ? "even" : "odd";
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1363#2>.

Задача: билети за мач

Група запалянковци решили да си закупят билети за Евро 2016. Цената на билета се определя спрямо **две** категории:

- VIP – 499.99 лева.
- Normal – 249.99 лева.

Запалянковците имат определен бюджет, а броят на хората в групата определя какъв процент от бюджета трябва да се задели за транспорт:

- От 1 до 4 – 75% от бюджета.
- От 5 до 9 – 60% от бюджета.
- От 10 до 24 – 50% от бюджета.
- От 25 до 49 – 40% от бюджета.
- 50 или повече – 25% от бюджета.

Напишете програма, която да пресмята дали с останалите пари от бюджета могат да си купят билети за избраната категория, както и колко пари ще им

останат или ще са им нужни.

Входни данни

Входът се чете от конзолата и съдържа **точно 3 реда**:

- На **първия** ред е **бюджетът** – реално число в интервала [1 000.00 ... 1 000 000.00].
- На **втория** ред е **категорията** – "VIP" или "Normal".
- На **третия** ред е **броят на хората в групата** – цяло число в интервала [1 ... 200].

Изходни данни

Да се отпечата на конзолата **един ред**:

- Ако бюджетът е достатъчен:
 - "Yes! You have {N} leva left." – където N са останалите пари на групата.
- Ако бюджетът НЕ Е достатъчен:
 - "Not enough money! You need {M} leva." – където M е сумата, която не достига.

Сумите трябва да са форматирани с точност до два символа след десетичния знак.

Примерен вход и изход

Вход	Изход	Обяснения
1000 Normal 1	Yes! You have 0.01 leva left.	1 човек: 75% от бюджета отиват за транспорт. Остават: 1000 - 750 = 250. Категория Normal: билетът струва 249.99 * 1 = 249.99 249.99 < 250: остават му 250 - 249.99 = 0.01
Вход	Изход	Обяснения
30000 VIP 49	Not enough money! You need 6499.51 leva.	49 човека: 40% от бюджета отиват за транспорт. Остават: 30000 - 12000 = 18000. Категория VIP: билетът струва 499.99 * 49. 24499.51000000002 < 18000. Не стигат 24499.51 - 18000 = *6499.51

Насоки и подсказки

Ще прочетем входните данни и ще извършим изчисленията, описани в условието на задачата, за да проверим дали ще стигнат парите.

Обработка на входните данни

Нека прочетем внимателно условието и да разгледаме какво се очаква да получим като **входни данни**, какво се очаква да **върнем като резултат**, както и кои са **основните стъпки** при разбиването **на логическата схема**. Като за начало, нека обработим и запазим входните данни в **подходящи** за това **променливи**:

```
double budget;
cin >> budget;

string ticketType;
cin >> ticketType;

int people;
cin >> people;
```

Изчисления

Нека създадем и инициализираме нужните за изчисленията променливи:

```
double transportCharges = 0;
double moneyForTickets = 0;
double moneyDifference = 0;
```

Нека отново прегледаме условието. Трябва да направим **две** различни блок изчисления. От първите изчисления трябва да разберем каква част от бюджета ще трябва да заделим за **транспорт**. За логиката на тези изчисления забелязваме, че има значение единствено **броят на хората в групата**. Следователно ще направим логическата разбивка спрямо броя на запалянковците. Ще използваме условна конструкция - поредица от **if-else** блокове:

```
if (people <= 4) {
    transportCharges = 0.75 * budget;
}
else if (people <= 9) {
    transportCharges = 0.6 * budget;
}
else if (people <= 24) {
    transportCharges = 0.5 * budget;
}
```

```

else if (people <= 49) {
    transportCharges = 0.4 * budget;
}
else {
    transportCharges = 0.25 * budget;
}

```

От вторите изчисления трябва да намерим каква сума ще ни е необходима за закупуване на **билети за групата**. Според условието, това зависи единствено от типа на билетите, които трябва да закупим. Нека използваме **if-else** условна конструкция:

```

if (ticketType == "VIP") {
    moneyForTickets = people * 499.99;
}
else {
    moneyForTickets = people * 249.99;
}

```

След като сме изчислили какви са транспортните разходи и разходите за билети, ни остава да изчислим крайния резултат и да разберем ще успее ли групата от запалянковци да отиде на Евро 2016 или няма да успее при така подадените параметри:

```
moneyDifference = budget - transportCharges - moneyForTickets;
```

За извеждането на резултата, за да си спестим една **else** проверка в конструкцията, приемаме, че групата по подразбиране ще може да отиде на Евро 2016:

```

else {
    cout << "Yes! You have " <<
        fixed << setprecision(2) << moneyDifference <<
        " leva left." << endl;
}

```

Отпечатване на резултата

Накрая ни остава да покажем изчисления резултат на конзолата.

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1363#3>.

Задача: хотелска стая

Хотел предлага **два вида стаи: студио и апартамент**.

Напишете програма, която изчислява цената за целия престой за студио и апартамент. Цените зависят от **месеца** на престоя:

Май и октомври	Юни и септември	Юли и август
Студио – 50 лв./нощувка	Студио – 75.20 лв./нощувка	Студио – 76 лв./нощувка
Апартамент – 65 лв./нощувка	Апартамент – 68.70 лв./нощувка	Апартамент – 77 лв./нощувка

Предлагат се и следните **отстъпки**:

- За **студио**, при **повече** от 7 нощувки през **май и октомври**: 5% намаление.
- За **студио**, при **повече** от 14 нощувки през **май и октомври**: 30% намаление.
- За **студио**, при **повече** от 14 нощувки през **юни и септември**: 20% намаление.
- За **апартамент**, при **повече** от 14 нощувки, **без значение** от месеца: 10% намаление.

Входни данни

Входът се чете от конзолата и съдържа **точно** два реда:

- На първия ред е **месецът** – May, June, July, August, September или October.
- На втория ред е **броят на нощувките** – цяло число в интервала [0 ... 200].

Изходни данни

Да се **отпечатат** на конзолата два реда:

- На първия ред: "Apartment: { цена за целият престой } lv".
- На втория ред: "Studio: { цена за целият престой } lv".

Цената за целия престой да е форматирана с точност до два символа след десетичния знак.

Примерен вход и изход

Вход	Изход	Обяснения
May 15	Apartment: 877.50 lv.	През май , при повече от 14 нощувки , намаляваме цената на студиото с 30% ($50 - 15 = 35$), а на апартамента – с 10% ($65 - 6.5 = 58.5$).

Вход	Изход	Обяснения
	Studio: 525.00 lv.	Целият престой в апартамент - 877.50 лв. Целият престой в студио - 525.00 лв.

Вход	Изход
June 14	Apartment: 961.80 lv. Studio: 1052.80 lv

Вход	Изход
August 20	Apartment: 1386.00 lv. Studio: 1520.00 lv.

Насоки и подсказки

Ще прочетем входните данни и ще извършим изчисленията според описания ценоразпис и правилата за отстъпките и накрая ще отпечатаме резултата.

Обработка на входните данни

Съгласно условието на задачата очакваме да получим два реда входни данни - на първия ред **месец**, през който се планува престой, а на втория - **броя нощувки**. Нека обработим и запазим входните данни в подходящи за това променливи:

```
string month;
cin >> month;

int nights;
cin >> nights;
```

Изчисления

След това да създадем и инициализираме нужните за изчисленията променливи:

```
double studioPrice = 0;
double apartmentPrice = 0;
double studioRent = 0;
double apartmentRent = 0;
```

Разглеждайки отново условието забелязваме, че основната ни логика зависи от това какъв **месец** ни се подава, както и от броя на **нощувките**.

Като цяло има различни подходи и начини да се направят въпросните проверки и обикновено бихме използвали условна конструкция **switch-case** с **if** и **if-else** в различните **case** блокове, но тъй като **C++** не позволява за селектора да бъде използвана променлива от типа **string**, ще трябва да използваме само **if** и **if-else** конструкции.

Нека започнем с първата група месеци: **Май** и **Октомври**. За тези два месеца цената на престой е **еднаква** и за двета типа настаняване - в **студио** и в **апартамент**.

Съответно остава само да направим вътрешна проверка спрямо броят нощувки, за да преизчислим **съответната цена** (ако се налага):

```
if (month == "May" || month == "October") {
    studioPrice = 50;
    apartmentPrice = 65;

    studioRent = studioPrice * nights;
    apartmentRent = apartmentPrice * nights;

    if (nights > 14) {
        studioRent *= 0.7;
        apartmentRent *= 0.9;
    }
    else if (nights > 7) {
        studioRent *= 0.95;
    }
}
```

За следващите месеци логиката и изчисленията ще са донякъде **идентични**:

```
else if (month == "June" || month == "September") {
    studioPrice = 75.2;
    apartmentPrice = 68.7;

    studioRent = studioPrice * nights;
    apartmentRent = apartmentPrice * nights;

    if (nights > 14) {
        studioRent *= 0.8;
        apartmentRent *= 0.9;
    }
}

else if (month == "July" || month == "August") {
    studioPrice = 76;
    apartmentPrice = 77;

    studioRent = studioPrice * nights;
    apartmentRent = apartmentPrice * nights;
```

```
| if (nights > 14) {  
|     apartmentRent *= 0.9;  
}  
}
```

Отпечатване на резултата

Накрая остава да покажем изчислените резултати на конзолата, като не забравяме, че цената за престоя трябва да е форматирана с точност до два символа след десетичния знак:

```
cout << "Apartment: " <<  
    fixed << setprecision(2) << apartmentRent <<  
    " lv." << endl;  
cout << "Studio: " <<  
    fixed << setprecision(2) << studioRent <<  
    " lv." << endl;
```

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1363#4>.

Глава 5.1. Повторения (цикли)

В настоящата глава ще се запознаем с конструкциите за **повторение на група команди**, известни в програмирането с понятието "цикли". Ще напишем няколко цикъла с използване на оператора **for** в класическата му форма. Накрая ще решим няколко практически задачи, изискващи повторение на поредица от действия, като използваме цикли.

Видео

Гледайте видео урок по учебния материал от настоящата глава от книгата: https://www.youtube.com/watch?v=7D7fECmq_VQ.

Повторения на блокове код (for цикъл)

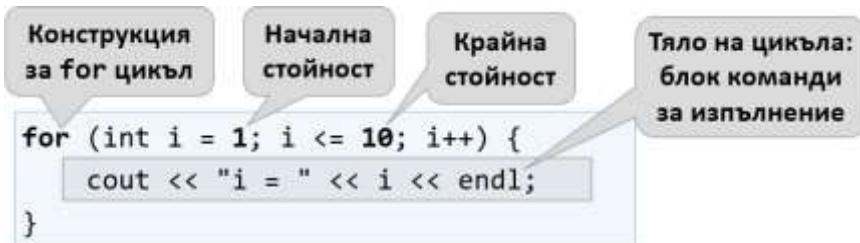
В програмирането често пъти се налага да изпълним блок с команди няколко пъти. За целта се използват т.нр. **цикли**. Нека разгледаме един пример за **for** цикъл, който преминава последователно през числата от 1 до 10 и ги отпечатва:

```
for (int i = 1; i <= 10; i++) {  
    cout << "i = " << i << endl;  
}
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/ForLoopExample>.

Цикълът започва с **оператора for** и преминава през всички стойности за дадена променлива в определен интервал, например всички числа от 1 до 10 включително, и за всяка стойност изпълнява поредица от команди.

В декларацията на цикъла трябва да се зададе **начална стойност** и **крайна стойност**. **Тялото на цикъла** обикновено се огражда с къдрави скоби **{ }** и представлява блок с една или няколко команди. На фигурана по-долу е показана структурата на един **for** цикъл:



Целта на цикъла е да се премине последователно през числата 1, 2, 3, ..., n и за всяко от тях да се изпълни някакво действие. В примера по-горе създаваме локалната променливата **i** с начална стойност 1 (**int i = 1**) и увеличаваме стойността на променливата с 1 (**i++**) всеки път, когато кодът между къдравите скоби се изпълни. Цикълът се повтаря, докато условието **i <= 10** е изпълнено

(**true**). Всяко от тези повторения се нарича "итерация". В примера по-горе променливата **i** приема стойности от 1 до 10 включително и в тялото на цикъла се отпечатва текущата стойност.

Пример: числа от 1 до 100

Да се напише програма, която **печатат числата от 1 до 100**. Програмата не приема вход и отпечатва числата от 1 до 100 едно след друго, по едно на ред.

Насоки и подсказки

Можем да решим задачата с **for цикъл**, с който преминаваме с променливата **i** през числата от 1 до 100 и ги отпечатваме в тялото на цикъла:

```
for (int i = 1; i <= 100; i++) {
    cout << i << endl;
}
```

Стартираме програмата с [Ctrl + F5] и я **тестваме**:

The screenshot shows the Microsoft Visual Studio Debug Console window. The title bar says "Microsoft Visual Studio Debug Console". The console output shows the numbers 93, 94, 95, 96, 97, 98, 99, and 100, each on a new line. At the bottom of the window, the text "C:\Users\Andrey\source\repos\Loops\Debug\OneToHundred.exe (process 15108) exited with code 0. Press any key to close this window . . ." is displayed.

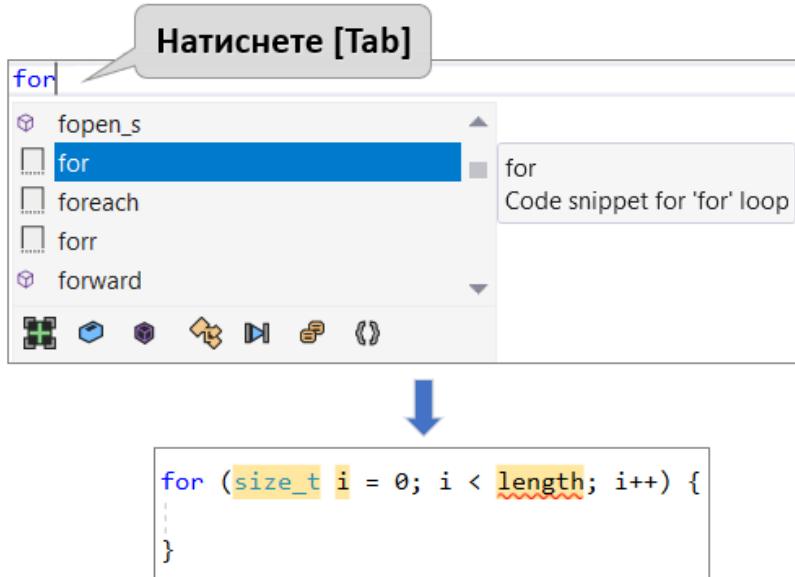
Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1364#0>.

Трябва да получите **100 точки** (напълно коректно решение).

Code Snippet за for цикъл във Visual Studio

Когато програмираме, често се налага да пишем цикли. Затова в повечето среди за разработка (IDE) има шаблони за код (code snippets) за писане на цикли. Един такъв шаблон е **шаблонът за for цикъл** във Visual Studio. Напишете **for** в редактора за C++ код във Visual Studio и натиснете един пъти [Tab]. Visual Studio ще разгъне за вас шаблон и ще напише цялостен **for цикъл**:



Опитайте сами, за да усвоите умението да ползвате шаблона за код за **for** цикъл във Visual Studio.

Пример: числа до 1000, завършващи на 7

Да се напише програма, която намира всички числа в интервала [1 ... 1000], които завършват на 7.

Насоки и подсказки

Задачата можем да решим като комбинираме **for** цикъл за преминаваме през числата от 1 до 1000 и проверка за всяко число дали завършва на 7. Има и други решения, разбира се, но нека решим задачата чрез завъртане на цикъл + проверка:

```
for (int i = 1; i <= 1000; i++) {  
    if (i % 10 == 7) {  
        // TODO: print i  
    }  
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1364#1>.

Пример: всички латински букви

Да се напише програма, която отпечатва буквите от латинската азбука: **a, b, ..., z**.

Насоки и подсказки

Символите в C++ биват запазени в паметта като числа. За конвертирането им от число в символ и обратно се използва така наречената **ASCII таблица**. Например символът **a** (малко "a") се запазва в компютърната памет като числото 97, а **z** (малко "z") - като 122. Това означава, че можем да създадем цикъл, който използва тип данни **char**, и да променяме символа, като увеличаваме числото, което му съответства:

```
cout << "Latin alphabet: ";
for (char i = 'a'; i <= 'z'; i++) {
    cout << i << endl;
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1364#2>.

Пример: събиране на числа

Да се напише програма, която **въвежда n** цели числа и ги събира.

- От първия ред на входа се въвежда броят числа **n**.
- От следващите **n** реда се въвежда по едно число.
- Числата се събират и накрая се отпечатва резултатът.

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
2		3		4	
10	30	-10		45	
20		-20	-60	-20	43
		-30		7	
				11	

Насоки и подсказки

Можем да решим задачата за събиране на числа по следния начин:

- Четем входното число **n** от конзолата.
- Започваме първоначално със сбор **sum = 0**.
- Създаваме цикъл от 0 до **n**. На всяка итерация от цикъла четем число **num** и го добавяме към сумата **sum**.
- Накрая отпечатваме полученият сбор **sum**.

Ето и сурс код на описаната идея:

```
int n, sum = 0;
cout << "How many numbers do you want to sum? ";
cin >> n;

cout << "Enter numbers:" << endl;
for (int i = 0; i < n; i++) {
    int num;
    cin >> num;
    sum = sum + num;
}

cout << "The sum is " << sum << endl;
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1364#3>.

Пример: най-голямо число

Да се напише програма, която въвежда **n** цели числа ($n > 0$), намира и отпечатва **най-голямото** измежду тях. На първия ред на входа се въвежда броят числа **n**. След това се въвеждат самите числа, по едно на ред. Примери:

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
2 100 99	100	3 -10 20 -30	20	4 45 -20 7 99	99

Насоки и подсказки

Можем да решим задачата за най-голямо число по следния начин:

- Въвеждат се броят числа **n**.
- Ще прочетем първото число отделно и ще инициализираме променливата **max** със стойността му, за да можем да сравняваме следващите числа с нея.
- Създаваме цикъл започващ от 1 (за да прескочим първото число, което прочетохме отделно) до **n**.
- На всяка стъпка прочитаме число и проверяваме дали то е по-голямо от

последното открыто най-голямо (т.е. променливата **max**) и ако е - запазваме стойността му в променливата **max**.

- Накрая отпечатваме променливата, в която пазим най-голямото число.

```
int n, max;
cin >> n;
cin >> max;

for (int i = 1; i < n; i++) {
    int number;
    cin >> number;

    if (number > max) {
        max = number;
    }
}

cout << max << endl;
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1364#4>.

Пример: най-малко число

Да се напише програма, която въвежда **n** цели числа ($n > 0$) и намира най-малкото измежду тях. Първо се въвежда броя числа **n**, след тях още **n** числа по едно на ред.

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
2 100 99	99	3 -10 20 -30	-30	4 45 -20 7 99	-20

Насоки и подсказки

Задачата е абсолютно аналогична с предходната, само че с различен знак за проверка:

```

int n, min;
cin >> n;
cin >> min;

for (int i = 1; i < n; i++) {
    int number;
    cin >> number;

    if (number < min) {
        min = number;
    }
}

cout << min << endl;

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1364#5>.

Пример: ляв и десен сбор

Да се напише програма, която въвежда $2 * n$ цели числа и проверява дали **сборът на първите n числа** (лев събор) е равен на **сбора на вторите n числа** (десен събор). При равенство се отпечатва "Yes" + **сбора**, иначе се отпечатва "No" + **разликата**. Разликата се изчислява, като положително число (по абсолютна стойност). Форматът на изхода трябва да е като в примерите по-долу.

Примерен вход и изход

Вход	Изход
2	
10	
90	Yes, sum = 100
60	
40	

Вход	Изход
2	
90	
9	No, diff = 1
50	
50	

Насоки и подсказки

Първо въвеждаме числото n , след това първите n числа (**лявата** половина) и ги събираме. Продължаваме с въвеждането на още n числа (**дясната** половина) и намираме и техния сбор. Изчисляваме **разликата** между намерените сборове по абсолютна стойност: **abs(leftSum - rightSum)**. Ако разликата е 0, отпечатваме "Yes, sum = " + **сбора**, в противен случай - отпечатваме "No, diff = " + **разликата**:

```

int n, leftSum = 0, rightSum = 0;
cin >> n;

for (int i = 0; i < n; i++) {
    int num;
    cin >> num;
    leftSum += num;
}

// TODO: Read and calculate the rightSum

if (leftSum == rightSum) {
    cout << "Yes, sum = " << leftSum << endl;
}
else {
    int diff = abs(leftSum - rightSum);
    cout << "No, diff = " << diff << endl;
}

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1364#6>.

Пример: четен / нечетен сбор

Да се напише програма, която въвежда **n** цели числа и проверява дали **сборът на числата на четни позиции** е равен на **сбора на числата на нечетни позиции**. При равенство отпечатва "Yes" + **сбора**, в противен случай отпечатва "No" + **разликата**. Разликата се изчислява по абсолютна стойност. Форматът на изхода трябва да е като в примерите по-долу.

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
4		4		3	
10		3		5	No
50	Yes	5		8	
60	Sum = 70	1	Diff = 1	1	Diff = 2
20		-2			

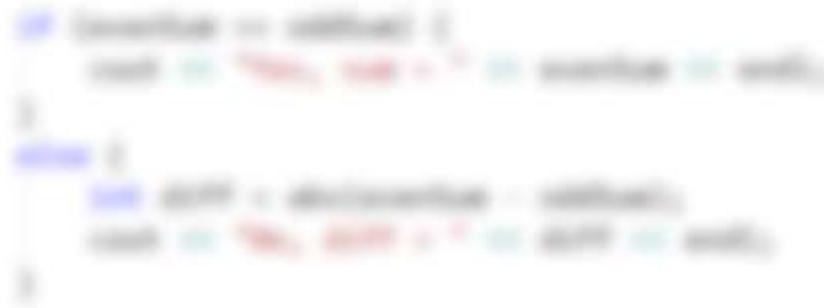
Насоки и подсказки

Въвеждаме числата едно по едно и изчисляваме двата **сбора** (на числата на **четни** позиции и на числата на **нечетни** позиции). Както в предходната задача - изчисляваме абсолютната стойност на разликата и отпечатваме резултата ("Yes" + **сбора** при разлика 0 или "No" + **разликата** в противен случай):

```
int n, evenSum = 0, oddSum = 0;
cin >> n;

for (int i = 0; i < n; i++) {
    int num;
    cin >> num;

    if (i % 2 == 0) {
        evenSum += num;
    }
    else {
        oddSum += num;
    }
}
```



Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1364#7>.

Пример: събиране на гласните букви

Да се напише програма, която въвежда **текст** (стринг), изчислява и отпечатва **сбора** от **стойностите на гласните букви** според таблицата по-долу:

a	e	i	o	u
1	2	3	4	5

Примерен вход и изход

Вход	Изход
hello	6 (e+o = 2+4 = 6)
hi	3 (i = 3)

Вход	Изход
bamboo	9 (a+o+o = 1+4+4 = 9)
beer	4 (e+e = 2+2 = 4)

Насоки и подсказки

Прочитаме входния текст **text**, инициализираме променлива за сбара и създаваме цикъл, който преминава през всеки символ от входния текст (от 0 до **text.length()**). Обхождаме въведените символи, като проверяваме дали текущия символ (**text[i]**) е гласна буква и ако е - добавяме съответното число към сбара:

```

string text;
getline(cin, text);

int sum = 0;

for (int i = 0; i < text.length(); i++) {
    char currentLetter = text[i];

    if (currentLetter == 'a') {
        sum += 1;
    }
    else if (currentLetter == 'e') {
        sum += 2;
    }
    else if (currentLetter == 'i') {
        sum += 3;
    }
    else if (currentLetter == 'o') {
        sum += 4;
    }
    else if (currentLetter == 'u') {
        sum += 5;
    }
}

cout << "Vowels sum = " << sum << endl;
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1364#8>.

Какво научихме от тази глава?

Можем да повтаряме блок код с **for** цикъл:

```
for (int i = 1; i <= 10; i++) {
    cout << "i = " << i << endl;
}
```

Можем да четем поредица от **n** числа от конзолата:

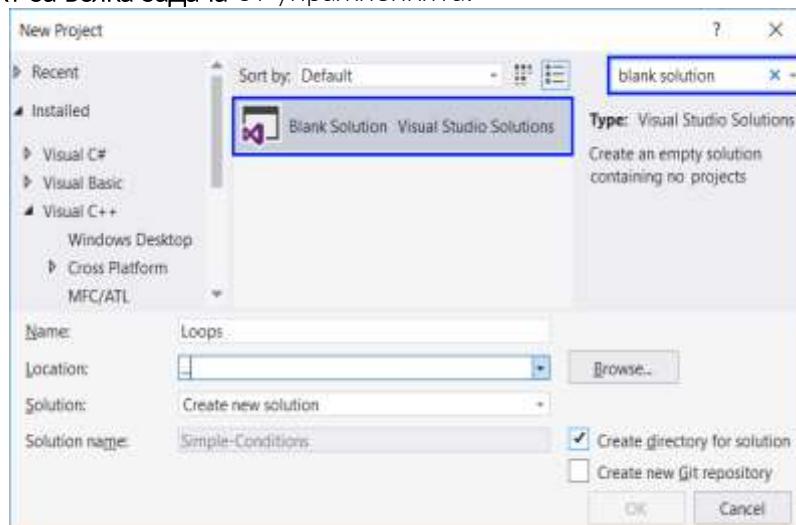
```
int n;
cin >> n;
for (int i = 0; i < n; i++) {
    int number;
    cin >> number;
}
```

Упражнения: повторения (цикли)

След като се запознахме с циклите, идва време да затвърдим знанията си на практика, а както знаете, това става с много писане на код. Да решим няколко задачи за упражнение.

Празно Visual Studio решение (Blank Solution)

Създаваме празно решение (**Blank Solution**) във Visual Studio, за да организираме по-добре задачите за упражнение. Целта на този **Blank Solution** е да съдържа **по един проект за всяка задача** от упражненията:



Задаваме да се стартира по подразбиране текущият проект (не първият в решението). Кликваме с десен бутон на мишката върху [Solution 'Loops'] -> [Set StartUp Projects...] -> [Current selection].

Задача: елемент, равен на сбора на останалите

Да се напише програма, която въвежда **n** цели числа и проверява дали сред тях съществува число, което е равно на сбора на всички останали. Ако има такъв елемент, се отпечатва "Yes Sum = " + неговата стойност, в противен случай - "No Diff = " + разликата между най-големия елемент и сбора на останалите (по абсолютна стойност).

Примерен вход и изход

Вход	Изход	Коментар	Вход	Изход	Коментар
7 3 4 1 1 2 12 1	Yes Sum = 12	$3 + 4 + 1 + 2 + 1 + 1 = 12$	3 1 1 10	No Diff = 8	$ 10 - (1 + 1) = 8$

Вход	Изход	Вход	Изход	Вход	Изход
3 1 1 1	No Diff = 1	3 5 5 1	No Diff = 1	4 6 1 2 3	Yes Sum = 6

Насоки и подсказки

Трябва да изчислим **сбора** на всички елементи, да намерим **най-големия** от тях и да проверим търсеното условие.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1364#9>.

Задача: четни / нечетни позиции

Напишете програма, която чете **n** числа и пресмята **сбора, минимума и максимума** на числата на **четни и нечетни** позиции (броим от 1). Когато няма минимален / максимален елемент, отпечатайте "No".

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
6 2 3 5 4 2 1	OddSum=9, OddMin=2, OddMax=5, EvenSum=8, EvenMin=1, EvenMax=4	2 1.5 -2.5	OddSum=1.5, OddMin=1.5, OddMax=1.5, EvenSum=-2.5, EvenMin=-2.5, EvenMax=-2.5	1 1	OddSum=1, OddMin=1, OddMax=1, EvenSum=0, EvenMin>No, EvenMax>No

Вход	Изход	Вход	Изход	Вход	Изход
3 -1 -2 -3	OddSum=-4, OddMin=-3, OddMax=-1, EvenSum=-2, EvenMin=-2, EvenMax=-2	1 -5	OddSum=-5, OddMin=-5, OddMax=-5, EvenSum=0, EvenMin>No, EvenMax>No	5 3 -2 8 11 -3	OddSum=8, OddMin=-3, OddMax=8, EvenSum=9, EvenMin=-2, EvenMax=11

Насоки и подсказки

Задачата обединява няколко предходни задачи: намиране на **минимум**, **максимум** и **сбор**, както и обработка на елементите от **четни** и **нечетни позиции**. Припомнете си ги.

В тази задача е по-добре да се работи с **дробни числа** (не цели). Сборът, минимумът и максимумът също са дробни числа.

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1364#10>.

Задача: еднакви двойки

Дадени са $2 * n$ числа. Първото и второто формират **двойка**, третото и четвъртото също и т.н. Всяка двойка има **стойност** – сума от съставящите я числа. Напишете програма, която проверява **дали всички двойки имат еднаква стойност**.

В случай, че стойността е еднаква, отпечатайте "Yes, value=..." + стойността, в противен случай отпечатайте **максималната разлика** между две последователни двойки в следния формат - "No, maxdiff=..." + максималната разлика.

Входът се състои от число n , следвано от $2 * n$ цели числа, всички по едно на ред.

Примерен вход и изход

Вход	Изход	Коментар	Вход	Изход	Коментар
3 1 2 0 3 4 -1	Yes, value=3	стойности = {3, 3, 3} еднакви стойности	2 1 2 2 2	No, maxdiff=1	стойности = {3, 4} разлики = {1} макс. разлика = 1

Вход	Изход	Коментар	Вход	Изход	Коментар
2 -1 2 0 -1	No, maxdiff=2	стойности = {1, -1} разлики = {2} макс. разлика = 2	1 5 5	Yes, value=10	стойности = {10} една стойност еднакви стойности

Насоки и подсказки

Прочитаме входните числа **по двойки**. За всяка двойка пресмятаме **сбора** ѝ. Докато четем входните двойки, за всяка двойка, без първата, трябва да пресметнем **разликата с предходната**. За целта е необходимо да пазим в отделна променлива **сбора** на предходната двойка. Накрая намираме **най-голямата разлика** между две двойки. Ако е 0, отпечатваме "Yes, value=" + стойността, в противен случай - "No, maxdiff=" + разликата.

Тестване в Judge системата

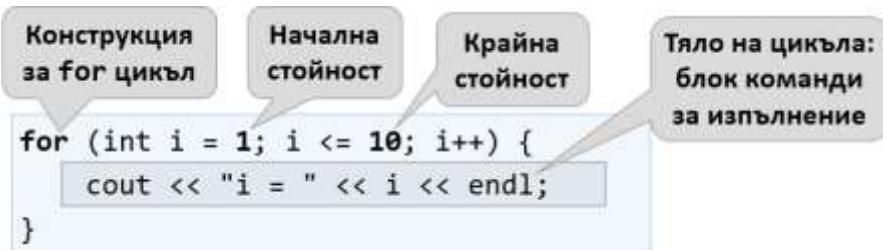
Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1364#11>.

Ако имате проблеми с задачите по-горе, **гледайте видеото** в началото на тази глава. Там са обяснени повечето от тях, на живо и стъпка по стъпка. Или питайте във **форума на СофтУни**: <https://softuni.bg/forum>.

Глава 5.2. Повторения (цикли) – изпитни задачи

В предходната глава научихме как да изпълним даден блок от команди **повече от веднъж**. Затова въведохме **for цикъл** и разглеждахме някои от основните му приложения. Целта на настоящата глава е да затвърдим знанията си, решавайки няколко по-сложни задачи с цикли, давани на приемни изпити. За някои от тях ще покажем примерни подробни решения, а за други ще оставим само напътствия. Преди да се захванем за работа е добре да си припомним конструкцията на цикъла **for**:



for циклите се състоят от:

- Инициализационен блок, в който се декларира променливата-брояч (**int i**) и се задава нейната начална стойност.
- Условие за повторение (**i <= 10**), изпълняващо се веднъж, преди всяка итерация на цикъла.
- Обновяване на брояча (**i++**) – този код се изпълнява след всяка итерация и указва с каква стъпка да се обновява променливата-брояч.
- Тяло на цикъла - съдържа произволен блок със сорс код.

Изпитни задачи

Да решим няколко задачи с цикли от изпити в СофтУни.

Задача: хистограма

Дадени са **n** цели числа в интервала [1 ... 1000]. От тях някакъв процент **p1** са под 200, процент **p2** са от 200 до 399, процент **p3** са от 400 до 599, процент **p4** са от 600 до 799 и останалите **p5** процента са от 800 нагоре. Да се напише програма, която изчислява и отпечатва процентите **p1, p2, p3, p4** и **p5**.

Пример: имаме **n = 20** числа: 53, 7, 56, 180, 450, 920, 12, 7, 150, 250, 680, 2, 600, 200, 800, 799, 199, 46, 128, 65. Получаваме следното разпределение и визуализация:

Група	Числа	Брой числа	Процент
< 200	53, 7, 56, 180, 12, 7, 150, 2, 199, 46, 128, 65	12	$p1 = 12 / 20 * 100 = 60.00\%$
200 ... 399	250, 200	2	$p2 = 2 / 20 * 100 = 10.00\%$
400 ... 599	450	1	$p3 = 1 / 20 * 100 = 5.00\%$
600 ... 799	680, 600, 799	3	$p4 = 3 / 20 * 100 = 15.00\%$
≥ 800	920, 800	2	$p5 = 2 / 20 * 100 = 10.00\%$

Входни данни

На първия ред от входа стои цялото число **n** ($1 \leq n \leq 1000$), което представлява броя редове с числа, които ще ни бъдат подадени. На следващите **n реда** стои по едно цяло число в интервала [1 ... 1000] – числата, върху които да бъде изчислена хистограмата.

Изходни данни

Да се отпечата на конзолата **хистограма от 5 реда**, всеки от които съдържа число между 0% и 100%, форматирано с точност две цифри след десетичния знак (например 25.00%, 66.67%, 57.14%).

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
3	66.67%	4	75.00%	7	14.29%
1	0.00%	53	0.00%	800	28.57%
2	0.00%	7	0.00%	801	14.29%
999	0.00%	56	0.00%	250	14.29%
	33.33%	999	25.00%	199	28.57%
				399	
				599	
				799	

Насоки и подсказки

Вход	Изход	Вход	Изход
9	33.33%	14	57.14%
367	33.33%	53	14.29%
99	11.11%	7	7.14%
200	11.11%	56	14.29%
799	11.11%	180	7.14%
999		450	
333		920	
555		12	
111		7	
9		150	
		250	
		680	
		2	
		600	
		200	

Програмата, която решава този проблем, можем да разделим мислено на три части:

- **Прочитане на входните данни** – в настоящата задача това включва прочитането на числото **n**, последвано от **n на брой цели числа**, всяко на отделен ред.
- **Обработка на входните данни** – в случая това означава разпределение на числата по групи и изчисляване на процентното разделение по групи.
- **Извеждане на краен резултат** – отпечатване на хистограмата на конзолата в посочения в условието формат.

Преди да продължим напред ще направим едно малко отклонение от настоящата тема, а именно ще споменем накратко, че в програмирането всяка променлива е от някакъв **тип данни**. В тази задача ще използваме числовите типове **int** за **цели числа** и **double** за **дробни**.

Сега ще преминем към имплементацията на всяка от горепосочените точки.

Прочитане на входните данни

Преди да преминем към самото прочитане на входните данни трябва да си декларираме **променливите**, в които ще ги съхраняваме. Това означава да им изберем подходящ тип данни и подходящи имена:

```
int n;
//Променливи, в които ще запазим
//процентното разделение на отделните групи
```

```

double p1Percentage = 0;
double p2Percentage = 0;
double p3Percentage = 0;
double p4Percentage = 0;
double p5Percentage = 0;

//Променливи, пазещи броя числа по групи
int cntP1 = 0;
int cntP2 = 0;
int cntP3 = 0;
int cntP4 = 0;
int cntP5 = 0;

```

В променливата **n** ще съхраняваме броя на числата, които ще четем от конзолата. Избираме **тип int**, защото в условието е упоменато, че **n е цяло число** в диапазона от 1 до 1000. За променливите, в които ще пазим процентите, избираме **тип double**, тъй като се очаква те **не винаги да са цели числа**. Допълнително си декларираме и променливите **cntP1**, **cntP2** и т.н., в които ще пазим броя на числата от съответната група, като за тях отново избираме **тип int**.

След като сме си декларирали нужните променливи, можем да пристъпим към прочитането на числото **n** от конзолата:

```
cin >> n;
```

Обработка на входните данни

За да прочетем и разпределим всяко число в съответната му група, ще си послужим с **for цикъл** от 0 до **n** (броя на числата). Всяка итерация на цикъла ще прочита и разпределя **едно единствено** число (**currentNumber**) в съответната му група. За да определим дали едно число принадлежи към дадена група, **правим** проверка в съответния ѝ диапазон. Ако това е така - увеличаваме броя на числата в тази група (**cntP1**, **cntP2** и т.н.) с 1.

```

for (int i = 0; i < n; i++) {
    int currentNumber;
    cin >> currentNumber;

    if (currentNumber < 200) {
        cntP1++;
    }
    else if (currentNumber >= 200 && currentNumber < 400) {
        cntP2++;
    }
}

```

```

else if (currentNumber >= 400 && currentNumber < 600) {
    cntP3++;
}
else if (currentNumber >= 600 && currentNumber < 800) {
    cntP4++;
}
else {
    cntP5++;
}
}

```

След като сме определили колко числа има във всяка група, можем да преминем към изчисляването на процентите, което е и главна цел на задачата. За това ще използваме следната формула:

$$(\text{процент на група}) = (\text{брой числа в група}) * 100 / (\text{брой на всички числа})$$

Тази формула в програмния код изглежда по следния начин:

```
p1Percentage = cntP1 * 100.0 / n;
```



Ако разделим на **100** (число от тип **int**) вместо на **100.0** (число от тип **double**), ще се извърши така нареченото **целочислено деление** и в променливата ще се запази само цялата част от делението, а това не е желания от нас резултат. Например: $5 / 2 = 2$, а $5 / 2.0 = 2.5$. Имайки това предвид, формулата за първата променлива ще изглежда така:

```
p1Percentage = cntP1 * 100.0 / n;
```

// Добавете формули и за останалите променливи

За да стане още по-ясно какво се случва, нека разгледаме следния пример:

Вход	Изход
3	66.67%
1	0.00%
2	0.00%
999	33.33%

В случая **n = 3**. За цикъла имаме:

- **i = 0** - прочитаме числото 1, което е по-малко от 200 и попада в първата

група (**p1**), увеличаваме брояча на групата (**cntP1**) с 1.

- **i = 1** – прочитаме числото 2, което отново попада в първата група (**p1**) и увеличаваме брояча ѝ (**cntP1**) отново с 1.
- **i = 2** – прочитаме числото 999, което попада в последната група (**p5**), защото е по-голямо от 800, и увеличаваме брояча на групата (**cntP5**) с 1.

След прочитането на числата в група **p1** имаме 2 числа, а в **p5** имаме 1 число. В другите групи **нямаме числа**. Като приложим гореспоменатата формула, изчисляваме процентите на всяка група. Ако във формулата умножим по **100**, вместо по **100.0** ще получим за група **p1** 66%, а за група **p5** – 33% (няма да има дробна част).

Извеждане на краен резултат

Остава само да отпечатаме получените резултати. В условието е казано, че процентите трябва да са **с точност две цифри след десетичната точка**. Това ще постигнем, използвайки **fixed** и **setprecision**. Като за да ползваме **setprecision**, трябва да включим и библиотеката **iomanip**:

```
cout << fixed << setprecision(2) << p1Percentage << "%" << endl;
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1365#0>.

Задача: умната Лили

Лили вече е на **N години**. За всеки свой **ръзден ден** тя получава подарък. За **нечетните** ръздени дни (1, 3, 5, ..., n) получава **играчки**, а за всеки **четен** (2, 4, 6, ..., n) получава **пари**. За **втория ръзден ден** получава **10.00 лв.**, като **сумата се увеличава с 10.00 лв.** за всеки **следващ четен ръзден ден** (2 -> 10, 4 -> 20, 6 -> 30 и т.н.). През годините Лили тайно е спестявала парите. **Братът** на Лили, в годините, които тя получава пари, взима по **1.00 лев** от тях. Лили **продала** **играчките**, получени през годините, **всяка за P лева** и добавила сумата към спестените пари. С парите искала да си **купи пералня за X лева**. Напишете програма, която да пресмята **колко пари е събрала** и **да дали ѝ стигат да купи пералня**.

Входни данни

От конзолата се прочитат **3 числа**, всяко на отделен ред:

- **Възрастта** на Лили – **цяло число** в интервала [1 ... 77].

- Цената на пералнята – число в интервала [1.00 ... 10 000.00].
- Единична цена на играчка – цяло число в интервала [0 ... 40].

Изходни данни

Да се отпечата на конзолата един ред:

- Ако парите на Лили са достатъчни:
 - "Yes! {N}" – където **N** е остатъка пари след покупката.
- Ако парите не са достатъчни:
 - "No! {M}" – където **M** е сумата, която не достига.
- Числата **N** и **M** трябва да са **форматирани до втория знак след десетичната точка.

Примерен вход и изход

Вход	Изход	Коментари
10 170.00 6	Yes! 5.00	Първи рожден ден получава играчка; 2ри -> 10 лв.; Зти -> играчка; 4ти -> $10 + 10 = 20$ лв.; 5ти -> играчка; 6ти -> $20 + 10 = 30$ лв.; 7ми -> играчка; 8ми -> $30 + 10 = 40$ лв.; 9ти -> играчка; 10ти -> $40 + 10 = 50$ лв. Спестила е -> $10 + 20 + 30 + 40 + 50 = 150$ лв.. Продала е 5 играчки по 6 лв. = 30 лв.. Брат ѝ взел 5 пъти по 1 лев = 5 лв. Остават -> $150 + 30 - 5 = 175$ лв. $175 \geq 170$ (цената на пералнята) успяла е да я купи и са ѝ останали $175 - 170 = 5$ лв.
21 1570.98 3	No! 997.98	Спестила е 550 лв.. Продала е 11 играчки по 3 лв. = 33 лв. Брат ѝ взимал 10 години по 1 лев = 10 лв. Останали $550 + 33 - 10 = 573$ лв. $573 < 1570.98$ – не е успяла да купи пералня. Не ѝ достигат $1570.98 - 573 = 997.98$ лв.

Насоки и подсказки

Решението на тази задача, подобно на предходната, също можем да разделим мислено на три части – **прочитане** на входните данни, **обработката** им и **извеждане** на резултат.

Отново започваме с избора на подходящи **типове данни** и имена на променливите. За годините на Лили (**age**) и единичната цена на играчката (**presentPrice**) по условие е дадено, че ще са **цели числа**. Затова ще използваме типа **int**. За цената на пералнята (**priceOfWashingMachine**) знаем, че е **дробно**

число и избираме **double**. В кода по-горе декларираме и инициализираме (присвояваме стойност) променливите:

```
int age;
double priceOfWashingMachine;
int presentPrice;
cin >> age;
```

За да решим задачата, ще се нуждаем от няколко помощни променливи – за броя на играчките (**numberOfToys**), за спестените пари(**savedMoney**) и за парите, получени на всеки рожден ден (**moneyForBirthday**). Като присвояваме на **moneyForBirthday** първоначална стойност 10, тъй като по условие е дадено, че първата сума, която Лили получава, е 10 лв:

```
int numberOfToys = 0;
int savedMoney = 0;
int moneyForBirthday = 10;
```

С **for** цикъл преминаваме през всеки рожден ден на Лили. Когато броят в нашия цикъл (което съответства на годините на Лили) е **четно число**, това означава, че Лили е **получила пари** и съответно прибавяме тези пари към общите ѝ спестявания. Едновременно с това **изваждаме по 1 лев** - парите, които брат ѝ взема. След това **увеличаваме** стойността на променливата **moneyForBirthday**, т.е. увеличаваме с 10 сумата, която тя ще получи на следващия си рожден ден. Обратно, когато броят (годините на Лили) е **нечетно число**, увеличаваме броя на **играчките**. Проверката за четност осъществяваме чрез **деление с остатък (%)** на 2 – когато остатъкът е 0, числото е четно, а при остатък 1 – нечетно:

```
for (int currentYear = 1; currentYear <= age; currentYear++) {
    if (currentYear % 2 == 0) {
        savedMoney += (moneyForBirthday - 1);
        moneyForBirthday += 10;
    }
    else {
        numberOfToys++;
    }
}
```

Към спестяванията на Лили прибавяме и парите от продадените играчки:

```
savedMoney += numberOfToys * presentPrice;
```

Накрая остава да отпечатаме получените резултати, като се съобразим с форматирането, указано в условието, т.е. сумата трябва да е **закръглена до две цифри след десетичния знак**:

```
if (savedMoney >= priceOfWashingMachine) {
    cout << fixed << setprecision(2) << "Yes! "
        << (savedMoney - priceOfWashingMachine) << endl;
}
else {
    cout << fixed << setprecision(2) << "No! "
        << (priceOfWashingMachine - savedMoney) << endl;
}
```

Проверяваме дали **събрани пари** от Лили стигат за една пералня. Ако те са повече или равни на цената на пералнята, то проверката **savedMoney >= priceOfWashingMachine** ще върне **true** и ще се отпечата "Yes! ...", а ако са по-малко – резултатът ще е **false** и ще се отпечата "No! ...".

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1365#1>.

Задача: завръщане в миналото

Иванчо е на **18 години** и получава наследство, което се състои от **X суми пари** и машина на времето. Той решава **да се върне до 1800 година**, но не знае дали парите ще са достатъчни, за да живее без да работи. Напишете програма, която пресмята дали Иванчо ще има достатъчно пари, за да не се налага да работи **до дадена година включително**. Като приемем, че **за всяка четна** (1800, 1802 и т.н.) година ще харчи 12 000 долара. За **всяка нечетна** (1801, 1803 и т.н.) ще харчи **12 000 + 50 * [годините, които е навършил през дадената година]**.

Входни данни

Входът се чете от конзолата и **съдържа точно 2 реда**:

- **Наследените пари** – реално число в интервала [1.00 ... 1 000 000.00].
- **Годината, до която трябва да живее (включително)** – цяло число в интервала [1801 ... 1900].

Изходни данни

Да се отпечатва на конзолата **1 ред**. Сумата трябва да е **форматирана до два знака след десетичния знак**:

- Ако парите са достатъчно:

- "Yes! He will live a carefree life and will have {N} dollars left." – където N е сумата, която ще му останат.
- Ако парите НЕ са достатъчно:
 - "He will need {M} dollars to survive." – където M е сумата, която НЕ достига.

Примерен вход и изход

Вход	Изход	Обяснения
50000 1802	Yes! He will live a carefree life and will have 13050.00 dollars left.	<p>1800 → четна → Харчи 12000 долара → Остават $50000 - 12000 = 38000$</p> <p>1801 → нечетна → Харчи $12000 + 19 * 50 = 12950$ долара → Остават $38000 - 12950 = 25050$</p> <p>1802 → четна → Харчи 12000 долара → Остават $25050 - 12000 = 13050$</p>
100000.15 1808	He will need 12399.85 dollars to survive.	<p>1800 → четна → Остават $100000.15 - 12000 = 88000.15$</p> <p>1801 → нечетна → Остават $88000.15 - 12950 = 75050.15$</p> <p>...</p> <p>1808 → четна → $-399.85 - 12000 = -12399.85$ 12399.85 не достигат</p>

Насоки и подсказки

Започваме с **деклариране и инициализиране** на нужните променливи. Казано е, че годините на Иванчо са 18, ето защо при декларацията на променливата **years** ѝ задаваме начална стойност **18**. Другите променливи прочитаме от конзолата:

```
double heritage;
int yearToLive;
int years = 18;
```

С помощта на **for** цикъл ще обходим всички години. Започваме от 1800 – годината, в която Иванчо се връща, и стигаме до годината, до която той трябва да живее. В цикъла проверяваме дали текущата година е четна или нечетна. Проверката за четност осъществяваме чрез **деление с остатък (%)** на 2. Ако годината е **четна**, изваждаме от наследството (**heritage**) 12000, а ако е **нечетна**, изваждаме от наследството (**heritage**) $12000 + 50 * (\text{годините на Иванчо})$. Като на всяка итерация на цикъла увеличаваме годините, на които е Иванчо.

```
for (int currentYear = 1800; currentYear <= yearToLive; currentYear++) {
    if (currentYear % 2 == 0) {
        heritage -= 12000;
    }
    else {
        heritage -= (12000 + 50 * years);
    }
    years++;
}
```

Накрая остава да отпечатаме резултатите, като за целта правим проверка дали наследството (**heritage**) му е било достатъчно да живее без да работи или не. Ако наследството (**heritage**) е положително число, отпечатваме: "**Yes! He will live a carefree life and will have {N} dollars left.**", а ако е отрицателно число: "**He will need {M} dollars to survive.**". Не забравяме да форматираме сумата до два знака след десетичната точка.

Hint: Обмислете използването на функцията **abs(...)** от библиотеката **cmath** при отпечатване на изхода, когато наследството е недостатъчно.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1365#2>.

Задача: болница

За даден период от време, всеки ден в болницата пристигат пациенти за преглед. Тя разполага **първоначално** със **7 лекари**. Всеки лекар може да преглежда **само по един пациент на ден**, но понякога има недостиг на лекари, затова **останалите пациенти се изпращат в други болници**. Всеки **трети ден** болницата прави изчисления и ако броят на непрегледаните пациенти е **по-голям от броя на прегледаните, се назначава още един лекар**. Като назначаването става преди да започне приемът на пациенти за деня.

Напишете програма, която изчислява **за дадения период** броя на прегледаните и

непрегледаните пациенти.

Входни данни

Входът се чете от конзолата и съдържа:

- На първия ред – **периода**, за който трябва да направите изчисления. Цяло число в интервала [1 ... 1000].
- На следващите редове (равни на броя на дните) – **броя пациенти**, които пристигат за преглед за **текущия ден**. Цяло число в интервала [0 ... 10 000].

Изходни данни

Да се отпечатат на конзолата **2 реда**:

- На първия ред: "Treated patients: {брой прегледани пациенти}."
- На втория ред: "Untreated patients: {брой непрегледани пациенти}."

Примерен вход и изход

Вход	Изход	Обяснения
4 7 27 9 1	Treated patients: 23. Untreated patients: 21.	1 ден: 7 прегледани и 0 непрегледани пациенти за деня 2 ден: 7 прегледани и 20 непрегледани пациенти за деня 3 ден: До момента прегледаните пациенти са общо 14, а непрегледаните – 20 –> Назначава се нов лекар –> 8 прегледани и 1 непрегледан пациент за деня 4 ден: 1 прегледан и 0 непрегледани пациенти за деня Общо: 23 прегледани и 21 непрегледани.

Вход	Изход	Вход	Изход
6 25 25 25 25 2	Treated patients: 40. Untreated patients: 87.	3 7 7 7	Treated patients: 21. Untreated patients: 0.

Насоки и подсказки

Отново започваме, като **декларираме и инициализираме** нужните променливи. Периодът, за който трябва да направим изчисленията, прочитаме от конзолата и запазваме в променливата **period**. Ще се нуждаем и от няколко помощни променливи: броя на излекуваните пациенти (**treatedPatients**), броя на неизлекуваните пациенти (**untreatedPatients**) и броя на докторите (**countOfDoctors**), който първоначално е 7:

```
int period;
int treatedPatients = 0;
int untreatedPatients = 0;
int countOfDoctors = 7;
```

С помощта на **for цикъл** обхождаме всички дни в дадения период (**period**). За всеки ден прочитаме от конзолата броя на пациентите (**currentPatients**). Увеличаването на докторите по условие може да стане **всеки трети ден**, но само ако броят на непрегледаните пациенти е **по-голям** от броя на прегледаните. За тази цел проверяваме дали денят е трети – чрез аритметичния оператор за деление с остатък (%): **day % 3 == 0**.

Например:

- Ако денят е **трети**, остатъкът от делението на 3 ще бъде 0 (**3 % 3 = 0**) и проверката **day % 3 == 0** ще върне **true**.
- Ако денят е **втори**, остатъкът от делението на 3 ще бъде 2 (**2 % 3 = 2**) и проверката ще върне **false**.
- Ако денят е **четвърти**, остатъкът от делението ще бъде 1 (**4 % 3 = 1**) и проверката отново ще върне **false**.

Ако проверката **day % 3 == 0** върне **true**, ще се провери дали и броят на неизлекуваните пациенти е **по-голям** от този на излекуваните: **untreatedPatients > treatedPatients**. Ако резултатът отново е **true**, то тогава ще се увеличи броят на лекарите (**countOfDoctors**).

След това проверяваме броя на пациентите за деня (**currentPatients**) дали е **по-голям** от броя на докторите (**countOfDoctors**). Ако броят на пациентите е **по-голям**:

- Увеличаваме стойността на променливата **treatedPatients** с броя на докторите (**countOfDoctors**).
- Увеличаваме стойността на променливата **untreatedPatients** с броя на останалите пациенти, който изчисляваме, като от всички пациенти извадим броя на докторите (**currentPatients - countOfDoctors**).

Ако броят на пациентите не е по-голям, увеличаваме само променливата **treatedPatients** с броя на пациентите за деня (**currentPatients**):

```
for (int day = 1; day <= period; day++) {
    int currentPatients;
    cin >> currentPatients;

    if ((day % 3 == 0) && (untreatedPatients > treatedPatients)) {
        countOfDoctors++;
    }

    if (currentPatients > countOfDoctors) {
        treatedPatients += countOfDoctors;
        untreatedPatients += currentPatients - countOfDoctors;
    }
    else {
        treatedPatients += currentPatients;
    }
}
```

Накрая трябва само да отпечатаме броя на излекуваните и броя на неизлекуваните пациенти.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1365#3>.

Задача: деление без остатък

Дадени са **n** цели числа в интервала [1 ... 1000]. От тях някакъв процент **p1** се делят без остатък на 2, процент **p2** се делят без остатък на 3, процент **p3** се делят без остатък на 4. Да се напише програма, която изчислява и отпечатва процентите **p1**, **p2** и **p3**.

Пример: имаме **n = 10** числа: 680, 2, 600, 200, 800, 799, 199, 46, 128, 65. Получаваме следното разпределение и визуализация:

Деление без остатък на:	Числа	Брой	Процент
2	680, 2, 600, 200, 800, 46, 128	7	$p1 = (7 / 10) * 100 = 70.00\%$
3	600	1	$p2 = (1 / 10) * 100 = 10.00\%$

Деление без остатък на:	Числа	Брой	Процент
4	680, 600, 200, 800, 128	5	$p3 = (5 / 10) * 100 = 50.00\%$

Входни данни

На първия ред от входа стои цялото число n ($1 \leq n \leq 1000$) – брой числа. На следващите n реда стои по едно цяло число в интервала $[1 \dots 1000]$ – числата, които да бъдат проверени на колко се делят.

Изходни данни

Да се отпечатат на конзолата **3 реда**, всеки от които съдържа процент между 0% и 100%, с точност две цифри след десетичния знак, например 25.00%, 66.67%, 57.14%.

- На **първия ред** – процентът на числата, които **се делят на 2**.
- На **втория ред** – процентът на числата, които **се делят на 3**.
- На **третия ред** – процентът на числата, които **се делят на 4**.

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
10	70.00%	3	33.33%	1	100.00
680	10.00%	3	100.00	12	%
2	50.00%	6	%		100.00
600		9	0.00%		%
200					100.00
800					%
799					
199					
46					
128					
65					

Насоки и подсказки

За тази и следващата задача ще трябва сами да напишете програмния код, следвайки дадените напътствия.

Програмата, която решава текущия проблем, е аналогична на тази от задача **Хистограма**, която разгледахме по-горе. Затова можем да започнем с декларацията на нужните ни променливи. Примерни имена на променливи може

да са: **n** – брой на числата (който трябва да прочетем от конзолата) и **divisibleBy2**, **divisibleBy3**, **divisibleBy4** – помощни променливи, пазещи броя на числата от съответната група.

За да прочетем и разпределим всяко число в съответната му група, ще трябва да завъртим **for цикъл** от **0** до **n** (броя на числата). Всяка итерация на цикъла трябва да прочита и разпределя **едно единствено число**. Различното тук е, че **едно число може да попадне в няколко групи едновременно**, затова трябва да направим **три отделни if проверки за всяко число** – съответно дали се дели на 2, 3 и 4 и да увеличим стойността на променливата, която пази броя на числата в съответната група. Конструкцията **if-else** в този случай няма да ни свърши работа, защото след като намери съвпадение се прекъсва по-нататъшното проверяване на условията).

Накрая трябва да отпечатате получените резултати, като спазвате посочения формат в условието.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1365#4>.

Задача: логистика

Отговаряте за логистиката на различни товари. В зависимост от теглото на всеки товар е нужно различно превозно средство и струва различна цена на тон:

- До 3 тона – микробус (200 лева на тон).
- От над 3 и до 11 тона – камион (175 лева на тон).
- Над 11 тона – влак (120 лева на тон).

Изчислете средната цена на тон превозен товар, както и колко процента от товара се превозват с **всяко превозно средство**.

Входни данни

От конзолата се четат **поредица от числа**, всяко на отделен ред:

- Първи ред: брой на товарите за превоз – цяло число в интервала [1 ... 1000].
- На всеки следващ ред се подава тонажът на поредния товар – цяло число в интервала [1 ... 1000].

Изходни данни

Да се отпечатат на конзолата **4 реда**, както следва:

- Ред #1 – средната цена на тон превозен товар (закръглена до втория знак след десетичната точка).

- Ред #2 – процентът товар, превозван с **микробус** (между 0.00% и 100.00%, закръглен до втория знак след десетичната точка).
- Ред #3 – процентът товар, превозвани с **камион** (между 0.00% и 100.00%).
- Ред #4 – процентът товар, превозвани с **влак** (между 0.00% и 100.00%).

Примерен вход и изход

Вход	Изход	Обяснения
4	143.80	С микробус се превозват два от товарите 1 + 3 , общо 4 тона.
1	16.00%	С камион се превозва един от товарите: 5 тона.
5	20.00%	С влак се превозва един от товарите: 16 тона.
16	64.00%	Сумата от всички товари е: $1 + 5 + 16 + 3 = 25$ тона.
3		Процент товар с микробус : $4/25 * 100 = 16.00\%$ Процент товар с камион : $5/25 * 100 = 20.00\%$ Процент товар с влак : $16/25 * 100 = 64.00\%$ Средна цена на тон превозен товар: $(4 * 200 + 5 * 175 + 16 * 120) / 25 = 143.80$

Вход	Изход	Вход	Изход
5	149.38	4	120.35
2	7.50%	53	0.00%
10	42.50%	7	0.63%
20	50.00%	56	99.37%
1		999	
7			

Насоки и подсказки

Първо ще прочетем теглото на всеки товар и ще сумираме колко тона се превозват съответно с **микробус**, **камион** и **влак** и ще изчислим и общите тонове превозени товари. Ще пресметнем **цените за всеки вид транспорт** според превозените тонове и **общата цена**. Накрая ще пресметнем и отпечатаме **общата средна цена на тон** и каква част от товара е превозена с всеки вид транспорт процентно.

Декларираме си нужните променливи, например: **countOfLoads** – броя на товарите за превоз (прочитаме ги от конзолата), **sumOfTons** – сумата от тонажа на всички товари, **microbusTons**, **truckTons**, **trainTons** – променливи, пазещи сумата от тонажа на товарите, превозвани съответно с микробус, камион и влак.

Ще ни тряба **for** цикъл от **0** до **countOfLoads**, за да обходим всички товари. За всеки товар прочитаме теглото му (в тонове) от конзолата и го запазваме в променлива, например **tons**. Прибавяме към сумата от тонажа на всички товари (**sumOfTons**) теглото на текущия товар (**tons**). След като сме прочели теглото на текущия товар, трябва да определим кое превозно средство ще се ползва за него (микробус, камион или влак). За целта ще ни трябват **if-else** проверки:

- Ако стойността на променливата **tons** е **по-малка от 3**, увеличаваме стойността на променливата **microbusTons** със стойността на **tons**:
 - **microbusTons += tons;**
- Иначе, ако стойността **tons** е **до 11**, увеличаваме **truckTons** с **tons**.
- Ако **tons** е **повече от 11**, увеличаваме **trainTons** с **tons**.

Преди да отпечатаме изхода трябва да изчислим процента на тоновете, превозвани с всяко превозно средство и средната цена на тон. За средната цена на тон ще си декларираме още една помощна променлива **totalPrice**, в която ще сумираме общата цена на всички превозвани товари (с микробус, камион и влак). Средната цена ще получим, разделяйки **totalPrice** на **sumOfTons**. Остава **сами да изчислите** процента на тоновете, превозвани с всяко превозно средство, и да отпечатате резултатите, спазвайки формата в условието.

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1365#5>.

Глава 6.1. Вложени цикли

В настоящата глава ще разгледаме **вложените цикли** и как да използваме **for** цикли за **чертане** на различни **фигурки на конзолата**, които се състоят от символи и знаци, разположени в редове и колони на конзолата. Ще използваме **единични** и **вложени цикли** (цикли един в друг), **изчисления** и **проверки**, за да отпечатваме на конзолата различни фигури по зададени размери.

Видео

Гледайте видео урок по учебния материал от настоящата глава от книгата: https://www.youtube.com/watch?v=K_ibDVv4Kd0.

Пример: правоъгълник от 10 x 10 звездички

Да се начертате в конзолата правоъгълник от **10 x 10** звездички.

Вход	Изход
(няма)	***** ***** ***** ***** ***** ***** ***** ***** ***** *****

Насоки и подсказки

```
for (int i = 0; i < 10; i++) {  
    cout << string(10, '*') << endl;  
}
```

Как работи примерът? Създава се **променлива от тип string** Инициализира се **цикъл с променлива **i = 0****, която се увеличава на всяка итерация на цикъла, докато е **по-малка от 10**. Така кодът в тялото на цикъла се изпълнява **10 пъти**. В тялото на цикъла се добавя по една звездичка към низа. Това означава, че **след края** на цикъла низът ще съдържа **10 звездички**. Инициализира се **втори цикъл**, който **също** се изпълнява **10 пъти**. В тялото на втория цикъл се печата на нов ред в конзолата **низа от 10 звездички**, създаден преди това.

Как работи примерът? Инициализира се **цикъл с променлива **i****. Началната стойност по подразбиране на променливата е **i = 0**. С всяка итерация на цикъла променливата се увеличава с **1**, докато е **по-малка от 10**. Така кодът в тялото на цикъла се изпълнява **10 пъти** - от **0-ия** до **9-ия** път включително. В тялото на цикъла

се печата на нов ред в конзолата `string(10, '*')`, което създава низ (стринг) от 10 звездички.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1366#0>.

Пример: правоъгълник от N x N звездички

Да се напише програма, която въвежда цяло положително число **n** и печата на конзолата правоъгълник от **N x N** звездички.

Вход	Изход	Вход	Изход	Вход	Изход
2	** **	3	*** *** ***	4	**** **** **** ****

Насоки и подсказки

Задачата е аналогична с предходната:

```
int n;
cin >> n;

for (int i = 0; i < n; i++) {
    cout << string(n, '*') << endl;
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1366#1>.

Вложени цикли

Вложените цикли представляват конструкция, при която **в тялото на един цикъл** (външен) **се изпълнява друг цикъл** (вътрешен). За всяко завъртане на външния цикъл, вътрешният се извърта **целият**. Това се случва по следния начин:

- При стартиране на изпълнение на вложени цикли първо **стартира външният цикъл**: извършва се **инициализация** на неговата управляваща променлива и след проверка за край на цикъла, се изпълнява кодът в тялото му.
- След това се **изпълнява вътрешният цикъл**. Извършва се инициализация на началната стойност на управляващата му променлива, прави се проверка за край на цикъла и се изпълнява кодът в тялото му.

- При достигане на зададената стойност за **край на вътрешния цикъл**, програмата се връща една стъпка нагоре и се продължава започналото изпълнение на външния цикъл. Променя се с една стъпка управляващата променлива за външния цикъл, проверява се дали условието за край е удовлетворено и **започва ново изпълнение на вложените (вътрешни) цикъл**.
- Това се повтаря, докато променливата на външния цикъл достигне условието за **край на цикъла**.

Ето и един **пример**, с който нагледно да илюстрираме вложените цикли. Целта е да се отпечата отново правоъгълник от **N x N звездички**, като за всеки ред се извърта цикъл от **1** до **N**, а за всяка колона се извърта вложен цикъл от **1** до **N**:

```
int n;
cin >> n;

for (int r = 0; r < n; r++) {
    for (int c = 0; c < n; c++) {
        cout << "*";
    }
    cout << endl;
}
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/NestedLoopExample>.

Да разгледаме примера по-горе. След инициализацията на **първия (външен) цикъл**, започва да се изпълнява неговото **тяло**, което съдържа **втория (вложен) цикъл**. Той сам по себе си печата на един ред **n** на брой звездички. След като **вътрешният цикъл приключи** изпълнението си при първата итерация на външния, то след това **външният ще продължи**, т.е. ще отпечата един празен ред на конзолата. **След това ще се извърши обновяване** на променливата на **първия цикъл** и отново ще бъде изпълнен целият **втори (вложен) цикъл**. Вътрешният цикъл ще се изпълни толкова пъти, колкото се изпълнява тялото на външния цикъл, в случая **n** пъти.

Пример: квадрат от звездички

Да се начертате на конзолата квадрат от **N x N** звездички:

Вход	Изход	Вход	Изход	Вход	Изход
2	* *	3	* * * * * * * * *	4	* * * * * * * * * * * * * * * *

Насоки и подсказки

Задачата е аналогична на предходната. Разликата тук е, че трябва да обмислим как да печатаме интервал след звездичките по такъв начин, че да няма излишни интервали в началото или края:

```
int n;
cin >> n;

for (int r = 0; r < n; r++) {
    cout << "*";
    for (int c = 0; c < n - 1; c++) {
        cout << " *";
    }
    cout << endl;
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1366#2>.

Пример: триъгълник от долари

Да се напише програма, която въвежда число **n** и печата **триъгълник от долари**.

Вход	Изход	Вход	Изход
3	\$ \$ \$ \$ \$ \$	4	\$ \$ \$ \$ \$ \$ \$ \$ \$ \$

Насоки и подсказки

```
int n;
cin >> n;

for (int r = 0; r < n; r++) {
    cout << "$";
    for (int c = 0; c < r; c++) {
        cout << " $" ;
    }
    cout << endl;
}
```

Задачата е **сходна** с тези за рисуване на **правоъгълник** и **квадрат**. Отново ще използваме **вложени цикли**, но тук има **ловка**. Разликата е в това, че **броя на колонките**, които трябва да разпечатаме, зависят от **реда**, на който се намираме, а не от входното число **n**. От примерните входни и изходни данни забелязваме, че **броят на долларите зависи** от това на кой **ред** се намираме към момента на печатането, т.е. 1 доллар означава първи ред, 3 долара означават трети ред и т.н. Нека разгледаме долния пример по-подробно. Виждаме, че **променливата на вложения цикъл е обвързана с променливата на външния**. По този начин нашата програма печата желания триъгълник:

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1366#3>.

Пример: квадратна рамка

Да се напише програма, която въвежда цяло положително число **n** и чертае на конзолата **квадратна рамка** с размер **N x N**.

Вход	Изход	Вход	Изход	Вход	Изход
4	+ - - + - - - - + - - +	5	+ - - - + - - - - - - - - - + - - - +	6	+ - - - - + - - - - - - - - - - - - - - - - + - - - - +

Насоки и подсказки

Можем да решим задачата по следния начин:

- Четем от конзолата числото **n**.
- Отпечатваме **горната част**: първо знак **+**, после **n - 2** пъти **-** и накрая знак **+**.
- Отпечатваме **средната част**: печатаме **n - 2** реда като първо печатаме знак **|**, после **n - 2** пъти **-** и накрая отново знак **|**. Това можем да го постигнем с **вложени цикли**.
- Отпечатваме **долната част**: първо **+**, после **n - 2** пъти **-** и накрая **+**.

Ето и примерна имплементация на описаната идея, с вложени цикли:

```
int n;  
cin >> n;
```

```

// Print the top row: + - - - +
cout << "+";
for (int i = 0; i < n - 2; i++) {
    cout << " -";
}
cout << " +" << endl;

// Print the mid rows: | - - - |
for (int r = 0; r < n - 2; r++) {
    cout << "|";
    for (int c = 0; c < n - 2; c++) {
        cout << " -";
    }
    cout << " |" << endl;
}

// Print the bottom row: + - - - +
cout << "+";
for (int i = 0; i < n - 2; i++) {
    cout << " -";
}
cout << " +" << endl;

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1366#4>.

Пример: ромбче от звездички

Да се напише програма, която въвежда цяло положително число **n** и печата ромбче от звездички с размер **N**.

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
1	*	2	* * *	3	* * * * *	4	* * * * *

Насоки и подсказки

За решението на тази задача е нужно да **разделим** мислено ромба на **две части** - **горна**, която включва **и** средния ред, и **долна**. За **разпечатването** на всяка една част ще използваме **два** отделни цикъла, като оставяме на читателя сам да намери зависимостта между **n** и променливите на циклите. За първия цикъл може да използваме следните насоки:

- Отпечатваме **n - r - 1** интервала.
- Отпечатваме *****.
- Отпечатваме **r** пъти *****.

Втората (долна) част ще отпечатаме по **аналогичен** начин, което оставяме на читателя да се опита да направи сам.

```
int n;
cin >> n;

for (int r = 0; ..... ; r++) {
    for (int c = 0; ..... ; c++) {
        cout << " ";
    }

    cout << "*";

    for (int c = 0; ..... ; c++) {
        cout << " *";
    }

    cout << endl;
}

// TODO: print the downside of the rhombus
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1366#5>.

Пример: коледна елха

Да се напише програма, която въвежда число **n** ($1 \leq n \leq 100$) и печата коледна елха с височина **N + 1**.

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
1	* *	2	* * ** **	3	* * ** ** *** ***	4	* * ** ** *** *** **** ****

Насоки и подсказки

От примерите виждаме, че елхата може да бъде разделена на три логически части. Първата част включва звездичките и празните места преди и след тях, средната част е от |, а последната част - отново звездички, като този път празни места има само **преди** тях:

```
for (int r = 0; r <= n; r++) {
    string stars, spaces;
    for (int i = 0; i < r; i++) {
        stars += '*';
    }

    for (int i = 0; i < n - r; i++) {
        spaces += ' ';
    }

    cout << spaces << stars;
    cout << " | ";
    cout << stars << spaces << endl;
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1366#6>.

Чертане на по-сложни фигури

Да разгледаме как можем да чертаем на конзолата фигури с по-сложна логика на конструиране, за които трябва повече да помислим, преди да почнем да пишем.

Пример: слънчеви очила

Да се напише програма, която въвежда цяло число **n** ($3 \leq n \leq 100$) и печата слънчеви очила с размер **5*N x N** като в примерите:

Вход	Изход	Вход	Изход
3	***** ****** *///** *///** ***** ******	4	***** ****** */////* */////* */////* */////* ***** ******
5	***** ****** */////////* */////////* */////////* */////////* */////////* */////////* ***** ******		

Насоки и подсказки

От примерите виждаме, че очилата могат да се разделят на **три части** – горна, средна и долна. По-долу е част от кода, с който задачата може да се реши.

При рисуването на горния и долнния ред трябва да се отпечатат **2 * n** звездички, **n** интервала и **2 * n** звездички:

```
// Print the top part
for (int i = 0; i < 2 * n; i++) {
    cout << "*";
}

for (int i = 0; i < n; i++) {
    cout << " ";
}

for (int i = 0; i < 2 * n; i++) {
    cout << "*";
}
cout << endl;

// TODO: Print the middle part

// Print the bottom part
for (int i = 0; i < 2 * n; i++) {
    cout << "*";
```

```

}

for (int i = 0; i < n; i++) {
    cout << " ";
}

```

```

for (int i = 0; i < 2 * n; i++) {
    cout << "*";
}
cout << endl;

```

При печатането на средната част трябва да проверим дали редът е $(n - 1) / 2 - 1$, тъй като от примерите е видно, че на този ред трябва да печатаме вертикални чертички вместо интервали:

```

// Print the middle part
for (int i = 0; i < n - 2; i++) {
    // TODO: Print *////////*
}

for (int j = 0; j < n; j++) {
    cout << ((i == (n - 1) / 2 - 1) ? '|' : ' ');
}

// TODO: Print *////////*
cout << endl;
}

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1366#7>.

Пример: къщичка

Да се напише програма, която въвежда число n ($2 \leq n \leq 100$) и печата **къщичка** с размери $N \times N$, точно като в примерите.

Насоки и подсказки

Разбираме от условието на задачата, че къщата е с размер $n \times n$. Това, което виждаме от примерните вход и изход, е, че:

- Къщичката е разделена на 2 части: **покрив** и **основа**.
- Когато n е четно число, върхът на къщичката е тъп.

- Когато **n** е нечетно число, **покривът** е с остьр връх и с един ред по-голям от **основата**.

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
2	** 	3	-*- *** *	4	- ** - ***** ** **	5	--*-- -***- ***** *** ***



Покрив

- Съставен е от **звезди** и **тирета**.
- В най-високата си част има една или две звезди, спрямо това дали **n** е четно или нечетно, както и тиретата.
- В най-ниската си част има много звезди и малко или никакви тирета.
- С всеки един ред по-надолу **звездите** се увеличават с 2, а **тиретата** намаляват също с 2.

Основа

- Дълга е **n** на брой реда.
- Съставена е от **звезди** и **вертикални черти**.
- Редовете са съставени от 2 **вертикални черти** - по една в началото и в края на реда, както и **звезди** между чертите с дължина на низа **n - 2**.

Решение

Прочитаме входното число **n** от конзолата и записваме стойността му в променлива:

```
int n;
cin >> n;
```



Много е важно да проверяваме дали са валидни входните данни! В тези задачи не е проблем директно да обръщаме прочетения вход от конзолата в тип **int**, защото изрично е казано, че ще получаваме валидни целочислени числа. Ако обаче правим по-серииозни приложения, е добра практика да проверяваме данните. Какво ще стане, ако вместо число потребителят въведе например буквата "A"?

За да начертаем покрива, записваме колко ще е началният брой звезди в променлива **stars**:

- Ако **n** е нечетно, ще е 1 брой.
- Ако **n** е четно, ще са 2 броя.

```
int stars = 1;
if (n % 2 == 0) {
    stars++;
}
```

Изчисляваме дължината на покрива. Тя е равна на половината от **n**. Резултата записваме в променливата **roofLength**:

```
int roofLength = ceil(n / 2.0);
```

Забележка: За да използваме функцията **ceil()**, която закръгля към по-голямото цяло число, без значение от дробната част, е необходимо да въведем библиотеката **math.h**.

Важно е да съобразим, че когато **n** е нечетно число, дължината на покрива е по-голяма с един ред от тази на **основата**. В езика **C++**, когато два целочислени типа се делят и има остатък, то резултатът ще е число без остатъка.

Пример:

```
int result = 3 / 2; // Result is 1
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/DivisionExample>.

Ако искаме да закръглим нагоре, трябва да използваме функцията **ceil(...)**:

```
#include <iostream>
#include <math.h>

using namespace std;

int main() {
    int result1 = ceil(3 / 2.0); // резултат 2
    int result2 = ceil(7 / 2.7); // резултат 3
```

```
}
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/ExmapleWithCeil>.

В този пример делението не е от 2 целочислени числа. Когато цяло число е записано със суфикс '**.0**', това показва, че даденото число е от тип число с плаваща запетая. Резултатът от **3 / 2.0** е **1.5**, а функцията **ceil(...)** закръгля резултата от делението нагоре. В нашият случай **1.5** ще стане **2.0**, което се превръща в целочисленото **2**.

След като сме изчислили дължината на покрива, завъртаме цикъл от 0 до **roofLength**. На всяка итерация ще:

- Изчисляваме броя **тирета**, които трябва да изрисуваме. Броят ще е равен на **(n - stars) / 2**. Записваме го в променлива **padding**:

```
int padding = (n - stars) / 2;
```

- Отпечатваме на конзолата: "тирета" (**padding** на брой пъти) + "звезди" (**stars** пъти) + "тирета" (**padding** пъти):

```
for (int i = 0; i < padding; i++) {
    cout << "-";
}
```

```
for (int i = 0; i < stars; i++) {
    cout << "*";
}
```

```
for (int i = 0; i < padding; i++) {
    cout << "-";
}
```

- Преди да завърши въртенето на цикъла увеличаваме **stars** (броя на звездите) с 2:

```
stars += 2;
```

След като сме приключили с покрива, е време за **основата**. Тя е по-лесна за печатане:

- Започваме с цикъл от 0 до **n** (изключено).
- Отпечатваме на конзолата: **| + * (n - 2** на брой пъти) + **|**.

```

for (int i = 0; i < n / 2; i++) {
    cout << "|";
    for (int j = 0; j < n - 2; j++) {
        cout << "*";
    }
    cout << "|" << endl;
}

```

Ако всичко сме написали както трябва, задачата ни е решена.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1366#8>.

Пример: диамант

Да се напише програма, която приема цяло число n ($1 \leq n \leq 100$) и печата диамант с размер N , като в следните примери:

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
1	*	2	**	3	-*- *-*- -*-	4	-***- *--* -***-

Насоки и подсказки

Това, което знаем от условието на задачата, е че диамантът е с размер $n \times n$. От примерните вход и изход можем да си направим извода, че всички редове съдържат точно по n символа и всички редове, с изключение на горните върхове, имат по **2 звезди**. Можем мислено да разделим диаманта на 2 части:

- **Горна** част. Тя започва от горния връх до средата.
- **Долна** част. Тя започва от реда след средата до най-долния връх включително.

Горна част

- Ако n е **нечетно**, то тя започва с **1 звезда**.
- Ако n е **четно**, то тя започва с **2 звезди**.
- С всеки ред надолу звездите се отдалечават една от друга.
- Пространството между, преди и след **звездите** е запълнено с **тирета**.

Долна част

- С всеки ред надолу звездите се приближават една към друга. Това

означава, че пространството (**тиретата**) между тях намалява, а пространството (**тиретата**) отляво и отдясно се увеличава.

- В най-долната си част е с 1 или 2 **звезди**, спрямо това дали **n** е четно или не.

Горна и добрача на диаманта

- На всеки ред звездите са заобиколени от външни **тирета**, с изключение на средния ред.
- На всеки ред има пространство между двете **звезди**, с изключение на първия и последния ред (понякога **звездата е 1**).

Създаваме променлива **n** от тип **int** и прочитаме стойността ѝ от конзолата:

```
int n;
cin >> n;
```

Започваме да печатаме на конзолата горната част на диаманта. Първото нещо, което трябва да направим, е да изчислим началната стойност на външната бройка **тирета leftRight** (тиретата от външната част на звездите). Тя е равна на **(n - 1) / 2**, закръглено надолу:

```
int leftRight = (n - 1) / 2;
```

След като сме изчислили **leftRight**, започваме да чертаем **горната част** на диаманта. Може да започнем, като завъртим **цикъл** от **0** до **(n + 1) // 2** (т.e. закръглено надолу).

При всяка итерация на цикъла трябва да се изпълнят следните стъпки:

- Рисуваме по конзолата левите **тирета** (с дължина **leftRight**) и веднага след тях първата **звезда**.

```
for (int j = 0; j < leftRight; j++) {
    cout << "-";
}
cout << "*";
```

- Изчисляваме разстоянието между двете **звезди**. Може да го направим като извадим от **n** дължината на външните **тирета**, както и числото 2 (бройката на звездите, т.e. очертанията на диаманта). Резултата от тази разлика записваме в променлива **mid**.

```
int mid = n - 2 * leftRight - 2;
```

- Ако стойността на **mid** е по-малка от 0, то тогава знаем, че на този ред трябва да има 1 звезда. Ако е по-голяма или равно на 0, то тогава трябва да начертаем **тирета** с дължина **mid** и една **звезда** след тях.
- Печатаме на конзолата десните външни **тирета** с дължина също **leftRight**

```
for (int j = 0; j < leftRight; j++) {
    cout << "-";
}
```

- В края на цикъла намаляваме **leftRight** с 1 (звездите се отдалечават).

Чертането на долната част е доста подобно на това на горната част. Разликата е, че вместо да намаляваме стойността на **leftRight** с 1 към края на цикъла, ще я увеличаваме с 1 в началото на цикъла. Също така **цикълът ще се върти от 0 до (n - 1) // 2**:



Повторението на код се смята за лоша практика, защото кодът става доста труден за поддръжка. Нека си представим, че имаме парче код на още няколко места и решаваме да направим промяна. За целта би било необходимо да минем през всичките места и да направим промените. А сега нека си представим, че трябва да използвате код не 1, 2 или 3 пъти, а десетки пъти. Начин за справяне с този проблем е като се използват [функции](#).

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1366#9>.

Какво научихме от тази глава?

Запознахме се с един от начините за създаване на низове:

```
string printMe = string(10, '*');
```

Може да пуснете в действие и тествате кода от горния пример онлайн:
<https://repl.it/@vncpetrov/PrintMe>.

Научихме се да чертаем фигури с вложени **for** цикли:

```
for (int r = 0; r < 5; r++) {
    cout << "*";

    for (int c = 0; c < 5 - 1; c++) {
        cout << " *";
    }
    cout << endl;
}
```

Може да пуснете в действие и тествате кода от горния пример онлайн:

<https://repl.it/@vncpetrov/ForLoopFigure>.

Глава 6.2. Вложени цикли – изпитни задачи

В предходната глава разгледахме **вложените цикли** и как да ги използваме за **рисуване** на различни **фигури** на конзолата. Научихме се как да отпечатваме фигури с различни размери, измисляйки подходяща логика на конструиране с използване на **единични и вложени for** цикли в комбинация с различни изчисления и програмна логика:

```
for (int r = 1; r <= 5; r++) {  
    cout << '*';  
  
    for (int c = 1; c < 5; c++) {  
        cout << " *";  
    }  
    cout << endl;  
}
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/nestedLoops>.

Запознахме се и с **конструктора string**, който дава възможност да се печата даден символ определен от нас брой пъти:

```
string printMe (5, '*');
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/stringCtor>.

Изпитни задачи

Сега нека решим заедно няколко изпитни задачи, за да затвърдим наученото и да развием още алгоритмичното си мислене.

Задача: чертане на крепост

Да се напише програма, която прочита от конзолата **цяло число n** и чертае **крепост** с ширина **2 * n колони** и височина **n реда** като в примерите по-долу. Лявата и дясната колона във вътрешността си са широки $n / 2$.

Входни данни

Входът е **цяло число n** в интервала [3 ... 1000].

Изходни данни

Да се отпечатат на конзолата **n** текстови реда, изобразяващи **крепостта**, точно както в примерите.

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
3	/^\^/\^/ _/_/_	4	/^\^/\^/ _/_/_/_	5	/^\^/\^/ _/_/_/_/_

Насоки и подсказки

От условието на задачата виждаме, че **входните данни** ще се състоят само от един ред, който ще съдържа в себе си едно **цяло** числов интервала [3 ... 1000]. По тази причина ще използваме **променлива** от тип **int**:

```
int n;  
cin >> n;
```

След като вече сме декларирали и инициализирали входните данни, ще разделим **крепостта** на три части:

- покрив
- тяло
- основа

От примерите можем да разберем, че **покривът** е съставен от **две кули** и **междинна част**. Всяка кула се състои от начало /, среда ^ и край \.



\ е специален символ в езика C++ и използвайки го с обекта и потока за изход **cout <<**, конзолата няма да го разпечатва, затова с \\ показваме на конзолата, че искаме да отпечатаме точно този символ, без да се интерпретира като специален (екранираме го, на английски се нарича "character escaping").

Средата е с размер, равен на **n / 2**, следователно можем да отделим тази стойност в отделна променлива. Тя ще пази големината на средата на кулата.

```
int colSize = n / 2;
```

Декларираме и втора **променлива**, в която ще пазим **стойността** на частта **между двете кули**. Знаем, че по условие общата ширина на крепостта е **n * 2**. Освен това имаме и две кули с по една наклонена черта за начало и край (общо 4 знака) и ширина **colSize**. Следователно, за да получим броя на знаците в междинната част, трябва да извадим размера на кулите от ширината на цялата крепост: **2 * n - 2 * colSize - 4**:

```
int midSize = ... ;
```

За да отпечатаме на конзолата покрива, ще използваме два **for** цикъла, които ще разпечатат на екрана символите **^** и **_** съответно **colSize** и **midSize** на брой пъти:

```
cout << '/';

for (int i = 0; i < colSize; i++) {
    cout << '^';
}
cout << '\\';

for (int i = 0; i < midSize; i++) {
    cout << '_';
}
cout << '/';

for (int i = 0; i < colSize; i++) {
    cout << '^';
}
cout << '\\'
<< endl;
```



\ е специален символ в езика C++ и използвайки само него в функцията **cout**, конзолата няма да го разпечатва, затова с **** показваме на конзолата, че искаме да отпечатаме точно този символ, без да се интерпретира като специален (екранираме го, на английски се нарича "character escaping").

Тялото на крепостта се състои от начало |, среда (**празно място**) и край |. Средата от празно място е с големина **2 * n - 2**. Броят на редовете за стени, можем да определим от дадените ни примери - **n - 3**:

```
for (int row = 1; row <= n - 3; row++) {
    cout << '|';

    for (int i = 0; i < 2 * n - 2; i++) {
        cout << ' ';
    }
    cout << '|' << endl;
}
```

За да нарисуваме предпоследния ред, който е част от основата, трябва да отпечатаме начало |, среда (**празно място**)_(**празно място**) и край |. За да направим това, можем да използваме отново вече декларирани от нас

променливи **colSize** и **midSize**, защото от примерите виждаме, че са равни на броя в покрива.

```
cout << '|';
for (int i = 0; i < colSize + 1; i++) {
    cout << ' ';
}

for (int i = 0; i < [REDACTED]; i++) {
    cout << '_';
}

for (int i = 0; i < colSize + 1; i++) {
    cout << ' ';
}
cout << '|' << endl;
```

Добавяме към стойността на **празните места + 1**, защото в примерите имаме **едно** празно място повече.

Структурата на **основата на крепостта** е еднаква с тази на **покрива**. Съставена е от две **кули** и междинна част. Всяка една **кула** има начало \, среда _ и край /:

```
cout << '\\';

for (int i = 0; i < [REDACTED]; i++) {
    cout << '_';
}
cout << '/';

for (int i = 0; i < [REDACTED]; i++) {
    cout << ' ';
}
cout << '\\';

for (int i = 0; i < [REDACTED]; i++) {
    cout << '_';
}
cout << '/' << endl;
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1367#0>.

Задача: пеперуда

Да се напише програма, която прочита от конзолата **цяло число n** и чертае пеперуда с ширина $2 * n - 1$ колони и височина $2 * (n - 2) + 1$ реда като в примерите по-долу. Лявата и дясната ѝ част са **широки n - 1**.

Входни данни

Входът е **цяло число n** в интервала [3 ... 1000].

Изходни данни

Да се отпечатат на конзолата $2 * (n - 2) + 1$ текстови реда, изобразяващи пеперудата, точно както в примерите.

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
3	*\ /* @ */ *	5	***\ /** ---\ /--- ***\ /** @ ***/ *** ---/ \--- ***/ ***	7	*****\ /***** -----\ /----- *****\ /***** -----\ /----- *****\ /***** @ *****/ ***** -----/ \----- *****/ ***** -----/ \----- *****/ *****

Насоки и подсказки

От условието на задачата виждаме, че **входните данни** ще бъдат прочетени само от един ред, който ще съдържа в себе си едно **цяло число** в интервала [3 ... 1000]. По тази причина ще използваме **променлива** от тип **int**:

```
int n;  
cin >> n;
```

Можем да разделим фигурата на 3 части - **горно крило, тяло и долно крило**. За да начертаем горното крило на пеперудата, трябва да го разделим на части - начало *, среда \ / и край *. След разглеждане на примерите можем да кажем, че началото е с големина **n - 2**:

```
int halfRowSize = ...;
```

Виждаме също така, че горното крило на пеперудата е с размер **n - 2**, затова можем да направим цикъл, който да се повтаря **halfRowSize** пъти.

```
for (int i = 1; i <= halfRowSize; i++) {
}
```

От примерите можем също така да забележим, че на **четен** ред имаме начало *****, среда **\ /** и край *****, а на **нечетен** - начало **-**, среда **\ /** и край **-**. Следователно при всяка итерация на цикъла трябва да направим **if-else** проверка дали редът, който печатаме, е четен или нечетен. От примерите, дадени в условието, виждаме, че броят на звездичките и тиретата на всеки ред също е равен на **n - 2**, т. е. за тяхното отпечатване отново можем да използваме променливата **halfRowSize**:

```
for (int i = 1; i <= halfRowSize; i++) {
    if (i % 2 != 0) {
        for (int j = 1; j <= halfRowSize; j++) {
            cout << '*';
        }
        cout << "\\ /";
        for (int j = 1; j <= halfRowSize; j++) {
            cout << '*';
        }
        cout << endl;
    }
    else {
        for (int j = 1; j <= halfRowSize; j++) {
            cout << '-';
        }
        cout << "\\ /";
        for (int j = 1; j <= halfRowSize; j++) {
            cout << '-';
        }
        cout << endl;
    }
}
```

За да направим **тялото на пеперудата**, можем отново да използваме **променливата halfRowSize** и да отпечатаме на конзолата точно **един** ред. Структурата на

тялото е с начало (**празно място**), среда @ и край (**празно място**). От примерите виждаме, че броят на празните места е **n - 1**:

```
for (int i = 1; i <= [REDACTED]; i++) {
    cout << ' ';
}
cout << "@" << endl;
```

Остава да отпечатаме на конзолата и **долното крило**, което е **аналогично на горното крило**: единствено трябва да разменим местата на наклонените черти:

```
for (int i = 1; i <= [REDACTED]; i++) {
    if ([REDACTED]) {
        for (int j = 1; j <= [REDACTED]; j++) {
            [REDACTED]
        }
        [REDACTED]
        for (int j = 1; j <= [REDACTED]; j++) {
            [REDACTED]
        }
        cout << endl;
    }
    else {
        for (int j = 1; j <= [REDACTED]; j++) {
            cout << '-';
        }
        [REDACTED]
        for (int j = 1; j <= [REDACTED]; j++) {
            [REDACTED]
        }
        cout << endl;
    }
}
```

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:
<https://judge.softuni.bg/Contests/Practice/Index/1367#1>.

Задача: знак "Стоп"

Да се напише програма, която прочита от конзолата **цяло число n** и чертае предупредителен знак STOP с размери като в примерите по-долу.

Входни данни

Входът е **цяло число N** в интервала [3 ... 1000].

Изходни данни

Да се отпечатат на конзолата текстови редове, изобразяващи **предупредителния знак STOP**, точно както в примерите.

Примерен вход и изход

Вход	Изход	Вход	Изход
3//_____\\"... ...//_____\\"... .//_____\\". //____STOP!_____\\" ____/\// .____/\//. ..____/\//..	6_____.....//_____\\.....//_____\\.....//_____\\..... ...//_____\\... ..//_____\\.. ./_____\\.. //_____STOP!_____\\ ____/\// .____/\//.. ..____/\//...____/\//....____/\//.....

Насоки и подсказки

Както и при предходните задачи, **входните данни** ще бъдат прочетени само от един ред, който ще съдържа в себе си едно **цялов интervал** [3 ... 1000]:

```
int n;  
cin >> n;
```

Можем да разделим фигурата на **3 части** - горна, средна и долнна. **Горната част** се състои от две подчасти - начален ред и редове, в които знака се разширява. **Началния ред** е съставен от начало `.`, среда `_` и край `..`. След разглеждане на примерите можем да кажем, че началото е с големина **n + 1** и е добре да отделим тази **стойност** в отделна **променлива**.

```
int dots = ...;
```

```
int dots = ...;
```

Трябва да създадем и втора променлива, в която ще пазим стойността на средата на началния ред с големина $2 * n + 1$:

```
int underscores = 2 * n + 1;
```

След като вече сме декларирали и инициализирали двете променливи, можем да отпечатаме на конзолата началния ред:

```
for (int i = 0; i < ...; i++) {
    ...
}

for (int i = 0; i < ...; i++) {
    ...
}

for (int i = 0; i < ...; i++) {
    ...
}
cout << endl;
```

За да начертаем редовете, в които знае се "разширява", трябва да създадем цикъл, който да се завърти n брой пъти. Структурата на един ред се състои от начало ., // + среда _ + \\\ и край .. За да можем да използваме отново създадените променливи, трябва да намалим **dots** с 1 и **underscores** с 2, защото ние вече сме отпечатали първия ред, а точките и долните черти в горната част от фигурата на всеки ред намаляват:

```
underscores -= 2;
dots--;
```

На всяка следваща итерация началото и краят намаляват с 1, а средата се увеличава с 2:

```
for (int i = 0; i < ...; i++) {
    for (int j = 0; j < ...; j++) {
        cout << '.';
    }
    cout << "//";

    for (int j = 0; j < ...; j++) {
        cout << '_';
    }
}
```

```

cout << "\\\\";

for (int j = 0; j < _ ; j++) {
    cout << '.';
}
cout << endl;
underscores += 2;
dots--;
}

```

Средната част от фигурата има начало $// + _$, среда **STOP!** и край $_ + \\$. Броят на долните черти $_$ е $(underscores - 5) / 2$:

```

cout << "//";

for (int i = 0; i < _ ; i++) {
    cout << '_';
}
cout << "STOP!";

for (int i = 0; i < _ ; i++) {
    cout << '_';
}
cout << "\\\\" << endl;

```

Долната част на фигурата, в която знаци се **смалнява**, можем да направим като отново създадем **цикъл**, който да се завърти n брой пъти. Структурата на един ред е начало $. + \\$, среда $_$ и край $// + ..$ Броят на **точките** при първата итерация на цикъла трябва да е 0 и на всяка следваща да се **увеличава** с едно. Следователно можем да кажем, че големината на точките в **долната част от фигурата** е равна на i .

```

for (int i = 0; i < n; i++) {
    for (int j = 0; j < _ ; j++) {
        cout << '.';
    }
    cout << "\\\\";

    for (int j = 0; j < _ ; j++) {
        cout << '_';
    }
    cout << "//";
}

```

```

for (int j = 0; j < ; j++) {
    cout << '.';
}
cout << endl;
}

```

За да работи нашата програма правилно, трябва на всяка итерация от **цикъла** да намаляваме броя на _ с 2:

```
for (int i = 0; i < ; i++) {
```

```

    // ...
}

underscopes -= ;
}
```

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1367#2>.

Задача: стрелка

Да се напише програма, която прочита от конзолата **цяло нечетно число n** и чертае **вертикална стрелка** с размери като в примерите по-долу.

Входни данни

Входът е **цяло нечетно число n** в интервала [3 ... 79].

Изходни данни

Да се отпечата на конзолата вертикална стрелка, при която "#" (диез) очертава стрелката, а "." - останалото.

Примерен вход и изход

Вход	Изход
3	.####. .#.#. . ##.##. .#.#. . .#. . .

Вход	Изход
5	<pre>..#####.. ..#....#.. ..#....#.. ..#....#.. ##### .#.....#. ..#....#.. ...#.##....</pre>

Вход	Изход
9	<pre>.....#####.... #.....#... #.....#... #.....#... #.....#... #.....#... #.....#... #.....#... #.....#... #.....#... #####.....##### ..#.....#.....#. ..#.....#.....#. ..#.....#.....#. ..#.....#.....#. #.....#... #.....#... #.....#... #.....#...</pre>

Насоки и подсказки

От условието на задачата виждаме, че **входните данни** ще бъдат прочетени само от един ред, който ще съдържа в себе си едно **цяло число** в интервала [3 ... 79]. По тази причина ще използваме **променлива** от тип **int**:

```
int n;  
cin >> n;
```

Можем да разделим фигурата на 3 части - горна, средна и долната. Горната част се състои от две подчасти - начален ред и тяло на стрелката. От примерите виждаме, че броят на **външните точки** в началния ред и в тялото на стрелката са **(n - 1) / 2**. Тази стойност можем да запишем в променлива **outerDots**:

```
int outerDots = (n - 1) / 2;
```

Броят на **вътрешните точки** в тялото на стрелката е **(n - 2)**. Трябва да създадем променлива с име **innerDots**, която ще пази тази стойност:

```
int innerDots = n - 2;
```

От примерите можем да видим структурата на началния ред. Трябва да използваме декларираните и инициализирани от нас **променливи outerDots** и **n**, за да отпечатаме **началния ред**:

```
for (int i = 0; i < ; i++) {
    cout << '.';
}

for (int i = 0; i < ; i++) {
    cout << '#';
}

for (int i = 0; i < ; i++) {
    cout << '.';
}
cout << endl;
```

За да нарисуваме на конзолата тялото на стрелката, трябва да създадем цикъл, който да се повтори **n - 2** пъти:

```
for (int i = 0; i < ; i++) {
    for (int j = 0; j < ; j++) {
        cout << '.';
    }
    cout << '#';

    for (int j = 0; j < ; j++) {
        cout << '.';
    }
    cout << '#';

    for (int j = 0; j < ; j++) {
        cout << '.';
    }
    cout << endl;
}
```

Средата на фигурата е съставена от начало **#**, среда **.** и край **#**. От примерите виждаме, че броят на **#** е равен на **outerDots**, увеличен с 1 и за това можем да използваме отново същата **променлива**:

```

for (int i = 0; i < [REDACTED]; i++) {
}
for (int i = 0; i < [REDACTED]; i++) {
}
for (int i = 0; i < [REDACTED]; i++) {
}
cout << endl;

```

За да начертаем долната част на стрелката, трябва да зададем нови стойности на двете променливи **outerDots** и **innerDots**:

```

outerDots = 1;
innerDots = 2 * n - 5;

```

Цикъла, който ще направим, трябва да се завърти **n - 2** пъти и отделно ще отпечатаме последния ред от фигурата. На всяка итерация **outerDots** се увеличава с 1, а **innerDots** намалява с 2.

При всяка итерация **outerDots** се увеличава с 1, а **innerDots** намалява с 2. Забелязваме, че тъй като на предпоследния ред стойността на **innerDots** ще е 1 и при последвала итерация на цикъла тя ще стане **отрицателно число**. Тъй като конструкторът на **string** не може да съедини даден символ нула или отрицателен брой пъти в низ, няма да се изведе нищо на конзолата. Един вариант да избегнем това е да отпечатаме последния ред на фигурата отделно. Височината на долната част на стрелката е **n - 1**, следователно **ЦИКЪЛЪТ**, който ще отпечата всички редове без последния, трябва да се завърти **n - 2** пъти:

```

for (int i = 0; i < [REDACTED]; i++) {
    for (int j = 0; j < [REDACTED]; j++) {
    }
    cout << [REDACTED];
    for (int j = 0; j < [REDACTED]; j++) {
    }
    cout << [REDACTED];
}

```

```

for (int j = 0; j < ..... ; j++) {
    ...
    cout << endl;
    outerDots++;
    innerDots -= 2;
}

```

Последният ред от нашата фигура е съставен от начало ., среда # и край .. Броят на . е равен на **outerDots**:

```

for (int i = 0; i < ..... ; i++) {
    ...
    cout << . ;
    for (int i = 0; i < ..... ; i++) {
        ...
    }
    cout << endl;
}

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1367#3>.

Задача: брадва

Да се напише програма, която прочита **цяло число N** и чертае брадва с размери, показани по-долу. Ширината на брадвата е $5 * N$ колони.

Примерен вход и изход

Вход	Изход
2	<pre> -----**- -----*_-* *****_*_ -----***- </pre>

Вход	Изход
5	<pre> -----**- -----*_-* -----*_-* -----*_-* -----*_-* -----*_-* -----*_-* -----*_-* -----*_-* -----*****_- -----*****_- -----*****_- -----*****_- -----*****_- </pre>

Входни данни

Входът е **цяло число N** в интервала [2..42].

Изходни данни

Да се отпечата на конзолата **брадва**, точно както е в примерите.

Насоки и подсказки

От условието на задачата виждаме, че **входните данни** ще бъдат прочетени само от един ред, който ще съдържа в себе си едно **цяло число** в интервала [2 ... 42]. По тази причина ще използваме тип **int**. След което, за решението на задачата е нужно първо да изчислим големината на **тиратата от ляво**, **средните тирета**, **тиратата от дясно** и цялата дължина на фигурата:

```
width = 5 * n;  
leftDashes = 3 * n;  
middleDashes = 0;  
rightDashes = width - leftDashes - middleDashes - 2;
```

След като сме декларирали и инициализирали **променливите**, можем да започнем да изчертаваме фигурата като започнем с **горната част**. От примерите можем да разберем каква е структурата на **първия ред** и да създадем цикъл, който

се повтаря **n** на брой пъти. При всяка итерация от цикъла **средните тирета** се увеличават с 1, а **тиретата отдясно** се намаляват с 1:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        cout << " ";
    }
    cout << "*";

    for (int j = 0; j < n - i; j++) {
        cout << "-";
    }
    cout << "*";

    for (int j = 0; j < n - i - 1; j++) {
        cout << " ";
    }
    cout << endl;
}
```

Сега следва да нарисуваме **дръжката на брадвата**. За да можем да използваме отново създадените **променливи** при чертането на дръжката на брадвата, трябва да намалим **средните тирета** с 1, а **тези отдясно и отляво** да увеличим с 1:

```
middleDashes ;
rightDashes ;
```

Дръжката на брадвата можем да нарисуваме, като завъртим цикъл, който се повтаря **n / 2** пъти. От примерите можем да разберем, каква е нейната структура:

```
(int i = 0; i < n / 2; i++) {
    for (int j = 0; j < i; j++) {
        cout << " ";
    }
    cout << "*";

    for (int j = 0; j < n - i - 1; j++) {
        cout << " ";
    }
    cout << "*";
```

```

for (int j = 0; j < ; j++) {
    cout << ;
}
cout << endl;
}

```

Долната част на фигурата, трябва да разделим на две подчасти - **глава на брадвата** и **последния ред от фигурата**. Главата на брадвата ще отпечатаме на конзолата, като направим цикъл, който да се повтаря $n / 2 - 1$ пъти. На всяка итерация тиретата от ляво и тиретата от дясно намаляват с 1, а средните тирета се увеличават с 2:

```

for ( ) {
    for (int j = 0; j < ; j++) {
        cout << ;
    }
    cout << '*';

    for (int j = 0; j < ; j++) {
        cout << ;
    }
    cout << '*';

    for (int j = 0; j < ; j++) {
        cout << ;
    }
    cout << endl;
}

```

За последния ред от фигурата, можем отново да използваме трите, вече деклариирани и инициализирани променливи **leftDashes**, **middleDashes**, **rightDashes**:

```

for (int j = 0; j < ; j++) {
    cout << ;
}
cout << ;

```

```
for (int j = 0; j < ..... ; j++) {  
    cout << .....;  
}  
cout << .....;  
  
for (int j = 0; j < ..... ; j++) {  
    cout << .....;  
}  
cout << endl;
```

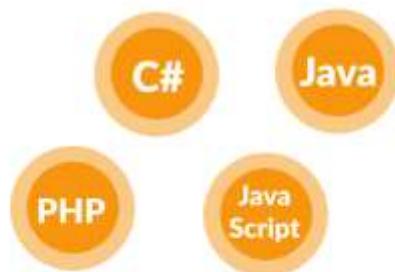
Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1367#4>.

Качествено образование,
професия и работа за
Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



СофТУни предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата **"Софтуерен университет"** изгражда истински **профессионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на СофТУни, за да усвоите **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

СофТУни работи пряко с компаниите от софтуерната индустрия, съдействайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ



1 - 1.5 години



1.5 - 2 години



Programming Basics

Кариерен Старт

Дипломиране

70/100 credits

80/100/150 credits

Кандидатствай

softuni.bg/apply

Глава 7.1. По-сложни цикли

След като научихме какво представляват и за какво служат **for** циклите, сега предстои да се запознаем с **други видове цикли**, както и с някои **по-сложни конструкции за цикъл**. Те ще разширят познанията ни и ще ни помогнат в решаването на по-трудни и по-предизвикателни задачи. По-конкретно, ще разгледаме как се ползват следните програмни конструкции:

- цикли **със стъпка**
- **while** цикли
- **do-while** цикли
- **безкрайни** цикли

В настоящата тема ще разберем и какво представлява операторът **break**, както и как чрез него да прекъснем един цикъл.

Видео

Гледайте видео урок по учебния материал от настоящата глава от книгата: <https://www.youtube.com/watch?v=JWaS7V0QIGk>.

Цикли със стъпка

В глава [5.1 Повторения \(цикли\)](#) научихме как работи **for** цикълът и вече знаем кога и с каква цел да го използваме. В тази тема ще обърнем **внимание** на една определена и много важна **част от конструкцията** му, а именно **стъпката**.

Какво представлява стъпката?

Стъпката е тази **част** от конструкцията на **for** цикъла, която указва с **колко** да се **увеличи** или **намали** стойността на **водещата** му променлива. Тя се декларира последна в скелета на **for** цикъла.

Най-често е с **размер 1** и в такъв случай, вместо да пишем **i += 1** или **i -= 1**, можем да използваме операторите **i++** или **i--**. Ако искаме стъпката ни да е **различна от 1**, при увеличение използваме оператора **i += (размера на стъпката)**, а при намаляване **i -= (размера на стъпката)**. При стъпка 3, цикълът би изглеждал по следния начин:

```
for (int i = 1; i <= 10; i += 3) {  
    cout << i << endl;  
}
```

Задаване
на стъпка

Следва поредица от примерни задачи, решението на които ще ни помогне да разберем по-добре употребата на **стъпката** във **for** цикъл.

Пример: числата от 1 до N през 3

Да се напише програма, която отпечатва числата от 1 до n със стъпка 3. Например, ако $n = 100$, то резултатът ще е: 1, 4, 7, 10, ..., 94, 97, 100.

Можем да решим задачата чрез следната поредица от действия (алгоритъм):

- Четем числото **n** от входа на конзолата.
- Изпълняваме **for цикъл** от 1 до **n** (включително и **n**) с размер на стъпката 3.
- В **тялото на цикъла** отпечатваме стойността на текущата стъпка.

```
int n;
cin >> n;

//чрез i+= 3 повишаваме стойността на i с размера на стъпката
for (int i = 1; i <= n; i += 3) {
    cout << i << endl;
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1368#0>.

Пример: числата от N до 1 в обратен ред

Да се напише програма, която отпечатва числата от n до 1 в обратен ред (стъпка -1). Например, ако $n = 100$, то резултатът ще е: 100, 99, 98, ..., 3, 2, 1.

Можем да решим задачата по следния начин:

- Четем числото **n** от входа на конзолата.
- Създаваме **for цикъл**, от **n** до 0.
- Дефинираме размера на стъпката: **-1**.
- В **тялото на цикъла** отпечатваме стойността на текущата стъпка.

```
int n;
cin >> n;

for (int i = n; i >= 1; i--) {
    cout << i << endl;
}
```

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1368#1>.

Пример: числата от 1 до 2^n с for цикъл

В следващия пример ще разгледаме ползването на обичайната стъпка с размер 1.

Да се напише програма, която отпечатва числата от 1 до 2^n (две на степен n). Например, ако $n = 10$, то резултатът ще е 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024.

```
int n;
cin >> n;
int num = 1;

for (int i = 0; i <= n; i++) {
    cout << num << endl;
    num = num * 2;
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1368#2>.

Пример: четни степени на 2

Да се отпечатат четните степени на 2 до 2^n : $2^0, 2^2, 2^4, 2^8, \dots, 2^n$. Например, ако $n = 10$, то резултатът ще е 1, 4, 16, 64, 256, 1024.

Ето как можем да решим задачата:

- Създаваме променлива **num** за текущото число, на която присвояваме начална **стойност 1**.
- За **стъпка** на цикъла задаваме стойност **2**.
- В **тялото на цикъла**: отпечатваме стойността на текущото число и **увеличаваме текущото число num 4 пъти** (според условието на задачата).

```
int n;
cin >> n;
int num = 1;

for (int i = 0; i <= n; i += 2) {
    cout << num << endl;
    num = num * 2 * 2;
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1368#3>.

While цикъл

Следващият вид цикли, с които ще се запознаем, се наричат **while** цикли. Специфичното при тях е, че повтарят блок от команди, **докато дадено условие е истина**. Като структура се различават от тази на **for** циклите, даже имат опростен синтаксис.

Какво представлява while цикълът?

В програмирането **while** цикълът се използва, когато искаме да повтаряме извършването на определена логика, докато е в сила дадено условие. Под "условие", разбираем всеки израз, който връща **true** или **false**. Когато условието стане **грешно**, **while** цикълът прекъсва изпълнението си и програмата **продължава** с изпълнението на кода след цикъла. Конструкцията за **while** цикъл изглежда по този начин:



Следва поредица от примерни задачи, решението на които ще ни помогне да разберем по-добре употребата на **while** цикъла.

Пример: редица числа $2k+1$

Да се напише програма, която отпечатва всички **числа $\leq n$** от редицата: 1, 3, 7, 15, 31, ..., като приемем, че всяко следващо число = **предишно число * 2 + 1**.

Ето как можем да решим задачата:

- Създаваме променлива **num** за текущото число, на която присвояваме начална **стойност 1**.
- За условие на цикъла слагаме **текущото число $\leq n$** .
- В **тялото на цикъла**: отпечатваме стойността на текущото число и увеличаваме текущото число, използвайки формулата от условието на задачата.

Ето и примерна реализация на описаната идея:

```

int n;
cin >> n;
int num = 1;
    
```

```
while (num <= n) {
    cout << num << endl;
    num = num * 2 + 1;
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1368#4>.

Пример: число в диапазона [1 ... 100]

Да се въведе цяло число в диапазона [1 ... 100]. Ако въведеното число е невалидно, да се въведе отново. В случая, за невалидно число ще считаме всяко такова, което **не е** в зададения диапазон.

За да решим задачата, можем да използваме следния алгоритъм:

- Създаваме променлива **n**, на която присвояваме целочислената стойност, получена от входа на конзолата.
- За условие на цикъла слагаме израз, който е **true**, ако числото от входа **не е** в диапазона посочен в условието.
- В **тялото на цикъла**: отпечатваме съобщение със съдържание "Invalid number!" на конзолата, след което присвояваме нова стойност за **n** от входа на конзолата.
- След като вече сме валидирали въведеното число, извън тялото на цикъла отпечатваме стойността на числото.

Ето и примерна реализация на алгоритъма чрез **while** цикъл:

```
int n;
cin >> n;

while (n < 1 || n > 100) {
    cout << "Invalid number!" << endl;
    cin >> n;
}

cout << "The number is: " << n << endl;
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1368#5>.

Най-голям общ делител (НОД)

Преди да продължим към следващата задача, е необходимо да се запознаем с определението за **най-голям общ делител (НОД)**.

Определение за НОД: най-голям общ делител на две естествени числа **a** и **b** е най-голямото число, което се дели едновременно и на **a**, и на **b** без остатък.

Например:

a	b	НОД
24	16	8
67	18	1
12	24	12

a	b	НОД
15	9	3
10	10	10
100	88	4

Алгоритъм на Евклид

В следващата задача ще използваме един от първите публикувани алгоритми за намиране на НОД - **алгоритъм на Евклид**:

Докато не достигнем остатък 0:

- Делим по-голямото число на по-малкото.
- Вземаме остатъка от делението.

Псевдо-код за алгоритъма на Евклид:

```
while b ≠ 0
    int oldB = b;
    b = a % b;
    a = oldB;
print a;
```

Пример: най-голям общ делител (НОД)

Да се подадат цели числа **a** и **b** и да се намери НОД(**a**, **b**).

Ще решим задачата чрез **алгоритъма на Евклид**:

- Създаваме променливи **a** и **b**, на които присвояваме целочислени стойности, взети от входа на конзолата.
- За условие на цикъла слагаме израз, който е **True**, ако числото **b** е различно от 0.
- В тялото на цикъла следваме указанията от псевдо кода:
 - Създаваме временна променлива, на която присвояваме текущата стойност на **b**.
 - Присвояваме нова стойност на **b**, която е остатъка от делението на **a** и **b**.
 - На променливата **a** присвояваме предишната стойност на

променливата **b**.

- След като цикълът приключи и сме установили НОД, го отпечатваме на екрана.

```
int a, b;
cin >> a >> b;

while (b != 0) {
    int oldB = b;
    b = a % b;
    a = oldB;
}

cout << "GCD = " << a << endl;
```

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1368#6>.

Do-while цикъл

Следващият цикъл, с който ще се запознаем, е **do-while**, който в превод означава **прави-докато**. По структура, той наподобява **while**, но има съществена разлика между тях. Тя се състои в това, че при **do-while** тялото се изпълнява **поне веднъж**. Защо се случва това? В конструкцията на **do-while** цикъла, **условието** винаги се проверява **след** тялото му, което от своя страна гарантира, че при **първото завъртане** на цикъла, кодът ще се **изпълни**, а проверката за **край на цикъл** ще се прилага върху всяка **следваща** итерация на **do-while**:

Ще разгледаме примерни задачи, чито решения ще ни помогнат да разберем по-добре **do-while** цикъла. Той представлява удобен способ при извършването на циклична проверка на стойности, в следствие на която се изпълняват дадени програмни фрагменти.



Пример: изчисляване на факториел

За естествено число n да се изчисли $n! = 1 * 2 * 3 * \dots * n$. Например, ако $n = 5$, то резултатът ще бъде: $5! = 1 * 2 * 3 * 4 * 5 = 120$.

Ето как по-конкретно можем да пресметнем факториел:

- Създаваме променливата **n**, на която присвояваме целочислена стойност взета от входа на конзолата.
- Създаваме още една променлива - **fact**, чиято начална стойност е 1. Ней ще използваме за изчислението и съхранението на факториела.
- За условие на цикъла ще използваме **n > 1**, тъй като всеки път, когато извършим изчисленията в тялото на цикъла, ще намаляваме стойността на **n** с 1.
- В тялото на цикъла:
 - Присвояваме нова стойност на **fact**, която е резултат от умножението на текущата стойност на **fact** с текущата стойност на **n**.
 - Намаляваме стойността на **n** с **-1**.
- Извън тялото на цикъла отпечатваме крайната стойност на факториела.

```
do {
    fact = fact * n;
    n--;
}
while (n > 1);

cout << fact << endl;
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1368#7>.

Пример: сумиране на цифрите на число

Да се сумират цифрите на цяло **положително** число n . Например, ако $n = 5634$, то резултатът ще бъде: $5 + 6 + 3 + 4 = 18$.

Можем да използваме следната идея, за да решим задачата:

- Създаваме променливата **n**, на която присвояваме стойност, равна на въведеното от потребителя число.
- Създаваме втора променлива - **sum**, чиято начална стойност е 0. Ней ще използваме за изчислението и съхранението на резултата.
- За условие на цикъла ще използваме **n > 0**, тъй като след всяко изчисление на резултата в тялото на цикъла, ще премахваме последната

цифра от **n**.

- В тялото на цикъла:
 - Присвояваме нова стойност на **sum**, която е резултат от събирането на текущата стойност на **sum** с последната цифра на **n**.
 - Присвояваме нова стойност на **n**, която е резултат от премахването на последната цифра от **n**.
- Извън тялото на цикъла отпечатваме крайната стойност на сумата.

```
int n;
cin >> n;
int sum = 0;
do {
    sum = sum + (n % 10);
    n = n / 10;
}
while (n > 0);
cout << "Sum of digits: " << sum << endl;
```



n % 10: връща последната цифра на числото **n**.
n / 10: изтрива последната цифра на **n**.

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1368#8>.

Безкрайни цикли и операторът **break**

До момента се запознахме с различни видове цикли, като научихме какви конструкции и приложения имат. Следва да разберем какво е **безкраен цикъл**, кога възниква и как можем да прекъснем изпълнението му чрез оператора **break**.

Безкраен цикъл. Що е то?

Безкраен цикъл наричаме този цикъл, който **повтаря безкрайно** изпълнението на тялото си. При **while** и **do-while** циклите проверката за край е условен израз, който **винаги** връща **true**. Безкраен **for** възниква, когато **липсва условие за край**. Ето как изглежда **безкраен while** цикъл:

```
while (true) {
    cout << "Infinite loop" << endl;
}
```

А така изглежда **безкраен for цикъл**:

```
for (;;) {
    cout << "Infinite loop" << endl;
}
```

Оператор break

Вече знаем, че безкрайният цикъл изпълнява определен код до безкрайност, но какво става, ако желаем в определен момент при дадено условие, да излезем принудително от цикъла? На помощ идва операторът **break**, в превод - **спри, прекъсни**.



Операторът **break** спира изпълнението на цикъла към момента, в който е извикан, и продължава от първия ред след края на цикъла. Това означава, че текущата итерация на цикъла няма да бъде завършена до край и съответно останалата част от кода в тялото на цикъла няма да се изпълни.

Пример: прости числа

В следващата задача се изисква да направим **проверка за просто число**. Преди да продължим към нея, нека си припомним какво са простите числа.

Определение: едно цяло число е **просто**, ако се дели без остатък единствено на себе си и на 1. По дефиниция простите числа са положителни и по-големи от 1. Най-малкото просто число е **2**.

Можем да приемем, че едно цяло число **n** е просто, ако **$n > 1$** и **n** не се дели на число между **2** и **$n-1$** .

Първите няколко прости числа са: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, ...

За разлика от тях, **непростите (композитни) числа** са такива числа, чиято композиция е съставена от произведение на прости числа.

Ето няколко примерни непрости числа:

- $10 = 2 * 5$
- $42 = 2 \cdot 3 \cdot 7$
- $143 = 13 * 11$

Алгоритъм за проверка дали дадено цяло число е **просто**: проверяваме дали **$n > 1$** и дали **n** се дели на **2, 3, ..., n-1** без остатък.

- Ако се раздели на някое от числата, значи е **композитно**.
- Ако не се раздели на никое от числата, значи е **просто**.



Можем да оптимизираме алгоритъма, като вместо проверката да е до $n-1$, да се проверяват делителите до \sqrt{n} . Помислете защо.

Пример: проверка за просто число. Оператор break

Да се провери дали едно число n е просто. Това ще направим като проверим дали n се дели на числата между 2 и \sqrt{n} .

Ето го алгоритъмът за проверка за просто число, разписан по стъпки:

- Създаваме променливата n , на която присвояваме цяло число въведено от входа на конзолата.
- Създаваме булева променлива **isPrime** с начална стойност **true**.
Приемаме, че едно число е просто до доказване на противното.
- Създаваме **for** цикъл, на който като начална стойност за променливата на цикъла задаваме 2, за условие **текущата ѝ стойност $\leq \sqrt{n}$** . Стъпката на цикъла е 1.
- В тялото на цикъла проверяваме дали n , разделено на **текущата стойност** има остатък. Ако от делението **няма остатък**, то променяме **isPrime** на **false** и излизаме принудително от цикъла чрез оператор **break**.
- В зависимост от стойността на **isPrime** отпечатваме дали числото е просто (**true**) или съответно съставно (**false**).

Ето и примерна имплементация на описания алгоритъм:

```
int n;
cin >> n;
bool isPrime = true;

for (int i = 2; i <= sqrt(n); i++) {
    if (n % i == 0) {
        isPrime = false;
        break;
    }
}

if (isPrime) {
    cout << "Prime" << endl;
}
else {
    cout << "Not prime" << endl;
}
```

Оставаме да се провери дали входното число е по-голямо от 1, защото по дефиниция числата 0, 1, -1 и -2 не са прости.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1368#9>.

Пример: оператор break в безкрайен цикъл

Да се напише програма, която проверява дали едно число **n** е четно, ако е - да се отпечатва на екрана. За четно считаме число, което се дели на 2 без остатък. При невалидно число да се връща към повторно извеждане и да се изписва съобщение, което известява, че въведеното число не е четно.

Ето една идея как можем да решим задачата:

- Създаваме променлива **n**, на която присвояваме начална стойност 0.
- Създаваме безкрайен **while** цикъл, като за условие ще зададем **true**.
- В тялото на цикъла:
 - Вземаме целочислена стойност от входа на конзолата и я присвояваме на **n**.
 - Ако **числото е четно**, излизаме от цикъла чрез **break**.
 - В **противен случай** извеждаме съобщение, което гласи, че **числото не е четно**. Итерациите продължават, докато не се въведе четно число.
- Отпечатваме четното число на екрана.

Ето и примерна имплементация на идеята:

```
int n = 0;

while (true) {
    cout << "Enter even number: " << endl;
    cin >> n;

    if (n % 2 == 0) {
        break;
    }
    cout << "The number is not even." << endl;
}
cout << "Even number entered: " << n << endl;
```

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1368#10>.

Вложени цикли и операторът **break**

След като вече научихме какво са **вложените цикли** и как работи операторът **break**, е време да разберем как работят двете заедно. За по-добро разбиране, нека стъпка по стъпка да напишем **програма**, която трябва да направи всички възможни комбинации от **двойки числа**. Първото число от комбинацията е нарастващо от 1 до 3, а второто е намаляващо от 3 до 1. Задачата трябва да продължи изпълнението си, докато **i + j** не е равно на 2 (т.е. **i = 1** и **j = 1**).

Желаният резултат е:

```
C:\WINDOWS\system32\cmd.exe
1 3
1 2
Press any key to continue . . .
```

Ето едно **грешно решение**, което изглежда правилно на пръв поглед:

```
for (int i = 1; i <= 3; i++) {
    for (int j = 3; j >= 1; j--) {
        if (i + j == 2) {
            break;
        }

        cout << i << " " << j << endl;
    }
}
```

Ако оставим програмата ни по този начин, резултатът ни ще е следният:

```
C:\WINDOWS\system32\cmd.exe
1 3
1 2
2 3
2 2
2 1
3 3
3 2
3 1
Press any key to continue . . .
```

Защо се получава така? Както виждаме, в резултата липсва "1 1". Когато програмата стига до там, че **i = 1** и **j = 1**, тя влиза в **if** проверката и изпълнява **break** операцията. По този начин се **излиза от вътрешния цикъл**, но след това продължава изпълнението на външния. **i** нараства, програмата влиза във вътрешния цикъл и принтира резултата.



Когато във **вложен цикъл** използваме оператора **break**, той прекъсва изпълнението **само** на вътрешния цикъл.

Какво е **правилното решение**? Един начин за решаването на този проблем е чрез деклариране на **bool** променлива, която следи за това, дали трябва да продължава въртенето на цикъла. При нужда от изход (излизане от всички вложени цикли), се променя стойността на променливата на **true** и се излиза от вътрешния цикъл с **break**, а при последваща проверка се напуска и външният цикъл. Ето и примерна имплементация на тази идея:

```
bool hasToEnd = false;
for (int i = 1; i <= 3; i++) {
    if (hasToEnd == false) {
        for (int j = 3; j >= 1; j--) {
            if (i + j == 2) {
                hasToEnd = true;
                break;
            }

            cout << i << " " << j << endl;
        }
    }
}
```

По този начин, когато **i + j = 2**, програмата ще направи променливата **hasToEnd = true** и ще излезе от вътрешния цикъл. При следващото завъртане на външния цикъл, чрез **if** проверката, програмата няма да може да стигне до вътрешния цикъл и ще прекъсне изпълнението си.

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1368#11>.

Задачи с цикли

В тази глава се запознахме с няколко нови вида цикли, с които могат да се правят

повторения с по-сложна програмна логика. Да решим няколко задачи, използвайки новите знания.

Задача: числа на Фиbonачи

Числата на Фибоначи в математиката образуват редица, която изглежда по следния начин: **1, 1, 2, 3, 5, 8, 13, 21, 34,**, където всеки следващ член е равен по стойност на сума на предходните два. Първите два са приемат за равни на единица.

Формулата за образуване на редицата е:

$$F_0 = 1$$

$$F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$

Примерен вход и изход (стр. 272)

Вход (n)	Изход	Коментар
10	89	$F(11) = F(9) + F(8)$
5	8	$F(5) = F(4) + F(3)$
20	10946	$F(20) = F(19) + F(18)$

Вход (n)	Изход
0	1
1	1

Да се въведе **цяло** число **n** и да се пресметне **n**-тото число на Фибоначи.

Насоки и подсказки

Идея за решаване на задачата:

- Създаваме **променлива n**, на която присвояваме целочислена стойност от входа на конзолата.
- Създаваме променливите **f0** и **f1**, на които присвояваме стойност **1**, тъй като така започва редицата.
- Създаваме **for** цикъл от нула до **краяна стойност n - 1**.
- В **тялото на цикъла**:
 - Създаваме **временна** променлива **fNext**, на която присвояваме следващото число в поредицата на Фибоначи.
 - На **f0** присвояваме текущата стойност на **f1**.
 - На **f1** присвояваме стойността на временната променлива **fNext**.
- Извън цикъла отпечатваме числото n-тото число на Фибоначи.

Примерна имплементация:

```

int n;
cin >> n;
int f0 = 1;
int f1 = 1;

for (int i = 0; i < n - 1; i++) {
    int fNext = f0 + f1;
    f0 = f1;
    f1 = fNext;
}

cout << f1 << endl;

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1368#12>

Задача: пирамида от числа

Да се отпечатат числата 1 ... n в **пирамида** като в примерите по долу. На първия ред печатаме едно число, на втория ред печатаме две числа, на третия ред печатаме три числа и т.н. докато числата свършат. На последния ред печатаме толкова числа, колкото останат докато стигнем до n.

Примерен вход и изход

Вход	Изход
7	1 23 456 7

Вход	Изход
5	1 23 45

Вход	Изход
10	1 23 456 7 8 9 10

Насоки и подсказки

Можем да решим задачата с **два вложени цикъла** (по редове и колони) с печатане в тях и излизане при достигане на последното число. Ето идеята, разписана по-подробно:

- Създаваме променлива **n**, на която присвояваме целочислена стойност, прочетена от конзолата.
- Създаваме променлива **num** с начална стойност 1. Тя ще пази броя на отпечатаните числа. При всяка итерация ще я **увеличаваме** с **1** и ще я принтираме.
- Създаваме **външен for** цикъл, който ще отговаря за **редовете** в таблицата.

Наименуваме променливата на цикъла **row** и ѝ задаваме начална стойност 1. За условие слагаме **row <= n**. Размерът на стъпката е 1.

- В тялото на цикъла създаваме **вътрешен for** цикъл, който ще отговаря за **колоните** в таблицата. Наименуваме променливата на цикъла **col** и ѝ задаваме начална стойност 1. За условие слагаме **col <= row** (**row** = брой цифри на ред). Размерът на стъпката е 1.
- В тялото на вложения цикъл:
 - Проверяваме дали **col > 1**, ако да – принтираме разстояние. Ако не направим тази проверка, а директно принтираме разстоянието, ще имаме ненужно такова в началото на всеки ред.
 - Отпечатваме числото **num** в текущата клетка на таблицата и го **увеличаваме с 1**.
 - Правим проверка за **num > n**. Ако **num** е по-голямо от **n**, прекъсваме изпълнението на **вътрешния цикъл**.
- Отпечатваме **празен ред**, за да преминем на следващия.
- Отново проверяваме дали **num > n**. Ако е по-голямо, прекъсваме изпълнението на **програмата ни** чрез **break**.

```
int n;
cin >> n;
int num = 1;

for (int row = 1; row <= n; row++) {
    for (int col = 1; col <= row; col++) {
        if (col > 1) {
            cout << " ";
        }

        cout << num;
        num++;

        if (num > n) {
            break;
        }
    }
    cout << endl;

    if (num > n) {
        break;
    }
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1368#13>

Задача: таблица с числа

Да се отпечатат числата 1 ... n в таблица като в примерите.

Примерен вход и изход

Вход	Изход	Вход	Изход
3	1 2 3 2 3 2 3 2 1	4	1 2 3 4 2 3 4 3 3 4 3 2 4 3 2 1

Насоки и подсказки

Можем да решим задачата с **два вложени цикъла** и малко изчисления в тях:

- Четем от конзолата размера на таблицата в целочислена променлива **n**.
- Създаваме **for** цикъл, който ще отговаря за редовете в таблицата.
Наименуваме променливата на цикъла **row** и ѝ задаваме начална **стойност 0**. За условие слагаме **row < n**. Размерът на стълката е 1.
- В **тялото на цикъла** създаваме вложен **for** цикъл, който ще отговаря за колоните в таблицата. Наименуваме променливата на цикъла **col** и ѝ задаваме начална **стойност 0**. За условие слагаме **col < n**. Размерът на стълката е 1.
- В **тялото на вложния цикъл**:
 - Създаваме променлива **num**, на която присвояваме резултата от **текущият ред + текущата колона + 1** (+1, тъй като започваме броенето от 0).
 - Правим проверка за **num > n**. Ако **num** е **по-голямо от n**, присвояваме нова стойност на **num** равна на **два пъти n - текущата стойност за num**. **Това правим с цел да не превишаваме n**** в никоя от клетките на таблицата.
 - Отпечатваме числото от текущата клетка на таблицата.
- Отпечатваме **празен ред** във външния цикъл, за да преминем на следващия ред.

```
int n;
cin >> n;
```

```

for (int row = 0; row < n; row++) {
    for (int col = 0; col < n; col++) {
        int num = row + col + 1;

        if (num > n) {
            num = 2 * n - num;
        }

        cout << num << " ";
    }
    cout << endl;
}

```

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1368#14>.

Какво научихме от тази глава?

Можем да използваме **for** цикли със стъпка:

```

for (int i = 1; i <= n; i += 3) {
    cout << i << endl;
}

```

Може да пуснете в действие и тествате примера онлайн:

<https://repl.it/@vncpetrov/ForLoopWithStep>.

Циклите **while** / **do-while** се повтарят докато е в сила дадено **условие**:

```

int num = 1;
while (num <= n) {
    cout << num++ << endl;
}

```

Може да пуснете в действие и тествате примера онлайн:

<https://repl.it/@vncpetrov/WhileLoop>.

Ако се наложи да прекъснем изпълнението на цикъл, го правим с оператора **break**:

```
int n = 0;

while (true) {
    cin >> n;

    if (n % 2 == 0) {
        break; // even number -> exit from the loop
    }
    cout << "The number is not even." << endl;
}
cout << "Even number entered: " << n << endl;
```

Може да пуснете в действие и тествате примера онлайн:

<https://repl.it/@vncpetrov/OperatorBreak>.

Глава 7.2. По-сложни цикли – изпитни задачи

Вече научихме как може да изпълним даден блок от команди повече от веднъж използвайки **for** цикъл. В предходната глава разглеждахме още няколко **циклични конструкции**, които биха ни помогнали при решаването на по-сложни проблеми, а именно:

- цикли със стъпка
- вложени цикли
- **while** цикли
- **do-while** цикли
- безкрайни цикли и излизане от цикъл (**break** оператор)

Изпитни задачи

Нека затвърдим знанията си като решим няколко по-сложни задачи с цикли, давани на приемни изпити.

Задача: генератор за тъпи пароли

Да се напише програма, която въвежда две цели числа **n** и **l** и генерира по азбучен ред всички възможни "тъпи" пароли¹, които се състоят от следните 5 символа:

- Символ 1: цифра от 1 до **n**.
- Символ 2: цифра от 1 до **n**.
- Символ 3: малка буква измежду първите **l** букви на латинската азбука.
- Символ 4: малка буква измежду първите **l** букви на латинската азбука.
- Символ 5: цифра от 1 до **n**, по-голяма от първите 2 цифри.

Входни данни

Входът се чете от конзолата и се състои от **две цели числа: n и l** в интервала [1 ... 9], по едно на ред.

Изходни данни

На конзолата трябва да се отпечатат всички "тъпи" пароли по азбучен ред, разделени с **интервал**.

Примерен вход и изход (стр. 282)

Вход	Изход	Вход	Изход
2 4	11aa2 11ab2 11ac2 11ad2 11ba2 11bb2 11bc2 11bd2 11ca2 11cb2 11cc2 11cd2 11da2 11db2 11dc2 11dd2	3 1	11aa2 11aa3 12aa3 21aa3 22aa3

Вход	Изход	Вход	Изход
4 2	11aa2 11aa3 11aa4 11ab2 11ab3 11ab4 11ba2 11ba3 11ba4 11bb2 11bb3 11bb4 12aa3 12aa4 12ab3 12ab4 12ba3 12ba4 12bb3 12bb4 13aa4 13ab4 13ba4 13bb4 21aa3 21aa4 21ab3 21ab4 21ba3 21ba4 21bb3 21bb4 22aa3 22aa4 22ab3 22ab4 22ba3 22ba4 22bb3 22bb4 23aa4 23ab4 23ba4 23bb4 31aa4 31ab4 31ba4 31bb4 32aa4 32ab4 32ba4 32bb4 33aa4 33ab4 33ba4 33bb4	3 2	11aa2 11aa3 11ab2 11ab3 11ba2 11ba3 11bb2 11bb3 12aa3 12ab3 12ba3 12bb3 21aa3 21ab3 21ba3 21bb3 22aa3 22ab3 22ba3 22bb3

Насоки и подсказки

Решението на задачата можем да разделим мислено на три части:

- **Прочитане на входните данни** – в настоящата задача това включва прочитането на две числа **n** и **l**, всяко на отделен ред.
- **Обработка на входните данни** – използване на вложени цикли за преминаване през всеки възможен символ, за всеки от петте символа на паролата.
- **Извеждане на резултат** – отпечатване на всяка "тъпа" парола, която отговаря на условията.

Прочитане и обработка на входните данни

За прочитане на **входните** данни ще декларираме две променливи от целочислен тип **int** - **n** и **l**:

```
int n, l;
cin >> n;
cin >> l;
```

Извеждане на резултат

Един от начините да намерим решението на тази задача е да създадем **пет for** цикъла, вложени един в друг, по един за всеки символ от паролата. За да гарантираме условието последната цифра **d3** да бъде **по-голяма** от първите две,

ще използваме вградената функция **max(...)**, която се намира в библиотеката **algorithm**. За водещите **променливите** на циклите, които ще **съхраняват символите** на паролата, се спират на следните типове: за **цифровите** символи (**d1, d2 и d3**) - тип **int**, а за **буквените** (**l1 и l2**) - тип **char**:

```
for (int d1 = 1; d1 < n; d1++) {
    for (int d2 = 1; d2 < n; d2++) {
        for (char l1 = 'a'; l1 < 'a' + 1; l1++) {
            for (char l2 = 'a'; l2 < ('a' + 1); l2++) {
                for (int d3 = max(d1, d2) + 1; d3 <= n; d3++) {
                    cout << d1 << d2 << l1 << l2 << d3 << " ";
                }
            }
        }
    }
}
```

Знаете ли, че...?

- Можем да дефинираме **for** цикъл с променлива от тип **char**:

```
for (char ch = 'a'; ch <= 'z'; ch++)
```

- Можем да прочетем променлива от тип **char** от конзолата със следната конструкция:

```
char ch;
cin >> ch;
```

- Можем да обърнем главен символ към малък, използвайки вградена функция в C++:

```
ch = tolower(ch);
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1369#0>.

Задача: магически числа

Да се напише програма, която въвежда едно цяло **магическо** число и изкарва всички възможни **6-цифрени** числа, за които **произведението** на техните цифри е **равно** на **магическото** число.

Пример: "Магическо число" → 2

- 111112 → $1 * 1 * 1 * 1 * 1 * 2 = 2$
- 111121 → $1 * 1 * 1 * 1 * 2 * 1 = 2$

- $111211 \rightarrow 1 * 1 * 1 * 2 * 1 * 1 = 2$
- $112111 \rightarrow 1 * 1 * 2 * 1 * 1 * 1 = 2$
- $121111 \rightarrow 1 * 2 * 1 * 1 * 1 * 1 = 2$
- $211111 \rightarrow 2 * 1 * 1 * 1 * 1 * 1 = 2$

Входни данни

Входът се чете от конзолата и се състои от **едно цяло число** в интервала [1 ... 600 000].

Изходни данни

На конзолата трябва да се отпечатат **всички магически числа**, разделени с интервал.

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
			111118 111124 111142 111181 111214 111222 111241 111412 111421 111811 112114 112122 112141 112212 112221 112411 114112 114121 114211 118111 121114 121122 121141 121212 121221 121411 122112 122121 122211 124111 141112 141121 141211 142111 181111 211114 211122 211141 211212 211221 211411 212112 212121 212211 214111 221112 221121 221211 222111 241111 411112 411121 411211 412111 421111 811111		
2	111112	8		5	
	111121			3	
	111211			1	999999
	112111			4	
	121111			4	
	211111			1	

Насоки и подсказки

Решението следва **същата** концепция като в предходната задача - "Генератор за тъпли пароли" (отново трябва да генерираме всички комбинации за n елемента). Следвайки следните стъпки, опитайте да решите задачата сами:

- Декларирайте и инициализирайте **променлива** от целочислен тип **int** и прочетете **входа** от конзолата - магическото число.
- Вложете **шест for цикъла** един в друг, по един за всяка цифра на търсените 6-цифрени числа.
- В последния цикъл, чрез **if** конструкция проверете дали **произведението** на шестте цифри е **равно** на **магическото** число.

Ето примерна имплементация на идеята:

```
int magicNumber;

for (int d1 = 0; d1 <= 9; d1++) {
    for (int d2 = 0; d2 <= 9; d2++) {
        for (int d3 = 0; d3 <= 9; d3++) {
            for (int d4 = 0; d4 <= 9; d4++) {
                for (int d5 = 0; d5 <= 9; d5++) {
                    for (int d6 = 0; d6 <= 9; d6++) {
                        if (d1 * d2 * d3 * d4 * d5 * d6 == magicNumber) {
                            // ...
                        }
                    }
                }
            }
        }
    }
}
```

В предходната глава разглеждахме и други циклични конструкции. Нека разгледаме примерно решение на същата задача, в което използваме цикъла **while**. Първо трябва да запишем **входното магическо число** в подходяща променлива и да инициализираме 6 променливи - по една за всяка от шестте цифри на търсените като **результат** числа:

```
int magicNumber;
cin >> magicNumber;

int d1, d2, d3, d4, d5, d6;
```

След това ще започнем да разписваме **while** циклите:

- Ще инициализираме **първата цифра**: **d1 = 1**.
- Ще зададем **условие за всеки цикъл**: цифрата да бъде по-малка или равна на 9.
- В началото на всеки цикъл задаваме стойност на **следващата цифра**, в случая: **d2 = 1**. При вложените **for** цикли инициализираме променливите във вътрешните цикли при всяко увеличение на външните. Искаме да постигнем същото поведение и тук.
- В **край** на всеки цикъл ще **увеличаваме** съответната цифра с едно: **d1 += 1**, **d2 += 1**, **d3 += 1** и т.н.
- В **най-вътрешния** цикъл ще направим **проверката** и ако е необходимо, ще принтираме на конзолата.

```

d1 = 1;
while (d1 <= 9) {
    d2 = 1;
    while (d2 <= 9) {
        d3 = 1;
        while (d3 <= 9) {
            ...
            ...
            ...
            if (d1 * d2 * d3 * d4 * d5 * d6 == magicNumber) {
                cout << d1 << d2 << d3 << d4 << d5 << d6 << " ";
            }
        }
    }
}
d3++;
}
d2++;
}
d1++;
}
}

```

Както виждаме, един проблем може да бъде решен с различни видове цикли. Разбира се, за всяка задача има най-подходящ избор. С цел да упражните всеки цикъл - опитайте се да решите всяка от следващите задачи с всички изучени цикли.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1369#1>.

Задача: спиращо число

Напишете програма, която принтира на конзолата всички числа от **N** до **M**, които се делят на 2 и на 3 без остатък, в обратен ред. От конзолата ще се чете още едно "спиращо" число **S**. Ако някое от делящите се на 2 и 3 числа е равно на спиращото число, то не трябва да се принтира и програмата трябва да приключи. В противен случай се принтират всички числа до **N**, които отговарят на условието.

Вход

От конзолата се четат 3 числа, всяко на отделен ред:

- N - цяло число: $0 \leq N < M$.
- M - цяло число: $N < M \leq 10000$.
- S - цяло число: $N \leq S \leq M$.

Изход

На конзолата се принтират на един ред, разделени с интервал, всички числа, отговарящи на условията.

Примерен вход и изход

Вход	Изход	Обяснения
1 30 15	30 24 18 12 6	Числата от 30 до 1, които се делят едновременно на 2 и на 3 без остатък са: 30, 24, 18, 12 и 6. Числото 15 не е равно на нито едно, затова редицата продължава .
Вход	Изход	Обяснения
1 36 12	36 30 24 18	Числата от 36 до 1, които се делят едновременно на 2 и на 3 без остатък, са: 36, 30, 24, 18, 12 и 6. Числото 12 е равно на спиращото число, затова спираме до 18 .
Вход	Изход	
20 1000 36	996 990 984 978 972 966 960 954 948 942 936 930 924 918 912 906 900 894 888 882 876 870 864 858 852 846 840 834 828 822 816 810 804 798 792 786 780 774 768 762 756 750 744 738 732 726 720 714 708 702 696 690 684 678 672 666 660 654 648 642 636 630 624 618 612 606 600 594 588 582 576 570 564 558 552 546 540 534 528 522 516 510 504 498 492 486 480 474 468 462 456 450 444 438 432 426 420 414 408 402 396 390 384 378 372 366 360 354 348 342 336 330 324 318 312 306 300 294 288 282 276 270 264 258 252 246 240 234 228 222 216 210 204 198 192 186 180 174 168 162 156 150 144 138 132 126 120 114 108 102 96 90 84 78 72 66 60 54 48 42	

Насоки и подсказки

Задачата може да се раздели на **четири** логически части:

- Прочитане на входните данни от конзолата.
- Проверка на всички числа в дадения интервал (съответно чрез завъртане на цикъл).
- Проверка на условията от задачата спрямо всяко едно число от въпросния интервал.

- Отпечатване на числата.

Първата част е тривиална - прочитаме **три** цели числа от конзолата, съответно ще използваме тип **int**.

С втората част също сме се сблъсквали - инициализиране на **for** цикъл. Тук има малка **уловка** - в условието е споменато, че числата трябва да се принтират в обратен ред. Това означава, че **началната** стойност на променливата **i** ще е **поголямото число**, което от примерите виждаме, че е **M**. Съответно, **крайната** стойност на **i** трябва да е **N**. Фактът, че ще печатаме резултатите в обратен ред и стойностите на **i** ни подсказват, че стъпката ще е **намаляваща с 1**:

```
for (int i = m; i >= n; i--)
```

След като сме инициализирали **for** цикъла, идва ред на **третата** част от задачата - **проверка** на условието дали даденото **число се дели на 2 и на 3 без остатък**. Това ще направим с една обикновена **if** проверка, която ще оставим на читателя сам да състави.

Другата **уловка** в тази задача е, че освен горната проверка, трябва да направим **още** една - дали **числото е равно на "спиращото" число**, подадено ни от конзолата на третия ред. За да се стигне до тази проверка, числото, което проверяваме, трябва да премине през горната. По тази причина ще построим **още** една **if** конструкция, която ще **вложим в предходната**. Ако условието е **вярно**, заданието е да спрем програмата, което в конкретния случай можем да направим с оператор **break**, който ще ни **изведе от for** цикъла.

Съответно, ако **условието** на проверката, дали числото съвпада със "спиращото" число, върне резултат **false**, по задание нашата програма трябва да **продължи да печата**. Това всъщност покрива и **четвъртата и последна** част от нашата програма.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1369#2>.

Задача: специални числа

Да се напише програма, която **въвежда едно цяло число N** и генерира всички възможни "специални" числа от **1111** до **9999**. За да бъде "специално" едно число, то трябва да отговаря на **условието** - **N да се дели на всяка една от неговите цифри без остатък**.

Пример: при **N = 16,2418** е специално число:

- $16 / 2 = 8$ **без остатък**
- $16 / 4 = 4$ **без остатък**
- $16 / 1 = 16$ **без остатък**
- $16 / 8 = 2$ **без остатък**

Входни данни

Входът се чете от конзолата и се състои от **едно цяло число** в интервала [1 ... 600 000].

Изходни данни

Да се отпечатат на конзолата **всички специални числа**, разделени с **интервал**.

Примерен вход и изход

Вход	Изход		Коментари
3	1111 1113 1131 1133 1311 1313 1331 1333 3111 3113 3131 3133 3311 3313 3331 3333		3 / 1 = 3 без остатък 3 / 3 = 1 без остатък 3 / 3 = 1 без остатък 3 / 3 = 1 без остатък
Вход	Изход	Вход	Изход
11	1111	16	1111 1112 1114 1118 1121 1122 1124 1128 1141 1142 1144 1148 1181 1182 1184 1188 1211 1212 1214 1218 1221 1222 1224 1228 1241 1242 1244 1248 1281 1282 1284 1288 1411 1412 1414 1418 1421 1422 1424 1428 1441 1442 1444 1448 1481 1482 1484 1488 1811 1812 1814 1818 1821 1822 1824 1828 1841 1842 1844 1848 1881 1882 1884 1888 2111 2112 2114 2118 2121 2122 2124 2128 2141 2142 2144 2148 2181 2182 2184 2188 2211 2212 2214 2218 2221 2222 2224 2228 2241 2242 2244 2248 2281 2282 2284 2288 2411 2412 2414 2418 2421 2422 2424 2428 2441 2442 2444 2448 2481 2482 2484 2488 2811 2812 2814 2818 2821 2822 2824 2828 2841 2842 2844 2848 2881 2882 2884 2888 4111 4112 4114 4118 4121 4122 4124 4128 4141 4142 4144 4148 4181 4182 4184 4188 4211 4212 4214 4218 4221 4222 4224 4228 4241 4242 4244 4248 4281 4282 4284 4288 4411 4412 4414 4418 4421 4422 4424 4428 4441 4442 4444 4448 4481 4482 4484 4488 4811 4812 4814 4818 4821 4822 4824 4828 4841 4842 4844 4848 4881 4882 4884 4888 8111 8112 8114 8118 8121 8122

Вход	Изход	Вход	Изход
			8124 8128 8141 8142 8144 8148 8181 8182 8184 8188 8211 8212 8214 8218 8221 8222 8224 8228 8241 8242 8244 8248 8281 8282 8284 8288 8411 8412 8414 8418 8421 8422 8424 8428 8441 8442 8444 8448 8481 8482 8484 8488 8811 8812 8814 8818 8821 8822 8824 8828 8841 8842 8844 8848 8881 8882 8884 8888

Насоки и подсказки

Решете задачата самостоятелно, използвайки наученото от предходните. Спомнете си разликата между операторите за **целочислено деление (/)** и **деление с остатък (%)** в C++.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1369#3>.

Задача: цифри

Да се напише програма, която прочита от конзолата 1 цяло число в интервала [100 ... 999], и след това го принтира определен брой пъти - модифицирайки го преди всяко принтиране по следния начин:

- Ако числото се дели на **5** без остатък, **извадете** от него **първата** **му** **цифра**.
- Ако числото се дели на **3** без остатък, **извадете** от него **втората** **му** **цифра**.
- Ако нито едно от горните условия не е вярно, **прибавете** към него **третата** **му** **цифра**.

Принтирайте на конзолата **N** **брой реда**, като всеки ред има **M** **на** **брой** **числа**, които са резултат от горните действия. Нека:

- **N** = **сбора** на **първата** и **втората** **цифра** на **числото**.
- **M** = **сбора** на **първата** и **третата** **цифра** на **числото**.

Входни данни

Входът се чете от конзолата и е цяло число в интервала [100 ... 999].

Изходни данни

На конзолата трябва да се отпечатат **всички цели числа**, които са резултат от дадените по-горе изчисления в съответния брой редове и колони, както в примерите по-долу.

Примерен вход и изход

Вход	Изход	Коментари
132	129 126 123 120 119 121 123 120 119 121 123 120	$(1 + 3) = 4$ и $(1 + 2) = 3 \rightarrow 4$ реда по 3 числа на всеки ред Входното число 132 $132 \rightarrow$ деление на 3 $\rightarrow 132 - 3 =$ $= 129 \rightarrow$ деление на 3 $\rightarrow 129 - 3 =$ $= 126 \rightarrow$ деление на 3 $\rightarrow 126 - 3 =$ $= 123 \rightarrow$ деление на 3 $\rightarrow 123 - 3 =$ $= 120 \rightarrow$ деление на 5 $\rightarrow 120 - 1 =$ 121 \rightarrow нито на 5, нито на 3 $\rightarrow 121 + 2 = 123$
376	382 388 394 400 397 403 409 415 412 418 424 430 427 433 439 445 442 448 454 460 457 463 469 475 472 478 484 490 487 493 499 505 502 508 514 520 517 523 529 535 532 538 544 550 547 553 559 565 562 568 574 580 577 583 589 595 592 598 604 610 607 613 619 625 622 628 634 640 637 643 649 655 652 658 664 670 667 673 679 685 682 688 694 700 697 703 709 715 712 718	10 реда по 9 числа на всеки Входното число 376 \rightarrow нито на 5, нито на 3 \rightarrow $376 + 6 \rightarrow =$ $= 382 \rightarrow$ нито на 5, нито на 3 $\rightarrow 382 + 6 =$ $= 388 + 6 = 394 + 6 =$ $400 \rightarrow$ деление на 5 \rightarrow $400 - 3 = 397$

Насоки и подсказки

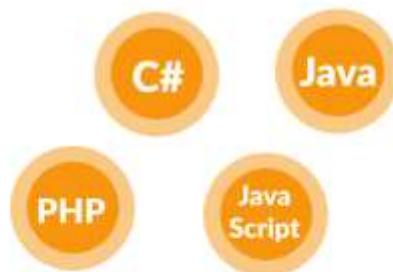
Решете задачата **самостоятелно**, използвайки наученото в предходните. Не забравяйте, че ще е нужно да дефинирате **отделна** променлива за всяка цифра на входното число.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1369#4>.

Качествено образование,
професия и работа за
Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



СофТУни предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата **"Софтуерен университет"** изгражда истински **профессионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на СофТУни, за да усвоите **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

СофТУни работи пряко с компаниите от софтуерната индустрия, съдейтайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ



1 - 1.5 години



1.5 - 2 години



Programming Basics

Кариерен Старт

Дипломиране

70/100 credits

80/100/150 credits

Кандидатствай

softuni.bg/apply

Глава 8.1. Подготовка за практически изпит – част I

В настоящата глава ще разгледаме няколко **задачи** с ниво на **трудност**, каквото може да очаква от **задачите** на практическия **изпит** по "Основи на програмирането" в СофтУни. Ще **преговорим** и **упражним** всички знания, които сме придобили от настоящата книга и през курса "Programming Basics".

Видео

Гледайте видео урок по учебния материал от настоящата глава от книгата: <https://www.youtube.com/watch?v=YtvGjXzSgfE>.

Практически изпит по "Основи на програмирането"

Курсът "Programming Basics" приключва с **практически изпит**. Включени са **6** задачи, като ще имате на разположение **4 часа**, за да ги решите. **Всяка** от задачите на изпита ще **засяга** една от изучаваните **теми** по време на курса. Темите на задачите са както следва:

- Задача с прости сметки (без проверки).
- Задача с единична проверка.
- Задача с по-сложни проверки.
- Задача с единичен цикъл.
- Задача с вложени цикли (чертане на фигурка на конзолата).
- Задача с вложени цикли и по-сложна логика.

Система за онлайн оценяване (Judge)

Всички изпити и домашни се **тестват** автоматизирано през онлайн **Judge** система: <https://judge.softuni.bg>. За **всяка** от задачите има **открити** (нулеви) тестове, които ще ви помогнат да разберете какво се очаква от задачата и да поправите грешките си, както и **състезателни** тестове, които са **скрити** и проверяват дали задачата ви работи правилно. В **Judge** системата се влиза с вашия **softuni.bg** акаунт.

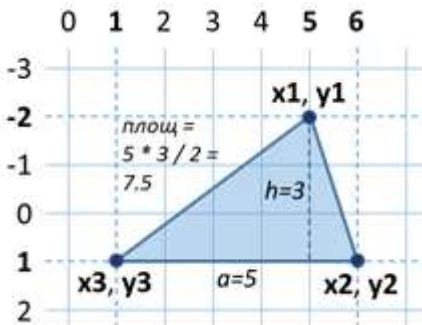
Как работи тестването в **Judge** системата? **Качвате** сорс кода и от менюто под него избирате да се компилира като **C++** програма. Програмата бива **тествана** с поредица от тестове, като за всеки **успешен** тест получавате **точки**.

Задачи с прости пресмятания

Първата задача на практическия изпит по "Основи на програмирането" обхваща **прости пресмятания без проверки и цикли**. Ето няколко примера:

Задача: лице на триъгълник в равнината

Триъгълник в равнината е зададен чрез координатите на трите си върха. Първо е зададен върхът (x_1, y_1) . След това са зададени останалите два върха: (x_2, y_2) и (x_3, y_3) , които лежат на обща хоризонтална права (т.е. имат еднакви Y координати). Напишете програма, която пресмята лицето на триъгълника по координатите на трите му върха.



Вход

От конзолата се четат 6 цели числа (по едно на ред): $x_1, y_1, x_2, y_2, x_3, y_3$.

- Всички входни числа са в диапазона $[-1000 \dots 1000]$.
- Гарантирано е, че $y_2 = y_3$.

Изход

Да се отпечата на конзолата лицето на триъгълника.

Примерен вход и изход

Вход	Изход	Чертеж	Обяснения
5 -2 6 1 1 1	7.5		Страната на триъгълника: $a = 6 - 1 = 5$ Височината на триъгълника: $h = 1 - (-2) = 3$ Лицето на триъгълника: $S = a * h / 2 = 5 * 3 / 2 = 7.5$
4 1 -1 -3 3 -3	8		Страната на триъгълника: $a = 3 - (-1) = 4$ Височината на триъгълника: $h = 1 - (-3) = 4$ Лицето на триъгълника: $S = a * h / 2 = 4 * 4 / 2 = 8$

Насоки и подсказки

Изключително важно при подобен тип задачи, при които се подават някакви координати, е да обърнем внимание на **реда**, в който се подават, както и правилно да осмислим кои от координатите ще използваме и по какъв начин. В случая, на входа се подават **$x_1, y_1, x_2, y_2, x_3, y_3$** в този си ред. Ако не спазваме тази последователност, решението става грешно. Първо пишем кода, който чете подадените данни:

```
int x1, y1, x2, y2, x3, y3;
cin >> x1 >> y1 >> x2 >> y2 >> x3 >> y3;
```

Трябва да пресметнем страната и височината на триъгълника. От картинките, както и от условието **$y_2 = y_3$** забелязваме, че едната **страна** винаги е успоредна на хоризонталната ос. Това означава, че нейната **дължина** е равна на дълчината на отсечката между нейните координати **x_2 и x_3** , която е равна на разликата между по-голямата и по-малката координата. Аналогично можем да изчислим и **височината**. Тя винаги ще е равна на разликата между **y_1 и y_2** (или **y_3** , тъй като са равни). Тъй като не знаем дали винаги **x_2** ще е по-голям от **x_3** , или **y_1** ще е под или над страната на триъгълника, ще използваме **абсолютните стойности** на разликата, за да получаваме винаги положителни числа, понеже една отсечка не може да има отрицателна дължина. Налага се и да включим библиотеката **cmath** в началото на програмата, за да имаме достъп до функцията **fabs(...)**:

```
int a = fabs(x2 - x3);
int h = fabs(y2 - y1);
```

По познатата ни от училище формула за намиране на **лице на триъгълник** ще пресметнем лицето. Важно нещо, което трябва да съобразим, е въпреки че на входа получаваме само цели числа, **лицето** не винаги ще е цяло число. Затова за лицето използваме променлива от тип **double**. Налага се да конвертираме и дясната страна на уравнението, понеже ако подадем цели числа като параметри на уравнението, резултатът ни също ще е цяло число:

```
double s = (double)h*a / 2;
```

Единственото, което остава, е да отпечатаме лицето на конзолата:

```
cout << s << endl;
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1370#0>.

Задача: пренасяне на тухли

Строителни работници трябва да пренесат общо **x тухли**. **Работниците** са **w** на брой и работят едновременно. Те превозват тухлите в колички, всяка с **вместимост m** тухли. Напишете програма, която прочита целите числа **x , w и m** и

пресмята **колко най-малко курса** трябва да направят работниците, за да превозят тухлите.

Вход

От конзолата се четат **3 цели числа** (по едно на ред):

- **Броят тухли x** се чете от първия ред.
- **Броят работници w** се чете от втория ред.
- **Вместимостта на количката m** се чете от третия ред.

Всички входни числа са цели и в диапазона [1 ... 1000].

Изход

Да се отпечата на конзолата **минималният брой курсове**, необходими за превозване на тухлите.

Примерен вход и изход

Вход	Изход	Обяснения
120 2 30	2	Имаме 2 работника, всеки вози по 30 тухли на курс. Общо работниците возят по 60 тухли на курс. За да превозят 120 тухли, са необходими точно 2 курса.
Вход	Изход	Обяснения
355 3 10	12	Имаме 3 работника, всеки вози по 10 тухли на курс. Общо работниците возят по 30 тухли на курс. За да превозят 355 тухли, са необходими точно 12 курса: 11 пълни курса превозват 330 тухли и последният 12-ти курс пренася последните 25 тухли.
Вход	Изход	Обяснения
5 12 30	1	Имаме 5 работника, всеки вози по 30 тухли на курс. Общо работниците возят по 150 тухли на курс. За да превозят 5 тухли, е достатъчен само 1 курс (макар и непълен, само с 5 тухли).

Насоки и подсказки

Входът е стандартен, като единствено трябва да внимаваме за последователността, в която прочитаме данните:

```
int x, w, m;
cin >> x >> w >> m;
```

Пресмятаме колко **тухли** носят работниците на един курс:

```
int bricksInOneCourse = w * m;
```

Като разделим общия брой на тухлите на броя на **тухлите, пренесени за 1 курс**, ще получим броя **курсове**, необходими за пренасянето им. Трябва да съобразим, че при деление на цели числа се пренебрегва остатъка и се закръгля винаги надолу. За да избегнем това ще конвертираме дясната страна на уравнението към **double** и ще използваме функцията **ceil(...)** от библиотеката **cmath**, за да закръглим получения резултат винаги нагоре. Когато тухлите могат да се пренесат с **точен брой курсове**, делението ще връща точно число и няма да има нищо за закръгляне. Съответно, когато не е така, резултатът от делението ще е **броя на точните курсове**, но с десетична част. Десетичната част ще се закръгли нагоре и така ще се получи нужният **1 курс** за оставащите тухли:

```
double totalCourses = ceil((double)x / bricksInOneCourse);
```

Накрая извеждаме резултата на конзолата.

```
cout << totalCourses << endl;
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1370#1>.

Задачи с единична проверка

Втората задача от практическия изпит по "Основи на програмирането" обхваща **условна конструкция и прости пресмятания**. Ето няколко примера:

Задача: точка върху отсечка

Върху хоризонтална права е разположена **хоризонтална отсечка**, зададена с **x** координатите на двата си края: **first** и **second**. **Точка** е разположена **върху** същата хоризонтална права и е зададена с **x** координатата си. Напишете програма, която проверява дали точката е **вътре или вън** от отсечката и изчислява **разстоянието до по-близкия край** на отсечката.

Вход

От конзолата се четат **3 цели числа** (по едно на ред):

- На първия ред стои числото **first** – **единия край на отсечката**.
- На втория ред стои числото **second** – **другия край на отсечката**.
- На третия ред стои числото **point** – **местоположението на точката**.

Всички входни числа са цели и в диапазона [-1000 ... 1000].

Изход

Резултатът да се отпечата на конзолата по следния начин:

- На първия ред да се отпечата "in" или "out" – дали точката е върху отсечката или извън нея.
- На втория ред да се отпечата разстоянието от точката до най-близкия край на отсечката.

Примерен вход и изход

Вход	Изход	Визуализация
10 5 7	in 2	
8 10 5	out 3	
1 -2 3	out 2	

Насоки и подсказки

Първо прочитаме входните данни от конзолата:

```
int first, second, point;
cin >> first >> second >> point;
```

Тъй като не знаем коя **точка** е от ляво и коя е от дясно, ще си направим две променливи, които да ни отбелнязват това. Тъй като **лявата точка** е винаги тази с по-малката **x координата**, ще ползваме функцията **min(...)**, за да я намерим. За да използваме тази функция ще се наложи да включим библиотеката **algorithm** в началото на програмата. Съответно, **дясната** е винаги тази с по-голяма **x координата** и за намирането ѝ ще ползваме функцията **max(...)**. Ще намерим и разстоянието от **точката x** до **двете точки**. Понеже не знаем положението им една спрямо друга, ще използваме **fabs(...)** от библиотеката **cmath**, за да получим положителен резултат:

```
int left = min(first, second);
int right = max(first, second);

int distanceLeft = fabs(left - point);
int distanceRight = fabs(right - point);
```

По-малкото от двете **разстояния** ще намерим ползвайки **min(...)**:

```
int minDistance = min(distanceLeft, distanceRight);
```

Остава да намерим дали **точката** е на линията или извън нея. Точката ще се намира на **линията** винаги, когато тя **съвпада** с някоя от другите две точки или х координатата ѝ се намира **между тях**. В противен случай, точката се намира **извън линията**. След проверката изкарваме едното от двете съобщения ("in" или "out"), спрямо това кое условие е удовлетворено:

```
if (point >= left && point <= right) {
    cout << "in" << endl;
}
else {
    cout << "out" << endl;
}
```

Накрая извеждаме **разстоянието**, намерено преди това.

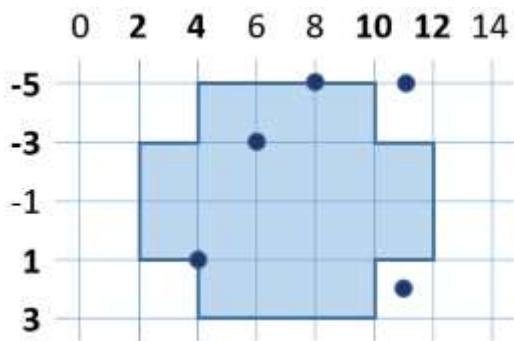
```
cout << minDistance << endl;
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1370#2>.

Задача: точка във фигура

Да се напише програма, която проверява дали дадена точка (с координати **x** и **y**) е **вътре** или **извън** следната фигура:



Вход

От конзолата се четат **две цели числа** (по едно на ред): **x** и **y**.

Всички входни числа са цели и в диапазона [-1000 ... 1000].

Изход

Да се отпечата на конзолата "in" или "out" – дали точката е **вътре** или **извън** фигурата. Ако точката е на контура, приемаме, че е вътре.

Примерен вход и изход

Вход	Изход
8 -5	in

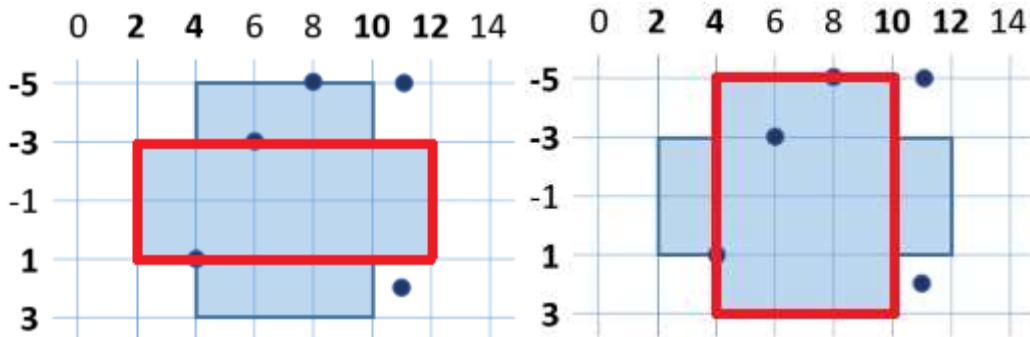
Вход	Изход
6 -3	in

Вход	Изход
11 -5	out

Вход	Изход
11 2	out

Насоки и подсказки

За да разберем дали **точката** е във фигурата, ще разделим **фигурата** на 2 четириъгълника. Достатъчно условие е **точката** да се намира в един от тях, за да бъде във **фигурата**:



Четем от конзолата входните данни:

```
int x, y;
cin >> x >> y;
```

Ще създадем две променливи, които ще отбелоязват дали точката се намира в някой от правоъгълниците:

```
bool pointInRect1 = x >= 2 && x <= 12 && y >= -3 && y <= 1;
bool pointInRect2 = x >= 4 && x <= 10 && y >= -5 && y <= 3;
```

При отпечатването на съобщението ще проверим дали някоя от променливите е приела стойност **true**. Достатъчно е **само една** от тях да е **true**, за да се намира точката във фигурата:

```
if (pointInRect1 || pointInRect2) {
    cout << "in" << endl;
}
else {
```

```

    cout << "out" << endl;
}

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1370#3>.

Задачи с по-сложни проверки

Третата задача от практическия изпит по “Основи на програмирането” включва няколко вложени проверки съчетани с прости пресмятания. Ето няколко примера:

Задача: дата след 5 дни

Дадени са две числа d (ден) и m (месец), които формират **дата**. Да се напише програма, която отпечатва датата, която ще бъде **след 5 дни**. Например 5 дни след 28.03 е датата 2.04. Приемаме, че месеците: април, юни, септември и ноември имат по 30 дни, февруари има 28 дни, а останалите имат по 31 дни. Месеците да се отпечатат с **водеща нула**, когато са едноцифриeni (например 01, 08).

Вход

Входът се чете от конзолата и се състои от два реда:

- На първия ред стои едно цяло число d в интервала [1 ... 31] – ден. Номерът на деня не надвишава броя дни в съответния месец (напр. 28 за февруари).
- На втория ред стои едно цяло число m в интервала [1 ... 12] – месец. Месец 1 е януари, месец 2 е февруари, ..., месец 12 е декември. Месецът може да съдържа водеща нула (напр. април може да бъде изписан като 4 или 04).

Изход

Отпечатайте на конзолата един единствен ред, съдържащ дата след 5 дни във формат **day.month**. Месецът трябва да бъде двуцифрене число с водеща нула, ако е необходимо. Денят трябва да е без водеща нула.

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
28 03	2.04	27 12	1.01	25 1	30.01	26 02	3.03

Насоки и подсказки

Прочитаме входните данни от конзолата:

```
int d, m;
cin >> d >> m;
```

За да си направим по-лесно проверките, ще си създадем една променлива, която ще съдържа **броя дни**, които има в месеца, който сме задали:

```
int daysInMonth = 31;
if (m == 2) {
    daysInMonth = 28;
}

if (m == 4 || m == 6 || m == 9 || m == 11) {
    daysInMonth = 30;
}
```

Увеличаваме **дения** с 5:

```
d += 5;
```

Проверяваме дали **денят** не е станал по-голям от броя дни, които има в съответния **месец**. Ако това е така, трябва да извадим дните от месеца от получения ден, за да получим нашият ден на кой ден от следващия месец съответства:

```
if (d > daysInMonth) {
    d -= daysInMonth;
}
```

След като сме минали в **следващия месец**, това трябва да се отбележи, като увеличим първоначално зададения месец с 1. Трябва да проверим, дали той не е станал по-голям от 12 и ако е така, да коригираме. Тъй като няма как да прескочим повече от **един месец**, когато увеличаваме с 5 дни, следната проверка е достатъчна:

```
if (d > daysInMonth) {
    d -= daysInMonth;
    m++;

    if (m > 12) {
        m = 1;
    }
}
```

Остава само да изведем резултата на конзолата. Важно е да форматираме изхода правилно, за да се появява водещата нула в първите 9 месеца:

```

if (m <= 9) {
    cout << d << ".0" << m << endl;
}
else {
    cout << d << "." << m << endl;
}

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1370#4>.

Задача: суми от 3 числа

Дадени са **3 цели числа**. Да се напише програма, която проверява дали **сумата на две от числата е равна на третото**. Например, ако числата са **3, 5 и 2**, сумата на две от числата е равна на третото: $2 + 3 = 5$.

Вход

От конзолата се четат **три цели числа**, по едно на ред. Числата са в диапазона [1 ... 1000].

Изход

- Да се отпечата на конзолата един ред, съдържащ решението на задачата във формат " $a + b = c$ ", където a , b и c са измежду входните три числа и $a \leq b$.
- Ако задачата няма решение, да се отпечата "No" на конзолата

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
3		2		1		2	
5	$2 + 3 = 5$	2	$2 + 2 = 4$	1	No	6	
2		4		5		3	No

Насоки и подсказки

Приемаме входа от конзолата:

```

int a, b, c;
cin >> a >> b >> c;

```

Трябва да проверим дали **сумата** на някоя двойка числа е равна на третото. Имаме три възможни случая:

- $a + b = c$

- $a + c = b$
- $b + c = a$

Ще напишем **рамка**, която после ще допълним с нужния код. Ако никое от горните три условия не е изпълнено, ще зададем на програмата да принтира "No":

```
if (a + b == c) {
    // TODO
}
else if (a + c == b) {
    // TODO
} else if (b + c == a) {
    // TODO
}
else {
    cout << "No" << endl;
}
```

Сега остава да разберем реда, в който ще се изписват **двете събираеми** на изхода на програмата. За целта ще направим **вложено условие**, което проверява кое от двете числа е по-голямото. При първия случай, ще стане по следния начин:

```
if (a + b == c) {
    if (a > b) {
        cout << b << " + " << a << " = " << c << endl;
    }
}
```

Аналогично, ще допълним и другите два случая. Пълният код на проверките и изходът на програмата ще изглеждат така:

```
if (a + b == c) {
    if (b < a) {
        cout << b << " + " << a << " = " << c << endl;
    }
    else {
        cout << a << " + " << b << " = " << c << endl;
    }
}
```

```

else if (a + c == b) {
    if (a < c) {
        cout << a << " + " << c << " = " << b << endl;
    }
    else {
        cout << c << " + " << a << " = " << b << endl;
    }
} else if (b + c == a) {
    if (b < c) {
        cout << b << " + " << c << " = " << a << endl;
    }
    else {
        cout << c << " + " << b << " = " << a << endl;
    }
}
else {
    cout << "No" << endl;
}

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1370#5>.

Задачи с единичен цикъл

Четвъртата задача от практическия изпит по “Основи на програмирането” включва **единичен цикъл с пристап към паметта**. Ето няколко примера:

Задача: суми през 3

Дадени са n цели числа a_1, a_2, \dots, a_n . Да се пресметнат сумите:

- $sum1 = a_1 + a_4 + a_7 + \dots$ (сумират се числата, започвайки от първото със стъпка 3).
- $sum2 = a_2 + a_5 + a_8 + \dots$ (сумират се числата, започвайки от второто със стъпка 3).
- $sum3 = a_3 + a_6 + a_9 + \dots$ (сумират се числата, започвайки от третото със стъпка 3).

Вход

Входните данни се четат от конзолата. На първия ред стои цяло число n ($0 \leq n \leq$

1000). На следващите n реда стоят n цели числа в интервала [-1000 ... 1000]: a_1, a_2, \dots, a_n .

Изход

На конзолата трябва да се отпечатат 3 реда, съдържащи търсените 3 суми, във формат като в примерите.

Примерен вход и изход

Вход	Изход
2	sum1 = 3
3	sum2 = 5
5	sum3 = 0

Вход	Изход
4	
7	sum1 = 19
-2	sum2 = -2
6	sum3 = 6
12	

Вход	Изход
5	sum1 =
3	10
5	sum2 =
2	13
7	sum3 =
8	2

Насоки и подсказки

Ще вземем броя на числата от конзолата и ще декларираме **начални стойности** на трите суми:

```
int n;
cin >> n;

int sum1 = 0;
int sum2 = 0;
int sum3 = 0;
```

Тъй като не знаем предварително колко числа ще обработваме, ще ги прочитаме едно по едно в **цикъл**, който ще се повтори n на брой пъти и ще ги обработваме в тялото на цикъла:

```
for (int i = 0; i < n; i++) {
    int a;
    cin >> a;
    // TODO
}
```

За да разберем в коя от **трите суми** трябва да добавим числото, ще разделим **поредния му номер на три** и ще използваме **остатъка**. Ще използваме променливата **i**, която следи **броя завъртания** на цикъла, за да разберем на кое поред число сме. Когато резултатът от **i % 3** е **нула**, това означава, че ще

добавяме това число към първата сума, когато е **1** към **втората** и съответно, когато е **2** към **третата**:

```
for (int i = 0; i < n; i++) {
    int a;
    cin >> a;

    if (i % 3 == 0) {
        sum1 += a;
    }
    else if (i % 3 == 1) {
        sum2 += a;
    }
    else if (i % 3 == 2) {
        sum3 += a;
    }
}
```

Накрая, ще отпечатаме резултата на конзолата в изисквания от условието формат:

```
cout << "sum1 = " << sum1 << endl;
cout << "sum2 = " << sum2 << endl;
cout << "sum3 = " << sum3 << endl;
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1370#6>.

Задача: поредица от нарастващи елементи

Дадена е редица от **n** числа: a_1, a_2, \dots, a_n . Да се пресметне **дължината на най-дългата нарастваща поредица** от последователни елементи в редицата от числа.

Вход

Входните данни се четат от конзолата. На първия ред стои цяло число **n** ($0 \leq n \leq 1000$). На следващите **n** реда стоят **n** цели числа в интервала $[-1000 \dots 1000]$: a_1, a_2, \dots, a_n .

Изход

На конзолата трябва да се отпечата едно число – **дължината на най-дългата нарастваща редица**.

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
3		4		4		4	
5		2		1		5	
2	2	8	2	2	3	6	4
4		7		4		7	
		6		4		8	

Насоки и подсказки

За решението на тази задача трябва да помислим малко **по-алгоритично**. Дадена ни е **редица от числа** и трябва да проверяваме дали всяко **следващо** число е **по-голямо от предходното** и ако е така да броим колко дълга е тази подредица, в която това условие е изпълнено. След това трябва да намерим **коя подредица** е **най-дълга**. За целта, нека да си направим няколко променливи, които ще ползваме през хода на задачата.

Забележка: Тъй като променливите са от един и същи тип (**int**), бихме могли да ги декларираме и инициализираме на един ред, като ги отделяме със запетая.

```
int n;
cin >> n;
int countCurrentLongest = 0;
int countLongest = 0;
int aPrev = 0;
int a = 0;
```

Променливата **n** е **броя** **числа**, които ще получим от конзолата. В **countCurrentLongest** ще запазваме **броя на елементите** в **нарастваща** **подредица**, която **броим в момента**. Напр. при редицата: 5, 6, 1, 2, 3 **countCurrentLongest** ще бъде 2, когато сме стигнали **втория елемент** от броенето (5, 6, 1, 2, 3) и ще стане 3, когато стигнем **последния елемент** (5, 6, 1, 2, 3), понеже **нарастваща** **редица** 1, 2, 3 има 3 елемента. Ще използваме **countLongest**, за да запазим **най-дългата** **нарастваща** **редица**. Останалите променливи, който ще използваме, са **a** - **числото, на което се намираме в момента**, и **aPrev** - **предишното число**, което ще сравним с **a**, за да разберем дали **редицата расте**.

Започваме да обхождаме числата и проверяваме дали настоящото число **a** е **по-голямо** от **предходното aPrev**. Ако това е изпълнено, значи подредицата **е нарастваща** и трябва да увеличим **броя** ѝ с **1**. Това запазваме в променливата, която следи дължината на редицата, в която се намираме в момента, а именно - **countCurrentLongest**. Ако **числото a не е по-голямо** от предходното, това означава, че започва **нова подредица** и трябва да стартираме броенето от **1**. Накрая, след всички проверки, **aPrev** става **числото**, което използваме **в момента**, и започваме цикъла от начало със **следващото** въведено **a**.

Ето и примерна реализация на описания алгоритъм:

```
for (int i = 0; i < n; i++) {
    cin >> a;

    if (a > aPrev) {
        countCurrentLongest++;
    }
    else {
        countCurrentLongest = 1;
    }
    aPrev = a;
}
```

Остава да разберем коя от всички редици е **най-дълга**. Това ще направим с проверка в цикъла дали редицата, в която се намираме **в момента**, е станала по-дълга от дължината на **най-дългата намерена до сега**. Целият цикъл ще изглежда така:

```
for (int i = 0; i < n; i++) {
    cin >> a;

    if (a > aPrev) {
        countCurrentLongest++;
    }
    else {
        countCurrentLongest = 1;
    }

    if (countCurrentLongest > countLongest) {
        countLongest = countCurrentLongest;
    }
    aPrev = a;
}
```

Накрая извеждаме дължината на **най-дългата** намерена редица.

```
cout << countLongest << endl;
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1370#7>.

Задачи за чертане на фигурки на конзолата

Петата задача от практическия изпит по “Основи на програмирането” изискава използване на един или няколко вложени цикъла за рисуване на някаква фигурка на конзолата. Може да се изискват логически размишления, извършване на прости пресмятания и проверки. Задачата проверява способността на студентите да мислят логически и да измислят прости алгоритми за решаване на задачи, т.е. да мислят алгоритично. Ето няколко примера за изпитни задачи:

Задача: перфектен диамант

Да се напише програма, която прочита от конзолата цяло число **n** и чертае перфектен диамант с размер **n** като в примерите по-долу.

Вход

Входът е цяло число **n** в интервала [1 ... 1000].

Изход

На конзолата трябва да се отпечата диамантът като в примерите.

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
2	<pre> * *_* * </pre>	3	<pre> * *_* *-*_* * </pre>	4	<pre> * *_* *-*_*_ *-*_*_* * </pre>	5	<pre> * *_* *-*_*_ *-*_*_*_ *-*_*_*_* *-*_*_* * </pre>

Насоки и подсказки

В задачите за чертане на фигурки най-важното, което трябва да преценим е **последователността**, в която ще рисуваме. Кои елементи се **повтарят** и с какви **стъпки**. В конкретната задача, ясно може да забележим, че **горната и долната** част на диаманта са **еднакви**. Най-лесно ще я решим, като направим **един цикъл**, който чертае **горната част**, и след това още **един**, който чертае **долната** (обратно на горната).

Първо прочитаме числото **n** от конзолата:

```

int n;
cin >> n;
  
```

Започваме да рисуваме горната половина на диаманта. Ясно виждаме, че **всеки ред** започва с няколко празни места и *. Ако се вгледаме по- внимателно, ще забележим, че **празните места** са винаги равни на **n - броя на реда** (на първия ред са n-1, на втория n-2 и т.н.). Ще започнем с това да нарисуваме броя **празни места**, както и **първата звездичка**. На края на реда пишем **cout << endl;**, за да преминем на **нов ред**. Забележете, че започваме да броим от 1, а не от 0. След това ще остане само да добавим няколко пъти -*, за да **довършим реда**.

Ето фрагмент от кода за начертаване на горната част на диаманта:

```
for (int i = 1; i < n; i++) {
    for (int j = 0; j < n - i; j++) {
        cout << ' ';
    }
    cout << "*";
    //TODO: Draw the rest of the line.
    cout << endl;
}
```

Остава да **довършим** **всеки ред** с нужния брой -* елементи. На всеки ред трябва да добавим **i - 1** такива **елемента** (на първия 1-1 -> 0, на втория -> 1 и т.н.).

Ето и пълният код за начертаване на горната част на диаманта:

```
for (int i = 1; i < n; i++) {
    for (int j = 0; j < n - i; j++) {
        cout << ' ';
    }
    cout << "*";

    for (int j = 0; j < i - 1; j++) {
        cout << "-*";
    }
    cout << endl;
}
```

За да изрисуваме **долната част** на диаманта, трябва да обърнем **горната** на обратно. Ще броим от **n - 1**, тъй като ако започнем от **n**, ще изрисуваме средния ред два пъти. Не забравяйте да смените **стъпката** от **++** на **--**.

Ето го и кода за начертаване на **долната част** на диаманта:

```

for (int i = n - 1; i >= 1; i--) {
    for (int j = 0; j < n - i; j++) {
        cout << ' ';
    }
    cout << "*";

    for (int j = 0; j < i - 1; j++) {
        cout << "-*";
    }
    cout << endl;
}

```

Остава да сглобим цялата програма като първо четем входа, печатаме горната част на диаманта и след него и долната част на диаманта.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1370#8>.

Задача: правоъгълник със звездички в центъра

Да се напише програма, която прочита от конзолата цяло число **n** и чертае правоъгълник с размер **n** с две звездички в центъра, като в примерите по-долу.

Вход

Входът е цяло число **n** в интервала [2 ... 1000].

Изход

На конзолата трябва да се изведе правоъгълникът като в примерите.

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
2	%%%%% %*% %%%%	3	%%%%%% % % % ** % % % %%%%%%	4	%%%%%% % % % ** % % % %%%%%%	5	%%%%%%%%% % % % % % * * % % % % % %%%%%%%%%

Насоки и подсказки

Прочитаме входните данни от задачата:

```
int n;
cin >> n;
```

Първото нещо, което лесно забелязваме, е че **първият и последният ред** съдържат **$2 * n$** символа **%**. Ще започнем с това и после ще нарисуваме средата на четириъгълника:

```
for (int i = 0; i < n * 2; i++) {
    cout << "%";
}

// TODO: Draw the middle of the rectangle.

for (int i = 0; i < n * 2; i++) {
    cout << "%";
}
```

От дадените примери виждаме, че **средата** на фигурата винаги има **нечетен брой** редове. Забелязваме, че когато е зададено **четно число**, броят на редовете е равен на **предишното нечетно** ($2 \rightarrow 1$, $4 \rightarrow 3$ и т.н.). Създаваме променлива, която представлява броя редове, които ще има нашият правоъгълник, и я коригираме, ако числото **n** е **четно**.

```
int numRows;
if (n % 2 == 0) {
    numRows = n - 1;
}
else {
    numRows = n;
}
```

След това ще нарисуваме **правоъгълника без звездичките**. Всеки ред има за **начало и край** символа **%** и между тях $2 * n - 2$ празни места (ширината е $2 * n$ и вадим 2 за двата процента в края). Не забравяйте да преместите кода за **последния ред** след **цикъла**:

```
cout << endl;
for (int i = 0; i < numRows; i++) {
    cout << "%";
    //TODO: Place the stars.
    cout << "%" << endl;
}
```

Можем да **стартираме** и **тестваме** кода до тук. Всичко без двете звездички в средата трябва да работи коректно.

Сега остава **в тялото** на цикъла да добавим и **звездичките**. Ще направим проверка дали сме на **средния ред**. Ако сме там, ще нарисуваме **реда** заедно **със звездичките**, ако не - ще нарисуваме **нормален ред**. Редът със звездичките има **n**

- **2 празни места**(**n** е половината дължина и махаме звездичката и процента), **две звезди** и отново **n - 2 празни места**. Двата процента в началото и в края на реда си ги оставяме извън проверката:

```
cout << endl;
for (int i = 0; i < numRows; i++) {
    cout << "%";

    if (i == numRows / 2) {
        for (int j = 0; j < n - 2; j++) {
            cout << ' ';
        }
        cout << "***";

        for (int j = 0; j < n - 2; j++) {
            cout << ' ';
        }
    }
    else {
        for (int j = 0; j < n * 2 - 2; j++) {
            cout << ' ';
        }
    }
    cout << "%" << endl;
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1370#9>.

Задачи с вложени цикли с по-сложна логика

Последната (шеста) задача от практическия изпит по “Основи на програмирането” изисква използване на **няколко вложени цикъла и по-сложна логика в тях**. Задачата проверява способността на студентите да мислят алгоритично и да решават нетривиални задачи, изискаващи съставянето на цикли. Следват няколко примера за изпитни задачи.

Задача: четворки нарастващи числа

По дадена двойка числа **a** и **b** да се генерират всички четворки **n1, n2, n3, n4**, за които $a \leq n1 < n2 < n3 < n4 \leq b$.

Вход

Входът съдържа две цели числа **a** и **b** в интервала [0 ... 1000], по едно на ред.

Изход

Изходът съдържа всички търсени четворки числа, в нарастващ ред, по една на ред.

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
	3 4 5 6				
3	3 4 5 7	5		10	
7	3 4 6 7	7	No	13	10 11 12 13
	3 5 6 7				
	4 5 6 7				

Насоки и подсказки

Ще прочетем входните данни от конзолата. Създаваме и допълнителната променлива **count**, която ще следи дали има съществуваща редица числа:

```
int a, b;
cin >> a >> b;
int count = 0;
```

Най-лесно ще решим задачата, ако логически я разделим на части. Ако се изиска да изведем всички редици от едно число между **a** и **b**, ще го направим с един цикъл, който изкарва всички числа от **a** до **b**. Нека помислим как ще стане това с редици от две числа. Отговорът е лесен - ще ползваме вложени цикли:

```
for (int i = a; i <= b; i++) {
    for (int j = i + 1; j <= b; j++) {
        cout << i << " " << j << endl;
    }
}
```

Можем да тестваме недописаната програма, за да проверим дали работи коректно до този момент. Тя трябва да отпечата всички двойки числа **i, j**, за които **i ≤ j**.

Тъй като всяко **следващо число** от редицата трябва да е **по-голямо** от **предишното**, вторият цикъл ще се върти от **i + 1**(следващото по-голямо число). Съответно, ако **не съществува редица** от две нарастващи числа (**a** и **b** са равни), вторият цикъл **няма да се изпълни** и няма да се изведе нищо на конзолата.

Аналогично, остава да реализираме по същия начин **вложените цикли** и за четири числа. Ще добавим и **увеличаване на брояча(count)**, който инициализираме в началото, за да знаем дали **съществува** такава **редица**:

```

for (int i = a; i <= b; i++) {
    for (int j = i + 1; j <= b; j++) {
        for (int k = j + 1; k <= b; k++) {
            for (int l = k + 1; l <= b; l++) {
                cout << i << " " << j << " " << k << " " << l << endl;
                count++;
            }
        }
    }
}

```

Накрая ще проверим дали броячът е равен на 0 и съответно ще принтираме "No" на конзолата, ако е така:

```

if (count == 0) {
    cout << "No" << endl;
}

```

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:
<https://judge.softuni.bg/Contests/Practice/Index/1370#10>.

Задача: генериране на правоъгълници

По дадено число n и **минимална площ** m да се генерират всички правоъгълници с цели координати в интервала $[-n \dots n]$, с площ поне m . Генерираните правоъгълници да се отпечатат в следния формат:

$(left, top)$ $(right, bottom)$ -> area

Правоъгълниците се задават чрез горния си ляв и долния си десен ъгъл. В сила са следните неравенства:

- $-n \leq left < right \leq n$
- $-n \leq top < bottom \leq n$

Вход

От конзолата се въвеждат две числа, по едно на ред:

- Цяло число n в интервала $[1 \dots 100]$ – задава минималната и максималната координата на връх.
- Цяло число m в интервала $[0 \dots 50\,000]$ – задава минималната площ на генерираните правоъгълници.

Изход

- На конзолата трябва да се отпечатат описаните правоъгълници във формат

като в примерите по-долу.

- Ако за числата **n** и **m** няма нито един правоъгълник, да се изведе "No".
- Редът на извеждане на правоъгълниците е без значение.

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
1 2	(-1, -1) (0, 1) -> 2 (-1, -1) (1, 0) -> 2 (-1, -1) (1, 1) -> 4 (-1, 0) (1, 1) -> 2 (0, -1) (1, 1) -> 2	2 17	No	3 36	(-3, -3) (3, 3) -> 36

Насоки и подсказки

Да прочетем входните данни от конзолата. Отново ще създадем и един **брояч**, в който ще пазим броя на намерените правоъгълници:

```
int n, m;
cin >> n >> m;
int count = 0;
```

Изключително важно е да успеем да си представим задачата, преди да започнем да я решаваме. В нашия случай се изисква да търсим правоъгълници в координатна система. Нещото, което знаем е, че **лявата точка** винаги ще има координата **x**, **по-малка от дясната**. Съответно **горната** винаги ще има **по-малка** координата **y** от **долната**. За да намерим всички правоъгълници, ще трябва да направим **цикъл**, подобен на този от предходната задача, но този път **не всеки следващ цикъл** ще започва от **следващото число**, защото някои от **координатите** може да са **равни** (например **left** и **top**):

```
for (int left = -n; left < n; left++) {
    for (int top = -n; top < n; top++) {
        for (int right = left + 1; right <= n; right++) {
            for (int bottom = top + 1; bottom <= n; bottom++) {
                //TODO
            }
        }
    }
}
```

С променливите **left** и **right** ще следим координатите по **хоризонталата**, а с **top** и **bottom** – по **вертикалата**. Важното тук е да знаем кои координати кои са, за да можем да изчислим правилно страните на правоъгълника. Сега трябва да

намерим лицето на правоъгълника и да направим проверка дали то е по-голямо или равно на m . Едната страна ще е разликата между **left** и **right**, а другата - между **top** и **bottom**. Тъй като координатите евентуално може да са разменени, ще ползваме абсолютни стойности. Отново добавяме и брояча в цикъла, като броим само четириъгълниците, които изписваме. Важно е да забележим, че поредността на изписване е **left**, **top**, **right**, **bottom**, тъй като така е зададено в условието:

```
for (int left = -n; left < n; left++) {
    for (int top = -n; top < n; top++) {
        for (int right = left + 1; right <= n; right++) {
            for (int bottom = top + 1; bottom <= n; bottom++) {
                int area = fabs(right - left) * fabs(bottom - top);

                if (area >= m) {
                    cout << "(" << left << ", " << top << ")" << "(" <<
                        right << ", " << bottom << ")" -> " << area << endl;
                    count++;
                }
            }
        }
    }
}
```

Накрая принтираме "No", ако не съществуват такива правоъгълници:

```
if (count == 0) {
    cout << "No" << endl;
}
```

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1370#11>.

Глава 8.2. Подготовка за практически изпит – част II

В настоящата глава ще разгледаме един практически изпит по основи на програмирането, проведен в СофтУни на 18 декември 2016 г. Задачите дават добра представа какво можем да очакваме на приемния изпит по програмиране в СофтУни. Изпитът покрива изучавания учебен материал от настоящата книга и от курса "Programming Basics" в СофтУни.

Изпитни задачи

Традиционно приемният изпит в СофтУни се състои от **6 практически задачи по програмиране**:

- Задача с прости сметки (без проверки).
- Задача с единична проверка.
- Задача с по-сложни проверки.
- Задача с единичен цикъл.
- Задача с вложени цикли (чертане на фигурука на конзолата).
- Задача с вложени цикли и по-сложна логика.

Да разгледаме една **реална изпитна тема**, задачите в нея и решенията им.

Задача: разстояние

Напишете програма, която да пресмята **колко километра изминава кола**, за която знаем **първоначалната скорост** (км/ч), **времето** в минути, след което **увеличава скоростта с 10%**, **второ време**, след което **намалява скоростта с 5%**, и **времето до края** на пътуването. За да намерите разстоянието трябва да **превърнете минутите в часове** (например 70 минути = 1.1666 часа).

Входни данни

От конзолата се четат **4 реда**:

- Първоначалната скорост в км/ч – цяло число в интервала [1 ... 300].
- Първото време в минути – цяло число в интервала [1 ... 1000].
- Второто време в минути – цяло число в интервала [1 ... 1000].
- Третото време в минути – цяло число в интервала [1 ... 1000].

Изходни данни

Да се отпечата на конзолата едно число: **изминатите километри**, форматирани до **втория символ след десетичния знак**.

Примерен вход и изход

Вход	Изход	Обяснения
90 60 70 80	330.90	<p>Разстояние с първоначална скорост: $90 \text{ км/ч} * 1 \text{ час (60 мин)} = 90 \text{ км}$</p> <p>След увеличението: $90 + 10\% = 99.00 \text{ км/ч} * 1.166 \text{ часа (70 мин)} = 115.50 \text{ км}$</p> <p>След намаляването: $99 - 5\% = 94.05 \text{ км/ч} * 1.33 \text{ часа (80 мин)} = 125.40 \text{ км}$</p> <p>Общо изминати: 330.9 км</p>
140 112 75 190	917.12	<p>Разстояние с първоначална скорост: $140 \text{ км/ч} * 1.86 \text{ часа (112 мин)} = 261.33 \text{ км}$</p> <p>След увеличението: $140 + 10\% = 154.00 \text{ км/ч} * 1.25 \text{ часа (75 мин)} = 192.5 \text{ км}$</p> <p>След намаляването: $154.00 - 5\% = 146.29 \text{ км/ч} * 3.16 \text{ часа (190 мин)} = 463.28 \text{ км}$</p> <p>Общо изминати: 917.1166 км</p>

Насоки и подсказки

Вероятно е подобно условие да изглежда на пръв поглед **объркващо** и непълно, което **придава** допълнителна **сложност** на една лесна задача. Нека **разделим** заданието на няколко **подзадачи** и да се опитаме да **решим** всяка една от тях, което ще ни отведе и до крайния резултат:

- **Прочитане** на входните данни.
- **Изпълнение** на основната програмна логика.
- **Пресмятане** и оформяне на крайния резултат.

Съществената част от програмната логика се изразява в това да пресметнем какво ще бъде **изминатото разстояние след всички промени** в скоростта. Тъй като по време на **изпълнението** на програмата, част от данните, с които разполагаме, се променят, то бихме могли да **разделим решението** на няколко логически обособени стъпки:

- **Пресмятане** на изминатото **разстояние** с първоначална скорост.
- Промяна на **скоростта** и пресмятане на изминатото **разстояние**.
- Последна промяна на **скоростта** и **пресмятане**.
- **Сумиране**.

За **прочитането** на данните от **конзолата** използваме функцията **cin**:

```
int initialSpeed;
cin >> initialSpeed;
```

По условие **входните данни** се въвеждат на **четири** отделни реда, по тази причина следва да изпълним **предходния** код общо **четири** пъти:

```
int initialSpeed, firstInterval, secondInterval, thirdInterval;
cin >> initialSpeed;
cin >> firstInterval;
// ...
```

По този начин успяхме да се справим успешно с **първата подзадача** - приемане на входните данни.

За **извършване** на **пресмятанията** избираме да използваме тип **double**. Първоначално **запазваме** една помощна **променлива**, която ще използваме многократно. Този подход на централизация ни дава **Гъвкавост** и **възможност** да **променяме** цялостния резултат на програмата с минимални усилия. В случай, че се наложи да променим стойността, трябва да го направим само на **едно място в кода**, което ни спестява време и усилия:

```
const double minutesPerHour = 60.0;
```

	<p>Избягването на повтарящ се код (централизация на програмната логика) в задачите, които разглеждаме в настоящата книга, изглежда на пръв поглед излишна, но този подход е от съществено значение при изграждането на мащабни приложения в реална работна среда и упражняването му в начален стадий на изучаване само ще подпомогне усвояването на един качествен стил на програмиране.</p>
	<p>Чрез запазената дума const, поставена пред типа double, показваме, че променливата minutesPerHour е константа. Веднъж запазена в константата стойността 60.0 не може да бъде променяна. На пръв поглед използването на константи в задачите, които разглеждаме в настоящата книга може би изглежда излишно, но те също имат своето приложение в мащабните проекти. Например могат да предотвратят допускането на бъдещи грешки при разработката на софтуер, защото ако компилатора забележи опит за промяна на константа, той веднага ще даде грешка по време на компилиация. Освен това, в зависимост от компилатора, използването на константи може леко да ускори изпълнението на програмата и/или леко да намали използваната от нея памет.</p>

Изминалото време в часове пресмятаме като **разделим** подаденото ни **време на 60** (минутите в един час). **Изминатото разстояние** намираме като **умножим**

началната скорост с изминалото време (в часове). След това променяме скоростта, като я увеличаваме с 10% (по условие). Пресмятането на **процентите**, както и следващите изминати **разстояния**, извършваме по следния начин:

- **Интервалът от време** (в часове) намираме като **разделим** зададения интервал в минути на минутите, които се съдържат в един час (60).
- **Изминатото разстояние** намираме като **умножим** интервала (в часове) по скоростта, която получаваме след увеличението.
- Следващата стъпка е да **намалим скоростта с 5%**, както е зададено по условие.
- Намираме **оставащото разстояние** по описания начин в първите две точки.

```
double fistIntervalHours = firstInterval / minutesPerHour;
double firstDistance = initialSpeed * fistIntervalHours;
```

```
double speedAfterIncrease = initialSpeed
    + ( (initialSpeed * 10.0) / 100.0);
double secondIntervalHours = secondInterval / minutesPerHour;
double secondDistance = speedAfterIncrease * secondIntervalHours;
```

До този момент успяхме да изпълним две от най-важните подзадачи, а именно приемането на **данните** и **тяхната обработка**. Остава ни само да пресметнем **крайния резултат**. Тъй като по условие се изисква той да бъде **форматиран до 2 символа** след десетичния знак, можем да направим това по следния начин:

```
double finalDistance =
    firstDistance + secondDistance + thirdDistance;
cout << fixed;
cout << setprecision(2) << finalDistance << endl;
```

В случай, че сте работили правилно и изпълните програмата с входните данни от условието на задачата, ще се уверите, че тя работи коректно.

Тестване в Judge системата

Тествайте решението си в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1371#0>.

Задача: смяна на плочки

Хараламби има **събрани пари**, с които иска да **смени плочките** на пода в банята. Като **подът е правоъгълник**, а **плочките са триъгълни**. Напишете програма, която да пресмята дали събранныте пари ще му стигнат. Подават се широчината и дължината на пода, както и едната страна на триъгълника с височината към нея. Трябва да **пресметнете колко плочки са нужни**, за да се покрие пода. Броят на

плочките трябва да се закръгли към по-високо цяло число и да се прибавят още 5 броя за фира. В допълнение ни се подават още – цената на плочка и сумата за работата на майстор.

Входни данни

От конзолата се четат 7 реда:

- Събрани пари.
- Широчината на пода.
- Дължината на пода.
- Страната на триъгълника.
- Височината на триъгълника.
- Цена на една плочка.
- Сумата за майстора.

Обърнете внимание, че всички числа са реални числа в интервала [0.00 ... 5000.00].

Изходни данни

На конзолата трябва да се отпечата на един ред:

- Ако парите са достатъчно:
 - "{Оставащите пари} lv left."
- Ако парите НЕ са достатъчно:
 - "You'll need {Недостигащите пари} lv more."

Резултатът трябва да е форматиран до втория символ след десетичния знак.

Примерен вход и изход

Вход	Изход	Обяснения
500 3 2.5 0.5 0.7 7.80 100	25.60 lv left.	Площ на пода $\rightarrow 3 * 2.5 = 7.5$ Площта на плочка $\rightarrow 0.5 * 0.7 / 2 = 0.175$ Необходими плочки $\rightarrow 7.5 / 0.175 = 42.857\dots = 43 + 5$ фира = 48 Обща сума $\rightarrow 48 * 7.8 + 100$ (майстор) = 474.4 $474.4 < 500 \rightarrow$ остават 25.60 лева

Вход	Изход	Обяснения
1000		Площ на пода $\rightarrow 5.55 * 8.95 = 49.67249$
5.55		Площта на плочка $\rightarrow 0.9 * 0.85 / 2 = 0.3825$
8.95	You'll need 1209.65 lv more.	Необходими плочки $\rightarrow 49.67249 / 0.3825 = 129.86... = 130 + 5$ фира = 135
0.90		Обща сума $\rightarrow 135 \cdot 13.99 + 321$ (майстор) = 2209.65
0.85		$2209.65 > 1000 \rightarrow$ *не достигат 1209.65 лева
13.99		
321		

Насоки и подсказки

Следващата задача изиска от нашата програма да приема повече входни данни и извърши по-голям брой изчисления, въпреки че решението е **идентично**. Приемането на данните от потребителя извършваме по добре **познатия ни начин**. Обърнете внимание, че в раздел **Вход** в условието е упоменато, че всички входни данни ще бъдат **реални числа** и поради тази причина ще използваме тип **double**.

След като вече разполагаме с всичко необходимо, за да изпълним програмната логика, можем да пристъпим към следващата част. Как бихме могли да **изчислим** какъв е **необходимият** брой плочки, които ще бъдат достатъчни за покриването на целия под? Условието, че плочките имат **триъгълна** форма, би могло да доведе до объркване, но на практика задачата се свежда до съвсем **прости изчисления**. Бихме могли да пресметнем каква е **общата площ на пода** по формулата за намиране на площ на правоъгълник, както и каква е **площта на една плочка** по съответната формула за триъгълник.

За да пресметнем какъв **брой плочки** са необходими, **разделяме площта на пода на площта на една плочка** (като не забравяме да прибавим 5 допълнителни броя плочки, както е по условие).



Обърнете внимание, че в условието е упоменато да закръглим броя на плочките, получен от делението, до по-високо цяло число, след което да прибавим 5. Потърсете повече информация за системната функционалност за това: **ceil(...)**.

До крайния резултат можем да стигнем, като **пресметнем общата сума**, която е необходима, за да бъде покрит целия под, като **съберем цената на плочките с цената за майстора**, която имаме от входните данни. Можем да се досетим, че **общият разход** за плочките можем да получим, като **умножим броя плочки по цената за една плочка**. Дали сумата, с която разполагаме, ще бъде достатъчна, разбираме като сравним събраните до момента пари (от входните данни) и общите разходи:

```
if (budget >= totalCost) {
    // TODO: Print message
}
```

```

else {
    // TODO: Print message
}

```

Тестване в Judge системата

Тествайте решението си в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1371#1>.

Задача: магазин за цветя

Магазин за цветя предлага 3 вида цветя: **хризантеми**, **рози** и **лалета**. Цените зависят от сезона.

Сезон	Хризантеми	Рози	Лалета
пролет / лято	2.00 лв./бр.	4.10 лв./бр.	2.50 лв./бр.
есен / зима	3.75 лв./бр.	4.50 лв./бр.	4.15 лв./бр.

В празнични дни цените на всички цветя се **увеличават с 15%**. Предлагат се следните **отстъпки**:

- За закупени повече от 7 лалета през пролетта – **5% от цената** на целия букет.
- За закупени 10 или повече рози през зимата – **10% от цената** на целия букет.
- За закупени повече от 20 цветя общо през всички сезони – **20% от цената** на целия букет.

Отстъпките се правят по така написания ред и могат да се наслагват! Всички отстъпки важат след осъкъпяването за празничен ден!

Цената за аранжиране на букета винаги е **2 лв.** Напишете програма, която изчислява **цената за един букет**.

Входни данни

Входът се чете от **конзолата** и съдържа точно 5 реда:

- Броят на закупените **хризантеми** – цяло число в интервала [0 ... 200].
- Броят на закупените **рози** – цяло число в интервала [0 ... 200].
- Броят на закупените **лалета** – цяло число в интервала [0 ... 200].
- Сезонът – [Spring, Summer, Autumn, Winter].
- Дали денят е празник – [Y - да / N - не].

Изходни данни

Да се отпечата на конзолата 1 число – **цената на цветята**, форматирана до втория символ след десетичния знак.

Примерен вход и изход

Вход	Изход	Обяснения
2		Цена: $2 * 2.00 + 4 * 4.10 + 8 * 2.50 = 40.40$ лв.
4		Празничен ден: $40.40 + 15\% = 46.46$ лв.
8		5% намаление за повече от 7 лалета през пролетта: 44.14
Spring	46.14	Общо цветята са 20 или по-малко: няма намаление
Y		$44.14 + 2 \text{ за аранжиране} = 46.14$ лв.

Вход	Изход	Обяснения
3		Цена: $3 * 3.75 + 10 * 4.50 + 9 * 4.15 = 93.60$ лв.
10		Не е празничен ден: няма увеличение
9		10% намаление за 10 или повече рози през зимата: 84.24
Winter	69.39	Общо цветята са повече от 20: 20% намаление = 67.392
N		$67.392 + 2 \text{ за аранжиране} = 69.392$ лв.

Насоки и подсказки

След като прочитаме внимателно условието разбираме, че отново се налага да извършваме **прости пресмятания**, но с разликата, че този път ще са необходими и **повече логически проверки**. Следва да обърнем повече **внимание** на това, в какъв момент се **извършват промените** по крайната цена, за да можем правилно да изградим логиката на нашата програма. Отново, удебеленият текст ни дава достатъчно **насоки** как да подходим. Като за начало, отделяме вече **декларирани** стойности в **променливи**, както направихме и в предишните задачи:

```
// Initial price list
const double roseAutumnWinterPrice = 4.5;
const double roseSpringSummerPrice = 4.1;
const double tulipAutumnWinterPrice = 4.15;
const double tulipSpringSummerPrice = 2.5;
const double chrysanthemumAutumnWinterPrice = 3.75;
const double chrysanthemumSpringSummerPrice = 2;
const double arrangePrice = 2;
```

Правим същото и за останалите вече декларирани стойности:

```
// Price increases
const double priceIncreasePercentage = 15.0;

// Price decreases
const double tulipPriceDecreasePercentage = 5.0;
const double rosePriceDecreasePercentage = 10.0;
const double totalPriceDecreasePercentage = 20.0;

// Price decrease thresholds
const double tulipPriceDecreaseThreshold = 7.0;
const double rosePriceDecreaseThreshold = 10.0;
const double totalPriceDecreaseThreshold = 20.0;
```

Следващата ни подзадача е да прочетем правилно **входните** данни от конзолата. Подхождаме по добре познатия ни начин:

```
int chrysantemumsPurchased, rosesPurchased,
    tulipsPurchased;
string season, isSpecialDay;
cin >> chrysantemumsPurchased;
// ...
```

Нека помислим кой е най-подходящият начин да **структуриме** нашата програмна логика. От условието става ясно, че пътят на програмата се разделя основно на две части: **пролет / лято и есен / зима**. Разделението ще направим с условна конструкция **if-else**, като преди това заделяме променливи за цените на отделните цветя, както и за **крайния резултат**:

```
if (season == "Winter" || season == "Autumn") {
    rosesPrice = rosesPurchases * roseAutumnWinterPrice;
    chrysantemumsPrice = chrysantemumsPurchases
        * chrysanthemumAutumnWinterPrice;
    tulipsPrice = tulipsPurchased * tulipAutumnWinterPrice;
    totalCost = rosesPrice + chrysantemumsPrice + tulipsPrice;
}
else {
    // TODO
}
```

Остава ни да извършим **няколко проверки** относно **намаленията** на различните видове цветя, в зависимост от сезона, и да модифицираме крайния резултат.

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1371#2>.

Задача: оценки

Напишете програма, която да пресмята статистика на оценки от изпит. В началото програмата получава **броя на студентите**, явили се на изпита, и за **всеки студент неговата оценка**. На края програмата трябва да отпечата процента на студенти с оценка между 2.00 и 2.99, между 3.00 и 3.99, между 4.00 и 4.99, 5.00 или повече, както и **средни успех** на изпита.

Входни данни

От конзолата се четат поредица от числа, всяко на отделен ред:

- На първия ред – броят на студентите, явили се на изпит – цяло число в интервала [1 ... 1000].
- За всеки един студент на отделен ред – оценката от изпита – реално число в интервала [2.00 ... 6.00].

Изходни данни

Да се отпечатат на конзолата **5 реда**, които съдържат следната информация:

- "Top students: {процент студенти с успех 5.00 или повече}%".
- "Between 4.00 and 4.99: {между 4.00 и 4.99 включително}%".
- "Between 3.00 and 3.99: {между 3.00 и 3.99 включително}%".
- "Fail: {по-малко от 3.00}%".
- "Average: {среден успех}".

Резултатите трябва да са форматирани до втория символ след десетичния знак.

Примерен вход и изход

Вход	Изход
6	
2	Top students: 33.33%
3	Between 4.00 and 4.99: 16.67%
4	Between 3.00 and 3.99: 16.67%
5	Fail: 33.33%
6	Average: 3.70
2.2	

Вход	Изход	Обяснения
10		
3.00		
2.99	Top students: 30.00%	5 и повече - трима = 30% от 10
5.68	Between 4.00 and 4.99: 30.00%	Между 4.00 и 4.99 - трима = 30% от 10
3.01	Between 3.00 and 3.99: 20.00%	Между 3.00 и 3.99 - двама = 20% от 10
4	Fail: 20.00%	Под 3 - двама = 20% от 10
4	Average: 4.06	Средният успех е: $3 + 2.99 + 5.68 + 3.01 + 4 + 4 + 6 + 4.50 + 2.44 + 5 = 40.62 / 10 = 4.062$
6.00		
4.50		
2.44		
5		

Насоки и подсказки

От условието виждаме, че **първо** ще ни бъде подаден **броя** на студентите, а едва след това **оценките** им. По тази причина **първо** ще прочетем **броя** на студентите и ще го запазим в променлива от тип **int**. За да прочетем и обработим самите оценки, ще използваме **for** цикъл. Всяка итерация на цикъла ще прочита и обработва по една оценка:

```
int numberOfRowsStudents;
cin >> numberOfRowsStudents;

for (int i = 0; i < numberOfRowsStudents; i++) {
    // TODO
}
```

Преди да се изпълни кода от **for** цикъла заделяме променливи, в които ще пазим **броя на студентите** за всяка група: слаби резултати (до 2.99), резултати от 3 до 3.99, от 4 до 4.99 и оценки над 5. Ще ни е необходима и още една променлива, в която да пазим **сумата на всички оценки**, с помощта на която ще изчислим средната оценка на всички студенти:

```
int numberOfFailedStudents = 0;
int numberOfRowsAverageStudents = 0;
int numberOfRowsGoodStudents = 0;
int numberOfRowsExcellentStudents = 0;
double totalResult = 0;
```

Завъртаме **цикъла** и в него **декларираме още една** променлива, в която ще запазваме **текущата** въведена оценка. Променливата ще е от тип **double** и на всяка итерация проверяваме **каква е стойността** ѝ. Според тази стойност,

увеличаваме броя на студентите в съответната група с **1**, като не забравяме да увеличим и **общата** сума на оценките, която също следим:

```
int numberOfStudents;
cin >> numberOfStudents;

for (int i = 0; i < numberOfStudents; i++) {
    double grade;
    cin >> grade;

    totalResult += grade;
    if (grade < 3) {
        numberOfFailedStudents++;
    }
    // else if ()
    // TODO: check other groups
}
```

Какъв **процент** заема дадена **група студенти** от общия брой, можем да пресметнем като **умножим** броя на **студентите** от съответната група по **100** и след това разделим на **общия брой студенти**.



Обърнете внимание с какъв числен тип данни работите при извършване на тези пресмятания.

Крайният резултат оформяме по добре познатия ни начин **до втория символ** след десетичния знак.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1371#3>.

Задача: коледна шапка

Да се напише програма, която прочита от конзолата **цяло число n** и чертае **коледна шапка** с ширина **4 * N + 1** колони и височина **2 * N + 5** реда като в примерите по-долу.

Входни данни

Входът се чете от конзолата – едно **цяло число N** в интервала [3 ... 100].

Изходни данни

Да се отпечата на конзолата **коледна шапка**, точно както в примерите.

Примерен вход и изход

Насоки и подсказки

При задачите за **чертане** на конзолата, най-често потребителят въвежда **едно цяло число**, което е свързано с **общата големина на фигурката**, която трябва да начертаем. Тъй като в условието е упоменато как се изчисляват общата дължина и широчина на фигурката, можем да ги използваме за **отправни точки**. От примерите ясно се вижда, че без значение какви са входните данни, винаги имаме **първи два реда**, които са с почти идентично съдържание.

..... / \

Забелязваме също така, че последните три реда винаги присъстват, два от които са напълно еднакви.

A decorative horizontal border consisting of a repeating pattern of asterisks (*). The border is approximately 840 units wide and 100 units high.

От тези наши наблюдения можем да изведем **формулата** за **височина на променливата част** на коледната шапка. Използваме зададената по условие

формула за общата височина, като изваждаме големината на непроменливата част. Получаваме **(2 * n + 5) - 5** или **2 * n**.

За начертаването на **динамичната** част от фигурката ще използваме **цикъл**. Размерът на цикъла ще бъде от **0** до **широкината**, която имаме по условие, а именно **4 * n + 1**. Тъй като тази формула ще използваме на **няколко места** в кода, е добра практика да я изнесем в **отделна променлива**. Преди изпълнението на цикъла би следвало да **заделим променливи за броя** на отделните символи, които участват в динамичната част: **точки и тирета**. Чрез изучаване на примерите можем да изведем формули и за **стартовите стойности** на тези променливи. Първоначално **тиретата** са **0**, но броя на **точките** ясно се вижда, че можем да получим като от **общата широчина** извадим **3** (броя символи, които изграждат върха на коледната шапка) и след това **разделим на 2**, тъй като броя точки от двете страни на шапката е **еднакъв**.

```
.....***.....
.....*-*-*.....
....*--*--*.....
....*---*---*...
...*----*----*...
..*-----*-----*..
.*-----*-----*.
*-----*-----**
```

Остава да изпълним тялото на цикъла, като **след всеки** начертан ред **намаляваме** броя на точките с **1**, а **тиретата увеличим** с **1**. Нека не забравяме да начертаем и по една **звездичка** между тях. Последователността на чертане в тялото на цикъла е следната:

- Символен низ от точки
- Звезда
- Символен низ от тирета
- Звезда
- Символен низ от тирета
- Звезда
- Символен низ от точки

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1371#4>.

Задача: комбинации от букви

Напишете програма, която да принтира на конзолата **всички комбинации** от **3**

букви в зададен интервал, като се пропускат комбинациите, **съдържащи зададена от конзолата буква**. Накрая трябва да се принтира броят отпечатани комбинации.

Входни данни

Входът се чете от **конзолата** и съдържа **точно 3 реда**:

- Малка буква от английската азбука за начало на интервала – от 'a' до 'z'.
- Малка буква от английската азбука за край на интервала – от **първата буква** до 'z'.
- Малка буква от английската азбука – от 'a' до 'z' – като комбинациите, съдържащи тази буква се пропускат.

Изходни данни

Да се отпечатат на един ред **всички комбинации**, отговарящи на условието, следвани от **броя им**, разделени с интервал.

Примерен вход и изход

Вход	Изход	Обяснения
a c b	aaa aac aca acc caa cac cca ccc 8	Всички възможни комбинации с буквите 'a', 'b' и 'c' са: aaa aab aac aba abb abc aca acb acc baa bab bac bba bbb bbc bca bcb bcc caa cab cac cba cbb cbc cca ccc ccc Комбинациите, съдържащи 'b' , не са валидни. Остават 8 валидни комбинации.
f k h	ffff ffg ffi ffj ffk fgf fgg fgi fgj fgk fif fig fii fij fik fjf fji fjj fjk fkf fkg fki fkj fkk gff gfg gfi gfj gfk ggf ggg ggi ggi ggk gif gig giij gik gif gjg gjii gjj gjk gkf gkg gki gkj gkk iff ifg ifi ifj ifk ifg iff igg iig iij iik iif ijj iij iji ijk iif ikg iki ijk ikk jff jfg jfi jfj jfk jgf jgg jgi jgj jkg jif jig jii jjk jik jjf jgg jjl jjj jjk jkf jkg jki jkj jkk kff kfg kfi kfj kfk kgf kgg kgi kgj kgk kif kig kii kij kik kjf kkg kjj kjk kkf kkg kki kkj kkk 125	

Насоки и подсказки

За последната задача имаме по условие входни данни на **3 реда**, които са представени от по един символ от **ASCII таблицата** (<https://www.asciitable.com/>). Бихме могли да използваме функцията **cin** в езика C++, за да преобразуваме входните данни в типа **char** по следния начин:

```
char startLetter;
cin >> startLetter;
```

Нека помислим как бихме могли да стигнем до **крайния резултат**. В случай че условието на задачата е да се принтират всички от началния до крайния символ (с пропускане на определена буква), как бихме постъпили?

Най-лесният и удачен начин е да използваме **цикъл**, като преминем през **всички символи** и принтираме тези, които са **различни** от **буквата**, която трябва да пропуснем. Едно от предимствата на езика C++ е, че имаме възможност да използваме различен тип данни за циклична променлива:

```
for (char i = 'a'; i <= 'z'; i++) {
    cout << i << " ";
}
```

Резултатът от изпълнението на горния код е всички букви от **a** до **z** включително, принтирани на един ред и разделени с интервал. Това прилика ли на крайния резултат от нашата задача? Трябва да измислим **начин**, по който да се принтират по **3 символа**, както е по условие, вместо по **1**. Изпълнението на програмата много прилича на игрална машина. Там най-често печелим, ако успеем да наредим няколко еднакви символа. Да речем, че на машината имаме места за три символа. Когато **спрем** на даден **символ** на първото място, на останалите две места **продължават** да се изреждат символи от всички възможни. В нашия случай **всички възможни** са буквите от началната до крайната такава, зададена от потребителя, а решението на нашата програма е идентично на начина, по който работи игралната машина.

Използваме **цикъл**, който минава през **всички символи** от началната до крайната буква включително. На **всяка итерация** на **първия цикъл** пускаме **втори** със същите параметри (но **само ако** буквата на първия цикъл е валидна, т.е. не съвпада с тази, която трябва да изключим по условие). На всяка итерация на **втория цикъл** пускаме още **един** със **същите параметри** и същата проверка. По този начин ще имаме три вложени цикъла, като в тялото на **последния** ще добавяме символите към крайния резултат:

```
for (char i = startLetter; i <= endLetter; i++) {
    if (i != exceptLetter) {
        ...
    }
}
```

Нека не забравяме, че се изисква от нас да принтираме и **общия брой валидни комбинации**, които сме намерили, както и че те трябва да се принтират на **същия ред**, разделени с интервал. Тази подзадача оставяме на читателя.

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1371#5>.

Глава 9.1. Задачи за шампиони – част I

В настоящата глава ще предложим на читателя няколко малко по-трудни задачи, които имат за цел развиване на алгоритмични умения и усвояване на програмни техники за решаване на задачи с по-висока сложност.

По-сложни задачи върху изучавания материал

Ще решим заедно няколко задачи по програмиране, които обхващат изучавания в книгата учебен материал, но по трудност надвишават обичайните задачи от приемните изпити в СофтУни. Ако искате да станете шампиони по основи на програмирането, ви препоръчваме да тренирате решаване на подобни по-сложни задачи, за да ви е лесно на изпитите.

Задача: пресичащи се редици

Имаме две редици:

- редица на Трибоначи (по аналогия с редицата на Фиbonачи), където всяко число е **сумата от предните три** (при дадени начални три числа).
- редица, породена от **числова спирала**, дефинирана чрез спираловидно обхождане на матрица от числа (дясно, долу, ляво, горе, дясно, долу, ляво, горе и т.н.), стартирайки от нейния център с дадено начално число и стъпка на увеличение, със записване на текущите числа в редицата всеки път, когато направим завой.

Да се напише програма, която намира най-малкото число, което се появява **и в двете така дефинирани редици**.

Пример

Нека редицата на Трибоначи да започне с 1, 2 и 3.

Това означава, че **първата редица** ще съдържа числата 1, 2, 3, 6, 11, 20, 37, 68, 125, 230, 423, 778, 1431, 2632, 4841, 8904, 16377, 30122, 55403, 101902 и т.н.

Същевременно, нека **числата в спиралата** да започнат с 5 и спиралата да се увеличава с 2 на всяка стъпка:

Тогава **втората редица** ще съдържа числата 5, 7, 9, 13, 17, 23, 29, 37, 45, 55, 65 и т.н. Виждаме, че 37 е първото число (най-малкото), което се среща в редицата на Трибоначи и в спиралата, следователно това е търсеното решение на задачата.

45	55
...	17	19	21	23	...
...	15	5	7
...	13	11	9
37	29	...
...	65

Входни данни

Входните данни трябва да бъдат прочетени от конзолата.

- На първите три реда от входа ще прочетете **три цели числа**, представляващи **първите три числа** в редицата на Трибоначи.
- На следващите два реда от входа, ще прочетете **две цели числа**, представляващи **първото число и стъпката** за всяка клетка на матрицата за спиралата от числа.

Входните данни винаги ще бъдат валидни и винаги ще са в описания формат. Няма нужда да ги проверявате.

Изходни данни

Резултатът трябва да бъде принтиран на конзолата.

На единствения ред от изхода трябва да принтирате **най-малкото число**, което се среща и в двете последователности. Ако няма число в **диапазона [1 ... 1 000 000]**, което да се среща и в двете последователности, принтирайте "No".

Ограничения

- Всички числа във входа ще бъдат в диапазона **[1 ... 1 000 000]**.
- Позволено работно време за програмата: 0.5 секунди.
- Позволена памет: 16 MB.

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход	Вход	Изход
1		13		99		4	
2		25		99		1	
3	37	99	13	99	No	7	71
5		5		2		23	
2		2		2		3	

Насоки и подсказки

Задачата изглежда доста сложна и затова ще я разбием на по-прости подзадачи:

- обработване на входа
- генериране на редица на Трибоначи
- генериране на числовата спирала
- намиране на най-малкото общо число за двете редици

Обработване на входа

Първата стъпка от решаването на задачата е да прочетем и обработим входа. Входните данни се състоят от **5 цели числа**: **3** за редицата на Трибоначи и **2** за числовата спирала:

```
int tribonacciFirst, tribonacciSecond, tribonacciThird;
cin >> tribonacciFirst >> tribonacciSecond >> tribonacciThird;

int spiralCurrent, spiralStep;
cin >> spiralCurrent >> spiralStep;
```

След като имаме входните данни, трябва да помислим как ще генерираме числата в двете редици.

Генериране на редица на Трибоначи

За редицата на Трибоначи всеки път ще събираме предишните три стойности и след това ще отнемваме стойностите на тези числа (трите предходни) с една позиция напред в редицата, т.е. стойността на първото трябва да приеме стойността на второто, стойността на второто - стойността на третото, а стойността на третото трябва да стане новото намерено число.

Когато сме готови с числото, ще добавяме стойността му във **вектор** (списък). За да използваме вектори в езика C++, трябва да включим библиотеката **vector** чрез директивата **#include <vector>** в началото на кода. Когато имаме създаден вектор, добавянето на елементи в него става с метода **push_back(...)**.

Понеже в условието на задачата е казано, че числата в редиците не превишават 1,000,000, можем да спрем генерирането на тази редица именно при 1,000,000:

```
vector<int> tribonacciNumbers;
tribonacciNumbers.push_back(tribonacciFirst);
tribonacciNumbers.push_back(tribonacciSecond);
tribonacciNumbers.push_back(tribonacciThird);

int tribonacciCurrent = tribonacciThird;
while (tribonacciCurrent < 1000000) {
    tribonacciCurrent = tribonacciFirst +
        tribonacciSecond +
        tribonacciThird;

    tribonacciNumbers.push_back(tribonacciCurrent);
```

```

tribonacciFirst = tribonacciSecond;
tribonacciSecond = tribonacciThird;
tribonacciThird = tribonacciCurrent;
}

```

Генериране на числовата спирала

Трябва да измислим **зависимост** между числата в числовата спирала, за да можем лесно да генерираме всяко следващо число, без да се налага да разглеждаме матрици и тяхното обхождане. Ако разгледаме внимателно картиката от условието, можем да забележим, че **на всеки 2 "завоя" в спиралата числата, които прескачаме, се увеличават с 1**, т.е. от 5 до 7 и от 7 до 9 не се прескача нито 1 число, а директно **събираме със стъпката** на редицата. От 9 до 13 и от 13 до 17 прескачаме едно число, т.е. събираме два пъти стъпката. От 17 до 23 и от 23 до 29 прескачаме две числа, т.е. събираме три пъти стъпката и т.н.

Така виждаме, че при първите две имаме **последното число + 1 * стъпката**, при следващите две събираме с **2 * стъпката** и т.н. Всеки път, когато искаме да стигнем до следващото число от спиралата, ще трябва да извършваме такива изчисления:

```
spiralCurrent += spiralStep * spiralStepMul;
```

Това, за което трябва да се погрижим, е **на всеки две числа нашият множител** (нека го наречем "коффициент") **да се увеличава с 1(spiralStepMul++)**, което може да се постигне с праста проверка за четност (**spiralCount % 2 == 0**). Целият код от генерирането на спиралата във **вектор** е даден по-долу:

```

vector<int> spiralNumbers;
spiralNumbers.push_back(spiralCurrent);
int spiralCount = 0;
int spiralStepMul = 1;

while (spiralCurrent < 1000000) {
    spiralCurrent += spiralStep * spiralStepMul;
    spiralNumbers.push_back(spiralCurrent);
    spiralCount++;
    if (spiralCount % 2 == 0) {
        spiralStepMul++;
    }
}

```

Намиране на общо число за двете редици

След като сме генерирали числата и в двете редици, можем да пристъпим към обединението им и изграждане на крайното решение. Как ще изглежда то? За **всяко от числата** в едната редица (започвайки от по-малкото) ще проверяваме дали то съществува в другата. Първото число, което отговаря на този критерий ще бъде **отговорът** на задачата.

Търсенето във втория вектор ще направим **линейно**, а за по-любопитните ще оставим да си го оптимизират, използвайки техниката наречена **двоично търсене** (Binary Search), тъй като вторият вектор се генерира сортиран, т.е. отговаря на изискването за прилагането на този тип търсене.

Ще използваме малко по-специален **for** цикъл за обработката на всяко число от двета вектора. С дадения по-долу синтаксис всеки един от двета цикъла минава по всяко от числата във векторите едно по едно и ги записва в съответната променлива.

Кодът за намиране на нашето решение ще изглежда така:

```
bool found = false;
for (int tribonacciNumber : tribonacciNumbers) {
    for (int spiralNumber : spiralNumbers) {
        if (tribonacciNumber == spiralNumber && tribonacciNumber <= 1000000) {
            cout << tribonacciNumber << endl;
            found = true;
            break;
        }
    }

    if (found) {
        break;
    }
}

if (!found) {
    cout << "No" << endl;
}
```

Решението на задачата използва вектори за запазване на стойностите. Векторите не са необходими за решаването на задачата. Съществува **алтернативно решение**, което генерира числата и работи директно с тях, вместо да ги записва във вектор. Можем на **всяка стъпка** да проверяваме дали **числата от двете редици съвпадат**. Ако това е така, ще принтираме на конзолата числото и ще прекратим изпълнението на нашата програма. В противен случай, ще видим текущото число на **коя редица е по-малко и ще генерираме следващото, там където "изоставаме"**. Идеята е, че **ще генерираме числа от редицата, която е "по-назад"**, докато не прескочим текущото число на другата редица и след това обратното, а ако междувременно намерим съвпадение, ще прекратим изпълнението:

```

while (tribonacciCurrent <= 1000000 && spiralCurrent <= 1000000) {
    if (tribonacciCurrent == spiralCurrent) {
        cout << "TODO: Print and stop execution" << endl;
    }
    else if (tribonacciCurrent < spiralCurrent) {
        cout << "TODO: Generate next Tribonacci number" << endl;
    }
    else {
        cout << "TODO: Generate next Spiral number" << endl;
    }
}

```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1372#0>.

Задача: магически дати

Дадена е **дата** във формат "**дд-мм-гггг**", напр. 15-02-2019. Изчисляваме **теглото на тази дата**, като вземем всичките ѝ цифри, умножим всяка цифра с останалите след нея и накрая съберем всички получени резултати. В нашия случай имаме 8 цифри: 17032007, така че теглото е $1*7 + 1*0 + 1*3 + 1*2 + 1*0 + 1*0 + 1*7 + 7*0 + 7*3 + 7*2 + 7*0 + 7*0 + 7*7 + 0*3 + 0*2 + 0*0 + 0*0 + 0*7 + 3*2 + 3*0 + 3*0 + 3*7 + 2*0 + 2*0 + 2*7 + 0*0 + 0*7 + 0*7 = 144$.

Нашата задача е да напишем програма, която намира всички **магически дати** - дати между две определени години (включително), отговарящи на дадено във **входните данни тегло**. Датите трябва да бъдат принтираны в нарастващ ред (по дата) във формат "**дд-мм-гггг**". Ще използваме само валидните дати в традиционния календар (високосните години имат 29 дни през февруари).

Входни данни

Входните данни трябва да бъдат прочетени от конзолата. Състоят се от 3 реда:

- Първият ред съдържа цяло число: **начална година**.
- Вторият ред съдържа цяло число: **краяна година**.
- Третият ред съдържа цяло число: **търсеното тегло** за датите.

Входните данни винаги ще бъдат валидни и винаги ще са в описания формат. Няма нужда да се проверяват.

Изходни данни

Резултатът трябва да бъде принтиран на конзолата, като последователни дати във формат "**дд-мм-гггг**", подредени по дата в нарастващ ред. Всяка дата трябва

да е на отделен ред. В случай, че няма съществуващи магически дати, да се принтира "No".

Ограничения

- Началната и крайната година са цели числа в периода [1900 - 2100].
- Магическото тегло е цяло число в диапазона [1 ... 1000].
- Позволено работно време за програмата: 2 секунди.
- Позволена памет: 16 MB.

Примерен вход и изход

Вход	Изход	Вход	Изход
2007	17-03-2007		09-01-2013
2007	13-07-2007		17-01-2013
144	31-07-2007		23-03-2013
Вход	Изход		11-07-2013
2003			01-09-2013
2004	No		10-09-2013
1500			09-10-2013
Вход	Изход		17-10-2013
2011	01-01-2011		07-11-2013
2012	10-01-2011		24-11-2013
14	01-10-2011		14-12-2013
	10-10-2011		23-11-2014

Насоки и подсказки

Започваме с вмъкване на необходимите функционалност и с четенето на входните данни от конзолата. В случая имаме вмъкваме библиотеките **sstream** и **iomanip** за обръщане на символни низове към дата и обратно и четем 3 цели числа:

```
#include <iostream>
#include <iomanip>
#include <sstream>

using namespace std;
```

```

int main()
{
    int firstYear, secondYear, numberToSearchFor;
    cin >> firstYear >> secondYear >> numberToSearchFor;

    // ...
}

```

Разполагайки с началната и крайната година, е хубаво да разберем как ще минем през всяка дата, без да се объркваме от това колко дена има в месеца и дали годината е високосна и т.н.

Обхождане на всички дати

За обхождането ще се възползваме от функционалността, която ни дава **tm** типът в C++. Ще си дефинираме променлива за **началната дата**, което можем да направим, използвайки **stringstream** и метода **get_time(...)**, чрез които от символен низ можем да създадем обект от тип **tm**. Знаям, че годината е началната година, която сме получили като параметър, а месецът и денят трябва да са съответно януари и 1-ви:

```

tm date = {};
stringstream ss("01-01-" + to_string(firstYear));
ss >> get_time(&date, "%d-%m-%Y");

```

След като имаме началната дата (в променливата **date**), искаме да направим цикъл, който се изпълнява, докато не превишим крайната година (или докато не преминем 31 декември в крайната година, ако сравняваме целите дати), като на всяка стъпка **date** се увеличава с по 1 ден.

За да увеличаваме с 1 ден при всяко завъртане, ще използваме **tm_mday**, което е част от данните на всяка една променлива от тип **tm**. В C++, за да постигнем това, трябва да увеличим **tm_mday** с 1, след което да извикаме метода **mktime**. Добавянето на 1 ден към датата ще помогне C++ да се погрижи вместо нас за прескачането в следващия месец и година, да следи колко дни има даден месец и да се погрижи вместо нас за високосните години:

```

date.tm_mday++;
mktime(&date);

```

Важна особеност при работата с дати чрез типа **tm** в C++ е, че променливата **tm_year**, чрез която можем да вземем годината от дадена дата, има стойност 0 за година 1900 и съответно стойност 119 за година 2019. Това означава, че всеки път когато искаме да вземем реалната година трябва да прибавяме 1900 към **tm_year**. Друга особеност е, че месец януари се отбележва със стойност 0 в

променливата **tm_mon** и съответно стойност 11 за декември. Тук ще прибавяме 1 към **tm_mon**, за да получим номера на месеца в интервал от 1 до 12, каквото е изискването в задачата.

В крайна сметка нашият цикъл ще изглежда по следния начин:

```
while ((date.tm_year + 1900) <= secondYear) {

    // TODO: Check the date for numberToSearchFor

    date.tm_mday++;
    mktime(&date);
}
```

Пресмятане на теглото

Всяка дата се състои от точно 8 символа (цифри) - 2 за деня (**d1**, **d2**), 2 за месеца (**d3**, **d4**) и 4 за годината (**d5** до **d8**). Това означава, че всеки път ще имаме едно и също пресмятане и може да се възползваме от това, за да дефинираме формулата **статично**(т.е. да не обикаляме с цикли, реферирайки различни цифри от датата, а да изпишем цялата формула). За да успеем да я изпишем, ще ни трябват **всички цифри от датата** в отделни променливи, за да направим всички нужни умножения. Използвайки операциите деление и взимане на остатък върху отделните компоненти на датата, чрез **tm_mday**, **tm_mon** и **tm_year**, можем да извлечем всяка цифра:

```
int d1 = date.tm_mday / 10; // First day digit
int d2 = date.tm_mday % 10; // Second day digit

int month = date.tm_mon + 1;
int d3 = month / 10; // First month digit
int d4 = month % 10; // Second month digit

int year = date.tm_year + 1900;
int d5 = year / 1000; // First year digit
int d6 = (year / 100) % 10; // Second year digit
int d7 = (year / 10) % 10; // Third year digit
int d8 = year % 10; // Fourth year digit
```

Нека обясним и един от по-интересните редове тук. Нека вземе за пример взимането на втората цифра от годината (**d6**). При нея делим годината на 100 и взимаме остатък от 10. Какво постигаме така? Първо с деленето на 100 отстраняваме последните 2 цифри от годината (пример: **2018 / 100 = 20**). С

остатъка от деление на 10 взимаме последната цифра на полученото число (**20 % 10 = 0**) и така получаваме 0, което е втората цифра на 2018.

Остава да направим изчислението, което ще ни даде магическото тегло на дадена дата. За да не изписваме всички умножения, както е показано в примера, ще приложим просто групиране. Това, което трябва да направим, е да умножим всяка цифра с тези, които са след нея. Вместо да изписваме **d1 * d2 + d1 * d3 + ... + d1 * d8**, може да съкратим този израз до **d1 * (d2 + d3 + ... + d8)**, следвайки математическите правила за групиране, когато имаме умножение и събиране. Прилагайки същото опростяване за останалите умножения, получаваме следната формула:

```
int weight = d1 * (d2 + d3 + d4 + d5 + d6 + d7 + d8) +
             d2 * (d3 + d4 + d5 + d6 + d7 + d8) +
             d3 * (d4 + d5 + d6 + d7 + d8) +
             d4 * (d5 + d6 + d7 + d8) +
             d5 * (d6 + d7 + d8) +
             d6 * (d7 + d8) +
             d7 * d8;
```

Отпечатване на изхода

След като имаме пресметнатото теглото на дадена дата, трябва да проверим дали съвпада с търсеното от нас магическо тегло, за да знаем, дали трябва да се принтира или не. Проверката може да се направи със стандартен **if** оператор, като трябва да се внимава при принтирането датата да е в правилният формат. За целта ще се възползваме от друг помощен метод в езика C++ - **put_time(...)**, който взима дата от тип **tm** и символен формат (в нашия случай **%d - %m - %Y**) и изкарва данните за датата в посочения формат (в нашия случай **дата-месец-година**):

```
if (weight == numberToSearchFor) {
    cout << put_time(&date, "%d-%m-%Y") << endl;
    found = true;
}
```

Внимание: тъй като обхождаме датите от началната към крайната, те винаги ще бъдат подредени във възходящ ред, както е по условие.

И накрая, ако не сме намерили нито една дата, отговаряща на условията, ще имаме **false** стойност във променливата **found** и ще отпечатаме **"No"**:

```
if (!found) {
    cout << "No" << endl;
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1372#1>.

Задача: пет специални букви

Дадени са две числа: **начало** и **край**. Напишете програма, която генерира всички комбинации от 5 букви, всяка измежду множеството `{'a', 'b', 'c', 'd', 'e'}`, така че "теглото" на тези 5 букви да е число в интервала **[начало ... край]**, включително. Принтирайте ги по азбучен ред, на един ред, разделени с интервал.

Теглото на една буква се изчислява по следния начин:

```
weight('a') = 5;
weight('b') = -12;
weight('c') = 47;
weight('d') = 7;
weight('e') = -32;
```

Теглото на редицата от букви **c1, c2, ..., cn** е изчислено, като се премахват всички букви, които се повтарят (от дясно наляво), и след това се пресметне формулата:

$$\text{weight}(c_1, c_2, \dots, c_n) = 1 * \text{weight}(c_1) + 2 * \text{weight}(c_2) + \dots + n * \text{weight}(c_n)$$

Например, теглото на **bcd dc** се изчислява по следния начин:

Първо премахваме повтарящите се букви и получаваме **bcd**. След това прилагаме формулата: $1 * \text{weight}('b') + 2 * \text{weight}('c') + 3 * \text{weight}('d') = 1 * (-12) + 2 * 47 + 3 * 7 = 103$.

Друг пример: $\text{weight}("cadae") = \text{weight}("cade") = 1 * 47 + 2 * 5 + 3 * 7 + 4 * (-32) = -50$.

Входни данни

Входните данни се четат от конзолата. Състоят се от два реда:

- Числото за **начало** е на първия ред.
- Числото за **край** е на втория ред.

Входните данни винаги ще бъдат валидни и винаги ще са в описания формат. Няма нужда да се проверяват.

Изходни данни

Резултатът трябва да бъде принтиран на конзолата като поредица от низове, подредени по азбучен ред. Всеки низ трябва да бъде отделен от следващия с едно

разстояние. Ако теглото на нито един от 5-буквените низове не съществува в зададения интервал, да се принтира "No".

Ограничения

- Числата за **начало и край** ще бъдат цели числа в диапазона [-10000 ... 10000].
- Позволено работно време за програмата: 0.5 секунди.
- Позволена памет: 16 MB.

Примерен вход и изход

Вход	Изход	Обяснения	Вход	Изход
40 42	bcead bdcea	weight("bcead") = 41 weight("bdcea") = 40	300 400	No

Вход	Изход	Вход	Изход
-1 1	bcdea cebda eaaad eaada eaadd eaade eaaed eadaa eadad eadae eadda eaddd eadde eadea eaded eadee eaead eaeda eaedd eaede eaeed eeaad eeada eeadd eeade eeaed eeead	200 300	baadc babdc badac badbc badca badcb badcc badcd baddc bbadc bbdac bdaac bdabc bdaca bdacb bdacc bdacd bdadc bdbac bddac beadc bedac eabdc ebadc ebdac edbac

Насоки и подсказки

Като всяка задача, започваме решението с **обработване на входните данни**:

```
int firstNumber, secondNumber;
cin >> firstNumber >> secondNumber;
```

В задачата имаме няколко основни момента - **генерирането на всички комбинации с дължина 5** включващи 5-те дадени букви, **премахването на повтарящите се букви и пресмятането на теглото** за дадена вече опростена дума. Отговорът ще се състои от всяка дума, чието тегло е в дадения интервал [**firstNumber, secondNumber**].

Генериране на всички комбинации

За да генерираме **всички комбинации с дължина 1** използвайки 5 символа, бихме използвали **цикъл от 0..4**, като всяко число от цикъла ще искаем да отговаря на един символ. За да генерираме **всички комбинации с дължина 2** използвайки 5 символа (т.е. "aa", "ab", "ac", ..., "ba", ...), бихме направили **два вложени цикъла**, всеки

обхождащ цифрите от 0 до 4, като отново ще направим, така че всяка цифра да отговаря на конкретен символ. Тази стълпка ще повторим 5 пъти, така че накрая да имаме 5 вложени цикъла с индекси **i1, i2, i3, i4 и i5**:

```
for (int i1 = 0; i1 < 5; i1++) {
    for (int i2 = 0; i2 < 5; i2++) {
        for (int i3 = 0; i3 < 5; i3++) {
            for (int i4 = 0; i4 < 5; i4++) {
                for (int i5 = 0; i5 < 5; i5++) {
```

Имайки всички 5-цифрени комбинации, трябва да намерим начин да "превърнем" петте цифри в дума с буквите от 'a' до 'e'. Един от начините да направим това е, като си предефинираме стринг променлива, съдържаща буквите, които имаме:

```
string pattern = "abcde";
```

и за всяка цифра взимаме буквата от конкретната позиция. По този начин числото 00000 ще стане "aaaaa", числото 02423 ще стане "acecd". Можем да направим стринга от 5 букви по следния начин:

```
string fullWord;
fullWord += pattern[i1];
fullWord += pattern[i2];
fullWord += pattern[i3];
fullWord += pattern[i4];
fullWord += pattern[i5];
```

Друг начин: можем да преобразуваме цифрите до букви, използвайки подредбата им в ASCII таблицата. Изразът (**char**)((**int**)'a' + **i**) ще ни даде резултата 'a' при **i = 0**, 'b' при **i = 1**, 'c' при **i = 2** и т.н.

Така вече имаме генериирани всички 5-буквени комбинации и можем да продължим със следващата част от задачата.

Внимание: тъй като сме подбрали **pattern**, съобразен с азбучната подредба на буквите и циклите се въртят по подходящ начин, алгоритъмът ще генерира думите в азбучен ред и няма нужда от допълнително сортиране преди извеждане.

Премахването на повтарящи се букви

След като имаме вече готовия низ, трябва да премахнем всички повтарящи се символи. Ще направим тази операция, като **добавяме буквите от ляво надясно в нов низ и всеки път преди да добавим буква ще проверяваме дали вече я има** - ако я има ще я пропускаме, а ако я няма ще я добавяме. За начало ще добавим първата буква към началния символен низ (string):

```
string word;
word += pattern[i1];
```

След това ще направим същото и с останалите 4, проверявайки всеки път дали ги има чрез метода **find(...)**. Когато методът **find(...)** върне отрицателното число **-1**, това означава, че символният низ, който търсим не е намерен. За служебната стойност "не е намерено" ползваме **string::npos**. Тази част от кода можем да имплементираме и с цикъл по символите на **fullWord**, но това ще оставим на читателя за упражнение. По-долу е показан по-мързеливия начин с copy-paste:

```
if (word.find(pattern[i2]) == string::npos) {
    word += pattern[i2];
}

if (word.find(pattern[i3]) == string::npos) {
    word += pattern[i3];
}

if (word.find(pattern[i4]) == string::npos) {
    word += pattern[i4];
}

if (word.find(pattern[i5]) == string::npos) {
    word += pattern[i5];
}
```

Методът **find(...)** връща индекса на конкретния елемент, ако бъде намерен или **-1**, ако елементът не бъде намерен. Следователно всеки път, когато получим **-1**, ще означава, че все още нямаме тази буква в новия низ с уникални букви и можем да я добавим, а ако получим стойност различна от **-1**, ще означава, че вече имаме буквата и няма да я добавяме.

Пресмятане на теглото

Пресмятането на теглото е просто **обхождане** на **уникалната дума (word)**, получена в миналата стъпка, като за всяка буква трябва да вземем теглото ѝ и да я умножим по позицията. За всяка буква в обхождането трябва да пресметнем с каква стойност ще умножим позицията ѝ, например чрез използването на поредица от **if** конструкции:

```

int weight = 0;
for (int i = 0; i < word.length(); i++) {
    int multiplier = 0;

    if (word[i] == 'a') {
        multiplier = 5;
    }

    if (word[i] == 'b') {
        multiplier = -12;
    }

    if (word[i] == 'c') {
        multiplier = 47;
    }

    if (word[i] == 'd') {
        multiplier = 7;
    }

    if (word[i] == 'e') {
        multiplier = -32;
    }

    weight += multiplier * (i + 1);
}

```

След като имаме стойността на дадената буква, следва да я **умножим по позицията ѝ**. Тъй като индексите в стринга се различават с 1 от реалните позиции, т.е. индекс 0 е позиция 1, индекс 1 е позиция 2 и т.н., ще добавим 1 към индексите. Затова на последния ред от цикъла имаме израза:

```
weight += multiplier * (i + 1);
```

Всички получени междинни резултати трябва да бъдат добавени към **обща сума (weight)** за всяка една буква от 5-булевната комбинация.

Оформяне на изхода

Дали дадена дума трябва да се принтира, се определя по нейната тежест. Трябва ни условие, което да определи дали **текущата тежест е в интервала [начало ... край]**, подаден ни на входа в началото на програмата. Ако това е така, принтираме **пълната дума (fullWord)**.

Внимавайте да не принтирате думата от уникални букви (**word**), тя ни бе

необходима само за пресмятане на тежестта!

Думите са **разделени с интервал** и ще ги натрупваме в междинна променлива **result**, която е дефинирана като празен низ в началото:

```
if (firstNumber <= weight && weight <= secondNumber) {  
    result += fullWord + " ";  
}
```

Финални щрихи

Условието е изпълнено **с изключение случаите, в които нямаме нито една дума в подадения интервал**. За да разберем дали сме намерили такава дума, можем просто да проверим дали низът **result** има началната си стойност (а именно празен низ), чрез метода **empty()** и ако е така - отпечатваме **"No"**, иначе печатаме целия низ без последния интервал. Махането на последния интервал от символния низ **result** можем да направим чрез метода **substr(...)**, който ще ни върне подниза, започващ от 0 и имащ дължина: дължината на символния низ минус 1.

```
if (result.empty()) {  
    cout << "No" << endl;  
}  
else {  
    cout << result.substr(0, result.size() - 1) << endl;  
}
```

За по-любознателните читатели ще оставим за домашно да оптимизират работата на циклите като измислят начин да намалят броя итерации на вътрешните цикли.

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1372#2>.

Глава 9.2. Задачи за шампиони – част II

В тази глава ще разгледаме още три задачи, които причисляваме към категорията "за шампиони", т.е. по-трудни от стандартните задачи в тази книга.

По-сложни задачи върху изучавания материал

Преди да преминем към конкретните задачи, трябва да поясним, че те могат да се решат по-лесно с **допълнителни знания за програмирането и езика C++** (функции, масиви, рекурсия и т.н.), но всяко едно решение, което ще дадем сега, ще използва единствено материал, покрит в тази книга. Целта е да се научите да съставяте **по-сложни алгоритми** на базата на сегашните си знания.

Задача: дни за страстно пазаруване

Лина има истинска страст за пазаруване. Когато тя има малко пари, веднага отива в първия голям търговски център (мол) и се опитва да изхарчи възможно най-много за дрехи, чанти и обувки. Но любимото ѝ нещо са зимните намаления. Нашата задача е да анализираме странното ѝ поведение и да **изчислим покупките**, които Лина прави, когато влезе в мола, както и парите, които ѝ остават, когато приключи с пазаруването си.

На **първия ред** от входа ще бъде подадена **сумата**, която Лина има **преди** да започне да пазарува. След това при получаване команда "mall.Enter", Лина влиза в мола и започва да пазарува, докато не получи команда "mall.Exit". Когато Лина започне да пазарува, на **всяка линия** от входа ще получите стрингове, които представляват **действия**, които Лина **изпълнява**. Всеки **символ** в стринга представлява **покупка или друго действие**. Стинговите команди могат да съдържат само символи от ASCII таблицата. ASCII кода на всеки знак има **връзка** с това колко Лина трябва да плати за всяка стока. Интерпретирайте символите по следния начин:

- Ако символът е **главна буква**, Лина получава **50% намаление**, което означава, че трябва да намалите парите, които тя има, с 50% от цифровата репрезентация на символа от ASCII таблицата.
- Ако символът е **малка буква**, Лина получава **70% намаление**, което означава, че трябва да намалите парите, които тя има, с 30% от цифровата репрезентация на символа от ASCII таблицата.
- Ако символът е **"%"**, Лина прави **покупка**, която намалява парите ѝ на половина.
- Ако символът е **"*"**, Лина **изтегля пари от дебитната си карта** и добавя към наличните си средства 10 лева.
- Ако символът е **различен от упоменатите горе**, Лина просто прави покупка без намаления и в такъв случай просто извадете стойността на символа от ASCII таблицата от наличните ѝ средства.

Ако някоя от стойностите на покупките е **по-голяма** от текущите налични средства, Лина **НЕ** прави покупката. Парите на Лина **не могат да бъдат по-малко от 0**.

Пазаруването завършва, когато се получи команда **"`mall.Exit`"**. Когато това стане, трябва да се **принтират** броя на **извършени покупки** и парите, които са останали на Лина.

Входни данни

Входните данни трябва да се четат от конзолата. На **първия ред** от входа ще бъде подадена **сумата**, която Лина има **преди да започне да пазарува**. На всеки следващ ред ще има определена команда. Когато получите команда **"`mall.Enter`"**, на всеки следващ ред ще получавате стрингове, съдържащи **информация относно покупките / действията**, които Лина иска да направи. Тези стрингове ще продължат да бъдат подавани, докато не се получи команда **"`mall.Exit`"**.

Винаги ще се подава само една команда **"`mall.Enter`"** и само една команда **"`mall.Exit`"**.

Изходни данни

Когато пазаруването приключи, на конзолата трябва да се принтира определен изход в зависимост от това какви покупки са били направени:

- Ако **не са били направени никакви покупки** – "No purchases. Money left: {останали пари} lv."
- Ако е направена **поне една покупка** - "{брой покупки} purchases. Money left: {останали пари} lv."

Парите трябва да се принтират с **точност от 2 символа** след десетичния знак.

Ограничения

Задължително е бъдат спазени следните изисквания при изпълнението на поставената задача:

- Парите са число с **плаваща запетая** в интервала: $[0 - 7.9 \times 10^{28}]$.
- Броят стрингове между **"`mall.Enter`"** и **"`mall.Exit`"** ще е в интервала: $[1 - 20]$.
- Броят символи във всеки стринг, който представлява команда, ще е в интервала: $[1 - 20]$.
- Позволено време за изпълнение: **0.1 секунди**.
- Позволена памет: **16 MB**.

Примерен вход и изход

Вход	Изход	Коментар	
110 mall.Enter d mall.Exit	1 purchases. Money left: 80.00 lv.	'd' има ASCII код 100. 'd' е малка буква и за това Лина получава 70% отстъпка и така тя харчи $30\% * 100 = 30$ лв. След покупката ѝ остават $110 - 30 = 80$ лв.	
Вход	Изход	Вход	Изход
110 mall.Enter % mall.Exit	1 purchases. Money left: 55.00 lv.	100 mall.Enter Ab ** mall.Exit	2 purchases. Money left: 58.10 lv.

Насоки и подсказки

Ще разделим решението на задачата на три основни части:

- **Обработка** на входа.
- **Алгоритъм** на решаване.
- **Форматиране** на изхода.

Нека разгледаме всяка една част в детайли.

Обработване на входа

Входът за нашата задача се състои от няколко компонента:

- На **първия ред** имаме **всички пари**, с които Лина ще разполага за пазаруването.
- На **всеки следващ ред** ще имаме някакъв вид **команда**.

Първата част от прочитането е тривиална:

```
double shoppingMoney;
cin >> shoppingMoney;
```

Но във втората част има детайл, с който трябва да се съобразим. Условието гласи следното:

Всеки следващ ред ще има определена команда. Когато получите командата "mall.Enter", на всеки следващ ред ще получите стрингове, съдържащи информация относно покупките/действията, които Лина иска да направи.

Тук е моментът, в който трябва да се съобразим, че от **втория ред** нататък трябва да започнем да четем команди, но едва след като получим командата "mall.Enter",

трябва да започнем да ги обработваме. Как можем да направим това? Използването на **while** или **do-while** цикъл е добър избор. Ето примерно решение как можем да пропуснем всички команди преди получаване на командата "mall.Enter":

```
string command;
cin >> command;

while (command != "mall.Enter") {
    cin >> command;
}

cin >> command;
```

Тук е мястото да отбележим, че извикването на **cin** след края на цикъла се използва за **преминаване към първата команда** за обработване.

Алгоритъм за решаване на задачата

Алгоритъмът за решаването на самата задача е праволинеен - продължаваме да четем команди от конзолата, докато не бъде подадена команда "mall.Exit". През това време обработваме всеки един ASCII знак (**char**) от всяка една команда спрямо правилата, указанi в условието, и едновременно с това модифицираме парите, които Лина има, и съхраняваме броя на покупките.

Нека разгледаме първите два проблема пред нашия алгоритъм. Първият проблем засяга начина, по който можем да четем командите, докато не срещнем "mall.Exit". Решението, както видяхме по-горе, е да се използва **while** цикъл. Вторият проблем е задачата да достъпим всеки един знак от подадената команда. Имайки предвид, че входните данни с командите са от тип **string**, то най-лесният начин да достъпим всеки знак в тях е чрез **for** цикъл:

Ето как би изглеждало използване на два такива цикъла:

```
while (command != "mall.Exit") {
    for (char action : command) {
        }

    cin >> command;
}
```

Следващата част от алгоритъма ни е да обработим символите от командите, спрямо следните правила от условието:

- Ако символът е **главна буква**, Лина получава 50% намаление, което означава, че трябва да намалите парите, които тя има, с 50% от цифровата репрезентация ASCII символа.

- Ако символът е **малка буква**, Лина получава 70% намаление, което означава, че трябва да намалите парите, които тя има, с 30% от цифровата репрезентация ASCII символа.
- Ако символът е **"%"**, Лина прави покупка, която намалява парите ѝ на половина.
- Ако символът е **"**"**, Лина изтегля пари от дебитната си карта и добавя към наличните си средства 10 лева.
- Ако символът е **различен от упоменатите горе**, Лина просто прави покупка без намаления и в такъв случай просто извадете стойността на ASCII символа от наличните ѝ средства.

Нека разгледаме проблемите от първото условие, които стоят пред нас. Единият е как можем да разберем дали даден **символ представлява главна буква**. Можем да използваме следния начин:

- Имайки предвид, факта, че буквите в азбуката имат ред, можем да използваме следната проверка, за да проверим дали нашият символ се намира в интервала от големи букви: **(action >= 'A') && (action <= 'Z')**.

Другият проблем е как можем **да пропуснем даден символ**, ако той представлява операция, която изисква повече пари, отколкото Лина има? Това е възможно да бъде направено чрез използване на **continue** конструкцията.

Примерната проверка за първата част от условието изглежда по следния начин:

```
if ((action >= 'A') && (action <= 'Z')) {
    double price = action * 0.5;
    if (shoppingMoney < price) {
        continue;
    }

    shoppingMoney -= price;
    purchases++;
}
```

Забележка: **purchases** е променлива от тип **int**, в която държим броя на всички покупки.

Смятаме, че читателят не би трябвало да изпита проблем при имплементацията на всички други проверки, защото са много сходни с първата.

Форматиране на изхода

В края на задачата трябва да **принтираме** определен **изход**, в зависимост от следното условие:

- Ако не са били направени никакви покупки – "No purchases. Money left: {останали пари} lv."
- Ако е направена поне една покупка - "{брой покупки} purchases. Money left: {останали пари} lv."

Операциите по принтиране са тривиални, като единственото нещо, с което трябва да се съобразим е, че **парите трябва да се принтират с точност от 2 символа след десетичния знак**.

Как можем да направим това? Ще оставим отговора на този въпрос на читателя.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1373#0>.

Задача: Числен израз

Бони е изключително могъща вещица. Тъй като силата на природата не е достатъчна, за да се бори успешно с вампири и върколаци, тя започнала да усвоява силата на Изразите. Изразът е много труден за усвояване, тъй като заклинанието разчита на способността за **бързо решаване на математически изрази**.

За използване на "Израз заклинание", вещицата трябва да знае резултата от математически израз предварително. **Израз заклинанието** се състои от няколко прости математически израза. Всеки математически израз може да съдържа оператори за **събиране, изваждане, умножение и/или деление**.

Изразът се решава **без да се вземат под внимание математическите правила при пресмятане на числови изрази**. Това означава, че приоритет има последователността на операторите, а не това какъв вид изчисление правят. Израза **може да съдържа скоби**, като **всичко в скобите се пресмята първо**. Всеки израз може да съдържа множество скоби, но не може да съдържа вложени скоби:

- Израз съдържащ (...(...)...) е невалиден.
- Израз съдържащ (...)...(...) е валиден.

Пример

Изразът

$4 + 6 / + (4 * 9 - 8) / 7 * 2$

бива решен по следния начин:

$$\begin{aligned}
 & 4 + 6 / 5 + (4 * 9 - 8) / 7 * 2 = \\
 & 10 / 5 + (4 * 9 - 8) / 7 * 2 = \\
 & 2 + (4 * 9 - 8) / 7 * 2 = \\
 & 2 + (36 - 8) / 7 * 2 = \\
 & 2 + 28 / 7 * 2 = \\
 & 30 / 7 * 2 = \\
 & 4.285714285714286 * 2 = \\
 & 8.5712857142571 = \\
 & 8.57
 \end{aligned}$$

Бони е много красива, но не чак толкова съобразителна, затова тя има нужда от нашата помощ, за да усвои силата на Изразите.

Входни данни

Входните данни се състоят от един ред, който бива подаван от конзолата. Той съдържа **математическият израз за пресмятане**. Редът винаги завършва със символа "**=**". Символът "**=**" означава **край на математическия израз**.

Входните данни винаги са валидни и във формата, който е описан. Няма нужда да бъдат валидирани.

Изходни данни

Изходните данни трябва да се принтират на конзолата. Изходът се състои от един ред – резултата от **пресметнатия математически израз**.

Резултатът трябва да бъде **закръглен до втората цифра след десетичния знак**.

Ограничения

- Изразите ще се състоят от **максимум 2500 символа**.
- Числата от всеки математически израз ще са в интервала **[1 ... 9]**.
- Операторите в математическите изрази винаги ще бъдат измежду **+** (събиране), **-** (изваждане), **/** (деление) или ***** (умножение).
- Резултатът от математическия израз ще е в интервала **[-100000.00 ... 100000.00]**.
- Позволено време за изпълнение: **0.1 секунди**.
- Позволена памет: **16 MB**.

Примерен вход и изход

Вход	Изход	Вход	Изход
$4+6/5+(4*9-8)/7*2=$	8.57	$3+(6/5)+(2*3/7)*7/2*(9/4+4*1)=$	110.63

Насоки и подсказки

Както обикновено, първо ще прочетем и обработим входа, след това ще решим задачата и накрая ще отпечатаме резултата, форматиран, както се изиска в условието.

Обработване на входа

Входните данни се състоят от точно един ред от конзолата. Тук имаме **два начина**, по които можем да обработим входа. Първият е чрез **прочитането на целия ред с**

командата `cin` и достъпването на всеки един символ (`char`) от реда чрез `for` цикъл. Вторият е чрез прочитане на входа символ по символ чрез командата `cin` и обработване на всеки символ.

За решаване на задачата ще използваме втория вариант.

```
int symbol;
cin >> symbol;
```

Алгоритъм за решаване на задачата

За целите на нашата задача ще имаме нужда от две променливи:

- Една променлива, в която ще пазим **текущия резултат**.
- Още една променлива, в която ще пазим **текущия оператор** от нашия израз.

```
double result = 0;
int expressionOperator = '+';
```

Относно кода по-горе трябва да поясним, че стойността по подразбиране на оператора е `+`, за да може още първото срещнато число да бъде събрано с резултата ни.

След като вече имаме началните променливи, трябва да помислим върху това, каква ще е основната структура на нашата програма. От условието разбираме, че **всеки израз завършва с `=`**, т.е. ще трябва да четем и обработваме символи, докато не срещнем `=`. Следва точното изписване на **while цикъл**:

```
while (symbol != '=') {
    cin >> symbol;
}
```

Следващата стъпка е обработването на нашата `symbol` променлива. За нея имаме 3 възможни случая:

- Ако символът е **начало на подизраз, заграден в скоби**, т.е. срещнатият символ е `(`.
- Ако символът е **цифра между 0 и 9**. Но как можем да проверим това? Как можем да проверим дали символът ни е цифра? Тук идва на помощ ASCII кодът на символа, чрез който можем да използваме следната формула: **[ASCII кода на нашия символ] - [ASCII кода на символа 0] = [цифрата, която репрезентира символа]**. Ако **результатът от тази проверка е между 0 и 9**, то тогава нашият символ наистина е **число**.
- Ако символът е **оператор**, т.е. е `+`, `-`, `*` или `/`.

```
if (symbol == '(') {
```

```

else if ((0 <= symbol - '0') && (symbol - '0' <= 9)) {
}
else if (symbol == '+' ||
    symbol == '-' ||
    symbol == '/' ||
    symbol == '*') {
}

```

Нека разгледаме действията, които трябва да извършим при съответните случаи, които дефинирахме:

- Ако нашият символ е **оператор**, то тогава единственото, което трябва да направим, е да зададем нова стойност на променливата **expressionOperator**.
- Ако нашият символ е **цифра**, тогава трябва да променим текущия резултат от израза в зависимост от текущия оператор, т.е. ако **expressionOperator** е `-`, тогава трябва да намалим резултата с цифровата репрезентация на **текущия символ**. Можем да вземем цифровата репрезентация на текущия символ, чрез формулата, която използвахме при проверката на този случай (**[ASCII кода на нашия символ] - [ASCII кода на символа 0] = [цифрата, която репрезентира символа]**).

```

else if (0 <= symbol - '0' && symbol - '0' <= 9) {
    switch (expressionOperator) {
        case '+':
            result += symbol - '0';
            break;
        case '-':
            result -= symbol - '0';
            break;
        case '*':
            result *= symbol - '0';
            break;
        case '/':
            result /= symbol - '0';
            break;
    }
}
else if (symbol == '+' ||
    symbol == '-' ||
    symbol == '/' ||
    symbol == '*') {
    expressionOperator = symbol;
}

```

- Ако нашият символ е `(`, това индицира **началото на подизраз** (израз в скоби). По дефиниция подизразът трябва да се калкулира преди да се модифицира резултата от **целия израз** (действията в скобите се извършват първи). Това означава, че ще имаме локален резултат за подизраза и локален оператор.

```
if (symbol == '(') {
    double innerResult = 0;
    int innerOperator = '+';
    cin >> symbol;
}
```

След това, за **пресмятане стойността на подизраза** използваме същите методи, които използвахме за пресмятане на главния израз - използваме **while цикъл**, за да четем **символи** (докато не срещнем символ `)`). В зависимост от това дали прочетения символ е цифра или оператор, модифицираме резултата на подизраза. Имплементацията на тези операции е аналогична с имплементацията за пресмятане на изрази, описана по-горе, затова смятаме, че читателят не би трявало да има проблем с нея.

След като приключим калкулацията на резултата от подизраза ни, **модифицираме резултата на целия израз** в зависимост от стойността на **expressionOperator**.

```
switch (expressionOperator) {
    case '+':
        result += innerResult;
        break;
    case '-':
        result -= innerResult;
        break;
    case '*':
        result *= innerResult;
        break;
    case '/':
        result /= innerResult;
        break;
}
```

Форматиране на изхода

Единствения изход, който програмата трябва да принтира на конзолата, е **резултатът от решаването на израза, с точност два символа след десетичния знак**. Как можем да форматираме изхода по този начин? Отговора на този въпрос оставяме на читателя.

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1373#1>.

Задача: бикове и крави

Всички знаем играта „Бикове и крави“ - https://en.wikipedia.org/wiki/Bulls_and_cows

При дадено 4-цифreno тайно число и 4-цифreno предполагаемо число, използваме следните правила:

- Ако имаме цифра от предполагаемото число, която съвпада с цифра от тайното число и е на **същата позиция**, имаме **бик**.
- Ако имаме цифра от предполагаемото число, която съвпада с цифра от тайното число, но е на **различна позиция**, имаме **крава**.

Тайно число	1	4	8	1	Коментар
Предполагаемо число	8	8	1	1	Бикове = 1 Крави = 2
Тайно число	2	2	4	1	Коментар
Предполагаемо число	9	9	2	4	Бикове = 0 Крави = 2

При дадено тайно число и брой на бикове и крави, нашата задача е **да намерим всички възможни предполагаеми числа** в нарастващ ред. Ако **не съществуват предполагаеми числа**, които да отговарят на зададените критерии на конзолата, трябва да се отпечата "No".

Входни данни

Входните данни се четат от конзолата. Входът се състои от 3 реда:

- Първият ред съдържа **секретното число**.
- Вторият ред съдържа **броя бикове**.
- Третият ред съдържа **броя крави**.

Входните данни ще бъдат винаги валидни. Няма нужда да бъдат проверявани.

Изходни данни

Изходните данни трябва да се принтират на конзолата. Изходът трябва да се състои от **един единствен ред** – **всички предполагаеми числа**, разделени с единично празно място. Ако **не съществуват предполагаеми числа**, които да отговарят на зададените критерии на конзолата, трябва **да се изпише "No"**.

Ограничения

- Тайното число винаги ще се състои от **4 цифри в интервала [1 ... 9]**.
- Броят на **кравите и биковете** винаги ще е в интервала **[0 ... 9]**.
- Позволено време за изпълнение: **0.15 секунди**.
- Позволена памет: **16 MB**.

Примерен вход и изход

Вход	Изход
2228 2 1	1222 2122 2212 2232 2242 2252 2262 2272 2281 2283 2284 2285 2286 2287 2289 2292 2322 2422 2522 2622 2722 2821 2823 2824 2825 2826 2827 2829 2922 3222 4222 5222 6222 7222 8221 8223 8224 8225 8226 8227 8229 9222
Вход	Изход
1234 3 0	1134 1214 1224 1231 1232 1233 1235 1236 1237 1238 1239 1244 1254 1264 1274 1284 1294 1334 1434 1534 1634 1734 1834 1934 2234 3234 4234 5234 6234 7234 8234 9234

Насоки и подсказки

Ще решим задачата на няколко стъпки:

- Ще прочетем **входните данни**.
- Ще генерираме всички възможни **четирицифрени комбинации** (кандидати за проверка).
- За всяка генерирана комбинация ще изчислим **колко бика и колко крави** има в нея спрямо секретното число. При съвпадение с търсените бикове и крави, ще **отпечатаме комбинацията**.

Обработване на входа

За входа на нашата задача имаме 3 реда:

- Секретното число.
- Броят желани бикове.
- Броят желани крави.

Прочитането на тези входни данни е тривиално:

```
int guessNumber;
cin >> guessNumber;
```

```

int targetBulls;
cin >> targetBulls;

int targetCows;
cin >> targetCows;

```

Алгоритъм за решаване на задачата

Преди да започнем писането на алгоритъма за решаване на нашия проблем, трябва да **декларираме флаг**, който да указва дали е намерено решение:

```
bool solutionFound = false;
```

Ако след приключването на нашия алгоритъм, този флаг все още е **false**, тогава ще принтираме **No** на конзолата, както е указано в условието.

```

if (!solutionFound) {
    cout << "No" << endl;
}

```

Нека започнем да размишляваме над нашия проблем. Това, което трябва да направим, е да **анализираме всички числа от 1111 до 9999** без тези, които съдържат в себе си нули (напр. **9011, 3401** и т.н. са невалидни числа). Какъв е най-лесният начин за **генериране** на всички тези **числа?** С **вложени цикли.** Тъй като имаме **4-цифрен**о число, ще имаме **4 вложени цикъла**, като всеки един от тях ще генерира **отделна цифра** от нашето число за тестване:

```

for (int digit1 = 1; digit1 <= 9; digit1++) {
    for (int digit2 = 1; digit2 <= 9; digit2++) {
        for (int digit3 = 1; digit3 <= 9; digit3++) {
            for (int digit4 = 1; digit4 <= 9; digit4++) {

```

Благодарение на тези цикли, **имаме достъп до всяка една цифра** на всички числа, които трябва да проверим. Следващата ни стъпка е да **разделим секретното число на цифри.** Това може да се постигне много лесно чрез **комбинация от целочислено и модулно деление:**

```

int guessDigit1 = (guessNumber / 1000) % 10;
int guessDigit1 = (guessNumber / 100) % 10;
int guessDigit1 = (guessNumber / 10) % 10;
int guessDigit1 = (guessNumber / 1) % 10;

```

Остават ни последните две стъпки преди да започнем да анализираме колко крави и бикове има в дадено число. Съответно, първата е **декларацията на counter променливи** във вложените ни цикли, за да **броим** кравите и биковете за текущото число. Втората стъпка е да направим **копия на цифрите** на текущото

ЧИСЛО, което ще анализираме, за да предотвратим евентуални проблеми с работата на вложите цикли (напр., ако правим промени по цифрите):

```
int digitToCheck1 = digit1;
int digitToCheck2 = digit2;
int digitToCheck3 = digit3;
int digitToCheck4 = digit4;

int currentBulls = 0;
int currentCows = 0;
```

Вече сме готови да започнем анализирането на генерираните числа. Каква логика можем да използваме? Най-елементарният начин да проверим колко крави и бикове има в едно число е чрез **поредица от if-else проверки**. Да, не е най-оптималният начин, но с цел да не използваме знания извън пределите на тази книга, ще изберем този подход.

От какви проверки имаме нужда?

Проверката за бикове е елементарна - проверяваме дали **първата цифра** от генерираното число е еднаква със **същата цифра** от секретното число. Премахваме проверените цифри с цел да избегнем повторения на бикове и крави.

```
//Find all bulls, count them and remove them (assign -1 and -2)
if (digitToCheck1 == guessDigit1) {
    //Bull at position #1 found -> count it and remove it
    currentBulls++;
    guessDigit1 = -1;
    digitToCheck1 = -2;
}
```

Повтаряме действието за втората, третата и четвърта цифра.

Проверката за крави можем да направи по следния начин - първо проверяваме дали **първата цифра** от генерираното число **съвпада с втората, третата или четвъртата цифра** на секретното число. Примерна имплементация:

```
//Find all cows for digitToCheck1, count them and remove them (assign -1)
if (digitToCheck1 == guessDigit2) {
    //Cow at position #2 found -> count it and remove it
    currentCows++;
    guessDigit2 = -1;
}

else if (digitToCheck1 == guessDigit3) {
    //Cow at position #3 found -> count it and remove it
    currentCows++;
    guessDigit3 = -1;
}
```

```

else if (digitToCheck1 == guessDigit4) {
    //Cow at position #4 found -> count it and remove it
    currentCows++;
    guessDigit4 = -1;
}

```

След това последователно проверяваме дали **втората цифра** от генерираното число съвпада с първата, третата или четвъртата цифра на секретното число, дали **третата цифра** от генерираното число съвпада с първата, втората или **четвъртата цифра** на секретното число и накрая проверяваме дали **четвъртата цифра** от генерираното число съвпада с **първата**, **втората** или **третата цифра** на секретното число.

Отпечатване на изхода

След като приключим всички проверки, ни остава единствено да проверим дали биковете и кравите в текущото генерирано число съвпадат с желаните бикове и крави, прочетени от конзолата. Ако това е така, принтираме текущото число на конзолата:

```

if (currentBulls == targetBulls && currentCows == targetCows) {
    if (solutionFound) {
        cout << " ";
    }

    cout << digit1 << digit2 << digit3 << digit4;
    solutionFound = true;
}

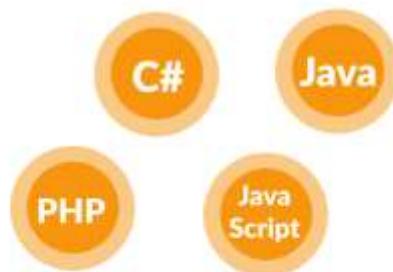
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1373#2>

Качествено образование,
професия и работа за
Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



СофТУни предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата **"Софтуерен университет"** изгражда истински **профессионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на СофТУни, за да усвоите **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

СофТУни работи пряко с компаниите от софтуерната индустрия, съдействайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ



1 - 1.5 години



1.5 - 2 години



Programming Basics

Кариерен Старт

Дипломиране

70/100 credits

80/100/150 credits

Кандидатствай

softuni.bg/apply

Глава 10. Функции

В настоящата глава ще се запознаем с **функциите** и ще научим какво **представляват** те, както и кои са базовите концепции при работа с тях. Ще научим защо е добра практика да ги използваме, как да ги **дефинираме** и **извикваме**. Ще се запознаем с параметри и връщана стойност от функция, както и как да използваме тази връщана стойност. Накрая на главата, ще разгледаме утвърдените практики при използването на функции.

Какво е "функция"?

До момента установихме, че при **писане** на програма, която решава даден проблем, ни **улесява** това, че **разделяме** задачата на **части**. Всяка част отговаря за **дадено действие** и по този начин не само ни е **по-лесно** да решим задачата, но и значително се подобрява както **четливостта** на кода, така и проследяването за грешки.

В контекста на програмирането, **функция** (метод) се нарича **именувана група от инструкции**, които изпълняват дадена операция (функционалност). Тази група от инструкции е логически отделена и именувана, така че изпълнението на инструкциите в групата може да бъде стартирано чрез извикване на нейното име в хода на изпълнението на програмата. Стартирането на изпълнението на инструкциите във функцията се нарича **извикване на функцията** (на английски function call или invoking a function).

Една функция може да бъде извикана толкова пъти, колкото ние преценим, че ни е нужно за решаване на даден проблем. Това ни **спестява** повторението на един и същи код няколко пъти, както и **намалява** възможността да пропуснем грешка при евентуална корекция на въпросния код.

Ще разгледаме два типа функции - "прости" (без параметри) и "сложни" (с параметри).

Прости функции

Простите функции отговарят за изпълнението на дадено **действие**, което **спомага** за решаване на определен проблем. Такива действия могат да бъдат например разпечатване на даден низ в конзолата, извършване на някаква проверка, изпълнение на цикъл и други.

Нека разгледаме следния пример за **проста функция**:

```
void printHeader() {  
    cout << "-----";  
}
```

Функцията обаче има още една част, независима от горната:

```
void printHeader();
```

За разликите между двете - структурата, употребата и значението, им ще коментираме подробно в главата.

Тази функция има задачата да отпечата заглавие, което представлява поредица от символа `-`. Поради тази причина името ѝ е **printHeader**. Кръглите скоби `()` винаги следват името, независимо как сме именували функцията. По-късно в тази глава ще разгледаме утвърдени практики за именуване на функции, а за момента ще отбележим само, че е важно **името да описва действието**, което тя извършва. Ще поговорим и за ключовата дума **void**.

Тялото на функцията, което се намира между къдрявите скоби `{ }` , съдържа **програмния код** (инструкциите), който решава проблема, описан от името ѝ. Тялото на функцията се изписва по-навътре, обикновено 4 интервала (една табулация), които го обособяват като отделен блок инструкции, прилежащи към функцията.

Зашто да използваме функции?

Дотук установихме, че функциите спомагат за **разделянето на обемна задача на по-малки части**, което води до **по-лесно решаване** на въпросното задание. Това прави програмата ни не само по-добре структурирана и лесно четима, но и по-разбираема.

Чрез функциите **избягваме повторението** на програмен код. **Повтарящият** се код е **лоша практика**, тъй като силно **затруднява поддръжката** на програмата и води до грешки. Ако дадена част от кода ни присъства в програмата няколко пъти и се наложи да променим нещо, то промените трябва да бъдат направени във всяко едно повторение на въпросния код. Вероятността да пропуснем място, на което трябва да нанесем корекция, е много голяма, което би довело до некоректно поведение на програмата. Това е причината, поради която е **добра практика**, ако използваме даден фрагмент код **повече от веднъж** в програмата си, да го **дефинираме като отделна функция**.

Функциите ни предоставят **възможността** да използваме **даден код няколко пъти**. С решаването на все повече и повече задачи ще установите, че използването на вече съществуващи функции спестява много време и усилия. Нещо повече - **всеки път, когато пишем програма на C++, използваме главната функция `main()`**.

Деклариране и дефиниция на функции

В езика за програмиране **C++** съществува разлика между понятията **декларация** и **дефиниция** на функция:

- **Декларацията** на функция **информира** компилатора или интерпретатора, че функцията със съответното име и параметри **съществува**, без да **съдържа имплементация** (тялото на функцията).
- **Дефиницията** на функция **съдържа** нейната **имплементация** (тялото ѝ).

Декларация

Нека разгледаме следния пример за **деклариране на функция**, намираща лицето на квадрат по зададена страна **num**:

```
double getSquare(double);
```

Следната декларация е **еквивалентна** на горната, като това ще коментираме по-надолу в главата:

```
double getSquare(double num);
```

Нека обърнем внимание на следното - даденият програмен фрагмент **само съобщава** на компилатора или интерпретатора, че в програмата **ще бъде дефинирана и използвана** функцията **getSquare()**.

Декларирането на една функция се осъществява **след using namespace std;** и преди главната функция **main()**. В примера тук са показани декларациите на функциите **getSquare(), multiply() и printHeader()**:

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

double getSquare(double num);
int multiply(int, int);
void printHeader();

int main() {
}
```

Задължителните елементи при **деклариране** на функция:

- **Тип на връщаната стойност.** В случая типът е **double**, което заявява, че функцията **getSquare(...)** ще върне резултат, който е от тип **double**. Връщаната стойност може да бъде както **int, double, char** и т.н., така и **void**. Ако типът е **void**, то това означава, че функцията **не връща** резултат, а само **изпълнява** дадена операция.
- **Име на функцията.** Името на функцията е **определеното от нас**, като не забравяме, че трябва да **описва операцията**, която е изпълнявана от кода в тялото ѝ. В примера името е **getSquare**, което ни указва, че задачата на тази функция е да изчисли лицето на квадрат.
- **Списък с параметри.** Декларира се между скобите **()**, които изписваме след името на функцията. Тук изброяваме поредицата от **параметри**, които тя ще използва, като се подразбира, че **могат** да бъдат от **различен** тип (**int, double, char** и т.н.). Интересното тук е, че може (дори обикновено така се прави) да се запише **само типът** на използвани параметри (т.е. да не се идентифицират). **Това е допустимо само при декларирането на функция.** Може да присъства **само един** параметър, **няколко** такива или да е

празен списък. Ако няма параметри, то ще запишем единствено скобите **()**. В конкретния пример декларираме параметъра **double num**.

- Накрая поставяме точка и запетая **;**.



Задължително се поставя знакът "точка и запетая" **;** в края на декларирането на дадена функция.

При декларацията на функции е важно да спазваме **последователността** на основните ѝ елементи - първо изписваме **типа на връщаната стойност**, след това **името на функцията** и **списък от параметри**, ограден с кръгли скоби **()**, и знакът **;** в края.

Дефиниция

След като сме декларирали функцията, следва нейната **дефиниция**. Дефиницията на функция представлява **описването** на **нейното тяло** (**нейната имплементация**). Тялото съдържа кода (програмен блок), който реализира **логиката** на функцията. В показания пример изчисляваме лицето на квадрат, а именно **num * num**. Това е дефиницията на декларираната по-горе функция **getSquare(...)**:

```
double getSquare(double num) {
    return num * num;
}
```

Дефинициите на функции се осъществяват след главната **main()** функция. След малко ще разгледаме цялостен пример.

Задължителните елементи при **дефиниране** на функция:

- **Тип на връщаната стойност.** Типът трябва да бъде **съобразен с типа** на връщаната стойност **при декларацията** на функцията, т.е. той трябва да е **същият**. Ако при декларацията е записано, че функцията ще връща стойност от тип **int**, то и при дефиницията ще трябва типът да е **int**.
- **Име на функция.** Името трябва да е **същото** като **при декларацията** на функцията.
- **Списък с параметри.** Декларира се между скобите **()**. Отново се съобразяваме със записаното **при декларацията** на функцията. Ако има повече от един **параметър**, то ги записваме в **същия ред**, както **при декларацията** на функцията. Например, ако сме декларирали функция **function(double, int, double, char)**, то при нейната дефиниция ще подредим параметрите в същата последователност - **double, int, double, char**. Освен това тук **задължително** трябва да се **идентифицират** въпросните **параметри** (за разлика от декларацията, където това можеше да се пропусне). Следователно примерна идентификация на параметрите ще изглеждала по следния начин: **(double var1, int var2, double var3, char var4)**.

- **Имплементация (тяло).** Записва се в **областта**, отделена от къдравите скоби { и }. Тази област се създава **след** списъка с параметри. В тялото на функцията описваме **алгоритъма** (инструкциите), по който тя решава даден проблем, т.е. тялото съдържа кода, който реализира **логиката** на функцията. Ако функцията връща стойност, то в края на имплементацията ще добавим оператор **return**, а след него това, което ще връщаме, и знак ;. За връщащи и невръщащи стойности ще говорим по-нататък.
- Тук **не се поставя** ; в края, както при декларирането.

Видове променливи

Когато декларираме дадена променлива **в тялото на една функция**, я наричаме **локална** променлива за функцията. Областта, в която съществува и може да бъде използвана тази променлива, започва от реда, на който сме я декларирали и стига до затварящата къдрава скоба } на тялото на функцията. Тази област се нарича **област на видимост** на променливата (variable scope). Когато декларираме дадена променлива **външно от всички функции** в дадена програма (включително и от **main()** функцията), я наричаме **глобална** променлива за програмата. Тя може да бъде достъпвана във всяка част на програмния код.

Параметри на функции

Това са данните (променливи или дори константи), които се подават на функцията по време на нейното извикване, за да ги обработи, връщайки стойност или изпълнявайки някакво действие с тях. Трябва да се внимава параметрите да бъдат подредени в **една и съща последователност** при декларирането и дефинирането на функцията.

Извикване на функции

Извикването на функция представлява **стартирането на изпълнението на кода**, който се намира в **тялото на функцията**. Това става като изпишем **името** ѝ, последвано от кръглите скоби () и знака ; за край на реда. Ако функцията ни изисква входни данни, то те се подават в скобите (), като последователността на фактическите параметри трябва да съвпада с последователността на подадените при декларирането на функцията. Ето един пример:

```
//Function declaration
void sum(int, int, int);

//Calling the function
int main() {
    int a1, b1, c1;
    cin >> a1 >> b1 >> c1;
    sum(a1, b1, c1);
    return 0;
}
```

```
//Function definition
void sum(int a, int b, int c) {
    cout << a + b + c;
}
```

Дадена функция може да бъде извикана от **няколко места** в нашата програма. Единият начин е да бъде извикана от **главната функция**:

```
//Using the function in main
int main() {
    printHeader();
    return 0;
}
```

Функция може да бъде извикана и от **тялото** на произволна друга функция:

```
//Using two functions in another
void print() {
    printHeader();
    printFooter();
}
```

Важно: Разгледаните тук функции не връщат стойност (резултат). Тях ще ги разгледаме малко по-късно, в секция [Връщане на резултат от функция](#).

Съществува вариант функцията да бъде извикана от **собственото си тяло**. Това се нарича **рекурсия** и можете да намерите повече информация за нея в [Wikipedia](#) или да потърсите сами в Интернет.

Обобщение

Дотук подробно разгледахме **елементите** на една функция в езика **C++**, както и **разпределението** им в кода на програмата. Оказва се, **методиката** за работа с функции, която разгледахме, е **стандартна**, но **не** е единствената. Ще разгледаме **два различни** подхода, като единия го разгледахме подробно.

Нека напишем програма, използваща функция (**printSentence()**) за извеждане на изречението **I am learning functions in C++**. Използваме познатия ни вече алгоритъм:

- Декларираме функцията **printSentence()** преди **main()** функцията.
- Дефинираме функцията **printSentence()** след **main()** функцията.
- Извикваме функцията **printSentence()** в главната **main()**.

```
// First approach
#include <iostream>
using namespace std;
```

```
// Declaration
void printSentence();

int main() {
    // Function call
    printSentence();
    return 0;
}

// Definition
void printSentence() {
    cout << "I am learning functions in C++.";
}
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/printSentence1>.

Какъв е другият подход? Нека разгледаме следната програма, решаваща същата задача:

```
// Second approach
#include <iostream>
using namespace std;

// Declaration and definition at same time
void printSentence() {
    cout << "I am learning functions.";
}

int main() {
    // Function call
    printSentence();
    return 0;
}
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/printSentence2-1>.

Оказва се, че резултатът от двете програми е **еднакъв**. Разликата е само, че при втория подход **декларирането и дефинирането** на функцията са слети в едно, като това се случва преди главната функция **main()**. Факт е, че вторият подход изиска по-малко писане на код, но в **практиката** често се използва **първият подход**, тъй като дава по-добра **четливост** на програмата, както и поради **други причини**.

В настоящата книга се използва първият подход (т.е. с отделена декларация и дефиниция). Това е и препоръчителният подход.

Прототип на функция

Нека да поясним още нещо - вместо **деклариране** на функция (както дотук използвахме за яснота), ще казваме **прототип на функция**. Самото наименование ни подсказва изведеното по-горе, а именно че **преди** функция **main()** съобщаваме за дадена функция (**правим ѝ прототип**), създаваме я (пишем нейното **тяло**) след функция **main()**, а я използваме (**извикваме**) или в **main()**, или в друга функция. Единствената функция, на която не правим прототип, е главната **main()**.

Пример: празна касова бележка

Да се напише функция, който печата празна касова бележка. Функцията трябва да извика други три функции: една за принтиране на заглавието, една за основната част на бележката и една за долната част.

Част от касовата бележка	Текст	
Горна част	CASH	RECEIPT -----
Средна част	Charged to _____ Received by _____	
Долна част	----- (c) SoftUni	

Примерен вход и изход

Вход	Изход
(няма)	CASH RECEIPT ----- Charged to _____ Received by _____ ----- (c) SoftUni

Насоки и подсказки

Първата ни стъпка е да създадем **void** функция за **принтиране на заглавната част** от касовата бележка (header). Под създаване на функция трябва да разбираме писане на **прототип** и **дефиниция отделно**. Нека ѝ дадем смислено име, което описва кратко и ясно задачата ѝ, например **printReceiptHeader()**. Ето как ще изглежда нейната дефиниция:

```
void printReceiptHeader() {
    cout << "CASH RECEIPT" << endl;
    cout << "-----" << endl;
}
```



Оттук нататък даваните примери в главата ще съдържат **само дефиниции** на функции. Оставяме читателя **сам** да създава техните прототипи.

Съвсем аналогично създаваме още две функции - една за разпечатване на средната част на бележката (тяло) **printReceiptBody()** и една за разпечатване на долната част на бележката (footer) **printReceiptFooter()**. След това създаваме и **още една функция**(**printReceipt()**), която ще извиква една след друга трите функции, които написахме до момента:

```
void printReceipt {
    printReceiptHeader();
    printReceiptBody();
    printReceiptFooter();
}
```

Накрая ще **извикаме** функцията **printReceipt()** от тялото на главната **main()** функция за нашата програма:

```
int main() {
    printReceipt();
    return 0;
}
```

Броят на съответните символи в изхода трябва да е както в показания по-горе пример!

Тестване в Judge системата

Програмата с общо пет функции, които се извикват една от друга, е готова и можем **да я изпълним и тестваме**, след което да я пратим за проверка в Judge системата: <https://judge.softuni.bg/Contests/Practice/Index/1374#0>.

Функции с параметри

Много често в практиката, за да бъде решен даден проблем, функцията, с чиято помош постигаме това, се нуждае от **допълнителна информация**, която зависи от задачата ѝ. Именно тази информация представляват **параметрите на функцията** и нейното поведение зависи от тях.

Използване на параметри във функциите

Както отбелязахме по-горе, **параметрите освен нула на брой**, могат също така да

са **един или няколко**. При декларацията им ги разделяме със запетая. Те могат да бъдат от различен тип данни (**int**, **string** и т.н.), а по-долу е показан пример как точно ще бъдат използвани от функцията.

Създаваме прототип на функцията, като **най-малкото**, което упоменаваме в списъка с параметри, е **типът данни** на параметрите, т.е. **не** е задължително те да бъдат именувани. **При дефиницията обаче това е задължително**. Там е и мястото, където се записва тялото на функцията.

```
// Definition
void printNumbers(int start, int end) {
    for (int i = start, i <= end; i++) {
        cout << i;
    }
}
```

След това извикваме функцията като ѝ подаваме конкретни стойности:

```
int main() {
    int a = 5; b = 10;
    printNumbers(a, b);
}
```

При **декларирането на параметри** можем да използваме **различни** типове променливи, като трябва да внимаваме всеки един параметър да има **тип** (и **име** при дефиницията). Важно е да отбележим, че при последващото извикване на функцията, трябва да подаваме **стойности** за параметрите в **реда**, в който са **деклариирани**. Ако имаме функция с параметри **int** и **char** в тази последователност, при извикването ѝ не може да подадем първо стойност за **char** и след това за **int**. Единствено може да разменяме местата на подадените параметри, ако изрично изпишем преди това името на параметъра, както ще забележим малко по-нататък в един от примерите. **Това като цяло не е добра практика!**

Нека разгледаме пример за декларация на функция, която има няколко параметъра от различен тип:

```
//Definition
void data(char letter, int number1, double number2 {
    cout << "The letter is " << letter << endl;
    cout << "The integer is" << number1 << endl;
    cout << "The real number is " << number2;
}
```

Пример: знак на цяло число

Да се създаде функция, която печата знака на цяло число n.

Примерен вход и изход

Вход	Изход
2	The number 2 is positive.

Вход	Изход
-5	The number 2 is negative.

Вход	Изход
0	The number 0 is zero.

Насоки и подсказки

Първата ни стъпка е **създаването** на функция и даването ѝ на описателно име, например **printSign(...)**. Тази функция ще има един параметър (от тип **int**) - числото, чийто знак искаме да проверим. Следващата ни стъпка е **имплементирането** на логиката, чрез която програмата ни ще проверява какъв точно е знакът на числото. От примерите виждаме, че има три случая - числото е по-голямо от нула, равно на нула или по-малко от нула, което означава, че ще направим три проверки в тялото на функцията:

```
void printSign(int a) {
    cout << "The number" << a << "is ";
    if (a == 0) {
        cout << "zero";
    }
    else {
        if (a > 0) {
            cout << "positive";
        }
        else {
            cout << "negative";
        }
    }
}
```

Следващата ни стъпка е да прочетем входното число и да извикаме новата функция от тялото на **main()** функцията:

```
int main() {
    int n;
    cin >> n;
    printSign(n);
    return 0;
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1374#1>.

Може да ви направи впечатление, че променливата от **main()**, която подаваме

като параметър на **printSign(...)** функцията, е с име **n**, а във **printSign(...)** е именувана **a**. Можехме във въпросната функция да използваме параметър, който носи **същото** име като променливата в **main()**, т.е. **n**.



В C++ е **задължително** при **извикване** на функции списъкът с параметри да се запълва, тъй както са **декларирани** самите функции - същият брой параметри, същите типове данни и подредба.

Пример: принтиране на триъгълник

Да се създаде функция, която принтира триъгълник, както е показано в примерите.

Примерен вход и изход

Вход	Изход	Вход	Изход
3	1 1 2 1 2 3 1 2 1	4	1 1 2 1 2 3 1 2 3 4 1 2 3 1 2 1

Насоки и подсказки

Преди да създадем функция за принтиране на един ред с дадени начало и край, прочитаме входното число от конзолата. След това избираме смислено име за функцията, което описва целта ѝ, например **printLine(...)**, и я имплементираме:

```
void printLine(int start, int end) {
    for (int i = start; i <= end; i++) {
        cout << i;
    }
    cout << endl;
}
```

От задачите за рисуване на конзолата си спомняме, че е добра практика **да разделяме фигурата на няколко части**. За наше улеснение ще разделим триъгълника на три части - горна, средна и долната.

Следващата ни стъпка е с цикъл да разпечатаме **горната половина** от триъгълника:

```
for (int i = 0; i < n; i++) {
    printLine(1, i);
}
```

След това разпечатваме **средната линия**:

```
printLine(1, n);
```

Накрая разпечатваме **долната част** от триъгълника, като този път стъпката на цикъла намалява.

```
for (int i = n-1; i > 0; i--) {
    printLine(1, i);
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1374#2>.

Пример: рисуване на запълнен квадрат

Да се нарисува на конзолата запълнен квадрат със страна n , както е показано в примерите.

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
4	<pre>----- - \/\ \/\ - - \/\ \/\ \/ -----</pre>	3	<pre>----- - \/\ \/ - \/\ \/ -----</pre>	2	<pre>-----</pre>

Насоки и подсказки

Първата ни стъпка е да прочетем входа от конзолата. След това трябва да създадем функция, която ще принтира първия и последен ред, тъй като те са еднакви. Нека не забравяме, че трябва да й дадем **описателно име** и да й зададем като **параметър** дълчината на страната:

```
// Definition
void printHeaderFooter(int n) {
    for (int i = 0; i < n; i++) {
        cout << "-";
    }
    cout << endl;
}
```

Следващата ни стъпка е да създадем функция, която ще рисува на конзолата средните редове. Отново задаваме описателно име, например `printMiddleRow(...)`:

```
// Definition
void printMiddleRow(int n) {
    cout << "-";
    for (int i = 0; i <= n - 1; i++) {
        cout << "\\/";
    }
    cout << "-" << endl;
}
```

Нека обърнем внимание, че за да се отпечата символът "\", трябва да се въведе в програмата "\" (т.e трябва да го екранираме). Накрая извикваме създадените функции в главния `main()` функция на програмата, за да нарисуваме целия квадрат:

```
int main() {
    int input;
    cin >> input;
    printHeaderFooter(input);
    // TODO: Draw the rest of the square
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1374#3>.

Връщане на резултат от функция

До момента разглеждахме функции, които извършват дадено действие, например отпечатване на даден текст, число или фигура на конзолата. Освен този тип функции, съществуват и такива, които могат да **връщат** някакъв **резултат** от своето изпълнение - например резултатът от умножението на две числа. Именно тези функции ще разгледаме в следващите редове.

Тип на връщаната от функцията стойност

До сега разглеждахме примери, в които при декларация на функции използвахме ключовата дума **void**, която указва, че функцията **не** връща резултат, а изпълнява определено действие:

```
void addOne(int n) {
    n += 1;
```

```

    cout << n;
}

```

Ако **заменим** ключовата дума **void** с някакъв **тип** данни, то това ще укаже на програмата, че функцията трябва да върне някаква стойност от указания тип. Тази върната стойност може да бъде от всяка към тип – **int, char, double** и т.н.



За да върне една функция **резултат**, е нужно да напишем очаквания тип на резултата при декларацията на функцията на мястото на **void**.

```

int plusOne(int n) {
    return n + 1;
}

```

Важно е да отбележим, че **резултатът**, който се връща от функцията, може да е от **тип, съвместим с типа на връщаната стойност** на функцията. Например, ако декларираният тип на връщаната стойност е **double**, то може да върнем резултат от тип **int**.

Оператор return

За да върнем резултат от функция, на помощ идва операторът **return**. Той трябва да бъде **използван в тялото** на функцията и указва на програмата да **спре изпълнението** си и да **върне** на извиквача на функцията определена **стойност**, която се определя от израза след въпросния оператор **return**.

Операторът **return** може да бъде използван и във функции, които не връщат резултат (**void** функции). След оператора не трябва да има израз, който да бъде върнат. Тогава самата функция ще спре изпълнението си, без да връща стойност. В този случай употребата на **return** е единствено за излизане от функцията. Възможно е и операторът **return** да бъде използван на повече от едно място в тялото на функцията.

В примера по-долу имаме функция, която сравнява две числа и връща резултат съответно **-1, 0** или **1** според това дали първият аргумент е по-малък, равен или по-голям от втория аргумент, подаден на функцията. Функцията използва ключовата дума **return** на три различни места, за да върне три различни стойности според логиката на сравненията на числата:

```

// Definition
int compareTo(int number1, int number2) {
    if (number1 > number2) {
        return 1;
    }
}

```

```

else {
    if (number1 == number2) {
        return 0;
    }
    else {
        return -1;
    }
}

```

Кодът след `return` е недостъпен

След оператора `return`, в текущия блок, **не** трябва да има други редове код, тъй като изпълнението на функцията се прекратява и програмата продължава от мястото, откъдето е извикана функцията. Ако след оператора `return` има други инструкции, то те няма да бъдат изпълнени. Някои редактори, в това число и Visual Studio, ще покажат предупреждение, съобщавайки ни, че е засечен код, който **не може да бъде достъпен**:

```

int someFunction() {
    return 7;
    return 5; // This code is unreachable
}

```



В програмирането не може да има два пъти оператор `return` един след друг, защото изпълнението на първия няма да позволи да се изпълни вторият.

Понякога програмистите се шегуват с фразата “**пиши `return; return;` и да си ходим**”, за да обяснят, че логиката на програмата е объркана.

Употреба на връщаната от функцията стойност

След като дадена функция е изпълнена и върне стойност, то тази стойност може да се използва по **няколко** начина. Първият е да **присвоим** резултата като стойност на **променлива** от съвместим тип:

```
int max = GetMax(5, 10);
```

Вторият е резултатът да бъде използван в израз:

```
double total = getPrice() * 10 + addition;
```

Третият е да **подадем** резултата от работата на функцията към **друга функция**:

```
cout << function(a);

function1(function2(variable));
```

Пример: пресмятане на лицето на триъгълник

Да се напише функция, която изчислява лицето на триъгълник по дадени основа и височина и връща стойността му.

Примерен вход и изход

Вход	Изход
3	
4	6

Насоки и подсказки

Първо създаваме функция, която да изчислява лицето на базата на две променливи - дължината на страната **length** и височината **height**. Този път внимаваме при **декларацията** да подадем коректен **тип** данни, който искаме функцията да връне, а именно **float**(може да бъде и **double**):

```
float getTriArea(float length, float height) {
    return (length * height) / 2;
}
```

Следващата ни стъпка е да прочетем входните данни и да **извикаме новата** функция с тях. Резултатът **записваме в подходяща променлива** и извеждаме на екрана:

```
int main() {
    float a, b;
    cin >> a;
    cin >> b;

    float area = getTriArea(a, b);
    cout << area;

    return 0;
}
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1374#4>.

Пример: степен на число

Да се напише функция, която изчислява и връща резултата от повдигането на число на дадена степен.

Примерен вход и изход

Вход	Изход
2	256
8	

Вход	Изход
3	81
4	

Насоки и подсказки

Първата ни стъпка отново ще е да прочетем входните данни от конзолата. Следващата стъпка е да създадем функция, която ще приема два параметра (числото и степента) и ще връща като резултат число от тип **double**:

```
// Definition
double calculatePower(double num, double pow) {
    double result = 1;
    // Use a loop to calculate
    return result;
}
```

Задачата може да се реши, като се използва функцията **pow(...)** от библиотеката **cmath**. Ако тя се използва, **препоръчително** е идентификаторът **pow** на параметъра да бъде **променен**, за да не се получи объркване в програмата. След като сме направили нужните изчисления, ни остава да извикаме дефинираната функция и да отпечатаме резултата.

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни: <https://judge.softuni.bg/Contests/Practice/Index/1374#5>.

Функции, връщащи няколко стойности

Понякога се налага една функция да връща **повече** от една стойност. Разбира се, очевидно е, че такава функция би била **сложна** и може да бъде **разделена** на **отделни** функции. Но все пак, ако трябва **една** функция да връща няколко стойности, то се нуждаем от **друг подход**. Той може да бъде свързан с използването на **масиви, вектори** и т.н., но те не са обект на изучаване в настоящата книга. Нека обаче да разгледаме поне един универсален метод за групиране на данни - ще въведем понятието **структури** (или структурен тип данни).

Структури

Ако декларираме една структура, то това означава, че създаваме съвкупност от променливи, свързани логически, разбира се, които също се декларират. Ето един пример за структура:

```
struct student {
    string name;
    int age;
    char sex;
    int grade;
};
```

Променливите **name**, **age**, **sex** и **grade** са **елементи** на структурата **student**. Декларацията на **student** се прави **преди** главната **main** функция. Тялото на структурата се намира между къдрявите скобите **{** и **}**, като след затварящата скоба **}** **задължително** се поставя точка и запетая **;**. А как се ползва в програмата?

```
int main() {
    struct student pupil;
    cin >> pupil.name;
    cin >> pupil.age;
    cin >> pupil.sex;
    cin >> pupil.grade;
    return 0;
}
```

Това, което правим в **main()**, е деклариране на променлива **pupil** от структурен тип данни **student**, който пък, от своя страна, вече е деклариран преди **main()**. Въвежданите след това стойности всъщност са **елементите** на **pupil**. До тях достигаме, като изпишем името на декларираната от нас променлива от структурен тип данни (в случая **pupil**), след това поставим точка **.** и накрая изпишем желания от нас елемент (например **name**). Така, за да въведем името на дадения ученик, използваме **pupil.name**.

Важно е да отбележим, че действия могат да бъдат извършвани само с **елементите** на структурите, но не и с имената на самите структури.

Структури и функции

Ето как ще изглежда прототипът на една функция, използваща променлива от структурен тип данни:

```
student include(struct student pupil);
```

А дефиницията:

```
student include(struct student pupil) {
    cin >> pupil.name;
    cin >> pupil.age;
    cin >> pupil.sex;
    cin >> pupil.grade;
    return pupil;
}
```

В случая това е функция, която ще върне променлива от **структурен тип данни student**. По този начин функцията връща четири стойности (за името, възрастта, пола и класа на ученика).

Нека читателят направи самостоятелно цялата програма, използваща функция, като въвежда четирите компонента от характеристиката на един ученик, а след това да допише програмата така, че функцията да се изпълнява за всеки ученик от едно училище с **n** на брой ученици.

Варианти на функции

В много езици за програмиране една и съща функция може да е декларирана в **няколко варианта** с еднакво име и различни параметри. Това е известно с термина "**function overloading**". Сега нека разгледаме как се пишат подобни функции (overloaded functions).

Сигнатура на функцията

В програмирането начинът, по който се **идентифицира** една функция, е чрез **двойката елементи** от декларацията ѝ – **име** на функцията и **списък** от неговите параметри. Тези два елемента определят нейната **спецификация**, т. нар. **сигнатура** на функцията:

```
void print(char symbol) {
    cout << symbol;
}
```

В този пример сигнатурата на функцията е нейното име **print**, както и нейният параметър **char symbol**.

Ако в програмата ни има **функции с еднакви имена, но с различни сигнатури**, то казваме, че имаме **варианти на функции** (function overloading).

Варианти на функции

Както споменахме, ако използваме **едно и също име** за **няколко функции с различни сигнатури**, то това означава, че имаме **варианти на функция**. Кодът по-

долу показва как три различни функции могат да са с едно и също име, но да имат различни сигнатури и да изпълняват различни действия:

```
void print(char symbol) {
    cout << symbol;
}

void print(int number) {
    cout << number;
}

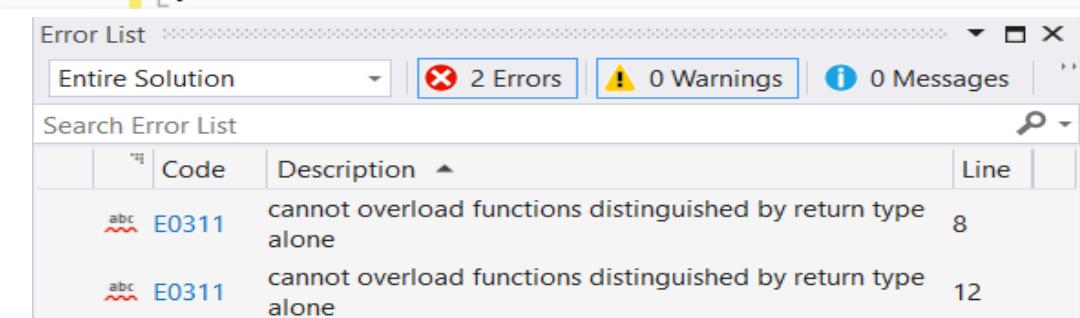
void print(char symbol, int number) {
    cout << symbol << " " << number;
}
```

Сигнатура и тип на връщаната стойност

Важно е да отбележим, че **връщаният тип като резултат** на функцията **не е част от сигнатурата му**. Ако връщаната стойност беше част от сигнатурата на функцията, то няма как компилаторът да знае коя функция точно да извика.

Нека разгледаме следния пример - имаме две функции с различен тип на връщаната стойност. Въпреки това Visual Studio ни показва, че има грешка, защото сигнатурите и на двете са еднакви. Съответно при опит за извикване на функция с име **print(...)**, компилаторът не би могъл да прецени коя от двете функции да изпълни:

```
8     void print(int number) {
9         cin >> number;
10    }
11
12    int print(int number) {
13        number = 7;
14    }
```



Пример: по-голямата от две стойности

Като входни данни са дадени две стойности от един и същ тип. Стойностите могат да са от тип **int**, **char** или **string**. Да се създаде функция **getMax(...)**, която връща като резултат по-голямата от двете стойности.

Примерен вход и изход

Вход	Изход
int	
2	16
16	

Вход	Изход
char	
a	z
z	

Вход	Изход
string	
Ivan	Todor
Todor	

Насоки и подсказки

За да създадем тази функция, първо трябва да създадем три други функции с едно и също име и различни сигнатури. Първо създаваме функция, която ще сравнява цели числа:

```
// Definition
int getMax(int first, int second) {
    if (first >= second) {
        return first;
    }
    else return second;
}
```

Следвайки логиката от предходната функция, създаваме такава със същото име, която обаче ще сравнява символи:

```
// Definition
char getMax(char first, char second) {
    // TODO: Create logic
}
```

Следващата функция, която трябва да създадем, ще сравнява стрингове. Тук логиката ще е малко по-различна, тъй като стойностите от тип **string** не позволяват да бъдат сравнявани чрез операторите **<** и **>**. Ще използваме функцията **compare(...)**, която връща числова стойност: **по-голяма** от 0 (сравняваният обект е **по-голям**), **по-малка** от 0 (сравняваният обект е **по-малък**) и 0 (при два **еднакви** обекта). За да използваме функцията, трябва да бъде декларирана библиотеката **string**:

```
// Definition
string getMax(string first, string second) {
    if (first.compare(second) >= 0) {
        return first;
    }
    else return second;
}
```

Последната стъпка е да прочетем входните данни, да използваме подходящи променливи и да извикаме функцията **getMax(...)** от тялото на функцията **main()**:

```
int main() {
    string type;
    cin >> type;

    if (type == "int") {
        int first, second, max;
        cin >> first;
        cin >> second;
        max = getMax(first, second);
        cout << max;
    }
    else {
        if (type == "char") {
            char first, second, max;
            // TODO: get the max value
        }
        else {
            string first, second, max;
            // TODO: get the max value
        }
    }
    return 0;
}
```

Важно е да се отбележи, че един стринг може да бъде съставен от **няколко думи**, между които има интервал (space). Например низът "Software University" е валиден стринг. За да можем, обаче да прочетем такъв низ в C++ е необходимо да използваме различен от досегашния подход - чрез функцията **getline(...)**:

```
string str;
```

```
getline(cin, str);
```

Може да пуснете в действие и тествате кода от горния пример онлайн:

<https://repl.it/@vncpetrov/ReadingStringWithGetline>.

За да използваме функцията **getline(...)**, трябва предварително да сме реферирали библиотеката **string** (в зависимост от средата за програмиране, може да бъде срещната и като **cstring** или **string.h**): **#include <string>**.

Заради известни съображения, на които няма да се спирате сега, ще въведем употребата на още една функция, част от подхода за прочитане на стринг, съдържащ интервали: **cin.ignore()**. Тази функция е необходимо да бъде извикана преди **getline(...)**, ако преди това сме прочели стойности от друг тип данни (**int**, **char** и т.н.). Ако последователно се четат стрингове, съдържащи интервали, то между тях не е нужно да се извиква споменатата функция. Нека разгледаме фрагмент от програма, която чете и печата различни стойности, сред които стринг, съдържащ интервали:

```
int number;
char symbol;
string word;
string sentence;

cout << "Enter a number: ";
cin >> number;

cout << "Enter a symbol: ";
cin >> symbol;

cout << "Enter a single word: ";
cin >> word;

cin.ignore();
cout << "Enter a sentence: ";
getline(cin, sentence);

cout << number << endl;
cout << symbol << endl;
cout << word << endl;
cout << sentence << endl;
```

Може да пуснете в действие и тествате кода от горния пример онлайн:

<https://repl.it/@vncpetrov/ReadingAndPrintingMultipleValues>.

При закоментиране на реда **cin.ignore();** програмата няма да работи както очакваме и прочитането на последната стойност ще се пропусне.

Низовете, в които има интервали, както и низовете, в които няма интервали, се сравняват по един и същ начин от функцията **compare(...)** - лексикографски.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1374#6>.

Вложени функции (локални функции)

В някои езици за програмиране може една функция да бъде декларирана и дефинирана в тялото на друга функция, т.е тя е локална. В езика C++ това е недопустимо!



В езика за програмиране C++ не съществува понятието локална функция!

Именуване на функции. Добри практики при работа с функции

В тази част ще се запознаем с някои **утвърдени практики** при работа с функции, свързани с именуването, подредбата на кода и неговата структура.

Именуване на функции

Когато именуваме дадена функция е препоръчително да използваме **смислени имена**. Тъй като всяка функция **отговаря** за някаква част от нашия проблем, то при именуването ѝ трябва да вземем предвид **действието, което тя извършва**, т.е. добра практика е **името да описва нейната цел**.

Името започва с **малка буква** и трябва да е съставено от глагол или от двойка: глагол + съществително име. Форматирането на името става, спазвайки **lowerCamelCase** конвенцията, т.е. **първата дума започва с малка буква, а всяка следваща с главна**. Кръглите скоби (и) винаги следват името на функцията.



Всяка функция трябва да изпълнява самостоятелна задача, а името на функцията трябва да описва каква е операцията, която извършва.

Няколко примера за **коректно** именуване на функции:

- **findStudent**
- **loadReport**
- **sine**

Няколко примера за **лошо** именуване на функции:

- **function1**
- **doSomething**

- `handleStuff`
- `simpleFunction`
- `dirtyHack`

Ако не можем да измислим подходящо име, то най-вероятно функцията решава повече от една задача или няма ясно дефинирана цел. В такива случай е добре да помислим как да я разделим на няколко отделни функции.

Именуване на параметрите на функциите

При именуването на **параметрите** на функцията важат почти същите правила, както и при самите функции. Разликата тук е, че за имената на параметрите е добре да използваме съществително име или двойка от прилагателно и съществително име. При именуването на параметрите пак се спазва **lowerCamelCase** конвенцията. Трябва да отбележим, че е добра практика името на параметъра да **указва** каква е **мерната единица**, която се използва при работа с него.

Няколко примера за **коректно** именуване на параметри на функции:

- `firstName`
- `report`
- `speedKmH`
- `usersList`
- `fontSizeInPixels`
- `font`

Няколко примера за **некоректно** именуване на параметри на функции:

- `p`
- `p1`
- `p2`
- `populate`
- `LastName`
- `last_name`

Добри практики при работа с функции

Нека отново припомним, че една функция трябва да изпълнява **само една** точно определена **задача**. Ако това не може да бъде постигнато, то тогава трябва да помислим как да **разделим** функцията на няколко отделни такива. Както казахме, името на функцията трябва точно и ясно да описва нейната цел. Друга добра практика в програмирането е да **избягваме** функции, по-дълги от екрана ни

(приблизително). Ако все пак кода стане много обемен, то е препоръчително функцията да се **раздели** на няколко по-кратки, както в следния пример:

```
void printReceipt {
    printReceiptHeader();
    printReceiptBody();
    printReceiptFooter();
}
```

Структура и форматиране на кода

При писането на функции трябва да внимаваме да спазваме коректна **индентация** (отместване навътре с една табулация или 4 интервала). В C++ неправилната индентация не би довела до некоректна работа на програмата, но прави кода по-трудно четим. Пример за **правилно** форматиран C++ код:

```
int main() {
    // some code
    // some more code
}
```

Пример за **некоректно** форматиран C++ код:

```
int main() {
    // some code
// some more code
}
```

Когато заглавният ред на функцията е **твърде дълъг**, се препоръчва той да се раздели на няколко реда, като всеки ред след първия се отмества с две табулации надясно (за по-добра четливост):

```
double getTriangleArea(double length,
    double height) {
    return (length * height) / 2;
}
```

Друга добра практика при писане на код е да **оставяме празен ред** между функциите, след циклите и условните конструкции. Също така, опитвайте да **избягвате** да пишете **дълги редове и сложни изрази**. С времето ще установите, че това подобрява четливостта на кода и спестява време.

Препоръчваме винаги да се **използват къдрави скоби за тялото на проверки и цикли**. Скобите не само подобряват четливостта, но и намалят възможността да бъде допусната грешка и програмата ни да се държи некоректно.

Какво научихме от тази глава?

В тази глава се запознахме с базовите концепции при работа с функции:

- Научихме, че **целта** на функциите е да **разделят** големи програми с много редове код на по-малки и ясно обособени задачи.
- Запознахме се със **структурата** на функциите - как да ги **декларираме**, **дефинираме** и **извикваме** по тяхното име.
- Разгледахме примери за функции с **параметри** и как да ги използваме в нашата програма.
- Научихме какво представляват **сигнатурата** и **връщаната стойност** на функцията, както и каква е ролята на оператора **return** в функциите.
- Запознахме се с **добрите практики** при работа с функции - как да **именуваме** тях и техните параметри, как да **форматираме** кода и други.

Упражнения

За да затвърдим работата с функции, ще решим няколко задачи. В тях се изисква да напишете функция с определена функционалност и след това да я извикате като ѝ подадете данни, прочетени от конзолата, точно както е показано в примерния вход и изход.

Задача: "Hello, Име!"

Да се напише функция, който получава като параметър име и принтира на конзолата "Hello, \!".

Примерен вход и изход

Вход	Изход
Peter	Hello, Peter!

Насоки и подсказки

Дефинираме функция **printName(string name)** и я имплементираме, след което в главната програма трябва да прочетем от конзолата име на човек и да извикаме функцията, като ѝ подаваме прочетеното име.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1374#7>.

Задача: по-малко число

Да се създаде функция **getMin(int a, int b)**, която връща по-малкото от две

числа. Да се напише програма, която чете като входни данни от конзолата три числа и печата най-малкото от тях. Да се използва функцията **getMin(...)**, която е вече създадена.

Примерен вход и изход

Вход	Изход	Вход	Изход
1		-100	
2	1	-101	
3		-102	

Насоки и подсказки

Дефинираме функция **getMin(int a, int b)** и я имплементираме, след което я извикваме от главната програма, както е показано по-долу. За да намерите минимума на три числа, намерете първо минимума на първите две от тях и след това минимума на резултата и третото число:

```
int min = getMin(getMin(num1, num2), num3);
```

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1374#8>.

Задача: повтаряне на низ

Да се напише функция **repeatString(str, count)**, която получава като параметри променлива от тип **string** и цяло число **n** и отпечатва низа **n** пъти.

Примерен вход и изход

Вход	Изход	Вход	Изход
str 2	strstr	roki 6	rokirokirokirokirokirokiroki

Насоки и подсказки

Задачата може да бъде решена по два начина - единият е чрез **void** функция, другият е чрез функция, връщаща стойност от тип **string**. Във втория случай това означава, че ще трябва да **съединяваме** отделните низове. Това може да стане чрез **обикновено събиране** (конкатенация), както в примера долу:

```
string str1, str2;
cin >> str1;
cin >> str2;
```

```
str1 += str2;
cout << str1;
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/workWithTypeString>.

В по-нататъшното си обучение в областта на C++ ще научим и други подходи при работа с низове.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1374#9>.

Задача: n-та цифра

Да се напише функция **findNthDigit(number, index)**, която получава число и индекс N като параметри и печата N-тата цифра на числото (като се брои отляво на право, започвайки от 1). След това, резултатът да се отпечата на конзолата.

Примерен вход и изход

Вход	Изход
83746 2	4

Вход	Изход
93847837 6	8

Вход	Изход
2435 4	2

Насоки и подсказки

За да изпълним алгоритъма, ще използваме **while** цикъл, докато дадено число не стане 0. На всяка итерация от **while** цикъла ще проверяваме дали настоящият индекс на цифрата не отговаря на индекса, който търсим. Ако отговаря, ще върнем като резултат цифрата на индекса (**number % 10**). Ако не отговаря, ще премахнем последната цифра на числото (**number = number / 10**). Трябва да следим коя цифра проверяваме по индекс (от дясно на ляво, започвайки от 1). Когато намерим цифрата, ще върнем индекса.

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1374#10>.

Задача: число към бройна система

Да се напише функция **integerToBase(number, toBase)**, която получава като параметри цяло число и основа на бройна система и връща входното число, конвертирано към посочената бройна система. След това, резултатът да се отпечата на конзолата. Входното число винаги ще е в бройна система 10, а параметърът за основа ще е между 2 и 10.

Примерен вход и изход

Вход	Изход	Вход	Изход	Вход	Изход
3 2	11	4 4	10	9 7	12

Насоки и подсказки

За да решим задачата, ще декларираме стрингова променлива, в която ще пазим крайния резултат. След това трябва да изпълним следните изчисления, нужни за конвертиране на числото:

- Извисляваме **остатъка** от числото, разделено на основата.
- **Вмъкваме остатъка** от числото в началото на низа, представящ резултата.
- **Разделяме** числото на основата.
- **Повтаряме** алгоритъма, докато входното число не стане 0.

При решаването на тази задача се достига до това, че трябва променлива то тип данни **int** да се превърне в **string**. Това може да стане по начина, описан по-долу. Този метод ще използваме **не само** за да конвертираме **int**, но и **други** типове данни (**char**, **double** и други) в **string**.

```
int number;
char symbol;
cin >> number;
cin >> symbol;

string numberAsString = to_string(number);
string symbolAsString = to_string(symbol);

cout << numberAsString << endl;
cout << symbolAsString << endl;
```

Може да пуснете в действие и тествате кода от горния пример онлайн:

<https://repl.it/@vncpetrov/IntToStringConversion>.

Кодът на самата задача може да бъде представен по следния начин:

```
string integerToString(int, int);

int main() {
    // TODO: Implement the missing conversion logic

    return 0;
}
```

```

string integerToBase(int number, int toBase) {
    string result = "";
    while (number != 0) {
        // TODO: Implement the missing conversion logic
    }

    return result;
}

```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/IntegerToBase>.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1374#11>

Задача: известия

Да се напише програма, която прочита цяло число **N** и на следващите редове въвежда **N съобщения** (като за всяко съобщение се прочитат по няколко реда). За всяко съобщение може да се получат различен брой параметри. Всяко съобщение започва с **message_type**: **success**, **warning** или **error**:

- Когато **messageType** е **success** да се четат **operation + message** (всяко на отделен ред, в тази последователност).
- Когато **messageType** е **warning** да се чете само **message**.
- Когато **messageType** е **error** да се четат **operation + message + errorCode** (всяко на отделен ред, в тази последователност).

На конзолата да се отпечата **всяко прочетено съобщение**, форматирано в зависимост от неговия **messageType**. Като след заглавния ред за всяко съобщение да се отпечатат толкова на брой символа **=**, **колкото е дълъг** съответният **заглавен ред** и да се сложи по един **празен ред** след всяко съобщение (за по-детайлно разбиране погледнете примерите).

Задачата да се реши с дефиниране на четири функции: **showSuccessMessage()**, **showWarningMessage()**, **showErrorMessage()** и **readAndProcessMessage()**, като само последната функция да се извика от главната **main()** функция:

```

// Prototypes
void showSuccessMessage(string operations,
                        string message);
void showWarningMessage(string message);
void showErrorMessage(string operation,
                      string message, int errorCode);
void readAndProcessMessage();

```

Примерен вход и изход

Вход	Изход
4 error credit card purchase Invalid customer address 500 warning Email not confirmed success user registration User registered successfully warning Customer has not email assigned	Error: Failed to execute credit card purchase. =====Reason: Invalid customer address. Error code: 500. Warning: Email not confirmed. ===== Successfully executed user registration. =====User registered successfully. Warning: Customer has not email assigned. =====

Насоки и подсказки

Дефинираме и имплементираме посочените четири функции в условието.

В **readAndProcessMessage()** прочитаме типа съобщение от конзолата и според прочетения тип прочитаме останалите данни (които може да са още един, два или три реда). След това извикваме съответната функция за печатане на съответния тип съобщение.

За да се изведе ред от символи, дълъг колкото даден низ, е необходимо първо да намерим дължината на съответния низ. Това може да се направи по следния начин:

```
string str = "Software University";
int length = str.size();
cout << string(length, '=') << endl;
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/GetStringLength>.

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни:

<https://judge.softuni.bg/Contests/Practice/Index/1374#12>.

Задача: числа към думи

Да се напише функция **letterize(number)**, която прочита цяло число и го

разпечатва с думи на английски език според условията по-долу:

- Да се отпечатат с думи стотиците, десетиците и единиците (и евентуални минус) според правилата на английския език.
- Ако числото е по-голямо от **999**, трябва да се принтира "**too large**".
- Ако числото е по-малко от **-999**, трябва да се принтира "**too small**".
- Ако числото е **отрицателно**, трябва да се принтира "**minus**" преди него.
- Ако числото не е съставено от три цифри, не трябва да се принтира.

Примерен вход и изход

Вход	Изход
3	nine-hundred and ninety
999	nine
-420	minus four-hundred and twenty
1020	too large

Вход	Изход
2	
15	
350	three-hundred and fifty

Вход	Изход
4	
311	three-hundred and eleven
418	four-hundred and eighteen
509	five-hundred and nine
-	
9945	too small

Вход	Изход
3	
500	
123	
9	

Насоки и подсказки

Можем първо да отпечатаме **стотиците** като текст - **числото / 100**, след тях **десетиците** - **(числото / 10) % 10** и накрая **единиците** - **(числото % 10)**.

Първият специален случай е когато числото е точно **закръглено на 100** (напр. 100, 200, 300 и т.н.). В този случай отпечатваме "one-hundred", "two-hundred", "three-hundred" и т.н.

Вторият специален случай е когато числото, формирано от последните две цифри на входното число, е **по-малко от 10** (напр. 101, 305, 609 и т.н.). В този случай отпечатваме "one-hundred and one", "three-hundred and five", "six-hundred and nine" и т.н.

Третият специален случай е когато числото, формирано от последните две цифри на входното число, е **по-голямо от 10** и **по-малко от 20** (напр. 111, 814, 919 и т.н.).

В този случай отпечатваме "one-hundred and eleven", "eight-hundred and fourteen", "nine-hundred and nineteen" и т.н.

Тестване в Judge системата

Тествайте решението си: <https://judge.softuni.bg/Contests/Practice/Index/1374#13>

Задача: криптиране на низ

Да се напише функция **encrypt(char letter)**, която криптира дадена буква по следния начин:

- Вземат се първата и последна цифра от ASCII кода на буквата и се залепят една за друга в низ, който ще представя резултата.
- Към началото на стойността на низа, който представя резултата, се залепя символът, който отговаря на следното условие:
 - ASCII кода на буквата + последната цифра от ASCII кода на буквата.
- След това към края на стойността на низа, който представя резултата, се залепя символът, който отговаря на следното условие:
 - ASCII кода на буквата - първата цифра от ASCII кода на буквата.
- функцията трябва да върне като резултат криптирания низ.

Пример:

- $j \rightarrow p16i$
 - ASCII кодът на j е **106** → Първа цифра - **1**, последна цифра - **6**.
 - Залепяме първата и последната цифра → **16**.
 - Към **началото** на стойността на низа, който представя резултата, залепяме символа, който се получава от сбора на ASCII кода + последната цифра → $106 + 6 \rightarrow 112 \rightarrow p$.
 - Към **края** на стойността на низа, който представя резултата, залепяме символът, който се получава от разликата на ASCII кода - първата цифра → $106 - 1 \rightarrow 105 \rightarrow i$.

Използвайки функцията, описана по-горе, да се напише програма, която чете **поредица от символи, криптира ги** и отпечатва резултата на един ред.

Приемаме, че входните данни винаги ще бъдат валидни. От конзолата трябва да се прочетат входните данни, подадени от потребителя – цяло число **N**, следвани от по един символ на всеки от следващите **N** реда. Да се криптират символите и да се добавят към криптирания низ. Накрая като резултат трябва да се отпечата **криптиран низ от символи** като в следващия пример:

- $S, o, f, t, U, n, i \rightarrow V83Kp11nh12ez16sZ85Mn10mn15h$

Примерен вход и изход

Подробен да примера за вход и изход може да видите на следващата страница – криптиране на „SoftUni“ и „BiraHax“.

Вход	Изход
7 S o f t U n i	V83Kp11nh12ez16sZ85Mn10mn15h
Вход	Изход
7 B i r a H a x	H66< n15hv14qh97XJ72Ah97xx10w

Насоки и подсказки

Всички операции ще извършим, като всеки път запазваме резултатите в променливи от тип **string**. След това ще "залепим" всички тези променливи както при задача "Повтаряне на низ". Така създаваме крайната променливата **result** от тип **string** и я извеждаме. Трябва да се завърти цикъл **n** пъти, като на всяка итерация към променливата **result** ще прибавяме криптириания символ.

За да намерим първата и последната цифри от ASCII кода, ще използваме алгоритъма, който използвахме за решаване на задача "n-та цифра", а за да създадем низа, ще процедурираме както в задача "Число към бройна система".

Тестване в Judge системата

Тествайте програмата от примера в judge системата на СофтУни: <https://judge.softuni.bg/Contests/Practice/Index/1374#14>.

Глава 11. Хитрости и хакове

В настоящата глава ще разгледаме някои хитрости, хакове и техники, които ще улеснят работата ни с езика **C++** в среда за разработка **Visual Studio**. По-специално ще се запознаем:

- Как правилно да **форматираме код**.
- С конвенции за **именуване на елементи от код**.
- С някои **бързи клавиши** (keyboard shortcuts).
- С някои **шаблони с код** (code snippets).
- С техники за **дебъгване на код**.

Форматиране на кода

Правилното форматиране на нашия код ще го направи **по-четим и разбирам**, в случай че се наложи някой друг да работи с него. Това е важно, защото в практиката ще е необходимо да работим в екип с други хора и е от голямо значение дали пишем кода си така, че колегите ни да могат **бързо да се ориентират** в него.

Има определени правила за правилно форматиране на кода, които събрани в едно се наричат **конвенции**. Конвенциите са група от правила, общоприети от програмистите на даден език, и се ползват масово. Тези конвенции помагат за изграждането на норми в дадени езици - как е най-добре да се пише и какви са **добрите практики**. Приема се, че ако един програмист ги спазва, то кодът му е лесно четим и разбирам.

Езикът **C++** е създаден от **Бьорн Строуstrup** (Bjarne Stroustrup - https://en.wikipedia.org/wiki/Bjarne_Stroustrup). Трябва да знаете, че дори да не се спазват конвенциите, кодът ще работи (стига да е написан правилно), но просто няма да бъде лесно разбирам. Това, разбира се, не е фатално на основно ниво, но колкото по-бързо свикнете да пишете качествен код, толкова по-добре.

За езика **C++ съществуват множество код конвенции**, като например една често използвана е на Bjarne Stroustrup, която е публикувана в статията "Bjarne Stroustrup's C++ Style and Technique FAQ", която може да намериет в https://www.stroustrup.com/bs_faq2.html.

Използваните конвенции в текущата книга са описани на следващите редове. За форматиране на кода се препоръчва **къдрявите скоби {}** да са по следния начин:

- Отварящата скоба **{** да е на същия ред и точно след конструкцията, към която се отнася.
- Затварящата скоба **}** да е самостоятелно на отделен ред.

```
if (someCondition) {
```

```

    cout << "Inside the if statement" << endl;
}

```

Вижда се, че командалата **cout** в примера е **4 празни полета навътре (една табулация)**, което също се препоръчва от Microsoft като добра практика. Без значение колко таба навътре започва определена конструкция с къдрави скоби (т.е. без значение колко влагания на различни оператори имаме), **къдравите скоби {}** трябва да са в **началото на конструкцията**, както е в примера по-долу:

```

if (someCondition) {
    if (anotherCondition) {
        cout << "Inside the if statement" << endl;
    }
}

```

Ето това е пример за **лошо форматиран код** спрямо общоприетите конвенции за писане на езика C++:

```

if(someCondition)
{
    cout << "Inside the if statement" << endl;
}

```

Първото, което се забелязва са **къдравите скоби {}**. Първата (отваряща) скоба трябва да е **точно след if условието**, а втората (затваряща) скоба - **под командалата cout << (...)**, **на отделен празен ред**. В допълнение, командалата вътре в **if** конструкцията трябва да бъде **4 празни полета навътре (един таб)**. Веднага след ключовата дума **if** и преди условието на проверката се оставя **интервал**.

Същото правило важи и за **for цикли**, **както и за всякащи други конструкции, включващи къдрави скоби {}**. Ето още няколко примера:

Правилно:

```

for (int i = 0; i < 5; i++) {
    cout << i << endl;
}

```

Грешно:

```

for(int i=0;i<5;i++){
    cout << i << endl;
}

```

За наше удобство има **бързи клавиши във Visual Studio**, за които ще обясним по-късно в настоящата глава, но засега ни интересуват 2 конкретни комбинации. Едната комбинация е за форматиране на **кода в целия документ**, а другата комбинация - за форматиране на **част от кода**. Ако искаме да форматираме **целия**

код, то трябва да натиснем [CTRL + K + D]. В случай, че искаме да форматираме само **част от кода**, то ние трябва да **маркираме с мишката частта**, която искаме да форматираме, и да натиснем [CTRL + K + F].

Нека използваме **грешния пример** от преди малко:

```
for(int i=0;i<5;i++){
    cout << i << endl;
}
```

Ако натиснем [CTRL + K + D], което е нашата комбинация за форматиране на **целия документ**, ще получим код, форматиран според **общоприетите конвенции за C++**, който ще изглежда по следния начин:

```
for (int i = 0; i < 5; i++) {
    cout << i << endl;
}
```

Тази комбинация може да ни помогне, ако попаднем на лошо форматиран код.

Именуване на елементите на кода

В тази секция ще се фокусираме върху **общоприетите конвенции за именуване на проекти, файлове и променливи** в C++ света.

Именуване на проекти и файлове

За **именуване на проекти и файлове** се препоръчва описателно име, което подсказва **каква е ролята** на въпросния файл / проект и в същото време се препоръчва **PascalCase** конвенцията. Това е **конвенция за именуване** на елементи, при която всяка дума, включително първата, започва с **главна буква**, например **ExpressionCalculator**.

Пример: в този курс се започва с лекция **First steps in coding** и следователно едно примерно именуване на проекта (Solution) за тази лекция може да бъде **FirstStepsInCoding**. Същата конвенция важи и за файловете в даден проект. Ако вземем за пример първата задача от лекцията **First steps in coding**, тя се казва **Hello World** и следователно нашият файл в проекта ще се казва **HelloWorld**.

Именуване на променливи

В програмирането променливите пазят някакви данни и за да е по-разбираме кодът, името на една променлива трябва да подсказва нейното предназначение. Ето и още няколко препоръки за имената на променливите:

- Името трябва да е **кратко и описателно** и да обяснява за какво служи дадената променлива.
- Името трябва да се състои само от буквите **a-z, A-Z, цифрите 0-9**, както

и символа '_'.

- В C++ е прието променливите да **започват** винаги с **малка буква** и да **съдържат малки букви**, като **всяка следваща дума** в тях започва с **главна буква** (това именуване е още познато като **camelCase** конвенция).
- Трябва да се внимава за главни и малки букви, тъй като C++ прави разлика между тях. Например **age** и **Age** са различни променливи.
- Имената на променливите **не могат да съвпадат със служебна дума** (keyword) от езика C++, например **int** е невалидно име на променлива.



Въпреки че използването на променливи от изцяло главни букви в имената е разрешено, в C++ това не се препоръчва и се счита за лош стил на именуване. Добрата практика е променливи с изцяло главни букви да се ползват само за имена на **Macros** (<https://docs.microsoft.com/en-us/cpp/preprocessor/macros-c-cpp?view=vs-2017>)

Ето няколко примера за **добре именувани** променливи:

- **firstName**
- **age**
- **startIndex**
- **lastNegativeNumberIndex**

Ето няколко примера за **лошо именувани променливи**, макар и имената да са коректни от гледна точка на компилатора на C++:

- **_firstName** (започва с '_')
- **last_name** (съдържа '_')
- **AGE** (изписана е с главни букви)
- **Start_Index** (започва с главна буква и съдържа '_')
- **lastNegativeNumber_Index** (съдържа '_')

Първоначално всички тези правила може да ви се струват безсмислени и ненужни, но с течение на времето и натрупването на опит ще видите нуждата от норми за писане на качествен код, за да може да се работи по-лесно и по-бързо в екип. Ще разберете, че е изключително досадна работата с код, който е написан без да се спазват никакви правила за качествен код.

Бързи клавиши във Visual Studio

В предната секция споменахме за две от комбинациите, които се отнасят за форматиране на код. Едната комбинация [CTRL + K + D] беше за **форматиране на целия код в даден файл**, а втората [CTRL + K + F] ни служеше в случай, че искаме

да форматираме само дадена част от кода. Тези комбинации се наричат **бързи клавиши** и сега ще дадем по-подробна информация за тях.

Бързи клавиши са **комбинации**, които ни предоставят възможността да извършваме някои действия **по-лесно и по-бързо**, като всяка среда за разработка на софтуер си има своите бързи клавиши, въпреки че повечето се повтарят. Сега ще разгледаме някои от **бързите клавиши** във Visual Studio.

Комбинация	Действие
[CTRL + F]	Комбинацията отваря търсачка , с която можем да търсим в нашия код .
[CTRL + K + C]	Закоментира част от кода.
[CTRL + K + U]	Разкоментира код, който е вече закоментиран.
[CTRL + Z]	Връща една промяна назад (т.нр. Undo).
[CTRL + Y]	Комбинацията има противоположно действие на [CTRL + Z] (т.нр. Redo).
[CTRL + K + D]	Форматира кода според конвенциите по подразбиране.
[CTRL Backspace] +	Изтрива думата вляво от курсора.
[CTRL + Del]	Изтрива думата вдясно от курсора.
[CTRL + Shift + S]	Запазва всички файлове в проекта.
[CTRL + S]	Запазва текущия файл.

Повече за **бързите клавиши** във Visual Studio може да намерите тук: <https://visualstudioshortcuts.com/2017/>.

Шаблони с код (code snippets)

Във Visual Studio съществуват т.нр. **шаблони с код** (code snippets), при изписването на които се изписва по шаблон някакъв блок с код. Например при изписването на кратък код **for** и натискане на **[Tab]** се генерира следният код, в тялото на нашата програма, на мястото на краткия код:

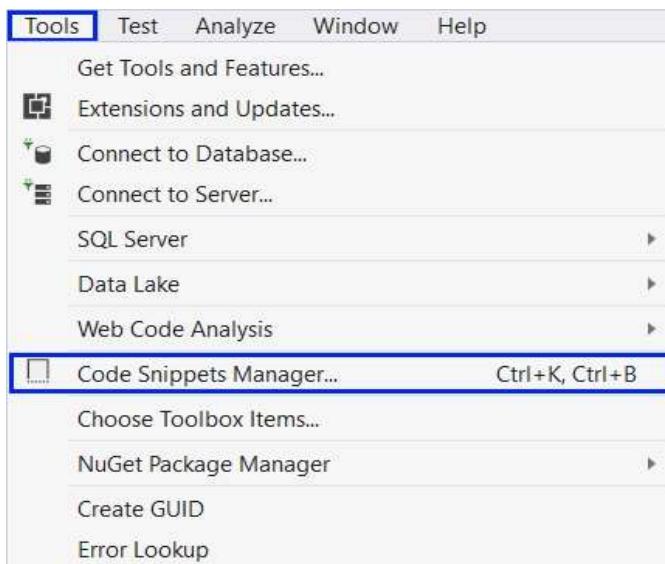
```
for (size_t i = 0; i < length; i++) {  
}
```

Това се нарича "разгъване на шаблон за кратък код". На фигурата по-долу е показано действието на шаблона **for**:

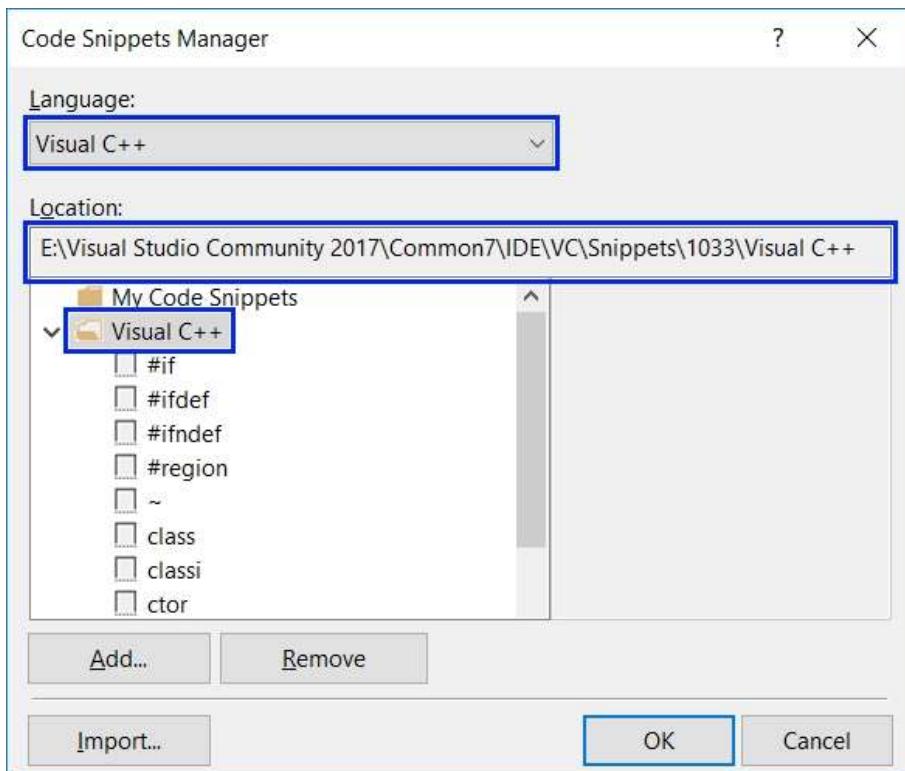


Шаблони за код

Можем да създаваме собствени шаблони или да свалим такива от интернет и да ги добавим (**Add**) или вмъкнем (**Import**). Това се прави от [Tools] -> [Code Snippets Manager], както е показано на картинката:



В отворилия се прозорец трябва да изберем **[Language] -> [Visual C++]**, а от секцията **[Locations] -> [Visual C++]**. Там се намират всички съществуващи шаблони за езика C++:



От бутоните [Add] и [Import] можем да добавим нови шаблони. Бутона [Remove] служи за изтриване.

Техники за дебъгване на кода

Дебъгването играе важна роля в процеса на създаване на софтуер, която ни позволява **стъпка по стъпка** да проследим изпълнението на нашата програма. С помощта на тази техника можем да **следим стойностите на локалните променливи**, тъй като те се променят по време на изпълнение на програмата, и да **отстраним** евентуални **грешки** (бъгове). Процесът на дебъгване включва:

- **Забелязване** на проблемите (бъговете).
- **Намиране** на кода, който причинява проблемите.
- **Коригиране** на кода, причиняващ проблемите, така че програмата да работи правилно.
- **Тестване**, за да се убедим, че програмата работи правилно след нанесените корекции.

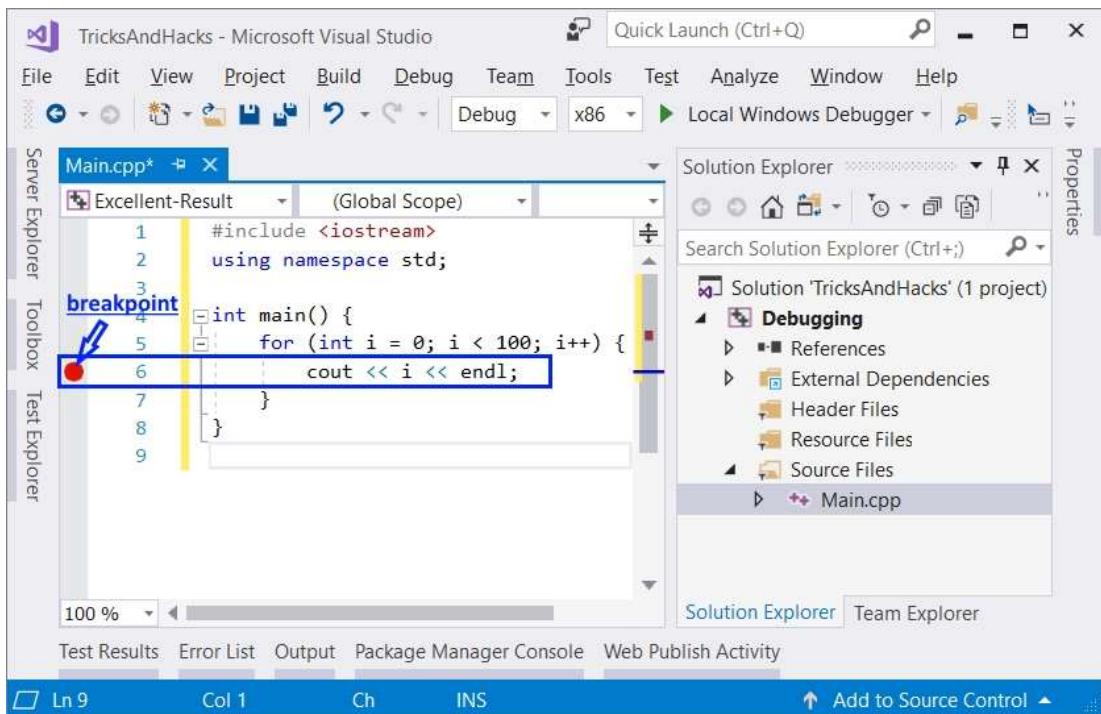
Visual Studio ни предоставя **вграден дебъгер** (debugger), чрез който можем да поставяме **точки на прекъсване** (или breakpoints), на избрани от нас места. При среща на **стопер** (breakpoint), програмата **спира изпълнението** си и позволява

последователно изпълнение на останалите редове. Дебъгването ни дава възможност да **вникнем в детайлите на програмата** и да видим къде точно възникват грешките и каква е причината за това.

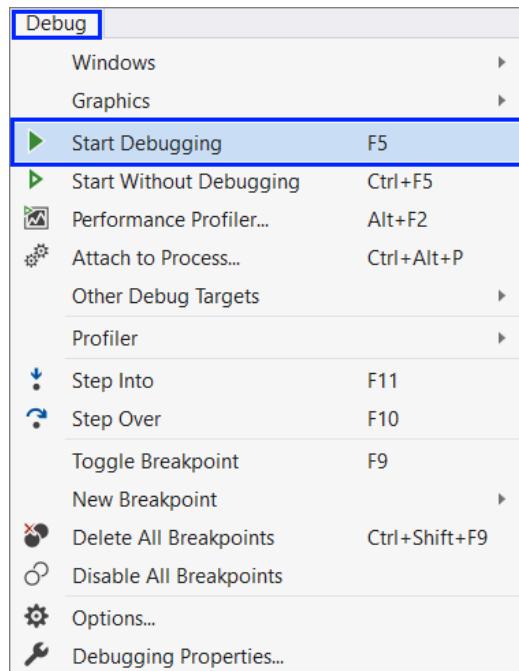
За да демонстрираме работа с дебъгера ще използваме следната програма:

```
int main() {
    for (int i = 0; i < 100; i++) {
        cout << i << endl;
    }
}
```

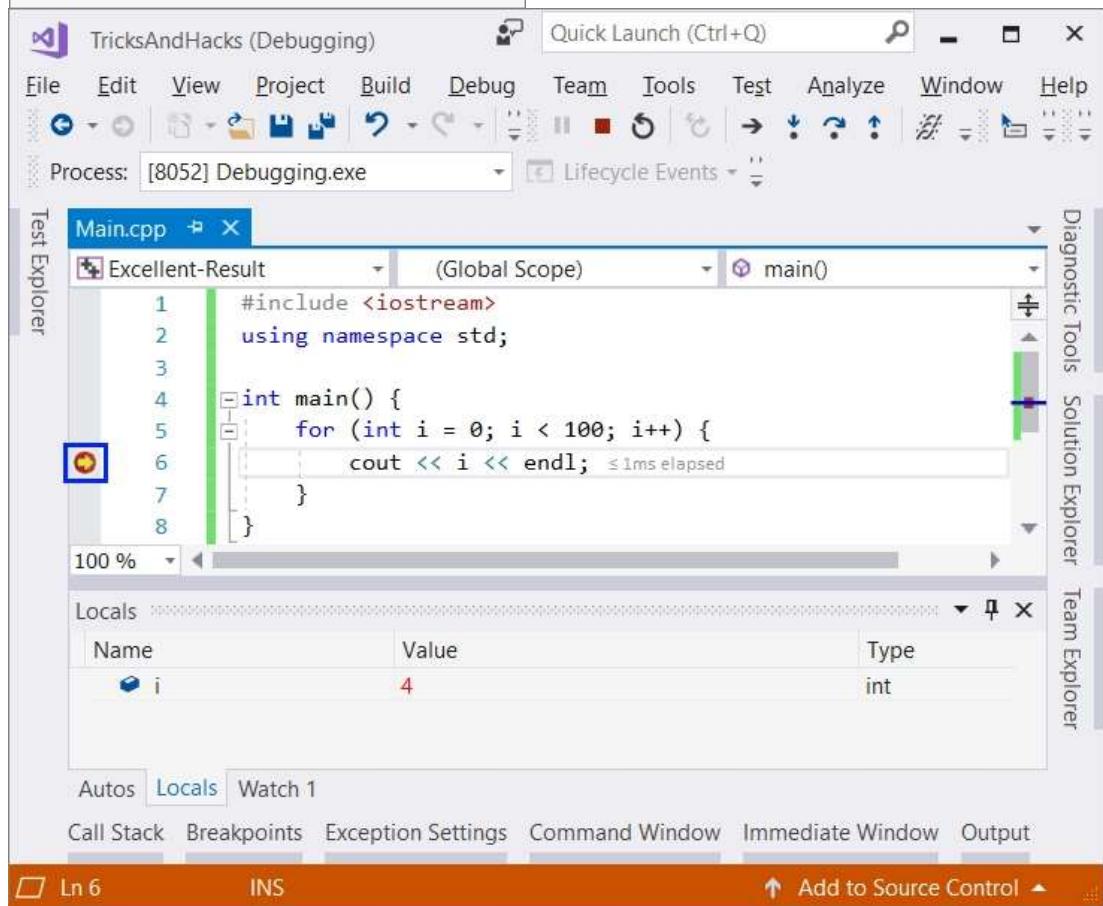
Ще сложим **стопер** (breakpoint) на функцията **cout**. За целта трябва да преместим курсора на реда, който печата на конзолата, и да натиснем [F9]. Появява се **стопер**, където програмата ще **спре** изпълнението си:



За да стартираме програмата в режим на дебъгване, избираме [Debug] -> [Start Debugging] или натискаме [F5]:



След стартиране на програмата виждаме, че тя **спира изпълнението си** на ред 6, където сложихме стопера (breakpoint). Стрелката вляво показва текущия ред. За да преминем на **следващ ред** използваме клавиш [F10]. Забелязваме, че кодът на текущия ред все още не е изпълнен. Изпълнява се, когато преминем на следващия ред:



От прозореца [Locals] можем да наблюдаваме промените по локалните променливи. За да отворим прозореца, избираме [Debug] -> [Windows] -> [Locals]:

Name	Value	Type
i	4	int

Справочник с хитрости

В тази секция ще припомним накратко **хитрости и техники** от програмирането с езика C++, разглеждани вече в тази книга, които ще са много полезни при явяването на изпит по програмиране за начинаещи:

Отпечатване на текст на конзолата

```
string text = "some text";
cout << text << endl;
// Това ще отпечата на конзолата "some text"
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/PrintText>.

Вкарване на променливи в стринг (string)

Placeholder представлява израз, който ще бъде заменен с конкретна стойност при отпечатване. Функцията **printf(...)** поддържа печatanе на текст по шаблон, като първият аргумент, който трябва да подадем, е форматиращият низ, следван от броя аргументи, равен на броя на плейсхолдърите. Функцията се намира в библиотеката **stdio.h** и е необходимо да я реферираме в началото на програмата:

```
string text = "some text";
int number = 5;

printf("%s %d %s\n",
       text.c_str(), number, text.c_str());
// Това ще отпечата "some text 5 some text"
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/VarInString>.

В този пример забелязваме, че можем да подаваме **не само текстови променливи**. С **%** се указва типа на следващата променлива: **%d** се използва с числов тип **int**, **%s** - за стринг и т.н.

Форматиране с 2 цифри след десетичния знак

```
double number = 5.432432;
cout << fixed << setprecision(2) << number;
// Това ще отпечата на конзолата "5.43"
```

Може да пуснете в действие и тествате кода от горния пример онлайн:

<https://repl.it/@vncpetrov/RoundingSetprecision>.

Закръгляме резултата до **2 цифри след десетичния знак** използвайки **fixed << setprecision(2)**, като не забравяме да реферираме библиотеката **<iomanip>**.

Ако искаме да закръглим до **2 цифри след десетичния знак** и третата цифра е по-малка от 5, както в примера по-горе, то закръглянето е надолу, но ако третата цифра е 5 или по-голяма - закръглянето е нагоре, както е в примера по-долу:

```
double number = 5.439;
cout << fixed << setprecision(2) << number << endl;
// Това ще отпечата на конзолата "5.44"
```

Може да пуснете в действие и тествате кода от горния пример онлайн:

<https://repl.it/@vncpetrov/RoudinginSetprecision2>.

Закръгляне на числа

При нужда от закръгляне можем да използваме един от следните функции, които се намират в библиотеката **math.h**:

- **round(...)** - закръглянето се извършва по основното математическо правило - ако десетичната част е по-малка 5, числото се закръгля надолу и обратно: ако е по-голяма от 5 - нагоре. Важно е да отбележим, че тази функция закръгля до цяло число:

```
double firstNumber = 5.431;
cout << round(firstNumber) << endl;
// Това ще отпечата на конзолата "5"

double secondNumber = 5.539;
cout << round(secondNumber) << endl;
// Това ще отпечата на конзолата "6"
```

Може да пуснете в действие и тествате примера онлайн:

<https://repl.it/@vncpetrov/RoundingWithRound>.

- **floor(...)** - в случай, че искаме закръглянето да е винаги **надолу**. Тази функция също закръгля до цяло число. Например, ако имаме числото 5.99 и използваме **floor(5.99)**, ще получим числото 5:

```
double numberToFloor = 5.99;
cout << floor(numberToFloor) << endl;
// Това ще отпечата на конзолата 5
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/RoundFloor>.

- **ceil(...)** - в случай, че искаме закръглянето да е винаги **нагоре**. Тази функция също закръгля до цяло число. Например, ако имаме числото 5.11 и използваме **ceil(5.11)**, ще получим числото 6:

```
double numberToCeiling = 5.11;
cout << ceil(numberToFloor) << endl;
// Това ще отпечата на конзолата 6
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/RoundCeil>.

- **trunc(...)** - в случай, че искаме да **премахнем дробната част**. Например, ако имаме числото 2.63 и използваме **trunc(2.63)**, ще получим 2:

```
double numberToTrunc = 2.63;
cout << trunc(numberToTrunc) << endl;
// Това ще отпечата на конзолата 2
```

Може да тествате примера онлайн: <https://repl.it/@vncpetrov/RoundTrunc>.

Закръгляне чрез placeholder

```
```cpp
double num = 5.439524;
printf("%.3f\n", num);
// Това ще отпечата на конзолата "5.440"
```

```

Може да пуснете в действие и тествате кода от горния пример онлайн:

<https://repl.it/@vncpetrov/RoundingWithPlaceholder>.

В случая след числото добавяме **.3f**, което ще ограничи числото до 3 цифри след десетичния знак. Поведението ще е като на функцията **setprecision(...)**. Трябва да имаме предвид, че числото преди буквата **f** означава до колко цифри след

десетичния знак да е закръглено числото (т.е. може да е примерно **2f** или **5f**). Не забравяйте и **точката преди числото** - тя е задължителна.

Как се пише условна конструкция?

Условната **if** конструкция се състои от следните елементи:

- Ключова дума **if**.
- Булев израз (условие).
- Тяло на условната конструкция.
- Незадължително: **else** клавза.

```
if (условие) {
    // тяло
}
else (условие) {
    // тяло
}
```

За улеснение може да използваме code snippet за **if** конструкция:

- **if** + [Tab]

Как се пише for цикъл?

За **for** цикъл ни трябват няколко неща:

- Инициализационен блок, в който се декларира променливата-брояч (**int i**) и се задава нейна начална стойност.
- Условие за повторение (**i <= 10**).
- Обновяване на брояча (**i++**).
- Тяло на цикъла.

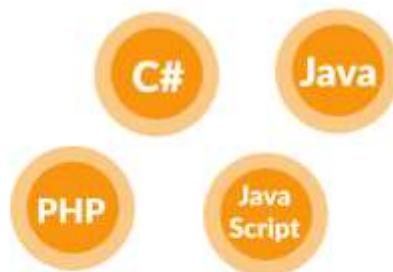
```
for (int i = 0; i < 5; i++) {
    // тяло
}
```

За улеснение може да използваме code snippet за **for** цикъл:

- **for** + [Tab]
- **forr** + [Tab]

Качествено образование,
професия и работа за
Софтуерни инженери

- ✓ Безплатен старт за начинаещи - **присъствено и онлайн**
- ✓ Избор измежду **най-търсените професии** в софтуерната индустрия
- ✓ Съдействие за **кариерен старт**
- ✓ Отлични преподаватели, ментори и студентска общност



СофТУни предоставя съвременно иновативно образование за програмиране, ИТ и дигитални умения на хиляди млади хора. Програмата **"Софтуерен университет"** изгражда истински **профессионалисти в света на програмирането**.

Включете се в цялостната програма по софтуерно инженерство на СофТУни, за да усвоите **най-търсените съвременни софтуерни технологии** чрез модерна учебна методика и с много практически задачи и проекти. **Учебният план** е разработен съвместно с ИТ фирмите и изгражда най-търсените от тях умения.

СофТУни работи пряко с компаниите от софтуерната индустрия, съдейтайки на своите студенти за реализацията им като успешни софтуерни инженери.

ПЪТЯТ НА СТУДЕНТА В СОФТУНИ



1 - 1.5 години



1.5 - 2 години



Programming Basics

Кариерен Старт

Дипломиране

70/100 credits

80/100/150 credits

Заключение

Ако сте прочели **цялата** книга и сте решили всички задачи от упражненията и сте стигнали до настоящото заключение, заслужавате **поздравления!** Вече сте направили **първата стъпка** от изучаването на **професията на програмиста**, но имате още доста **дълъг път** докато станете **истински добри** и превърнете **писането на софтуер** в своя професия.

Спомнете си за четирите основни групи умения, които всеки програмист трябва да притежава, за да работи своята професия:

- Умение #1 – **писане на програмен код** (20% от уменията на програмиста) - покриват се до голяма степен от тази книга, но трябва да изучите още базови структури от данни, класове, обекти, функции, стрингове и други елементи от писането на код.
- Умение #2 – **алгоритмично мислене** (30% от уменията на програмиста) - покриват се частично от тази книга и се развиват най-вече с решаване на голямо количество разнообразни алгоритмични задачи.
- Умение #3 – **фундаментални знания за професията** (25% от уменията на програмиста) - усвояват се за няколко години с комбинация от учене и практикуване (четене на книги, гледане на видео уроци, посещаване на курсове и най-вече писане на разнообразни проекти от различни технологични области).
- Умение #4 – **езици за програмиране и софтуерни технологии** (25% от уменията на програмиста) - усвояват се продължително време, с много практика, здраво четене и писане на проекти. Тези знания и умения бързо остаряват и трябва непрестанно да се актуализират. Добрите програмисти учат всеки ден нови технологии.

Тази книга е само първа стъпка!

Настоящата книга по основи на програмирането е само **първа стъпка** от изграждането на уменията на един програмист. Ако сте успели да решите **всички задачи**, това означава, че сте **получили ценни знания** за принципите на програмиране с езика C++ на **базисно ниво**. Тепърва ви предстои да изучавате **поглътчено** програмирането, както и да развивате **алгоритмичното си мислене**, след което да добавите и **технологични знания** за езика C++ и още редица концепции, технологии и инструменти за разработка на софтуер.

Ако **не сте успели** да решите всички задачи или голяма част от тях, върнете се и ги решете! Помните, че за да **станете програмисти** се изискват **много труд и усилия**. Тази професия не е за мързеливци. Без **да практикувате сериозно** програмирането години наред, няма как да го научите!

Както вече обяснихме, първото и най-базово умение на програмиста е **да се научи да пише код** с лекота и удоволствие. Именно това е мисията на тази книга: да ви научи да кодите. Препоръчваме ви освен книгата, да запишете и практическия

[курс "Основи на програмирането" в СофтУни](#), който се предлага напълно безплатно в присъствена или онлайн форма на обучение.

Накъде да продължим след тази книга?

С тази книга сте **поставили стабилни основи**, благодарение на които ще ви е лесно да продължите да се развивате като програмисти. Ако се чудите как да продължите развитието си, помислете за следните няколко възможности:

- Да учене за **софтуерен инженер** в СофтУни и да направите програмирането своя професия.
- Да продължите развитието си като програмист **по свой собствен път**, например чрез самообучение или с някакви онлайн уроци.
- Да си **останете на ниво кодер**, без да се занимавате с програмиране по-сериозно.

Професия "софтуерен инженер" в СофтУни

Първата, и съответно препоръчителната, възможност да овладеете цялостно и на ниво професията "софтуерен инженер", е да започнете своето обучение по **цялостната програма на СофтУни за софтуерни инженери**: <https://softuni.bg/curriculum>. Учебният план на СофтУни е внимателно разработен от **д-р Светлин Наков и неговия екип**, за да ви поднесе последователно и с градираща сложност всички умения, които един софтуерен инженер трябва да притежава, за **да стартира кариера като разработчик на софтуер** в ИТ фирма.

Продължителност на обучението в СофтУни

Обучението в СофтУни е с продължителност **2-3 години** (в зависимост от професията и избраните специализации) и за това време е нормално да достигнете добро начално ниво (Junior Developer), но **само ако учене сериозно** и здраво пишете код всеки ден. При добър успех един типичен студент **започва работа на средата на обучението си** (след около 1.5 години). Благодарение на добре развита партньорска мрежа **кариерният център на СофтУни предлага работа** в софтуерна или ИТ фирма на всички студенти в СофтУни, които имат много добър или отличен успех. **Започването на работа** по специалността при силен успех в СофтУни, съчетан с желание за работа и разумни очаквания спрямо работодателя, е почти гарантирано.

Програмист се става за най-малко година здраво писане на код

Предупреждаваме ви, че **програмист се става с много усилия**, с писане на десетки хиляди редове код и с решаване на стотици, дори хиляди практически задачи, и отнема години! Ако някой ви предлага "**по-лека програма**" и ви обещава да станете програмисти и да започнете работа за 3-4 месеца, значи или ви **льже**, или ще ви даде толкова ниско ниво, че **няма да ви вземат даже за стажант**, дори и да си плащате на фирмата, която си губи времето с вас. Има и изключения, разбира

се, например ако не започвате от нулата или ако имате екстремно развито инженерно мислене или ако кандидатствате за много ниска позиция (например техническа поддръжка), но като цяло **програмист за по-малко от 1 година здраво учене и писане на код не се става!**

Приемен изпит в СофтУни

За **да се запишете в СофтУни** е нужно да се явите на **приемен изпит** по "Основи на програмирането" върху материала от тази книга. Ако решавате с лекота задачите от упражненията в книгата, значи сте готови за изпита. Обърнете внимание и на няколкото глави за **подготовка за практически изпит по програмиране**. Те ще ви дадат добра представа за трудността на изпита и за типовете задачи, които трябва да се научите да решавате.

Ако задачите от книгата и подготвителните примерни изпити ви затрудняват, значи имате **нужда от още подготовка**. Запишете се на [бесплатния курс по "Основи на програмирането"](#) или преминете внимателно през книгата още веднъж отначало, без да пропускате да решавате **задачите от всяка една учебна тема!** Трябва да се научите **да ги решавате с лекота**, без да си помагате с насоките и примерните решения.

Учебен план за софтуерни инженери

След изпита ви очаква **сериозен учебен план** по програмата на СофтУни за обучение на софтуерни инженери. Той е поредица от **модули с по няколко курса** по програмиране и софтуерни технологии, изцяло насочени към усвояване на фундаментални познания от разработката на софтуер и придобиване на **практически умения за работа** като програмист с най-съвременните софтуерни технологии. На студентите се предоставя избор измежду **няколко професии** и специализации с фокус върху C#, Java, JavaScript, PHP и други езици и технологии. Всяка професия се изучава в няколко модула с продължителност от 4 месеца и всеки модул съдържа 2 или 3 курса. Учебните занятия са разделени на **теоретична подготовка (30%) и практически упражнения, проекти и занимания (70%)**, а всеки курс завършва с практически изпит или практически курсов проект.

Колко часа на ден отнема обучението?

Обучението за софтуерен инженер в СофтУни е **много сериозно занимание** и трябва да му отделите като **минимум поне по 4-5 часа всеки ден**, а за препоръчване е да посветите цялото си време на него. Съчетанието на **работка с учене** невинаги е успешно, но ако работите нещо леко с много свободно време, е добър вариант. СофтУни е подходяща възможност за **ученици, студенти и работещи други професии**, но най-добре е да отделите цялото си време за вашето образование и овладяването на професията. Не става с 2 или 4 часа на седмица!

Формите на обучение в СофтУни са **присъствена** (по-добър избор) и **онлайн** (ако нямate друга възможност). И в двете форми, за да успеете да научите предвиденото в учебния план (което се изисква от софтуерните фирми за

започване на работа), е необходимо **здраво учене**. Просто трябва да намерите време! Причина #1 за бускуване по пътя към професията в СофтУни е недостатъчно време, отделено за обучението: като минимум поне 20-30 часа на седмица.

Софтуни за работещи и учащи другаде

На всички, които изкарат отличен резултат на приемния изпит в СофтУни и се запалят истински по програмирането и мечтаят да го направят своя професия, препоръчваме да се освободят от останалите си ангажименти и **да отделят цялото си време**, за да научат професията "софтуерен инженер" и да започнат да си изкарват хляба с нея.

- За **работещите** това означава да си напуснат работата (и да вземат заем или да си свият финансовите разходи, за да изкарат с по-нисък доход 1-2 години до започване на работа по новата професия).
- За **учащите** в традиционен университет това означава да си изместят силно фокуса към програмирането и практическите курсове в СофтУни, като намалят до минимум времето, което отделят за традиционния университет.
- За **бездработните** това е отличен шанс да вложат цялото си време, сили и енергия, за да придобият една перспективна, добре платена и много търсена професия, която ще им осигури високо качество на живот и дългосрочен просперитет.
- За **учениците** от средните училища и гимназии това означава **да си сложат приоритет** какво е по-важно за тяхното развитие: да учат практическо програмиране в СофтУни, което ще им даде професия и работа или да отделят цялото си внимание на традиционната образователна система или да съчетават умело и двете начинания. За съжаление, често пъти приоритетите се задават от родителите и за тези случаи нямаме решение.

На всички, които **не могат да изкарат отличен резултат на приемния изпит в СофтУни** препоръчваме да набледнат върху по-доброто изучаване, разбиране и най-вече практикуване на учебния материал от настоящата книга. Ако не се справяте с лекота със задачите от тази книга, няма да се справяте и за напред при изучаването на програмирането и разработката на софтуер.

Не пропускайте основите на програмирането! В никакъв случай не трябва да взимате смели решения да напускате работата си или традиционния университет и да кроите велики планове за бъдещата си професия на софтуерен инженер, ако нямаете отличен резултат на входния изпит в СофтУни! Той е мерило доколко ви се отдава програмирането, доколко ви харесва и доколко наистина сте мотивирани да го учите сериозно и да го работите след това години наред всеки ден с желание и наслада.

Професия "софтуерен инженер" по ваш собствен път

Другата възможност за развитие след тази книга е **да продължите да изучавате програмирането извън СофтУни**. Можете да запишете или да следите **видео курсове**, които навлизат в по-голяма дълбочина в програмирането със C++ или други езици и платформи за разработка. Можете **да четете книги** за програмиране и софтуерни технологии, да следвате **онлайн ръководства (tutorials)** и други онлайн ресурси - има безкрайно много безплатни материали в Интернет. Запомнете, обаче, че най-важното по пътя към професията на програмиста е **да правите практически проекти!**

Без писане на много, много код и здраво практикуване, не се става програмист. Отделете си **достатъчно време**. Програмист не се става за месец или два. В Интернет ще намерите голям набор от **свободни ресурси** като книги, учебници, видео уроци, онлайн и присъствени курсове за програмиране и разработка на софтуер. Обаче, ще трябва да инвестирирате **поне година-две**, за да добиете начално ниво като за започване на работа.

След като понапреднете, намерете начин или да започнете **стаж в някоя фирма** (което ще е почти невъзможно без поне година здраво писане на код преди това) или да си измислите **ваш собствен практически проект**, по който да поработите няколко месеца, даже година, за да се учите чрез проба-грешка.



Запомнете, че има много начини да станете програмисти, но всички те имат обща пресечна точка: **здраво писане на код и практика години наред!**

Онлайн общности за стартиращите в програмирането

Независимо по кой път сте поели, ако ще се занимавате сериозно с програмиране, е препоръчително да следите специализирани **онлайн форуми, дискусационни групи и общности**, от които можете да получите помощ от свои колеги и да следите новостите от софтуерната индустрия.

Ако ще учите програмиране сериозно, **обградете се с хора**, които се занимават с **програмиране** сериозно. Присъединете се към **общности от софтуерни разработчици**, ходете по софтуерни конференции, ходете на събития за програмисти, намерете си приятели, с които да си говорите за програмиране и да си обсъждате проблемите и бъговете, намерете среда, която да ви помага. В София и в големите градове има безплатни събития за програмисти, по няколко на седмица. В по-малките градове имате Интернет и достъп до цялата онлайн общност.

Ето и някои препоръчителни **ресурси**, които ще са от полза за развитието ви като програмист:

- <https://softuni.bg> - официален **уеб сайт на СофтУни**. В него ще намерите безплатни (и не само) курсове, семинари, видео уроци и обучения по програмиране, софтуерни технологии и дигитални компетенции.
- <https://softuni.bg/forum> - официален **форум на СофтУни**. Форумът за дискусии на СофтУни е изключително позитивен и изпълнен с желаещи да помогат колеги. Ако зададете смислен въпрос, свързан с програмирането и изучаваните в СофтУни технологии, почти сигурно ще получите смислен отговор до минути. Опитайте, нищо не губите.
- <https://www.facebook.com/SoftwareUniversity/> - официална **Facebook страница на СофтУни**. От нея ще научавате за нови курсове, семинари и събития, свързани с програмирането и разработката на софтуер.
- <https://www.introprogramming.info> - официален уеб сайт на **книгите "Въведение в програмирането"** със **C#** и **Java** от д-р Светлин Наков и колектив. Книгите разглеждат в дълбочина основите на програмирането, базовите структури от данни и алгоритми, ООП и други базови умения и са отлично продължение за четене след настоящата книга. Обаче **освен четене, трябва и здраво писане**, не забравяйте това!
- <https://stackoverflow.com> - **Stack Overflow** е един от **най-големите** в световен мащаб дискусионни форуми за програмисти, в който ще получите помощ за всеки възможен въпрос от света на програмирането. Ако владеете английски език, търсете в StackOverflow.
- <https://fb.com/groups/bg.developers> - групата "**Програмиране България @ Facebook**" е една от най-големите онлайн общности за програмисти и дискусии по темите на софтуерната разработка на български език във Facebook.
- <https://www.meetup.com/find/tech/> - потърсете **технологични срещи (tech meetups)** около вашия град и се включете в общностите, които харесвате. Повечето технологични срещи са безплатни и новобранци са добре дошли.
- Ако се интересувате от ИТ събития, технологични конференции, обучения и стажове, разгледайте и по-големите **сайтове за ИТ събития** като <https://techevents.bg>, <https://dev.bg> и <https://iteventz.bg>.

Успех на всички!

От името на целия авторски колектив ви **пожелаваме нестирни успехи в професията и в живота!** Ще сме невероятно щастливи, ако с наша помощ сте се **запалили по програмирането** и сме ви вдъхновили да поемете смело към професията "софтуерен инженер", която ще ви донесе добра работа, която да работите с удоволствие, качествен живот и просперитет, като и страховити перспективи за развитие и възможности да правите значими проекти с вдъхновение и страсть.



C++

В ръцете си държите нещо повече от **книга за програмиране**, учебник или учебно помагало. Това съвременно образователно пособие ви повежда по **първите стъпки към програмирането** чрез малко текст и **много код**, наситено с примери и внимателно подбрани **практически задачи** със система за моментално **автоматично оценяване**.

Учебното съдържание е разработено лично от **д-р Светлин Наков**, който през 15-годишния си опит с обучението на софтуерни инженери помага на **над 70 000 души** да навлязат в програмирането и намира как да поднася информацията на **малки порции**, с много практика и с **нарастваща сложност**.

Запомнете, че програмиране се учи с **много писане на код и усилено решаване на задачи** и не става само с четене на книги, така че преминете старательно през **упражненията**. Успех!

Сайт: cpp-book.softuni.bg



Авторски колектив:

Андрей Царев
Божидар Колев
Борис Горанов
Венцислав Петров
Георги Лацев
Георги Шавов
Денис Миланов
Димитър Кацарски
Димо Димов
Димо Чанев
Зорница Йоханова
Игнес Симеонова
Йордан Добрев
Любомир Господинов
Мирела Дамянова
Николай Костов
Пламен Динев
Светлин Наков
Цветан Иванов

ISBN 978-619-00-0951-1



9 786190 009511 >