MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE

NATIONAL TECHNICAL UNIVERSITY
"KHARKIV POLYTECHNIC INSTITUTE"

V.Volovshchikov, L.Ivanov, E.Rubin, T.Goncharenko

THE C++ LANGUAGE IN PROGRAMMING AND COMPUTER SCIENCE

Textbook for courses "Fundamentals of programming","Algorithmization and programming"

Edited by prof. M.Godlevsky

Kharkiv
2017

ЗМІСТ

CONTENTS

ВСТУП

На сучасному етапі науково-технічного прогресу все більш важливу роль відіграють інформаційні та програмні технології. Зростає потреба у кадрах, підвищуються вимоги до їхньої кваліфікації. Майбутні фахівці повинні володіти фундаментальними знаннями засад комп'ютерних наук і програмування, а також навичками самостійного навчання, постійного вдосконалення свого професійного рівня. Одночасно зростає потреба у підручниках, які можуть забезпечити можливість як засвоєння основ, так і подальшого самостійного поглиблення знань і вмінь.

Інформаційні та програмні технології стають усе більш інтернаціоналізованими. Іноді розробка великого програмного проекту вимагає залучення фахівців з різних країн та континентів. Запорукою успіху таких проектів є, з одного боку, висока кваліфікація розробників, їхнє вміння працювати в єдиній команді, з іншого – володіння іноземними мовами, зокрема, англійською. Необхідність володіння англійською термінологією пов'язана також з тим, що більшість сучасних джерел інформації в галузі комп'ютерних наук та програмування є англомовними.

З метою підготовки професіональних фахівців, зажаданих на ринку високих технологій, за напрямками «Комп'ютерні науки» та «Програмна інженерія» при вивченні курсів «Основи програмування» відповідно і пропонується даний навчальний посібник, головною особливістю якого є наведення матеріалу двома мовами – українською та англійською.

Посібник присвячено вивченню синтаксису та засвоєнню практичних навичок програмування мовою C++. Ця мова є досить поширеною у сучасному світі програмування. На C++ створюють різноманітні програмні рішення з застосуванням сучасних середовищ програмування.

З іншого боку, більшість сучасних мов об'єктно-орієнтованого програмування (Java, JavaScript, PHP, C#) основана на базовому синтаксисі C++. Вільне володіння C++ може дозволити легше опанувати інші мови програмування.

Навчальний посібник «Мова C++ в програмуванні та комп'ютерних науках» складається з трьох розділів.

INTRODUCTION

At the present stage of scientific and technological progress the importance of information and software technology increases. There is a growing need for skilled workers; furthermore the requirements to their qualification are being stepped up. Future professionals must have a basic knowledge of computer science and programming principles, as well as improve their self-learning skills and enhance continuously their job skills. Simultaneously, the increasing need in the textbooks that can ensure both assimilation framework and further independent improvement of knowledge and skills still exists.

Information and software technology is becoming increasingly internationalized. Sometimes a large software development project requires the involvement of experts from different countries and continents. The key to the success of such projects is, on one hand, highly skilled developers, and their ability to work in a team, on the other hand – the knowledge of foreign languages, especially English. The necessity of English terminology is also connected with the fact that most modern sources of information in computer science and programming are English.

This textbook is offered to train professional specialists that are in demand on the market of high technologies in the areas of "Computer Science" and "Software Engineering" while mastering the courses "Fundamentals of programming". The main feature of the textbook is the material presented in two languages - Ukrainian and English.

The textbook is devoted to the syntax study and mastering practical skills of programming in C++. This language is quite popular in today's world of programming. C++ is used for a creation of software solutions variety using modern programming environments.

Besides that, the most modern languages of object-oriented programming (Java, JavaScript, PHP, C #) are based on the basic syntax of C++. The proficiency in C++ could allow learning other programming languages easier.

The textbook "C++ language in programming and computer science" consists of three chapters.

У першому розділі наведено матеріал щодо базового синтаксису мови програмування C++. Поданий матеріал поєднує три загальних етапи розробки програмного забезпечення: проектування, розробки та тестування.

Другий розділ присвячений процедурно-орієнтованому програмуванню та крім теоретичних основ і класичних практичних прикладів містить матеріал, що ілюструє командний підхід до розробки програм.

У третьому розділі акцентовано увагу на модульному програмуванні, крім того, наводяться загальні відомості про об’єктно-орієнтоване програмування.

Кожне наступне питання, що розглядається, базується на попередніх знаннях, містить велику кількість прикладів, для яких наведено пояснення, контрольні запитання й додаткові практичні завдання.

The first chapter explains material on the basic syntax of the C++ programming language. The submitted material combines three common phases of software development: design, development, and testing.

The second chapter focuses on procedure-oriented programming and, except theoretical fundamentals and classic practical examples, contains the material that illustrates a team approach to programming.

The third chapter is focused on modular programming; in addition it provides an overview of the object-oriented programming.

The considered questions are based on the previous knowledge, they contain many examples for which explanations, test questions and more practical tasks are proposed.

Розділ 1. БАЗОВИЙ СИНТАКСИС МОВИ ПРОГРАМУВАННЯ C++

У розділі розглядаються основні базові елементи мови програмування C++. Пропонується до вивчення проста програма і приділяється увага середовищу програмування Microsoft Visual Studio .NET. Ряд простих, але завершених програм ілюструють використання введених конструкцій.

1.1. Знайомство з мовою програмування C++

Кращий спосіб початку вивчення мови програмування полягає в розборі простої програми і засобів, за допомогою яких дану програму можна реалізувати. Причому важливим також є і розуміння відповідностей між алгоритмом програми і її програмним кодом.

1.1.1. Еволюція підходів до проектування програмного забезпечення

Проектування складних систем різної природи традиційно базується на застосуванні графічного зображення окремих складових частин. Зокрема, графічна нотація (система позначень) може бути використана для подолання складнощів, пов'язаних з алгоритмами. *Алгоритм* являє собою детальний опис послідовності дій, спрямованих на розв'язання визначеної задачі. Мета алгоритму повинна бути досягнута за скінчену кількість кроків.

Традиційною формою подання алгоритмів є так звані блок-схеми. *Блок-схема* складається з функціональних блоків (елементів), послідовний зв'язок яких за допомогою переходів формує алгоритм розв'язання задачі. Рисунок 1.1 показує найбільш часто вживані елементи традиційної блок-схеми

Part 1. THE BASE SYNTAX OF C++ PROGRAMMING LANGUAGE

The section reviews the main basic elements of the C++ programming language. It is proposed to study a simple program and focuses on Microsoft Visual Studio .NET programming environment. Some simple, but completed programs illustrate the use of introduced constructs.

1.1. Acquaintance with the C++ Programming Language

The best way to start learning a programming language is analysis of a simple program and means with the help of which it can be implemented, whereas the understanding of correspondences between program algorithm and its program code is of importance as well.

1.1.1. The Evolution of Approaches to Software Design

The designing of complex systems of different nature is traditionally based on the use of graphic representation of individual components. In particular, the graphical notation (notation system) can be used to overcome difficulties associated with algorithms. The *algorithm* is a detailed sequence of actions aimed at solving a particular problem. The purpose of the algorithm should be reached by finite number of steps.

The traditional form of representation algorithms are so-called flowcharts (block diagrams). *Flowchart* consists of functional elements, which sequential connection using transitions forms algorithm of problem solving. Figure 1.1 shows the most commonly used elements of conventional flowchart.

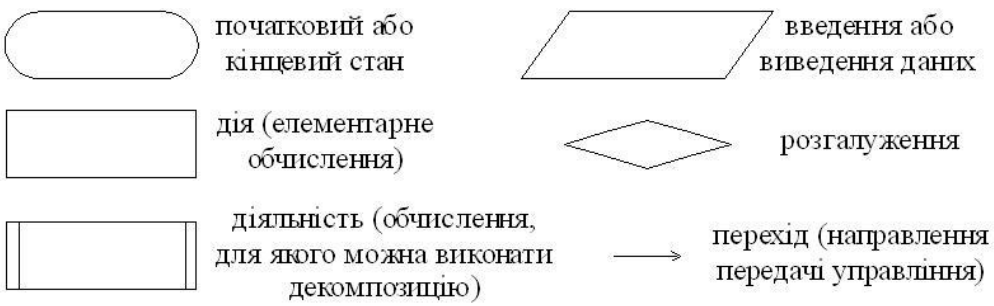

Рисунок 1.1

Потреба у розв'язанні більш складних проблем та розробки відповідних програмних комплексів привела до появи уніфікованої мови моделювання UML (Unified Modeling Language) [1]. UML є графічною мовою для візуалізації, специфікації, конструювання та документування систем, в яких велика роль належить програмному забезпеченню. За допомогою UML можна розробити детальний план системи, що створюється, який відображає не тільки її концептуальні елементи, такі, як системні функції та бізнес-процеси, але й конкретні особливості реалізації. Все це стає можливим завдяки різним видам діаграм. Зокрема, діаграма діяльності застосовується в UML для моделювання динамічних аспектів поведінки системи, а саме для моделювання послідовних (паралельних) кроків обчислювального процесу. Як показано на рисунку 1.2, складовими елементами діаграми діяльності є:

– початковий, кінцевий стани (відповідно заповнений круг та заповнений круг всередині порожнього кола); початковий стан може бути лише один на діаграмі, кінцевих станів – скільки завгодно, але не менш ніж один;

– стани діяльності (овал); це може бути введення, обчислення, виведення і взагалі будь-яка діяльність у межах алгоритму; складні елементи цього типу можуть бути подані за допомогою інших діаграм діяльності;

– перехід (стрілка);

– розгалуження (ромб); як і в блок-схемі, їх використовують для описання різних варіантів розв'язання задачі залежно від деякої умови; на відміну від блок-схем, умова записується не всередині ромба, а на стрілках, які виходять із ромба; у точці об'єднання окремих гілок іноді також розміщують ромб;

– примітка, або анотація; зв'язок у вигляді пунктирної лінії дозволяє визначити належність пояснення до певного елемента діаграми діяльності;

– розподіл та злиття (лінійка); цей елемент застосовується для формалізації паралельних обчислень.

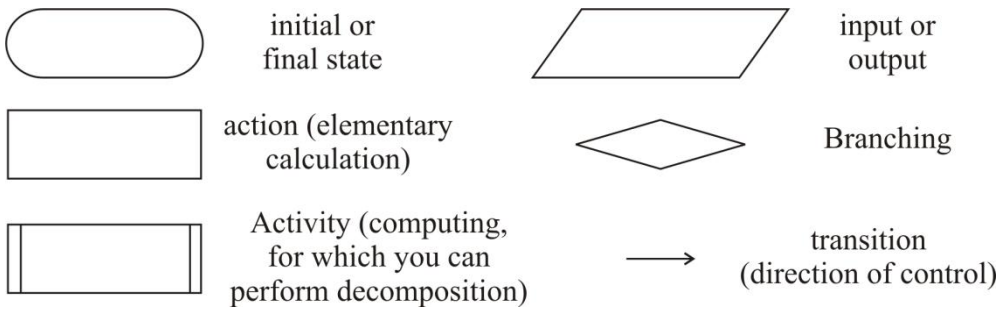

Figure 1.1

The need for solving more complex problems and developing appropriate software packages led to the invention of Unified Modeling Language (UML) [1]. UML is a graphical language for visualizing, specifying, constructing, and documenting of the artifacts of an object-oriented software intensive system under development. UML allows you to develop a detailed plan of system design process that reflects not only its conceptual elements such as system functions and business processes, but also specific of its implementation. This is possible due to different types of diagrams. In particular, Activity diagram is used for modeling the dynamic aspects of system behavior, namely to model sequential (parallel) computing steps of the process. As shown in Figure 1.2 constituent elements of the Activity diagram are:

– initial and final state (as filled circle and filled circle inside a hollow circle, respectively); the initial state can be only on the diagram; it can be several final states, but one at least;

– activity state (oval), it can be input, calculation, output, and generally any activity within the algorithm, complex elements can be represented using other activity diagrams;

– transition (arrow);

– branching (diamond), as in flowchart, they are used to describe the various options for solving the problem, depending on certain conditions, in contrast to the flowchart, the condition is not recorded within the diamond, and the arrows coming from the diamond, sometimes a diamond is placed in a join point of individual branches;

– note (or annotation); relationship as a dotted line allows to determine belonging of explanation to certain element of an activity diagram;

– distribution and merging (bar), this element is used for the formalization of parallel computing.

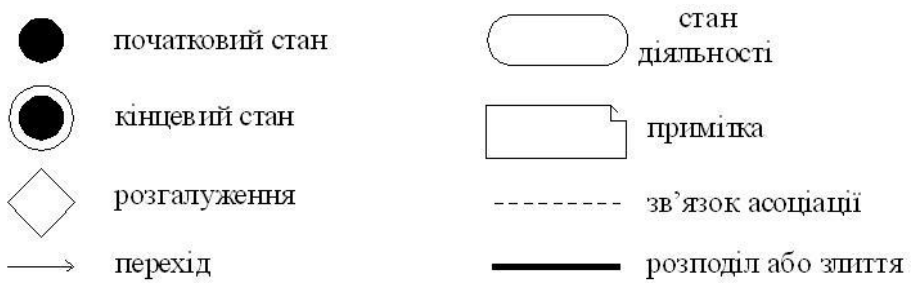

Рисунок 1.2

Три основні підходи до опису алгоритмів можна проілюструвати на прикладі розв'язання задачі виведення рядку "Добридень, світ" (рисунок 1.3).

Рисунок 1.3

1.1.2. Початкове знайомство з мовою програмування С++

Мова програмування – це формальна мова, якою створюються комп'ютерні програми. Програма, створена за допомогою певної мови програмування, зазвичай складається з послідовності інструкцій. Опис мови складається:

– з визначення синтаксису мови;

– визначення семантики.

За своїм рівнем мови програмування класифікуються на низькорівневі та високорівневі.

Низькорівневі мови призначені для конкретного типу комп'ютера і віддзеркалюють його машинні коди; крім мови машинних команд, до низькорівневих мов належить також мова асемблера.

Високорівневі мови не залежать від машинного коду конкретного комп'ютера і дозволяють працювати з абстрактними даними.

З точки зору рівня абстракції даних мови високого рівня можна розділити на дві групи:

– процедурні (не підтримують концепцій класів і об'єктної орієнтації), такі, як PASCAL, BASIC, MODULA 2, C;

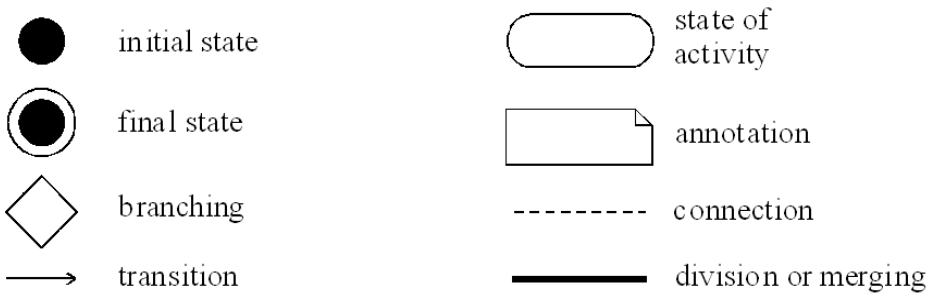

Figure 1.2

Three main approaches to the description of algorithms can be illustrated by the example of solving the problem of output "Hello, Word" string (Figure 1.3).

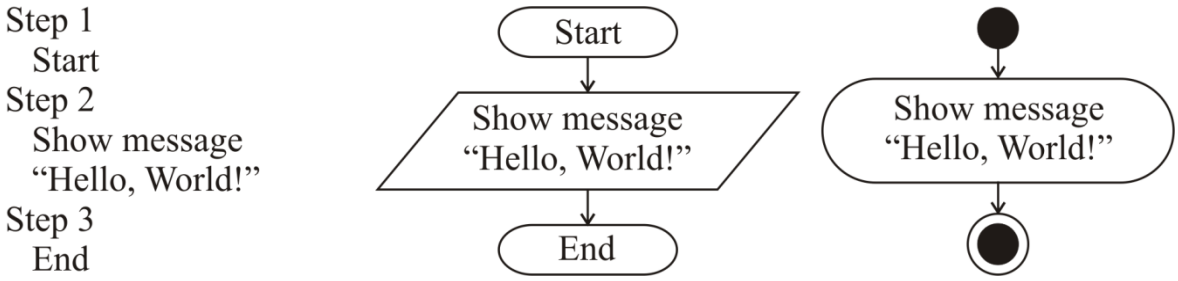

Figure 1.3

1.1.2. Initial Acquaintance with the C++ Programming Language

Programming Language is a formal language, which is used for creation of computer programs. The program created using a programming language typically consists of a sequence of instructions. Language description consists of:

– syntax definition;

– semantics definition.

According to the level, programming languages can be classified as low-level languages and high-level languages.

Low-level languages are designed for a specific type of computer and reflect its machine code, machine instructions as well as assembly language belong to low-level languages.

High-level languages do not depend on the machine code of a particular computer and allow you to work with abstract data.

In terms of the level of data abstraction, high-level languages can be divided into two groups:

– procedural languages (do not support concepts of classes and object orientation), such as PASCAL, BASIC, MODULA 2, C;

– об’єктно-орієнтовані, такі, як Smalltalk, C++, Java, C# та інші.

Текст програми мовою високого рівня називається *вихідним текстом*, або *вихідним кодом*.

Для того щоб підготувати програму для виконання на комп'ютері, потрібні *системи програмування*, або *програмні середовища* – програмні засоби, призначені для створення й трансляції програм, написаних мовою високого рівня, у машинні коди. Прикладами систем програмування є Turbo Pascal, Delphi, MS Visual Studio, Borland C++ Builder та інші.

Основною функцією систем програмування є *трансляція*. *Транслятори* (програми, що виконують цю роботу) можна розділити на дві групи:

– інтерпретатори – програми, що здійснюють послідовне читання, переклад у машинні команди й негайне виконання інструкцій без окремого збереження машинних кодів;

– компілятори – програми, що здійснюють повний переклад вихідного тексту в машинні команди та зберігають цей код в окремому файлі.

Усі транслятори, які обробляють вихідний код на С та С++, реалізовані як компілятори.

Мова програмування С++ була створена в 1983 році Б’єрном Страуструпом. С++ базується на мові С. На відміну від С, С + + є мовою об'єктно-орієнтованого програмування. С++ призначена для підтримки кількох стилів програмування (процедурне програмування, модульне програмування, об'єктно-орієнтоване програмування та узагальнене програмування). Мова С++ поєднує в собі можливості формалізації складних структур даних і написання ефективних низькорівневих програм.

У 1998 році був ратифікований стандарт мови С++. Стандарт С++ являє собою документ, що повністю описує синтаксис мови і його можливе використання. Крім базових засобів мови програмування С++, у стандарт увійшла так звана Стандартна бібліотека (Standard C++ Library).

Алфавіт мови С++ включає:

– латинські літери (A...Z, a...z);

– цифри (0...9);

– знаки операцій і розділювачі: { } [ ] ( ) . , –> & * + - ~ ! / % ? : ; = <.

У С++ розрізняють заголовні та малі літери. Вихідний текст на С++ складається з *лексем*. *Лексеми* діляться на групи: ключові слова, ідентифікатори, розділювачі, коментарі та знаки операцій.

Набір ключових слів обмежений. Іноді окремі слова додаються розробниками певних компіляторів. Ключові слова не можна використати як ідентифікатори. Нижче наводиться список ключових слів С++:

– object-oriented languages, such as Smalltalk, C++, Java, C#, etc.

The text of the program on high-level language is called *source code*.

In order to prepare program for execution on a computer, so-called *programming system*, or *programming environment*, is required - software designed for the creation and translation of programs written in high-level language into machine code. Examples of programming environments are Turbo Pascal, Delphi, MS Visual Studio, Borland C++ Builder, etc.

The main function of programming environment is *translation*. *Translators* (programs that perform this work) can be divided into two groups:

- interpreters - programs that perform sequential read, translated into machine instructions and immediate execution of instructions without saving separate machine codes;

- compilers - programs that perform a complete translation of source code into machine instructions and store this code in a separate file.

All translators which process source code on C and C++ are implemented as compilers.

The C++ programming language was invented in 1983 by Bjarne Stroustrup. C++ is based on C. C is retained as a subset. In contrast to C, C++ is a language of object-oriented programming. C++ is designed to support multiple programming styles (procedural programming, modular programming, object-oriented programming, and generic programming). The C++ programming language joins possibilities of complex data structures formalization and writing efficient low-level programs.

The C++ programming language standard was ratified in 1998. The C++ programming language standard is a document that fully describes the syntax of the language and its possible usage. In addition to the basic means of the C++ programming language, this standard includes so-called Standard C++ Library.

The Alphabet of C++ includes:

– letters of the Roman alphabet (A...Z, a...z);

– digits (0...9);

– operations and separators: { } [ ] ( ). , –> & * + - ~ ! / % ? : ; = <.

In C++ uppercase and lowercase letters are treated as different symbols. The C++ source code consists of tokens. Tokens are divided into following groups: keywords, identifiers, delimiters, comments and operators.

A set of *keywords* is limited. Sometimes developers of certain compilers add extra words as keywords. Keywords cannot be used as identifiers. Here is a list of C++ keywords:

asm	auto	bool	break	case
catch	char	class	const	const_cast
continue	default	delete	do	double
dynamic_cast	else	enum	explicit	export
extern	false	float	for	friend
goto	if	inline	int	long
mutable	namespace	new	operator	private
protected	public	register	reinterpret_cast	return
short	signed	sizeof	static	static_cast
struct	switch	template	this	throw
true	try	typedef	typeid	typename
union	unsigned	using	virtual	void
asm	auto	bool	break	case

Ідентифікатори використаються для іменування змінних, функцій та інших програмних об'єктів. Першим символом повинна бути літера або символ "_". Далі можуть використовуватися також цифри.

Розділювачі – це пропуски, горизонтальна та вертикальна табуляція, перехід на новий рядок. Розділювачами можуть також служити коментарі.

Коментарі – це довільний текст, який ігнорується компілятором. Коментарі можуть містити довільний текст із використанням будь-яких символів. Є два типи коментарів:

– коментар стилю мови C (символи `/*` починають коментар, що завершується символами `*/)`; такі коментарі не можуть бути вкладеними;

– коментар стилю мови C++ (символи `//` починають коментар, що завершується кінцем поточного рядка).

1.1.3. Етапи обробки вихідного тексту

Текст програми міститься в модулях, які мають назву вихідних файлів. У MS Windows це найчастіше файли з розширеннями `.cpp` або `.c`. Вихідні файли містять так званий вихідний код, який є послідовністю інструкцій певною мовою програмування.

З логічної точки зору файл транслюється за кілька проходів. Перший прохід полягає в *препроцесорній обробці*, на якій відбувається обробка вихідного коду (включення або видалення тексту, заміна тощо) відповідно до змісту так званих директив препроцесора. *Директива препроцесора* – це рядок, перший символ якого (відмінний від пробілу) є # (перед ним можуть йти пробіли та символи горизонтальної табуляції). Директиви препроцесора можуть знаходитись у будь-якому місці програми.

asm	auto	Bool	break	case
catch	char	class	const	const_cast
continue	default	delete	do	double
dynamic_cast	else	enum	explicit	export
extern	false	float	for	friend
goto	if	inline	int	long
mutable	namespace	new	operator	private
protected	public	register	reinterpret_cast	return
short	signed	sizeof	static	static_cast
struct	switch	template	this	throw
true	try	typedef	typeid	typename
union	unsigned	using	virtual	void
asm	auto	Bool	break	case

Identifiers are used to name variables, functions, and other objects within a program. The first character must be a letter or underscore character ("_"). In subsequent positions digits can be also used.

Separators are spaces, horizontal and vertical tab symbols, and linefeed characters. Comments also can be treated as separators.

Comments are arbitrary text that is ignored by the compiler. Comments can contain any text; any characters can be used. There are two types of comments:

– C-style comments: slash and star characters (`/*`) start comment that ends with star and slash characters (* /); such comments cannot be nested;

– C++-style comments: two slash characters (`//`) start comment that ends with the end of the current line.

1.1.3. Steps of Source Code Processing

Source code is contained in modules that are called source files. In MS Windows, these are files with extensions .cpp or .c. Source files contain so-called source code, which is a sequence of instructions on specific programming language.

From a logical point of view, file is processed in several steps. The first pass is so called preprocessing, where source code is processed (inclusion or deletion of text replacement, etc.) within the meaning of the so-called preprocessor directives. *Preprocessor directive* is a string whose first non-blank character is `#` (before it can go spaces and horizontal tab characters). Preprocessor directives can be allocated in any place of the program.

Нижче наводяться деякі з директив препроцесора.

Керуючий рядок вигляду

```
#include <ім'я_файлу>
```

приводить до включення вмісту файлу із зазначеним ім'ям – ім'я_файлу. Пошук зазначеного файлу проходить у певній послідовності та визначається реалізацією системи програмування. Аналогічно керуючий рядок

```
#include "ім'я_файлу"
```

приводить до тих самих дій, за винятком того, що пошук цього файлу починається з теки поточного проекту.

Директива вигляду

```
#define ідентифікатор рядків-лексем
```

називається макровизначенням. Вона вказує препроцесору, що треба зробити заміну всіх наступних входжень ідентифікатора на визначену послідовність лексем, яку називають рядком заміни. Після появи макровизначення ідентифікатор з нього вважається таким, що визначений, і залишається в поточній області видимості до кінця одиниці трансляції або поки його визначення не буде скасовано за допомогою директиви #undef. Команда #undef має вигляд:

```
#undef ідентифікатор
```

Вона змушує препроцесор "забути" макровизначення з цим ідентифікатором. Якщо зазначений ідентифікатор не є визначеним у цей момент макроім'ям, то команда #undef ігнорується.

За допомогою препроцесора можна організувати умовну трансляцію програми, для цього використовуваються директиви:

```
#if вираження-константа
#ifdef ідентифікатор
#ifndef ідентифікатор
#elif вираження-константа
#else
#endif
```

За допомогою `#ifdef`, `#ifndef` і `#endif` можна домогтися, щоб частина рядків компілювалася, якщо ідентифікатор визначений (не визначений) раніше за допомогою `#define`.

Some preprocessor directives are shown below.

The following control string

```
#include <filename>
```

results in inclusion of the file contents with the specified filename. Searching a file takes place in a certain sequence determined by the implementation of programming system. The analogous control string

```
#include "filename"
```

results in the same activities except that the search for the file starts with a folder of the current project.

The following directive

```
#define identifier tokens
```

is called macro definition. It forces the preprocessor to replace all subsequent occurrences of the identifier to a specific sequence of tokens, called the replacement string. After the appearance of a macro its identifier is defined, and it remains in the current scope (to the end of the translation unit), or until its definition will be canceled by the #undef directive. The #undef is as follows:

```
#undef identifier
```

This command forces the preprocessor to "forget" macro definition with that identifier. If the specified identifier is not defined, the `#undef` command will be ignored.

Preprocessor allows so-called conditional program translation; the following directives are used:

```
#if constant-statement

#ifdef identifier
#ifndef identifier
#elif constant-statement
#else
#endif
```

Using `#ifdef`, `#ifndef`, and `#endif` we can ensure that certain lines will be compiled, if the identifier was defined (not defined) before using `#define`.

Результатом роботи препроцесора є послідовність *лексем*. Таку послідовність лексем, тобто файл після препроцесорної обробки, називають *одиницею трансляції*. Результатом трансляції цих одиниць у загальному випадку є об'єктні модулі (у MS Windows це файли з розширенням `.obj`). Логічно зв’язані об'єктні модулі іноді групують у так звані бібліотеки (файли з розширенням `.lib`).

Незважаючи на те, що об'єктні модулі містять машинні коди, вони не можуть бути безпосередньо виконані процесором, оскільки вони не містять коду зовнішніх функцій (введення-виведення, математичних тощо). Оскільки без таких функцій не може працювати навіть найпростіша програма, необхідним є ще один етап – так зване *компонування* (linking): збирання з об’єктних модулів та бібліотек файлу, який може бути виконаний процесором (у MS Windows це файли з розширенням `.exe`).

1.1.4. Створення проекту в середовищі програмування Microsoft Visual Studio .NET

Microsoft Visual Studio .NET – це комплект засобів розробки, що включає Visual Basic .NET, Visual C++ .NET і Visual C# .NET, а також підтримку Web-програмування [1, 6]. Три основні мови розробки (Visual Basic NET, Visual C++ .NET і Visual C# .NET) використовують загальне середовище розробки (Integrated Development Environment, IDE), що, зокрема, сприяє створенню проектів, які комбінують кілька мов програмування.

При першому завантаженні IDE користувачеві пропонується стартова сторінка (Start Page). За умовчанням розкривається закладка "Почати роботу" (Get Started). На панелі цієї закладки пропонується список проектів, які відкривалися останнім часом. Тут також є кнопки "Відкрити проект" (Open Project) і "Новий проект" (New Project).

Ключовим поняттям розробки прикладної програми у Visual C++ є проект (project). Проект – це набір взаємозалежних вихідних файлів, компіляція й компонування яких дозволяє створити програму, що виконується, або DLL (Dynamic link library, бібліотека, яка динамічно компонується). Вихідні файли проекту звичайно зберігаються в окремому підкаталозі. Крім того, найчастіше проект залежить від багатьох файлів, розташованих поза підкаталогом проекту, таких, як бібліотечні файли та інші.

Проекти можна структурувати шляхом об’єднання їх у так звані *рішення* (Solutions) – логічно взаємозв’язані групи проектів.

The result preprocessing is a sequence of tokens. Such a sequence of tokens obtained after preprocessing is called *translation unit*. In general case, the result of such units' translation is a creation of so-called object files (in MS Windows these files with the `.obj` extension). Logically linked object modules are sometimes grouped in the so-called libraries (files with `.lib` extension).

Despite the fact that object files contain machine code, they cannot be directly executed by a processor, since they do not contain code of external functions (input and output, math, etc.). Since even the simplest program may not work without these functions, another essential step is needed the so-called *linking*: collecting of object modules and libraries into a file that can be executed by a processor (in MS Windows, there are files with the `.exe` extension).

1.1.4. Creating a new project in the Microsoft Visual Studio .NET programming environment

Microsoft Visual Studio .NET is a set of development tools, including Visual Basic .NET, Visual C++ .NET, and Visual C# .NET, as well as support for Web programming [1, 6]. The three main development languages (Visual Basic .NET, Visual C++ .NET, and Visual C# .NET) use common Integrated Development Environment (IDE), which in particular supports creation projects that combine multiple programming languages.

When you first load the IDE, Start Page is offered. By default, the tab called "Get Started" opens. The panel of this tab offers the list of recently opened projects. There are also "Open Project" and "New Project" buttons.

The key concept of applications development in Visual Studio is a project. A project is a set interconnected source files, which compiling and linking allows to create a program that can be executed or DLL (Dynamic link library, a library that is dynamically linked into program). Project source files usually stored in a project subdirectory. In addition, the project often depends on many files located outside the project subdirectory such as library files and others.

Projects can be structured by combining them in so-called *Solutions*, which are logically interrelated groups of projects. In the simplest case, each project can be created in a separate solution.

У найпростішому випадку для кожного проекту створюється окреме рішення

Існує поняття конфігурації проекту. Кожний проект містить як мінімум дві конфігурації – Debug і Release.

Новий проект може бути створений декількома способами:

– з використанням стартової сторінки;

– з використанням головного меню;

– з використанням кнопки швидкого доступу.

Наша мета – створити проект, що реалізує консольний додаток на С++. Після вибору функції створення нового проекту виконують такі дії:

– на панелі "Project Types" вибрати тип проекту Visual C++ Projects;

– на панелі "Templates" вибрати "Win32 Project";

– у рядку введення "Name" обов'язково потрібно ввести ім'я проекту.

Розташування проекту за умовчанням пропонується автоматично, однак його можна змінити;

– натиснути кнопку "OK" для запуску майстра проекту (Project Wizard);

– у вікні майстра проекту варто вибрати закладку "Application Settings" (Установки додатка);

– на закладці "Application Settings" встановити прапорці "Console application" (консольний додаток) і "Empty project" (порожній проект);

– натиснути кнопку "Finish".

Наступним етапом необхідно створити новий файл із вихідним текстом. Для цього потрібно виконати наступне:

– у підменю "Project" головного меню вибрати "Add New Item";

– на панелі "Templates" вікна "Add New Item" вибрати тип файлу C++ File; потрібно також увести ім'я файлу з вихідним текстом у рядку введення "Name";

– натиснути "OK", після чого відкривається новий порожній файл.

IDE містить *редактор вихідного тексту*, що підтримує велику кількість можливостей: виділення кольорами синтаксичних конструкцій, розкладки клавіатурних команд та інші. Наприклад, функція AutoComplete пропонує вибрати «все слово» зі списку можливих варіантів після набору декількох початкових символів оператора програми.

У порожній файл необхідно записати програмний код, який є відображенням діаграми діяльності, що забезпечує розв’язання деякої задачі.

Для виконання програми можна використати клавішну комбінацію Ctrl-F5.

За умов, коли програма не містить помилок, натискання комбінації клавіш Ctrl-F5 приведе до запуску програми.

In the simplest case, a separate solution is created for each project.

There is a concept of project configuration. Each project contains at least two configurations: Debug and Release.

New project can be created in several ways:

– using the Start page;

– using main menu;

– using speed button.

Our goal is to create a project that implements a C++ console application. After selecting “New project…” function, you should perform the following steps:

– choose project type “Visual C++ Projects” on the “Project Types” pane;

– choose project type “Win32 Project” on “Templates” pane;

it is necessary to enter the project name in the "Name" field.

The location of the project is proposed by default automatically, but you can change it;

– press "OK" button to start Project Wizard;

– within Project Wizard, choose "Application Settings" tab;

– check “Console application” and “Empty project” options on "Application Settings" tab;

– press "Finish" button.

The next step is creation of a new file with the source text. To do this, perform the following:

– choose “Add New Item” in “Project” submenu;

– choose “C++ File (.cpp)” on “Templates” pane within “Add New Item” window; you must also enter the name of the source file in the "Name" input field;

click "OK", and then a new empty file will be open.

IDE contains *source code editor* that supports many features: syntax highlighting, keyboard layout commands and others. For example, the “AutoComplete” function allows choosing of whole word from the list of variants after you type several initial characters of the operator.

Here you can type the code that reflects activity diagram, which provides solution of some problem.

To run the program, you can use keyboard shortcut Ctrl-F5.

If our program has no errors, it can be normally executed after pressing Ctrl-F5.

Але трапляються випадки, коли в написаній програмі існують помилки. Всі помилки можна розділити на два класи: *синтаксичні* та *логічні*. До *синтаксичних помилок* віднесемо помилки, пов'язані, наприклад, з некоректним записом ключових слів, порушенням правил створення імен, неприпустимим розташуванням окремих інструкцій тощо. Такі помилки найбільш легко виправляються, бо їх перелік стає відомим на етапі компіляції проекту. Доти, поки всі синтаксичні помилки не будуть виправлені, програму виконати неможливо. Найбільш складними помилками є *логічні помилки*. Останні не впливають на неможливість виконання програми, але призводять до некоректного її виконання.

Іноді для пошуку логічних помилок може стати в нагоді налагоджувач (Debugger). *Налагоджувач* використовується для пошуку та виправлення помилок часу виконання. Для запуску налагоджувача використовується функція Debug|Start (F5).

Попередньо у вихідному тексті розставляються точки переривання (функція Debug | New Breakpoint або Ctrl-B). Для виконання програми рядок за рядком використовуються функціональні клавіші F11 (Step Into – із заходженням у функції) і F10 (Step Over, без заходження у функції).

Після запуску налагоджувача змінюється зовнішній вигляд і склад вікон інтегрованого середовища розробки. Відповідні кнопки на панелі інструментів дозволяють розставляти та прибирати точки переривання й керувати покроковим виконанням програм. Замість точок переривання можна використати функцію "Виконувати до курсора" (Ctrl-F10).

У вікнах Autos, Locals і Watch можна переглядати зміну значень змінних. Якщо помістити курсор миші на скалярну змінну, налагоджувач покаже її значення в маленькому віконці.

1.1.5. Приклад програми

Приклад 1. Як перший приклад програми наведемо програму, діаграма діяльності якої показана на рисунку 1.3.

```
01 // Програма виведення привітання “Добридень, світ”
02       #include <iostream>
03       using namespace std;

04       int main(int argc, char* argv[])
05       {
06           cout << " Добридень, світ "<< endl;
07           return 0;
08       }
```

However, there are cases when the written program has errors. All errors can be divided into two classes: syntactic and logical. *Syntax errors* are such errors as incorrect recording of keywords, violation of the rules of creating names, invalid usage of some instructions etc. Such errors are most easily corrected, because their list is known at compile time. Until all syntax errors are corrected, the program cannot be started. The most difficult errors are *logical errors*. They do not affect the inability of starting program, but it leads to incorrect execution.

Sometimes, to find logic errors, you need to use so called Debugger. *Debugger* is used to find and fix a runtime error. Use Debug | Start (F5) menu function to start debugger.

Before starting debugger, you should place breakpoints in the source code (function Debug | New Breakpoint or Ctrl-B). To run the program line-by-line, use key shortcut F11 (Step Into - execute with entry into functions) or F10 (Step Over, execute without entry into functions).

After you start debugger, the appearance and structure of the IDE window changing in some way. The corresponding buttons on the toolbar allow you to place, remove breakpoints, and allow stepping through control programs. Instead of breakpoints, you can use "Run to Cursor" (Ctrl-F10).

The Autos, Locals, and Watch views allow you to watch how change values of variables. If you put your mouse over a scalar variable, debugger displays its value in a small window.

1.1.5. Sample program

Example 1. As a first example we give an application, the diagram of which is shown in Figure 1.3.

```
01 // This program outputs greeting "Hello, World!"
02       #include <iostream>
03       using namespace std;

04       int main(int argc, char* argv[])
05       {
06           cout << "Hello, World!" << endl;
07           return 0;
08       }
```

Примітки:

1. Тут і далі в прикладах зарезервовані слова виділяються жирним шрифтом.
2. Номера рядків не набираються. Текст набирається з першої позиції екрана з наступними відступами, де це необхідно.
3. Використання в консольному додатку букв кирилиці утруднено розходженням кодувань, прийнятих у віконних і консольних програмах. Для повідомлень, які виводяться на екран, доцільне використання англійської мови.

Рядок `01` являє собою коментар, що визначає призначення програми. Зміст коментарю може бути довільним.

Рядок `02` – директива препроцесора. Директива `#include` вказує на необхідність підключення файлу `iostream` при компіляції.

У програму підключаються описи, необхідні для роботи стандартних потоків введення-виведення. Зокрема, компіляторові відомим стає ім'я `cout`, оголошене в даному файлі.

У рядку `03` здійснюється підключення так званого простору імен. У цьому випадку підключається простір імен std.

Рядок 04 являє собою заголовок функції `main()`. У кожній програмі на С++ повинна бути своя функція `main()`, програма починається з виконання цієї функції. Зарезервоване слово **`int`** на початку рядка – тип результату функції (у цьому випадку – цілий). Результат функції main() використовується операційною системою. Параметри функції (int `argc`, `char* argv[]`) можуть бути використані в програмі для аналізу вмісту командного рядка. У даному прикладі ці параметри не використовуються. Починаючи з `05` рядка виконується перетворення діаграми діяльності (рисунок 1.3) на відповідний програмний код.

Фігурна дужка, що відкривається, у рядку `05` вказує на початок групи тверджень (операторів), які належать до функції `main()` і є аналогом початкового стану діаграми діяльності . Дану групу завершує фігурна дужка, що закривається, у рядку `08`. Остання фігурна дужка відповідає кінцевому стану діаграми діяльності.

Починаючи з `06` рядка, виконується перетворення діаграми діяльності (рисунок 1.3) на відповідний програмний код за виключенням початкового та кінцевого станів.

Оскільки діаграма діяльності містить тільки стан дії, який забезпечує виведення фрази "`Добридень світ`", то в рядку використовується потік `cout`, операція помістити в потік `<<` та неіменована константа.

Notes:

1. Hereinafter in examples, the reserved words are bolded.

2. Line number is not typed. Text is typed at the first position of the editor window with the following indents where it is necessary.

3. Using of Cyrillic letters in console applications is difficult because of different encodings adopted in window and console applications. For messages that are displayed on the screen, the use of English is more appropriate.

Line 01 contains a comment that defines the purpose of the program. The comments can be arbitrary.

Line 02 represents a preprocessor directive. According to `#include` directive, preprocessor inserts the text of a given header file into a program.

The standard header file iostream contains declarations of standard input and output. In particular, the compiler becomes known name `cout`, declared in the file.

Line `03` carries out connection to so-called namespace. In this case, we connect to a namespace called `std`.

Line 04 is the header of function called `main()`. Every program in C++ should have its own `main()` function; the program starts with this function. The **`int`** reserved word at the beginning of the line is so called returning type of the function (in this case, result is integer). The result of `main()` function is used by the operating system. Function parameters (**`int`** `argc`, **`char`** `* argv []`) can be used in the program to analyze content of command line. In this example, these parameters are not used.

Starting with line `05` the conversion of activity diagram (figure 1.3) into the corresponding code is carried out.

The brace that opens in the line `05` indicates the beginning of a group of statements that relate to the `main()` function and is analogous to the initial state of the activity diagram. This group completes the brace, closing at line `08`. This brace corresponds to the end state of an activity diagram.

Starting with line `06` the conversion of an activity diagram (figure 1.3) into the corresponding code is executed except for the initial and final states.

Since the activity diagram contains only action state that provides output of sentence `"Hello, world"`, the line contains "put into the stream (<<) operation that puts unnamed constant into `cout` stream.

Занесення в потік виведення маніпулятора `endl` забезпечує переведення курсора на новий рядок.

Твердження **`return`** 0 урядку 07 повертає операційній системі значення 0, що означає успішне завершення програми.

Слід зауважити, що перетворення діаграми діяльності у відповідний програмний код починається з рядка `05`. Більш детальний опис рядків `01-04` та `07-08` буде наведено нижче після знайомства з наступними елементами мови програмування C++.

1.1.6. Вправи для самостійної роботи

Вправа 1. Напишіть програму, яка виводить два будь-яку фразу з будь-якої кількості слів таким чином, аби кожне слово розташовувалось на окремому рядку.

Вправа 2. Напишіть програму, яка виводить будь-яке слово таким чином, щоб кожна окрема буква розташовувалась на окремому рядку.

Контрольні запитання

1. Підходи до проектування схеми розв’язання задачі дослідження.
2. Основні елементи діаграми діяльності та їхнє призначення.
3. Визначити поняття вихідного тексту програми та навести основні етапи обробки вихідного тексту.
4. Порядок створення проекту в середовищі програмування Microsoft Visual Studio .NET.
5. Порядок налагодження програми.

1.2. Операції та інструкції

Написання програми для вирішення конкретної задачі вимагає коректного використання обгрунтованих правил побудови програми. Фундамент цих правил зосереджений довкола таких понять, як типи даних, змінні, константи, вирази, операції і інструкції.

1.2.1. Типи даних

З точки зору комп’ютерного подання усі дані, незалежно від мови програмування, операційної системи та класу комп’ютерів, можна розділити на дві групи:

Adding `endl` manipulator to the output stream provides moving the cursor to the next line.

The **`return`** 0 statement on line 07 returns zero value to the operating system, which means the successful completion of the program.

It should be noted that the reflection of activity diagram to the corresponding source code starts with line `05`. A more detailed description of lines `01-04` and `07-08` will be shown below after you get more acquaintance with the appropriate elements of the C++ programming language.

1.1.6. Exercises for self-study

Exercise 1. Write a program that prints any sentence from any number of words so that each word appears on separate lines.

Exercise 2. Write a program that prints any word so that every single letter will be located on a separate line.

Advancement Questions

1. The approaches to designing scheme of solving the research problems.
2. The key elements of the Activity diagram and their purpose.
3. Define the concept of source code and list basic stages of processing source code.
4. The order of projects creation in the Microsoft Visual Studio .NET programming environment.
5. The procedure for program debugging.

1.2 Operations and Statements

Writing a program for solving a particular problem requires correct use of reasonable rules of construction applications. The foundation of these rules is centered around concepts such as data types, variables, constants, expressions, and operations, and statements.

1.2.1. Data Types

In terms of computer representation, all data, regardless of programming language, operating system and type of computer, can be divided into two groups:

– цілі;

– дійсні.

Ці дві групи також мають назви даних з фіксованою та плаваючою крапкою.

Дані цілого типу можуть бути звичайними цілими зі знаком (**signed**) і цілими без знака (**unsigned**). За числом розрядів, які використовуються для подання даних (діапазону значень), розрізняють звичайні цілі (**int**), короткі цілі (short int) та довгі цілі (**long int**). У 32-бітних операційних системах значення типу **short** займає 2 байти, а **long** – 4 байти (як і **int**).

Серед цілих типів окремо виділяють логічний і символьний типи.

До *логічного типу даних* належить тип bool. Можливими значеннями цього типу є **true** та **false**. Однак замість них у C++ можна використовувати 1 та 0. Взагалі, будь-яке значення, відмінне від 0, інтерпретується як істинне, 0 – хибне.

Дані символьного типу (char) мають подвійну природу, а саме: можуть розглядатися як цілі (коди символів) і бути зі знаком і без знака, а також як символи, наприклад '#'.

Дійсні числа можуть бути значеннями одного з трьох типів: **float**, **double**, **long double**. Діапазон значень цих типів залежить від системи програмування, яка використовується.

Для подання незмінних значень у мові програмування C++ існує два різновиди *констант – неіменовані та іменовані*, які поєднані в чотири групи. Три групи з чотирьох відповідають першим трьом основним типам даних, а четверта містить константи-рядки. Спочатку розглянемо неіменовані константи.

Константи цілого типу записуються як послідовності десяткових цифр. Тип константи залежить від числа цифр у записі константи і може бути уточнений додаванням наприкінці константи букв `L` чи `l` (тип **long**), `U` або `u` (тип **unsigned**) або у сполученні. Цілі константи можуть записуватися у вісімковій системі числення, у цьому випадку першою цифрою повинна бути цифра 0, число може містити тільки цифри 0...7. Цілі константи можна записувати й у шістнадцятковій системі числення. У цьому випадку запис константи починається із символів 0x чи 0X. Для позначення цифр понад 9 використовуються латинські букви `a`, `b`, `c`, `d`, `e` та `f` (великі або маленькі).

Константи дійсних типів можуть записуватись у формі з плаваючою точкою або в експонентному форматі і за умовчанням мають тип **double** (число з плаваючою точкою подвійної точності). За необхідності тип константи можна уточнити, записавши наприкінці

– integer (whole) numbers
– floating (real) numbers

These two groups are also called fixed-point and floating-point data.

Integer data can be conventional integral with the sign (**signed**) and without sign (**unsigned**). According to the number of bits used to represent the data (value range), we distinguish regular integers (**int**), short integers (short int) and long integers (**long int**). In 32-bit operating systems, the value of the short requires 2 bytes, and **long** requires 4 bytes (like **int**).

Among the integer types can be distinguished logical and character types.

The only *logical data type* is **bool**. The possible values of this type are **true** or **false**. However, instead of them, C++ allows you to use 1 and 0. Generally, any value other than 0 is interpreted as **true**, and zero value is **false**.

Data of *character type* (**char**) have a dual nature, namely, they can be considered as integers (character codes), and can be signed and unsigned, as well as characters, such as '#'.

Values of *real numbers* can be one of three types: **float**, **double**, and **long double**. The range for these types depends on the programming system used.

Constant values in C++ programming language can be represented in two ways: *named* and *unnamed*. Unnamed constants are arranged in four groups. Three groups correspond to the first three basic types of data, and the fourth group contains string constants. We consider the unnamed constants first.

Integer constants are written as sequences of decimal digits. The type of constant depends on the number of digits in the constant representation and may be revised by adding postfixes at the end the constant: letters `L` or `l` (type long), `U` or `u` (type unsigned) or combination of these letters. Integer constants can be written in octal notation; in this case, the first digit must be 0, the number can only contain digits 0 ... 7. Integer constants can be written in hexadecimal. In this case, write the constant starts with 0x or 0X notation. In order to denote digits over 9, letters `a`, `b`, `c`, `d`, `e`, and `f` (capital or small) are used.

Constants of real types can be written in the form of floating point or exponential format; by default, they are of type **double** (floating point double precision). If necessary, type of constant can be specified, by recording at the end of it

Суфікс `f` чи `F` для типу **`float`**, суфікс `l` чи `L` для типу **`long double`**. Наприклад:

```
1.5f    // 1.5  типу float
2.5E-2d // 0.25 типу double
```

Для визначення *констант логічного типу* використовують ключові слова true та false.

Константи символьного типу (**`char`**) беруть в одиночні лапки (апострофи), значення константи визначається знаком з поточного набору символів або цілою константою, якій передує зворотна коса риска (символ із заданим кодом). Є ряд спеціальних символів, що можуть використовуватись як значення константи типу char (такі подвійні символи називаються керуючими послідовностями, або escape-послідовностями):

`'\n'` – новий рядок,
`'\t'` – горизонтальна табуляція,
`'\v'` – вертикальна табуляція,
`'\r'` – переведення на початок рядка,
`'\f'` – нова сторінка,
`'\a'` – звуковий сигнал,
`'\''` – одиночні лапки (апостроф),
`'\"'` – подвійні лапки,
`'\\'` – зворотна коса риска (backslash).

Константа-рядок складається із символів, які беруть у подвійні лапки. Наприклад:

```
"Це рядок"
```

Константа-рядок може включати керуючі послідовності. Якщо між двома константами нічого немає, окрім пропусків та переходів на новий рядок, компілятор поєднує їх в одну. Константа-рядок подається масивом символів, що містить, крім символів рядка, символ з кодом 0 (`'\0'`).

1.2.2. Оголошення та визначення змінних. Вирази та операції

Змінна, чи об'єкт, – це іменована область пам'яті, до якої є доступ у програмі. Кожна змінна має певний тип, що визначає розмір та внутрішню організацію цієї області пам'яті, діапазон значень, які вона може зберігати, і набір операцій, які можуть бути застосовані до цієї змінної.

Suffixes `f` or `F` are for type **`float`**, suffixes `l` or `L` for **`long double`**. For example:

```
1.5f    // 1.5  of type float
2.5E-2d // 0.25 of type double
```

To determine the *Boolean constants*, keywords true and false are used.

Character constants (**`char`**) are written in single quotes (apostrophes); the value is set whether using characters of the current character set or using an integer constant preceded by backslash (so we get character with specified code). There are a several special characters that can be used as constant values of type char (these double characters are called escape-sequences):

`'\n'` – new line,
`'\t'` – horizontal tab,
`'\v'` – vertical tab,
`'\r'` – jump at the beginning of line,
`'\f'` – new page,
`'\a'` – Bell (alert),
`'\''` – single quote (apostrophe),
`'\"'` – double quote,
`'\\'` – backslash character itself.

A *string constant* consists of characters enclosed in double quotes. e.g.:

```
"This is string"
```

String constant can contain escape sequences. If the two constants are separated by blanks and new lines, the compiler combines them into one. Constant string is represented by array of characters that contains, in addition to string characters, a character with code 0 (`'\0'`).

1.2.2. Declaration and definition of variables. Expressions and Operations

Variable (or object) is a named region of memory that can be accessed within program. Each variable has its own type, which determines the size and internal organization of this region of memory, a range of values that it can keep, and a set of operations that can be applied to this variable.

Таким чином, зі змінною пов'язується її значення, яке зберігається в певному місці пам'яті (rvalue – праве значення, у присвоюванні стоїть праворуч), і її місце розташування, тобто адреса в пам'яті, за якою зберігається її значення (lvalue – ліве значення).

Змінна може бути *оголошена* і повинна бути *визначена*. Синтаксис оголошення змінних буде розглянуто нижче у контексті створення заголовних файлів.

Визначення змінної викликає виділення пам'яті. Визначення задає ім'я змінної та її тип. Крім цього, може бути зазначене ініціююче значення для змінної, яке може бути достатньо складним виразом. Повинно бути одне і тільки одне визначення змінної в програмі. Змінна не може використовуватись до її визначення:

```
int a = 10; // Змінна цілого типу
            // з початковим значенням 10
int b(10);  // Те ж саме
```

Якщо необхідно визначити кілька змінних однакового типу, їхні ідентифікатори записуються через кому:

```
int x, y;  // Дві цілих з невизначеними значеннями
```

Оголошення і визначення спільно називаються описами. Слід зауважити, що коли в описах відсутнє ім’я типу але присутні інші модифікатори (long, short, signed, unsigned), припускається, що змінна має тип int. Наприклад

```
short s;      // коротке ціле
unsigned s;   // беззнакове ціле
```

Для подання константних величин можуть використовуватися так звані іменовані константи. Для визначення *іменованої константи* перед ім'ям типу розташовують ключове слово const, іменованій константі потрібно визначити ініціююче значення. Значення іменованої константи не можна змінювати в програмі. Приклад опису константи:

```
const int k = 100;
```

Якщо під час опису ім’я типу було пропущене, константа вважається цілою. Наприклад:

```
const n = 2; // те саме, що const int n = 2;
```

Слід запобігати подібному стилю опису констант.

Вираз складається з однієї чи кількох *операцій*. Об'єктами операцій є *операнди*. Залежно від кількості операндів виділяють унарні операції

Thus, variable is associated with a value stored in a memory location (rvalue - right value in the assignment expression), and its location, that is the address in memory at which it is stored value (lvalue - left value).

Variables can be *declared* and must be *defined*. Syntax for declaring variables will be discussed below in the context of header files.

Definition of variable causes memory allocation. Definition specifies the name of a variable and its type. Furthermore, an initial value for the variable can be specified, which can be quite a complicated expression. There must be one and the only definition of a variable in the program. A variable cannot be used before its definition:

```
int a = 10; // Variable of integer type
            // with an initial value of 10
int b(10);  // The same
```

If you define multiple variables of the same type, their identifiers are written separated by commas:

```
int x, y;  // Two integers with undefined values
```

Declaration and the definition are jointly called descriptions. Note that when you omit type name but add other modifiers (long, short, signed, unsigned), it is assumed that the variable is of int type. For example:

```
short s;     // short int
unsigned s;  // unsigned int
```

So-called *named constants* can be used for presentation of constant values. To define named constant, the const keyword is placed before the name of the type; named constant requires initial value. The value of the named constants cannot be changed in the program. An example of constant definition:

```
const int k = 100;
```

If the type name has been omitted in definition, the constant is integer by default. For example:

```
const n = 2; // the same as const int n = 2;
```

You should avoid such style of descriptions.

The *expression* consists of one or more *operations*. The objects of operations are *operands*. Depending on the number of operands, unary

(один операнд), бінарні операції (два операнди) і тернарні операції (три операнди).

Розглянемо унарні та бінарні операції, серед який виділяють: *арифметичні*, *операції відношення*, *логічні операції* та *операції присвоювання*. Далі у лапках наводятся відповідні знаки операцій.

До арифметичних операцій належать:

– бінарні операції додавання «+», віднімання «-», множення «*», ділення «/» та отримання залишку від ділення «%» (тільки для цілих). Якщо «/» застосовується до цілих операндів, результатом ділення буде ціле, а залишок відкидається;

– унарні операції підтвердження «+» та зміни знака «–» (застосовуються зліва від операнда);

– унарні операції інкременту та декременту «++» та «--» (забезпечують збільшення або зменшення цілої змінної на одиницю).

Операції інкремента та декремента мають дві форми: префіксну і постфіксну. Префіксна форма забезпечує збільшення або зменшення змінної до того, як значення операції буде використано, а постфіксна – після.

До *операцій відношення* належать перевірка на рівність «==» та на нерівність «!=», а також перевірки «>» (більше), «>=» (більше або дорівнює), «<» (менше), «<=» (менше або дорівнює).

До *логічних операцій* належать логічне І – «&&» та АБО – «||». Операції відношення та логічні повертають значення типу **`bool`** (**`true`** – якщо умова правильна, і **`false`** – якщо умова хибна).

Логічне І повертає **`true`**, якщо обидва операнди істинні, та **`false`** в іншому випадку. Логічне АБО повертає **`false`**, якщо обидва вирази хибні, та **`true`** в іншому випадку. Логічне НІ повертає **`false`**, якщо операнд істиний, та **`true`**, якщо операнд хибний.

Тип **`bool`** сумісний з цілим типом: істиною вважається будь-яке ненульове значення, а хибністю – нуль. Будь-який вираз, що приводиться до цілого, може використовуватися в логічних умовах. Логічне заперечення ! перетворить свій операнд в істину, якщо він дорівнював 0, і в хибність, якщо він не дорівнює нулю.

До *операцій присвоювання* належать операції простого і складеного присвоювання. Результатом операції простого присвоювання є значення того виразу, що присвоюється лівому операнду.

Складене присвоювання можна подати в загальному вигляді таким чином:

```
a op= b;
```

У цьому випадку `op` – арифметична операція: `+ - * / %`.

operations (one operand), binary operations (two operands) and ternary operation (three operands) are distinguished .

Consider unary and binary operations, which are grouped into arithmetical, relational, logical and assignment operations. Further, the corresponding operators are given in quotation marks.

Arithmetic operations include the following:

– binary addition "+", subtraction "-", multiplication "*", division "/", and a remainder of the division "%" (for integers). If "/" is applied to integer operands, the result is an integer division and the remainder is discarded;

– unary operations of confirmation "+" and change the sign "-" (placed before operand);

– unary increment and decrement operations "++" and "--" (providing an increase or decrease of the integer variable by one).

The increment and decrement operations have two forms: prefix and postfix. Prefix form provides an increase or decrease value of variable before this value will be used in expression, and postfix form modifies the value after its usage.

The relational operations include testing for equality "==" and on inequality "!=", as well as checking ">" (more than), ">=" (greater or equal), "<" (less than), "<=" (less than or equal to).

The logical operations include logical AND "&&" and OR "||". Relational and logical operations return a value of **bool** type (**true** and **false**).

A logical AND evaluates **true** if both expressions are **true**. Otherwise, it evaluates **false**. A logical OR evaluates **false** if both expressions are **false**. Otherwise, it evaluates **true**. A logical NOT evaluates **false** if the expression being tested is **true**. It evaluates **true** if the expression being tested is **false**.

The **bool** type is compatible with integer type: the truth is any non-zero value, and falsehood is zero. Any expression reducible to an integer can be used in logical expressions. Logical negation ! transform its operand to the truth, if it is 0, and false if it is nonzero.

Assignment operations are operations of simple and compound assignment. The result of a simple assignment operation is the value of the expression that is assigned to the left operand.

Compound assignment can be represented in general form as follows:

```
a op= b;
```

In this case, op is arithmetic operation: + - * / %.

Кожна складена операція еквівалентна наступному присвоюванню:

```
a = (a) op (b);
```

Наприклад,

```
x += 5;
```

що еквівалентно

```
x = x + 5;
```

Крім наведених вище прикладів унарних та бінарних операцій, існує ще декілька стандартних базових операцій.

Операція *sizeof* повертає розмір (у байтах) виразу чи специфікації типу. Цю операцію можна використовувати в двох варіантах:

```
sizeof(тип)
sizeof вираз
```

Наприклад, наступними двома способами можна визначити розмір, який займає об'єкт типу **double**:

```
int size = sizeof(double);
```

або

```
double d;
int size = sizeof d;
```

Операція "кома" дозволяє групувати два або більше виразів. Ці вирази обчислюються зліва направо. Типом і значенням результату є тип і значення правого (останнього) виразу. Наприклад, запис:

```
x = (i = 0, j = i + 4, k = j);
```

еквівалентний запису:

```
i = 0; j = i + 4; k = j; x = k;
```

У мові програмування C++ є одна *тернарна* (з трьома операндами) умовна операція, яка має такий вигляд:

```
умова ? вираз1 : вираз2
```

Спочатку обчислюється значення умови. Якщо воно істинне, то обчислюється вираз1 і його значення повертається умовною операцією. Якщо значення умови хибне, то обчислюється вираз2 і повертається його значення.

C++ підтримує так звані побітові операції, які застосовують до окремих бітів. Вони схожі на логічні операції, але багато в чому відрізняються.

Each compound assignment is equivalent to the following assignment:

```
a = (a) op (b);
```

For example,

```
x += 5;
```

which is equivalent to

```
x = x + 5;
```

In addition to the above examples of unary and binary operators, there are several standard basic operations.

The *sizeof* operation returns the size (in bytes) of an expression or type specification. This operation can be used in two ways:

```
sizeof(type)
sizeof expression
```

For example, the following two expressions determine the size that is an object of type **double**:

```
int size = sizeof(double);
```

or

```
double d;
int size = sizeof d;
```

The *comma* operation allows you to group two or more expressions. These expressions are evaluated from left to right. The type and value of the result are the type and value of the most right-hand (last) expression. For example, the expression:

```
x = (i = 0, j = i + 4, k = j);
```

is equivalent to

```
i = 0; j = i + 4; k = j; x = k;
```

The C++ programming language supports the only *ternary* conditional operation (with three operands), which is as follows:

```
condition ? expression1 : expression2
```

At first, value of condition should be calculated. If it is true then expression1 will be calculated and its value will be returned by conditional operation. If the condition is false, then the expression2 will be calculated and its value will be returned.

C++ supports the so-called *bitwise operations* that are applied to individual bits. They look like logical operations, but differ in many ways.

Операнди побітових операцій повинні бути цілими (бажано ціле без знака). Наприклад,

```
unsigned char c1 = 168;      // 10101000
unsigned char c2 =  26;      // 00011010
```

Оператор І (AND) визначається за допомогою одного амперсанда (&), на відміну від логічних &&, для яких застосовують два амперсанди. Аналогічно за допомогою одного символу | визначається операція АБО (OR) Операції «&» та «|» застосовуються для кожного окремого біта двійкового подання числа. При застосуванні І результатом буде 1, якщо обидва біти дорівнюють 1, та 0, якщо один або обидва біти дорівнюють 0. При застосуванні АБО результат буде 1, якщо хоча б один з бітів дорівнює одиниці:

```
unsigned char c3 = c1 & c2; // 00001000, або 8
unsigned char c4 = c1 | c2; // 10111010, або 186
```

Ексклюзивне АБО («^») забезпечує результат 1, якщо два біти різні, та 0 у протилежному випадку:

```
unsigned char c5 = c1 ^ c2; // 10110010, або 178
```

Операція ПОБІТОВЕ НІ («~») перетворює нулі на одиниці, а одиниці на нулі:

```
unsigned char c6 = ~c1;      // 01010111, або 87
```

Оператор ЗСУВ ВЛІВО переміщує біти вліво. Праворуч додаються нулі. Наприклад:

```
unsigned char c7 = c1 << 1;  // 01010000, або 80
```

Оператор ЗСУВ ВПРАВО переміщує біти вправо. Оператор зсуву вправо переміщує біт праворуч. Ліворуч додаються нулі. Наприклад:

```
unsigned char c8 = c1 >> 2; // 00101010, або 42
```

Можна також використовувати складені присвоєння «<<=» та «>>=».

Порядок застосування унарних операцій та операцій присвоювання – "справа наліво", а всіх інших операцій – "зліва направо".

Усі операції мають пріоритет. Пріоритет додавання і віднімання нижчий, ніж множення і ділення, пріоритет присвоювання нижчий, ніж арифметичних операцій, операції порівняння мають більш високий пріоритет, ніж логічні операції і т.д. Для зміни послідовності операцій слід використовувати дужки.

Operands of bitwise operations should be integers (preferably unsigned integers). For example,

```
unsigned char c1 = 168;      // 10101000
unsigned char c2 =  26;      // 00011010
```

The AND operator is determined by one ampersand (&), as opposed to corresponding Boolean operator, which uses two ampersands. Similarly, single | character represents OR operation. The AND and OR operators are applied to each bit of the binary representation of the number. The result of AND operation is 1 if both bits are equal to 1, and 0 otherwise. The result of OR operation is 1 if at least one of the bits is equal to one:

```
unsigned char c3 = c1 & c2; // 00001000, or 8
unsigned char c4 = c1 | c2; // 10111010, or 186
```

Exclusive-OR ("^") provides the result to 1 if the two bits are different, and 0 otherwise:

```
unsigned char c5 = c1 ^ c2; // 10110010, or 178
```

Bitwise NOT ("~") converts zeros to ones and ones to zeros:

```
unsigned char c6 = ~c1;      // 01010111, or 87
```

The SHIFT LEFT operator moves the bits to the left. This operation discards the far left bits and assigns 0 to the right most bits. For example:

```
unsigned char c7 = c1 << 1;  // 01010000, or 80
```

The SHIFT RIGHT operator moves the bits to the right. This operation discards the far right bits and assigns 0 to the left most bits. For example:

```
unsigned char c8 = c1 >> 2; // 00101010, or 42
```

You can use self-assigned shift “<<=” and “>>=”.

The application of unary operations and assignment is "right to left", and "left to right" for all other operations.

All operations have *precedence* (priority). The precedence of addition and subtraction is lower than multiplication and division, the priority assignment is lower than arithmetic operations, comparison operations have a higher priority than logical operations, etc. To change the sequence, use parentheses

У C++ є можливість перевизначення операцій для типів, які визначає користувач. Зокрема, стандарт C++ визначає об'єкти-потоки, які надають зручні можливості введення та виведення даних. Для цих об'єктів перевизначені операції побітового зсуву як операції занесення в потік «<<» та читання з потоку «>>». Для виведення у стандартний потік (консольне вікно) до об'єкта `cout` застосовується операція «<<». У наступному прикладі на екран виводиться значення змінної `k`:

```
cout << k;
```

Можна також виводити значення констант, зокрема рядків:

```
cout << "ДОБРИДЕНЬ, СВІТ";
```

Для введення даних зі стандартного потоку (з клавіатури) використовують потік `cin` у сполученні з операцією «>>». Наступний приклад показує, як значення змінної `k` прочитати з клавіатури:

```
cin >> k;
```

Виконання цієї операції передбачає переривання роботи програми, яка чекатиме введення даних з клавіатури. Введення повинно завершатись натисненням клавіші `Enter`.

1.2.3. Базові інструкції мови програмування C++

Інструкції (оператори, твердження, речення) – це синтаксичні конструкції, що визначають дії, які виконує програма.

Порожня інструкція складається з однієї крапки з комою.

Інструкція-вираз є повний вираз, який закінчується крапкою з комою. Наприклад:

```
k = i + j + 1;
```

Складена інструкція – це послідовність операторів, укладена у фігурні дужки `{}`. Складену інструкцію часто іменують блоком. Після фігурної дужки, яка закриває блок, крапка з комою не ставиться. Синтаксично блок може розглядатися як окрема інструкція, однак вона також має значення у визначенні видимості і часу життя ідентифікаторів. Ідентифікатор, оголошений усередині блока, має область видимості від точки визначення до фігурної дужки, що закривається. Блоки можуть необмежено вкладатися один в одного.

Одними з ключових інструкцій мови C++ є *інструкції вибору* – умовна інструкція та перемикач.

In C++, it is possible to override operations for user-defined types. In particular, the C++ standard defines stream objects that provide convenient features for input and output data. For these objects, operations of bitwise shift are redefined as an output to stream "<<" and reading from stream ">>". The "<<" operation can be applied to the `cout` object for output to standard output device (console window). The following example shows how to display the value of the variable `k`:

```
cout << k;
```

You can also print the value of constants, including strings:

```
cout << "Hello, World!";
```

To enter data from a standard device (keyboard), you can use the `cin` stream in combination with operation ">>". The following example shows how the value of variable `k` can be read from the keyboard:

```
cin >> k;
```

This activity involves a program interruption that waits for an input from the keyboard. The input should be completed by pressing `Enter`.

1.2.3. Basic statements of the C++ programming language

Statements are syntactic constructs that define actions performed by program.

Empty statement consists of a semicolon.

Expression statement is a full expression, which ends with a semicolon. For example:

```
k = i + j + 1;
```

Compound statement is a sequence of statements enclosed in braces {}. Compound statement is often mentioned as a block. After a brace that closes the block, the semicolon is not used. Syntactically, a block can be considered as a separate statement, but it is also important in determining the visibility and lifetime of identifiers. Identifier defined inside the block has scope from point of definition to closing brace. Blocks can be infinitely nested.

The key statements of C++ are *selection statements*, namely a conditional statement and a switch.

Умовна інструкція застосовується в двох варіантах. На рисунку 1.4 наведено фрагмент діаграми діяльності, який відповідає умовній інструкції першого типу (умовна інструкція повної форми):

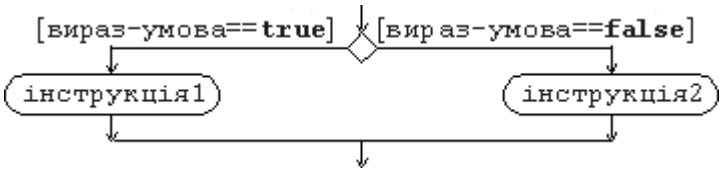

Рисунок 1.4

Формалізація умовної інструкції повної форми виконується за допомогою оператора **if...else**. Нижче наведено приклад визначення умовної інструкції, який відповідає фрагменту діаграми діяльності, що зображена на рисунку 1.6.

```
if (вираз-умова)
    інструкція1
else
    інструкція2
```

При виконанні цієї інструкції обчислюється вираз-умова, і якщо це істина (значення, відмінне від 0), то виконується інструкція1, а інакше – інструкція2. Вираз-умова може бути будь-якого типу, що приводиться до цілого. В умові **if** може бути розміщене визначення змінної з ініціалізацією. Така змінна може застосовуватись тільки в межах умовної інструкції.

На рисунку 1.5 наведено фрагмент діаграми діяльності, що відповідає умовній інструкції другого типу (умовна інструкція скороченої форми).

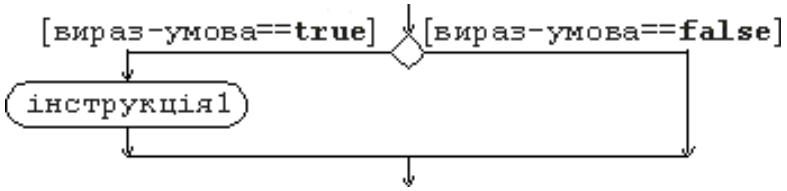

Рисунок 1.5

Розкриття умовної інструкції скороченої форми виконується за допомогою оператора if. Нижче наведено приклад скороченої форми умовної інструкції, який відповідаю фрагменту діаграми діяльності, що зображена на рисунку 1.5.

```
if (вираз-умова)
    інструкція1
```

Conditional statement is used in two ways. Figure 1.4 shows a fragment of the diagram, which corresponds to the first type of conditional statement (the full form of conditional statement):

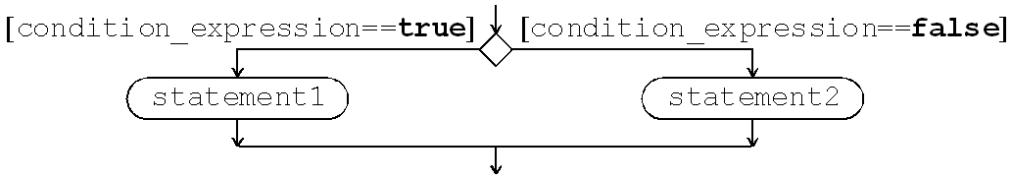

Figure 1.4

The formalization of a conditional statement in complete form is performed by using **`if...else`** construct. Below is an example of conditional statement that corresponds to a fragment of the diagram, shown in Figure 1.6.

```
if (condition_expression)
    statement1
else
    statement2
```

When this statement executed, condition_expression is calculated, and if it is true (value other than 0), then runs statement1, but otherwise runs statement2. A condition_expression can be of any type, converted to an integer. The **`if`** condition can contain definition of variable with initialization. This variable can be used only within a conditional statement.

Figure 1.5 shows a fragment of the diagram, corresponding to the second type of conditional statement (short form).

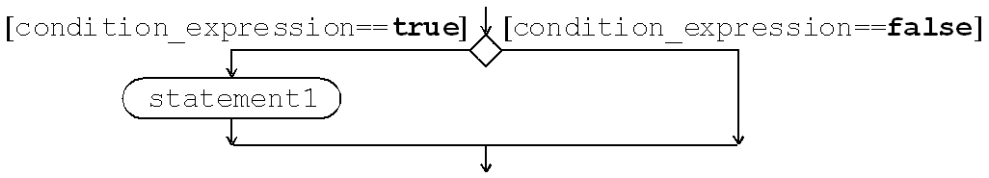

Figure 1.5

Conditional construct in short form represented by the if statement. Below is an example of the reduced form of conditional statement that matches a fragment of the diagram, shown in Figure 1.5.

```
if (condition_expression)
    statement1
```

Перемикач дозволяє вибрати одну з кількох можливих гілок обчислень і будується за схемою, яка зображена на рисунку 1.6 у вигляді фрагмента діаграми діяльності.

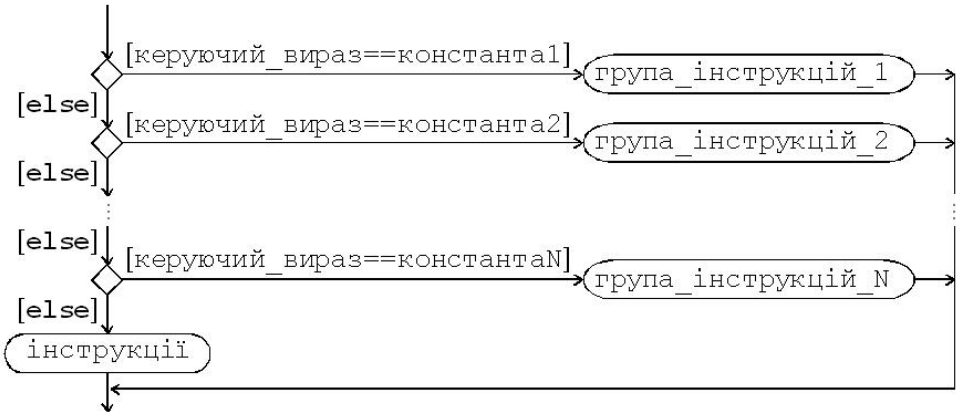

Рисунок 1.6

Програмний аналог перемикача наведено нижче:

```
switch (керуючий_вираз)
    блок
```

Блок має такий вигляд:

```
{
    case константа_1: група_інструкцій_1
    case константа_2: група_інструкцій_2
    ...
    case константа_N: група_інструкцій_N
    default: інструкції
}
```

Виконання перемикача складається з обчислення керуючого_виразу і переходу до групи інструкцій, позначених **case**-міткою, значення якої дорівнює керуючому виразу. Якщо такої мітки немає, виконуються інструкції після мітки **default** (яка може бути відсутня). При виконанні перемикача відбувається перехід на інструкції з обраною міткою, і далі інструкції виконуються у нормальному порядку. Для того щоб не виконувати інструкції, які залишилися у тілі перемикача, необхідно використовувати оператор **break**.

Особливістю перемикача є вимога до *керуючого виразу*, який повинен бути цілим.

Інструкції циклу в мові C++ подані в трьох варіантах: цикл із передумовою, цикл із постумовою і цикл із параметром.

Цикл із передумовою з точки зору діаграми діяльності будується на

The *switch* allows you to select one of several possible branches of calculations and is based on the scheme, which is shown in Figure 1.6 as a fragment of the diagram.

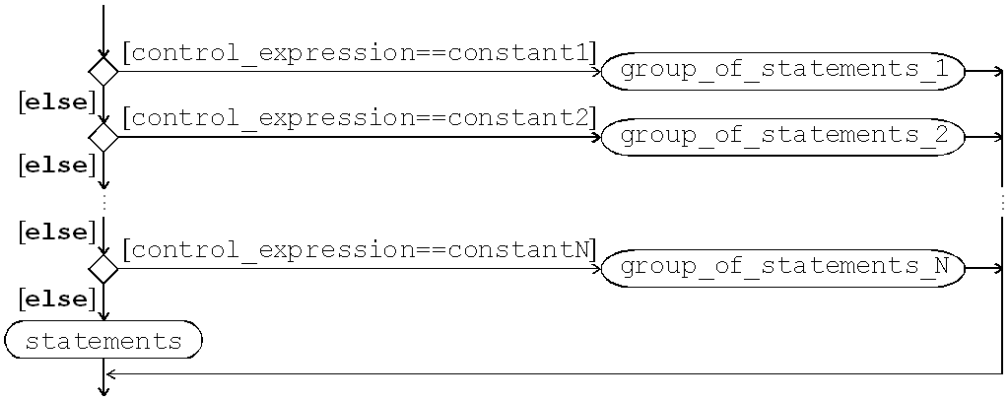

Figure 1.6

Program analogue of switch is as follows:

```
switch (control_expression)
    block
```

The block is as follows:

```
{
    case constant_1: group_of_statements_1
    case constant_2: group_of_statements_2
    ...
    case constant_N: group_of_statements_N
    default: statements
}
```

The implementation of switch consists of calculating control_expression and jumping to the group statements marked with case-label which value is equal to the control_expression. If no such label is present, statements after the **`default`** label are executed (the default label may be absent). If a label with necessary value is present, the flow control is passed to the statements after these label instructions and further instructions are executed in the normal manner. In order not to follow the statements that are left in the body of the switch, the **`break`** operator should be used.

The switch feature is the requirement to control expression, which must be an integer.

Cyclic statements in C++ are available in three versions: a cycle with a precondition, a cycle with postcondition, and a cycle with a parameter.

The cycle with the precondition in terms of activity diagram is based on

основі розгалуження та особливого використання переходу, коли потік керування передається не на наступні стани дії (діяльності), а на попередні.

На рисунку 1.7 наведено фрагмент діаграми діяльності, який ілюструє цикл з передумовою, а нижче подано аналог цього фрагмента у вигляді інструкції циклу while.

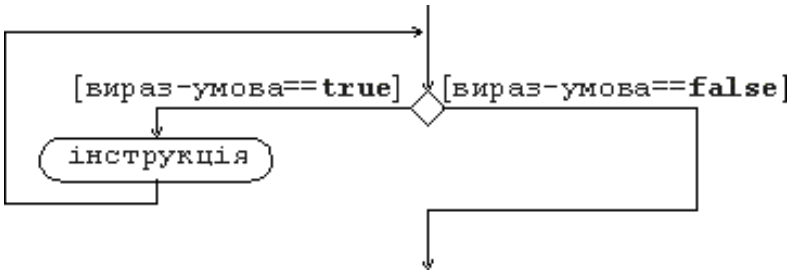

Рисунок 1.7

```
while (вираз-умова)
    інструкція
```

При кожному повторенні циклу обчислюється вираз-умова, і якщо значення цього виразу не дорівнює нулю, виконується інструкція – тіло циклу. В іншому випадку виконання тіла циклу закінчується і потік управління передається на наступні стани дії (діяльності).

Цикл із постумовою будується за схемою, що зображена на рисунку 1.8 Аналогом цього фрагмента є інструкція do...while.

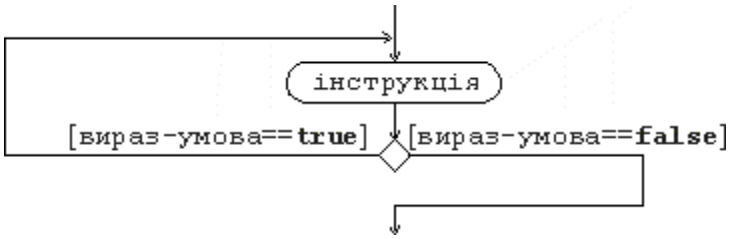

Рисунок 1.8

```
do
    інструкція
while (вираз-умова);
```

Вираз-умова обчислюється і перевіряється після кожного повторення інструкції – тіла циклу, цикл повторюється, поки вираз-умова виконується. Тіло циклу в циклі з постумовою виконується хоча б один раз.

На рисунку 1.9 наведено фрагмент діаграми діяльності, який

branch and special use of jump when the flow control is not transferred to the next action states (activities), but to the previous ones.

Figure 1.7 shows a fragment of diagram that illustrates the cycle with precondition and below analogue of this fragment in the form of while is shown.

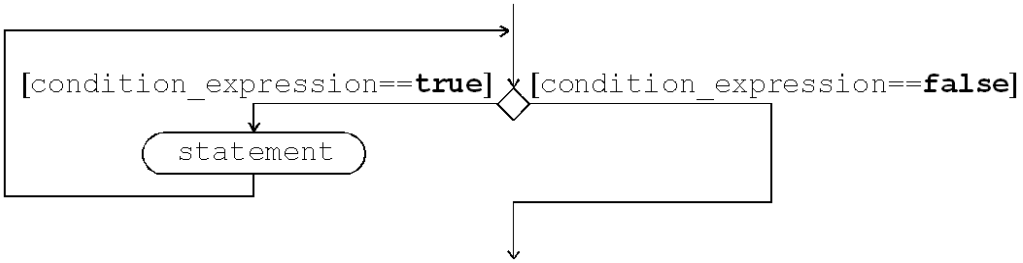

Figure 1.7

```
while (condition)
    statement
```

With each cycle iteration, condition is calculated, and if the value of the expression is not zero, a statement within the body of the loop is executed. Otherwise, execution of the loop body ends and the flow control is passed to the next activities.

The cycle with postcondition is based on the scheme shown in Figure 1.8 The analogous C++ construct is do...while instruction.

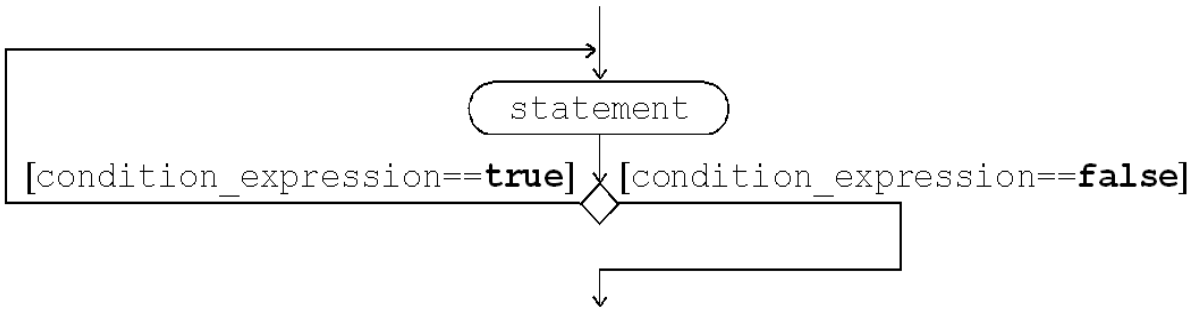

Figure 1.8

```
do
    statement
while (condition);
```

Condition calculated and checked after each iteration of statement that is loop body; cycle is repeated until the condition-expression is executed. The body of the loop in the cycle postcondition is executed at least once.

Figure 1.9 shows a fragment of diagram that illustrates the cycle with a

ілюструє цикл з параметром, а нижче подано аналог цього фрагмента у вигляді інструкції циклу **for**.

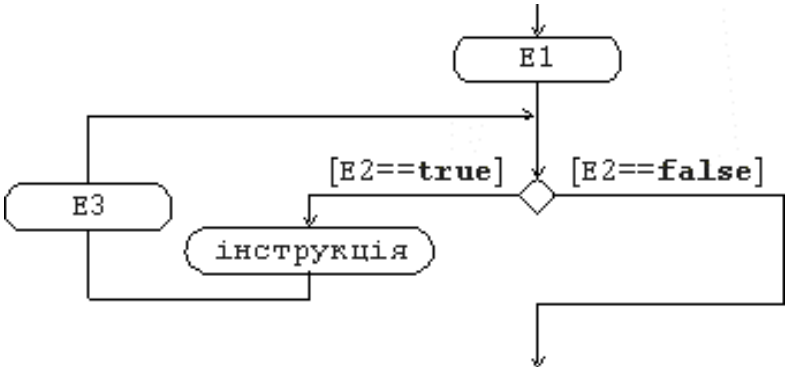

Рисунок 1.9

```
for (E1; E2; E3)
    інструкція
```

де E1, E2 та E3 – вирази скалярного типу. Цикл з параметром реалізується за таким алгоритмом:

– обчислюється вираз E1 (зазвичай цей вираз виконує підготовку до початку циклу);

– обчислюється вираз E2, і якщо він дорівнює нулю, виконується перехід до наступної інструкції програми (вихід з циклу);

– якщо E2 не дорівнює нулю, виконується інструкція – тіло циклу;

– обчислюється вираз E3 – виконується підготовка до повторення циклу, після чого знову обчислюється вираз E2.

Синтаксис циклічної інструкції з параметром за необхідності дозволяє вирази E1, E2 та E3 опускати. В межах виразу E1 можливе оголошення та визначення змінних.

У наступному прикладі сума

$$y = 1^2 + 2^2 + 3^2 + ... + n^2$$

знаходиться за допомогою трьох різних циклічних інструкцій.

За допомогою циклу while:

```
int y = 0;
int i = 1;
while (i <= n)
{
    y += i * i;
    i++;
}
```

parameter and then the analogue of this fragment is represented in the form of **`for`** loop.

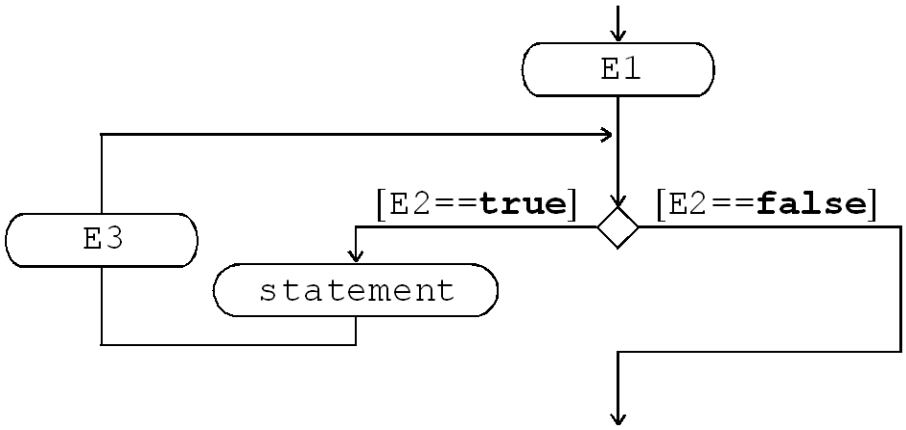

Figure 1.9

```
for (E1; E2; E3)
    statement
```

where E1, E2, and E3 – expressions of the scalar type. Cycle with parameter is implemented according to the following algorithm:

– calculation of expression E1 (usually this expression implements preparation to the cycle);

– calculation of expression E2; if it is zero, flow of control jumps to the next statement of a program (out of the cycle);

– if E2 is not zero, the statement (loop body) is executed;

– expression E3 is calculated, which prepares to repeat the cycle, then expression E2 is evaluated again.

The syntax of cyclic instruction with a parameter allows you to omit the expressions E1, E2, and E3, if necessary. You can declare and define variables within the expression E1.

Assume we need to calculate the following sum:

$$y = 1^2 + 2^2 + 3^2 + ... + n^2$$

This can be done using three different cyclic statements.

Using while loop:

```
int y = 0;
int i = 1;
while (i <= n)
{
    y += i * i;
    i++;
}
```

За допомогою циклу **do...while**:

```
int y = 0;
int i = 1;
do
{
    y += i * i;
    i++;
}
while (i <= n);
```

За допомогою циклу **for**:

```
int y = 0;
for (int i = 1; i <= n; i++)
    y += i * i;
```

У сполученні з інструкціями циклу використовується *інструкція переходу* – break, яка дозволяє перервати виконання самого внутрішнього з циклів, інструкція continue, яка перериває поточну ітерацію самого внутрішнього з циклів while, do...while або for. Найчастіше break використовують у конструкції:

```
if (умова_дострокового_завершення_циклу)
    break;
```

Інструкція **goto** дозволяє перейти на *мітку*. *Мітка* – ідентифікатор із двокрапкою, що стоїть перед інструкцією. Використання інструкції **goto** у більшості випадків недоцільне.

1.2.4. Приклади програм

Приклад 1. Припустимо, що нам потрібно написати програму, яка зчитує значення відстані між двома містами та час поїздки і показує середню швидкість.

```
01 // Average speed
02 #include <iostream>
03 using namespace std;
04 int main(int argc, char* argv[])
05 {
06     float s, t;
07     cout << "Уведіть відстань та час:" << endl;
08     cin >> s >> t;
09     float v = s / t;
```

Using **do...while** loop:

```
int y = 0;
int i = 1;
do
{
    y += i * i;
    i++;
}
while (i <= n);
```

Using **for** loop:

```
int y = 0;
for (int i = 1; i <= n; i++)
    y += i * i;
```

In conjunction with cyclic statements you can use *jump* statements: break, which allows you to interrupt the execution of the innermost loop, continue, which terminates the current iteration of the innermost while, do ... while, or for loop. The following construction of break is used the most often:

```
if (condition_of_early_leaving_cycle)
    break;
```

The **goto** statement allows you to jump to a label. *Label* is an identifier followed by a colon, which is placed before statement. Using of **goto** statement in most cases inappropriate.

1.2.4. Examples of programs

Example 1. Assume that we need to write a program that reads values of a distance between two cities and a time of a trip and shows the average speed.

```
01 // Average speed
02 #include <iostream>
03 using namespace std;
04 int main(int argc, char* argv[])
05 {
06     float s, t;
07     cout << "Enter a distance and a time" << endl;
08     cin >> s >> t;
09     float v = s / t;
```

```
10      cout << "Швидкість дорівнює " << v << endl;
11      return 0;
12 }
```

Перші рядки програми (01 – 05) аналогічні відповідним рядкам попереднього прикладу. Рядок 06 містить визначення двох з плаваючою крапкою (дійсних) змінних. Оператор в рядку 07 виводить повідомлення на екрані. Користувач вводить два значення (рядок 08). Нова змінна v отримує нове розраховане значення (рядок 09). Рядок 10 містить твердження, яке виконує виведення раніше розрахованого значення.

Приклад 2. Наступна програма обчислює цілий степінь.

```
01 // Integer power

02 #include <iostream>
03 using namespace std;

04 int main()
05 {
06      float x;
07      int n;
08      cout << "Уведіть основу степеня та показник:";
09      cin >> x >> n;
10      float power = 1;
11      for (int i = 1; i <= n; i++)
12      power *= x;
13      cout << "Степінь дорівнює " << power << endl;
14      return 0;
15 }
```

На відміну від основи степеня x, показник n визначається як ціле значення (рядок 07).

Значення 1, присвоєне змінній з іменем power, розглядається як перше наближення шуканого результату.

Рядок 11 містить твердження **for**. Цикл **for** обумовлює повторення послідовності тверджень. До початку циклу індекс i отримує значення 1. Це значення поступово збільшується на 1 (i++) до тих пір, поки умова i <= n залишається чинною.

Рядок 12 – тіло циклу. Значення змінної power дорівнює попередньому значенню, помноженому на x n разів.

Приклад 3. Розглянемо задачу пошуку кореня лінійного рівняння $ax + b = 0$. Особливістю цього, на перший погляд нескладного, рівняння є те, що при деяких значеннях a та b його не завжди можна розв’язати.

```
10      cout << "The speed is " << v << endl;
11      return 0;
12 }
```

First lines of the program (01 – 05) similar to the corresponding lines of the previous example. Line 06 contains a definition of two floating point (real) variables. The statement on line 07 performs an output of some message onto the screen. The user inputs two values (line 08). New variable v obtains a new calculated value (line 09). Line 10 contains a statement which performs an output of previously calculated value.

Example 2. The following program calculates an integer power.

```
01 // Integer power

02 #include <iostream>
03 using namespace std;

04 int main()
05 {
06      float x;
07      int n;
08      cout << "Enter base of power and exponent:";
09      cin >> x >> n;
10      float power = 1;
11      for (int i = 1; i <= n; i++)
12          power *= x;
13      cout << "The power is " << power << endl;
14      return 0;
15 }
```

In contrast to a base of power x, the exponent n is defined as an integer variable (line 07).

The value 1 assigned to a floating variable named power is considered as the first approximation of a sought result.

Line 11 contains the for statement. A for loop causes your program to repeat a sequence of statements. Before this, the cycle index i obtains a value 1. This value stepwise increases by 1 (i++) as long as the condition i <= n remains true.

Line 12 constitutes a loop body. The value of power variable becomes equal to previous value multiplied by x n times.

Example 3. Consider the problem of finding the root of a linear equation $ax + b = 0$. This simple at first glance problem has a peculiarity: the equation cannot be resolved for some values of *a* and *b*

.

Таким чином при проектуванні схеми розв'язання цієї задачі необхідно передбачити обробку таких ситуацій:

– якщо *a* не дорівнює нулю, то задача з пошуку значення невідомої змінної x може бути однозначно розв'язана;

– якщо a дорівнює нулю, то може мати місце одна з наступних ситуацій. По-перше, при *b* нерівному нулю задача не може бути розв'язана, бо ділення на нуль неможливе. По-друге, при *b* рівному нулю маємо невизначений результат.

Формалізація умов, наведених вище щодо розв'язання задачі пошуку кореня лінійного рівняння $ax + b = 0$, потребує використання в діаграмі діяльності такого елемента, як розгалуження. На рисунку 1.10 подана діаграма діяльності для розв'язання задачі $ax + b = 0$.

Для написання програми відповідно до діаграми діяльності (рисунок 1.10) на даному етапі вивчення мови програмування C++ спершу необхідно визначити всі ідентифікатори, що присутні в діаграмі.

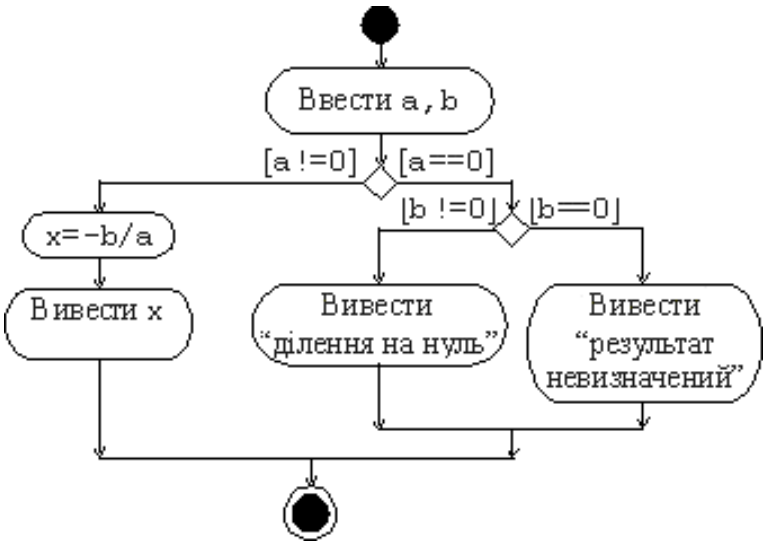

Рисунок 1.10

Кожен ідентифікатор пов'язати з змінною відповідного типу даних. А вже тільки після цього починати ставити у відповідність кожному блоку ту чи іншу інструкцію тощо. Слід зауважити, що в діаграмі визначено розгалуження, причому існує одне вкладене розгалуження. Кожне з розгалужень слід буде розкривати умовними інструкціями, тому що потік керування передається на наступні стани дії.

Програма матиме такий вигляд:

```
// Програма обчислення кореня рівняння ax+b=0
#include <iostream>
using namespace std;
```

Therefore, at the scheme design of solving this problem it is necessary to provide following situations handling:

– if *a* is not zero, the task of finding the value of an unknown variable x can be uniquely solved;

– if *a* is zero, then there may be a one of the following situations. First, when *b* is unequal zero the problem cannot be solved, because division by zero is not possible. Second, when *b* is zero we have undefined result.

The formalization of the conditions listed above to resolve the problem of finding the root of a linear equation $ax + b = 0$, requires the use of such activity diagram element, as branching. Figure 1.10 shows activity diagram of solving $ax + b = 0$.

At this stage of learning the programming language C++, to write a program according to the activity diagram (Figure 1.10), you should first determine all identifiers that are present on the diagram.

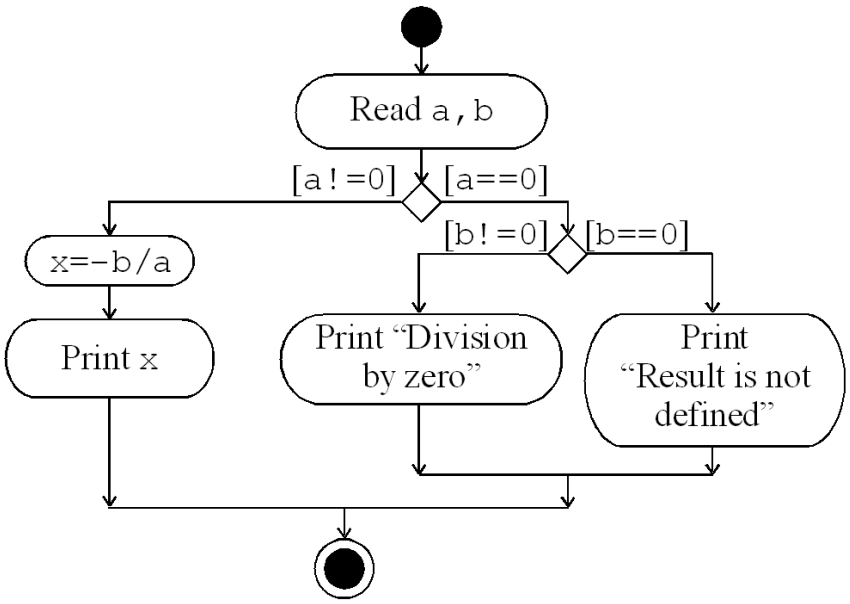

Figure 1.10

Each identifier associated with a variable of the appropriate data type. Then you can begin to put into compliance statements to each block. It should be noted that in the diagram defined branching, wherein there is one nested branch. Each of the conditional branching statements should be disclosed, as the flow of control is transferred to the following action states.

The program will look like this:

```
// The program computing the root of equation ax+b=0
#include <iostream>
using namespace std;
```

```
int main(int argc, char* argv[])
{
    double a, b, x; //визначення змінних
    cin >> a;   // читання значення a
    cin >> b;   // читання значення змінній b
    if (a != 0) // перевірка змінної a
    {
        // корінь рівняння може бути обчислений
        x = -b / a; //обчислення кореня
        cout << x;  //виведення результату на екран
    }
    else
    {
        // корінь рівняння не можна обчислити,
        // бо a дорівнює нулю
        if (b != 0)//перевірка b
        {
            // рівняння не можна обчислити –
            // чисельник дорівнює нулю
            // але знаменник дорівнює нулю
            cout << "ділення на нуль";
        }
        else
        {
            // рівняння не можна обчислити: чисельник
            // та знаменник дорівнює нулю
            cout << "результат невизначений ";
        }
    }
    return 0;
}
```

З метою перевірки коректності розробки діаграми діяльності або програми *обов'язково* необхідно виконати їх тестування на різних вихідних даних (це забезпечить пошук та виправлення логічних помилок). Кількість наборів тестових даних залежить від кількості існуючих траєкторій потоків керування, які поєднують початковий та кінцевий стани.

Для задачі пошуку кореня лінійного рівняння ax+b=0 кількість таких траєкторій дорівнює трьом. Отже, для перевірки як діаграми, так і програми необхідно задати три набори тестових даних, таких, щоб кожен з наборів дозволив протестувати кожну траєкторію:

– a=10.0, b=5.0, результат x=-0.5;
– a=0, b=5, результат "ділення на нуль";
– a=0, b=0, результат "результат невизначений".

```
int main(int argc, char* argv[])
{
    double a, b, x; // definition of variables
    cin >> a; // reading a
    cin >> b; // reading b
    if (a != 0) // test whether a is not zero
    {
        // root of the equation can be calculated
        x = -b/a;  // calculating the root
        cout << x; // output the result on the screen
    }
    else
    {
        // root of the equation cannot be calculated
        // because a is zero
        if (b != 0) // test whether b is not zero
        {
            // equation cannot be solved:
            // the numerator is not zero,
            // but the denominator is zero
            cout << "division by zero";
        }
        else
        {
            // equation cannot be solved: both
            // numerator and denominator are zero
            cout << "result is uncertain";
        }
    }
    return 0;
}
```

In order to verify the correctness of the diagram design or program development one necessarily needs to perform their testing on different input data (this will find and fix logical errors). The number Number of test cases based on the number of existing control flow paths that connect the initial and final states.

For the task of finding the root of a linear equation ax + b = 0 the number of such paths is three. So, to test both diagram and the program, you should specify three sets of such test data that each of the sets allows testing each path:

– a=10.0, b=5.0, result is x=-0.5;
– a=0, b=5, result is "division by zero";
– a=0, b=0, result is “result is uncertain”.

Приклад 4. Розглянемо задачу табуляції функції $y=x^2$ на інтервалі [a,b] з кроком E. Під табуляцією функції розуміють знаходження пар значень (x,y) на заданому інтервалі при визначеному кроці. Як правило кроком є невелика позитивна величина.

Розв'язання в загальному видгляі задачі табуляції функції потребує дещо більшої уваги, ніж розв'язання попередньої задачі. Наведемо основні особливості цієї задачі, які не дозволяють використовувати для її розв'язання діаграму діяльності лінійної структури:

– визначення вихідної інформації щодо значень a, b може бути виконано помилково, а саме, якщо a≥b. Тому в цьому випадку логічно виконати перевірку значень a, b та за необхідності запропонувати визначити a та b повторно;

– виходячи з допущення щодо кроку табуляції (кроком є невелика позитивна величина), припустимо, щоб E не перевищувала одиниці та була би більшою за нуль;

– за умов коректно заданої вихідної інформації щодо a, b, E задача табуляції функції в межах одного кроку (наперед відомого числа кроків) не може бути розв'язана. Останнє пов'язано з тим, що a, b, E можуть бути довільними коректними числами. Тому визначати кожну наступну пару значень (x,y) необхідно до тих пір, поки поточне значення x не стане або дорівнювати b або не стане більше за b.

Отже, формалізація умов, наведених вище щодо розв'язання задачі табуляції функції $y=x^2$, потребує використання в діаграмі діяльності розгалуження з однієї точки зору. З іншої точки зору ці розгалуження необхідно використовувати для виконання дій, що повторюються (виконання повторного визначення a, b, E та обчислення множини пар значень (x,y)). На рисунку 1.11 подана діаграма діяльності для розв'язання задачі табуляції функції $y=x^2$ на інтервалі [a,b] з кроком E.

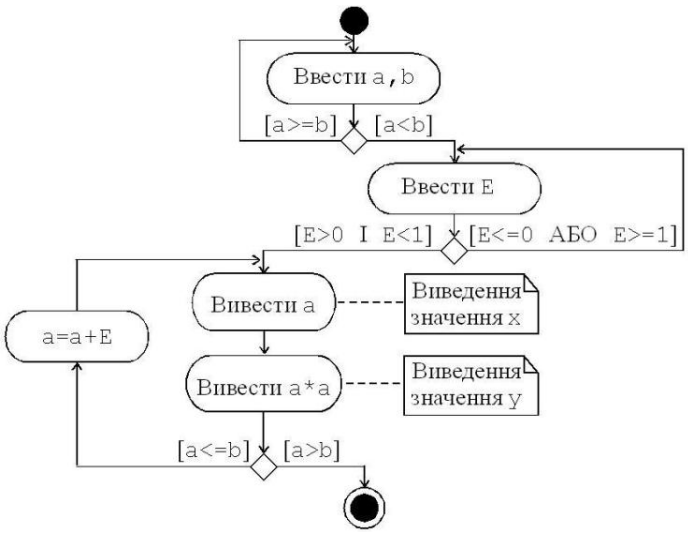

Рисунок 1.11

Example 4. Consider the task of tabulation the function $y=x^2$ on the interval [a, b] with a step E. Function tabulation is finding pairs of values (x, y) on a given interval at a certain step. Typically, step is a small positive value.

Solving function tabulation problem in general requires a little more attention than solving the previous problem. Here are the main features of this problem, which do not allow to solve it using activity diagram with linear structure:

– values of a and b can be set in a wrong way, namely, if $a \geq b$. So in this case you need to check a and b, and if necessary, propose enter a or b again.

– based on assumptions about a step (step is a small positive value), suppose that E does not exceed one and would be greater than zero;

– even if a, b, and E are correctly specified, the problem of functions within a single step (the number of steps known in advance) cannot be solved. This is because a, b, and E can be arbitrary correct numbers. Therefore, finding each successive pair of values (x, y) should be performed as long as the current value of x becomes equal or greater than b.

Thus, the formalization of the requirements listed above for solving problem tabulation of $y=x^2$, requires the use of branching activity diagram from one point of view. From another point of view, these branching should be used to perform actions that are repeated (rereading a, b, E and calculating a set of pairs of values (x,y)). Figure 1.11 shows a diagram that presented for solving tab function $y=x^2$ on the interval [a, b] with step E.

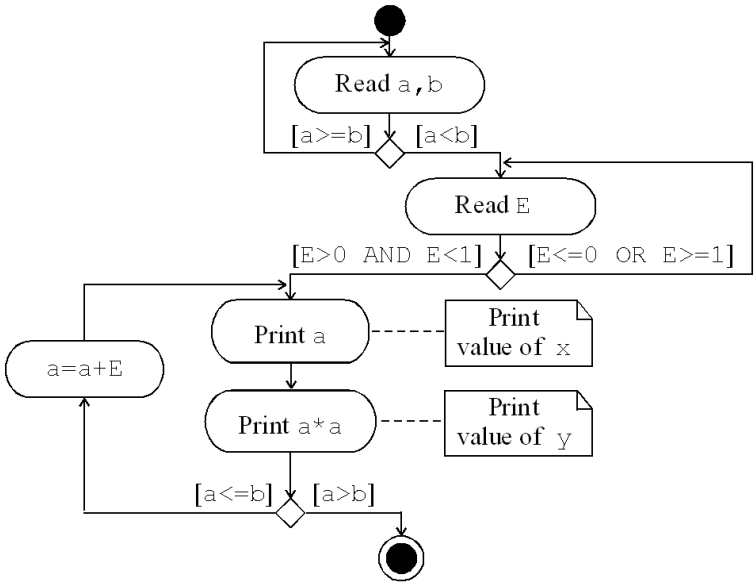

Figure 1.11

Відмінністю задачі табуляції функції є те, що в ній визначено розгалуження, в межах якого потік керування передається не на наступні стани дії, а на попередні. Тому при розкритті цих розгалужень слід використовувати відповідні циклічні інструкції.

Нижче наведена програмна реалізація задачі:

```
// Програма табуляції функції y= x * x

#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
    double a, b, e; //оголошення необхідних змінних
    do //використання циклу з постумовою
    {
        cin >> a; // уведення значення a
        cin >> b; // уведення значення b
    }
    while (a >= b)
    do//використання циклу з постумовою
    {
        cin >> e;// уведення значення e
    } while (e <= 0 || e >= 1)
    //використання цику з параметром
    for (; a <= b; a += e)
    {
        cout << a << " ";  //виведення значення x
        cout << b << endl; //виведення значення y
    }
    return 0;
}
```

Приклад 5. Нехай необхідно вводити дійсні числа і отримувати їх цілу частину доти, поки не буде отримано 0.

Для отримання цілої частини достатньо цілій змінній задати дійсне значення, тому необхідно визначити дві змінні – дійсну та цілу.

```
// Програма отримання цілої частини

#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
    double d;
```

The specific of this problem is that it defines the branching within the flow control and is not transmitted to the following activity state, but to the previous one. Therefore, the implementation of these branches should use the appropriate cyclic statements.

Here is a program implementation of a problem:

```
// Tabulation of y = x * x

#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
    double a, b, e; // Definition of variables
    do // use of a cycle with the postcondition
    {
        cin >> a; // reading value of a
        cin >> b; // reading value of b
    }
    while (a >= b)
    do // use of a cycle with the postcondition
    {
        cin >> e; // reading e
    } while (e <= 0 || e >= 1)
    // use of cyclic instruction with the parameter
    for (; a <= b; a += e)
    {
        cout << a << " ";  // output x
        cout << b << endl; // output y
    }
    return 0;
}
```

Example 5. Suppose you should enter real numbers and get their integer part, until you obtain 0.

For obtaining integer part it is enough to assign the real value to integer variable, so you need to define two variables, real and integer.

```
// Program of getting integer part

#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
    double d;
```

```
    int i;
    do
    {
        cin >> d; // Читання дійсного числа
        i = d;    // Отримання цілої частини
        cout << d << ' ' << i << '\n';
    }
    while (i != 0);
    return 0;
}
```

Оскільки перевірка дійсного числа на рівність нулю в загальному випадку некоректна, ми виконуємо порівняння з нулем цілого числа.

Приклад 6. Наступна програма дозволяє друкувати коди символів, які послідовно вводяться:

```
#include <iostream>

using namespace std;

int main()
{
    char c;
    do
    {
        cin >> c;   // Уведення символу
        int i = c;  // Перетворення символу на ціле
        cout << i << '\n'; // Виведення кодів
    }
    while (c != 'A'); // поки не виконається с == 'A'
    return 0;
}
```

Приклад 7. Наступна програма здійснює введення дійсного значення x і цілого значення n та обчислює y за допомогою інструкції switch відповідно до наведеної таблиці:

N	y
1	x
2	x^2
3	x^3
4	1 / x
інші значення	0

```
    int i;
    do
    {
        cin >> d; // Reading real number
        i = d;    // Getting integer part
        cout << d << ' ' << i << '\n';
    }
    while (i != 0);
    return 0;
}
```

Since checking whether the real number is not equal to zero in the general case is incorrect, we perform a comparison with the integer.

Example 6. The following program allows you to print codes of characters that are entered sequentially:

```
#include <iostream>

using namespace std;

int main()
{
    char c;
    do
    {
        cin >> c;   // Reading character
        int i = c;  // Converting into an integer
        cout << i << '\n'; // Printing codes
    }
    while (c != 'A');    // until c == 'A'
    return 0;
}
```

Example 7. The following program performs input of real value x and the integer value of n and calculates y by using the switch according to the table:

N	y
1	x
2	x^2
3	x^3
4	1 / x
other values	0

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

int main()
{
    double x, y;
    int n;
    cin >> x >> n;
    switch (n)
    {
        case 1 : y = x; break;
        case 2 : y = x * x; break;
        case 3 : y = x * x * x; break;
        case 4 : y = 1 / x; break;
        default: 0;
    }
    cout << y;
    return 0
}
```

Приклад 8. Наступна програма здійснює введення дійсного значення x і цілого значення n та обчислення суми:

$$y = 1/(x - 1) + 1/(x - 2) + ... + 1/(x - n).$$

Програма повинна вивести повідомлення про помилку, якщо знаменник дорівнює нулю.

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

int main()
{
    double x, y = 0;
    int i, n;
    cout << "Input x and n: ";
    cin >> x >> n;
    for (i = 1; i <= n; i++)
    {
        if (x == i)
        {
            cout << "Error!\n";
            break;
        }
    y += 1/(x - i);
    }
    if (i > n)    // Sum is calculated
        cout << "y = " << y << "\n";
    return 0;
}
```

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

int main()
{
    double x, y;
    int n;
    cin >> x >> n;
    switch (n)
    {
        case 1 : y = x; break;
        case 2 : y = x * x; break;
        case 3 : y = x * x * x; break;
        case 4 : y = 1 / x; break;
        default: 0;
    }
    cout << y;
    return 0
}
```

Example 8. The following program carries out input real value x and the integer value of n and the calculation of the sum:

$$y = 1/(x - 1) + 1/(x - 2) + ... + 1/(x - n).$$

The program should display an error message if the denominator is zero.

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

int main()
{
    double x, y = 0;
    int i, n;
    cout << "Input x and n: ";
    cin >> x >> n;
    for (i = 1; i <= n; i++)
    {
        if (x == i)
        {
            cout << "Error!\n";
            break;
        }
        y += 1/(x - i);
    }
    if (i > n)    // Sum is calculated
        cout << "y = " << y << "\n";
    return 0;
}
```

1.2.5. Вправи для самостійної роботи

Вправа 1. Напишіть програму, яка зчитує деяку відстань у дюймах і показує її у міліметрах (1 дюйм = 25,4 мм).

Вправа 2. Напишіть програму, яка читає вісім значень і повертає середнє арифметичне.

Вправа 3. Напишіть програму, яка читає ціле значення змінної `n` і повертає `n!`.

Вправа 4. Напишіть програму, яка зчитує дійсні значення та виводить округлені значення.

Вправа 5. Напишіть програму, яка читає ціле значення n та обчислює y за допомогою інструкції **`switch`** відповідно до наведеної таблиці:

N	Y
0	2
1	4
2	5
3	3
4	1
інші значення	0

Вправа 6. Напишіть програму, яка читає `x`, `k` і `n` та виводить `y`:

$$y = 1/(x + 2) + 2/(x + 4) + ... + (k - 1)/(x + 2(k - 1)) + (k + 1)/(x + 2(k + 1)) + ... + n/(x + 2n)$$

Програма повинна вивести повідомлення про помилку, якщо знаменник дорівнює нулю.

Контрольні запитання

1. Класифікувати існуючі типи даних.
2. Класифікувати існуючі операції як мінімум за двома ознаками.
3. Вирази та операції.
4. Навести всі існуючі варіанти умовної інструкції.
5. Навести всі існуючі варіанти інструкції перемикача.
6. Цикл з передумовою.
7. Цикл з постумовою.
8. Цикл з параметром.

1.2.5. Exercises for self-study

Exercise 1. Write an algorithm and a program that reads some distance in inches and shows it in millimeters (1 inch = 25.4 mm).

Exercise 2. Write an algorithm and a program that reads eight values and returns an average.

Exercise 3. Write an algorithm and a program that reads value of integer variable `n` and returns `n!`.

Exercise 4. Write a program that reads real values and displays rounded values.

Exercise 5. Write a program that reads an integer n and calculates `y` using **`switch`** instruction according to the following table:

n	Y
0	2
1	4
2	5
3	3
4	1
other values	0

Exercise 6. Write a program that reads x, k, and n and displays y:

$$y = 1/(x + 2) + 2/(x + 4) + ... + (k - 1)/(x + 2(k - 1)) + (k + 1)/(x + 2(k + 1)) + ... + n/(x + 2n)$$

The program should display an error message if the denominator is zero.

Advancement Questions

1. Classify existing data types.
2. Classify existing operations at least on two features.
3. Expressions and operations.
4. Give all existing variants of conditional instruction.
5. Give all existing variants of switch instruction.
6. Cycle with precondition.
7. Cycle with postcondition.
8. Cycle with parameter.

Розділ 2. ПРОЦЕДУРНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

У розділі розглядаються особливості побудови програми, за допомогою реалізації окремих підпрограм і подальшої їх інтеграції. Матеріал розділу зосереджений довкола таких конструкцій мови програмування С++, як функції та посилання. Також вводяться в розгляд складені типи даних і принципи роботи з ними.

2.1. Функції та посилання

У ряді випадків розробники програм стикаються з ситуацією, коли необхідно запрограмувати деякі дії, які повинні виконуватися багато разів з різними даними. Для уникнення багатократного дублювання програмного коду використовують функції. Функції дозволяють описати послідовність дій, яку можна виконати багато разів, але з різною вихідною інформацією (параметрами). Для спрощення процесу зміни параметрів усередині функції використовують посилання.

2.1.1. Функції

Наступним етапом розробки програм з використанням мови програмування С++ є застосування процедурно-орієнтованого підходу. Цей підхід припускає, що задача, для якої необхідно розробити програму, може бути подана у вигляді сукупності окремих підзадач. Об'єднання певним чином цих підзадач повинно складатись у вихідну задачу.

Наприклад, розглянемо задачу пошуку такої суми:

$$x + x^2 + x^3 + ... + x^n.$$

Цю задачу можна перетворити таким чином: $\sum_{i=1}^{n} x^i$. Останнє дозволяє в задачі виділити підзадачу. В даному випадку підзадачею є обчислення x^i.

Part 2. PROCEDURE ORIENTED PROGRAMMING

This chapter discusses program design features that include implementation of separate routines and their further integration. The contents of this chapter focus on such structures of the programming language C++ as functions and references. Complex data types and principles of working with them are also introduced.

2.1. Functions and References

In some cases, developers are faced with a situation when they need to program some actions that should be performed many times with different data. To avoid multiple duplication of code, functions are used. Functions allow to describe the sequence of actions that can be performed many times, but with different initial data (parameters). To simplify the changing parameters within the function, so-called references are used.

2.1.1. Functions

The next step of application development using the C++ programming language is the application of the procedure-oriented approach. This approach assumes that the task for which you want to develop an application may be represented as a set of separate subtasks. Appropriate combining of these subtasks should be composed into the original problem.

For example, consider the problem of calculation such a sum:

$$x + x^2 + x^3 + ... + x^n .$$

This problem can be transformed as follows: $\sum_{i=1}^{n} x^i$. This allows us to identify subtask in our task. In this case, a subtask is calculation of x^i.

Визначимо діаграми діяльності, які забезпечують розв'язання цієї задачі. В нашому випадку слід використовувати якнайменш дві діаграми – перша для проектування схеми розв'язання підзадачі x^i, а друга для задачі $\sum_{i=1}^{n} x^i$. На рисунку 2.1 зображена діаграма діяльності для підзадачі x^i, а на рисунку 2.2 – діаграма для обчислення $\sum_{i=1}^{n} x^i$. Характерною рисою другої діаграми є використання в її складі стану діяльності, який формалізує обчислення x^i.

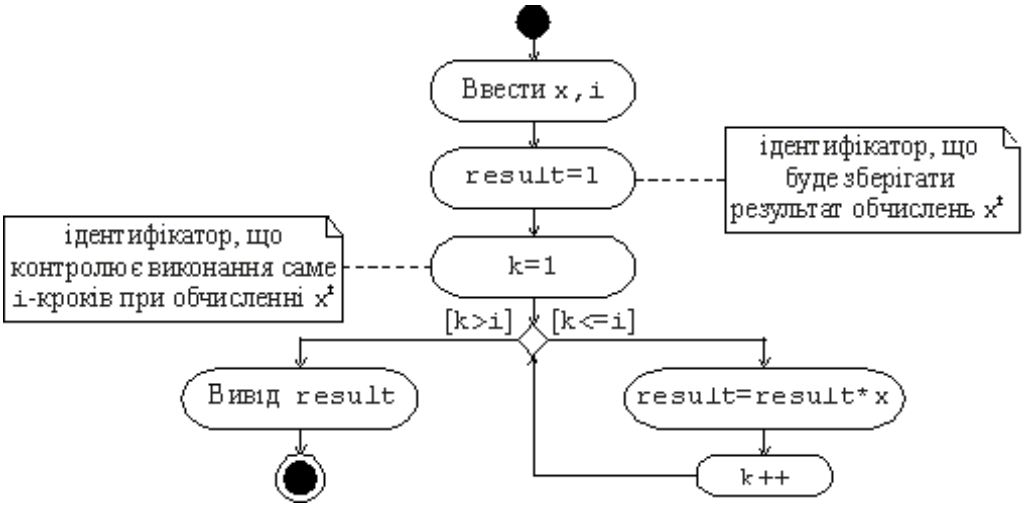

Рисунок 2.1

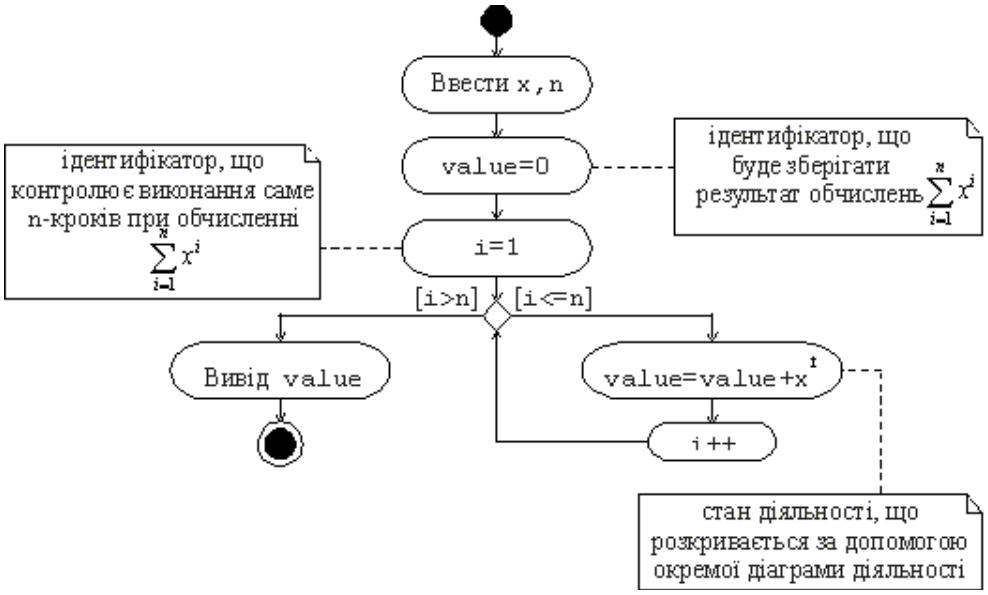

Рисунок 2.2

Аналогічно тому, як задачі можна розділити на підзадачі, програми декомпозуються на окремі підпрограми (функції).

Now we can determine the activity diagrams that provide solution to this problem. In this case, we should use at least two diagrams: the first diagram is used for representation of the subtask x^i, the second one represents task of calculation $\sum_{i=1}^{n} x^i$. Figure 2.1 shows an activity diagram for the subtask x^i, and Figure 2.2 represents a diagram for calculation $\sum_{i=1}^{n} x^i$. A characteristic feature of the second diagram is usage of activity state, which formalizes the calculation of x^i.

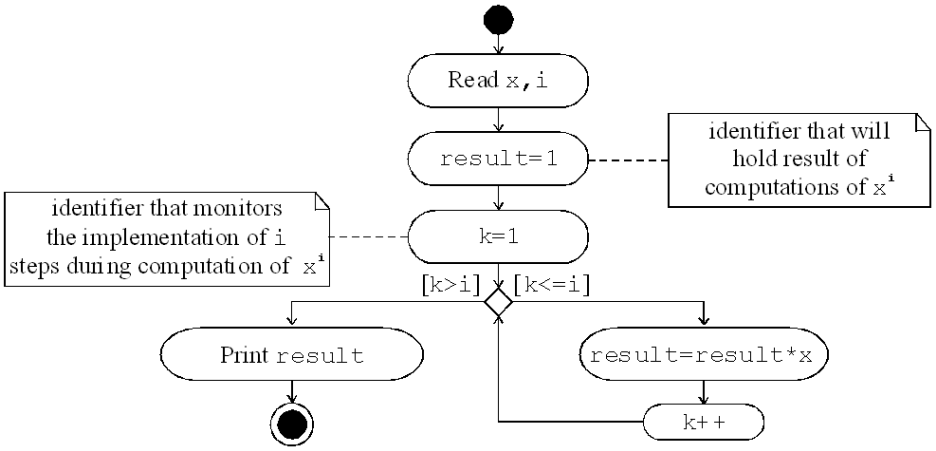

Figure 2.1

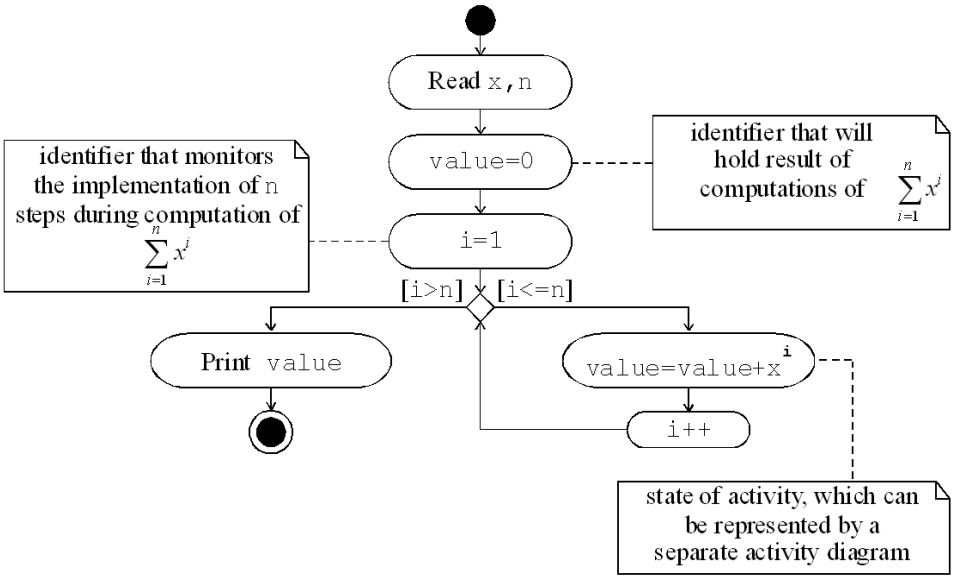

Figure 2.2

So far tasks can be split into subtasks, programs can be split into several subprograms (functions).

Функції є основними складовими програми. Всі інструкції можуть розташовуватися тільки усередині функцій. Поза функціями можуть знаходитись тільки оголошення та визначення, зокрема *оголошення* та *визначення функцій*. *Оголошення функції* без визначення складається із заголовка функції та крапки з комою:

```
тип_результату ім'я_функції(список_формальних_параметрів);
```

Оголошення функції без визначення має також назву *прототипу функції*. Прототип може розміщатися як поза іншими функціями, так і усередині їх.

Параметри (аргументи) функції, що вказуються в списку формальних параметрів при визначенні функції, називаються *формальними*. Такі параметри можна порівняти з вихідною інформацією, яка потрібна для розв'язання задачі. Для кожного параметра задається його тип та ім'я. З ім'ям функції пов'язується ім'я підзадачі, яке може бути довільним. *Тип результату* функції повинен збігатися з типом значення, яке є результатом розв'язання підзадачі.

Параметри, що вказуються при виклику функції, називаються *фактичними*. При виклику функції виділяється пам'ять під її формальні параметри, потім кожному формальному параметру присвоюється значення фактичного параметра.

Визначення функції відрізняється відсутністю крапки з комою після заголовка і наявністю тіла функції. *Тіло функції* являє собою складений оператор (блок).

Наприклад, оголошення (або прототип) функції, що обчислює суму двох цілих чисел, може бути таким:

```
int sum(int a, int b); // прототип
```

Визначення цієї функції може мати вигляд:

```
int sum(int a, int b)
{
    int c = a + b;
    return c;
}
```

Можна взагалі обійтися без змінної c:

```
int sum(int a, int b)
{
    return a + b;
}
```

Functions are the main components of the program. All instructions can be placed only within functions. Only declarations and definitions can be placed outside function bodies. Function declaration without definition consists of function header followed by semicolon:

```
resulting_type function_name(formal_arguments_list);
```

The declaration of a function without definition is also called function *prototype*. Prototypes can be placed both between other functions and within their bodies.

Function parameters (arguments) that are specified in the list of formal parameters in the function definition are called *formal parameters*. These parameters can be paralleled with the initial information required to solve the problem. A definition of a parameter includes setting of its type and name. The function name is associated with name of a subtask; this name can be arbitrary. *The result type* of a function must match the type of value that will be got as a result of the subtask.

The parameters specified in the function call are called *actual parameters*. When you call the function, its formal parameters are allocated in memory, then value of each actual parameter is assigned to the appropriate formal parameter.

A *function definition* does not contain semicolon after the title; instead of semicolon, a function body is present. The *function body* is a compound statement (block).

For instance, declaration (or prototype) of a function, which calculates sum of two integers, can be as follows:

```
int sum(int a, int b); // prototype
```

Definition of this function can be as follows:

```
int sum(int a, int b)
{
    int c = a + b;
    return c;
}
```

You can omit variable **c**:

```
int sum(int a, int b)
{
    return a + b;
}
```

Виклик функції може бути здійснений у виразі в описі або в тілі іншої функції. При виклику функції у вихідному коді вказується її ім'я і список фактичних параметрів (без зазначення їхніх типів):

```
int x = 4;
int y = 5;
int z = sum(x, y);
int t = sum(1, 3);
```

Функція може бути без параметрів:

```
int zero()
{
    return 0;
}
```

При виклику такої функції також необхідно використовувати дужки:

```
cout << zero();
```

Інструкція **return** у тілі функції забезпечує завершення роботи функції та повернення у місце виклику. У випадку функції `main()` керування повертається операційній системі. Значення виразу після **return** стає значенням функції, яке ця функція повертає.

Функція може не повертати ніякого результату. Для позначення цього використовується тип **void**.

```
void print(double a)
{
    cout << a << endl;
}
```

У цьому випадку в тілі функції **return** може бути відсутнім. Якщо інструкція **return** присутня, то після неї не повинно бути ніякого виразу. Таку функцію можна викликати тільки окремою інструкцією.

Функції-підстановки, або вбудовані, мають модифікатор **inline**:

```
inline int min(int a, int b)
{
    return (a < b) ? a : b;
}
```

Якщо функція оголошена як функція-підстановка, компілятор підставляє в точку виклику її тіло. Доцільно з модифікатором **inline** описувати найпростіші функції, багаторазова підстановка яких неістотно вплине на розміри скомпільованої програми.

Function invocation (or function call) can be allocated within an expression in both definitions and other function bodies. Invocation assumes placing of function name and actual arguments list (without type definition) into source code:

```
int x = 4;
int y = 5;
int z = sum(x, y);
int t = sum(1, 3);
```

A function can be defined without parameters:

```
int zero()
{
    return 0;
}
```

Invocation of such function also requires parentheses:

```
cout << zero();
```

The **return** statement within function body terminates the execution of a function and returns control to the calling code. In the case of the `main()` function, **return** statement transfers control back to the operating system. The value of expression after return statement is returned to the calling function.

Sometimes, function does not return any results. To indicate this, type **void** is used.

```
void print(double a)
{
    cout << a << endl;
}
```

In this case, **return** statement can be omitted in function body. If the **return** statement is present, it cannot be followed by any expression. Such a function can be called only in a separate statement.

Inline functions are prefixed by **inline** modifier:

```
inline int min(int a, int b)
{
    return (a < b) ? a : b;
}
```

If the function is declared as an **inline**, the compiler inserts its body at the point of invocation. It is advisable to apply **inline** modifier to simple functions, the frequent inclusion of which does not significantly affect the size of the compiled program.

За умовчанням у C++ *параметри передаються за значенням*, тобто значення фактичних параметрів копіюються в пам'ять, відведену для формальних параметрів. При цьому значення, з якими працює функція, – це її власні локальні копії фактичних параметрів і їхня зміна на ці параметри не впливає. Таким чином, при передачі за значенням вміст фактичних параметрів не змінюється:

```
void f(int k)
{
    k++;        // k = 2;
}

int main()
{
    int k = 1;
    f(k);
    cout << k; // k = 1;
    return 0;
}
```

У C++ можна перевантажувати імена функцій. *Перевантаженням імені* називається його використання для позначення різних операцій над різними типами.

Оголошення функцій можуть зустрічатися в тексті програми кілька разів з однаковими типами, але з різними іменами параметрів. Якщо в двох описах списки формальних параметрів і тип значення, що повертається, цілком збігаються, то вважається, що друге оголошення повторює перше. Якщо списки формальних параметрів збігаються, але типи значень, що повертаються, різні, компілятор повідомляє про помилку. Якщо списки формальних параметрів двох функцій відрізняються числом чи типами її параметрів, то ці дві функції *вважаються перевантаженням* імені однієї функції і повинні мати різні визначення.

Перевантажені функції використовуються в тих випадках, якщо кілька функцій виконують схожі дії над об'єктами різних типів. У цьому випадку зручно дати однакові імена всім цим функціям:

```
// Вибір максимального з двох дійсних чисел:
int max(int, int);

// Вибір максимального з трьох дійсних чисел:
double max(double, double, double);
```

By default, parameters are *passed by value*, i.e. the values of the actual parameter are copied into memory reserved for formal parameters. These values, which are processed in function, are its own local copies of the actual parameters and their change has no effect on actual parameters. Thus, the transfer by value does not change values of the actual parameters:

```
void f(int k)
{
    k++;        // k = 2;
}

int main()
{
    int k = 1;
    f(k);
    cout << k; // k = 1;
    return 0;
}
```

Names of C++ functions can be overloaded. The *function name overloading* means its use to denote different operations on different types.

Function declarations within the single program may occur several times with the same types, but with different parameter names. If the two descriptions of the formal parameter lists and return type are exactly the same, it is considered that the second declaration repeats the first one. If the formal parameter lists are the same, but the return types are different, the compiler reports an error. If the formal parameter lists of two functions differ in the number or types of their parameters, then these two functions are assumed as overloading of function name and they must have different definitions.

Overloaded functions are used in cases where multiple functions perform similar actions on objects of various types. In this case, it is convenient to give the same name to all these functions:

```
// Choosing a maximum of two integers:
int max(int, int);

// Choosing a maximum of three integers:
double max(double, double, double);
```

Для того щоб визначити, яку саме функцію варто викликати, порівнюються кількість і типи фактичних параметрів, зазначені у виклику, з кількістю і типами формальних параметрів усіх описів функцій з даним ім'ям. У результаті викликається та функція, в якої формальні параметри щонайкраще зіставилися з параметрами виклику, чи видається помилка, якщо такої функції не знайшлося.

Ще одна можливість C++ – це використання *параметрів функції за умовчанням*. Такий підхід дозволяє викликати одну функцію з різною кількістю параметрів. Наприклад:

```
double sum(double a, double b = 0, double c = 0)
{
  return a + b + c;
}
```

Цю функцію можна викликати або з одним, або з двома, або з трьома параметрами:

```
double x = 0.1;
double y = 0.2;
double z = 0.3;
cout << sum(x) << sum(x, y) << sum(x, y, z);
```

Параметри за умовчанням задаються в списку останніми. Умовчання визначається тільки у першому оголошенні функції.

У випадку, коли функція викликає сама себе, говорять, що використовується *рекурсія*, а функцію називають *рекурсивною*. Рекурсія може бути *прямою* та *опосередкованою*. При використанні *прямої рекурсії* виклик функції знаходиться в ній самій. *Опосередкована рекурсія* виконується через іншу функцію. Інколи рекурсію можна використовувати замість циклів. У наступному прикладі розраховується сума квадратів натуральних чисел.

```
#include <iostream>
using namespace std;
double sum(int n)
{
    if (n <= 1)
        return 1;
    else
        return n * n + sum(n - 1);
}

int main()
```

In order to determine which function should be invoked, the number and types of the actual parameters specified at the point of invocation are compared to the number and types of the formal parameters of all functions with the given name. As a result, the compiler adds invocation of function, in which the formal parameters are well correlated with the call parameters, or produces an error if such a function was not found.

Another feature of C++ is the use of so-called *default parameters*. This approach allows you to call a function with different number of parameters. For example:

```
double sum(double a, double b = 0, double c = 0)
{
    return a + b + c;
}
```

This function can be called by sending either one, or two, or three parameters:

```
double x = 0.1;
double y = 0.2;
double z = 0.3;
cout << sum(x) << sum(x, y) << sum(x, y, z);
```

Default parameters should be the last in argument list. Default arguments are determined only by the first function declaration.

In the case when the function calls itself, we say that *recursion* is used, and this function is called *recursive function*. Recursion can be direct and indirect. When using *direct recursion*, function calls itself from its body. *Indirect recursion* is performed through another function. Sometimes, recursion can be used instead of the cycles. In the following example, we calculate sum of the squares of integer numbers.

```
#include <iostream>
using namespace std;
double sum(int n)
{
    if (n <= 1)
        return 1;
    else
        return n * n + sum(n - 1);
}

int main()
```

```
{
    cout << sum(5);
    return 0;
}
```

Помилкове використання рекурсії може призвести до переповнення програмного стека.

2.1.2. Області видимості та час життя об'єктів

Поняття *часу життя об'єктів* (змінних, функцій і таке інше) в мові С++ визначає проміжок часу при виконанні програми, коли пам'ять зайнята цим об'єктом. Цей час може бути явно встановлений при визначенні об'єктів.

Досі вважалось, що кожний ідентифікатор у програмі повинен бути унікальним. Але перехід до процедурно-орієнтованого програмування дозволяє використовувати для різних програмних об'єктів однакові імена, але в різних контекстах – *областях видимості*.

Мова програмування С++ підтримує три різновиди областей видимості: файлову (глобальну), локальну та класову.

Файлова видимість – це та частина тексту програми, яка не входить ні в яку функцію, ані у визначення класу. Файлову область видимості формує вся програма, крім тіл функцій, їх заголовків та визначень класів. Це найбільш зовнішня видимість у програмі. Вона містить у собі локальну видимість та класову. Мова С++ гарантує, що пам'ять змінної, яка визначена у файловій області видимості, без явної ініціалізації буде ініціалізована нулем.

У загальному вигляді *локальну видимість* утворює та частина тексту програми, що міститься усередині визначення функцій. Вважається, що кожна функція являє собою окрему локальну область видимості. Усередині функції кожна складена інструкція (блок {}), що містить один або більш операторів опису, теж задає пов'язану з ним локальну область видимості. Локальні області видимості можуть бути вкладеними. Значення неініціалізованої локальної змінної не визначено. Список формальних параметрів функцій входить у локальну область видимості функцій, наприклад:

```
int i;//файлова видимість, і ініціалізована нулем
int f (int a, int b)//локальна область видимості
{
    if (a==10)
    {//вкладена локальна область видимості
```

```
{
    cout << sum(5);
    return 0;
}
```

Misuse of recursion can lead to call stack overflow.

2.1.2. Scope and Lifetime of Objects

The concept of *objects lifetime* (for variables, functions, etc.) in C++ defines the time during program execution when the memory occupied by the object. This time may be explicitly set in the definition of objects.

It was previously thought that each identifier in the program must be unique. However, the transition to procedure-oriented programming allows the usage of the same name for different program objects defined in different contexts, so-called *scopes*.

The C++ programming language supports three types of scope: a file (global) scope, a local scope, and a class scope.

A *file scope* is a part of the program that is not part of functions or class definitions. The file scope forms the entire program, in addition to the bodies of functions their titles and definitions of classes. This is the external scope of the program. It includes a local visibility and a class visibility. The C++ language guarantees that the variable, which is defined in the file scope without an explicit initialization, is initialized to zero.

In general, the *local scope* forms the part of the program text contained within the definitions of functions. Each function defines a separate local scope. Inside the function, each compound statement (block {}), which contains one or more descriptions, also defines the associated local scope. Local scopes can be nested. Values of uninitialized local variables are undefined. The list of formal parameters of functions belongs to the local scope of the function, for example:

```
int i;// file scope, i initialized by zero
int f (int a, int b)// local scope {

if (a==10)
{// nested local scope of compound statement
```

```
            int j=0;
            //...
        }
//...
}

int main()//локальна область видимості функції main()
{
  //...
}
```

Змінна, що визначена у файловій області видимості, доступна в усій програмі. Але є виняток з цього факту: локальна змінна може повторно використовувати ім'я глобальної змінної. В цьому випадку кажуть, що локальна змінна ховає глобальну:

```
int a;

int main()
{
    int a=7266;//локальна змінна a ховає глобальну
}
```

Для доступу до такої глобальної змінної в локальній області видимості можна використовувати *операцію глобальної області видимості*. Ця *операція позначається* :: та є унарною операцією. Якщо така операція стоїть перед глобальним ідентифікатором, то відбувається доступ до глобального об'єкта.

Про *класову область видимості* буде сказано пізніше, при розгляді питань щодо класів.

Функції між собою можуть "спілкуватися" двома способами. Перший спосіб – використання глобальних змінних, а інший – застосування параметрів функцій. Більш прийнятним є другий спосіб, оскільки перший пов'язаний з такими недоліками:

– функція, яка використовує глобальну змінну, залежить від неї. Це ускладнює використання цієї функції в іншій програмі;

– з виклику функції, що використовує глобальні змінні, незрозуміло, чи будуть змінені їх значення; це обтяжує розуміння програми та її модифікацію;

– якщо глобальна змінна отримала неправильне значення, то необхідно буде переглядати всю програму, аби знайти помилку.

Для того, аби зберігати значення локальних змінних між викликами функцій, використовують статичні локальні змінні. *Статичні локальні змінні* – це такі змінні, в опису яких використовується службове слово **`static`**. Для таких змінних виділяється довгочасова (статична) пам'ять.

```
        int j=0;
        //...
    }
    //...
}

int main()// local scope of main() function
{
    //...
}
```

A variable defined within file scope, is available throughout the program. However, there is an exception of this fact: the local variable can reuse the name of a global variable. In this case, we say that the local variable hides the global variable:

```
int a;

int main()
{
    int a=7266; // a local variable hides
                // the global variable a
}
```

You can use the *global scope resolution operation* to access a global variable from the local scope. This unary operation is denoted by two colons (::). If such an operation is preceded by a global identifier, it means an access to the global object.

The *class scope* will be considered later in the context of classes.

Functions can interact with each other in two ways. The first way assumes the use of global variables, and the other way is based on the usage of function parameters. The second method is more appropriate, since the first one is associated with such disadvantages:

– a function that uses a global variable, depends on this variable; it makes difficult to use this function in another program;

– if you call a function that uses global variables, it is unclear whether values of global variables were changed; this fact impairs understanding and program modification;

– if a global variable received an incorrect value, you will need to review the entire program to find the error.

In order to store the values of local variables between function calls, *static local variables* are used. To define static local variable, **`static`** keyword is used. These variables are allocated in long-term (static) memory.

Ініціалізація статичної змінної відбувається тоді, коли керування в перший раз проходить через неї. За умовчанням така змінна ініціалізується нулем:

```
int f()
{
    static int index = 0;
    cout << index << "  ";
}

int main()
{
for (int i = 0; i < 5; i++)
    f();
    //отримаємо на екрані значення – 0 1 2 3 4
    return 0;
}
```

2.1.3. Посилання. Передача параметрів функції за посиланням

Посилання – це альтернативне ім'я об'єкта (змінної). Усі дії, пов’язані з посиланням, застосовуються до реального об’єкта. Посилання може бути оголошено зазначенням його типу, за яким іде оператор &, за яким іде ім'я посилання. Посилання ініціалізують іменем існуючого об'єкта:

```
int i = 10;
int &j = i; // reference to i
j = 11;
cout << i;  // 11
```

Посилання іноді також називаються псевдонімами. Посилання повинні бути ініціалізовані під час створення.

```
int &k; // Помилка!
```

У C++ не існує константних посилань, але можна описати посилання на константні об’єкти:

```
int m = 2;
const int &n = m; // Посилання на i
double x = n + 1; // OK
n = 12;           // Помилка!
```

Static variable initialization occurs when the flow of control passes through it for the first time. By default, this variable is initialized to zero:

```
int f()
{
    static int index = 0;
    cout << index << "  ";
}

int main()
{
for (int i = 0; i < 5; i++)
    f();
    // get the values on the screen: 0 1 2 3 4
    return 0;
}
```

2.1.3. References. Transferring Arguments by Reference

Reference is an alternative name of an object (variable). All activities concerned with the reference are applied to the real object. Reference can be declared by writing the type, followed by & operator, followed by the reference name. References are initialized using the name of a real object:

```
int i = 10;
int &j = i; // reference to i
j = 11;
cout << i;  // 11
```

References are sometimes also called aliases. References must be initialized at the time of creation.

```
int &k; // Error!
```

C++ does not allow you to create constant references, but you can define constant references to objects:

```
int m = 2;
const int &n = m; // reference to i
double x = n + 1; // OK
n = 12;           // Error!
```

Посилання на константний об'єкт можна ініціалізувати константою та об'єктом не її типу.

```
const double x = 3.14;
const int &n = x; // n == 3
```

Не можна створювати масиви посилань, але можна описати посилання на окремі елементи масиву:

```
double a[] = {1, 2, 4, 8};
double& y = a[3];

y = 16; // a[3] = 16
```

Посилання можуть бути використані як аргументи функцій. У цьому випадку можна змінювати значення фактичних параметрів і використовувати ці значення після виклику функції. Оскільки посилання – це інші імена формальних параметрів, усі дії, які виконуються з посиланням, є діями над формальними параметрами. Такий спосіб передачі параметрів функції має назву *передача параметрів за посиланням*.

Наступна функція здійснює обмін значень двох змінних:

```
void swap (int &v1, int &v2)
{
    int temp = v1;
    v1 = v2;
    v2 = tmp;
}

int main()
{
    int a = 10, b = 20;
    swap(a, b);
    //після виклику функції: a == 20, b == 10
    return 0;
}
```

Передача параметрів за значенням більш безпечна, оскільки функція працює з копіями фактичних параметрів. Але іноді це може бути незручно, наприклад, коли як параметр передається великий об’єкт і необхідно заощаджувати час на копіювання та пам’ять, що використовується для цього.

References to the constant object can be initialized by constants and objects of other types.

```
const double x = 3.14;
const int &n = x; // n == 3
```

You cannot create an array of references, but you can declare a reference to any particular element of an array:

```
double a[] = {1, 2, 4, 8};
double& y = a[3];

y = 16; // a[3] = 16
```

References can be used as function arguments. In this case, you can modify values of actual arguments and use these values after invocation of a function. Since references are alternative names of actual arguments, all modifications of reference type parameters within function body are applied to actual arguments. This method of passing parameters to the function is called *passing arguments by reference*.

The following function exchanges values of two variables:

```
void swap (int &v1, int &v2)
{
    int temp = v1;
    v1 = v2;
    v2 = tmp;
}

int main()
{
    int a = 10, b = 20;
    swap(a, b);
    // after invocation: a == 20, b == 10
    return 0;
}
```

Passing parameters by value is more secure because function works with the copies of the actual parameters. However, sometimes it can be inconvenient, when a parameter is a large object and we need to save time by copying it and the memory being used for this.

2.1.4. Приклади програм

Приклад 1. Розглянемо порядок розв'язання задачі щодо визначення $\sum_{i=1}^{n} x^i$ на основі діаграм діяльності, наведених на рисунках 2.1-2.2. Почнемо з обчислення виразу x^i та оформимо його у вигляді функції. Для цього необхідно визначити вихідні дані – вони будуть формальними параметрами функції, результат обчислень – він буде значенням функції, що повертається, та окреслити решту діаграми діяльності, саме вона і буде складати тіло функції. Наостанку потрібно визначитись з ім'ям функції, нехай функція буде мати ім'я `function`.

Наведемо визначення функції `function`.

```
double function(double x, int i)
{
    double result;
    int k;
    for (k=1;k<=I;k++)
        result *= x;
    return result;
}
```

Обчислення виразу $\sum_{i=1}^{n} x^i$ в цілому виконаємо в межах функції main. Зауважимо, що саме оператори, розташовані в тілі функції main(), визначають, як буде працювати програма.

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

int main(int argc, char* argv[])
{
    double x, n, value;
    cin >> x;
    cin >> n;
    // Основний цикл програми:
    for (int i = 1; i <= n; i++)
        value += function(x,i);//виклик функції
  cout << value;
  return 0;
}
```

2.1.4. Examples of programs

Example 1. Consider the order of solving the problem of calculation $\sum_{i=1}^{n} x^i$ based on activity diagrams shown in Figures 2.1-2.2. Let's start with the evaluation of expression x^i and encapsulate it in a function. We need to determine the source data first. They will be the formal parameters of the function. The Result of the calculation will be the return value of the function. The remainder of the activity diagram forms the function body. Finally, we need to determine the function name. Let our function be called `function`.

Here is the definition of the function called `function`.

```
double function(double x, int i)
{
    double result;
    int k;
    for (k = 1;k <= i; k++)
        result *= x;
    return result;
}
```

The calculation of the expression $\sum_{i=1}^{n} x^i$ itself will be implemented within main() function. Note that only statements exclusively, located in the body of the main() function, define how the program will work.

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

int main(int argc, char* argv[])
{
    double x, n, value;
    cin >> x;
    cin >> n;
    // Main loop of the program:
    for (int i = 1; i <= n; i++)
        value += function(x, i);// invocation of a
                                // function
    cout << value;
    return 0;
}
```

Приклад 2. Як наступний приклад розглянемо більш складну задачу обчислення виразу *y* за такою формулою:

$$y = \begin{cases} \sum_{i=1}^{n} (i - x)^2, & x < 0 \\ \prod_{j=1}^{n} (x + i), & x \geq 0 \end{cases},$$

при заданій вихідній інформації щодо *x* та *n*.

Обчислення виразу розмістимо в окремій функції. Програма може мати такий вигляд:

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

double f(double x, int n)
{
    double y;
    if (x < 0)
    {
        y = 0;
        for (int i = 1; i <= n; i++)
            y += (i - x) * (i - x);
    }
    else
    {
        y = 1;
        for (int j = 1; j <= n; j++)
            y *= (x + j);
    }
    return y;
}

int main(int argc, char* argv[])
{
    double x, y;
    int n;
    cout << "Input x and n:";
    cin >> x >> n;
    y = f(x, n); // Виклик функції
    cout << "y = " << y << "\n";
    return 0;
}
```

Example 2. As the next example, we consider more difficult task of calculation expression *y* by the formula:

$$y = \begin{cases} \sum_{i=1}^{n} (i - x)^2, & x < 0 \\ \prod_{j=1}^{n} (x + i), & x \geq 0 \end{cases},$$

for given *x* and *n*.

The calculation of an expression will be allocated in a separate function. The program might look as follows:

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

double f(double x, int n)
{
    double y;
    if (x < 0)
    {
        y = 0;
        for (int i = 1; i <= n; i++)
            y += (i - x) * (i - x);
    }
    else
    {
        y = 1;
        for (int j = 1; j <= n; j++)
            y *= (x + j);
    }
    return y;
}

int main(int argc, char* argv[])
{
    double x, y;
    int n;
    cout << "Input x and n:";
    cin >> x >> n;
    y = f(x, n); // Invocation of a function
    cout << "y = " << y << "\n";
    return 0;
}
```

Приклад 3. Наступна програма обчислює та виводить суму цілих значень, які вводить користувач. Для обчислення суми застосована статична локальна змінна:

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

int add(int i)
{
    static int sum = 0;
    sum += i;
    return sum;
}

int main()
{
    int i;
    do
    {
        cin >> i;
        cout << add(i) << endl;
    }
    while (i);
    return 0;
}
```

Приклад 4. Функції дозволяють розділити програми на окремі незалежні частини. Майже кожна програма може бути розділена на відносно самостійні частини. Програма, яка обчислює певні значення, зазвичай містить частини, відповідальні за введення, розрахунок та виведення. Припустимо, що нам необхідно реалізувати програму, яка обчислює суму других степенів цілих чисел:

$$y = 1^2 + 2^2 + 3^2 + ... + n^2$$

Перша реалізація програми не поділена на частини:

```
#include <iostream>

using namespace std;

int main()
{
    int n;
    cout << "Input n:";
```

Example 3. The following program calculates and displays the sum of integer values entered by user. To calculate this sum, static local variable is used:

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

int add(int i)
{
    static int sum = 0;
    sum += i;
    return sum;
}

int main()
{
    int i;
    do
    {
        cin >> i;
        cout << add(i) << endl;
    }
    while (i);
    return 0;
}
```

Example 4. Functions allow you to split the program into separate independent parts. Almost each program can be divided into relatively independent parts. A program, which calculates a certain value, typically includes parts responsible for input, calculation, and output. Suppose that we need to implement a program that calculates the sum of the second powers of integers:

$$y = 1^2 + 2^2 + 3^2 + ... + n^2$$

The first implementation of the program is not divided into parts:

```
#include <iostream>

using namespace std;

int main()
{
    int n;
    cout << "Input n:";
```

```
    cin >> n;
    int y = 0;

    for (int i = 1; i <= n; i++)
    {
        y += i * i;
    }
    cout << "y = " << y;
    return 0;
}
```

Найпростіше рішення щодо поділу на окремі функції полягає у використанні глобальних змінних:

```
#include <iostream>

using namespace std;

int n;
int y = 0;

void read()
{
    cout << "Input n:";
    cin >> n;
}

void calc()
{
    for (int i = 1; i <= n; i++)
    {
        y += i * i;
    }
}

void write()
{
    cout  << "y = " << y;
}

int main()
{
    read();
    calc();
    write();
    return 0;
}
```

```
    cin >> n;
    int y = 0;

    for (int i = 1; i <= n; i++)
    {
        y += i * i;
    }
    cout << "y = " << y;
    return 0;
}
```

The simplest solution to the problem concerning the division into separate functions is to use global variables:

```
#include <iostream>

using namespace std;

int n;
int y = 0;

void read()
{
    cout << "Input n:";
    cin >> n;
}

void calc()
{
    for (int i = 1; i <= n; i++)
    {
        y += i * i;
    }
}

void write()
{
    cout << "y = " << y;
}

int main()
{
    read();
    calc();
    write();
    return 0;
}
```

Використання глобальних змінних не є хорошою ідеєю. Кращий підхід полягає у використанні аргументів:

```
#include <iostream>

using namespace std;

int read()
{
    int n;
    cout << "Input n:";
    cin >> n;
    return n;
}

int calc(int n)
{
    int y = 0;
    for (int i = 1; i <= n; i++)
    {
        y += i * i;
    }
    return y;
}

void write(int y)
{
    cout << "y = " << y;
}

int main()
{
    int n = read();
    int y = calc(n);
    write(y);
    return 0;
}
```

Функція main() може бути реалізована без будь-яких змінних.

```
int main()

    write(calc(read()));
    return 0;
}
```

The use of global variables is not a good idea. The best approach is to use arguments:

```
#include <iostream>

using namespace std;

int read()
{
    int n;
    cout << "Input n:";
    cin >> n;
    return n;
}

int calc(int n)
{
    int y = 0;
    for (int i = 1; i <= n; i++)
    {
        y += i * i;
    }
    return y;
}

void write(int y)
{
    cout << "y = " << y;
}

int main()
{
    int n = read();
    int y = calc(n);
    write(y);
    return 0;
}
```

The main() method can be implemented without any variables.

```
int main()
{
    write(calc(read()));
    return 0;
}
```

Приклад 5. Функція `solve()` в наступному прикладі знаходить корінь лінійного рівняння і повертає **`true`**, якщо цей корінь є. В іншому випадку вона повертає **`false`**.

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

bool solve(double a, double b, double& x)
{
    if (a == 0)
    {
        return false;
    }
    x = -b / a;
    return true;
}

int main()
{
    double a, b, x;
    cin >> a >> b;
    if (solve(a, b, x))
    {
        cout << x;
    }
    else
    {
        cout << "Error";
    }
    return 0;
}
```

2.1.5. Вправи для самостійної роботи

Вправа 1. Напишіть програму, яка реалізує та тестує функцію `signum()`.

Вправа 2. Напишіть програму, яка обчислює та виводить мінімальне та максимальне значення цілих чисел, які вводить користувач. Застосуйте статичні локальні змінні.

Вправа 3. Напишіть програму, яка вводить x та обчислює y з використанням рекурсивної функції:

$$y = (x + 1)(x + 2)(x + 3)(x + 4) \ldots (x + n)$$

Вправа 4. Створіть функцію, яка повертає 1, значення аргументу, або добуток аргументів, залежно від кількості аргументів. Здійсніть тестування у функції main(). Реалізуйте програму двома способами: за допомогою перевантаження імен функцій і використання аргументів за умовчанням.

Example 5. The `solve()` function in the following example finds the root of a linear equation and returns **`true`**, if the root can be found. Otherwise, it returns **`false`**.

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

bool solve(double a, double b, double& x)
{
    if (a == 0)
        return false;
    x = -b / a;
    return true;
}

int main()
{
    double a, b, x;
    cin >> a >> b;
    if (solve(a, b, x))
    {
        cout << x;
    }
    else
    {
        cout << "Error";
    }
    return 0;
}
```

2.1.5. Exercises for self-study

Exercise 1. Create a program that provides and tests signum() function.

Exercise 2. Write a program that calculates and shows the minimum and maximum of integers as the user inputs those integers. Use static local variables.

Exercise 3. Write a program that reads *x* and *n* and calculates *y* using recursive function:

$$y = (x + 1)(x + 2)(x + 3(x + 4) \ldots (x + n)$$

Exercise 4. Create a function that returns 1, argument, and product of arguments, depending on arguments count. Test this function in main() function. Implement program in two ways: using function overloading and using default arguments

Вправа 5. Створіть функцію для розв'язання квадратного рівняння. Функція повинна повертати **`false`**, якщо дискримінант менше нуля, і **`true`** в інших випадках. Функція повинна отримувати коефіцієнти як аргументи-значення і повертати корені як аргументи-посилання.

Контрольні запитання

1. Оголошення та визначення функції.
2. Формальні та фактичні параметри.
3. Функції-підстановки.
4. Перевантаження імен функцій.
5. Параметри за умовчанням.
6. Рекурсивні функціі, їх різновиди.
7. Види областей видимості та час життя об'єктів.
8. Недоліки та переваги використання глобальних та локальних змінних.
9. Посилання.
10. Параметри, що передаються за значенням.
11. Параметри, що передаються за посиланням.
12. Способи передачі параметрів функції та їх особливості.

2.2. Складені типи даних

В деяких випадках доводиться стикатися з ситуацією, коли потрібно використовувати відносно багато змінних одного типа. В цьому випадку будуть потрібні, як значні витрати на опис таких змінних, так і виникнуть істотні труднощі з їх обробкою. Вирішення вказаної проблеми забезпечується за рахунок використання таких понять, як масив, покажчик і динамічний розподіл пам'яті.

2.2.1. Масиви

Масиви належать до так званих складених чи складених типів даних. *Масив* – це колекція комірок пам'яті, кожна з яких дозволяє зберігати дані одного типу. Комірки пам'яті мають назву *елементів*. Масив може бути створений шляхом визначення типу елементів, імені та кількості елементів. Для визначення кількості елементів масиву після ідентифікатора в квадратних дужках задається константа (зокрема, іменована константа або константний вираз):

.

Exercise 5. Create a function for solving of quadratic equation. A function should return **`false`** if the discriminant is less than zero and **`true`** otherwise. The function must obtain coefficients as arguments and return roots as reference type arguments.

Advancement Questions

1. Declarations and definition of a function.
2. Formal and actual parameters.
3. Inline functions.
4. Function name overloading.
5. Default arguments.
6. Recursive functions and their variations.
7. Types of scope and lifetime of objects.
8. Disadvantages and advantages of global and local variables.
9. References.
10. Passing parameters by value.
11. Passing parameters by reference.
12. Ways of function parameters transfer and their features.

2.2. Compound Data Types

In some cases, you need to use a relatively large amount of variables of the same type. In this case, additional efforts for description of such variables are required, and considerable difficulties arise from their processing. The solution to this problem is provided by the use of concepts of arrays, pointers, and dynamic memory allocation.

2.2.1. Arrays

Arrays belong to the so-called complex (or compound) data types. An *array* is a collection of data storage locations, each of which holds the same type of data. Each storage location is called an element of the array. An array can be created by identifying the type of item, name and number of items. Number of items can be set using a constant (symbolic constant or constant expression)

Наприклад:

```
int a[5];       // Масив з п'яти елементів типу int
const int n = 6;
double a1[n];  // Масив з п'яти елементів типу double
```

Таким чином, маємо два одновимірних масиви (a та a1).

Кількість елементів масиву не можна визначати за допомогою змінної чи неконстантного виразу. Масив може бути ініціалізований списком значень у фігурних дужках. Наприклад:

```
int b[4] = {1, 2, 3, 4};
```

При наявності списку ініціалізації, що охоплює всі елементи масиву, можна не вказувати кількість елементів масиву, вона буде визначена компілятором:

```
int c[] = {1, 2, 3}; // Масив із трьох елементів
```

Якщо число елементів масиву більше числа значень у списку, то елементи масиву, що явно не ініціалізовані, будуть установлені в 0. Якщо число елементів масиву менше числа значень у списку, будемо мати помилку компіляції.

Один масив не може бути ініціалізований іншим масивом і не може бути присвоєний іншому масиву. Щоб скопіювати один масив в інший, необхідно скопіювати кожен елемент по черзі в циклі.

Доступ до кожного елемента здійснюється за його номером у масиві (індексом), що вказується після імені масиву в квадратних дужках. Як індекс масиву може використовуватися будь-який вираз, який приводиться до цілого значення. Індекси елементів масиву завжди починаються з 0, індекс останнього елемента на одиницю менше кількості елементів у масиві. При використанні стандартних масивів неможливий будь-який автоматичний контроль виходу за межі масиву.

Масиви з розмірністю 2 і більше розглядаються як масиви масивів, і для кожного виміру вказується число елементів:

```
double a[2][2] = {{1, 2}, {3, 4}};  // Матриця 2 * 2
```

Перша розмірність – це розмірність по рядках, а друга – по стовпцях. При ініціалізації списку значень друга і наступні розмірності повинні вказуватися явно.

Для індексування в багатовимірному масиві потрібна пара дужок для кожного виміру:

```
int i = 0, j = 1;
```

For example

```
int a[5];        // Array of five items of double type
const int n = 6;
double a1[n];    // Array of six items of double type
```

Now we have two one-dimensional arrays (a and a1).

Array size cannot be set using variable or non-constant expression. Array can be initialized by a list of comma-separated values enclosed in braces. For example:

```
int b[4] = {1, 2, 3, 4};
```

If there is an initialization list, which covers all items of the array, there is no need to specify the number of items in the array, because it will be determined by the compiler:

```
int c[] = {1, 2, 3}; // An array of three items
```

If the number of items in the array is more than the number of values in the list, all non-initialized array items are set to 0. If the number of items in the array is less than the number of values in the list, there will be a compilation error.

One array cannot be initialized by another array and cannot be assigned to another array. To copy one array to another one, you must copy each element one by one in a loop.

Each element is *accessed* by its number in the array (index), which is indicated after the array name in square brackets. The index of an item can be any expression, which evaluates to an integer value. Indices of the array items always starts at 0, the index of the last element is one less than the number of items in the array. When using standard arrays impossible any automatic control of going beyond the array.

Arrays of dimension 2 or more are treated as arrays of arrays, and each dimension indicates the number of items:

```
double a[2][2] = {{1, 2}, {3, 4}};  // Matrix 2 * 2
```

The first dimension concerned with rows, and the second one concerned with columns. If you initialize an array with a list of values, the second and subsequent dimensions must be specified explicitly.

To index items of a multidimensional array, a pair of brackets for each dimension is needed:

```
int i = 0, j = 1;
```

```
a[i][j] = i + j; // a[0][1]==1
```

2.2.2. Вказівники

Вказівник – це об'єкт, що містить адресу іншого об'єкта і дозволяє маніпулювати цим об'єктом. Кожен указівник асоціюється з певним типом даних, що визначає тип указівника.

Для визначення вказівника необхідно перед його ідентифікатором (після імені типу) розмістити знак *:

```
int *pi; // Вказівник на об'єкт цілого типу
```

Унарна операція отримання адреси позначається знаком &. За допомогою цієї операції можна отримати адресу об'єкта, до якого вона застосовується. Операція & може бути застосована тільки до об'єктів, розташованих у пам'яті (lvalue). Такими об'єктами є змінні та елементи масивів. Операцію отримання адреси не можна застосовувати до виразів або до констант.

Операція * (*розіменування*), застосована до вказівника, дозволяє одержати об'єкт, на який посилається вказівник.

За допомогою операції отримання адреси вказівник при оголошенні може бути проініціалізований об'єктом відповідного типу:

```
int i = 0;
int *pi = &i; // Вказівник на i
```

Існує спеціальний тип указівника **void***. Указівнику такого типу можна присвоїти адресу об'єкта будь-якого типу без явного приведення. Варто уникати використання вказівника **void***, бо це може зумовити помилки, які важко ідентифікувати.

Вказівнику можна присвоювати ціле значення 0 (трактується як "указівник ні на що не вказує") і порівнювати з нулем.

Можна визначити *вказівник, що зберігає адресу константи* (вказівник на константу). Для цього потрібно до усього визначення додати префікс **const**:

```
const double *pd;
```

Константним є об'єкт, а не вказівник, що на нього вказує. Вказівнику на константу можна присвоїти адресу звичайної змінної. Але таку змінну не можна змінити через цей вказівник.

Існує *константний вказівник*, що може вказувати на будь-які об'єкти. Значення цих об'єктів можна змінювати, але сам указівник не можна змінити після визначення.

```
a[i][j] = i + j; // a[0][1]==1
```

2.2.2. Pointers

A *pointer* is a variable that holds an address of another object and allows you to manipulate the object. Each pointer is associated with a particular type of data that identifies the type of the pointer.

To determine the pointer, it is necessary to place a sign * before the identifier (named after the type):

```
int *pi; // Pointer to an object of integer type
```

Unary *operation of obtaining the address* is designated by the character &. With this operation, you can get the address of the object to which it is applied. Operation & can be applied only to objects located in memory (lvalue). These objects can be variables and array items. The operation of obtaining address cannot be applied to expressions or constants.

The * (*dereferencing*) operation applied to the pointer allows you to get the object referenced by the pointer.

Using operation of getting the address, the pointer by its definition can be initialized by object of the appropriate type:

```
int i = 0;
int *pi = &i; // Pointer to i
```

It is a special type of pointers: **void***. You can assign pointers of any types to **void*** without explicit cast. One should avoid the use of pointer **void***, because it can lead to errors that are difficult to identify.

The integer zero can value be assigned to a pointer (interpreted as a "pointer to nothing"); pointers can be compared with zero.

You can define a pointer that stores the address of a constant (a pointer to a constant). To do this, you need to add **const** prefix before definition:

```
const double *pd;
```

Constant is an object, but not a pointer that points to it. An address of some ordinary variable can be assigned to the constant pointer. However, this variable cannot be changed through this pointer.

There is a constant pointer that can point to any object. The values of this object can be changed, but the pointer itself cannot be changed after its definition.

Для опису константного вказівника потрібно поставити **const** не перед усім визначенням, а перед ім'ям указівника праворуч від *. Такий вказівник потрібно ініціалізувати при визначенні:

```
int i = 1;
int * const pi = &i;
```

Вказівнику не може бути присвоєно значення, яке не є адресою. Не можна присвоїти вказівнику одного типу значення, що є адресою об'єкта іншого типу.

Вказівники можуть бути використані в арифметичних виразах. Для вказівників визначені операції порівняння, додавання до вказівника цілого числа, віднімання двох указівників. Додавання до вказівника 1 збільшує значення, що міститься в ньому, на розмір області пам'яті, що відводиться об'єкту відповідного типу. Такі дії над елементами масиву одержали назву *адресної арифметики*.

Ім'я масиву в C++ є константним указівником, що містить адресу першого елемента масиву (з індексом 0). Його можна використовувати для ініціалізації іншого вказівника (на об'єкт типу елемента масиву):

```
int a[10];
int *pa = a;
```

До вказівника *pa* також застосовна операція звертання за індексом:

```
pa[0] = 0;      // Те ж саме, що a[0] = 0;
cout << pa[0]; // Те ж саме, що cout << a[0];
```

За допомогою виразу pa+i (або a+i) можна одержати адресу i-го елемента масиву.

Початковий елемент масиву можна просто записати як *a:

```
a[0] == *a;
```

Щоб одержати адресу наступного елемента, можна використовувати операцію інкремента:

```
pa++; // але не a++, a - константний вказівник
```

Аналогічно можуть використовуватися операції обчислення цілого значення і декремента. Якщо вказівники вказують на елементи одного масиву, то вони можуть порівнюватися (по <, >, >= ,<= == і !=). Для двох указівників на елементи одного масиву допускається операція обчислення. Її результат – цілий (кількість елементів, починаючи з елемента з меншим індексом, до елемента з більшим індексом).

To define the constant pointer, you need to put **const** keyword between the * sign and pointer name. Such pointers must be initialized by their definition:

```
int i = 1;
int * const pi = &i;
```

The value that is not an address cannot be assigned to a pointer. The address of an object of a type cannot be assigned to a pointer to another type.

Pointers can be used in arithmetic expressions. The following operations are provided for pointers: comparison operation, adding an integer value to a pointer, subtracting two pointers. Adding 1 to the pointer increases the value contained in it, by the size of the memory that is given to the object type. Such actions on the items of an array are called *address arithmetic*.

The name of an array in C++ is a constant pointer that contains the address of the first element of an array (namely, element with index 0). It can be used to initialize another pointer to an object of array element type:

```
int a[10];
int *pa = a;
```

The subscript operation can be also applied to *pa*:

```
pa[0] = 0;      // The same as a[0] = 0;
cout << pa[0]; // The same as cout << a[0];
```

You can get an address of i-th element of an array using *pa*+i (or *a*+i).

Initial array element can be simply written as *a:

```
a[0] == *a;
```

To get the address of the next element, you can use the increment operation:

```
pa++; // but not a++, a is the constant pointer
```

The decrement operation is used in a similar way. If two pointers point to items of the same array, they can be compared (by <, >, >=, <=, == and !=). The subtraction operation is allowed for two pointers to items of the same array. Its result is a number of items, starting with an element with a lower index to an element with a higher index.

2.2.3. Динамічний розподіл пам'яті

Динамічним розподілом пам'яті називається явне виділення і звільнення пам'яті під час виконання програми. Вільна пам'ять виділяється для збереження об'єкта за допомогою операції **new**. Після слова new указується тип об'єкта, який повинний бути розміщений динамічно. Операція new повертає вказівник на динамічно створений об'єкт:

```
int *p = new int;
```

Якщо після спеціалізації типу вказати в квадратних дужках значення вимірності, це приведе до динамічного розміщення одновимірного масиву. Вимірність може бути виразом будь-якої складності (не тільки константою). Операція **new** повертає вказівник на перший елемент масиву:

```
int i = 100;
double *a = new double[i*2];
```

Після того як динамічно розміщений об'єкт виявився непотрібним, пам'ять варто звільнити за допомогою операції **delete**:

```
int *p = new int; // виділення пам'яті
delete p;         // звільнення пам'яті
```

Якщо динамічно був розміщений масив, для звільнення пам'яті необхідно між **delete** і вказівником уставити пари порожніх квадратних дужок:

```
int *a = new int[100];
delete [] a;
```

Необхідно стежити, щоб уся пам'ять, виділена за допомогою операції **new**, була звільнена за допомогою операції delete. До вказівника на константний об'єкт не можна застосовувати операцію delete.

По ходу виконання програми вільна пам’ять може вичерпатись. У цьому випадку логічно може виникнути питання – що відбудеться, якщо для опреації **new** не вистачить пам’яті. Скоріш за все такий виклик, як

```
double *a = new double [99999999999999999999999];
```

навряд чи виконається вдало. За умовчанням операція **new** повертає 0, якщо недостатньо вільної пам’яті. В нашому випадку скоріш за все буде повернено 0. Таким чином, при роботі з динамічною пам’яттю необхідно враховувати цей факт:

```
double *a = new double [99999999999999999999999];
if (a)
```

2.2.3. Dynamic Memory Allocation

Dynamic memory allocation is an explicit allocation and release of memory during program execution. To allocate an object in a free store (heap), you should use a **new** operation, followed by object type. The new operation returns a pointer to a dynamically created object:

```
int *p = new int;
```

If you specify a value of the size in square brackets after the type specialization, a one-dimensional array will be dynamically allocated. The value of size can be an expression of any complexity (not just a constant). The **new** operator returns a pointer to the first element of the array:

```
int i = 100;
double *a = new double[i*2];
```

Once the dynamically allocated object becomes unnecessary, memory should be released by the **delete** operation:

```
int *p = new int; // memory allocation
delete p;         // memory release
```

If a dynamic array was allocated, to release the memory you should place pair of square brackets between **delete** operator and pointer name:

```
int *a = new int[100];
delete [] a;
```

You should be sure that all the memory allocated using the new operation, was released by the **delete** operation. The **delete** operation cannot be applied to a pointer to a constant object.

At runtime, program may exhaust free memory. In this case, there may arise a logical question: what happens if a **new** operation does not have enough memory. Such an expression

```
double *a = new double [999999999999999999999999];
```

is unlikely to be finished successfully. By default, the new operation returns 0 if there is not enough free memory. In our case, 0 is more likely to be returned. Thus, when dealing with dynamic memory, you must take into account this fact:

```
double *a = new double [999999999999999999999999];
if (a)
```

```
{
    //пам'яті вистачило, виконуємо роботу з a
}
else
{
    //помилка, не вистачило пам'яті
}
```

2.2.4. Передача параметрів функції за вказівником

Розглянемо ще один підхід до передачі параметрів функції. Використання параметрів-указівників було єдиним способом зміни значень фактичних параметрів у мові програмування С.

При передачі параметрів функції за вказівником сам указівник передається за значенням, але об'єкт, на який він указує, можна змінювати, і ці зміни будуть доступні. При цьому у функцію необхідно передавати адресу того об'єкта, який необхідно змінити, а всередині функції використовувати операцію розіменування для параметра-вказівника. Наведено приклад функції, яка повинна змінювати значення двох змінних:

```
void swap (int *v1, int *v2)
{
    int temp = *v1;
    *v1 = *v2;
    *v2 = tmp;
}

int main()
{
    int a = 10, b = 20;
    swap(&a, &b);
    //після виклику функції a=20, b=10
}
```

2.2.5. Масиви як параметри функцій

Масиви аналогічно до змінних можна передавати як параметри функцій. Для визначення одновимірного масиву як формального параметра функції можна використовувати один із способів, наведених нижче:

```
//формальний параметр –масив з 10 елементів
```

```
{
    // we have enough memory, can work with a
}
else
{
    // error, not enough memory
}
```

2.2.4. Transfer Function Parameters by Pointer

Consider another approach to transfer function parameters. Use of pointer type parameters was the only way of changing values of the actual parameters in the C programming language.

When passing parameters function by pointer, the pointer itself is passed by value, but object to which it points can be changed and these changes will be available. In this case, the function should obtain the address of the object to be modified, and within a function, dereferencing operation is needed. Here is an example of a function that should change the values of two variables:

```
void swap (int *v1, int *v2)
{
    int temp = *v1;
    *v1 = *v2;
    *v2 = tmp;
}

int main()
{
    int a = 10, b = 20;
    swap(&a, &b);
    // after the function call: a=20, b=10
}
```

2.2.5. Arrays as Parameters of Functions

Arrays, like other variables, can be passed as parameters of functions. To declare the one-dimensional array as a formal parameter functions you can use one of the methods below:

```
// formal parameter is an array of 10 items
```

```
void function (int v[10]);

//формальний параметр – одновимірний масив
void function (int *v);
```

У мові програмування С++ масиви ніколи не передаються за значенням. Коли ім'я масиву передається функції, ця функція отримає як аргумент указівник на його початковий елемент. Виходячи з того, що вказівник передається за значенням, у функції він є локальною змінною та його можна змінювати. Звідси можна зробити два висновки:

– зміни значень в параметрі-масиві у функції неодмінно приведуть до зміни масиву, який був переданий:

```
void function (int v[10])
{
  //...
  v[2]=100;
}

int main()
{
    int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    function(a);
    // a[]=={1, 2, 100, 4, 5, 6, 7, 8, 9, 10}
}
```

– розмір масиву не є частиною його опису як параметра-масиву. Функція, якій передається масив, не знає його реального розміру. Таким чином, перевірка розмірів масиву повністю відсутня:

```
void function (int v[10])
{
    //...
    v[2]=100;
}

int main()
{
    int a[2]={1, 2};
    function(a); //помилка на етапі виконання,
                 // a[2] не існує
}
```

В останньому випадку доцільно при передачі масивів також передавати число елементів як додатковий параметр функції:

```
void function (int *v, int size)
```

```
void function (int v[10]);

// formal parameter is a one-dimensional array
void function (int *v);
```

The C++ programming language does not allow you to transfer arrays by value. When the name of an array is transferred to the function, this function receives a pointer to its initial element. Assuming that the pointer is transferred by value, it can be changed within a function as any local variable. We can draw two conclusions from this:

– item changes of the array-type parameter within the function body certainly lead to changes in the array that was passed:

```
void function (int v[10])
{
  //...
  v[2]=100;
}
int main()
{
    int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    function(a);
    // a[]=={1, 2, 100, 4, 5, 6, 7, 8, 9, 10}
}
```

– the size of an array is not part of its definition. Function, which gets this array, cannot determine its real size. Therefore, any size checking is completely impossible:

```
void function (int v[10])
{
    //...
    v[2]=100;
}

int main()
{
    int a[2]={1, 2};
    function(a); // runtime error,
                 //no such element: a[2]
}
```

In the last case, it is reasonable to send the number of elements as an additional parameter of the function:

```
void function (int *v, int size)
```

```
{
    for (int i=0; i<size; i++)
    {
        cout << v[i] << "  ";
    }
}

int main()
{
    const int size = 10;
    int a[size] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    function(a, size);
}
```

Одним із доцільних способів передачі як параметра функції двовимірного масиву є використання одновимірного масиву та двох параметрів, перший з яких зберігає кількість рядків (rows), а інший – кількість стовбців (cols). Однак при цьому треба відстежувати розташування елемента в одновимірному масиві, як би це було в двовимірному:

```
void print (int *v, int rows, int cols)
{
    int k = 0;
    for (int i=0; i< rows; i++)
    {
        for (int j=0; j<cols; j++, k++)
        {
            cout << v[k] << "  ";
        }
        cout << endl;
    }
}

int main()
{
    const int size = 9;
    int a[size] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    print(a, 3, 3);
    //результат
    //1 2 3
    //4 5 6
    //7 8 9
}
```

```
{
    for (int i=0; i<size; i++)
    {
        cout << v[i] << "  ";
    }
}

int main()
{
    const int size = 10;
    int a[size] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    function(a, size);
}
```

One of the reasonable ways to transfer two-dimensional array to a function is usage of a one-dimensional array, and two extra parameters, the first of which stores the number of lines (rows), and the other - the number of columns (cols). However, it is necessary to track the position of the element in the one-dimensional array, as it was in the two-dimensional array:

```
void print (int *v, int rows, int cols)
{
    int k = 0;
    for (int i=0; i< rows; i++)
    {
        for (int j=0; j<cols; j++, k++)
        {
            cout << v[k] << "  ";
        }
        cout << endl;
    }
}

int main()
{
    const int size = 9;
    int a[size] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    print(a, 3, 3);
    // the result
    //1 2 3
    //4 5 6
    //7 8 9
}
```

2.2.6. Приклади програм

Розробка програм, що маніпулюють масивами достатньо складна задача. Тому, як завжди, необхідно до розробки програм виконувати їх проектування за допомогою діаграм діяльності. Наведемо кілька прикладів класичних задач та визначимо порядок їх розв'язання.

Приклад 1. Розглянемо задачу знаходження суми елементів одновимірного масиву з m елементів. Для визначеності припустимо, що m=4. Продемонструємо технологію знаходження розв'язку задачі, для чого почнемо з розробки діаграми діяльності.

Задачу в цілому можна подати у вигляді трьох підзадач: введення даних, виконання обчислень зі знаходженням суми елементів одновимірного масиву та виведення результату.

Спочатку проілюструємо міркування щодо першої підзадачі. В самому простому випадку цю підзадачу можна розв'язати за допомогою діаграми діяльності лінійної структури. Однак значне збільшення значення m приведе до значного збільшення розміру діаграми діяльності і як слідство до збільшення обсягу тексту програми. Тому лінійну структуру необхідно деяким чином змінити, аби отримати універсальність у діаграмі. Неважко помітити, що всі чотири стани дії схожі між собою та можуть бути замінені формою Ввести a[i]. Крім того, аби формою Ввести a[i] були б розкриті всі чотири стани дії, треба ввести закон зміни ідентифікатора i. Ідентифікатор i повинен набувати початкового значення, що дорівнює 0, кінцевого значення – m і кожного разу збільшуватись на 1. На рисунку 2.3 наведені варіанти розв'язання першої підзадачі.

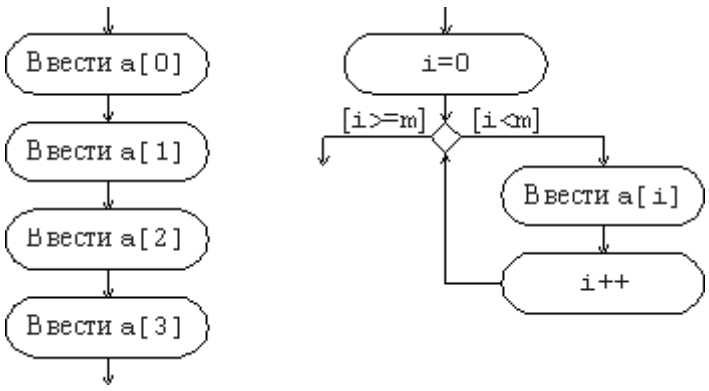

Рисунок 2.3

Аналогічні мірковування можна провести щодо розв'язання третьої підзадачі.

Розв'язання другої підзадачі можливе за схемою, наведеною на рисунку 2.4.

2.2.6. Examples of programs

The development of programs that manipulate arrays is rather complicated task. Therefore, before the development of programs it is usually necessary to carry out their design using activity diagrams. Here are a few examples of classical problems and the order of their decision.

Example 1. Consider the problem of finding the sum of one-dimensional array of m items. For definiteness, assume that m = 4. Demonstrating the technology of problem-solving, we'll start with the development of an activity diagram.

The problem in general can be split into three subtasks: data input, finding the sum of the items of a one-dimensional array, and the result output.

First, consider the first subtask. In the simplest case, this subtask can be solved using an activity diagram of linear structure. However, a significant increase in the value of m leads to a significant increase in the size of activity diagrams and program text. Therefore, a linear structure should be changed in some way to get universal solution.

It is easy to see that all four action states are similar and can be replaced with Enter a [i]. To cover all four action states, it is necessary to define a rule of changing the identifier i. Identifier i should receive an initial value of 0, the final value of m, and each time it should be increased by 1. Figure 2.3 shows the solutions of the first subtask.

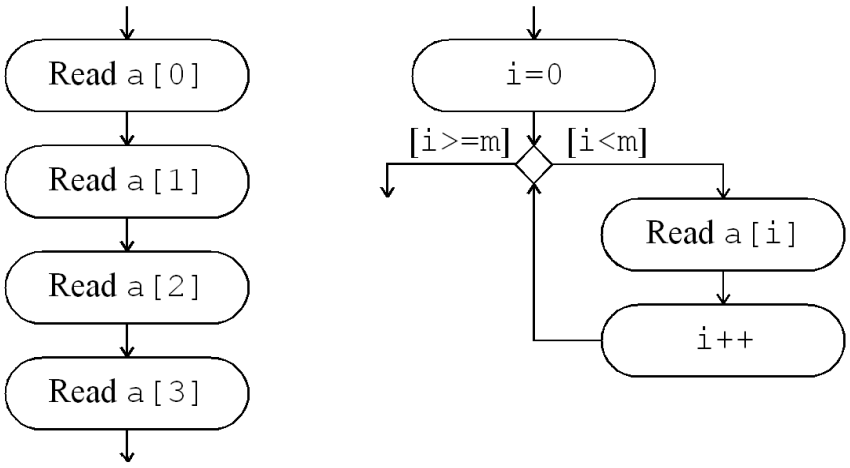

Figure 2.3

Similar arguments can be carried out in solving the third s ubtask.

The solution of the second subtask is possible according to the diagram shown in Figure 2.4.

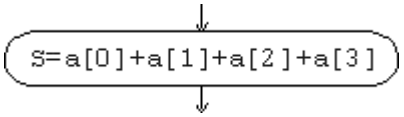

Рисунок 2.4

Недоліки цього варіанта розв'язання задачі схожі на недоліки першої підзадачі з використанням лінійної структури. Невеличкою модифікацією попередньої схеми можна отримати другу підзадачу у більш зручному вигляді, а саме: $S = \sum_{i=0}^{m} a[i]$. На рисунку 2.5 зображено фрагмент діаграми діяльності, що розв'язує другу підзадачу.

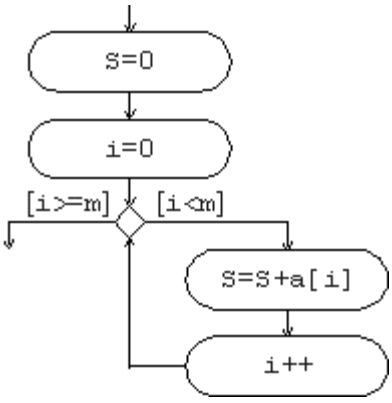

Рисунок 2.5

Поєднання схем розв'язання трьох підзадач дозволить розв'язання задачі в цілому (рисунок 2.6).

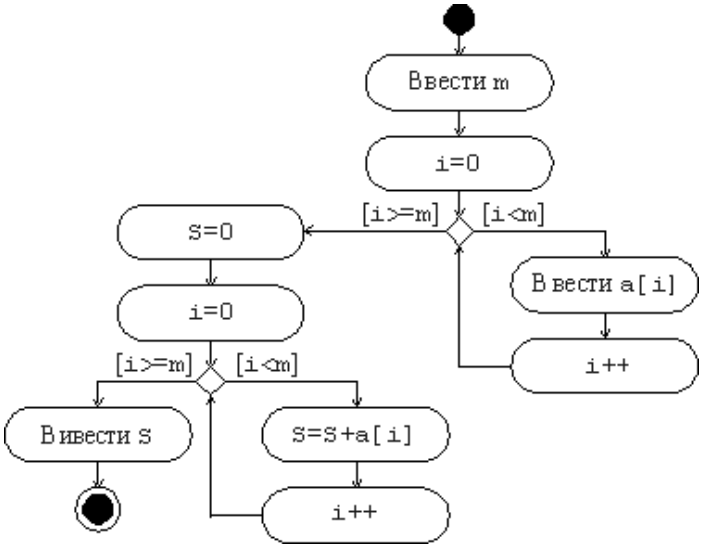

Рисунок 2.6

Наведемо програмний код, що відповідає діаграмі, зображеній на рисунку 2.6.

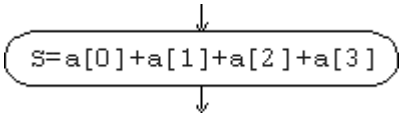

Figure 2.4

Disadvantages of this way of solving the problem similar to disadvantages of the first sub-task based on a linear structure. Slight modification of the previous scheme allows us to obtain the second subtask in a more convenient form, namely: $S = \sum_{i=0}^{m} a[i]$. Figure 2.5 shows the fragment of activity diagram that solves the second subtask.

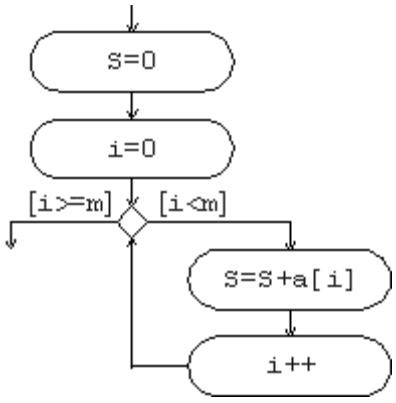

Figure 2.5

The combination of the three schemes for solving the subtasks allows solving the problem as a whole (fig. 2.6).

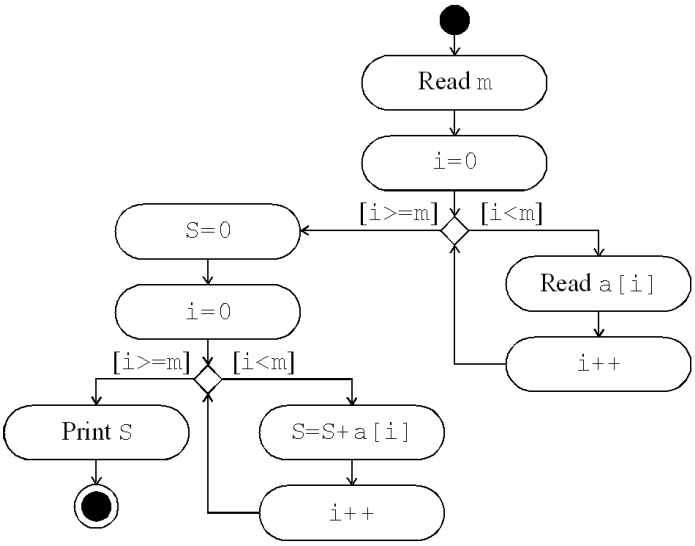

Figure 2.6

Here is the code that corresponds to the diagram shown in Figure 2.6.

```
int main()
{
    int m; // кількість елементів масиву
    cin >> m; // задання вихідного значення змінній m
    // Динамічний розподіл пам'яті:
    int *a = new int [m];
    // Читання елементів масиву:
    for (int i = 0; i < m; i++)
        cin >> a[i];
    // Визначення змінної, що зберігатиме суму
    // елементів масиву:
    double s = 0; // задання початого значення
    // обчислення суми елементів масиву
    for (int i = 0; i < m; i++)
        s += a[i];
    cout << s;    // виведення результату
    delete [] a;  // звільнення пам'яті
    return 0;
}
```

Приклад 2. Звернемось до трохи складнішої задачі знаходження суми елементів побічної діагоналі двовимірного масиву розмірності `m×m`. В цьому випадку необхідно проводити розмірковування, аналогічні *задачі 1*. В результаті отримаємо діаграму діяльності, зображену на рисунку 2.7, та відповідний програмний код.

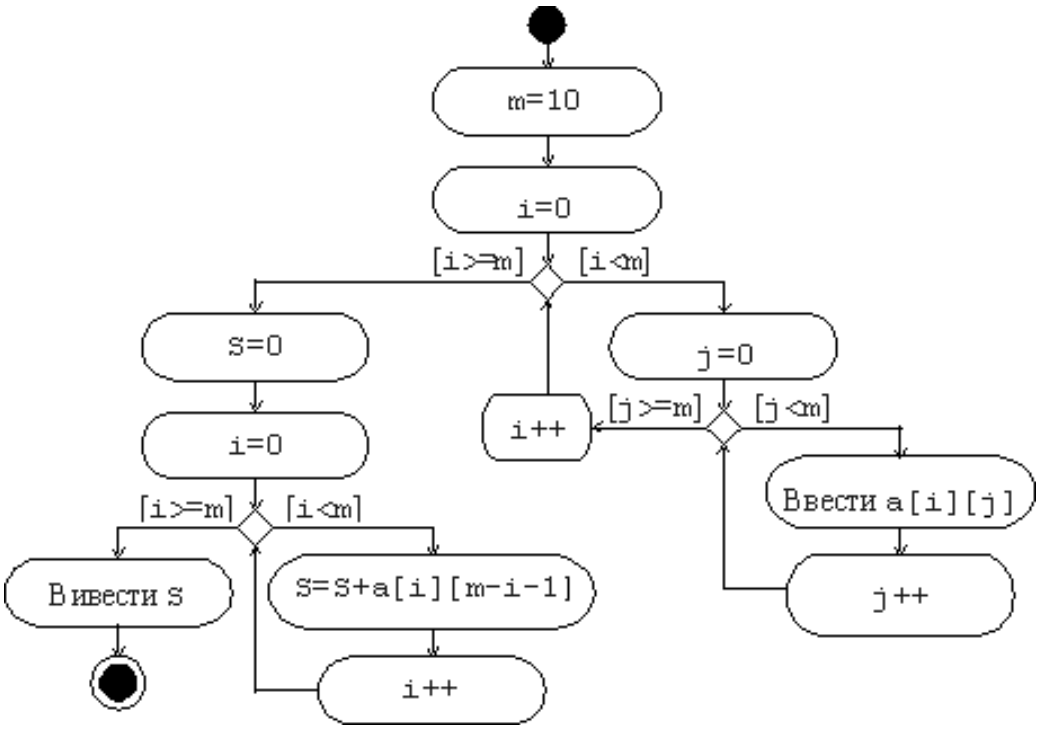

Рисунок 2.7

```
int main()
{
    int m; // number of array items
    cin >> m; // input m
    // // dynamic memory allocation:
    int *a = new int [m];
    // input array items
    for (int i = 0; i < m; i++)
        cin >> a[i];
    // Definition of a variable that will store
    // the sum of array items:
    double s = 0; // setting started value
    // Calculating the sum of array items:
    for (int i = 0; i < m; i++)
        s += a[i];
    cout << s;    // output result
    delete [] a;  // release memory
    return 0;
}
```

Example 2. Consider a slightly more complicated task of finding the sum of items of the secondary diagonal of two-dimensional array with `m` rows and `m` columns. In this case, it is necessary to carry out arguments similar to the previous problem. The result is an activity diagram shown in Figure 2.7, and the corresponding code.

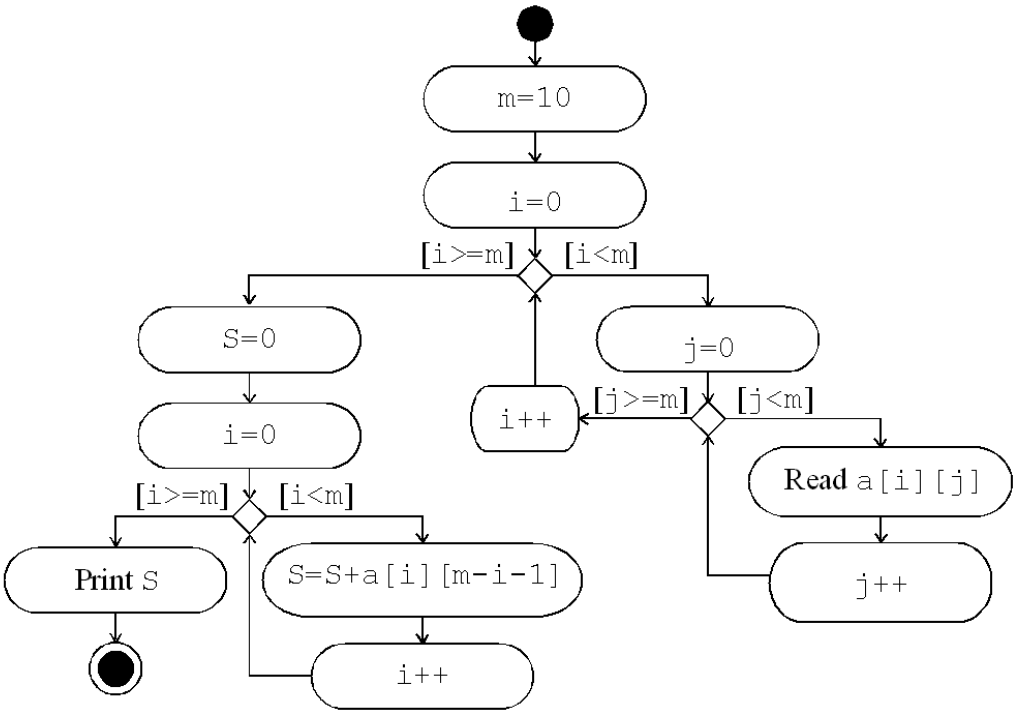

Figure 2.7

```
int main()
{
    // Визначення константи, що зберігає кількість
    // рядків та стовбців двовимірного масиву:
    const int m = 10;
    int a[m][m];// визначення масиву
    // Введення вихідних значень елементів массиву:
    for (int i = 0; i < m; i++)
        for (int j = 0; j < m; j++)
            cin >> a[i][j];
    // визначення змінної, що зберігатиме суму
    // елемен  тів побічної діагоналі масиву:
    double s = 0;//задання початого значення
    // Обчислення суми елементів масиву:
    for (int i = 0; i < m; i++)
        s += a[i][m - i - 1];
    cout << s;// виведення результату
    return 0;
}
```

Приклад 3. Звернемось до більш цікавої задачі, для розв'язання якої потрібно використовувати математичні методи, а саме до задачі сортування елементів одновимірного масиву за зростанням.

Як метод сортування оберемо метод бульбашки [2, 10]. Загальні положення цього методу полягають у наступному.

Розташуємо масив згори донизу, від нульового елемента до останнього (рисунок 2.8).

Рисунок 2.8

На кожному кроці сортування виконується прохід знизу догори масиву. Переглядаються пари сусідніх елементів. Якщо елементи деякої пари знаходяться в неправильній послідовності, то їх міняють містами. Після нульового проходу (рисунок 2.9) по масиву "найлегший" елемент опиниться нагорі (звідси і назва методу – бульбашка).

```
int main()
{
    // Definition of the constant that stores the
    // number of rows and columns
    // of two-dimensional array:
    const int m = 10;
    int a[m][m];// definition of an array
    // Input values of array items
    for (int i = 0; i < m; i++)
        for (int j = 0; j < m; j++)
            cin >> a[i][j];
    // definition of a variable that will store
    // the sum of adverse array diagonal:
    double s = 0;// // setting started value
    // Calculating the sum of array items:
    for (int i = 0; i < m; i++)
        s += a[i][m - i - 1];
    cout << s;// output result
    return 0;
}
```

Example 3. Consider a more interesting problem which solution requires mathematical methods, namely the problem of sorting items of one-dimensional array in ascending order.

We'll choose a method of bubble sort. General concepts of this method are as follows.

First, we place the array from top to bottom, from zero to the last element (fig. 2.8).

Figure 2.8

At each step of the sorting algorithm program scans the array from bottom to top. Pairs of adjacent items are looked through. If the items of a pair are in the wrong order, their values are swapped. After the zero traversal over an array (Figure 2.9) "lightest" item appears on the top (hence the name of the method - bubble).

Рисунок 2.9

Наступний прохід робиться до другого згори елемента, таким чином, другий за величиною елемент піднімається на правильну позицію. Аналогічним чином роблять проходи по нижній частині масиву, яка з кожним проходом зменшується до тих пір, поки в ній не залишиться тільки один елемент. На цьому сортування закінчується, тому що послідовність впорядкована за зростанням. Приклад сортування елементів масиву за кроками наведений на рисунку 2.10.

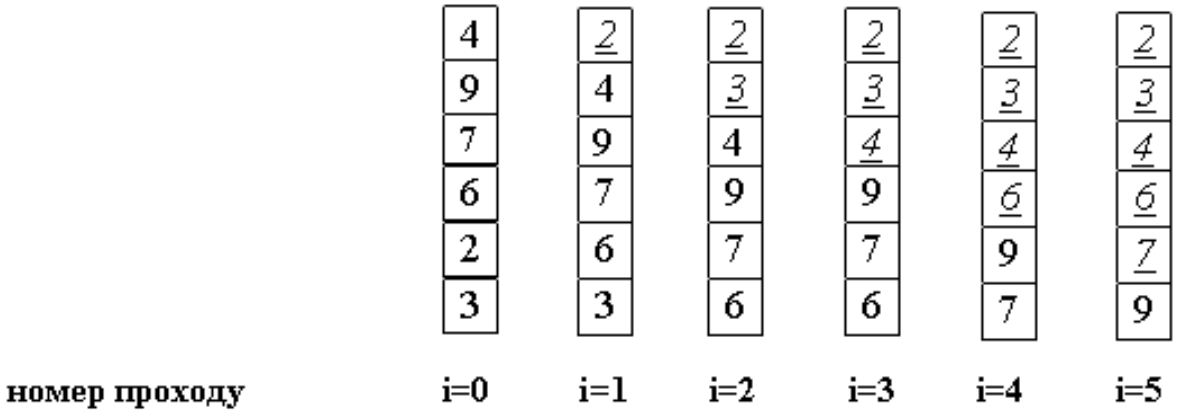

Рисунок 2.10

Середнє число порівнянь та обмінів значень в методі бульбашки має квадратичний порядок. Звідси можна зробити висновок, що метод бульбашки повільний та малоефективний. Але у нього є достатньо велика перевага – він простий і його можна поліпшувати. Приклад поліпшення методу бульбашки у вигляді програмного коду наведений в 3.1.6.

На рисунку 2.11 зображена діаграма діяльності, яка реалізує стандартний метод бульбашки.

Відповідно до діаграми діяльності наведемо текст програми.

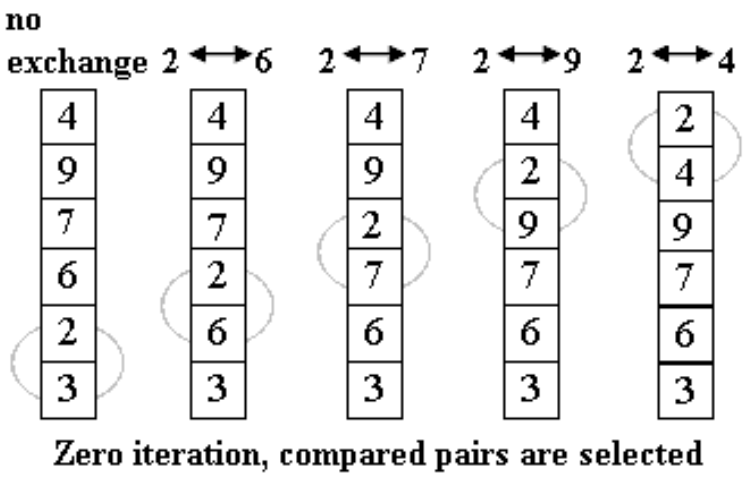

Figure 2.9

The next traversal from the top to the second element is done, thus the second largest element rises to the correct position. Then we traverse over bottom part of an array until it contains the only element. Now sorting is over, because our sequence is already sorted. The example of stepwise sorting array items is shown in Figure 2.10.

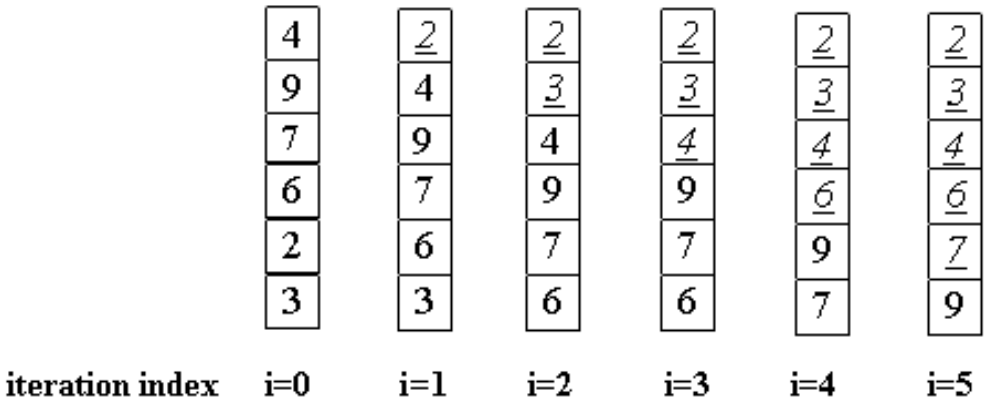

Figure 2.10

The average number of comparisons and exchanges of values in a bubble sort has a quadratic order. Hence, we can conclude that the bubble sort is slow and inefficient. However, it has an advantage: it is simple and it can be improved. An example of improvement bubble sort in the form of program code is given in 3.1.6.

Figure 2.11 shows a diagram of activity that implements the standard method of bubble sorting.

Here is the text of a program corresponding to the activity diagram:

```
int main()
{
    // Визначення константи, що зберігає кількість
    // елементів массиву:
    const int m = 10;
    int a[m], x;// визначення масиву
    // Введення вихідних значень елементам масиву:
    for (int i = 0; i < m; i++)
        cin >> a[i];
    // Сортування елементів за методом бульбашки:
    for (int i = 0; i < m; i++)
        for (int j = m - 1; j > i; j--)
          if (a[j - 1] > a[j])
          {
              x = a[j - 1];
              a[j - 1] = a[j];
              a[j] = x;
          }
    // Виведення елементів відсортованого масиву:
    for (int i = 0; i < m; i++)
        cout << a[i] << "  ";
    return 0;
}
```

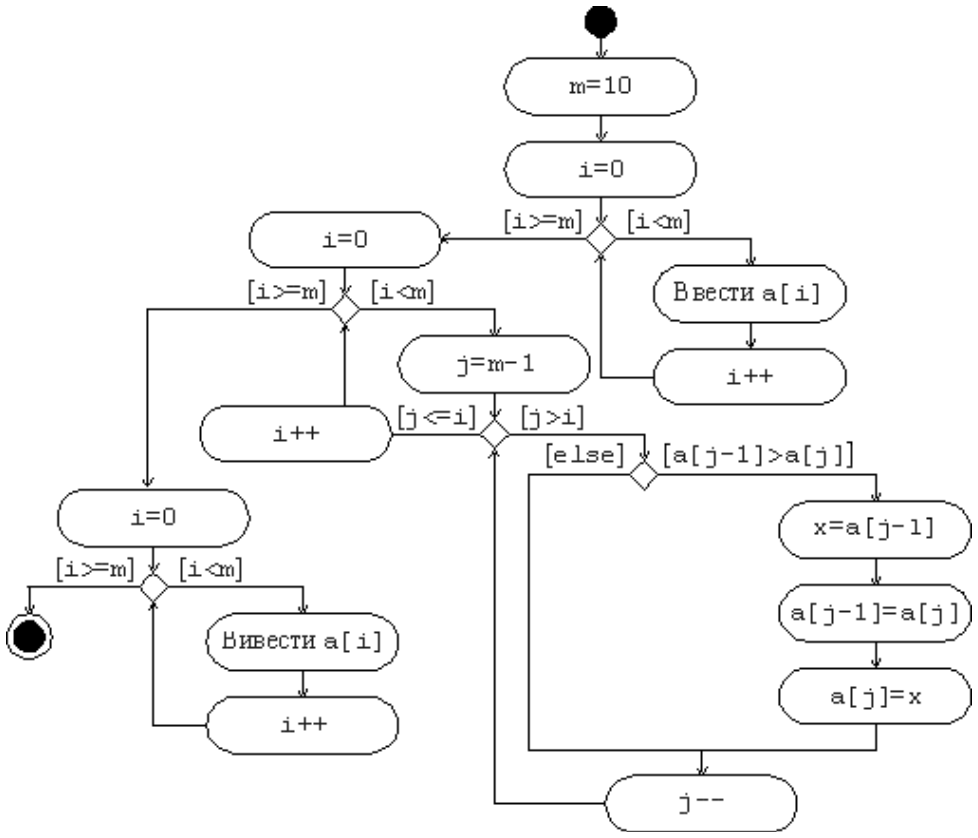

Рисунок 2.11

```
int main()
{
    // Definition of the constant that stores
    // the number of items:
    const int m = 10;
    int a[m], x; // definition of an array
    // Input values of array items:
    for (int i = 0; i < m; i++)
        cin >> a[i];
    // bubble sort
    for (int i = 0; i < m; i++)
    for (int j = m - 1; j > i; j--)
        if (a[j - 1] > a[j])
        {
            x = a[j - 1];
            a[j - 1] = a[j];
            a[j] = x;
        }
    // Output items of the sorted array:
    for (int i = 0; i < m; i++)
        cout << a[i] << "  ";
    return 0;
}
```

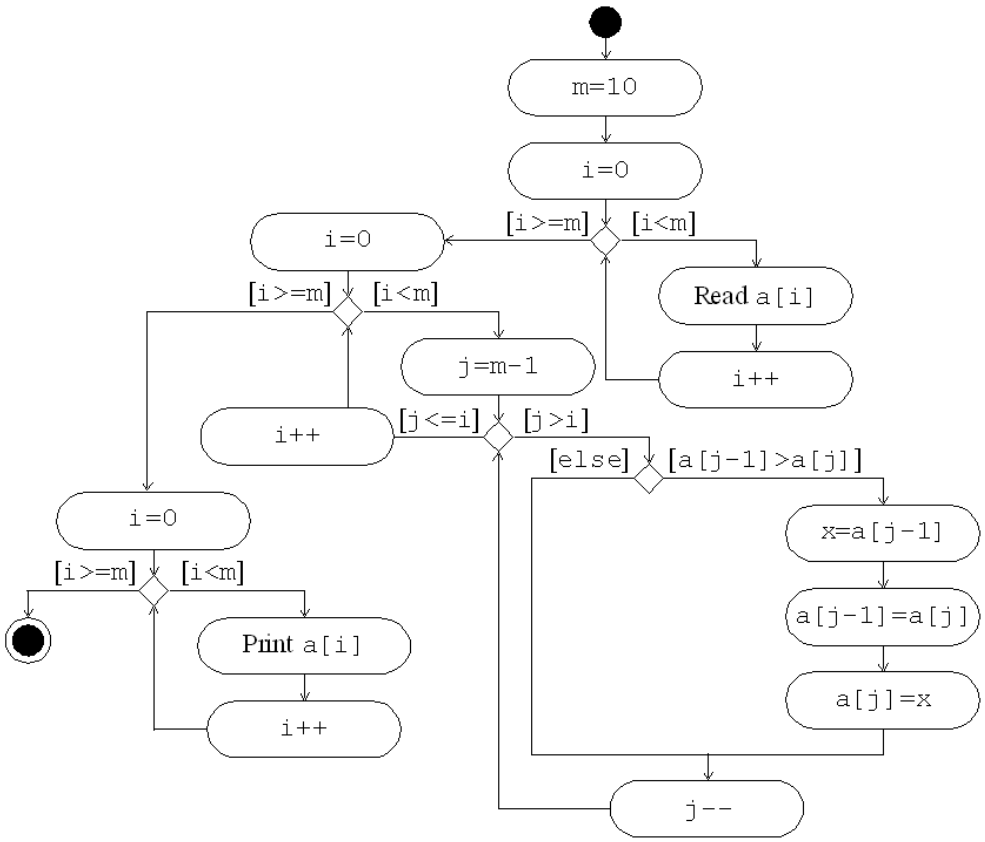

Figure 2.11

Приклад 4. Необхідно проініціалізувати одновимірний масив дійсних чисел списком початкових значень, визначити кількість від'ємних елементів, динамічно розмістити новий масив розмірності, необхідної для збереження всіх від'ємних елементів вихідного масиву. Записати від'ємні елементи в новий масив і вивести результат на екран.

Ініціалізація масиву:

```
const int n = 7;
double a[n] = {10, 11, -3, -2, 0, -1, 4};
```

Підраховуємо кількість від'ємних елементів:

```
int count = 0;
for (int i = 0; i < n; i++)
    if (a[i] < 0)
        count++;
```

Динамічно розміщаємо новий масив розмірності `count`:

```
double *b = new double[count];
```

Записуємо в новий масив від'ємні елементи. При цьому необхідно використовувати окрему змінну для індексування масиву `b`:

```
int j = 0;  // Індекс масиву b
for (int i = 0; i < n; i++)
{
    if (a[i] < 0)
    {
        b[j] = a[i];
        j++;
    }
}
```

Виводимо новий масив і звільняємо динамічну пам'ять:

```
for (j = 0; j < count; j++)
{
    cout <<  b[j] << ' ';
}
delete [] b;
```

Приклад 5. Необхідно ввести з клавіатури розміри двовимірного масиву (кількість рядків і кількість стовпців), розмістити двовимірний масив цілих чисел у динамічній пам'яті, ввести з клавіатури значення елементів масиву, знайти суму перших елементів рядків і вивести її на екран.

Спочатку користувач вводить розміри масиву – `m` (кількість рядків) і `n` (кількість стовпців):

Example 4. We need to do the following: to initialize a one-dimensional array of real numbers using a list of initial values, to determine the number of negative items, to allocate a new array of size required to keep all the negative items in the heap, to write the negative items into the new array, and to display the result on screen.

Array initialization:

```
const int n = 7;
double a[n] = {10, 11, -3, -2, 0, -1, 4};
```

Counting the number of negative items:

```
int count = 0;
for (int i = 0; i < n; i++)
    if (a[i] < 0)
        count++;
```

Dynamic allocation of a new array of `count` items:

```
double *b = new double[count];
```

Negative items are stored in a new array. It is necessary to use a separate variable for indexing of `b`:

```
int j = 0;  // Index of b
for (int i = 0; i < n; i++)
{
    if (a[i] < 0)
    {
        b[j] = a[i];
        j++;
    }
}
```

Output of a new array and its deallocation:

```
for (j = 0; j < count; j++)
    cout <<  b[j] << ' ';
delete [] b;
```

Example 5. It is necessary to enter the size of two-dimensional array (the number of rows and number of columns), to place a two-dimensional array of integers in the heap, to enter the values of array items, to find the sum of the first column, and to display it on the screen.

First, the user enters the size of the array: `m` (number of rows) and `n` (number of columns):

```
int m, n;
cout << "Enter m and n:" << endl;
cin >> m >> n;
```

Для розв'язання задачі розміщення двовимірного масиву в динамічній пам'яті можна використовувати такий підхід: розмістити спочатку масив указівників на ціле розмірності `m`, потім кожен рядок розмістити в динамічній пам'яті в циклі, причому кожен указівник розміщеного раніше масиву вказівників буде вказувати на відповідний рядок:

```
int **a = new int* [m];
for (int i = 0; i < m; i++)
    a[i] = new int [n];
```

Далі в циклі вводяться всі елементи масиву:

```
cout << "Enter items of array:" << endl;
for (int i = 0; i < m; i++)
{
    for (int j = 0; j < n; j++)
    {
        cin >> a[i][j];
    }
}
```

Обчислюється сума елементів першого (нульового) стовпця і виводиться на екран:

```
int b = 0;
for (int i = 0; i < m; i++)
    b += a[i][0];
cout << b;
```

Зайняту раніше динамічну пам'ять необхідно звільнити:

```
for (int i = 0; i < m; i++)
    delete [] a[i];
delete [] a;
```

2.2.7. Вправи для самостійної роботи

Вправа 1. Напишіть програму, яка обчислює суму додатних елементів масиву дійсних чисел.

Вправа 2. Напишіть програму, яка обчислює суму мінімального і максимального елементів масиву дійсних чисел.

Вправа 3. Напишіть програму, що сортує елементи цілого масиву за зменшенням.

Вправа 4. Напишіть програму, яка обчислює суму додатних елементів двовимірного масиву.

```
int m, n;
cout << "Enter m and n:" << endl;
cin >> m >> n;
```

To solve the problem of locating a two-dimensional array on the heap, you can use the following approach: allocate the array of m pointers to integer in the heap, then allocate each row in the heap in a loop, so each pointer of previously created array of pointers should point to the appropriate line:

```
int **a = new int* [m];
for (int i = 0; i < m; i++)
    a[i] = new int [n];
```

Now we can enter all the items of the array in a loop:

```
cout << "Enter items of array:" << endl;
for (int i = 0; i < m; i++)
{
    for (int j = 0; j < n; j++)
    {
        cin >> a[i][j];
    }
}
```

The sum of the items of the first (zero) column is calculated and displayed:

```
int b = 0;
for (int i = 0; i < m; i++)
    b += a[i][0];
cout << b;
```

Previously allocated dynamic memory should be released:

```
for (int i = 0; i < m; i++)
    delete [] a[i];
delete [] a;
```

2.2.7. Exercises for self-study

Exercise 1. Write a program that calculates sum of positive items of an array of doubles.

Exercise 2. Write a program that calculates the sum of the minimum and maximum items of an array of real numbers.

Exercise 3. Write a program that sorts items of integer array in descending order.

Exercise 4. Write a program that calculates the sum of positive items two-dimensional array.

Вправа 5. Напишіть програму, у якій визначається та ініціалізується двовимірний масив і обчислює добуток максимальних елементів її стовпців.

Вправа 6. Напишіть програму, яка обчислює максимальний і мінімальний елементи масиву, використовуючи дві окремі функції.

Вправа 7. Напишіть програму, яка читає розміри двовимірного масиву, розміщає масив у динамічній пам'яті, читає елементи масиву, обчислює суми рядків і поміщає ці суми у новоий масив.

Контрольні запитання

1. Опис масивів.
2. Обмеження на вирази для визначення кількості елементів масиву.
3. Виділення пам'яті для розташування масивів.
4. Сполучення визначення та ініціалізації масивів.
5. Індексація елементів масиву. Початкове та кінцеве значення індексу.
6. Використання циклів для обходу масиву.
7. Використання імені масиву без квадратних дужок.
8. Особливості роботи з двовимірними масивами.
9. Поняття вказівника.
10. Операція отримання адреси.
11. Використання вказівника **`void`***.
12. Вказівники на константний об'єкт та константні вказівники.
13. Зв'язок масивів з указівниками.
14. Адресна арифметика.
15. Виділення та звільнення динамічної пам'яті.
16. Розміщення масивів у динамічній пам'яті.
17. Передача параметрів функції за вказівником.
18. Масиви як параметри функцій.
19. Алгоритм сортування елементів одновимірного масиву.

Exercise 5. Write a program that declares two-dimensional array and calculates the product of the maximal items of its columns.

Exercise 6. Write a program that calculates the sum of the minimal and maximal items of an array of doubles using two separate functions.

Exercise 7. Write a program that reads the size of two-dimensional array, allocates an array in free store, reads items of an array, calculates sums of rows and puts these sums into a new array.

Advancement Questions

1. Definition of arrays.
2. Restrictions on expression that determine the number of array items.
3. Memory allocation for the array.
4. The combination of definition and initialization of arrays.
5. Indexing of array items. Starting and ending indices.
6. Use of cycles for bypass the array.
7. Use of array names without brackets.
8. Features of two-dimensional arrays.
9. The concept of pointer.
10. Operation of getting the address.
11. Use of void* pointer.
12. Pointers to constants and constant pointers.
13. Relationship between arrays and pointers.
14. Address arithmetic.
15. Obtaining and release of memory in the heap.
16. Placing arrays in the heap.
17. Passing function parameters by pointer.
18. Arrays as parameters of functions.
19. Algorithm for sorting the items of a one-dimensional array.

Розділ 3. МОДУЛЬНЕ ТА ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

У розділі розглядається два способи створення і підтримки великих програм – модульне і об'єктно-орієнтоване програмування. Позитивною особливістю даних способів є те, що конструювання великої складної програми виконується з окремих фрагментів, кожен з яких можна вважати більш керованим, ніж складна програма в цілому.

3.1. Модульне програмування мовою C++

Найпростіші програми, написані мовою C++, навіть ті, які використовують окремі функції для реалізації підзадач, можуть бути розміщені всередині одного файлу та не потребують розподілу на модулі. Кількість глобальних констант, змінних та функцій у таких програмах обмежена. Як наслідок, легко унеможливити виникнення конфліктів імен.

На жаль, складність реальних програм не дозволяє розмістити увесь необхідний код в одному вихідному файлі. Необхідне групування вихідного тексту в так звані модулі. Зокрема доцільно виділити в окремі модулі функції, які можуть придатися у декількох програмних проектах.

Модульне програмування є результатом подальшого розвитку процедурно-орієнтованого програмування. Фактично здійснюється групування коду на більш високому рівні. Модульне програмування дозволяє залучити до проекту кількох програмістів, кожний з яких відповідає за окремий модуль.

Модулі визначають структуру програми. Мова C++ дозволяє окремо здійснювати логічне та фізичне структурування. Ці підходи реалізовані відповіно через механізми просторів імен та одиниць трансляції.

3.1.1. Простори імен

Простори імен визначають логічну структуру програми на C++.

Part 3. MODULAR AND OBJECT-ORIENTED PROGRAMMING

Two ways of creating and maintaining large programs - modular and object-oriented programming are considered in this part. A positive feature of these methods is that the design of large complex programs executed on separate pieces, each of which can be considered more manageable, than a complicated program as a whole.

3.1. Modular Programming in C++

The simplest programs written in C++, even those that use separate functions to implement subtasks, can be placed inside a single file and do not require division into modules. The number of global constants, variables and functions in such programs is limited. As a result, it is easy to prevent name conflicts.

Unfortunately, the complexity of real applications does not allow to place all the necessary code in the same source file. Grouping of the source text in the so-called modules is necessary. In particular, it is advisable to allocate functions in separate modules that can be useful in several software projects.

Modular programming is the result of further development of the procedure oriented programming. In fact, it carries out grouping of code at a higher level. Modular programming will allow to involve several programmers to the project, each of whom is responsible for a separate module.

Modules define the structure of the program. C++ allows you to perform separately logical and physical structuring. These approaches are implemented, respectively, through the mechanisms of namespaces and translation units.

3.1.1. Namespaces

Namespaces define the logical structure of the C++ program.

Простір імен (**namespace**) – пойменована частина файлової області видимості, в якій можуть міститись оголошення та визначення. Простори імен допомагають уникнути конфліктів імен. Вони надають механізм логічного групування вихідного тексту програми.

Для того щоб визначити простір імен, використовують ключове слово **namespace**, після якого вказують оригінальне ім'я, яке ще не було застосовано у глобальній області видимості. Далі у фігурних дужках розташовують необхідні описи.

```
namespace MySpace
{
    int  k = 10;

    void f(int n)
    {
        k = n;
    }
}
```

Функції, оголошення яких здійснено у просторі імен, можуть бути реалізовані поза простором. Наприклад,

```
namespace MySpace
{
    int  k = 10;
    void f(int n);
}



void MySpace::f(int n)
{
    k = n;
}
```

Простори імен можна вкладати в інші простори імен:

```
namespace FirstSpace
{
    namespace SecondSpace
    {

    }
}
```

Можна визначити синонім простору імен:

```
namespace YourSpace = MySpace;
```

Namespace is an optionally named part of a file scope, which can contain declarations and definitions. Namespaces help to avoid name conflicts. They give a mechanism for logical grouping of source code.

In order to define the namespace, you should use **namespace** keyword, followed by original name that previously was not used in global scope. Then necessary definitions are placed, put in braces.

```
namespace MySpace
{
    int  k = 10;

    void f(int n)
    {
        k = n;
    }
}
```

Functions declared in namespace can be defined outside it. For example,

```
namespace MySpace
{
    int  k = 10;
    void f(int n);
}



void MySpace::f(int n)
{
    k = n;
}
```

Namespaces can be nested in other namespaces:

```
namespace FirstSpace
{
    namespace SecondSpace
    {

    }
}
```

You can define an alternate name to refer to a namespace identifier:

```
namespace YourSpace = MySpace;
```

Простори імен можуть закриватися та знову розкриватися для подальшого розширення. Для доповнення простору імен його необхідно знову відкрити. Оголошення можуть знаходитись у кількох файлах:

```
namespace FirstSpace
{
    // Перша частина простору імен FirstSpace
}

... // Інші описи

namespace SecondSpace
{
    // Простір імен SecondSpace
}

namespace FirstSpace
{
    // Друга частина простору імен FirstSpace
}
```

Є три можливості доступу до елементів простору імен: явна вказівка простору перед кожним ім'ям (використання повного кваліфікатора), використання **using**-оголошення та using-директиви. Повний кваліфікатор складається з імені простору імен, оператора дозволу області видимості (::) та безпосереднього імені елемента:

```
int x = MySpace::k;
```

Для індивідуального підключення імен певного простору можна скористатись **using**-оголошенням, яке додає відповідний ідентифікатор до глобальної або локальної області видимості залежно від того, де здійснено оголошення.

```
using MySpace::k;
using MySpace::f;

int y = k + f(k);
```

Якщо необхідно мати доступ до кількох (або усіх) елементів певного простору, мова С++ пропонує спосіб, який базується на використанні **using**-директиви, після включення якої всі ідентифікатори простору імен стають доступними нарівні з іменами глобальної області видимості. Наприклад:

```
using namespace MySpace;
```

Namespaces are discontinuous and can be open for additional development. If you want to add new declarations to existing namespace, you should only redeclare it. Declarations of the same namespace can be placed into several files:

```
namespace FirstSpace
{
    // The first part of FirstSpace namespace
}

... // Other declarations

namespace SecondSpace
{
    // namespace SecondSpace
}

namespace FirstSpace
{
    // The second part of FirstSpace namespace
}
```

There are three ways to access the elements of a namespace: by explicit access qualification, through the **using**-declaration, or through the using-directive. The full qualifier consists of the namespace name, scope resolution operator (::) and the name of the element itself:

```
int x = MySpace::k;
```

In order to connect the individual names of a certain namespace, you can apply **using**-declaration, which adds the corresponding identifier to the global or local scope, depending on where the announcement is made

```
.
using MySpace::k;
using MySpace::f;

int y = k + f(k);
```

If you need to access some (or all) elements of certain space, C++ provides a method, based on the so-called **using**-directives, after the inclusion of it all namespace identifiers are available along with the names of global scope. For example:

```
using namespace MySpace;
```

Слід запобігати вживанню **using**-директиви, бо це підвищує небезпеку виникнення конфліктів імен.

Можна використати **using**-директиви для об'єднання кількох просторів імен:

```
namespace NewSpace
{
    using namespace FirstSpace;
    using namespace SecondSpace;
}
```

За допомогою **using**-оголошення можна вибрати необхідні імена з інших просторів імен:

```
namespace NewSpace
{
    using OtherSpace::name1;
    using OtherSpace::name2;
}
```

Таке оголошення може бути використано в кількох проектах. Це можна здійснити завдяки використанню заголовних файлів. Про цей механізм буде повідомлено далі.

Майже всі елементи Стандартної бібліотеки C++ згруповані у просторі імен `std`. У цьому просторі оголошені додаткові підпростори, такі, як `std::rel_ops`.

Існує можливість опису так званих *безіменних просторів імен*. Такі простори не можна підключити за допомогою **using**. У межах файлу, в якому створено безіменний простір, є вільний доступ до всіх імен.

```
namespace
{
    int j = 10;
}

int k = j; // j з безіменного простору
```

За межами файлу доступ взагалі неможливий.

3.1.2. Використання заголовних файлів

Кожна нетривіальна програма може бути розділена на відносно універсальні частини, які можуть бути використані в кількох проектах, та частини, специфічні для конкретного проекту. Універсальні та специфічні частини доцільно зберігати в окремих файлах (одиницях трансляції).

You should avoid the use of **using**-directive, because it increases the risk of name conflicts.

The **using**-directive can be used if you want to join several namespaces:

```
namespace NewSpace
{
    using namespace FirstSpace;
    using namespace SecondSpace;
}
```

With **using**-declaration, you can select the required names from other namespaces:

```
namespace NewSpace
{
    using OtherSpace::name1;
    using OtherSpace::name2;
}
```

Such a declaration may be used in several projects. This can be done through the use of header files. The mechanism of header files will be considered later.

Almost all elements of the Standard C++ Library are grouped into namespace called `std`. This namespace includes additional nested namespaces, such as `std::rel_ops`.

It is possible to declare so-called anonymous namespaces. These spaces cannot be connected by means of **using**. Within the file, which contains declaration of anonymous space, all names can be free accessed.

```
namespace
{
    int j = 10;
}

int k = j; // j from the anonymous space
```

Outside the file, the access is generally impossible.

3.1.2. The Use of Header Files

Every non-trivial program can be divided into relatively universal pieces that can be used in several projects, and parts that are specific to a particular project. It is advisable to keep universal and specific parts in separate files (translation units).

Найпростіший шлях для реалізації цієї ідеї полягає у застосуванні директиви препроцесора `#include`, яка дозволяє перед компіляцією включити вихідний текст з одного файлу всередину іншого. Препроцесор не здійснює фізичного копіювання вмісту файлу. Замість цього препроцесор створює новий вихідний текст в оперативній пам'яті. Цей текст має назву *одиниці трансляції* (результат обробки вихідного тексту препроцесором).

Можна також видалити певні частини вихідного тексту за допомогою директив `#define`, `#ifdef` та `#ifndef`. Директива `#define` дозволяє визначити нову змінну препроцесора в будь-якому місці. В іншому місці можна перевірити цей факт за допомогою директив `#ifdef` або `#ifndef`. Наприклад:

```
#define New_Name

...

#ifdef New_Name
// Включаємо цей код в одиницю трансляції
#else
// Не включаємо цього коду в одиницю трансляції
#endif
```

Один проект може містити кілька одиниць трансляції. Імена, визначені в одних одиницях, повинні бути оголошені в інших одиницях. Включення заголовних файлів гарантує точне відтворення оголошень у всіх одиницях трансляції.

У заголовний файл можна включати такі елементи:

– іменовані простори імен;
– визначення типів;
– оголошення функцій;
– оголошення даних (з ключовим словом **`extern`**);
– визначення констант;
– директиви препроцесора;
– коментарі.

Заголовні файли не повинні містити:

– визначень звичайних функцій;
– визначень даних;
– безіменних просторів імен.

Традиційно заголовні файли мають розширення `.h`, тоді як файли з реалізацією функцій та визначенням даних мають розширення `.cpp`.

Існує велика кількість стандартних заголовних файлів з описами класів та функцій. Для включення таких файлів замість лапок слід

The simplest way to implement this idea is to apply the preprocessor directive `#include`, which allows you to include a source code from one file into another before compilation. Preprocessor does not copy the contents of the physical file. Instead, the preprocessor creates a new source code in memory. This text is called translation unit (the result of processing the original text by preprocessor).

You can also remove some parts of the source text using directives `#define`, `#ifdef`, and `#ifndef`. The `#define` directive allows you to define a new preprocessor variable anywhere in the source text. Elsewhere, you can check this fact by using the directives `#ifdef` or `#ifndef`. For example:

```
#define New_Name

...

#ifdef New_Name
// Include this code into translation unit
#else
// Do not include this code into translation unit
#endif
```

One project can contain multiple translation units. The names defined in the same units should be declared in other units. Inclusion of header files ensures accurate reproduction of declarations in all translation units.

Header file can include the following elements:

– named namespaces;
– type definitions;
– function declarations;
– data declarations (with extern keyword);
– constant definitions;
– preprocessor directives;
– comments.

Header files cannot contain:

– ordinary function definitions;
– data definitions;
– anonymous namespaces.

Header files are conventionally suffixed by `.h`, whereas files containing function or data definitions are suffixed by `.cpp`.

There are numerous standard header files that contain declarations of standard classes and functions. The names of such files in `#include` directive

вживати кутові дужки <> для того, щоб препроцесор шукав такі файли в стандартних теках. В іншому випадку препроцесор починає пошук з поточної теки.

3.1.3. Стражі включення (Include Guards)

Дуже часто у вихідний текст необхідно включати більше ніж один заголовний файл. Крім того, дуже часто є необхідність включення тексту одного заголовного файлу в інший заголовний файл. Наприклад, є необхідність включення файлу `f1.h` у файл `f2.h`, файл `f3.h` потребує включення обох файлів `f1.h` та `f2.h`, а всі ці файли необхідні для компіляції основної програми:

```
//f1.h
...

//f2.h
#include "f1.h"
...

//f3.h
#include "f1.h"
#include "f2.h"
...

//main.cpp
#include "f1.h"
#include "f2.h"
#include "f3.h"
...
```

Препроцесор включає вміст файлу `f1.h` в одиницю трансляції, коли він обробляє файл `main.cpp`. Потім він включає вміст файлу `f2.h`, який також містить включення `f1.h`. Після включення `f3.h` одиниця трансляції містить три копії тексту `f1.h` та дві копії `f2.h`. Таке включення не має сенсу, а також може викликати помилки під час компіляції.

Найчастіше цю проблему вирішують за допомогою так званих *стражів включення* – спеціальної конструкції з директив препроцесора. Наприклад, текст заголовного файлу `f1.h` можна організувати в такий спосіб:

```
#ifndef F1_H
#define F1_H
```

must be written in <> instead of "". That causes preprocessor to look for such files in standard directories. Otherwise, preprocessor searches the header starting from current directory.

3.1.3. Include Guards

Very often, the source code needs to include more than a single header file. In addition, very often, it is necessary to include the text of one header file into another header file. For example, header file `f2.h` needs to include file `f1.h`, header file `f3.h` needs to include files `f1.h` and `f2.h`, and you need to include all of them into main program:

```
//f1.h
...

//f2.h
#include "f1.h"
...

//f3.h
#include "f1.h"
#include "f2.h"
...

//main.cpp
#include "f1.h"
#include "f2.h"
#include "f3.h"
...
```

Preprocessor includes contents of `f1.h` into translation unit when it processes the file `main.cpp`. Then it includes the contents of `f2.h` that also contains inclusion of `f1.h`. After inclusion of `f3.h`, the translation unit contains four copies of `f1.h` and two copies `f2.h`. This inclusion makes no sense and may cause errors during compilation.

In most cases, this problem can be solved by so-called *include guards* (inclusion guards), which are a special construct of the preprocessor directives. For example, the text of header file `f1.h` can be organized as follows:

```
#ifndef F1_H
#define F1_H
```

```
... // текст заголовного файлу
#endif
```

Вперше, коли у програмі згадується включення заголовного файлу, препроцесор здійснює читання першого рядка (`#ifndef`) та включає весь текст в одиницю трансляції, оскільки змінна `F1_H` ще не була визначена. Окрім того, препроцесор визначає цю змінну. Вдруге та будь-коли потім препроцесор не включає тексту заголовного файлу, оскільки змінна `F1_H` вже визначена. Ім'я, яке визначається у директиві `#define`, може бути довільним. Важливо тільки, щоб воно було визначене тільки в одному заголовному файлі. В іменах змінних препроцесора не можна застосовувати крапку.

3.1.4. Глобальні змінні та модульне програмування

Звернемо увагу на два винятка щодо видимості глобальних змінних у програмах, побудованих з кількох одиниць трансляції.

По-перше, глобальна змінна, визначена в одному файлі може бути використана в іншому або навіть у декількох файлах:

```
//файл a1.cpp
int a = 10; //глобальна змінна
//.....................

//файл a2.cpp
int main()
{
//помилкова спроба використання a з файлу a1.cpp
int k = a;
//.....................
}
```

У цьому випадку для використання змінної `a` у файлі `a2.cpp` в ньому повинно бути її оголошення. Останнє реалізується з використанням механізму *зовнішніх змінних*.

По-друге, якщо дві глобальні змінні оголошуються (визначаються) в двох різних програмних файлах і мають однакове ім’я, то це призведе до помилки. Помілка полягає в тому, що має місце повторне визначення ім’я:

```
//файл a1.cpp
int a = 10; //глобальна змінна
//.....................

//файл a2.cpp
```

```
... // text of header file
#endif
```

For the first time, when the inclusion of the header file is mentioned in the program, the preprocessor reads the first line (`#ifndef`) and includes all the text into the translation unit, because the variable F1_H has not been defined. Additionally, the preprocessor determines the variable. In the second and subsequent times, preprocessor does not include the text of the header file, because the variable `F1_H` has been already defined. The name that is defined in the `#define` directive, can be arbitrary. The only important thing is that it must be defined within the only header file. Preprocessor variable names cannot contain dots.

3.1.4. Global Variables and Modular Programming

Note two exceptions under the scope of global variables in programs built from multiple translation units.

First, the global variables defined in the same file can be used in another, or even multiple files:

```
// file a1.cpp
int a = 10; // global variable
//.....................

// file a2.cpp
int main()
{
// erroneous attempt to use a from a1.cpp
int k = a;
//.....................
}
```

In this case, if you need to use a variable within the file `a2.cpp`, there must be present its declaration. This is realized with the use of *external variables*.

Second, if the two global variables are defined in two different program files and have the same name, you get an error. The error consists in the fact that there is a redefinition of the name:

```
// file a1.cpp
int a = 10; // global variable
//.....................

// file a2.cpp
```

```
int a = 10; //друге глобальне ім'я a - помилка

int main()
{
//.....................
}
```

Розглянемо механізм використання *зовнішніх змінних*. Раніш вказувалось, що змінна може бути визначена та оголошена. Різниця між цими поняттями практично тотожня, якщо програма створюється в межах однієї одиниці трансляції. При використанні модульного програмування поняття оголошення змінної дещо змінюється. А саме, для того щоб оголосити змінну, перед її описом необхідно поставити службове слово **extern**:

```
extern int i;
```

Це за дією аналогічно прототипу функції. Такий запис вказує на те, що в програмі існує визначення ідентифікатора і з тим самим типом.

Оголошення **extern** не призводить до виділення пам'яті та може зустрічатись в програмі багаторазово. Наведемо найпростіший приклад:

```
//файл a1.h
#ifndef A1_H
#define A1_H

extern int a; //оголошення зовнішньої змінної a

// текст заголовного файлу
#endif

//файл a1.cpp
int a = 10; //визначення глобального ідентифікатора a
//.....................

//файл a2.cpp
#include "a1.h" //тут присутнє оголошення змінної a

int main()
{
    int k=a; //тут можна використовувати a з a2.cpp
//.....................
}
```

```
int a = 10; // error: the second global name a

int main()
{
//.....................
}
```

Consider the mechanism for *external variables*. Previously it was mentioned that a variable can be defined and declared. The difference between these concepts is practically unessential if the program is created within a single translation unit. The usage of modular programming concept of a variable declaration is somewhat different. Namely, in order to declare a variable before its definition, it is necessary to prefix its description with the keyword **`extern`**:

```
extern int i;
```

This is similar to the function prototype declaration. This record indicates that somewhere in the program there is a definition of identifier `i` of the same type.

The **`extern`** declaration does not involve memory allocation and can occur multiple times in the program. Consider the simplest example:

```
// file a1.h
#ifndef A1_H
#define A1_H

extern int a; // declaration of extern variable a

// the text of header file
#endif

// file a1.cpp
int a = 10; // definition of global variable a
//.....................

// file a2.cpp
#include "a1.h" // here is a declaration of a

int main()
{
    int k=a; // here you can use a from a2.cpp
//.....................
}
```

```
//файл a3.cpp
#include "a1.h" // оголошення зовнішньої змінної a

int f()
{
    int m=a; // можна використовувати a з a1.cpp
//.....................
}
```

Якщо при оголошенні зовнішньої змінної буде виконана її ініціалізація, то таке оголошення одразу перетворюється у визначення і будь-які наступні визначення змінної призведуть до помилки:

```
extern int a = 20; // визначення, а не оголошення
```

Службове слово **extern** можна використовувати при оголошенні функцій для зручності, щоб одразу відрізняти оголошення від заголовка та визначення:

```
extern int minimum(int, int); //оголошення функції
```

3.1.5 Вказівники на функцію

Використання окремих одиниць трансляції надає можливість створення бібліотек функцій, які потім будуть використані в різних програмах. У більшості випадків уся необхідна інформація може бути надана функціям у вигляді даних – чисел, символів, рядків тощо. Але іноді необхідно передати функціональну залежність. Наприклад, якщо необхідно знайти корені, похідну, інтеграл від довільної функції, побудувати графік. Для реалізації таких алгоритмів необхідно передбачити можливість обчислення певної функції у будь-якій точці визначеного інтервалу. З точки зору програмування, такі алгоритми потребують доступу до програмного коду, який може бути використаний довільну кількість разів залежно від логіки алгоритму. У мовах С та С++ таку можливість надають так звані вказівники на функції.

Вказівник на функцію – це адреса, починаючи з якої зберігається скомпільований код функції, тобто адреса, за якою передається керування при виклику цієї функції. Подібно до того, як ім'я масиву є постійним указівником на перший елемент масиву, ім'я функції є постійним указівником на функцію. Можна визначити змінну-вказівник, який вказує на функцію, та викликати функцію за допомогою цього вказівника.

Вказівник на функцію повинен вказувати на функцію з відповідним типом результату та сигнатурою. У визначенні

```
int (*funcPtr) (double);
```

```
// file a3.cpp
#include "a1.h" // declaration of a variable a

int f()
{
    int m=a; // here you can use a from a2.cpp
//.....................
}
```

If the declaration of an external variable included its initialization, such declaration turns into definition and any subsequent definitions of the variable will result in an error:

```
extern int a = 20; // definition, but not declaration
```

The **extern** keyword can be also used before function declaration for convenience to distinguish declaration from function header and definition:

```
extern int minimum(int, int); // function declaration
```

3.1.5 Pointers to Functions

The use of separate translation units allows creating a library of functions, which can be used then in various applications. In most cases, all necessary information can be provided to functions in the form of data: numbers, characters, strings, and more. However, sometimes, it is necessary to transfer functional dependence. For example, if you want to find the roots, derivative, integral of an arbitrary function, or draw the graph. To implement these algorithms, it is necessary to make an available calculation of a certain function in any part of the specified interval. In terms of programming, such algorithms require an access to program code that can be used any times depending on the logic of the algorithm. C and C++ provide this opportunity through so-called pointers to functions.

Pointer to a function is the address from which the compiled code of a function is stored, that is an address, to which control is passed when calling this function. Just as the name of an array is a constant pointer to the first element of the array, the function name is a constant pointer to a function. You can define a pointer variable, which points to a function, and you can call the function through this pointer.

A pointer to a function should indicate the function of the corresponding result type and signature. In the definition

```
int (*funcPtr) (double);
```

Змінна `funcPtr` оголошується як указівник, що вказує на функцію, яка отримує параметр з плаваючою точкою і повертає ціле значення. Дужки навколо `*funcPtr` є необхідними. Без перших дужок це буде оголошення функції, яка отримує **`double`** і повертає вказівник на int. Оголошення вказівника на функцію завжди буде містити тип значення, що повертається, і дужки із зазначенням типів параметрів, якщо такі є.

Можна присвоювати вказівнику на функцію адресу конкретної функції шляхом присвоєння імені функції без дужок. Указівник на функцію може бути використаний як ім'я фунції для її виклику. Тип указівника на функцію має бути погоджений з типом функції. Наприклад:

```
int round(double x)
{
  return x + 0.5;
}

void main()
{
  int (* funcPtr) (double);
  double z;
  cin >> z;
  funcPtr = round;
  cout << funcPtr(z);
}
```

Вказівник на функцію не повинен розіменовуватися, хоча це можна зробити. Таким чином, якщо `pFunc` вказівник на функцію і йому присвоїти відповідну функцію, можна викликати цю функцію як

```
pFunc(x);
```

або

```
(*pFunc)(x);
```

Обидві форми є ідентичними.

Оголошення **`typedef`**, можуть бути використані, щоб оголосити типи вказівників на функції:

```
typedef int (*FuncType)(int);
FuncType pf;
```

Можна створити масив указівників на функції. Вказівники на функції зазвичай використовують як тип аргументу функції.

Variable `funcPtr` is declared to be a pointer that points to a function that takes a floating point parameter and returns an integer value. The parentheses around `*funcPtr` are obligatory. Without the first pair of parentheses, this expression would declare a function that takes an argument of type **`double`** and returns a pointer to an int. The declaration of a function pointer will always include the return type and the parentheses indicating types of the parameters, if any.

You can assign a specific function to a pointer by assigning the function name without parentheses. Pointer to a function can be used as the function name for the call. The pointer to a function must agree in return type with the function. For example:

```
int round(double x)
{
  return x + 0.5;
}

void main()
{
  int (* funcPtr) (double);
  double z;
  cin >> z;
  funcPtr = round;
  cout << funcPtr(z);
}
```

Pointer to a function should not be dereferenced, although this can be done. Thus, if `pFunc` is pointer to a function and the address of an appropriate function is assigned to it, you can call this function as

```
pFunc(x);
```

or

```
(*pFunc)(x);
```

Both forms are identical.

The **`typedef`** definition can be used to declare the types of pointers to functions:

```
typedef int (*FuncType)(int);
FuncType pf;
```

You can create an array of pointers to functions. Pointers to functions commonly used in functions as parameters types.

3.1.6 Приклади програм

Приклад 1. Припустимо, необхідно створити невеличку бібліотеку математичних функцій, зокрема для обчислення другої та третьої степені.

Створюємо порожній проект і додаємо до нього заголовний файл `Funcs.h` та файл реалізації `Funcs.cpp`. Спочатку у заголовному файлі необхідно розмістити стражів включення:

```
#ifndef Funcs_h
#define Funcs_h

#endif
```

Тепер перед `#endif` можна додати прототипи функцій:

```
#ifndef Funcs_h
#define Funcs_h

double sqr(double x);
double cube(double x);

#endif
```

Файл `Funcs.cpp` містить визначення функцій, які розміщуються після директиви `#include "Funcs.h"`:

```
#include "Funcs.h"

double sqr(double x)
{
  return x * x;
}

double cube(double x)
{
    return x * x * x;
}
```

Тепер можна додати файл `main.cpp` для тестування функцій Необхідно додати включення заголовного файлу `Funcs.h`:

```
#include <iostream>
#include "Funcs.h"

using namespace std;
```

3.1.6 Examples of programs

Example 1. Suppose you want to create a small library of mathematical functions, in particular for the calculation of second and third power.

We start with creation of an empty project and then add the header file `Funcs.h` and the implementation file `Funcs.cpp` to it. First, include guards should be placed into the header file:

```
#ifndef Funcs_h
#define Funcs_h

#endif
```

Now we can add function prototypes before `#endif`:

```
#ifndef Funcs_h
#define Funcs_h

double sqr(double x);
double cube(double x);

#endif
```

The `Funcs.cpp` file contains function definitions, which are placed after the directive `#include "Funcs.h"`:

```
#include "Funcs.h"

double sqr(double x)
{
  return x * x;
}

double cube(double x)
{
    return x * x * x;
}
```

Now we can add a file `main.cpp` for testing. It is necessary to add the inclusion of a header file `Funcs.h`:

```
#include <iostream>
#include "Funcs.h"

using namespace std;
```

```
int main()
{
    double x, y;
    cout << "Enter argument:" << endl;
    cin >> x;
    y = sqr(x);
    cout << "Second power is " << y << endl;
    y = cube(x);
    cout << "Third power is " << y << endl;
    return 0;
}
```

Функції `sqr(x)` та `cube(x)` можна розмістити в окремому просторі імен. У цьому випадку заголовний файл матиме такий вигляд:

```
#ifndef Funcs_h
#define Funcs_h

namespace Funcs
{
    double sqr(double x);
    double cube(double x);
}

#endif
```

Файл реалізації:

```
#include "Funcs.h"

double Funcs::sqr(double x)
{
    return x * x;
}

double Funcs::cube(double x)
{
    return x * x * x;
}
```

Файл main.cpp:

```
#include <iostream>

#include "Funcs.h"
```

```
int main()
{
    double x, y;
    cout << "Enter argument:" << endl;
    cin >> x;
    y = sqr(x);
    cout << "Second power is " << y << endl;
    y = cube(x);
    cout << "Third power is " << y << endl;
    return 0;
}
```

The `sqr(x)` and `cube(x)` functions can be allocated in a separate namespace. In this case, the header file would look like this:

```
#ifndef Funcs_h
#define Funcs_h

namespace Funcs
{
    double sqr(double x);
    double cube(double x);
}

#endif
```

The implementation file:

```
#include "Funcs.h"

double Funcs::sqr(double x)
{
  return x * x;
}

double Funcs::cube(double x)
{
  return x * x * x;
}
```

The main.cpp file:

```
#include <iostream>

#include "Funcs.h"
```

```
using namespace std;
using namespace Funcs;

int main()
{
    double x, y;
    cout << "Enter argument:" << endl;
    cin >> x;
    y = sqr(x);
    cout << "Second power is " << y << endl;
    y = cube(x);
    cout << "Third power is " << y << endl;
    return 0;
}
```

Приклад 2. Наступна програма знаходить корінь рівняння з використанням методу дихотомії. Використання вказівника на функцію в якості аргументу дозволяє використовувати функцію `root()` для знаходження коренів довільних функцій. Єдине обмеження на використання методу дихотомії є те, що рівняння повинно мати рівно один корінь на заданому інтервалі.

Оскільки функція знаходження коренів може бути використана у різних програмах, доцільно створити окрему одиницю трансляції. У заголовному файлі `Dichotomy.h` міститься опис типу вказівника на функцію та прототип функції `root()`:

```
#ifndef Dichotomy_h
#define Dichotomy_h

typedef double (*FuncType)(double);

double root(FuncType f, double a, double b,
            double eps = 0.001);

#endif
```

У файлі реалізації міститься визначення функції `root()`. Слід пам’ятати, що значення параматрів за умовчанням вказуються тільки один раз у першому оголошенні функції.

```
#include "Dichotomy.h"

double root(FuncType f, double a, double b, double eps)
{
    double x;
```

```
using namespace std;
using namespace Funcs;

int main()
{
    double x, y;
    cout << "Enter argument:" << endl;
    cin >> x;
    y = sqr(x);
    cout << "Second power is " << y << endl;
    y = cube(x);
    cout << "Third power is " << y << endl;
    return 0;
}
```

Example 2. The following program finds the root of an equation using the dichotomy method. Using a pointer to a function as an argument allows the use of `root()` function for finding roots of arbitrary functions. The only restriction on the use of the dichotomy method is that the equation must have exactly one root in a given interval.

Since the function of finding roots can be used in various applications, it is advisable to create a separate translation unit. The header file `Dichotomy.h` will contain type definition for pointer to a function and prototype of `root()` function:

```
#ifndef Dichotomy_h
#define Dichotomy_h

typedef double (*FuncType)(double);
double root(FuncType f, double a, double b,
            double eps = 0.001);

#endif
```

The implementation file contains a definition of `root()` function. Keep in mind that the default values of parameters are specified only once in the first function declaration.

```
#include "Dichotomy.h"

double root(FuncType f, double a, double b, double eps)
{
      double x;
```

```
    do
    {
        x = (a + b) / 2;
        if (f(a) * f(x) > 0)
        {
            a = x;
        }
        else
        {
            b = x;
        }
    }
    while (b - a > eps);
    return x;
}
```

У функції `main()` (файл `main.cpp`) визначаємо функцію, корінь якої треба знайти, та передаємо її у функцію `root()`:

```
#include <iostream>
#include <cmath>
#include "Dichotomy.h"

using namespace std;

double g(double x)
{
    return x * x - 2;
}

void main()
{
    cout << root(g, 0, 6) << endl;
    cout << root(g, 0, 6, 0.00001) << endl;
    cout << root(sin, 1, 4) << endl;
    cout << root(sin, 1, 4, 0.00001) << endl;
}
```

У наведеній програмі тестування здійснюється як для функції `g()`, яка була визначена користувачем, так і для стандартної функції `sin()`.

3.1.7. Вправи для самостійної роботи

Вправа 1. Створіть заголовний файл та файл реалізації, у яких представлена функція для розв'язання квадратного рівняння. Функцію необхідно також розмістити в окремому просторі імен. В іншій одиниці

```
    do
    {
        x = (a + b) / 2;
        if (f(a) * f(x) > 0)
        {
            a = x;
        }
        else
        {
            b = x;
        }
    }
    while (b - a > eps);
    return x;
}
```

The `main()` function (file `main.cpp`) determines the function, the root of which must be found, and transfers it to the `root()` function:

```
#include <iostream>
#include <cmath>
#include "Dichotomy.h"

using namespace std;

double g(double x)
{
    return x * x - 2;
}

void main()
{
    cout << root(g, 0, 6) << endl;
    cout << root(g, 0, 6, 0.00001) << endl;
    cout << root(sin, 1, 4) << endl;
    cout << root(sin, 1, 4, 0.00001) << endl;
}
```

The following program tests are done for the function `g()`, which has been defined by a user, and for the standard function `sin()`.

3.1.7. Exercises for self-study

Exercise 1. Create a header file and implementation file in which the function for solving the quadratic equation is defined. The function must also be placed in a separate namespace. Another translation unit

трансляції слід здійснити тестування створеного модуля.

Вправа 2. Створити програму, яка реалізує повний перебір значень довільної функції та пошук мінімального значення. Необхідне значення може бути знайдено шляхом тестування проміжних значень функції. Для забезпечення можливості використання функції у різних задачах слід використати вказівник на функцію.

Вихідний код повинен бути розділений на дві одиниці трансляції. Перша одиниця трансляції буде представлена заголовним файлом та файлом реалізації. Визначення typedef та оголошення функції пошуку потрібного значення повинні бути поміщені у заголовний файл.

Визначення цієї функції здійснюватиметься у файлі реалізації. Функція, на якій здійснюватиметься тестування, а також функція main(), повинні бути розміщені в іншій одиниці трансляції.

Контрольні запитання

1. Поняття простору імен.
2. **using**-об'ява та using-директива.
3. Підключення простору імен.
4. Безіменні простори імен.
5. Використання заголовних файлів.
6. Правила розміщення описів у заголовних файлах.
7. Стражі включення.
8. Опис та використання вказівників на функції.

3.2 Типи даних, які визначає користувач

У реальних задачах інформація, яку потрібно обробляти, може мати досить складну структуру. Для того, щоб її адекватно представити використовуються складні типи даних, побудовані на основі простих типів даних, масивів і покажчиків. У вигляді таких типів даних виступає тип перерахування і структура.

Тип перерахування дозволяє задати зв'язану множину символічних цілих констант.

Структури використовуються у випадках, коли необхідно зберігати набір різнорідних, але логічно зв'язаних даних, що описують, наприклад, стан деякого об'єкту реального світу, коли використання масивів, що дозволяють поводитися з набором логічно зв'язаних однотипних елементів як з єдиним цілим, є недостатнім.

should provide testing of previously created module.

Exercise 2. Write a program that implements an exhaustive search of values of arbitrary functions and finds the minimum value. The required value can be found by testing the intermediate values of the function. To be able to use the function in different tasks, you should use a pointer to a function.

The source code should be divided into two translation units. The first translation unit will be presented by header file and implementation file. The typedef definition and declaration of a function that finds the desired value should be placed in the header file.

The definition of this function should be placed in the implementation file. The function used for testing, as well as `main()` function, should be placed in another translation unit.

Advancement Questions

1. The concept of namespaces.
2. **using** declaration and using directive.
3. Connecting to namespaces.
4. Anonymous namespaces.
5. The use of header files.
6. Rules of placing descriptions in header files.
7. Include Guards.
8. Description and use of pointers to functions.

3.2 User Defined Types

The information that you need to process in real problems can have a rather complicated structure. In order to represent it adequately, complex data types that are based on simple data types, arrays and pointers, are used. These are primarily enumeration types and structures.

Type enumeration lets you specify a linked set of symbolic integer constants.

The structures are used in cases where it is necessary to save a set of diverse but logically related data that describe, for example, the state of a real world object when using arrays that allow to deal with a set of logically identical elements as a single whole is not sufficient.

3.2.1 Перелічення

Тип перелічення визначає набір цілих констант. *Перелічення може бути визначене* за допомогою ключового слова **enum**, після якого у фігурних дужках розміщують список елементів, розділених комами. За умовчанням перша константа має значення `0`, наступна – `1` і т.д.

Можна створити *безіменні* та *іменовані перелічення*. *Безіменне перелічення* фактично є списком констант. Наприклад,

```
enum { red, green, blue };
```

Таке перелічення можна застосовувати замість визначення констант:

```
const int red   = 0;
const int green = 1;
const int blue  = 2;
```

Необхідні значення можна визначити явно. Значення наступних елементів обчислюють шляхом додавання одиниці:

```
enum { one = 1, three = 3, four, nine = 9, ten };
// four == 4, ten == 10
```

Іменоване перелічення визначає новий тип даних. Цей новий тип може бути використаний для визначення змінних:

```
enum Colors { red, green, blue };
Colors c = blue;
```

На відміну від опису **typedef** [5, 7, 9], типи даних, які визначає користувач, зокрема перелічення, не є синонімами існуючих типів. Наприклад, в нашому випадку не можна присвоїти цілі значення змінним типу `Colors`, тільки значення `red`, `green` та `blue`. І навпаки, присвоєння цілим змінним значень типу `Colors` є цілком коректним. Змінні таких типів можна використовувати у виразах.

```
Colors c;
c = 1;     // Помилка!
c = green; // OK
int i = c; // OK
i = c + 1; // OK
```

Для перелічень можна перевантажувати деякі операції, наприклад `++` та `--`. Основи та приклади перевантаження операцій будуть викладені нижче.

3.2.1 Enumerations

Enumeration type defines a set of integer constants. Enumeration can be defined using the **enum** keyword, followed by a list of items enclosed in braces and separated by commas. By default, the first constant is 0, the next is 1, etc.

There are named and unnamed enumerations. An *unnamed enumeration* is essentially a list of constants. For example,

```
enum { red, green, blue };
```

Such enumeration can be used instead of a set of constant definitions:

```
const int red   = 0;
const int green = 1;
const int blue  = 2;
```

You can set necessary value explicitly. Each successive constant is one larger than the value of the previous one:

```
enum { one = 1, three = 3, four, nine = 9, ten };
// four == 4, ten == 10
```

A *named enumeration* defines a new data type. This new type can be used for definition of variables:

```
enum Colors { red, green, blue };
Colors c = blue;
```

In contrast to **typedef** definition [5, 7, 9], user defined data type, in particular enumeration, is not a synonym (alias) of some existing type. You cannot assign any integer values to variables of `Colors` type, only previously defined constants `red`, `green`, and `blue`. Conversely, assigning values of type `Colors` to integer variables is quite correct. Variables of such types can be used in expressions.

```
Colors c;
c = 1;     // Error!
c = green; // OK
int i = c; // OK
i = c + 1; // OK
```

You can overload some operations for enumerations, such as `++` and `--`. Bases and examples of operator overloading will be considered later.

3.2.2. Структури

Поняття *структур* дозволяють формувати новий тип даних, поєднуючи кілька елементів даних різних типів. З такою групою можна працювати як з одним цілим. Після визначення типу структури можна описати відповідну змінну.

```
struct City
{
    char name[20];
    long population;
};                      // Крапка з комою обов'язкова

City Kharkiv;           // Kharkov - змінна типу City
```

Структури найчастіше визначаються у глобальній області видимості. Синтаксис мови C++ дозволяє створювати змінні безпосередньо після визначення типу.

```
struct City
{
    char name[20];
    long population;
} Kharkiv;              // Kharkov - змінна типу City
```

Однак таке визначення не слід вважати доцільним.

На відміну від масивів, операція присвоєння здійснює заелементне копіювання структури.

```
City someCity;
someCity = Kharkiv; // Заелементне копіювання
```

Для доступу до елементів структури використовують операцію ".". Спеціальний оператор -> використовують для того, щоб отримати доступ через указівник на структуру:

```
cout << someCity.name;
City *pc = new City;
pc->population = 10000000;
```

Якщо структура містить набір булевих елементів або цілих, які можуть набувати дуже невеликих значень, такі елементи можна описати як так звані *бітові поля*. *Опис бітових полів* складається з типу, імені, двокрапки та розміру поля у бітах. Наприклад:

3.2.2. Structures

The concept of *structure* allows you to create a new data type, combining several elements of different types. Such a group can be treated as a single entity. After the definition of type structure, you can create a corresponding variable.

```
struct City
{
    char name[20];
    long population;
};                      // The semicolon is obligatory

City Kharkiv;           // Kharkiv is a variable of City type
```

Structures are commonly defined in global scope. The C++ language syntax allows you to create variables of new type directly after its definition.

```
struct City
{
    char name[20];
    long population;
} Kharkiv;              // Kharkiv is a variable of City type
```

However, such definition is not recommended.

Unlike arrays, assignment operator accomplishes elementwise copying of one structure to another.

```
City someCity;
someCity = Kharkiv; // Elementwise copying
```

You can access particular data fields using dot operator (".").To access members of structures, to which some pointer points, special operator `->` is used:

```
cout << someCity.name;
City *pc = new City;
pc->population = 10000000;
```

If your structure contains a series of members of Boolean type or very small numbers, you can define the so-called *bit fields*. The declaration of a bit field contains the name of some integer type, followed by field name, followed by colon, followed by size of a bit field (in bits). For example:

```
struct Flags
{
    unsigned int logical : 1; // Один біт
    unsigned int tinyInt : 3; // Дуже мале ціле
};
```

Використання бітових полів дозволяє більш раціонально використовувати оперативну пам'ять. Але це має сенс, тільки якщо такі поля розташовані послідовно. Бітові поля також можна використовувати в описі класів.

На відміну від мови програмування С, структури у С++ можуть мати *функції-елементи*. Взагалі структури підтримують увесь синтаксис класів. Єдина відмінність полягає в тому, що за умовчанням елементи структури є відкритими, але це можна змінити за допомогою відповідних директив.

3.2.3 Початкові відомості про роботу з текстовими файлами

Робота з текстовими файлами аналогічна консольному введенню та виведенню. Для цього використовуються так звані файлові потоки.

Класи std::ifstream та std::ofstream обголошені у заголовному файлі fstream. Файловий потік повинен бути підключений до файлу перш ніж він може бути використаний. У наступному прикладі програма читає ціле число з файлу "data.txt" в змінну `k`. Це значення буде записано в інший файл:

```
#include <fstream>
. . .
int k;
std::ifstream inFile("data.txt");
inFile >> k;

std::ofstream outFile("result.txt");
outFile << k;
. . .
```

Якщо необхідно перевірити готовність потоку до читання даних з файлу, можна проаналізувати результат функції `eof()`. Ви можете також перетворити об'єкти потоку в цілі та у такий спосіб перевірити стан потоку. Другий підхід є більш ефективним.

```
struct Flags
{
  unsigned int logical : 1; // One bit
  unsigned int tinyInt : 3; // Tiny integer
};
```

The use of bit fields allows more efficient use of memory. However, this makes sense only if these fields are allocated consistently. You can also use bit fields in class definitions.

In contrast to C programming language, structures in C++ can contain *member functions* and support other features of classes. The only difference is that the members of the structure are public by default, but this can be changed using corresponding directives.

3.2.3 Basic Information about Working with Text Files

Working with text files is similar to the console input and output. For this purpose, the so-called file streams are used.

The std::ifstream and std::ofstream classes are defined in a header file called fstream. A file stream needs connecting to a file before it can be used. In the following example, program reads integer value from a file "data.txt" into variable `k`. This value will be written into another file:

```
#include <fstream>
. . .
int k;
std::ifstream inFile("data.txt");
inFile >> k;

std::ofstream outFile("result.txt");
outFile << k;
. . .
```

If you need to check the readiness of the stream to read data from a file, you can test the result of `eof()` function. You can also convert stream objects into integers and therefore test the stream state. The second approach is more efficient.

3.2.4. Приклади програм

Приклад 1. У наступній програмі створюється структура, яка представляє точку на екрані. У програмі обчислюється відстань поміж двома точками.

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

struct Point
{
    int x, y;
};

double sqr(double x)
{
    return x * x;
}

double distance(Point p1, Point p2)
{
    return std::sqrt(sqr(p1.x - p2.x) + sqr(p1.y - p2.y));
}

int main()
{
    Point p1 = {1, 2};
    Point p2 = {4, 6};
    std::cout << distance(p1, p2);
    return 0;
}
```

Примітка. Використання у цій програмі **`using namespace`** std спричиняє конфлікт імен.

Приклад 2. Припустимо, необхідно спроектувати програму, в якій описується структура для зберігання даних про країни світу: назва, територія, кількість населення. Дані про країни треба прочитати з текстового файлу, розташувати у масиві структур, а потім здійснити сортування масиву за алфавітом назв. Треба також вивести на екран дані про країни з густотою населення більше 100 мешканців на 1 кв.км.

Спочатку у будь-якому текстовому редакторі треба створити текстовий файл `Countries.txt` з даними про країни: назва, територія та

3.2.4. Examples of programs

Example 1. In the following example, we create structure that represents point on the screen. The program calculates distance between two points.

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

struct Point
{
  int x, y;
};

double sqr(double x)
{
  return x * x;
}

double distance(Point p1, Point p2)
{
  return std::sqrt(sqr(p1.x - p2.x) + sqr(p1.y - p2.y));
}

int main()
{
  Point p1 = {1, 2};
  Point p2 = {4, 6};
  std::cout << distance(p1, p2);
  return 0;
}
```

Note. Adding **using** namespace std to this program causes a name conflict.

Example 2. Suppose we want to design a program that defines the structure for storing data of countries: the name, the area, and the population. Data on countries should be read from a text file, saved in an array of structures, and sorted in alphabetical order of names. The display of data on the countries with population density of over 100 inhabitants per 1 sq.km is also needed.

First, with any text editor you need to create a text file `Countries.txt` with data on countries: name, area and population.

населення. Окремі дані можна розділити символами табуляції. Файл може, наприклад, мати такий вміст:

```
Ukraine 603700 46800000
France  544000 57804000
Sweden  450000 8745000
Germany 357000 81338000
```

Створюємо нову консольну програму. Обираємо ім'я проекту – `Countries`. Файл `Countries.txt` доцільно перенести у теку `Countries`, яка створена для нового проекту.

Опис структури для подання даних про країну винесемо у файл `Countries.h`, який доцільно попередньо створити. Вона повинна містити відповідні елементи даних – `name` (назва), `area` (територія) та `population` (населення):

```
struct Country
{
    char   name [20];
    double area;
    int    population;
};
```

Для перевірки умови виведення даних (густота населення більше 100 мешканців на 1 кв.км.) у файлі `Countries.cpp` корисною буде функція обчислення густоти населення (`density()`) з параметром типу посилання на константну структуру:

```
double density(const Country& c1)
{
    return c1.population / c1.area;
}
```

У нашому прикладі кількість країн фіксована. Тому можна описати константу та використовувати її в циклах і для опису масивів.

```
const int countriesCount = 4; // Кількість країн
```

Окрема функція здійснює читання даних з текстового файлу. Для роботи з файловими потоками необхідно підключити заголовний файл `fstream` та простір імен `std`. Окрім того, для виведення даних на екран необхідно підключити `iostream`. Цей файл доцільно використовувати замість `iostream.h` у сполученні з підключенням простору імен `std`.

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

Separate data items can be divided by tabs. For example, the file may have the following contents:

```
Ukraine 603700 46800000
France  544000 57804000
Sweden  450000 8745000
Germany 357000 81338000
```

We create a new console application and choose a project name: `Countries`. File `Countries.txt` is advisable to move to a folder `Countries`, which was created for a new project.

Definition of the structure for the representation of a country will be placed into the file `Countries.h` that it is advisable to create first. This structure should contain appropriate data elements: `name`, `area` and `population`:

```
struct Country
{
    char   name [20];
    double area;
    int    population;
};
```

In order to check output condition (population density over 100 inhabitants per 1 sq. km), within `Countries.cpp`, we can add a useful function of population density calculation (`density()`), with a parameter of type reference to constant structure:

```
double density(const Country& c1)
{
    return c1.population / c1.area;
}
```

In our example, the number of countries is fixed. Therefore, we can describe the constant and use it in loops and in array definitions.

```
const int countriesCount = 4; // Number of countries
```

A separate function performs reading data from a text file. To work with file streams, we need to include the `fstream` header file and namespace `std`. In addition, for output to the screen you need to include `iostream`.

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

Функція читання даних отримує такі параметри – вказівник на країну, який використовується як ім'я масиву країн, та ім'я вихідного файлу у вигляді вказівника на константний символ. У функції створюється об'єкт in типу `ifstream`. Створення цього об'єкта передбачає одночасне відкриття відповідного файлу для читання. Робота з об'єктом in подібна до використання стандартного потоку cin. Змінну типу `ifstream` можна порівнювати з нулем. Якщо таке порівняння дає значення **`true`**, операція читання не відбулася через відсутність необхідного файлу або його неправильний формат. Тому доцільно зробити так, щоб функція читання з файлу повертала логічне значення (**`bool`**), перевіряючи яке, можна зробити висновки про успішність читання даних:

```
bool readCountries(Country *c, const char* fileName)
{
    ifstream in(fileName);
    for (int i = 0; i < countriesCount; i++)
        if ((in >> c[i].name >> c[i].area >>
                   c[i].population) == 0)
            return false;
    return true;
}
```

Функція `printCountries()` здійснює виведення даних на екран:

```
void printCountries(const Country *c)

{
    for (int i = 0; i < countriesCount; i++)
    {
        cout << c[i].name << "  \t" << c[i].area << "  \t" <<
          c[i].population << "  \t" << density(c[i]) << endl;
    }
    cout << endl;
}
```

У функції `sortCountries()` здійснюється сортування масиву країн. Для цього використовується алгоритм сортування модифікованим методом бульбашки (вихідний варіант методу бульбашки наведено в 2.2.6). У циклі порівнюються назви сусідніх елементів масиву країн. Якщо перший з порівнюваних елементів менший, ніж другий, вони міняються місцями. Для обміну значень елементів звичайно використовується допоміжна змінна. Якщо після перегляду всіх пар елементів виявилося, що жодного з двох елементів не довелося міняти місцями, сортування завершується, у протилежному випадку повторюється послідовне порівняння назв. Назви порівнюють за допомогою функції `strcmp()`, оголошення якої міститься у заголовному файлі `string.h`.

Function that reads data gets these parameters: a pointer to the country to be used as the name of an array of countries, and the name of the source file as a pointer to a constant character. Within the function body, an object of type `ifstream` is created. The creation of this object provides simultaneous opening the file for reading. Working with such an object is similar to the usage of standard stream `cin`. The `ifstream` variable can be compared to zero. If the comparison returns **`true`**, the read operation fails due to lack of a specific file or wrong format. It is therefore advisable to return a Boolean value (**`bool`**), which you can use to make conclusions about the success of reading:

```
bool readCountries(Country *c, const char* fileName)
{
    ifstream in(fileName);
    for (int i = 0; i < countriesCount; i++)
        if ((in >> c[i].name >> c[i].area >>
                   c[i].population) == 0)
            return false;
    return true;
}
```

The `printCountries()` function provides output to the screen:

```
void printCountries(const Country *c)

{
    for (int i = 0; i < countriesCount; i++)
    {
        cout << c[i].name << "  \t" << c[i].area << "  \t" <<
          c[i].population << "  \t" << density(c[i]) << endl;
    }
    cout << endl;
}
```

The `sortCountries()` function implements sorting an array of countries. It uses a modified algorithm of a bubble sorting (initial version of this method is given in 2.2.6). The names of neighboring array elements are compared in a loop. If the first of the elements being compared is less than the second one, they will exchange. To exchange the values of elements an auxiliary variable is used. If after reviewing all pairs of elements it was found that none of the two elements has to swap, sorting is completed, otherwise the comparison of names is repeated. The names are compared using the `strcmp()` function, which is declared in a header file `string.h`.

Якщо функція `strcmp()` повернула додатне значення, країни необхідно поміняти місцями.

Для перевірки того, чи довелося змінювати місцями елементи, використовується спеціальна логічна змінна, значення якої змінюється, якщо хоча б раз довелося міняти місцями елементи масиву. Наприклад, якщо визначити змінну:

```
bool mustSort;
```

то функція сортування масиву країн за алфавітом назв буде виглядати так:

```
void sortCountries(Country *c)
{
    bool mustSort;
    do
    {
        mustSort = true;
        for (int i = 0; i < countriesCount - 1; i++)
        {
            if (strcmp(c[i].name, c[i + 1].name) > 0)
            {
                Country c1;
                c1 = c[i];
                c[i] = c[i + 1];
                c[i + 1] = c1;
                mustSort = false;
            }
        }
    }
    while (mustSort);
}
```

Для виведення у файл даних про країни з густотою населення більше 100 мешканців на 1 кв.км створюємо функцію `writeIfGreater()`. У цій функції використовується об'єкт `out` типу `ofstream` – файловий потік, з яким можна працювати як із стандартним потоком `cout`:

```
void writeIfGreater(const Country *c, const char* fileName)
{
    ofstream out(fileName);
    for (int i = 0; i < countriesCount; i++)
    {
        if (density(c[i]) > 100)
        {
            out << c[i].name << "  \t" << c[i].area
                << "  \t" << c[i].population << "  \t"
```

If the function `strcmp()` returns a positive value, the countries should be interchanged.

To check whether we have to change the position of an item, we use a special logical variable which value is changed, if at least once swapping of array elements was needed. For example, if you define a variable:

```
bool mustSort;
```

the function that sorts an array of countries in alphabetical order of names would look like this:

```
void sortCountries(Country *c)
{
    bool mustSort;
    do
    {
        mustSort = true;
        for (int i = 0; i < countriesCount - 1; i++)
        {
            if (strcmp(c[i].name, c[i + 1].name) > 0)
            {
                Country c1;
                c1 = c[i];
                c[i] = c[i + 1];
                c[i + 1] = c1;
                mustSort = false;
            }
        }
    }
    while (mustSort);
}
```

In order to implement file output for data on countries with population density of more than 100 inhabitants per square km, we create a function called `writeIfGreater()`. This function uses the object `out` of `ofstream` type – a file stream that supports operations similar to working with standard stream `cout`:

```
void writeIfGreater(const Country *c, const char* fileName)
{
    ofstream out(fileName);
    for (int i = 0; i < countriesCount; i++)
    {
        if (density(c[i]) > 100)
        {
            out << c[i].name << "  \t" << c[i].area <<
                << "  \t" << c[i].population << "  \t"
```

```
                    << density(c[i]) << endl;
            }
        }
    }
```

У функції `main()` необхідно послідовно викликати функції, які були реалізовані вище.

3.2.5. Вправи для самостійної роботи

Вправа 1. Напишіть програму, яка обчислює відстань між двома точками у тривимірному просторі.

Вправа 2. Створіть консольну програму, в якій треба описати структуру, що містить такі дані про студентів певної групи:

– номер залікової книжки (ціле значення);

– прізвище (масив символів);

– оцінки за останню сесію (масив з п'яти цілих значень).

З відповідно підготовленого файлу прочитати дані про студентів та розташувати їх у масиві структур. Здійснити сортування масиву за збільшенням середнього бала. Результат сортування вивести на екран. Відшукати в масиві дані про студентів, прізвище яких закінчується літерою "а". Вивести дані про цих студентів у новий файл. Для перевірки умов сортування та пошуку використовувати окремі функції.

Контрольні запитання

1. Використання перелічень.
2. Опис структур. Елементи даних.
3. Операції вибору елементів структур.
4. Передача параметрів-структур.
5. Опис бітових полів.
6. Робота з файловими потоками.

3.3. Загальні відомості про об’єктно-орієнтоване програмування

Об'єктно-орієнтоване програмування є одним з підходів до створення комп'ютерних програм, який покликаний усунути багато з проблем, програмування, що існують в традиційних методиках. Вид програмування, який розглядався вище, часто приводить до створення так званих “монолітних” програм, всі функції яких сконцентровані в

```
                    << density(c[i]) << endl;
            }
        }
    }
```

The `main()` function must contain sequential call of function that have been implemented above.

3.2.5. Exercises for self-study

Exercise 1. Write a program that calculates distance between two 3D-points.

Exercise 2. Create a console application, which should describe the structure that contains the following information about a particular group of students:

– record book number (integer);

– name (array of characters);

– previous end-of-semester exam grades (array of five integers.

Implement a program that reads data about students from previously prepared file to and stores them in an array of structures, sorts array in ascending order of the average grade, displays result of sorting, finds data about students, which last name ends with the letter "a", and stores data on these students to a new file. Use separate functions for checking the conditions and for sorting.

Advancement Questions

1. Using enumerations.
2. Definition of structures. Data members.
3. Operations of choice structure elements.
4. Passing structure type parameters.
5. Description of bit fields.
6. Working with file streams.

3.3. Basic Information about Object-Oriented Programming

Object-oriented programming is one of the approaches to the creation of software, which is designed to eliminate many programming problems that exist in traditional methods. The approach to programming that is seen above, often leads to the creation of so-called “monolithic” programs, where all

декількох модулях коду, а інколи і в одному. У об'єктно-орієнтованому програмуванні зазвичай використовується значно більше модулів, кожен з яких забезпечує конкретні функції і може бути ізольований або навіть повністю відокремлений від всіх інших. Таке модульне програмування, що базується на ключових парадигмах об'єктно-орієтірованного програмування, забезпечує набагато більшу гнучкість і можливості для багатократного використання коду.

3.3.1. Поняття класу

Класи у C++ є основним засобом визначення *користувацьких типів*. Класи аналогічні структурам. *Визначення класу* включає описи *елементів даних* та *функцій-елементів*.

Виділяють три рівні доступу до елементів класу: **public**, **protected** та **private**. Елементи класу, декларовані як private (закриті), можуть використовуватися лише у функціях-елементах цього класу, а також у його друзях (класах та функціях). Елементи, декларовані як protected (захищені), додатково можуть бути використані у похідних класах. Концепції та механізми створення похідних класів будуть розглянуті пізніше. Елементи, декларовані як public (публічні), можуть бути використані у будь-якій частині програми. Всередині опису класу ключові слова public, protected та private визначають заголовки відповідних розділів. Певні розділи додають до класу залежно від необхідності.

Розглянемо приклад визначення класу, який представляє країну. У нашому випадку країна буде характеризуватися такими даними: назва країни, населення та площа. Крім даних, визначимо дію, яку можна було би виконувати над цими даними, а саме, розрахунок густоти населення країни. Формалізуємо визначене вище:

```
class Country  //Country – ім’я класу
{
public: //розділ, в якому описують відкриті елементи класу
    char name[40];//елемент даних, пов’язаний з назвою країни
    double area;  //елемент даних, пов’язаний з площею країни
    int population;  //елемент даних, пов’язаний з населенням
    double density();//функція-елемент для розрахунку густоти
                     //населення
};
```

Функції-елементи, оголошені всередині тіла класу, можуть бути реалізовані всередині (в тілі) класу або поза класом.

functions are concentrated in a few modules of code, and sometimes in a single module. Object-oriented programming usually requires a much larger number of modules, each of which provides a specific function and can be isolated or even completely separate from all others. This modular programming based on the key paradigms of object-oriented programming provides much more flexibility and opportunities to reuse code.

3.3.1. The Concept of Class

Classes in C++ programming language provide the main way for creation of user defined data types. Class is similar to structure. The definition of class can contain *data members* and *member functions*.

There are three levels of access to the class members: **`public`**, protected and private. Class member declared as private may be accessed only from member functions of this class, as well as his friends (classes and functions). Items declared as protected in addition can be used in derived classes. Concepts and mechanisms of derived class creation will be considered later. Items declared as public can be used in any part of the program. Inside the class description, keywords public, protected and private are used as headers of the relevant sections. Certain sections are added to the class, depending on real needs.

Consider the example of a class definition that represents the country. In our case, the country will be characterized by the following data: name of the country, population, and area. In addition to these, we can define the action that could be performed on these data, namely, the calculation of population density. This can be formalized as follows:

```
class Country  // Country is the name of a class
{
public: // a section, which contains public class elements
    char name[40];   // data member associated with
                     // the country name
    double area      // data member associated with
                     // the country area
    int population;  // data member associated with
                     // the population
    double density();// member function for calculating
                     //population density
};
```

Member functions declared in class body can be defined as in the class body or outside the class body.

Для того щоб визначити приналежність функції до класу, використовується операція дозволу області видимості (::). Наведемо реалізацію функції density():

```
double Country::density()
{
    return population / area;
}
```

Як видно з прикладу, функція-елемент має вільний доступ до елементів даних.

Функції-елементи можна повністю визначити всередині класу. Тоді вони є неявними **inline**-функціями. Можна так змінити визначення класу:

```
class Country
{
public:
    char   name[40];
    double area;
    int    population;
    double density() { return population / area; }
};
```

Тепер можна використовувати ім'я класу для опису об'єкта та виклику функції-елемента. Для доступу до елементів класу через об’єкт класу, що є змінною, використовують *операцію крапки* (.).Для доступу до елементів класу через об’єкт класу, що є вказівником, використовують *операцію вибору елемента за вказівником* (->):

```
void main()
{
// Створення об’єкта типу Country
    Country c;
// визначення даних про країну:
    c.name="France";
    c.area=551500;
    c.population=57981000;
    cout << c.density(); //в результаті отримаємо 105.133
}
```

3.3.2. Конструктори та деструктори. Функції доступу

Спеціальні функції-елементи, які називаються *конструкторами*, призначені для ініціалізації даних об’єкта. Кожного разу, коли об'єкт

The belonging to particular class can be denoted using scope resolution operator (`::`). Here is an implementation of `density()` function:

```
double Country::density()
{
  return population / area;
}
```

As you can see from the example, the member function has free access to data members.

Member functions can be fully defined inside of the class. These functions are implicit **`inline`** functions. The class definition can be modified in a such way:

```
class Country
{
public:
    char   name[40];
    double area;
    int    population;
    double density() { return population / area; }
};
```

Now you can use class name for definition of objects and with further invocation of member functions. To access the class members via variable of class type, use the *dot operator* (`.`). To access class members via pointer to object, use the *operator of element selection by pointer* (`->`):

```
void main()
{
// Create an object of type Country
    Country c;
// definition of data on the country:
    c.name="France";
    c.area=551500;
    c.population=57981000;
    cout << c.density(); // got 105.133
}
```

3.3.2. Constructors and Destructors. Access Functions

Special member functions called constructors are intended to initialize object data. Every time you create new object in global scope, in stack, or in

створюється у глобальній області видимості, у стеку або в динамічній пам'яті, конструктор викликається автоматично. Ім'я конструктора повинно збігатись з іменем класу. Можна описати будь-яку кількість конструкторів класу. Вони повинні відрізнятися кількістю або типами параметрів.

```
class X
{
private:
    int j;
public:
    X();       // Конструктор без параметрів
    X(int i); // Конструктор з параметрами
};
```

Конструктори можна реалізувати поза тілом класу:

```
X::X()
{
    j = 0;     // Ініціалізація елемента даних
}

X::X(int i)
{
    j = i;     // Ініціалізація елемента даних
}
```

Це можна зробити також всередині класу:

```
class X
{
private:
    int j;
public:
    X()       { j = 0; }
    X(int i) { j = i; }
};
```

Найпростіший конструктор – це *конструктор за умовчанням*. Він не має параметрів. Якщо конструктор без параметрів не визначений явно, компілятор створює такий конструктор автоматично. Такий конструктор здійснює ініціалізацію елементів даних значеннями за умовчанням (нулями). Якщо в класі визначений хоча б один явний конструктор, компілятор не створює автоматично конструктора за умовчанням.

Можна здійснювати ініціалізацію об'єкта іншим об'єктом того ж типу. В цьому випадку викликається спеціальний конструктор – так званий *конструктор копіювання*. Такий конструктор, якщо його не

free store, constructor is called automatically. The name of constructor must coincide with class name. You can declare any number of class constructors. They should be different in number or in types of the parameters.

```
class X
{
private:
    int j;
public:
    X();       // Constructor without arguments
    X(int i); // Constructor with integer argument
};
```

Constructors can be implemented outside a class body:

```
X::X()
{
  j = 0;    // Initialization of a data member
}

X::X(int i)
{
  j = i;    // Initialization of a data member
}
```

You can also do this within class body:

```
class X
{
private:
  int j;
public:
  X()       { j = 0; }
  X(int i) { j = i; }
};
```

The simplest constructor is *default constructor*. It has no arguments. If you don't declare a constructor, default constructor is created automatically by compiler. Such constructor initializes all data member with default values (zeros). The only explicit constructor prevents the automatic creation of default constructor.

You can initialize an object by another object of the same type. In this case, a special *copy constructor* is invoked. This constructor if it is not

перекрили, створюється автоматично. Для явного створення конструктора копіювання класу X слід додати такий опис:

```
class X
{
  . . .
public:
  . . .
    X(const X&);
    . . .
};
```

Деструктор – це спеціальна функція-елемент, яка має ім'я ~Ім'яКласу та автоматично викликається перед тим, як об'єкт має бути знищений. Деструктор не може мати параметрів:

```
class X
{
  . . .
public:
    . . .
    ~X()     { /* Тіло деструктора */ }
    . . .
};
```

Деструктори також можуть бути реалізовані за межами тіла класу:

```
class X
{
    . . .
public:
    . . .
    ~X(); // Оголошення деструктора
    . . .
};

X::~X()
{
    /* Тіло деструктора */
}
```

Об'єкти як параметри передаються до функцій за значенням. При цьому здійснюється виклик конструктора копіювання.

```
MyClass f(MyClass t) // Виклик конструктора копіювання
```

overridden it is created automatically. To create a copy constructor of class X explicitly, add the following description:

```
class X
{
  . . .
public:
  . . .
  X(const X&);
  . . .
};
```

Destructor is a special member function with the name `~ClassName` and which is automatically called before the object is be destroyed. Destructor cannot have arguments:

```
class X
{
  . . .
public:
  . . .
  ~X()      { /* Destructor body */ }
  . . .
};
```

Destructors can also be implemented outside the class body:

```
class X
{
  . . .
public:
  . . .
  ~X();// Destructor declaration
  . . .
};

X::~X()
{
  /* Destructor body */
}
```

Objects as parameters are passed to functions by value. Thus the copy constructor is called.

```
MyClass f(MyClass t) // Call of copy constructor
```

```
{
    return t;
}                       // Виклик деструктора
```

Для того щоб уникнути копіювання об'єкта та виклику відповідного конструктора, аргумент та тип резуьтату доцільно описати як посилання.

```
MyClass& f(MyClass &t) // Конструктор не викликається
{
  return t;
}                        // Деструктор не викликається
```

Є спеціальний різновид функцій-елементів – так звані *константні функції-елементи*. Такі функції не можуть змінити даних об'єкта, для якого вони викликані. Для того щоб описати таку функцію, після списку аргументів слід розмістити модифікатор **const**. Константні функції-елементи можна викликати тільки для константних об'єктів.

В одному класі можна описати дві функції з однаковими іменами та параметрами, які відрізняються лише наявністю модифікатора **const**.

Специфікатор **mutable** перед типом елемента-даних застосовують для того, щоб цей елемент можна було змінювати з константних функцій-елементів.

Функції-елементи мають доступ до інших елементів класу через так званий вказівник **this**. Функції-елементи отримують цей вказівник як неявний аргумент. Цей вказівник указує на об'єкт, для якого викликана функція-елемент. Найчастіше вказівник this застосовують для того, щоб уникнути конфліктів імен:

```
class X
{
    int value;
public:
    X(int value) { this->value = value; }
};
```

Якщо функція-елемент має модифікатор **const**, вона одержує вказівник this, який вказує на константний об'єкт.

Інкапсуляція (приховування даних) – одна з трьох парадигм об'єктно-орієнтованого програмування. Зміст інкапсуляції полягає у приховуванні від зовнішнього користувача деталей реалізації об'єкта. Зокрема, доступ до елементв даних, які зазвичай описані з модифікатором **private**, здійснюється через відкриті функції доступу. Як правило, це так звані сеттери та геттери. Якщо поле має ім’я `name`, відповідні функції доступу

```
{
  return t;
}                     // Call of destructor
```

To prevent object copying with call of copy constructor, you can declare arguments of reference types. You can also return reference.

```
MyClass& f(MyClass &t) // No constructor call
{
    return t;
}                        // No destructor call
```

There is a special kind of member functions: the so-called *constant member functions*. These functions cannot change the data of the object for which they are called. In order to declare this function, you should place **const** modifier after argument list. Constant member functions can be invoked for constant objects.

The single class can contain functions with the same name and parameter lists that differ only in the presence of the **const** modifier.

The **mutable** specifier applied to a data member allows modifying this data member within constant member functions.

Member functions get access to other elements of the class through a pointer called **this**. Member functions obtain this pointer as an implicit argument. This pointer points to the object for which member function is called. Often, this pointer is used to avoid name conflicts:

```
class X
{
    int value;
public:
    X(int value) { this->value = value; }
};
```

If some member function is prefixed with **const** modifier, it obtains this pointer as a constant pointer.

Encapsulation (hiding data) is one of three paradigms of object-oriented programming. The matter of encapsulation is to hide implementation details of the object from external access. In particular, access to data members that are usually described with **private** modifier private, is carried out through public access functions. Typically, these are so-called *setters* and *getters*. If some data member has a name `name` the corresponding access functions are named

мають імена `setName` та `getName`. Геттери зазвичай оголошують як константні функції-елементи. У сеттерах для уникнення конфліктів імен застосовують указівник **`this`**. Наприклад,

```
class Number
{
private:
    int value;
public:
    int getValue() const { return value; }
    void setValue(int value) { this->value = value; }
};
```

3.3.3. Область видимості класу

Клас визначає свою область видимості – *класову область видимості*. Усі елементи класу входять до його області видимості. Іноді до елементів класової області видимості можна звертатись за допомогою оператора `::`, який застосовується до імені класу. Наприклад, можна використовувати цей оператор замість указівника **`this`**:

```
class X
{
    int i;
public:
    void f() { int i; i = X::i; }
};
```

Можна також описати в тілі класу синонім типу за допомогою **`typedef`**

```
:
class Z
{
public:
  typedef double real;
  void f() { real r = 1; cout << r; }
};

Z::real x = 0;
```

Клас може бути описаний у глобальній області видимості, а також у класовій та локальній областях. Класи, які визначені всередині інших класів, мають назву внутрішніх. Об'єкт внутрішнього класу не створюється автоматично у зовнішньому класі. Такий об'єкт необхідно створювати окремо:

`setName` and `getName`. Getters are usually declared as constant member functions. In order to avoid name conflicts, setters use **`this`** pointer. For example,

```
class Number
{
private:
  int value;
public:
  int getValue() const { return value; }
  void setValue(int value) { this->value = value; }
};
```

3.3.3. Class Scope

Class defines its own scope – the *class scope*. All class members are included in its scope. Sometimes, class members can be accessed using the `::` operator, which is applied to the class name. For example, you can use this operator instead of **`this`** pointer:

```
class X
{
    int i;
public:
    void f() { int i; i = X::i; }
};
```

Within class body, you can also define an alias to some type using **`typedef`**:

```
class Z
{
public:
    typedef double real;
    void f() { real r = 1; cout << r; }
};

Z::real x = 0;
```

Class itself can be defined in a global scope, in a class scope, or in a local scope. Classes defined inside other classes are called *inner* (or *nested*) classes. The instance of inner class is not created automatically as a part of outer object. This object should be created explicitly:

```
class X
{
public:
    class Y
    {
    public:
        int t;
    };
private:
    class Z
    {
    public:
        int w;
    };
    Z z;    // Об'єкт внутрішнього класу
};


X::Y y;   // OK
X::Z z;   // Помилка! Немає доступу до Z
```

3.3.4. Статичні елементи класу

Іноді необхідно створювати змінні, які логічно мають відношення до певного класу, але їх недоцільно робити елементами даних об'єктів, бо вони повинні бути спільними для всіх об'єктів. *Статичні елементи даних* пропонують спосіб розміщення глобальних змінних в області видимості класу. Звертатись до таких елементів можна як через імена об'єктів, так і імена класів:

```
class X
{
    . . .
public:
    static int i;
};
. . .


X::i = 10; // Через ім'я классу

X x;
x.i  = 11; // Через ім'я об'єкта
```

Як видно з наведеного прикладу, статичні елементи не потребують створення об'єкта.

```
class X
{
public:
    class Y
    {
    public:
        int t;
    };
private:
    class Z
    {
    public:
        int w;
    };
    Z z;    // Об'єкт внутрішнього класу
};


X::Y y;   // OK
X::Z z;   // Error: Z not accessible
```

3.3.4. Static Class Members

Sometimes we need to create some variables which logically belong to some class, but they are not appropriate to become the elements of its objects, as they should be common to all objects. *Static data members* offer a way to organize the global variables in the class scope. You can access these elements as through the object names and through class names:

```
class X
{
    . . .
public:
    static int i;
};
. . .


X::i = 10; // Using class name
X x;
x.i  = 11; // Using object name
```

The previous example shows that working with static members does not require creation of objects.

Статичні елементи даних даних не можуть бути описані з модифікатором **mutable**.

Існують також *статичні функції-елементи*. Вони не отримують указівника **this**, і тому не мають доступу до нестатичних елементів об'єкта, для якого вони викликані.

Статичні елементи даних повинні бути визначені в глобальній області видимості класу.

```
class X
{
    static int x;
    static const int size = 5;
    class Inner             // Внутрішній клас
    {
        static float f;
        void func();
    };
public:
    char array[size];
};

int X::x = 1;       // Ініціалізація статичного елемента
float X::Inner::f = 3.14; // Ініціалізація статичного
                          // елемента
void X::Inner::func()
{
    // . . .
}
```

Можна оголошувати *статичні константи*:

```
class X
{
public:
    static int count;
};

int X::count = 100;
```

Локальні класи не можуть мати статичних елементів.

3.3.5. Друзі класу

Функції або класи, оголошені як *друзі класу*, мають доступ до його закритих та захищених елементів. Друзі не є частиною області видимості класу та не отримують указівника **this**.

Static data members cannot be declared with **`mutable`** modifier.

There are also *static member functions*. They do not get **`this`** pointer, and therefore they have no access to non-static members of object for which they called.

Static data members declared in a class body must be defined in a global scope.

```
class X
{
    static int x;
    static const int size = 5;
    class Inner             // Inner class
    {
        static float f;
        void func();
    };
public:
    char array[size];
};

int X::x = 1;              // Initialization of static
                           // data member
float X::Inner::f = 3.14; // Initialization of static
                           // data member
void X::Inner::func()
{
    // . . .
}
```

You can declare *static constants*:

```
class X
{
public:
  static int count;
};

int X::count = 100;
```

Local classes cannot have static members.

3.3.5. Friends of a Class

Functions or classes declared as *friends of a class* have access to its private and protected members. Friends are not a part of the class scope and do not get **`this`** pointer.

```
class SomeClass
{
    friend void f(SomeClass &sc);
private:
    int i;
};

void f(SomeClass &sc)
{
    cout << sc.i; // Доступ до закритого елемента даних
}

void main()
{
    SomeClass s1;
    f(s1);
}
```

Дружні функції можуть бути реалізовані в тілі класу. Такі функції мають неявний модифікатор **inline**. Друзі можуть бути оголошені у будь-якій частині класу (public, private або protected). Це не впливає на механізм доступу.

Можна здійснити оголошення класу без визначення. Завдяки цьому можна декларувати взаємну дружбу:

```
class Y;          // оголошення класу

class X
{
    friend Y;
    . . .
};

class Y;
{                 // визначення класу
    friend X;
    . . .
};
```

Функції-елементи класу `X` можуть бути оголошені як друзі класу `Y`:

```
class X
{
    . . .
    void member_funcX();
};
```

```
class SomeClass
{
    friend void f(SomeClass &sc);
private:
    int i;
};

void f(SomeClass &sc)
{
    cout << sc.i; // Access to private data member
}

void main()
{
    SomeClass s1;
    f(s1);
}
```

Friend functions can be implemented in a class body. These functions have an implicit modifier **`inline`**. Friends can be declared in any part of the class (public, private or protected). This does not affect the access mechanism.

You can declare some class without definition. This allows you to declare a mutual friendship:

```
class Y;          // оголошення класу

class X
{
    friend Y;
    . . .
};

class Y;
{                 // визначення класу
    friend X;
    . . .
};
```

Member functions of class `X` can be declared as friends of class `Y`:

```
class X
{
  . . .
    void member_funcX();
};
```

```
class Y
{
    friend void X::member_funcX();
    . . .
};
```

Якщо X – друг класу Y, Y – друг класу Z, X не є автоматично другом Z. Дружба не успадковується.

3.3.6. Композиція класів

Об'єкти класів можна робити елементами даних інших класів. Цей процес має назву *композиція класів*.

Конструктори класів, об'єкти яких розміщені у зовнішньому класі, викликаються до виконання конструктора зовнішнього класу. Виклик конструкторів без параметрів здійснюється автоматично. Конструктори завжди викликаються у порядку, в якому об'єкти оголошені в класі. Деструктори викликаються у зворотному порядку.

Іноді для внутрішніх об'єктів необхідно викликати конструктори з параметрами. Це можна зробити з використанням так званого *списку ініціалізації*. Цей список розміщується перед тілом конструктора. Наприклад:

```
class X
{
public:
    X(int j) { ... }
    . . .
};

class Y
{
  X x;
public:
Y(int k) : x(k)//приклад списку ініціалізації
{ ... }
  . . .
};
```

За допомогою списку ініціалізації можна також встановлювати початкові значення для елементів вбудованих типів:

```
class X
{
```

```
class Y
{
    friend void X::member_funcX();
    . . .
};
```

If X is a friend of class Y, and Y is a friend of class Z, X is not automatically a friend of Z. Friendship is not inherited.

3.3.6. Class Composition

You can place objects of class types into other classes as its data members. This is called *class composition*.

Constructors of data members are invoked before the host class constructor. Constructors without parameters are called automatically. Constructors are called in the order of data member declaration. Destructors are invoked in the reverse order.

Sometimes we need to invoke non-default constructors of class members. Arguments for such constructors can be passed using an *initialization list*. An initialization list is allocated prior to the function body. For example:

```
class X
{
public:
    X(int j) { ... }
    . . .
};

class Y
{
  X x;
public:
Y(int k) : x(k)//Example of initialization list
{ ... }
  . . .
};
```

An initialization list can also used for setting initial values for the members of built-in types:

```
class X
{
```

```
    int k;
    double d;
public:
    X(int j, double h) : k(j), d(h) { ... }
    . . .
};
```

3.3.7. Успадкування

Успадкування застосовують для створення класу, похідного від певного базового класу. Похідні класи успадковують елементи базових класів, а також можуть додавати нові елементи. Синтаксично успадкування визначається списком базових класів після імені класу і двокрапки. Перед іменем базового класу вказується тип успадкування (**`public`** або інше):

```
class X             // Базовий клас
{
 ...
};

class Y : public X // Похідний клас
{
 ...
};
```

Різницю між різними варіантами успадкування наведемо нижче:

Режим доступу до елемента в базовому класі	Режим доступу при успадкуванні класу	Режим доступу до елемента у похідному класі
private	public	недоступний
protected		protected
public		public
private	protected	недоступний
protected		protected
public		protected
private	private	недоступний
protected		private
public		private

Розглянемо більш детально зміст варіантів успадкування.

```
    int k;
    double d;
public:
    X(int j, double h) : k(j), d(h) { ... }
    . . .
};
```

3.3.7. Inheritance

Inheritance is used to create a classes derived from a certain base class. Derived classes inherit the base class members and may add new members. The syntax for inheritance is defined by a list of base classes after the class name and colon. Before the name of the base class, the inheritance mode can be specified (**`public`** etc):

```
class X             // Base class
{
  . . .
};

class Y : public X // Derived class
{
  . . .
};
```

The difference between the modes of the inheritance is given below:

Access to the member in the base class	Access by inheritance	Access to the member in the derived class
private	public	inaccessible
protected		protected
public		public
private	protected	inaccessible
protected		protected
public		protected
private	private	inaccessible
protected		private
public		private

Consider the content of inheritance modes in more details.

При використанні режиму доступу **public** при успадкуванні класу всі публічні та захищені елементи базового класу залишаються публічними та захищеними елементами похідного класу. Виняток становлять елементи, які в базовому класі були визначені з режимом доступу private. Такі елементи в похідному класі стають недоступними.

Режими доступу **protected** та private при успадкуванні класу змінюють режими доступу до елементів у похідному класі, а саме: у режимі доступу protected всі публічні та захищені елементи базового класу стають захищеними елементами похідного класу. Закриті елементи базового класу в похідному класі стають недоступними.

Якщо режим доступу при успадкуванні – **private**, то всі публічні та захищені елементи базового класу стають закритими елементами похідного класу, всі закриті елементи базового класу стають недоступними елементами похідного класу.

Мова C++ підтримує *одиночне* та *множинне успадкування*. Найбільш розповсюдженим є одиночне успадкування, коли клас походить від одного базового класу. У випадку використання множинного успадкування клас може походити від кількох базових класів. Наприклад:

```
class X             // Базовий клас
{
    . . .
};

class Z             // Базовий клас
{
    . . .
};

class Y : public X, public Z // Похідний клас
{
    . . .
};
```

Функції-елементи похідного класу мають доступ до елементів базових класів відповідно таблиці 4.1, причому захищені елементи призначені саме для використання в похідних класах.

Похідні класи успадковують усі елементи базового класу, окрім конструкторів, деструкторів та перевантаженої операції присвоєння.

Конструктори базових класів виконуються до виконання конструкторів об'єктів, які є елементами даних похідного класу, а також конструктора похідного класу. Деструктори виконуються у зворотному порядку.

When using inheritance with **public** modifier, all the public and protected members of the base class remain public and protected members of the derived class. However, private members of the base class are not available in a derived class.

Inheritance modes **protected** and private modify access levels of derived class members, namely in protected mode public and protected members of the base class become protected members of the derived class. Private members of the base class are not available in a derived class.

If the inheritance mode is **private**, then all public and protected members of the base class become private members of the derived class, all private members of the base class are unavailable in a derived class.

C++ supports single and multiple inheritance. The most common is a *single inheritance*, where a class is derived from one base class. When using *multiple inheritance*, class can be derived from multiple base classes. For example:

```
class X             // The base class
{
  . . .
};

class Z             // The base class
{
  . . .
};

class Y : public X, public Z // Derived class
{
  . . .
};
```

Member functions of a derived class can access the elements of base classes according to the table; protected members are in particular intended specifically for the use in derived classes.

Derived classes inherit all the members of the base class except constructors, destructors, and overloaded assignment operation.

The base class constructors are invoked before constructors of the derived class data members and the constructor of the derived class itself. Destructors are invoked in the reverse order.

Якщо конструктори базових класів потребують передачі аргументів, це здійснюється за допомогою списку ініціалізації:

```
class X
{
public:
  X(int j) { ... }
  . . .
};

class Y : public X
{
public:
  Y(int k) : X(k) { ... }
  . . .
};
```

3.3.8. Поліморфізм

Іноді є необхідним розміщення в одному масиві вказівників на різні об'єкти, причому виконання певної дії для різних елементів повинно обумовлювати різну реакцію. Один з підходів базується на використанні вказівників на функції, а також створення масивів указівників типу **void*** [5, 7, 9]. Це не зовсім безпечний шлях.

Мова C++ пропонує альтернативний спосіб розв’язання цієї проблеми. Цей спосіб базується на тому, що у C++ значення вказівників на похідні класи можна присвоювати змінним типу вказівників на базовий клас. Аналогічне правило є для посилань:

```
class X
{
  . . .
};

class Y : public X
{
  . . .
};

class Z : public Y
{
  . . .
};

X *px = new Y();
Z  z;
```

If the base class constructor requires transferring arguments, this is done via the initialization list:

```
class X
{
public:
  X(int j) { ... }
  . . .
};

class Y : public X
{
public:
  Y(int k) : X(k) { ... }
  . . .
};
```

3.3.8. Polymorphism

Sometimes it is necessary to allocate pointers to different objects in one array, applying an action for the various elements and getting different reactions. One approach is based on the use of function pointers, and creating arrays pointer of type **void** * [5, 7, 9]. It is not quite a safe way.

The C++ programming language provides alternative way of solving this problem: we can assign the pointers to derived objects to the pointers of base type objects. The rules of reference initialization are the same:

```
class X
{
  . . .
};

class Y : public X
{
  . . .
};

class Z : public Y
{
  . . .
};

X *px = new Y();
Z  z;
```

```
X &rx = z;
```

Для інших перетворень типів вказівників і посилань на об'єкти однієї ієрархії вживають спеціальний оператор **dynamic_cast**<тип>(вираз).

Тепер можна створити масив указівників на базовий клас та здійснити ініціалізацію елементів указівниками на похідні класи:

```
X *a[3];
x[0] = new X();
x[1] = new Y();
x[2] = new Z();
```

Точні типи об'єктів у масиві вказівників, а також адреси функцій-елементів цих об'єктів не можуть бути визначені компілятором. Спеціальний механізм – *поліморфізм* – дозволяє процесору отримати адреси функцій через указівники на об'єкти.

Поліморфізм C++ у реалізується через так звані віртуальні функції. *Віртуальні функції* – це функції-елементи, адреса яких може визначатись динамічно під час виконання програми. Це функції з фіксованим інтерфейсом виклику та різними реалізаціями у різних похідних класах. У визначенні класу для ідентифікації віртуальних функцій вживаєтья слово **virtual**:

```
class SomeClass
{
    . . .
public:
    virtual void firstFunc();
    virtual int secondFunc(int x);
};
```

Слово **virtual** не вживається, якщо функція визначається за межами класу:

```
void SomeClass::firstFunc()
{
    . . . // Реалізація
}
```

Похідні класи можуть перекривати всі або частину віртуальних функцій базового класу. Функція, яка перекриває віртуальну функцію, повинна точно повторювати її заголовок. Слово **virtual** для такої функції не є обов'язковим, але його застосування рекомендується. Розглянемо приклад:

```
X &rx = z;
```

For other type conversions of pointers and references to objects of common hierarchy, a special operator **dynamic_cast**<type>(expression) is used.

Now you can create an array of pointers to a base class and initialize items with pointers to derived classes:

```
X *a[3];
x[0] = new X();
x[1] = new Y();
x[2] = new Z();
```

The exact types of objects in the array of pointers and addresses of member functions of these objects cannot be determined by the compiler. A special mechanism, so-called *polymorphism*, allows the processor to get addresses of functions through pointers to objects.

Polymorphism in C++ is implemented through the so-called virtual function. *Virtual function* is a member function, which address can be determined dynamically at runtime. These are functions with a fixed calling interface, and different implementations in derived classes. The **virtual** keyword is used to notify compiler that declared method is a virtual function:

```
class SomeClass
{
    . . .
public:
    virtual void firstFunc();
    virtual int secondFunc(int x);
};
```

We do not use **virtual** keyword in definition of a function outside the class:

```
void SomeClass::firstFunc()
{
    . . . // Implementation
}
```

Derived classes can override all virtual methods or any part of them. Overridden functions must repeat declarations of base functions using same resulting types and argument list. The **virtual** keyword is not obligatory, but its use is recommended. Consider an example:

```
class FirstDescendant public SomeClass
{
    . . .
public:
    virtual void firstFunc();
    virtual int secondFunc(int x);
};

class SecondDescendant : public SomeClass
{
    . . .
public:
    void firstFunc();
    virtual double thirdFunc(double x, double y);
};

class ThirdDescendant : public SecondDescendant
{
    . . .
public:
    virtual double thirdFunc(double x, double y);
};
```

Клас `SecondDescendant` не перекриває функції `secondFunc()`. Об'єкти цього класу використовують функцію `secondFunc()`, яка визначена у базовому класі. Клас `ThirdDescendant` використовує `firstFunc()` класу `SecondDescendant`. Він використовує функцію `secondFunc()` класу `SomeClass` та перекриває `thirdFunc()`.

Є три варіанти виклику функцій-елементів: для об'єктів, для вказівників та для посилань. Виклик віртуальної функції для об'єкта визначається під час компіляції. В інших випадках адреса конкретної функції визначається під час виконання програми.

Об'єкт класу, у якому є віртуальні функції, має назву "поліморфний об'єкт". Кожний поліморфний об'єкт містить спеціальне поле, в яке конструктор записує адресу так званої таблиці віртуальних методів (Virtual Method Table, VMT). Така таблиця створюється для всіх класів з віртуальними функціями. Таблиця віртуальних методів містить адреси віртуальних функцій класу. Ці адреси використовуються для виклику віртуальних функцій.

Деструктори також можуть бути віртуальними. Після оголошення деструктора віртуальним деструктори всіх похідних класів є віртуальними автоматично. Віртуальні деструктори дозволяють точно визначити обсяг пам'яті, яку необхідно звільнити, коли знищується об'єкт. Це особливо

```
class FirstDescendant public SomeClass
{
    . . .
public:
    virtual void firstFunc();
    virtual int secondFunc(int x);
};

class SecondDescendant : public SomeClass
{
    . . .
public:
    void firstFunc();
   virtual double thirdFunc(double x, double y);
};

class ThirdDescendant : public SecondDescendant
{
    . . .
public:
    virtual double thirdFunc(double x, double y);
};
```

The `SecondDescendant` class does not override the function `secondFunc()`. Objects of this class use `secondFunc()`, which is defined in a base class. The `ThirdDescendant` class use `firstFunc()` of class `SecondDescendant`. It uses `secondFunc()` of `SomeClass` and overrides `thirdFunc()`.

There are three ways of calling virtual functions:, for objects, for pointers, and for references. Calling virtual function for an object is determined at compile time. In other cases, address of the specific function is determined at runtime.

Object of class, which has a virtual function, is called *polymorphic object*. Each polymorphic object contains a special field, in which the constructor stores the address of the so-called *virtual method table* (VMT). Such a table is created for each class with virtual functions. A virtual method table contains the addresses of virtual functions. These addresses are used for calls of virtual functions.

Destructors can also be virtual. After you declare destructor as virtual, destructors of all derived classes are virtual automatically. Virtual destructors can accurately determine the amount of memory that must be freed when the object is destroyed. This is especially important for objects created in the

важливо для об'єктів, які створені у динамічній пам'яті та знищуються за допомогою операції **delete**.

Віртуальні функції не можуть бути описані з модифікатором **static**. Віртуальні функції можуть бути inline, але цей специфікатор має ефект лише для виклику функції для об'єкта (не для або посилання).

Іноді в базовому класі недоцільно здійснювати реалізацію певної віртуальної функції, оскільки клас створено для представлення базової абстракції. В такому випадку функцію можна описати як чисто віртуальну. Для цього використовують спеціальний синтаксис:

```
virtual void f() = 0;
```

Абстрактний клас – це клас, який містить оголошення хоча б однієї чисто віртуальної функції:

```
class AbstractBaseClass
{
    . . .
public:
    virtual void f() = 0;
    virtual int secondFunc(int x);
};
```

Похідні класи повинні перевантажити чисто віртуальні функції. В іншому випадку такі класи залишаються абстрактними.

Неможна створити об'єкт абстрактного класу, але можна описати вказівник або посилання на абстрактний клас та проініціалізувати їх конкретними похідними класами.

3.3.9. Обробка винятків

Дуже часто функція програми, у якій виникає певна помилка, не має можливості виправити цю помилку, бо невідомим є контекст виклику цієї функції. Помилку необхідно передати до тієї частини програми, в якій її можна обробити.

Механізм генерації та обробки винятків надає спосіб розв’язання цієї проблеми.

Програма може згенерувати винятки за допомогою оператора **throw**. Наприклад, розглянемо розв’язання задачі `y=1/value`:

```
double SomeFunc(double value)
{
    if (value == 0)                // Помилка
        throw "Division by Zero"; // Генерується виняток
                                   // типу char*
```

heap and destroyed using the **delete** operation.

Virtual functions cannot be **static**. Virtual functions can be inline, but this specifier has effect only for the call applied to an object (not to a pointer or reference).

Sometimes it is not reasonable to implement some virtual function in the base class, since the class is created to represent the basic abstraction. In such case, the function can be described as pure virtual. To do this, use a special syntax:

```
virtual void f() = 0;
```

An *abstract class* is a class with a declaration of at least one pure virtual function:

```
class AbstractBaseClass
{
    . . .
public:
    virtual void f() = 0;
    virtual int secondFunc(int x);
};
```

Derived classes must override pure virtual functions. Otherwise, such classes remain abstract.

Abstract classes cannot be instantiated, but we can declare pointers and references to abstract classes. Such pointers and references can be initialized with objects of derived classes.

3.3.9. Exception Handling

Very often, some functions, in which runtime errors occur, are not able to fix this problem because the context of invoking these functions is unknown. Error should be transferred to that part of the program in which it can be processed.

The *mechanism of throwing and handling of exceptions* provides a way to solve this problem.

A program throws an exception by executing a **throw** statement. For example, consider the solution of the problem `y=1/value`:

```
double SomeFunc(double value)
{
    if (value == 0)                 // Error
        throw "Division by Zero";   // Exception of type
                                    // char*
```

```
    return 1 / value;
}
```

Твердження **throw** можна порівняти з оператором return. Вираз throw складається з відповідного ключового слова та виразу. Тип виразу визначає *тип винятку*. Цей тип ніяк не пов'язаний з типом, який повинна повертати функція.

При визначенні функції, яка генерує виняток, можна явно вказати в заголовку список типів цих винятків. Наприклад:

```
double SomeFunc(double value) throw (char*)
{
  . . .
}
```

При описі прототипу функції:

```
double SomeFunc(double value) : throw (char*);
```

Список винятків функції є частиною її заголовка. Його треба наводити в усіх оголошеннях функції. Якщо функція не може взагалі генерувати винятків, це визначається порожнім списком винятків:

```
bool g(int, int) throw(); // порожній список винятків
```

На жаль, компілятори здебільшого не перевіряють відповідності списку винятків.

Після генерації здійснюється так зване зворотне розкручування стека, яке можна порівняти з виконанням послідовності тверджень **return**, кожне з яких повертає той самий об'єкт. Цей об'єкт передається за посиланням до свого оброблювача.

Блок **try** містить код, у якому може бути згенерований виняток. Наприклад

```
try
{
  double x = 0;
  cout << SomeFunc(x); // Може генеруватись виняток
}
```

Після блока **try** повинні знаходитись один або кілька блоків, які обробляють винятки залежно від типів. Такі оброблювачі складаються з ключового слова catch, за яким у дужках стоїть тип винятку, та програмного блока. Спеціальна конструкція catch (...) застосовується для створення оброблювача всіх типів винятків. Наприклад:

```
    return 1 / value;
}
```

A **throw** expression statement is similar to a return statement. A throw statement consists of the keyword throw and an expression. The expression type determines type of an exception. There is no relation to a function returning type.

When defining a function that generates an exception, you can explicitly specify a list of these exceptions in the header. For example: :

```
double SomeFunc(double value) throw (char*)
{
  . . .
}
```

Or in declaration:

```
double SomeFunc(double value) : throw (char*);
```

The exception list is a part of a function's header. All declarations of a given function must have the same lists of exceptions. If a function never throws any exceptions, it can be declared as follows:

```
bool g(int, int) throw(); // empty list of exceptions
```

Unfortunately, compilers usually ignore exception specifications.

After the exception throwing, so-called reverse unwinding of the stack is carried out, which can be compared with the execution of a sequence of **return** statements with returning of the same object. This object is passed to its handler.

The **try** block contains code that might throw an exception. For example:

```
try
{
    double x = 0;
    cout << SomeFunc(x); // may throw an exception
}
```

A **try** block must be followed by one or more blocks, each of which handles a different type of exception. An exception handler consists of the catch keyword, followed by an exception type in parentheses, followed by programming block. A special construct catch (...) is used to create a handler for all types of exceptions. For example:

```
try
{
    double x = 0;
    cout << SomeFunc(x); // може згенерувати виняток
}
catch (char*)
{
    // обробляє виняток типу char*
}
catch (int)
{
    // обробляє виняток типу int
}
catch (...)
{
    // обробляє інші винятки
}
```

Оброблювач винятку може також вживати *змінну- виняток*:

```
try
{
    double x = 0;
    cout << SomeFunc(x); // може згенерувати виняток
}
catch (char* c)//використання змінної-винятку
{
    cout << c; // виведення "Division by Zero"
}
```

У деяких випадках оброблювач винятків не може повністю обробити виняток та повинен передати його зовнішньому оброблювачеві. Це можна зробити за допомогою оператора **throw** без виразу:

```
catch (char* c)
{
    // локальна обробка винятку
    throw;  // повторна генерація винятку
}
```

Для винятків можна застосовувати фундаментальні типи даних, але кращий підхід базується на створенні власних типів, оскільки розпізнавання винятків базується на типах. Найчастіше це нові класи, які в окремому випадку можуть бути порожніми:

```
class Not_Found {};
```

```
try
{
  double x = 0;
  cout << SomeFunc(x); // may throw an exception
}
catch (char*)
{
  // handles exceptions of type char*
}
catch (int)
{
  // handles exceptions of type int
}
catch (...)
{
  // handles other exceptions
}
```

Exception handler can also use the variable of exception type:

```
try
{
    double x = 0;
    cout << SomeFunc(x); // may throw exception
}
catch (char* c) // the use of exception variable
{
    cout << c;  // prints "Division by Zero"
}
```

In some cases, the exception handler cannot fully handle the exception and should pass it to the external handler. This can be done using **throw** operator without expression:

```
catch (char* c)
{
  // local handling for the exception
  throw;  // rethrow the exception
}
```

You can use fundamental types for exception objects in throw expression, but the best approach is to create your own types, because recognition of exceptions is based on their types. Most often, you should create new classes that sometimes may be empty:

```
class Not_Found {};
```

```
class Bad_Data {};
```

Дуже часто такі типи описують в області видимості класів, до яких вони належать:

```
class MyTramClass
{
public:
    class Not_Found {};
    class Bad_Data {};

    void f(int i)
    {
        if (i < 0)  // Створення тимчасового об'єкта
            throw Bad_Data(); // Генерується виняток типу
                              // Bad_Data
        if (i == 0) // Створення тимчасового об'єкта
            throw Not_Found();// Генерується виняток типу
                              // Not_Found
    }
};

void main()
{
    try
    {
        MyTramClass m;
        m.f(-2); // може згенерувати виняток
    }
    catch (MyTramClass::Bad_Data)
    {
        // обробка винятку типу Bad_Data
    }
    catch (MyTramClass::Not_Found)
    {
        // обробка винятку типу Not_Found
    }
}
```

Для більш зручної обробки помилок можна описати більш складні класи винятків.

На відміну від багатьох інших мов програмування, C++ не вимагає, щоб усі класи винятків походили з одного базового класу. Проте іноді доцільно створювати власні ієрархії винятків. У цьому випадку оброблювач винятка базового типу автоматично обробляє винятки похідних типів.

```
class Bad_Data {};
```

Very often, these types are defined in the scope of the class to which they belong:

```
class MyTramClass
{
public:
    class Not_Found {};
    class Bad_Data {};

    void f(int i)
    {
        if (i < 0)  // Creating of temporary object
            throw Bad_Data(); // Throwing exception of
                              // Bad_Data
        if (i == 0) // Creating of temporary object
            throw Not_Found();//Throwing exception of type
                              // Not_Found
    }
};

void main()
{
    try
    {
        MyTramClass m;
        m.f(-2); // may throw an exception
    }
    catch (MyTramClass::Bad_Data)
    {
        // handling exception of type Bad_Data
    }
    catch (MyTramClass::Not_Found)
    {
        // handling exception of type Not_Found
    }
}
```

For more convenient errors handling you can define more complex exception classes.

Unlike many other programming languages, C++ does not require that all exceptions classes to be derived from one base class. However, sometimes it is advisable to create your own exception hierarchy. In this case, the base type exception handler automatically handles exceptions of derived types.

3.3.10. Перевантаження операцій

При проектуванні класу можна визначити набір операцій, які можна виконувати над об'єктами. Визначення операції для об'єкта класу, структури чи перелічення здійснюється за допомогою так званої *операторної функції*. Ім'я операторної функції складається зі службового слова **`operator`**, за яким міститься одна з визначених операцій C++.

Операторна функція необов'язково повинна бути функцією-елементом. Якщо вона реалізована як звичайна функція, то принаймні один з її параметрів повинний бути типу, визначеного користувачем. Отже, призначення оператора не можна змінити для вбудованих типів.

Автор класу може перевантажити всі операції, визначені в C++, крім чотирьох: `::`, `*`, `.`, `?:`. Не можна задати нову операцію, наприклад `**` чи `<>`. Визначені пріоритети операторів змінити не можна. Повинне зберігатися визначене число аргументів операції. Не можна задавати значення параметрів за умовчанням для операторних функцій. Чотири операції ("`+`","`-`","`*`" і "`&`") можуть використовуватися і як бінарні, і як унарні.

Якщо операторна функція реалізується як функція-елемент класу, то вважається, що лівий операнд функції – об'єкт, на який указує **`this`**. Якщо потрібно, щоб лівий операнд був іншого типу, то операторна функція не може бути функцією-елементом.

Якщо операторна функція є елементом класу, кількість її параметрів повинна бути на 1 менше кількості операндів операції, що перевантажується, тому що в цьому випадку першим операндом є сам об'єкт.

Деякі операції – присвоювання "`=`", індексація "`[]`", виклик функції "`()`" і вибір елемента "`->`" повинні бути перевантажені тільки як функції-елементи.

Операторні функції можуть бути перевантажені, якщо їх можна розрізнити за списком параметрів.

Операторні функції, що реалізують операції введення-виведення (`>>` і `<<`), не повинні бути функціями-елементами. Першим параметром повинне бути посилання на об'єкт-потік (`ostream` для операції виведення та `istream` для введення). Другим параметром повинне бути посилання на об'єкт класу, для якого перевантажується операція. Для операції виведення таке посилання може бути посиланням на константний об'єкт. Операторні функції, що реалізують введення-виведення, повинні повертати посилання на потік, отриманий як перший операнд.

3.3.10. Operator Overloading

When designing a class, you can define a set of operations that can be performed on objects. The definition of operations for the object of class, structure, or enumeration is carried out using so-called *operator functions*. An operator function name consists of **`operator`** keyword operator, followed by one of the predefined C++ operators.

An operator function does not have to be a member function. If it is implemented as ordinary function, at least one of its parameters must be of user-defined type. Thus, the purpose of the operator cannot be changed for built-in types.

You can overload all defined in C++ operations for objects of your class, but not these four: `::`, `.*`, `.` , and `?:`. You cannot specify a new operation, such as `**` or `<>`. You cannot change the priorities of operators. A certain number of operands cannot be changed. You cannot specify default parameter values for the operator functions. Four operations ("`+`", "`-`", "`*`" and "`&`") can be used as both binary and unary.

If the operator function is implemented as a member function of a class, it is considered that the left operand of function is the object, to which **`this`** pointer points. If you want to assign the left operand to different type, the operator function cannot be implemented as a member function.

If the operator function is the class member, the number of parameters is to be one less than the number of operands of the overloaded operation, so in this case the first operand is the object itself.

Some operators, such as assignment ("`=`"), indexing "`[]`" function call "`()`" and member selection "`->`" can be overloaded using member functions only.

Operator functions can be overloaded if they can be distinguished by the parameter list.

Operator functions that implement input and output operators (`>>` and `<<`) cannot be member functions. The first arguments of operator functions are references to stream objects (`ostream` for output operation and `istream` for input operation). The second argument is the reference to an object of a user-defined type. Operator functions must return reference to stream objects obtained as the first argument.

Для перевантаження префіксних операцій ++ та -- використовуються функції-елементи вигляду:

```
X& X::operator++();
X& X::operator--();
```

Для перевантаження постфіксних операцій ++ та -- використовуються функції-елементи вигляду:

```
X X::operator++(int);
X X::operator--(int);
```

Операція присвоювання перевантажується в тих випадках, коли поелементне копіювання об'єктів призводить до виникнення помилки.

При перевантаженні операції звертання за індексом "[]" потрібно враховувати, що немає обмежень ні на тип вхідного параметра, ні на значення, що повертається функцією.

Перевантаження операції виклику функції реалізується через функцію-елемент. Допускається довільна кількість параметрів довільних типів.

Операторні функції можна викликати явно, наприклад:

```
X x, x1;
x = x.operator+(x1);
```

Існує особливий вид операторних функцій – *операції перетворення типів*, що дозволяють перетворювати об'єкт у заданий тип за умовчанням. *Операція перетворення* повинна бути реалізована у вигляді функції-елемента, і в загальному випадку вона має такий вигляд:

```
X::operator T(); // T - ім'я типу
```

Для операції перетворення не можна задавати тип значення, що повертається, чи вказувати список формальних параметрів. Наприклад, завдяки операції перетворення допустимі наступні дії:

```
class X
{
    int i;
public:
    operator int() { return i; }
    . . .
};
. . .
X x;
int k = x; // Використовується елемент даних i
int n = x + k;
```

To overload prefix ++ and -- operators, the following member functions are used:

```
X& X::operator++();
X& X::operator--();
```

To overload postfix ++ and -- operators, the following member functions are used:

```
X X::operator++(int);
X X::operator--(int);
```

You should overload assignment operator in cases where the elementwise copying of objects results in an error.

When you overload the index operator ("[]"), you should keep in mind that there are no restrictions on parameter type or on the return type of the function.

The overloading of function call operator can be implemented by member function only. An arbitrary number of parameters of arbitrary types is acceptable.

Operator functions can be invoked explicitly. For example:

```
X x, x1;
x = x.operator+(x1);
```

There is a special kind of operator functions, so-called *type conversion operations*, allowing to transform an object into the specified type in implicit way. The *conversion operation* must be implemented as a member function, and generally, it looks like this:

```
X::operator T(); // T - ім'я типу
```

You cannot specify returning type and a list of formal parameters for the conversion operator function. For example, due to conversion operation, the following actions are valid:

```
class X
{
    int i;
public:
    operator int() { return i; }
    . . .
};
. . .
X x;
int k = x; // Data member i is used
int n = x + k;
```

3.3.11 Створення та використання шаблонів

Механізм шаблонів у мові C++ дозволяє створювати шаблонні функції та параметризовані класи для реалізації узагальнених дій, не залежних від типів даних, до яких вони застосовуються.

Визначення й оголошення шаблона функції починається службовим словом **template**. За ним іде в кутових дужках ("<" і ">") список формальних параметрів шаблона, розділених комами. Список формальних параметрів шаблона не може бути порожнім. Кожен формальний параметр, що визначає тип, складається зі службового слова **class**, за яким іде ідентифікатор. Ім'я формального параметра в списку повинне бути унікальним.

Формальні параметри шаблонів можуть використовуватися для визначення типу результату і формальних параметрів шаблонної функції. У тілі шаблонної функції також можуть використовуватися формальні параметри шаблона.

Як приклад шаблона наведемо функцію підсумовування елементів масиву довільного типу. Головне, щоб для елементів масиву були визначені операції присвоювання, у тому числі присвоювання константи "нуль", і "+=".

```
template <class SomeType>
SomeType sumOfArray(SomeType *a, const int size)
{
    SomeType sum = 0;
    for (int i = 0; i < size; i++)
        sum += a[i];
    return sum;
}
```

Крім того, можна визначити шаблон функції виведення в стандартний потік елементів масиву:

```
template <class SomeType>
void printArray(SomeType *a, const int size)
{
    for (int i = 0; i = 0; i < size; i++)
        cout << a[i] << ' ';
    cout << endl;
}
```

Конкретні визначення функцій, що відповідають шаблону, компілятор генерує при виклику шаблонної функції з параметрами конкретного типу.

3.3.11. Creation and Using Templates

The mechanism of templates in C++ allows you to create template functions and parameterized classes to implement generic actions that are not dependent on the type of data to which they are applied.

The definition and declaration of the template function begins with keyword **template** followed by list of formal template parameters separated by commas and placed in angle brackets ("<" and ">"). The list of formal template parameters cannot be empty. Each formal parameter that determines the type consists of the keyword **class**, followed by an identifier. A name in formal parameter within list must be unique.

Formal parameters of the template can be used in template function for definition of the resulting type and of types of formal parameters. Formal template parameters can also be used in the function body.

As an example, we give a template function that sums the elements of an array of arbitrary type. It is essential to support the elements of the array assignment operator, including assignment of a constant 0 and "+=".

```
template <class SomeType>
SomeType sumOfArray(SomeType *a, const int size)
{
    SomeType sum = 0;
    for (int i = 0; i < size; i++)
        sum += a[i];
    return sum;
}
```

In addition, you can define a template function for output to array elements standard stream:

```
template <class SomeType>
void printArray(SomeType *a, const int size)
{
    for (int i = 0; i = 0; i < size; i++)
        cout << a[i] << ' ';
    cout << endl;
}
```

When calling the template function with the parameters of a particular type, the compiler generates a specific definition of the functions based on the template.

Для описаного вище прикладу можна запропонувати таке використання шаблонних функцій:

```
int main()
{
    int a[] = {1, 2, 3};
    printArray(a, 3);
    cout << sumOfArray(a, 3) << endl;
    double b[] = {1.1, 2.2, 3.3, 4.5};
    printArray(b, 4);
    cout << sumOfArray(b, 4) << endl;
    return 0;
}
```

Бувають випадки, коли для якихось конкретних типів потрібно дати особливе визначення шаблонної функції. У цьому випадку програміст повинен задати свій спеціальний варіант функції. Наприклад, шаблон функції `min()` працює для типів, для яких визначена операція "<":

```
template <class Type>
Type min(Type a, Type b)
{
    return a < b ? a : b;
}
```

Цей шаблон не підходить для варіанта порівняння рядків. Для них визначається спеціальний варіант функції:

```
char* min(char* s1, char* s2)
{
    return strcmp(s1, s2) < 0 ? s1 : s2;
}
```

Спочатку компілятор досліджує всі нешаблонні варіанти функції. Потім досліджуються всі шаблонні варіанти функції. Якщо шаблон не знайдено, повторно досліджуються всі нешаблонні варіанти функції з застосуванням перетворення типів.

Для того щоб можна було конкретизувати шаблон, компілятор повинен бачити не тільки оголошення, але і визначення функції. Тому визначення шаблонних функцій можна і треба поміщати в заголовні файли.

При виклику функції фактичний параметр шаблона можна вказати явно, наприклад:

```
int i = min<int>(2, 3);
```

For the example above, such use of template functions can be offered:

```
int main()
{
    int a[] = {1, 2, 3};
    printArray(a, 3);
    cout << sumOfArray(a, 3) << endl;
    double b[] = {1.1, 2.2, 3.3, 4.5};
    printArray(b, 4);
    cout << sumOfArray(b, 4) << endl;
    return 0;
}
```

There are cases when a special definition of template function should be given for some specific types. In this case, a programmer must specify a special version of the function. For example, the template function `min()` works for the types that support the operation "<":

```
template <class Type>
Type min(Type a, Type b)
{
    return a < b ? a : b;
}
```

This template is not suitable for string comparison option. A special version of the function can be defined for them:

```
char* min(char* s1, char* s2)
{
    return strcmp(s1, s2) < 0 ? s1 : s2;
}
```

First, the compiler examines all non-template variants of a function. Then it examines all the template variants of the function. If the template is not found, the compiler re-examines all the variants of non-template functions using type conversion.

In order to be able to specify a template, the compiler must not only see a function declaration, but also its definition. Therefore, the definition of the template functions can and should be placed in the header files.

When you call the function, the actual template parameter can be specified explicitly. For example:

```
int i = min<int>(2, 3);
```

Ідея узагальненого програмування знайшла своє найбільш повне втілення у так званих *параметризованих класах*.

Попереднє оголошення і визначення шаблонного класу починається зі службового слова **template**. За ним іде список формальних параметрів шаблона типу. Цей список не може бути порожнім. Наприклад:

```
template <class T> class X
{
    T t;
public:
    void set(T t1) { t = t1; }
};
```

Параметрами шаблонів можуть бути параметри-типи, параметри звичайних типів і параметри-шаблони. У шаблона може бути кілька параметрів. Цілі аргументи використовуються найчастіше для задання розмірів і границь масивів. Цілий аргумент шаблона повинен бути константою. Можливе задання параметрів шаблона за умовчанням.

Параметризовані класи допускають використання механізму спадкування. Можливе створення похідного параметризованого класу як від шаблона, так і від класу, що не є шаблоном.

Функція-елемент шаблонного класу вважається неявною шаблонною функцією, а параметри шаблона типу для її класу - параметрами її шаблона. Для деяких типів стандартні функції-елементи не підходять. У таких випадках можна явно задавати реалізацію функції, розрахованої на конкретний тип. Перед реалізацією таких функцій потрібно спеціальне оголошення **template**<> без параметрів. Крім того, можна дати особливе визначення шаблонного класу, розраховане на конкретний тип.

Шаблон класу може мати статичні елементи. Кожен клас, згенерований за шаблоном, має свою копію статичних елементів.

Створені за одним шаблоном типи будуть різними і між ними неможливе відношення спадкування, крім єдиного випадку, коли в цих типів ідентичні параметри шаблона. Наприклад:

```
X<int>   x1;
X<short> x2;
X<int>   x3;
```

Тут `x1` і `x3` одного типу, а `x2` – зовсім іншого. Автоматичне приведення типів не здійснюється.

```
x2 = x3; // помилка
```

The idea of generic programming has found its fullest expression in the so-called *parameterized classes*.

Pre-declaration and definition of template class begins with the keyword **`template`**, followed by a list of formal parameters of the type template. This list cannot be empty. For example:

```
template <class T> class X
{
    T t;
public:
    void set(T t1) { t = t1; }
};
```

Template parameters can be of conventional types and template types. Template can have several parameters. Integer arguments are mostly used for setting size and array boundaries. A template argument must be a constant. The settings of default template parameters are possible.

Parameterized classes allow the use of inheritance. You can create a parameterized class derived from both the template and non-template classes.

Member function of a template class is considered to be an implicit template function with template parameters defined for its class. Common implementation of member functions can be unsuitable for some types. In such cases, you can explicitly specify the implementation of functions designed for a particular type. Before the implementation of such functions, special declaration **`template`**<> without parameters is required. You could also give a special definition of a template class, intended for a particular type.

Template class can have static elements. Each class generated by such template has its own copy of static elements.

Types created from the same template, will be different and there is no compatibility between objects of these types, except for a single case, when these types of identical template parameters. For example:

```
X<int>   x1;
X<short> x2;
X<int>   x3;
```

Here `x1` and `x3` are of the same type, but `x2` is of completely another type. Automatic type casting is not done.

```
x2 = x3; // error
```

3.3.12 Використання стандартної бібліотеки C++

Стандартна бібліотека C++ побудована на інтенсивному використанні шаблонів. Вона пропонує низку класів, які дозволяють працювати з послідовностями даних. Це так звані *контейнерні класи*. Найбільш популярним серед них є вектор. *Вектор* багато в чому аналогічний традиційному одновимірному масиву. Для використання векторів до вихідного файлу треба підключити заголовний файл `<vector>`. Тип елемента вказується в кутових дужках ("<" та ">"). У наступному прикладі визначається змінна `a` як вектор з `n` дійсних чисел:

```
#include <vector>
using namespace std;
...
vector<double> a(n);
```

Тут `n` може бути як константою, так і змінною.

Звертатися до окремих елементів можна за індексом, як до елементів масиву, наприклад:

```
for (int i = 0; i < a.size(); i++)
    cin >> a[i];
```

За допомогою функції-елемента `size()` отримують поточну довжину вектора. Можна також описати "порожній" вектор (довжиною `0`), а потім додавати елементи у кінець за допомогою функції-елемента `push_back()`. Наступним способом можна отримати вектор, елементи якого дорівнюють 10, 11, 12, 13 та 14:

```
vector<int> b;
for (int i = 10; i < 15; i++)
    b.push_back(i);
```

Для роботи з послідовностями символів (рядків) стандартна бібліотека C++ пропонує спеціальний клас – `string`. Для використання рядків стандартної бібліотеки до файлу треба включити заголовний файл `string` та підключити простір імен `std`:

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

Завдяки наявності необхідних конструкторів, а також перевантаженню операцій, об'єкти класу `string` можна присвоювати один одному за допомогою операції `=`, порівнювати за допомогою операцій `<`, `<=`, `==`, `>=`, `>` та `!=`. Можна звертатись до окремих символів за індексом.

3.3.12 Using the C++ Standard Library

The C++ Standard Library is based on intensive use of templates. It offers a number of classes that let you work with sequences of data. These are so-called *container classes*. The most popular among them is vector. *Vector* is very similar to the traditional one-dimensional array. To use vectors, you should include the header file `<vector>` to the source code. Item type should be specified in angle brackets ("<" and ">"). In the following example, the variable is defined as a vector of `n` real numbers:

```
#include <vector>
using namespace std;
...
vector<double> a(n);
```

Here `n` can be both constant and variable.

You can access individual elements by the index, as the elements of an array, for example:

```
for (int i = 0; i < a.size(); i++)
    cin >> a[i];
```

With member function `size()` you can get the current length of the vector. You can also define the "empty" vector (with zero length), and then add items to the end using the member function `push_back()`. For example, you can get a vector, which elements are equal to 10, 11, 12, 13 and 14 in a such way:

```
vector<int> b;
for (int i = 10; i < 15; i++)
    b.push_back(i);
```

To work with sequences of characters (strings), C++ Standard Library offers a special class called `string`. To use the strings of the Standard Library, you should include the header file `string` and connect namespace `std`:

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

Due to the necessary constructors and operator overloading, objects of class `string` can be assigned to one another by using the operation = ; they can be compared by using the operations `<`, `<=`, `==`, `>=`, `>`, and `!=`. You can access individual characters by index.

Загальна кількість символів повертається функцією-елементом `length()`. Наприклад, усі символи рядка `s` у наступному прикладі замінюються символом `'a'`:

```
for (int i = 0; i < s.length(); i++)
    s[i] = 'a';
```

Стандартна бібліотека пропонує програмісту принципово новий підхід до проектування і реалізації програм на C++.

Для роботи з векторами та іншими контейнерами стандартної бібліотеки C++ прийнято використовувати підхід, оснований на застосуванні так званих алгоритмів та ітераторів.

Алгоритм – це спеціальна функція, призначена для роботи з послідовністю елементів. Завдяки використанню шаблонів, алгоритми можуть працювати з даними різної природи. Для використання алгоритмів до вихідного файлу треба підключити заголовний файл `algorithm`:

```
#include <algorithm>
```

В алгоритмах, які працюють з послідовностями, організується спеціальний цикл, початок і кінець якого задаються двома ітераторами. *Ітератор* – це об'єкт, за своїми функціями аналогічний вказівнику. Він використовується для організації циклу по об'єктах, що зберігаються в контейнері.

Найпростіший спосіб одержання ітераторів для організації циклу по всіх елементах вектора – використання функцій-елементів `begin()` та `end()`. У такий спосіб можна, наприклад, організувати сортування вектора цілих чисел:

```
vector<int> v(8);
. . .
// Використання алгоритму сортування:
sort(v.begin(), v.end());
```

При використанні більш складних типів, для яких не визначене відношення "більше" і "менше", функцію sort() потрібно викликати з трьома параметрами. Третій параметр – *функція-предикат* – функція, що повертає результат типу **`bool`**. Відповідна функція для алгоритму sort() повинна приймати два параметри типу елементів послідовності та повертати true, якщо елементи не потрібно міняти місцями, або false в іншому випадку.

Для виконання будь-яких дій зі всіма елементами послідовності можна використовувати алгоритм `for_each()`, при виклику якого третім параметром указується функція, що виконується для кожного елемента

The number of characters returned by member function `length()`. For example, all the characters of the string s are replaced by character `'a'`:

```
for (int i = 0; i < s.length(); i++)
    s[i] = 'a';
```

The Standard Library offers the programmer a fundamentally new approach to the design and implementation of programs in C++.

To work with vectors and other containers of the C++ Standard Library the approach based on the use of so-called algorithms and iterators is commonly used.

Algorithm is the special function that is designed to work with the sequence of items. Through the use of templates, algorithms can handle data of different nature. To use the algorithms, the `algorithm` header file should be included to the source file:

```
#include <algorithm>
```

To iterate over sequences, algorithms organize a special cycle, the beginning and end of which are specified by two iterators. *Iterator* is an object similar to pointer in its functions. It is used for a loop on the objects stored in the container.

The easiest way to obtain iterators for a loop over all the vector items is use of member functions `begin()` and `end()`. As an example, you can organize sorting vector of integers:

```
vector<int> v(8);
. . .
// Use sorting algorithm:
sort(v.begin(), v.end());
```

When using more complex types, which do not support relationship "more" and "less", function sort() must be called with three parameters. The third parameter, so-called *predicate function*, is a function that returns type **`bool`**. The corresponding function for the algorithm sort() should accept two elements of the sequence and return true, if the items do not need to swap, or false otherwise.

To perform any action over all the elements of a sequence, an algorithm `for_each()` can be used; the third parameter indicates the function executed for each item of a sequence.

.

послідовності. Функція повинна мати параметр типу елемента послідовності чи посилання на нього.

Основний недолік безпосереднього використання функції при роботі з алгоритмами полягає в неможливості збільшення кількості параметрів функції, яка викликається. Розв’язати проблему дозволить використання так званих функціональних об'єктів.

Функціональний об'єкт являє собою екземпляр якого-небудь класу, в якому за допомогою функції-елемента перевантажена операція "круглі дужки". Звертання до операції здійснюється кожного разу, коли функціональний об'єкт використовується як функція. Клас, що описує тип функціонального об'єкта, може мати елементи даних і конструктор, у якому ці елементи даних ініціалізуються необхідними значеннями. Ці значення використовуються у функції, що перевантажує операцію "круглі дужки".

3.3.13. Приклади програм

Приклад 1. Приклад створення класу. Припустимо, необхідно спроектувати програму, в якій описується клас для зберігання даних про країни світу: назва, площа, кількість населення. З відповідно підготовленого файлу треба прочитати дані про країни та розташувати їх у масиві, здійснити сортування масиву за алфавітом назв. Треба також вивести на екран дані про країни з густотою населення більше 100 мешканців на 1 кв.км.

Попередньо у будь-якому текстовому редакторі треба створити текстовий файл `Countries.txt` з даними про країни: назва, площа та населення. Файл може, наприклад, мати такий вміст:

```
Ukraine 603700 48800000
France  544000 57804000
Sweden  450000 8745000
Germany 357000 81338000
```

Створюємо новий проект, наприклад `CountiesClasses`. Додаємо заголовний файл, наприклад `CountiesClasses.h`. У цьому файлі слід розмістити опис класів `Country` та `Countries`:

```
#ifndef CountiesClasses_h
#define CountiesClasses_h
#include <string>
#include <vector>

using std::string;
```

The function must have a parameter of an element type or a reference to it.

The main disadvantage of the direct use of this function is the inability to increase the number of parameters of the function being called. To solve the problem, the so-called functional objects are used.

The *functional object* is an instance of a class, which overloads operation "parentheses". Call to this operation is performed whenever a functional object is used as a function. Class that represents functional object can have data members and a constructor, where these data members are initialized with required values. These values are used in the function that overloads the operation "parentheses".

3.3.13. Examples of programs

Example 1. An example of creation a class. Suppose we want to design a program that defines a class for storage data on country, namely the name, the area, the population. A program should read the information about countries from the corresponding file placing it in an array. It should sort the array in alphabetical order of names. It should also display the data on countries with population density over 100 inhabitants per one sq.km.

First, we need to create a text file `Countries.txt` with data on countries: name, area and population. Any text editor can be used. The file can, for example, have such content:

```
Ukraine 603700 48800000
France  544000 57804000
Sweden  450000 8745000
Germany 357000 81338000
```

Now we create a new project, for example `CountiesClasses`, and add header file, for example `CountiesClasses.h`. This file will contain definitions of classes `Country` and `Countries`

```
:
#ifndef CountiesClasses_h
#define CountiesClasses_h
#include <string>
#include <vector>

using std::string;
```

```
using std::vector;

class Country //клас, що визначає тип - країна
{
private:
    string name;//назва країни
    double area;//площа країни
    int    population;//населення країни
public:
    //функція-елемент, що дозволяє отримати назву країни
    string getName()         const { return name; }
    //функція-елемент, що дозволяє отримати площу країни
    double getArea()         const { return area; }
    //функція-елемент, що дозволяє отримати населення країни
    int   getPopulation()    const { return population; }
    //функція-елемент, що дозволяє задати назву країни
    void  setName(string value)    { name = value; };
    //функція-елемент, що дозволяє задати площу країни
    void  setArea(double value)    { area = value; }
    //функція-елемент, що дозволяє задати населення
    void  setPopulation(int value) { population = value; }
    //функція-елемент, що дозволяє отримати
    //густоту населення
    double density() const;
};


class Countries//клас, що визначає тип – країни
{
private:
    //вектор, що буде зберігати дані про країни
    vector<Country> vc;
public:
    //функція-елемент, що буде виконувати читання
    //вихідних даних
    bool readCountries(const char* fileName);
    //функція-елемент, що буде виконувати
    //виведення даних про країни
    void printCountries() const;
    //функція-елемент, що буде виконувати сортування країн
    void sortCountries();
    //функція-елемент, що буде виконувати відбір
    //країн за умовою
    void writeIfGreater(const char* fileName) const;
};


#endif
```

```
using std::vector;

class Country // class that defines the country type
{
private:
    string name; // name of the country
    double area; // area of the country
    int    population; // population of the country
public:
    // member function that allows getting a country name
    string getName()         const { return name; }
    // member function that allows getting a country area
    double getArea()         const { return area; }
    // member function that allows getting population
    int   getPopulation()    const { return population; }
    // member function that allows setting a country name
    void  setName(string value)    { name = value; };
    // member function that allows setting a country area
    void  setArea(double value)    { area = value; }
    // member function that allows setting population
    void  setPopulation(int value) { population = value; }
    // member function that allows getting
    // a population density
    double density() const;
};


class Countries // class that defines the countries type
{
private:
    // vector that will store data on countries
    vector<Country> vc;
public:
    // member function that will perform
    // the reading source data
    bool readCountries(const char* fileName);
    // member function that will carry
    // output data on countries
    void printCountries() const;
    // member function that will carry sorting data
    // on countries
    void sortCountries();
    // member function that will carry selecting data
    // on countries
    void writeIfGreater(const char* fileName) const;
};


#endif
```

Як видно з тексту, рядки типу `string` можна присвоювати іншим рядкам за допомогою операції =.

Тепер до проекту слід додати файл `CountiesClasses.cpp`, у якому здійснюється реалізація цих функцій:

```
#include <fstream>
#include <iostream>
#include "CountiesClasses.h"

using std::ifstream;
using std::ofstream;
using std::cout;
using std::endl;

// визначення функції-елемента, що обчислює густоту населення
double Country::density() const
{
    return population / area;
}

// визначення функції-елемента, що виконує
// читання вихідних даних
bool Countries::readCountries(const char* fileName)
{
    ifstream in(fileName);
    if (in == 0)
    {
        return false;
    }
    Country c;
    string  name;
    double  area;
    int     population;
    while (in >> name >> area >> population)
    {
        c.setName(name);
        c.setArea(area);
        c.setPopulation(population);
        vc.push_back(c);
    }
    return true;
}

// визначення функції-елемента, що виконує виведення даних
// про країни
void Countries::printCountries() const
```

As it can be seen from the code, objects of type `string` can be assigned to other strings using the operation =.

Now we need to add a new file `CountiesClasses.cpp` to the project, in which these functions are implemented:

```
#include <fstream>
#include <iostream>
#include "CountiesClasses.h"

using std::ifstream;
using std::ofstream;
using std::cout;
using std::endl;

// definition of a member function that calculates
// population density
double Country::density() const
{
  return population / area;
}

// definition of a member function that
// reads the source data
bool Countries::readCountries(const char* fileName)
{
    ifstream in(fileName);
    if (in == 0)
    {
        return false;
    }
    Country c;
    string  name;
    double  area;
    int     population;
    while (in >> name >> area >> population)
    {
        c.setName(name);
        c.setArea(area);
        c.setPopulation(population);
        vc.push_back(c);
    }
    return true;
}

// definition of a member function that performs
// data output
void Countries::printCountries() const
```

```
{
    for (int i = 0; i < vc.size(); i++)
    {
        cout << vc[i].getName() << "  \t" << vc[i].getArea()
             << "  \t" << vc[i].getPopulation() << "  \t"
             << vc[i].density() << endl;
    }
    cout << endl;
}

// визначення функції-елемента, що виконує
// сортування країн
void Countries::sortCountries()
{
    bool mustSort;
    do
    {
        mustSort = false;
        for (int i = 0; i < vc.size() - 1; i++)
        {
            if (vc[i].getName() > vc[i + 1].getName())
            {
                Country c1;
                c1 = vc[i];
                vc[i] = vc[i + 1];
                vc[i + 1] = c1;
                mustSort = true;
            }
        }
    }
    while (mustSort);
}

// визначення функції-елемента, що виконує
// відбір країн за умовою
void Countries::writeIfGreater(const char* fileName) const
{
    ofstream out(fileName);
    for (int i = 0; i < vc.size(); i++)
    {
        if (vc[i].density() > 100)
        {
            out << vc[i].getName() << "  \t"
                << vc[i].getArea() << "  \t"
                << vc[i].getPopulation()
                << "  \t" << vc[i].density() << endl;
        }
    }
}
```

```
{
    for (int i = 0; i < vc.size(); i++)
    {
        cout << vc[i].getName() << "  \t" << vc[i].getArea()
             << "  \t" << vc[i].getPopulation() << "  \t"
             << vc[i].density() << endl;
    }
    cout << endl;
}

// definition of a member function that
// sorts countries
void Countries::sortCountries()
{
    bool mustSort;
    do
    {
        mustSort = false;
        for (int i = 0; i < vc.size() - 1; i++)
        {
            if (vc[i].getName() > vc[i + 1].getName())
            {
                Country c1;
                c1 = vc[i];
                vc[i] = vc[i + 1];
                vc[i + 1] = c1;
                mustSort = true;
            }
        }
    }
    while (mustSort);
}

// definition of a member function that
// selects countries by criteria
void Countries::writeIfGreater(const char* fileName) const
{
    ofstream out(fileName);
    for (int i = 0; i < vc.size(); i++)
    {
        if (vc[i].density() > 100)
        {
            out << vc[i].getName() << "  \t"
                << vc[i].getArea() << "  \t"
                << vc[i].getPopulation()
                << "  \t" << vc[i].density() << endl;
        }
    }
}
```

Зауважимо, що кількість країн у файлі може бути довільною. Це можливо завдяки застосуванню функції `push_back()`, яка додає країни до вектора. Після заповнення вектора в усіх функціях можна отримувати кількість країн за допомогою функції `size()`. Завдяки можливостям класу `string` під час сортування можна порівнювати назви за допомогою операції > або <.

Тепер до проекту додаємо файл, у якому здійснюється тестування класів:

```
#include <iostream>
#include "CountiesClasses.h"
using std::cin;
using std::cout;
using std::endl;
int main(int argc, char* argv[])
{
    // створення об’єкту класу – Countries
    Countries cs;
    // читання вихідних даних з файлу
    if (cs.readCountries("Countries.txt"))
    {   // вихідні дані прочитані успішно
        // виводимо список країн на екран
        cs.printCountries();
        // сортуємо список країн
        cs.sortCountries();
        // виводимо список країн на екран після сортування
        cs.printCountries();
        // виводимо список країн, що задовольняють
        // вимоги, в файл
        cs.writeIfGreater("SomeCountries.txt");
    }
    else//при читанні вихідних даних виникла помилка
    {
        //виводимо повідомлення про помилку
        cout << "Error reading file!";
    }
    return 0;
}
```

Приклад 2. Приклад генерації та обробки винятків. Припустимо, необхідно спроектувати програму, в якій реалізується приклад програми, наведеної в підрозділі 3.2, та додається механізм успадкування, а також генерації та обробки винятків. Для роботи програми необхідно скористатися файлом `Countries.txt` з прикладу програми підрозділу 3.2.

Note that the number of countries in the file can be arbitrary. This is possible through the use of function `push_back()`, which adds countries to the vector. After filling vector, all functions can obtain the number of countries using the size() function. Due to the capabilities of the `string` class, we can compare names by using operations `>` or `<` when sorting.

Now we add a new file to the project, in which the classes are tested:

```
#include <iostream>
#include "CountiesClasses.h"
using std::cin;
using std::cout;
using std::endl;
int main(int argc, char* argv[])
{
    // create an object class Countries
    Countries cs;
    // reading source data from file
    if (cs.readCountries("Countries.txt"))
    {   // source data are read successfully
        // display a list of countries
        cs.printCountries();
        // sort the list of countries
        cs.sortCountries();
        // display a list after sorting
        cs.printCountries();
        // output to a file countries that
        // meet requirements
        cs.writeIfGreater("SomeCountries.txt");
    }
    else// an error occurred when reading source data
    {
        // display an error
        cout << "Error reading file!";
    }
    return 0;
}
```

Example 2. This is an example of generation and handling of exceptions. Suppose we want to design a program that implements sample application provided in 3.2, extending it by usage of inheritance and exceptions handling. The program needs to use the file `Countries.txt` from the example 3.2.

До заголовного файлу `CountiesClasses.h` додаємо нові класи для генерації винятків – базовий клас `CountriesError` та похідні класи `FileError` та `MathError`:

```
#ifndef CountiesClasses_h
#define CountiesClasses_h
#include <string>
#include <vector>

using std::string;
using std::vector;

class CountriesError { };

class FileError : public CountriesError { };

class MathError : public CountriesError  { };

class Country
{
private:
    string name;
    double area;
    int    population;
public:
    string getName()          const { return name; }
    double getArea()          const { return area; }
    int   getPopulation()     const { return population; }
    void  setName(string value)     { name = value; };
    void  setArea(double value)     { area = value; }
    void  setPopulation(int value) { population = value; }
    double density() const;
};

class Countries
{
private:
    vector<Country> vc;
public:
    void readCountries(const char* fileName);
    void printCountries() const;
    void sortCountries();
    void writeIfGreater(const char* fileName) const;
};

#endif
```

У випадку застосування винятків типом результату функції `readCountries()` може бути **`void`**.

Now we add definition of new exception classes to header file `CountiesClasses.h`. It could be a base class `CountriesError` and derived classes `FileError` and `MathError`:

```
#ifndef CountiesClasses_h
#define CountiesClasses_h
#include <string>
#include <vector>

using std::string;
using std::vector;

class CountriesError { };

class FileError : public CountriesError { };

class MathError : public CountriesError  { };

class Country
{
private:
    string name;
    double area;
    int    population;
public:
    string getName()          const { return name; }
    double getArea()          const { return area; }
    int   getPopulation()     const { return population; }
    void  setName(string value)     { name = value; };
    void  setArea(double value)     { area = value; }
    void  setPopulation(int value) { population = value; }
    double density() const;
};

class Countries
{
private:
    vector<Country> vc;
public:
    void readCountries(const char* fileName);
    void printCountries() const;
    void sortCountries();
    void writeIfGreater(const char* fileName) const;
};

#endif
```

When using exceptions, the result type of function readCountries() may be **`void`**.

Винятки генеруються у функціях `density()` та `readCountries()`. Решта файлу `CountiesClasses.cpp` залишилась без змін.

```
double Country::density() const
{
    if (area <= 0)
        throw MathError();
    return population / area;
}

void Countries::readCountries(const char* fileName)
{
    ifstream in(fileName);
    if (in == 0)
    {
        throw FileError();
    }
    Country c;
    string  name;
    double  area;
    int     population;
    while (in >> name >> area >> population)
    {
        c.setName(name);
        c.setArea(area);
        c.setPopulation(population);
        vc.push_back(c);
    }
}
```

Наведемо два варіанти програми тестування класів. Перший варіант:

```
#include <iostream>
#include "CountiesClasses.h"

using std::cin;
using std::cout;
using std::endl;

int main(int argc, char* argv[])
{
    Countries cs;
    try
    {
        cs.readCountries("Countries.txt");
        cs.printCountries();
        cs.sortCountries();
        cs.printCountries();
```

Exceptions can be thrown in the functions `density()` and `readCountries()`. The rest of the file `CountiesClasses.cpp` remains unchanged.

```
double Country::density() const
{
    if (area <= 0)
        throw MathError();
    return population / area;
}

void Countries::readCountries(const char* fileName)
{
    ifstream in(fileName);
    if (in == 0)
    {
        throw FileError();
    }
    Country c;
    string  name;
    double  area;
    int     population;
    while (in >> name >> area >> population)
    {
        c.setName(name);
        c.setArea(area);
        c.setPopulation(population);
        vc.push_back(c);
    }
}
```

Here are two variants of a program that tests classes. The first way:

```
#include <iostream>
#include "CountiesClasses.h"

using std::cin;
using std::cout;
using std::endl;

int main(int argc, char* argv[])
{
    Countries cs;
    try
    {
        cs.readCountries("Countries.txt");
        cs.printCountries();
        cs.sortCountries();
        cs.printCountries();
```

```
            cs.writeIfGreater("SomeCountries.txt");
        }
        catch (FileError)
        {
            cout << "Error reading file";
        }
        catch (MathError)
        {
            cout << "Error calculating density";
        }
        return 0;
}
```

Другий варіант:

```
#include <iostream>
#include "CountiesClasses.h"

using std::cin;
using std::cout;
using std::endl;

int main(int argc, char* argv[])
{
    Countries cs;
    try
      {
        cs.readCountries("Countries.txt");
        cs.printCountries();
        cs.sortCountries();
        cs.printCountries();
        cs.writeIfGreater("SomeCountries.txt");
    }
    catch (CountriesError)
    {
        cout << "Error";
    }
    return 0;
}
```

Приклад 3. Приклад використання засобів стандартної бібліотеки C++. Припустимо, необхідно спроектувати програму, в якій реалізується приклад програми, яка наведена в підрозділі 3.2, та додається використання засобів стандартної бібліотеки C++.

```
            cs.writeIfGreater("SomeCountries.txt");
        }
        catch (FileError)
        {
            cout << "Error reading file";
        }
        catch (MathError)
        {
            cout << "Error calculating density";
        }
        return 0;
}
```

The second way:

```
#include <iostream>
#include "CountiesClasses.h"

using std::cin;
using std::cout;
using std::endl;

int main(int argc, char* argv[])
{
    Countries cs;
    try
      {
        cs.readCountries("Countries.txt");
        cs.printCountries();
        cs.sortCountries();
        cs.printCountries();
        cs.writeIfGreater("SomeCountries.txt");
    }
    catch (CountriesError)
    {
        cout << "Error";
    }
    return 0;
}
```

Example 3. An example of application the C++ Standard Library. Suppose you want to design a program that implements sample program, which is presented in 3.2, and added the use of the C++ Standard Library.

Для використання рядків стандартної бібліотеки до файлу треба включити заголовний файл `string` та підключити простір імен `std`:

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

Перед функцією `main()` треба розташувати опис класу для подання даних про країну. Він містить відповідні елементи даних (`name`, `area`, `population`), функцію отримання назви країни (`getName()`) та функцію обчислення густоти населення (`density()`):

```
class Country
{
    string name;
    double area;
    int population;
public:
    string getName() const { return name; }
    double density() const { return population / area; }
};
```

Службове слово **`const`** у визначенні функції вказує, що функція не повинна змінювати об'єкт, для якого вона викликана.

Оскільки об'єкти класу повинні записуватись у потік і читатися з потоку, для них доцільно перевантажити відповідні операції. Першим операндом операцій "<<" і ">>" повинен бути об'єкт – потік, а отже, для реалізації перевантаження цих операцій не можна використовувати функції-елементи класу `Country`. Потрібні відповідні глобальні функції, які можна зробити друзями класу `Country`, для того щоб їм були доступні елементи даних:

```
class Country
{
    friend istream& operator>>(istream& in, Country& c);
    friend ostream& operator<<(ostream& out,
                               const Country& c);
private:
    string name;
    double area;
    int population;
public:
    string getName() const { return name; }
    double density() const { return population / area; }
};
```

Типи `ostream` та `istream`, а також інші класи та об'єкти-потоки

To use the strings of the Standard Library, we need to include the header file `string` and connect namespace `std`:

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

Before the `main()` function we place definition of class for representation data on the country. It contains appropriate data members (`name`, `area`, and `population`), the member function of receiving country name (`getName()`) and the function of calculating population density (`density()`):

```
class Country
{
    string name;
    double area;
    int population;
public:
    string getName() const { return name; }
    double density() const { return population / area; }
};
```

The keyword **`const`** in a function definition specifies that the function should not modify the object, for which it is called.

Since the class objects should be recorded in a stream and read from the stream, it is advisable for them to overload the corresponding operations. The first operand of operations "`<<`" and "`>>`" is to be the stream object, and hence to implement the overloading of these operations, we cannot use member functions of class `Country`. We need the appropriate global functions that can be declared as friends of a class `Country`, in order to enable them to get access to data members:

```
class Country
{
    friend istream& operator>>(istream& in, Country& c);
    friend ostream& operator<<(ostream& out,
                               const Country& c);
private:
    string name;
    double area;
    int population;
public:
    string getName() const { return name; }
    double density() const { return population / area; }
};
```

Types `ostream` and `istream`, as well as other stream classes and

стають доступними, якщо підключити заголовний файл `fstream`:

```
#include <fstream>
```

Наведемо реалізацію функцій, що перевантажують операції введення-виведення:

```
istream& operator>>(istream& in, Country& c)
{
    return in >> c.name >> c.area >> c.population;
}

ostream& operator<<(ostream& out, const Country& c)
{
    return out << c.name << '\t' << c.area << '\t'
               << c.population << '\t' << c.density();
}
```

Тепер, якщо ми хочемо ввести з потоку і вивести в інший потік дані про країну, це можна зробити в такий спосіб:

```
Country c1;
cin >> c1;
cout << c1;
```

У такий же спосіб можуть бути використані потоки для роботи з файлами. Для читання з файлу та розташування у векторі даних про країни можна використати такий цикл:

```
vector<Country> c;
Country c1;
ifstream in("Countries.txt");
while (in >> c1)
    c.push_back(c1);
```

Для сортування вектора країн можна використати алгоритм `sort()`. Для визначення ознаки сортування необхідно реалізувати таку функцію:

```
bool sortByAlphabet(Country c1, Country c2)
{
    return c1.getName() < c2.getName();
}
```

Тоді у функції `main()` можна звернутися до алгоритму сортування в такий спосіб:

```
int main()
```

objects, become available by inclusion of the header file `fstream`:

```
#include <fstream>
```

Here is the implementation of functions that overload input and output operations:

```
istream& operator>>(istream& in, Country& c)
{
    return in >> c.name >> c.area >> c.population;
}

ostream& operator<<(ostream& out, const Country& c)
{
    return out << c.name << '\t' << c.area << '\t'
               << c.population << '\t' << c.density();
}
```

Now, if we want to enter from the stream and display to another stream data about the country, it can be done as follows:

```
Country c1;
cin >> c1;
cout << c1;
```

In the same way, streams can be used for files. To read from the file and to store data in the vector of the countries, we can use this cycle:

```
vector<Country> c;
Country c1;
ifstream in("Countries.txt");
while (in >> c1)
    c.push_back(c1);
```

To sort the vector of countries, `sort()` algorithm can be used. To determine the criterion of sorting, it is necessary to implement such a function:

```
bool sortByAlphabet(Country c1, Country c2)
{
    return c1.getName() < c2.getName();
}
```

Now we can invoke sorting algorithm from the `main()` function:

```
int main()
```

```
{
    vector<Country> c;
    . . .
    // читання даних
    . . .
    sort(c.begin(), c.end(), sortByAlphabet);
    . . .
    // виведення даних
    . . .
}
```

Для виведення вектора на екран можна використати алгоритм `for_each()`, для якого треба додатково реалізувати функцію `writeCountry()`:

```
void writeCountry(const Country& c1)
{
    cout << c1 << endl;
}

int main()
{
    . . .
    for_each(c.begin(), c.end(), writeCountry);
    . . .
}
```

Якби в попередньому прикладі ми хотіли б виводити результати не в стандартний потік `cout`, а в деякий довільний потік (наприклад, файл), то посилання на цей потік потрібно було б передати у функції `writeCountry()` другим параметром. Але це неможливо, оскільки виклик функції здійснюється зсередини недоступного нам коду алгоритму `for_each()`. У цьому випадку треба застосовувати функціональні об'єкти. В описі відповідного класу доцільно як елемент-даних описати посилання на потік. Його можна використовувати при виведенні даних у потік. У такий спосіб ми отримаємо опис класу:

```
class WriteCountry
{
    ostream& out;
public:
    WriteCountry(ostream& strm) : out(strm) { }
    void operator()(Country& c) { out << c << endl; }
};
```

```
{
    vector<Country> c;
    . . .
    // reading the data
    . . .
    sort(c.begin(), c.end(), sortByAlphabet);
    . . .
    // output
    . . .
}
```

To display a vector on the screen, we can apply an algorithm `for_each()`, for which it is necessary to implement additional function `writeCountry()`:

```
void writeCountry(const Country& c1)
{
    cout << c1 << endl;
}

int main()
{
    . . .
    for_each(c.begin(), c.end(), writeCountry);
    . . .
}
```

If in the previous example, we would like to transfer the results not to standard output stream `cout`, but to some another stream (e.g. a file), a reference to the stream would be necessary to pass to the function `writeCountry()` as the second parameter. However, this is impossible because the function call is made in algorithm `for_each()` inaccessible within our code. In this case, it is necessary to use functional objects . In the description of the relevant class, it is advisable to describe the reference to the stream as data member. It can be used in output data stream. Thus, we get the description of the class:

```
class WriteCountry
{
    ostream& out;
public:
    WriteCountry(ostream& strm) : out(strm) { }
    void operator()(Country& c) { out << c << endl; }
};
```

Ініціалізація елементів-даних типу посилань у конструкторі повинна здійснюватися через список ініціалізації. При виклику алгоритму `for_each()` третім параметром можна вказати ім'я конструктора класу `WriteCountry` з передачею необхідного параметра (імені потоку):

```
ofstream outFile("Result.txt");
for_each(c.begin(), c.end(), WriteCountry(outFile));
```

Якщо у вихідний потік потрібно заносити дані тільки про ті об'єкти, для яких виконується задана умова (наприклад, периметр більше заданої величини), дані, необхідні для такої перевірки, також можуть бути елементами даних класу та ініціалізуватись при виклику конструктора.

```
class WriteIfGreaterThan
{
    ostream& out;
    double value;
public:
    WriteIfGreaterThan(ostream& strm, double newValue)
                        : value(newValue), out(strm) { }
    void operator()(Country& c)
    {
        if (c.density() > value)
            out << c << endl;
    }
};
```

Звертання до алгоритму `for_each()` при цьому може здійснюватися в такий спосіб:

```
for_each(c.begin(), c.end(),
         WriteIfGreaterThan(outFile, 100));
```

Тепер можна навести весь текст програми:

```
#include <string>
#include <vector>
#include <algorithm>
#include <fstream>
#include <iostream>

using namespace std;
class Country
{
    friend istream& operator>>(istream& in,  Country& c);
    friend ostream& operator<<(ostream& out,
                               const Country& c);
```

The initialization of reference type member in the constructor must be provided through the initialization list. When we call the algorithm `for_each()`, as the third parameter of it, the name of the class constructor `WriteCountry` with the stream name as an argument should be specified:

```
ofstream outFile("Result.txt");
for_each(c.begin(), c.end(), WriteCountry(outFile));
```

If we need to store only those objects in the output stream, for which the specified condition (e.g., perimeter is more than a given value) is true, the data necessary for such verification can also be defined as data members and initialized when the constructor being called.

```
class WriteIfGreaterThan
{
    ostream& out;
    double value;
public:
    WriteIfGreaterThan(ostream& strm, double newValue)
                        : value(newValue), out(strm) { }
    void operator()(Country& c)
    {
        if (c.density() > value)
            out << c << endl;
    }
};
```

Invocation of `for_each()` algorithm may be carried out in such a way:

```
for_each(c.begin(), c.end(),
         WriteIfGreaterThan(outFile, 100));
```

Here is the whole text of the program:

```
#include <string>
#include <vector>
#include <algorithm>
#include <fstream>
#include <iostream>

using namespace std;
class Country
{
    friend istream& operator>>(istream& in,  Country& c);
    friend ostream& operator<<(ostream& out,
                               const Country& c);
```

```
private:
    string name;
    double area;
    int population;
public:
    string getName() const { return name; }
    double density() const { return population / area; }
};


istream& operator>>(istream& in,  Country& c)
{
    return in >> c.name >> c.area >> c.population;
}


ostream& operator<<(ostream& out, const Country& c)
{
    return out << c.name << '\t' << c.area << '\t'
               << c.population << '\t' << c.density();
}


bool sortByAlphabet(Country c1, Country c2)
{
    return c1.getName() < c2.getName();
}


void writeCountry(const Country& c1)
{
    cout << c1 << endl;
}


class WriteIfGreaterThan
{
    ostream& out;
    double value;
public:
    WriteIfGreaterThan(ostream& strm, double newValue)
        : value(newValue), out(strm) { }
    void operator()(const Country& c)
    {
        if (c.density() > value)
        {
            out << c << endl;
        }
```

```
private:
    string name;
    double area;
    int population;
public:
    string getName() const { return name; }
    double density() const { return population / area; }
};


istream& operator>>(istream& in,  Country& c)
{
    return in >> c.name >> c.area >> c.population;
}


ostream& operator<<(ostream& out, const Country& c)
{
    return out << c.name << '\t' << c.area << '\t'
               << c.population << '\t' << c.density();
}


bool sortByAlphabet(Country c1, Country c2)
{
    return c1.getName() < c2.getName();
}


void writeCountry(const Country& c1)
{
    cout << c1 << endl;
}


class WriteIfGreaterThan
{
    ostream& out;
    double value;
public:
    WriteIfGreaterThan(ostream& strm, double newValue)
        : value(newValue), out(strm) { }
    void operator()(const Country& c)
    {
        if (c.density() > value)
        {
            out << c << endl;
        }
```

```
    }
};


int main(int argc, char* argv[])
{
    vector<Country> c;
    Country c1;
    ifstream in("Countries.txt");
    while (in >> c1)
    {
      c.push_back(c1);
    }
    sort(c.begin(), c.end(), sortByAlphabet);
    for_each(c.begin(), c.end(), writeCountry);
    cout << endl;
    for_each(c.begin(), c.end(),
             WriteIfGreaterThan(cout, 100));
    ofstream outFile("Result.txt");
    for_each(c.begin(), c.end(),
             WriteIfGreaterThan(outFile, 100));
    cin.get();
    return 0;
}
```

3.3.14. Вправа для самостійної роботи

Створити консольну програму, яка складається з двох одиниць трансляції. Для цього необхідно створити один заголовний файл та два файли реалізації. У заголовному файлі (з розширенням `.h`) слід розмістити стражів включення та описати клас для подання даних про студента, у якому описати такі закриті (`private`) елементи даних:

– номер залікової книжки (ціле значення);

– прізвище (рядок типу `std::string`);

– оцінки за останню сесію (масив з п'яти цілих значень).

Усередині класу реалізувати функції доступу, а також оголосити конструктор, який ініціалізує елементи-дані, та необхідні функції для роботи зі студентом згідно з практичним завданням п.3.2.5.

У цьому ж заголовному файлі здійснити опис класу для представлення групи студентів. В об'єкті такого класу повинні зберігатись дані про студентів у вигляді масиву об'єктів класу, який представляє студента. Масив розмістити в динамічній пам'яті. В окремому елементі-даному слід зберігати розмір масиву. Клас повинен містити конструктор, у якому визначається кількість студентів, деструктор, а також функції-елементи, які здійснюють:

}

```
    }
};


int main(int argc, char* argv[])
{
    vector<Country> c;
    Country c1;
    ifstream in("Countries.txt");
    while (in >> c1)
    {
      c.push_back(c1);
    }
    sort(c.begin(), c.end(), sortByAlphabet);
    for_each(c.begin(), c.end(), writeCountry);
    cout << endl;
    for_each(c.begin(), c.end(),
             WriteIfGreaterThan(cout, 100));
    ofstream outFile("Result.txt");
    for_each(c.begin(), c.end(),
             WriteIfGreaterThan(outFile, 100));
    cin.get();
    return 0;
}
```

3.3.14. Exercise for self-study

Create a console application, which consists of two translation units. To do this, create a header file and two implementation files.

The header file (with extension `.h`) should contain inclusion guards and a class definition for representation data on the student, which includes such private data members:

– a number of the record book (integer);

– the surname (string of type `std::string`);

– grades of the previous semester exams (an array of five integers) .

The class should also contain the following members: access functions, a constructor that initializes data members, necessary functions for working with the student according to the practical task 3.2.5.

In the same header file, you should implement definition of a class that represents a group of students. The object of this class will store data on students as an array of student type objects. The array should be placed in the heap.

You should store the size of the array in a separate data member. The class should include a constructor, which determines the number of students, destructor and the following member functions:

– читання даних про студентів з текстового файлу;

– сортування масиву за ознакою, яка наведена в практичному завданні п.3.2.5;

– виведення даних про студентів на екран;

– пошук даних про студентів, які відповідають умові, наведеній в практичному завданні п.3.2.5, та виведення даних про цих студентів у новий файл.

У файлі реалізації, назва якого відрізняється від заголовного файлу розширенням (`.cpp`), необхідно підключити заголовний файл та здійснити реалізацію функцій-елементів обох класів.

У файлі реалізації підключити заголовний файл та здійснити тестування функцій-елементів класу, який представляє групу.

Передбачити використання механізму генерації та обробки винятків. Обов’язково використати перевантаження операцій та засоби стандартної бібліотеки C++. Для створеного класу перевантажити операції читання з потоку та занесення у потік. Для сортування використовувати алгоритм `sort()`. Для визначення умови сортування використовувати функціональні об'єкти. Для виведення використовувати алгоритм `for_each()` та функціональні об'єкти.

Контрольні запитання

1. Поняття класу.
2. Створення об'єктів класів.
3. Конструктори та деструктори.
4. Конструктор копіювання.
5. Функції доступу.
6. Константні функції-елементи.
7. Використання специфікатора **`mutable`**.
8. Реалізація функцій-елементів у тілі класу та поза тілом класу.
9. Використання вказівника **`this`**.
10. Область видимості класу.
11. Статичні елементи опису класу.
12. Вкладені класи.
13. Друзі класу.
14. Використання векторів стандартної бібліотеки.
15. Використання рядків стандартної бібліотеки
16. Композиція класів.
17. Базові та похідні класи.

– reading data on students from a text file;
– sorting an array on the criteria given in practical task 3.2.5 ;
– printing data on students onto the screen;
– search for data on students who meet the conditions outlined in the practical task 3.2.5 and storing data on these students in a new file .

In the implementation file, which name differs from the header file name only in its extension (`.cpp`), you should include the header file and provide the implementation of member functions.

In another implementation file, which includes your header file, you should test member function of the class that represents the group.

Use the mechanism of generation and processing exceptions. The usage of operator overloading and means of the C++ Standard Library is obligatory. You should overload the stream operations. To sort data, use `sort()` algorithm. To determine the conditions for sorting, use functional objects. To output, use an algorithm `for_each()` and functional objects.

Advancement Questions

1. The concept of class.
2. Creation of objects.
3. Constructors and destructors.
4. Copy constructor.
5. Access functions.
6. Constant member functions.
7. Usage of **`mutable`** specifier.
8. Implementation of member functions within and outside class body.
9. Usage of **`this`** pointer.
10. Class scope.
11. Static class members.
12. Nested classes.
13. Class friends.
14. Use vectors of the standard library.
15. Use strings of the standard library
16. Composition of classes.
17. Base and derived classes.

18. Порядок виклику конструкторів та деструкторів базових та похідних класів.
19. Поліморфізм.
20. Генерація виключних ситуацій.
21. Перехоплення винятків.
22. Перевантаження операцій.
23. Вимоги до операторної функції.
24. Склад Стандартної бібліотеки C++.
25. Ітератори.
26. Алгоритми Стандартної бібліотеки.

18. Order of calling constructors and destructors of base and derived classes.
19. Polymorphism.
20. Throwing exceptions.
21. Catching exceptions.
22. Operator overloading.
23. Requirements for operator functions.
24. The structure of Standard C++ Library.
25. Iterators.
26. Algorithms of Standard Library.

СПИСОК ЛІТЕРАТУРИ

1. Брайан Джонсон, Крэйг Скибо, Марк Янг. Основы Microsift Visual Studio .NET / Брайан Джонсон, Крэйг Скибо, Марк Янг. – М.: Русская Редакция, 2003. – 512 с.

2. Давыдов В.Г. Программирование и основы алгоритмизации / В.Г. Давыдов. – М.: Высш. школа, 2003. – 447 с.

3. Кватрани Т. Rational Rose 2000 и UML. Визуальное моделирование / Т. Кватрани. – М.: ДМК Пресс, 2001. – 176 с.

4. Леоненков А.В. Самоучитель UML / А.В. Леоненков. – С.Пб.: БХВ, 2004. – 361 с.

5. Либерти Д., Хорват Д. Освой самостоятельно C++ за 24 часа / Д. Либерти, Д. Хорват. – М.: Вильямс, 2007. – 448 с.

6. Пахомов Б.И. C/C++ и MS Visual C++ 2010 для начинающих / Б.И. Пахомов. – СПб.: БХВ-Петербург, 2011. – 736 с.

7. Стенли Б. Липпман, Жози Лажойе, Барбара Му. Язык программирования C++. Вводный курс / Стенли Б. Липпман, Жози Лажойе, Барбара Му. – М.: Вильямс, 2007.

8. Фаулер М., Скотт К. UML. Основы / М. Фаулер, К. Скотт. – СПб.: Символ-Плюс, 2002. – 192 с.

9. Эккель Б. Философия C++. Введение в стандартный C++ / Б. Эккель. – СПб.: Питер, 2004. – 572 с.

10. http3. ://algolist.manual.ru/sort/bubble_sort.php сортування елементів масиву за методом бульбашки, 22.10.2013.

11. Bjarne Stroustrup. The C++ Programming Language. Third Edition. - Addison-Wesley, 1997.

12. Stanley B. Lippman, Josee Lajoie C++ Primer. Third Edition. - Addison-Wesley, 1988.

13. H. M. Deitel, P. J. Deitel. C++. How to Program. Third Edition. - Prentice Hall, 2001.

14. http://www.stroustrup.com/C++.html The C++ Programming Language (Bjarne Stroustrup's homepage)

BIBLIOGRAPHY

1. Брайан Джонсон, Крэйг Скибо, Марк Янг. Основы Microsift Visual Studio .NET / Брайан Джонсон, Крэйг Скибо, Марк Янг. – М.: Русская Редакция, 2003. – 512 с.

2. Давыдов В.Г. Программирование и основы алгоритмизации / В.Г. Давыдов. – М.: Высш. школа, 2003. – 447 с.

3. Кватрани Т. Rational Rose 2000 и UML. Визуальное моделирование / Т. Кватрани. – М.: ДМК Пресс, 2001. – 176 с.

4. Леоненков А.В. Самоучитель UML / А.В. Леоненков. – С.Пб.: БХВ, 2004. – 361 с.

5. Либерти Д., Хорват Д. Освой самостоятельно C++ за 24 часа / Д. Либерти, Д. Хорват. – М.: Вильямс, 2007. – 448 с.

6. Пахомов Б.И. C/C++ и MS Visual C++ 2010 для начинающих / Б.И. Пахомов. – СПб.: БХВ-Петербург, 2011. – 736 с.

7. Стенли Б. Липпман, Жози Лажойе, Барбара Му. Язык программирования C++. Вводный курс / Стенли Б. Липпман, Жози Лажойе, Барбара Му. – М.: Вильямс, 2007.

8. Фаулер М., Скотт К. UML. Основы / М. Фаулер, К. Скотт. – СПб.: Символ-Плюс, 2002. – 192 с.

9. Эккель Б. Философия C++. Введение в стандартный C++ / Б. Эккель. – СПб.: Питер, 2004. – 572 с.

10. http://algolist.manual.ru/sort/bubble_sort.php сортування елементів масиву за методом бульбашки, 22.10.2013.

11. Bjarne Stroustrup. The C++ Programming Language. Third Edition. - Addison-Wesley, 1997.

12. Stanley B. Lippman, Josee Lajoie C++ Primer. Third Edition. - Addison-Wesley, 1988.

13. H. M. Deitel, P. J. Deitel. C++. How to Program. Third Edition. - Prentice Hall, 2001.

14. http://www.stroustrup.com/C++.html The C++ Programming Language (Bjarne Stroustrup's homepage)

15. http://cs.nyu.edu/courses/summer12/CSCI-GA.2110-001/downloads/C++%20Standard%202003.pdf ISO/IEC 14882:2003 Programming languages - C++ (International Standard)

16. http://www.cplusplus.com The C++ Resources Network // { /}

14. http://www.learncpp.com The C++ Tutorial

15. http://cs.nyu.edu/courses/summer12/CSCI-GA.2110-001/downloads/C++%20Standard%202003.pdf ISO/IEC 14882:2003 Programming languages - C++ (International Standard)

16. http://www.cplusplus.com The C++ Resources Network // { /}

14. http://www.learncpp.com The C++ Tutorial

ПРЕДМЕТНИЙ ПОКАЗЧИК

INDEX