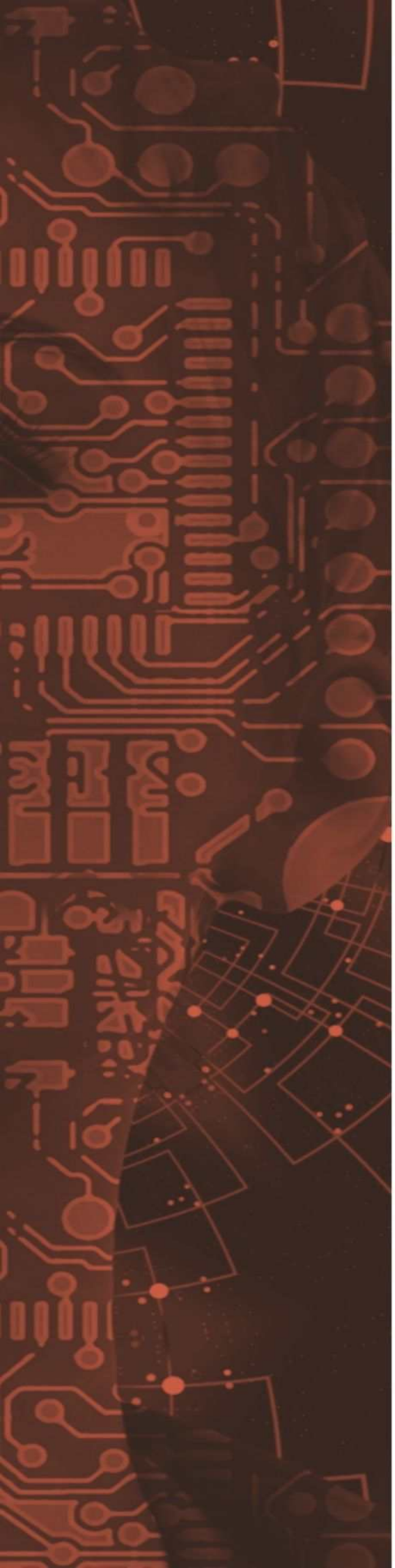

C++

Programming

A Step-by-Step Guide to Learn, in an Easy Way, the Fundamentals of C++ Programming Language

By John Bach

Programming ++ C

A Step-by-Step Guide to Learn, in an Easy Way, the Fundamentals of C++ Programming Language

Edition st 1

2020

By
Alexander Aronowitz

For information contact :
(alabamamond@gmail.com, memlnc)
http://www.memlnc.com

First Edition: 2020

"Programming isn't about what you know; it's about what you can figure Chris Pine out." -

Table of contents

FOREWORD

As promised in the first edition of the book, user requests have shaped the development of C ++. He was guided by the experience of a wide range of users working in different areas of programming. In the six years separating us from the first edition of the C ++ description, the number of users has increased hundreds of times. Over the years, many lessons have been learned, various programming techniques have been proposed and confirmed by practice. On some of them will be discussed below.

The extensions of the language made over these six years were primarily aimed at increasing the expressiveness of C ++ as a data abstraction language and object-oriented programming in general, and as a means for creating high-quality libraries with user-defined data types in

particular. We consider a library of high quality as a library that allows the user to define concepts using classes that combine convenience, efficiency, and reliability. Under the reliability meant that the class provides a secure interface between the types of users of the library and its developers. Efficiency assumes that using classes does not incur large memory or time overheads compared to "manual" C programs.

This book is a complete description of the C ++ language. Chapters 1 through 10 are a textbook introducing the language. Chapters 11 through 13 discuss software design and development. The book ends with a reference guide to the C ++ language. Naturally, all the extensions of the language and the ways of using them, which appeared after the publication of the first edition, are part of the presentation. These include refined rules for name overload resolution, memory and access controls, type-safe binding, static and constant member functions, abstract classes, multiple inheritance, type templates, and exception handling.

C ++ is a general-purpose programming language. Its natural field of application is system programming, understood in the broad sense of the word. In addition, C ++ has been successfully used in many areas of the application, far beyond the specified scope. C ++ implementations are now found on all machines, from the humblest microcomputer to the largest supercomputer, to virtually all operating systems. Therefore, the book gives only a description of the language itself, without explaining the features of specific implementations, programming environment or libraries.

The reader will find in the book many examples with classes, which, despite their undoubted benefits, can be considered toys. This style of presentation allows you to better highlight the basic concepts and useful techniques, whereas in real, complete programs, they would be hidden by a lot of details. For most of the classes proposed here, such as linked lists, arrays, character strings, matrices, graphic classes, associative arrays, etc., versions of "100% guaranteed" reliability and correctness are provided , obtained from classes from a wide variety of commercial and non-profit programs. Many of the "industrial" classes and libraries are derived directly or indirectly from the toy classes shown here as examples.

In this edition of the book, in comparison with the first, more attention is paid to the learning task. At the same time, the level of presentation also takes into account experienced programmers, in no way detracting from their knowledge and professionalism. The discussion of design issues is accompanied by a broader presentation of the material, going beyond the descriptions of language constructs and how they are used. This edition contains more technical details and increased rigor. This applies in particular to the reference manual, which has incorporated many years of experience in this area. It was supposed to create a book with a sufficiently high level of presentation, which would serve programmers not only as a book to read. So, before you is a book describing the C ++ language, its basic principles and programming methods. We hope you enjoy it.

FOREWORD TO THE FIRST EDITION

"Language forms the environment of thinking and forms the idea of what we think about."
(B.L. Wharf)

C ++ is a general-purpose language and is designed to make real programmers enjoy the programming process itself. Except for minor details, it contains the C language as a subset. The C language is being extended by the introduction of flexible and efficient means for constructing new types. The programmer structures his task by defining new types that exactly correspond to the concepts of the problem domain . This method of building a program is commonly referred to as data abstraction. Type information is contained in some objects of user-defined types. You can work with such objects reliably and simply even in those cases when their type cannot be established at the stage of translation. Programming using such objects is commonly referred to as object-oriented. If this method is applied correctly, programs become shorter, easier to understand, and easier to maintain.

The key concept in C ++ is class. A class is a user- defined type. Classes provide data hiding, initialization, implicit conversion of user-defined types, dynamic type assignment, user-controlled memory management, and facilities for overloading operations. In the C ++ language, the concepts of type control and modular construction of programs are implemented more fully than in C. In addition, C ++ contains improvements that are not directly related to classes: symbolic constants, substitution functions, standard values of function parameters, overloading of function names , free memory management operations and reference type. In C ++, all the capabilities of C are preserved for effective work with basic objects that reflect hardware "reality" (bits, bytes, words, addresses, etc.). This allows for a fairly efficient implementation of custom types.

Both the language and the C ++ standard libraries were designed with portability in mind. Existing implementations of the language will work on most systems that support C. C ++ programs can use C libraries. Most C-based utilities can also be used in C ++.

This book is primarily intended for professional programmers who want to learn a new language and use it for non-trivial tasks. The book provides a complete description of C ++, contains many complete examples and even more program fragments.

Acknowledgments

The C ++ language would never have become a reality without it, if suggestions and advice were not constantly used and constructive criticism from many friends and colleagues was not taken into account . Especially worth mentioning are Tom Cargill, Jim Copley, Stu Feldman, Sandy Fraser, Steve Johnson, Brian Kernighan, Bart Locanti, Doug McIlroy, Dennis Ritchie, Larry Rosler, Jerry Schwartz, and John Shapiro, who contributed important ideas to the development of the language. Dave Presotto has implemented the current version of the Stream I / O library .

Hundreds of people contributed to the development of C ++ and the creation of the translator, who sent me suggestions for improving the language, descriptions of the difficulties they encountered, and errors in the translator. I can only mention a few of them here: Gary Bishop, Andrew Hume, Tom Carces, Victor Milenkovich, Rob Murray, Leonie Ross, Brian Schmalt and Harry Walker.

Many contributed to the preparation of the book for publication, especially John Bentley, Laura Ives, Brian Kernighan, Ted Kowalski, Steve Mahani, John Shapiro, and participants in the C ++ seminar held by Bell Labs in Columbia, Ohio, June 26-27, 1985 g.

Murray Hill, New Jersey Bjorn Stroustrup

PRELIMINARY REMARKS

"About many things - said the Walrus, - it's time to talk."
L. Carroll

This chapter contains an overview of the book, a bibliography, and some additional notes about the C ++ language. The notes relate to the history of C ++, ideas that had a significant impact on the development of the language, and some thoughts about programming in C ++. This chapter is not an introduction; these notes are not necessary for understanding the following chapters. Some of them assume that the reader is familiar with C ++.

Book structure

The book is divided into three parts. Chapters 1 through 10 are a language textbook . Chapters 11 through 13 discuss design and development of software with C ++ in mind. There is a complete reference guide to the language at the end of the book . An exhaustive description of C ++ constructs is contained only there. The tutorial portion of the book contains examples, tips, warnings, and exercises for which there was no room in the manual.

The book is mainly devoted to the question of how to structure a program using the C ++ language, and not to the question of how to write an algorithm in it. Therefore, where it was possible to choose, preference was given not to professional, but difficult to understand, but to trivial algorithms. For example, one example uses bubble sort, although the quicksort algorithm is more suitable for a real program. Often writing the same program, but with a more efficient algorithm, is suggested as an exercise.

Chapter 1 provides an overview of basic C ++ concepts and constructs. It allows you to get to know the language in general terms. Detailed explanations of the language constructs and how they are used are contained in the following chapters. First of all, the means of providing data abstraction and object-oriented programming are discussed . Basic procedural programming tools are briefly mentioned.

Chapters 2, 3, and 4 describe C ++ features that are not used to define new types: basic types, expressions, and control structures . In other words, these chapters contain a description of the part of the language that essentially represents C. The presentation in these chapters is in -depth.

Chapters 5 - 8 are devoted to the means of constructing new types that have no analogues in C. Chapter 5 introduces the basic concept - a class. It shows how you can define custom types (classes), initialize them, access them, and finally, how to destroy them. Chapter 6 is devoted to the concept of derived classes, which allows you to build more complex ones from simple classes. It also makes it possible to work efficiently and safely (in terms of type) in situations where the types of objects at the translation stage are unknown. Chapter 7 explains how you can define unary and binary operations on custom types, how to define conversions to those types, and how you can create, copy, and delete objects that represent custom types. Chapter 8 deals with type templates, i.e. a C ++ facility that allows you to define a family of types and functions.

Chapter 9 discusses exception handling, possible responses to errors, and methods for constructing error robust systems. Chapter 10 defines the ostream and istream classes provided by the standard library for streaming I / O.

Chapters 11-13 are devoted to topics related to the use of C ++ for the design and implementation of large software systems. Chapter 11 focuses on the design and management of software projects. Chapter 12 discusses the relationship between the C ++ language and design issues. Chapter 13 shows you how to create libraries.

The book ends with the C ++ Reference Guide.

References to the various parts of the book are given in the form $$ 2.3.4, which means section 3.4 of chapter 2. The letter R is used to refer to the reference manual , eg $$ R.8.5.5.

Implementation notes

There are several redistributable independent C ++ implementations. A large number of service programs, libraries and integrated programming systems appeared. There are tons of books, tutorials, magazines, articles, e-mails, technical bulletins, conference reports and courses from which you can get all the information you need about the latest changes in C ++, its use, tools , libraries, new translators, and etc. If you are serious about C ++, it is worth accessing at least two sources of information, as each source can have a different position.

Most of the program fragments given in the book are taken directly from the program texts that were translated on a DEC VAX 11/8550 machine running UNIX version 10 [25]. The translator used was a direct descendant of the C ++ translator created by the author. It describes "pure C ++", i.e. do not use any implementation-specific extensions. Consequently, examples should come with any language implementation. However, type templates and exception handling are among the most recent extensions to the language, and it is possible that your translator does not contain them.

Exercises

Exercises are given at the end of each chapter. Most often they suggest writing a program. The solution can be considered a program that is broadcast and works correctly on at least several tests. Exercises can vary significantly in difficulty, therefore a rough estimate of the degree of difficulty is given. The increase in difficulty is exponential, so if exercise (* 1) takes you five minutes, then (* 2) may take an hour, and (* 3) may take a whole day. However, the time to write and debug a program depends more on the experience of the reader than on the exercise itself. Exercise (* 1) can take a whole day if the reader has to become familiar with a new computing system before running the program. On the other hand, someone who has the right set of programs at hand can do the exercise (* 5) in one hour.

Any book on C programming can be used as a source of additional exercise for Chapters 2 through 4. Aho's book [1] provides many general data structures and algorithms in terms of abstract data types. This book can also be used as a resource for exercises in chapters 5 through 8. However, the language used in this book lacks member functions and derived classes. Therefore, C ++ user-defined types can be written more elegantly.

Language draft notes

When developing the C ++ language, one of the most important selection criteria was simplicity. When the question arose of what to simplify: the language manual and other documentation or the translator, the choice was made in favor of the former. Great importance was attached to compatibility with the C language, which prevented the removal of its syntax.

There are no high-level datatypes and elementary operations in C ++. For example, there is no type matrix with an inversion operation or a string type with a concatenation operation. If the user needs these types, he can define them in the language itself. C ++ programming essentially boils down to defining generic or application domain types. A well-thought-out custom type differs from a built-in type only in the way it is defined, not in the way it is applied.

Features were excluded from the language that could result in memory or runtime overhead, even if not directly used in the program. For example, a proposal to store some service information in each object was rejected. If the user has described a structure that contains two values, each 16 bits each, it is guaranteed that it will fit into a 32-bit register.

The C ++ language was designed for use in a fairly traditional environment, namely, in the C programming system of the UNIX operating system. But there are good reasons for using C ++ in a richer programming environment. Features such as dynamic loading, advanced translation systems, and databases for storing type definitions can be used successfully without compromising the language.

C ++ types and data hiding mechanisms rely on specific parsing by the translator to detect accidental data corruption. They do not provide data privacy and protection against deliberate violation of access rules. However, these tools can be freely used without fear of memory overhead and program execution time. It is taken into account that the construction of a language is actively used when it is not only elegantly written in it, but also quite within the means of ordinary programs.

Historical reference

Of course, C ++ owes a lot to C [8], which is preserved as a subset of it. All the low-level tools inherent in C are also preserved, designed to solve the most urgent problems of system programming. C, in turn, owes a lot to its predecessor, BCPL [13]. The BCPL language comment has been restored to C ++. If the reader is familiar with the BCPL language, you may notice that C ++ still does not have a VALOF block. Another source of inspiration was the language SIMULA-67 [2,3]; it was from this that the concept of classes was borrowed (along with derived classes and virtual functions). The inspect statement from SIMULA-67 was deliberately not included in C ++.

The reason is the desire to promote modularity through the use of virtual functions. The possibility of overloading operations in C ++ and the freedom to place descriptions wherever an operator can occur are reminiscent of the Algol-68 language [24].

Since the first edition of this book was published, C ++ has undergone significant changes and refinements. It mainly deals with disambiguation on overloading, binding and memory management. However, minor changes were made to increase compatibility with the C language. Several generalizations and significant extensions were also introduced, such as: multiple inheritance, member functions with static and const specifications, protected members, type templates and handling special situations. All of these extensions and improvements were aimed at making C ++ a language in which libraries can be created and used. All changes are described in [10,18,20,21 and 23].

Type templates appeared partly out of a desire to formalize macro-tools, and partly were inspired by the description of generic objects in the Ada language (taking into account their advantages and disadvantages) and parameterized modules of the CLU language. The exception handling mechanism emerged partly under the influence of the Ada and CLU languages [11], and partly under the influence of ML [26]. Other extensions introduced between 1985 and 1991 (such as multiple inheritance, static member functions, and pure virtual functions) are more of a generalization of C ++ programming experience than they were gleaned from other languages.

Earlier versions of the language, called "C with Classes" [16], have been in use since 1980. This language arose because the author needed to write interrupt-driven simulation programs. The SIMULA-67 language is ideal for this, if efficiency is not considered . The C with Classes language was used for large modeling tasks . Then the possibility of writing programs on it, for which time and memory resources are critical, were subjected to a rigorous test . This language lacked overloading of operations, references, virtual functions, and many other features. For the first time, C ++ went beyond the research group in which the author worked in July 1983, but then many C ++ features had not yet been developed.

The name C ++ (ci plus plus), was coined by Rick Mascitti in the summer of 1983. This name reflects the evolutionary nature of changes in the language C. The designation ++ refers to the C augmentation operation. The slightly shorter name C + is a syntax error. In addition, it has already been used as the name of a completely different language. Connoisseurs of C semantics find that C ++ is worse than ++ C. The language was not named D because it is an extension of C, and it does not attempt to solve any problems at the expense of abandoning the capabilities of C. Another interesting interpretation of the name C ++ can be found in the appendix to [12].

C ++ was originally conceived so that the author and his friends did not need to program in assembly, C, or other modern high-level languages . Its main purpose is to simplify and make the programming process more pleasant for the individual programmer. Until recently, there was no paper C ++ development plan. The design, implementation and documentation went in parallel. There has never been a "C ++ Project" or a "C ++ Development Committee". Therefore, the language has evolved and continues to evolve in such a way as to overcome all the problems faced by users. The author's discussions with his friends and colleagues also serve as impulses for development .

Due to the avalanche process of increasing the number of C ++ users, the following changes had to be made. Around 1987, it became apparent that work to standardize C ++ was inevitable and that the foundation for it should begin immediately [22]. As a result, deliberate action was taken to establish contact between C ++ developers and the majority of users. Mail and email were used , and there was direct communication at C ++ conferences and other meetings.

AT&T Bell Laboratories has been a major contributor to this work by granting the author the right to study versions of the language reference manual with the developers and users mentioned. This contribution should not be underestimated as many of them work for companies that can be considered competitors of AT&T. A less enlightened company could simply do nothing, and the result would be several inconsistent versions of the language. About one hundred representatives from about 20 organizations reviewed and commented on what has become the current version of the reference manual and source material for ANSI on C ++ standardization. Their names can be found in the Annotated C ++ Language Reference [4]. The entire Reference Guide is included in this book. Finally, on the initiative of Hewlett-Packard in December 1989, the X3J16 committee was formed within ANSI. The work on C ++ standardization in ANSI (American Standard) is expected to be part of the work on standardization by ISO (International Organization for Standardization).

C ++ has evolved along with the development of some of the fundamental classes presented in this book. For example, the author developed the complex, vector, and stack classes while simultaneously creating the ability to overload operations. As a result of the same efforts and thanks to the assistance of D. Shapiro, string and list classes appeared. These classes were the first library classes to be actively used. The task library described in [19] and in Exercise 13 of $$ 6.8 became part of the very first program written in C with Classes. This program and the classes it uses were created for Simula-style modeling. The task library was significantly revised by D. Shapiro and continues to be actively used to this day. The streaming library, as stated in the first edition of the book, was developed and applied by the author. D. Schwartz transformed it into a stream I / O library ($$ 10), using, along with other techniques, the method of manipulators by E. Koenig ($$ 10.4.2). The map class ($$ 8.8) was proposed by E. Koenig. He also created the Pool class ($$ 13.10) to use the author's way of allocating memory for classes ($$ 5.5.6) for the library . The rest of the templates were influenced by the Vector, Map, Slist, and sort templates introduced in Chapter 8.

Comparison of C ++ and C languages

The choice of C as the base language for C ++ is explained by its following advantages:

(1) versatility, brevity and relatively low level;

(2) adequacy to most system programming tasks;

(3) it goes on any system and on any machine;

(4) fully suited for UNIX software environment.

There are problems in C, but in a language developed from scratch they would appear too, and C problems are at least well known. More importantly, the focus on C allowed "C with classes" to be used as a useful (albeit not very convenient) tool during the first months of thinking about introducing Simula-style classes into C.

C ++ became more widely used, but as its capabilities grew beyond C, the problem of compatibility arose again and again. It is clear that by rejecting part of the C inheritance, one can avoid some problems (see, for example, [15]). This was not done for the following reasons:

(1) there are millions of lines of C programs that can be improved with C ++, provided that a complete rewrite of them into C ++ is not required;

(2) there are millions of lines of C library functions and utility programs that could be used in C ++ if both languages are compatible at the linking stage and their syntactic similarity is strong ;

(3) there are hundreds of thousands of C programmers; it is enough for them to master only new C ++ tools and do not need to learn the basics of the language;

(4) since C and C ++ will be used by the same people on the same systems for many years, differences between languages should be either minimal or maximal to minimize errors and misunderstandings. The C ++ description has been reworked to ensure that any construct allowed in both languages means the same thing in both languages .

The C language itself has evolved over the past few years, which was partly due to the development of C ++ [14]. The ANSI standard for C [27] contains, for example, a syntax for describing functions borrowed from the C with Classes language . There is mutual borrowing, for example, the void * pointer type was invented for ANSI C, and was first implemented in C ++. As promised in the first edition of this book, the C ++ description has been refined to avoid unnecessary discrepancies. C ++ is now more compatible with the C language than it was in the beginning ($$ R.18). Ideally, C ++ should be as close as possible to ANSI C, but no more [9]. There has never been and never will be 100% compatibility , since this would violate the type reliability and consistency of the use of built-in and user-defined types, and these properties have always been among the main ones for C ++.

You don't need to know C to learn C ++. Programming in C helps you learn techniques and even tricks that are simply unnecessary in C ++ programming. For example, explicit type conversion (casting) is needed much less frequently in C ++ than in C (see Notes for C Programmers below). However, good C programs are essentially C ++ programs. For example, all programs from the classic C description [8] are C ++ programs. In the process of learning C ++, experience with any language with static types will be useful.

Efficiency and structure

The development of the C ++ language was based on the C language, and, with a few exceptions, C was retained as a subset of C ++. The basic C language was designed in such a way that there is a very close relationship between types, operations, operators and objects that the machine directly interacts with , i.e. numbers, symbols and addresses. Except for the new, delete, and throw operations, and the block being checked, no hidden dynamic hardware or software support is required to execute C ++ statements and expressions .

C ++ uses the same (or even more efficient) sequence of commands to call and return functions as in C. Even if these rather efficient operations become too expensive, the function call can be replaced by substitution of its body, and it remains convenient functional notation without any cost of calling the function.

Initially, the C language was conceived as a competitor to assembly language, capable of ousting it from the main and most resource-demanding tasks of system programming. The C ++ project took steps to ensure that C's advances in this area were not compromised. The difference between the two languages primarily lies in the degree of attention given to types and structures. C language is expressive and at the same time forgiving in relation to types. The C ++ language is even more expressive, but such expressiveness can only be achieved when types are given great attention. When the types of objects are known, the translator correctly recognizes expressions in which the programmer would otherwise have to write down operations in tedious detail. In addition, knowing the types allows the translator to detect bugs that would otherwise only be found in testing. Note that the use of strong typing of the language to control function parameters, protect data from illegal access, define new types and operations does not entail additional memory costs and increase program execution time.

A C ++ project pays special attention to program structuring. This is due to the increase in program size since the advent of C. A small program (say, no more than 1000 lines) can be made to work out of obstinacy, breaking all the rules of good programming style. However, by doing so, a person will no longer be able to cope with a large program. If your program has 10,000 lines of bad structure, you will find that new errors appear as quickly as old ones are removed . C ++ was created with the goal that a large program could be structured in such a way that one person does not have to work with 25,000 lines of text. At present, it can be considered that this goal has been fully achieved.

There are, of course, even larger programs. However, those that are actually used can usually be divided into several practically independent parts, each of which is much smaller than the size mentioned. Naturally, the difficulty of writing and maintaining a program is determined not only by the number of lines of text, but also by the complexity of the subject area. So the numbers cited here, on which our considerations were based, should not be taken too seriously.

Unfortunately, not every part of the program can be well structured, made independent of the hardware, understandable enough, etc. C ++ provides facilities that directly and effectively represent hardware capabilities. Their use allows you to get rid of concerns about the reliability and ease of understanding of the program. Such parts of the program can be hidden, providing a reliable and simple interface with them.

Naturally, if C ++ is used for a large program, it means that the language is used by groups of programmers. Modularity, flexibility, and strongly typed interfaces are useful here . C ++ has as good a set of tools for building large programs as many languages. But as a program gets bigger, the challenges of creating and maintaining it move from the language domain to the more global domain of software and project management. Chapters 11 and 12 address these issues.

This book focuses on techniques for creating generics, utility types, libraries, and more. These methods can be successfully applied to both small and large programs. Moreover , since all non-trivial programs consist of several largely independent parts of each other, the methods of programming the individual parts will be useful to both system and application programmers.

You might suspect that writing a program using the verbose type system will increase the size of the text. This is not the case for a C ++ program: a C ++ program, which describes the types of formal parameters of functions, defines classes, etc., is usually even shorter than its C equivalent, where these means are not used. When a C ++ program uses libraries, it is also shorter than its C equivalent, if it exists.

Philosophical remarks

The programming language solves two interrelated tasks: it allows the programmer to write down the actions to be performed and forms the concepts with which the programmer operates when thinking about his task. The first goal is ideally matched by the language, which is very "close to the machine". Then, with all its main "entities", you can simply and efficiently work in this language, and doing this in an obvious way for the programmer. This is what the creators of C had in mind. The second goal is ideally answered by a language that is so "close to the task at hand" that it directly and accurately expresses the concepts used in solving the problem. This is what was meant when the funds added to C. were originally determined.

The connection between the language in which we think and program, as well as between the problems and their solutions that can be imagined in our minds, is quite close. For this reason, limiting the capabilities of a language to only finding programmer errors is dangerous at best. As with natural languages, it is very helpful to be at least bilingual. The language provides the programmer with some concepts in the form of language tools; if they are not suitable for the task, they are simply ignored. For example, if you significantly restrict the concept of a pointer, then the programmer will be forced to create structures, pointers, etc. use vectors and integer operations. A good design of a program and the absence of errors in it cannot be guaranteed only by the presence or absence of certain features in the language.

Language typing should be especially useful for non-trivial tasks. Indeed, the notion of a class in C ++ has proven to be a powerful conceptual tool.

C ++ Programming Notes

It is assumed that, ideally, the development of a program is divided into three stages: first, it is necessary to achieve a clear understanding of the problem, then to define the key concepts used to solve it, and, finally, to express the resulting solution in the form of a program. However, the details of the solution and the exact concepts that will be used in it are often clarified only after they have been attempted to be expressed in the program. It is in this case that the choice of a programming language becomes very important.

Many tasks use concepts that are difficult to represent in a program as one of the basic types or as a function without associated static data. A class can represent such a concept in a program . A class is a type; it defines the behavior of the objects associated with it: their creation, processing and destruction. In addition, the class defines the implementation of objects in the language, but at the initial stages of program development this is not and should not be the main concern. To write a good program, you need to create a set of classes in which each class clearly represents one concept. This usually means that the programmer should focus on the questions: How are objects of a given class created? Can they be copied and / or destroyed? What operations can be defined on these objects? If there are no satisfactory answers to these questions , then most likely this means that the concept was not clearly formulated. Then, perhaps, it is still worth reflecting on the problem and the proposed solution, and not immediately start programming, hoping to find answers in the process .

The easiest way is to work with concepts that have a traditional mathematical form of representation: all kinds of numbers, sets, geometric shapes, etc. For such concepts, it would be useful to have standard class libraries, but at the time of this writing they did not exist yet. The software world has accumulated an amazing wealth of such libraries, but there is no formal or actual standard for them. The C ++ language is still quite young, and its libraries have not evolved to the same extent as the language itself.

A concept does not exist in a vacuum; concepts associated with it are always grouped around it . Determining the relationships of classes in the program, in other words, establishing exact relationships between the concepts used in the problem , is more difficult than defining each of the classes by itself. As a result, there should be no "mess" - when each class (concept) depends on all the others. Let there be two classes A and B. Then the connections between them like "A calls a function from B", "A creates objects B", "A has a member of type B" usually do not cause any difficulties. Relationships of the type "A uses data from B", as a rule, can be excluded altogether.

One of the most powerful intelligence in dealing with complexity is hierarchical ordering, i.e. the ordering of related concepts into a tree-like structure in which the most general concept is at the root of the tree. It is often possible to organize the classes of a program as a set of trees or as a directed acyclic graph. This means that the programmer defines a set of base classes, each of which has its own set of derived classes. The set of operations of the most general type for base classes (concepts) is usually defined using virtual functions ($$ 6.5). The interpretation of these operations, as needed, can be specified for each specific case, i.e. for each derived class.

Naturally, there are limitations with such an organization of the program. Sometimes the concepts used in the program cannot be ordered even with the help of a directed acyclic graph. Some concepts turn out to be interrelated in nature. Cyclic dependencies will not cause problems if the set of interconnected classes is so small that it is easy to understand it. Friendly classes can be used to represent many interdependent classes in C ++ ($$ 5. 4.1).

If the concepts of a program cannot be arranged in the form of a tree or a directed acyclic graph, and many interdependent concepts cannot be localized, then, apparently, you are in a predicament that no programming language can help you get out of. If you have not succeeded in simply formulating the connections between the basic concepts of the problem, then most likely you will not be able to program it.

Type templates provide another way of expressing commonality in a language. A template class defines a whole family of classes. For example, the template class list defines classes of the form "list of objects T", where T can be of any type. Thus, the template type indicates how a new type is obtained from the given as a parameter. The most typical templated classes are containers, specifically lists, arrays, and associative arrays.

Recall that many tasks can be programmed easily and simply using only simple types, data structures, regular functions, and a few classes from the standard libraries. The entire apparatus for constructing new types should be used only when it is really necessary.

The question "How do I write a good C ++ program?" very similar to the question "How is good English prose written?" There are two answers to it: "You need to know what you actually want to write" and "Practice and imitate good style." Both tips work in C ++ as well as in English, and both are difficult to follow.

Some helpful tips

Below is a "set of rules" to keep in mind when learning C ++. As you become more experienced, you can use these rules to formulate your own rules that are more suitable for your tasks and more in line with your programming style. Very simple rules have been deliberately chosen, and details have been omitted. They should not be taken too literally. A good program requires intelligence, taste, and patience. It usually doesn't work the first time, so experiment! So, a set of rules.

1] When you write a program, you create concrete representations of the concepts that were used to solve the problem. The structure of the program should reflect these concepts as explicitly as possible.

[a] If you think of "something" as a separate concept, then make it a class.

[b] If you consider "something" to exist independently, then make it an object of some class.

[c] If two classes have something essential, and it is common to them, then express that commonality using a base class.

[d] If the class is a container for some objects, make it a templated class.

2] If a class is defined that does not implement mathematical objects like matrices or complex numbers, and is not a low-level type like a linked list, then:

[a] Do not use global data.

[b] Do not use global functions (non-members).

[c] Do not use shared data members.

[d] Don't use friend functions (but only to avoid [a], [b], or [c]).

[e] Do not access data members of another object directly.

[f] Do not start "type field" in the class; use virtual functions.

[g] Use substitution functions only as a means of significant optimization.

A note for C programmers

The better a programmer knows C, the more difficult it will be for him to deviate from the C programming style when programming in C ++. Thus, he loses the potential advantages of C ++. Therefore, we advise you to look through the "Differences from C" section in the reference manual ($$ R.18). Here we will only point out the places where using additional C ++ features leads to a better solution than programming in pure C. Macros are practically unnecessary in C ++: use const ($$ 2.5) or enum ($$ 2.5.1) to define named constants; use inline ($$ 4.6.2) to avoid function call overhead; use templates like ($$ 8) to define a family of functions and types. Do not describe a variable until you really need it, and then you can initialize it right away, because in C ++ a description can appear anywhere where an operator is allowed. Don't use malloc (), new ($$ 3.2.6) better implements this operation. Unions are not needed as often as in C, because alternatives in structs are implemented using derived classes. Try to avoid unions, but if you do need them, don't include them in the main interfaces; use unnamed unions ($$ 2.6.2). Avoid using void * pointers, pointer arithmetic, C-style arrays, and casts. If you do use these constructs, hide them securely enough in some function or class. Note that C-style binding is possible for a C ++ function if it is described with the extern "C" specification ($$ 4.4).

But it is much more important to try to think of a program as a set of interrelated concepts represented by classes and objects than to think of it as a sum of data structures and functions that do something with that data.

List of references

There are few direct references to literature in the book. Here is a list of books and articles that are directly referenced, as well as those that are only mentioned.

1] AVAho, JE. Hopcroft, and JDUlman: Data Structures and Algoritms. Addison-Wesley, Reading, Massachusetts. 1983.

2] OJ Dahl, B. Myrhaug, and K. Nugaard: SIMULA Common Base Language. Norwegian Computing Ctnter S-22. Oslo, Norway. 1970

3] OJ Dahl and CARHoare: Hierarhical Program Construction in Structured Programming. Academic Press, New York. 1972. pp. 174-220.

4] Margaret A. Ellis and Bjarne Stroustrup: The Annotated C ++ Reference Manual. Addison-Wesley, Reading, Massachusetts. 1990.

5] A. Goldberg and D. Rodson: SMALLTALK-80 - The Language and Its Implementation. Addison-Wesley, Reading, Massachusetts. 1983.

6] REGriswold et.al .: The Snobol14 Programming Language. Prentice-Hall, Englewood Cliffs, New Jersy, 1970.

7] REGriswold and MTGriswold: The ICON Programming Language. Prentice-Hall, Englewood Cliffs, New Jersy. 1983.

8] Brian W. Kernighan and Dennis M. Ritchie: The C Programming Language. Prentice-Hall, Englewood Cliffs, New Jersy. 1978. Second edition 1988.

9] Andrew Koenig and Bjarne Stroustrup: C ++: As Close to C as possible - but no closer. The C ++ Report. Vol.1 No.7. July 1989.

10] Andrew Koenig and Bjarne Stroustrup: Exception Handling for C ++ (revised). Proc USENIX C ++ Conference, April 1990. Also, Journal of Object Oriented Programming, Vol.3 No.2, July / August 1990. pp.16-33.

11] Barbara Liskov et al .: CLU Reference Manual. MIT / LCS / TR-225.

12] George Orwell: 1984. Secker and Warburg, London. 1949.

13] Martin Richards and Colin Whitby-Strevens: BCPL - The Language and Its Compiler. Cambridge University Press. 1980.

14] L. Rosler: The Evolution of C - Past and Future. AT&T Bell Laboratories Technical Journal. Vol.63 No.8 Part 2. October 1984. pp. 1685-1700.

15] Ravi Sethi: Uniform Syntax for Type Expressions and Declarations. Software Practice & Experience, Vol.11. 1981. pp. 623-628.

16] Bjarne Stroustrup: Adding Classes to C: An Exercise in Language Evolution. Software Practice & Experience, Vol.13. 1983. pp. 139-61.

17] Bjarne Stroustrup: The C ++ Programming Language. Addison-Wesley. 1986.

18] Bjarne Stroustrup: Multiple Inheritance for C ++. Proc. EUUG Spring Conference, May 1987. Also USENIX Computer Systems, Vol. 2 No. 4, Fall 1989.

19] Bjarne Stroustrup and Jonathan Shopiro: A Set of C classes for Co-Routine Style Programming. Proc. USENIX C ++ conference, Santa Fe. November 1987. pp. 417-439.

20] Bjarne Stroustrup: Type-safe Linkage for C ++. USENIX Computer Systems, Vol.1 No.4 Fall 1988.

21] Bjurne Stroustrup: Parameterized Type for C ++. Proc. USENIX C ++ Conference, Denver, October 1988. pp. 1-18. Also, USENIX Computer Systems, Vol.2 No.1 Winter 1989.

22] Bjarne Stroustrup: Standardizing C ++. The C ++ Report. Vol.1 No.1. January 1989.

23] Bjarne Stroustrup: The Evolution of C ++: 1985-1989. USENIX Computer Systems, Vol.2 No.3. Summer 1989.

24] PM Woodward and SGBond: Algol 68-R Users Guide. Her Majesty's Stationery Office, London. 1974.

25] UNIX Time-Sharing System: Programmer's Manual. Research Version, Tenth Edition. AT&T Bell Laboratories, Murray Hill, New Jersy, February 1985.

26] Aake Wilkstroem: Functional Programming Using ML. Prentice-Hall, Englewood Cliffs, New Jersy. 1987.

27] X3 Secretariat: Standard - The C Language. X3J11 / 90-013. Computer and Business Equipment Manufactures Association, 311 First Street, NW, Suite 500, Washington, DC 20001, USA.

Links to sources on the design and development of large software systems can be found at the end of Chapter 11.

CHAPTER 1. BRIEF OVERVIEW OF C ++

"Let's start by picking up all these legalists, linguists."
("King Henry VI", Act II)

This chapter provides a brief overview of the basic concepts and constructs of the C ++ language. It serves as a quick introduction to the language. A detailed description of the language's capabilities and programming methods is given in the following chapters. The conversation revolves around data abstraction and object-oriented programming, but the main features of procedural programming are also listed.

1.1 INTRODUCTION

The C ++ programming language was conceived as a language that would:

- better than the C language;
- support data abstraction;
- support object-oriented programming.

This chapter explains the meaning of these phrases without detailing the language constructs.

$$ 1.2 contains an informal description of the differences between "procedural", "modular" and "object-oriented" programming. The language constructions are given, which are essential for each of the listed programming styles. The programming style inherent in C is discussed in the sections on "Procedural Programming and" Modular Programming. "C ++ is the" best option for C. "It supports this programming style better than C itself, and it does so without losing any generality or efficiency. compared to C. At the same time, C is a subset of C ++. Data abstraction and object-oriented programming are considered as "support for data abstraction" and "support for object-oriented programming." The first is based on the ability to define new types and work with them , and the second - on the ability to define a hierarchy of types.

$$ 1.3 describes the basic constructs for procedural and modular programming. In particular, functions, pointers, loops, I / O, and the concept

of a program as a collection of separately translated modules are defined. These features are described in detail in Chapters 2, 3 and 4.

$$ 1.4 contains a description of tools designed to effectively implement data abstraction. In particular, it defines classes, basic access control mechanisms, constructors and destructors, operation overloading, custom type conversions, exception handling, and type templates. These features are described in detail in Chapters 5, 7, 8, and 9.

$$ 1.5 contains a description of the object-oriented programming support tools. In particular, derived classes and virtual functions are defined, and some implementation issues are discussed. All of this is detailed in Chapter 6.

$$ 1.6 contains a description of certain restrictions on the way of improving both general-purpose programming languages in general and C ++ in particular. These constraints relate to efficiency, conflicting requirements for different areas of the application, learning challenges, and the need to translate and run programs on older systems.

If a section turns out to be incomprehensible to you, we strongly advise you to read the corresponding chapters, and then, having read the detailed description of the basic constructions of the language, return to this chapter. It is needed so that you can get a general idea of the language. There is not enough information in it to start programming immediately.

1.2 Programming paradigms

Object Oriented Programming is a programming technique, a way of writing "good" programs for a variety of tasks. If this term has any meaning, then it should imply: a programming language that provides good opportunities for an object-oriented programming style.

Important differences should be pointed out here. A language is said to support a certain style of programming if it has features that make programming in that style convenient (simple enough, reliable, and efficient). The language does not support a certain style of programming if it takes a lot of effort or even art to write a program in that style. However, this does not mean that the language prohibits writing programs in this style. Indeed, it is possible to write structured programs in Fortran and object-oriented programs in C, but this will be a waste of energy, since these languages do not support these programming styles.

A language's support for a certain programming paradigm (style) is clearly manifested in specific language constructs designed for it. But it can manifest itself in a more subtle, hidden form, when a deviation from the paradigm is diagnosed at the stage of translation or program execution. The most obvious example is type checking. In addition, language support for the paradigm can be supplemented with unambiguity checking and dynamic control. Support can be provided beyond the language itself, for example, standard libraries or a programming environment.

This is not to say that one language is better than another just because it has opportunities that are absent in the other. The opposite is often the case. It is more important here not what capabilities the language has, but how much the capabilities available in it support the chosen programming style for a certain range of tasks. Therefore, the following language requirements can be formulated:

1] All language constructs should be naturally and elegantly defined in it.

2] To solve a specific problem, it should be possible to use combinations of structures to avoid the need to introduce a new structure for this purpose.

3] There should be a minimum number of non-obvious special-purpose structures.

4] The construct must allow such an implementation so that no additional cost is incurred in a program that does not use it.

5] The user only needs to know the set of constructions that are directly used in his program.

The first requirement appeals to logic and aesthetic taste. The next two express the principle of minimality. The last two can be formulated differently as follows: "what you do not know cannot harm you."

Given the constraints specified in these rules, C ++ was designed to support data abstraction and object-oriented programming in addition to the traditional C style. However, this does not mean that the language requires any one programming style from all users.

Now let's move on to specific programming styles and see what are the main language constructs that support them. We are not going to give a complete description of these constructs.

1.2.1 Procedural programming

The original (and perhaps most used) programming paradigm was:

Determine what treatments you need; use the best algorithms you know!

The emphasis was on processing the data with an algorithm that performs the required calculations. To support this paradigm, languages provided a mechanism for passing parameters and retrieving function results. The literature reflecting this approach is filled with arguments about the methods of passing parameters, how to distinguish between parameters of different types, about different types of functions (procedures, subroutines, macros, ...), etc. The first procedural language was Fortran, and Algol60, Algol68, Pascal and C continued this trend.

A typical example of good style in this sense is the square root function. For a given parameter, it produces a result that is obtained using clear mathematical operations:

```
double sqrt (double arg)
{
        // program for calculating the square root
}
void some_function ()
{
        double root = sqrt (2);
// ..
}
```

The double slash // starts a comment that continues to the end of the line.

With this organization of the program, functions bring a certain order to the chaos of various algorithms.

1.2.2 Modular programming

Over time, the emphasis in software design has shifted from organizing procedures to organizing data structures. Among other things, this is due to the growth in the size of programs. A module is usually called a collection of related procedures and the data they control. The programming paradigm has taken the form:

Determine which modules are needed; split the program so that the data is hidden in these modules

This paradigm is also known as the principle of data hiding. If the language does not have the ability to group related procedures along with data, then it does not support the modular programming style. Now the method of writing "good" procedures is applied to individual procedures in a module. A typical example of a module is defining a stack. Here it is necessary to solve the following tasks:

1] Provide the user with an interface to the stack (for example, the push () and pop () functions).

2] Ensure that the stack view (for example, as an array of elements) is only accessible through the user interface.

3] Provide initialization of the stack before first use.

The Modula-2 language directly supports this paradigm, whereas C only allows this style. Below is a possible external interface of a module that implements the stack in C:

```
// description of the interface for the module that implements the symbol
stack:
void push (char);
char pop ();
const int stac k_size = 100;
```

Suppose that the interface description is in the stack.h file, then the stack implementation can be defined as follows:

```
#include "stack.h" // use the stack interface
static char v [stack_size]; // `` static " means local
                                                      // in this file /
module
static char * p = v; // stack is empty at first
void push (char c)
{
        // check for overflow and push onto the stack
}

char pop ()
{
        // check if the stack is empty and read from it
}
```

It is possible that the implementation of the stack may change, for example, if you use a linked list for storage. In any case, the user does not have direct access to the implementation: v and p are static variables, i.e. variables are local in the module (file) in which they are described. You can use the stack like this:

```
#include "stack.h" // use the stack interface

void some_function ()
{
            push ('c');
            char c = pop ();
            if (c! = 'c') error ("impossible");
}
```

Since data is the only thing that one wants to hide, the concept of hiding data is trivially extended to that of hiding information, i.e. names of variables, constants, functions and types, which can also be local in the module. Although C ++ was not specifically designed to support modular programming, classes support the concept of modularity ($$ 5.4.3 and $$ 5.4.4). In addition to this, C ++ naturally has the already demonstrated modularity capabilities that C has, i.e. presentation of the module as a separate unit of translation.

1.2.3 Data abstraction

Modular programming involves grouping all data of the same type around a single module that controls that type. If you need two different types of stacks, you can define a module that controls them with the following interface:

```
class stack_id {/ * ... * /}; // stack_id type only
                                                    // no information
about stacks
                                                    // not contained here

stack_id create_stack (int size); // create a stack and return
                                                    // its identifier

void push (stack_id, char);
char pop (stack_id);
```

```
destroy_stack (stack_id); // destroy the stack
```

Of course, such a solution is much better than the chaos inherent in traditional, unstructured solutions, but the types modeled in this way are clearly different from the "real" built-in ones. Each type-driving module must define its own algorithm for creating "variables" of that type. There are no universal rules for assigning identifiers for objects of this type. "Variables" of these types do not have names that would be known to the translator or other system programs, and these "variables" do not obey the usual rules of scoping and parameter passing.

The type implemented by the module that controls it differs significantly from built-in types in many important respects. Such types do not receive the same support from the translator (different kinds of control) that is provided for built-in types. The problem here is that the program is formulated in terms of small (one or two words) object descriptors, and not in terms of the objects themselves (stack_id is an example of such a descriptor). This means that the translator will not be able to catch stupid, obvious mistakes like the ones in the function below:

```
void f ()
{
stack_id s1;
stack_id s2;

s1 = create_stack (200);
            // error: forgot to create s2
push (s1, 'a');
char c1 = pop (s1);
destroy_stack (s2); // nasty error
            // error: forgot to destroy s1
s1 = s2; // this assignment is essentially
                                                  // by assigning
pointers,
                                                  // but here s2 is used
after destruction
}
```

In other words, the concept of modularity, which supports the data-hiding paradigm , does not prohibit this programming style, but does not promote it either .

In the languages of Ada, Clu, C ++ and the like, this difficulty is overcome due to the fact that the user is allowed to define his own types, which are treated in the language in almost the same way as built-in types . These types are commonly referred to as abstract data types, although it might be better to simply call them user-defined. A stricter definition of abstract data types would be their mathematical definition. If we could give it, what we call types in programming would be a concrete representation of truly abstract entities. How to define "more abstract" types is shown in $$ 4.6. The programming paradigm can now be expressed like this:

Determine what types you need; provide a complete set of operations for each type.

If there is no need for different objects of the same type, then the programming style , the essence of which boils down to hiding data, and the adherence to which is ensured using the concept of modularity, is quite adequate to this paradigm.

Arithmetic types, like rational and complex types, are typical examples of custom types:

```
class complex
{
            double re, im;
            public:
                        complex (double r, double i) {re = r; im = i; }
                        complex (double r) // conversion float-> complex
             {re = r; im = 0; }
                        friend complex operator + (complex, complex);
                        friend complex operator- (complex, complex); //
subtract
                        friend complex operator- (complex) // unary minus
                        friend complex operator * (complex, complex);
                        friend complex operator / (complex, complex);
                        // ...
};
```

The description of the class (that is, a user-defined type) complex specifies the representation of a complex number and a set of operations on complex numbers. The view is private: re and im are available only to the functions specified in the complex class description. Similar functions can be defined like this:

```
complex operator + (complex a1, complex a 2)
{
        return complex (a1.re + a2.re, a1.im + a2.im);
}
```

and used like this:

```
void f ()
{
        complex a = 2.3;
        complex b = 1 / a;
        complex c = a + b * complex (1, 2.3);
        // ...
        c = - (a / b) + 2;
}
```

Most (though not all) modules are better defined as custom types.

1.2.4 The Limits of Data Abstraction

An abstract data type is defined as a kind of "black box". Once defined, it essentially does not interact with the program in any way. It cannot be adapted for new purposes in any way without changing the definition. In this sense, it is an inflexible decision. Suppose, for example, you need to define the shape type for the graphics system. For now, we believe that the system can have such figures: a circle (circle), triangle (triangle) and square (square). Let there already be definitions of point and color:

```
class point {/ * ... * /};
class color {/ * ... * /};
```

The shape type can be defined as follows:

```
enum kind {circle, triangle, square};

class shape
{
        point center;
        color col;
        kind k;
        // shape representation
        public:
                point where () {return center; }
                void move (point to) {center = to; draw (); }
                void draw ();
                void rotate (int);
                // some more operations
};
```
A "field of type" k is needed so that operations such as draw () and rotate () can determine which shape they are dealing with (in languages like Pascal, you can use a variant notation for this, in which k is a field descriminant). The draw () function can be defined like this:

```
void shape :: draw ()
{
        switch (k)
        {
                case circle:
                        // draw a circle
                        break;
                case triangle:
                        // draw triangle
                        break;
                case square:
                        // draw a square
                        break;
        }
}
```
This is not a function, but a nightmare. In it, you need to take into account all the possible figures that are there. Therefore, it is supplemented with new operators as soon as a new figure appears in the system. The bad news is that after defining a new shape, all the old class operations need to be checked and possibly changed. Therefore, if the source code of each class operation is not available to you, it is simply impossible to introduce a new shape into the system. The appearance of any new shape results in text manipulation of every essential class operation. It takes a high enough level of skill to handle this task, but bugs can still appear in already debugged parts of the program that work with old shapes. The ability to choose a

representation for a particular shape is greatly reduced if you require that all its representations fit into a predefined format specified by the general shape definition (that is, the definition of the shape type).

1.2.5 Object Oriented Programming

The problem is that we do not distinguish between general properties of shapes (for example, a shape has a color, it can be drawn, etc.) and properties of a specific shape (for example, a circle is a shape that has a radius, it is drawn using the function drawing arcs, etc.). The essence of object-oriented programming is that it allows you to express these differences and exploits them. A language that has constructs for expressing and exploiting such distinctions supports object-oriented programming. All other languages do not support it. Here the main role is played by the inheritance mechanism borrowed from the Simula language. First, let's define a class that defines the general properties of all shapes:

```
class shape
{
        point center;
        color col;
        // ...
        public:
                point where () {return center; }
                void move (point to) {center = to; draw (); }
                virtual void draw ();
                virtual void rotate (int);
                // ...
};
```

Those functions for which the declared interface can be defined, but the implementation of which (i.e., the body with the operator part) is possible only for specific figures, are marked with the virtual service word. In Simula and C ++, the virtuality of a function means: "a function can be defined later in a class derived from this one." Given this definition of a class, you can write general functions that work with shapes:

```
void rotate_all (shape v [], int size, int angle)
        // rotate all elements of array "v" of size "size"
        // by an angle equal to "angle"
{
```

```
            int i = 0;
            while (i <size)
            {
                        v [i]. rotate (angle);
                        i = i + 1;
            }
   }
```

To define a specific figure, you should indicate, first of all, that it is a figure and set its special properties (including virtual functions):

```
   class circle : public shape
   {
            int radius;
            public:
                        void draw () {/ * ... * /};
                        void rotate (int) {} // yes, empty function for now
   };
```

In C ++, the circle class is said to be derived from the shape class , and the shape class is said to be the base class of the circle class. Other terminology is possible , using the names "subclass" and "superclass" for the classes circle and shape, respectively. Now the programming paradigm is formulated as follows:

Determine which class you need; provide a complete set of operations for each class; Express the generality of classes explicitly using inheritance.

If there is no commonality between the classes, data abstraction is sufficient . How applicable object-oriented programming is for a given area of application is determined by the degree of commonality between the different types that allows inheritance and virtual functions. In some areas, such as interactive graphics, there is wide scope for object-oriented programming. In other areas that use traditional arithmetic types and computation on them, it is difficult to find a use for more advanced programming styles than data abstraction. The facilities supporting object-oriented programming are clearly redundant here.

Finding commonality among certain types of systems is a non-trivial process. The degree of this generality depends on how the system is designed. In the design process, identifying commonality of classes should be an ongoing goal. It is achieved in two ways: either by designing special classes that are used as "bricks" when building others, or by looking for similar classes to isolate their common part into one base class.

Attempts to explain what object-oriented programming is without using specific constructs of programming languages can be found in [2] and [6], listed in the bibliography in Chapter 11.

So we've outlined the minimum support a programming language should provide for procedural programming, data hiding, data abstraction, and object-oriented programming. Now let us describe in more detail the language features, although not the most essential, but allowing more efficient implementation of data abstraction and object-oriented programming.

1.3 "Superior C"

Minimal support for procedural programming includes functions, arithmetic operations, operator selections, and loops. In addition, I / O operations must be provided. Basic language features C ++ inherited from C (including pointers), and I / O is provided by the library. The most rudimentary concept of modularity is realized with a split broadcast mechanism.

1.3.1 Program and standard output

The smallest C ++ program looks like this:

```
main () {}
```

This program defines a function called main, which has no parameters and does nothing. Curly braces {and} are used in C ++ to group operators. In this case, they represent the beginning and end of the body of the (empty) function main. Every C ++ program must have its own main () function, and the program starts by executing that function.

Usually the program produces some kind of results. Here is a program that prints out the Hello, World! (Hello!):

```
#include <iostream.h>

int main ()
{
        cout << "Hello, World! \ n";
}
```

The #include <iostream.h> line tells the translator to include in the program the descriptions necessary for the operation of the standard I / O streams that are in iostream.h. Without these descriptions, the expression

```
cout << "Hello, World! \ n"
```

wouldn't make sense. The operation << ("give out") writes its second parameter to the first parameter. In this case, the string "Hello, World! \ N" is written to the cout standard output stream. A string is a sequence of characters enclosed in double quotes. Two characters, backslash \ and immediately following it, denote some special character. In this case, \ n is an end-of- line (or line feed) character, so it is printed after the characters Hello, world!

The integer value returned by main (), if any, is considered the return value of the program to the system. If nothing is returned, the system will receive some kind of "garbage" value.

Stream library I / O facilities are detailed in Chapter 10.

1.3.2 Variables and arithmetic operations

Every name and every expression must have a type. It is the type that defines the operations that can be performed on them. For example, in the description

```
int inch;
```

it says that inch is of type int, i.e. inch is an integer variable.

Description is a statement that enters a name into a program. The description indicates the type of name. The type, in turn, determines how to properly use the name or expression.

The main types that are closest to the "hardware reality" of the machine are as follows:

```
char
short
int
long
```

They represent whole numbers. The following types:

```
float
double
long double
```

represent floating point numbers. A char variable is sized to hold one character on a given machine (usually one byte). The variable int has the size required for whole arithmetic on a given machine (usually one word).

The following arithmetic operations can be used on any combination of the listed types:

```
+ (plus, unary and binary)
- (minus, unary and binary)
* (multiplication)
/ (division)
% (remainder of the division)
```

The same is true for relation operations:

```
== (equal)
! = (not equal)
<(less than)
<= (less or equal)
> = (greater or equal)
```

For assignment and arithmetic operations in C ++, all meaningful conversions of basic types are performed so that any combination of them can be used indefinitely:

```
double d;
int i;
short s;
// ...
d = d + i;
i = s * i;
```

The = symbol indicates normal assignment.

1.3.3 Pointers and Arrays

The array can be described as follows:

```
char v [10]; // array of 10 characters
```

The pointer description looks like this:

```
char * p; // pointer to character
```

Here [] means "array of", and the * character means "pointer to". The lower subscript value for all arrays is zero, so v has 10 elements: v [0] ... v [9]. A variable of the pointer type can contain the address of an object of the corresponding type:

```
p = & v [3]; // p points to the 4th element of v
```

The unary operation & means taking an address.

1.3.4 Conditional statements and loops

C ++ has a traditional set of select operators and loops. The following are examples of if, switch, and while statements.

The following example shows the conversion of inches to centimeters and vice versa. In the input stream, it is assumed that the value in centimeters ends with an i, and the value in inches ends with a c:

```
#include <iostream.h>

int main ()
{
        const float fac = 2.54;
        float x, in, cm;
        char ch = 0;

        cout << "enter length:";
        cin >> x; // input a floating point number
        cin >> ch // enter trailing character
        if (ch == 'i')
        { // inch
                in = x;
                cm = x * fac;
        }
        else if (ch == 'c')
        {// centimeters
                in = x / fac;
                cm = x;
        }
        else
                in = cm = 0;
        cout << in << "in =" << cm << "cm \ n";
}
```

The >> ("enter from") operation is used as an input operator; cin is the standard input stream. The type of the operand to the right of the >> operator determines what value is entered; it is written to this operand.

The switch statement compares a value with a set of constants. The check in the previous example can be written like this:

```
switch (ch)
{
        case 'i':
                in = x;
                cm = x * fac;
                break;
        case 'c':
                in = x / fac;
                cm = x;
                break;
        default:
                in = cm = 0;
                break;
}
```

The break statements are used to exit a switch. All variant constants must be different. If the value being compared does not match any of them, the statement labeled default is executed. The default option may not be present.

Here is a record that specifies copying 10 elements of one array to another:

```
int v1 [10];
int v2 [10];
// ...
for (int i = 0; i <10; i ++) v1 [i] = v2 [i];
```

In words, it can be expressed like this: "Start with i equal to zero, and while i is less than 10, copy the i-th element and increment i." The increment (++) of an integer variable simply boils down to an increment of 1.

1.3.5 Functions

A function is a named part of a program that can be called from other parts of the program as many times as needed. Here is a program that gives out powers of two :

```
extern float pow (float, int);
            // pow () is defined elsewhere
int main ()
{
            for (int i = 0; i <10; i ++) cout << pow (2, i) << '\ n';
}
```

The first line is the description of the function. It defines pow as a float and int function that returns a float. The function description is needed to call it, its definition is in a different place.

When a function is called, the type of each actual parameter is checked against the type specified in the function description, just as if a variable of the described type was initialized. This ensures proper type checking and conversions. For example, a call to the function pow (12.3, "abcd") will be considered erroneous by the translator, since "abcd" is a string, not an int parameter. In the call to pow (2, i), the translator converts an integer constant (integer 2) to a floating point number (float) as required by the function. The pow function can be defined as follows:

```
float pow (float x, int n)
{
            if (n <0)
                        error ("error: negative exponent specified for pow ()");
            switch (n)
            {
                        case 0: return 1;
                        case 1: return x;
                        default: return x * pow (x, n-1);
            }
}
```

The first part of the function definition specifies its name, the return type (if any), and the types and names of formal parameters (if they exist). The value is returned from the function using the return statement.

Different functions usually have different names, but functions that perform similar operations on objects of different types should be given the same name. If the types of parameters of such functions are different, the translator can always figure out which function needs to be called. For example, you can have two exponentiation functions, one for integers and one for floating point numbers :

```
int pow (int, int);
double pow (double, double);
        // ...
x = pow (2,10); // call pow (int, int)
y = pow (2.0, 10.0); // call pow (double, double)
```

This reuse of a name is called function name overloading, or simply overloading; overloading is discussed specifically in chapter 7.

Function parameters can be passed either "by value" or "by reference". Consider the definition of a function that exchanges the values of two integer variables. If the standard way of passing parameters by value is used, then you will have to pass pointers:

```
void swap (int * p, int * q)
{
        int t = * p;
        * p = * q;
        * q = t;
}
```

The unary operation * is called an indirection (or dereference operation), it selects the value of the object that the pointer is set to . The function can be called like this:

```
void f (int i, int j)
{
        swap (& i, & j);
}
```

If you use parameter passing by reference, you can do without explicit operations with a pointer:

```
void swap (int & r1, int & r2)
{
        int t = r1;
        r1 = r2;
        r2 = t;
}
```

```
void g (int i, int j)
{
            swap (i, j);
}
```

For any type of T, the notation T & means "reference to T". A reference is synonymous with the variable with which it was initialized. Note that overloading allows two swap functions to coexist in the same program.

1.3.6 Modules

A C ++ program almost always consists of several separately translated "modules". Each "module" is usually called a source file, but sometimes a translation unit. It consists of a sequence of descriptions of types, functions, variables, and constants. The extern declaration allows one source file to refer to a function or object defined in another source file. For example:

```
extern "C" double sqrt (double);
extern ostream cout;
```

The most common way to ensure consistency of external descriptions in all source files is to place such descriptions in special files called header files. Header files can be included in all source files that require external descriptions. For example, the description of the sqrt function is stored in the header file of standard math functions called math.h, so if you want to extract the square root of 4, you can write:

```
#include <math.h>
// ...
x = sqrt (4);
```

Because standard header files can be included in many source files, they do not contain descriptions that could be duplicated to cause errors. So, the body of a function is present in such files, if only it is a substitution function, and initializers are specified only for constants ($$ 4.3). Apart from such cases, the header file usually serves as a repository for types, it provides an interface between separately translated parts of the program.

In the include command, the filename enclosed in angle brackets (in our example, <math.h>) refers to a file located in the standard include directory . This is often the / usr / include / CC directory. Files in other directories are denoted by their path names, enclosed in quotes. Therefore, in the following commands:

```
#include "math1.h"
#include "/usr/bs/math2.h"
```

includes the math1.h file from the user's current directory and the math2.h file from the / usr / bs directory.

Here's a small, complete example in which a string is defined in one file and printed in another. The required types are defined in the header.h file :

```
// header.h

extern char * prog_name;
extern void f ();
```

The main.c file is the main program:

```
// main.c

# include "header.h"
char * prog_name = "primitive but complete example";
int main ()
{
        f ();
}
```

and the line is printed by the function from the fc file:

```
// fc
#include <stream.h>
#include "header.h"
void f ()
{
        cout << prog_name << '\ n';
}
```

When starting the C ++ translator and passing it the required parameter files, different implementations may use different name extensions for C ++ programs. On the author's machine, the translation and launch of the program looks like this:

```
$ CC main.c fc -o silly
$ silly
primitive but complete example
$
```

In addition to separate translation, the concept of modularity in C ++ is supported by classes ($$ 5.4).

1.4 Support for data abstraction

Support for programming with data abstraction basically comes down to being able to define a set of operations (functions and operations) on a type. All calls to objects of this type are limited to operations from the specified set. However, with such capabilities, the programmer soon discovers that some more language extensions are needed to make the definition and use of new types more convenient . Operation overloading is a good example of such an extension .

1.4.1 Initialization and deletion

When the view of a type is hidden, it is necessary to give the user a means to initialize variables of that type. The simplest solution is to call some function to initialize it before using the variable.

For example:

```
class vector
{
            // ...
            public:
                        void init (init size); // call init () before the first
                                                                    // using a vector
object
                        // ...
};

void f ()
{
            vector v;
            // v cannot be used yet
            v.init (10);
            // now you can
}
```

But this is an ugly and error-prone decision. It would be better if the creator of the type defines some special function to initialize variables . If there is such a function, then two independent operations of allocation and initialization of a variable are combined in one (sometimes it is called installation or just construction). The initialization function is called a constructor. The constructor stands out from all other functions of this class in that it has the same name as the class itself. If objects of a certain type are built non-trivially, then one more additional operation is needed to delete them after the last use. The delete function in C ++ is called a destructor. The destructor has the same name as its class, but is preceded by a ~ (in C ++, this character is used for a complement operation). Let's give an example:

```
class vector
{
        int sz; // number of elements
        int * v; // pointer to integers
        public:
                vector (int); // constructor
                ~ vector (); // destructor
                int & operator [] (int index); // indexing operation
};
```

The vector class constructor can be used for error control and memory allocation:

```
vector :: vector (int s)
{
        if (s <= 0)
                error ("invalid vector size");
        sz = s;
        v = new int [s]; // place an array of s integers
}
```

The vector destructor frees the used memory:

```
vector :: ~ vector ()
{
        delete [] v; // free the array on which
                                        // configured pointer
v
}
```

The C ++ implementation is not required to free the memory allocated with new if no longer referenced by any pointer (in other words, automatic garbage collection is not required). Instead, you can define your own memory management functions in the class without user intervention . This is a typical use of constructors and destructors, although there are many non-memory-related uses of these functions (see, for example, $$ 9.4).

1.4.2 Assignment and Initialization

For many types, the task of managing them is reduced to the construction and destruction of objects associated with them, but there are types for which this is not enough. Sometimes it is necessary to manage all copy operations. Let's go back to the vector class:

```
void f ()
{
        vector v1 (100);
        vector v2 = v1; // build a new vector v2,
                                              // initialized v1
        v1 = v2; // v2 is assigned to v1
        // ...
}
```

It should be possible to define the interpretation of the v2 initialization and v1 assignment operations . For example, in the description:

```
class vector
{
        int * v;
        int sz;
        public:
                // ...
                void operator = (const vector &); // assignment
                vector (const vector &); // initialization
};
```

specifies that the assignment and initialization of objects of type vector should be performed using user-defined operations. The assignment can be defined like this:

```
void vector :: operator = (const vector & a)
        // size control and copying elements
{
        if (sz! = a.sz)
```

```
                    error ("invalid vector size for =");
            for (int i = 0; i <sz; i ++) v [i] = av [i];
   }
```

Since this operation uses the "old value" of the vector for the assignment , the initialization operation must be specified by another function, for example, like this:

```
   vector :: vector (const vector & a)
           // initialize the vector with the value of another vector
   {
            sz = a.sz; // same size
            v = new int [sz]; // allocate memory for the array
            for (int i = 0; i <sz; i ++) // copy elements
            v [i] = av [i];
   }
```

In C ++, a constructor of the form T (const T &) is called a copy constructor for type T. It performs any initialization of objects of type T using the value of some other object of type T. In addition to explicit initialization, constructors of the form T (const T &) are used to pass parameters over value and get the value returned by the function.

1.4.3 Type templates

Why would a programmer want to define a type such as a vector of integers? Typically, he needs a vector of elements whose type is unknown to the creator of the Vector class. Therefore, it is necessary to be able to define the type of the vector so that the type of elements in this definition is involved as a parameter denoting the "real" types of elements:

```
   template <class T> class Vector
   {// vector of elements of type T
            T * v;
            int sz;
            public:
                    Vector (int s)
                    {
                            if (s <= 0)
                                    error ("invalid size for Vector");
                            v = new T [sz = s];
```

```
                                                    // allocate memory
for array s of type T
              }
              T & operator [] (int i);
              int size () {return sz; }
       // ...
};
```

This is the definition of a type template. It specifies how to get a family of similar classes. In our example, the Vector template shows how you can get a Vector class for a given element type. This description differs from the usual class description by the presence of the initial template <class T> construction , which shows that it is not a class that is described, but a type template with a given type parameter (here it is used as an element type). Now you can define and use vectors of different types:

```
void f ()
{
       Vector <int> v1 (100); // vector of 100 integers
       Vector <complex> v2 (200); // vector of 200
                                                    // complex numbers
       v2 [i] = complex (v1 [x], v1 [y]);
       // ...
}
```

The capabilities that a type template implements are sometimes referred to as parametric types or generic objects. It is similar to the capabilities found in the Clu and Ada languages. Using a type template does not incur any additional overhead compared to using a class in which all types are specified directly.

1.4.4 Handling Exceptions

As programs grow, and especially with the active use of libraries, there is a need for standard error handling (or, more broadly, "special situations"). The languages Hell, Algol-68 and Clu support a standard way of handling exceptions.

Let's go back to the vector class again. What should be done when an index value is passed to an indexing operation that is outside the bounds of an array ? The creator of the vector class does not know what the user is counting on in this case, and the user cannot find such an error (if he could, this error would not occur at all). The way out is this: the creator of the class detects an error overflowing the array boundary, but only reports it to an unknown user. The user himself takes the necessary measures.

For example:

```
class vector {
        // define the type of possible exceptions
        class range {};
        // ...
};
```

Instead of calling the error function in vector :: operator [] (), you can jump to the part of the program that handles exceptions. This is called "throw the exception":

```
int & vector :: operator [] (int i)
{
        if (i <0 || sz <= i) throw range ();
        return v [i];
}
```

As a result, information placed there during function calls will be fetched from the stack until an exception handler with the range type for the vector class (vector :: range) is found; it will be executed.

An exception handler can only be defined for a special block:

```
void f (int i)
{
        try
        {
                // special situations are handled in this block
                // using the handler defined below
                vector v (i);
                // ...
    v [i + 1] = 7; // leads to exception range
   // ...
    g (); // may lead to exception range
                                                        // on some vectors
        }
```

```
            catch (vector :: range)
            {
                        error ("f (): vector range error");
                        return;
            }
}
```

Using exceptions makes error handling more streamlined and understandable. We postpone the discussion and details until Chapter 9.

1.4.5 Type conversions

User-defined type conversions, such as the floating-point to complex conversion required for the complex (double) constructor, have proven to be very useful in C ++. The programmer can specify these conversions explicitly, or can rely on the translator, which performs them implicitly when they are necessary and unambiguous:

```
complex a = complex (1);
complex b = 1; // implicitly: 1 -> complex (1)
a = b + complex (2);
a = b + 2; // implicitly: 2 -> complex (2)
```

Type conversions are needed in C ++ because mixed-type arithmetic is the norm for languages used in numeric problems. In addition, most of the user-defined types used for "calculations" (for example, matrices, strings, machine addresses) allow natural conversion to other types (or from other types).

Type conversions help write your program more naturally:

```
complex a = 2;
complex b = a + 2; // this means: operator + (a, complex (2))
b = 2 + a; // this means: operator + (complex (2), a)
```

In both cases, only one function is needed to perform the "+" operation, and its parameters are uniformly interpreted by the language's type system. Moreover, the complex class is defined so that there is no need to change anything for integers to naturally and seamlessly generalize the concept of number.

1.4.6 Multiple implementations

The main tools that support object-oriented programming, namely derived classes and virtual functions, can also be used to support data abstraction, if you allow multiple implementations of the same type. Let's go back to the stack example:

```
template <class T>
class stack
{
        pub lic:
                virtual void push (T) = 0; // pure virtual
function
                virtual T pop () = 0; // pure virtual function
};
```

The notation = 0 indicates that no definition is required for the virtual function , and the stack class is abstract, i.e. it can only be used as a base class. Therefore, stacks can be used, but not created:

```
class cat {/ * ... * /};
stack <cat> s; // error: stack is an abstract class

void some_function (stack <cat> & s, cat kitty) // ok
{
        s.push ( kitty);
        cat c2 = s.pop ();
        // ...
}
```

Since the stack interface does not communicate anything about its presentation, the implementation details are completely hidden from stack users.

Several different stack implementations can be proposed. For example, the stack can be an array:

```
template <class T>
class astack: public stack <T>
{
        // true representation of a stack object
        // in this case it's an array
        // ...
        public:
```

```
                        astack (int size);
                        ~ astack ();
                        void push (T);
                        T pop ();
   };
```
You can implement the stack as a linked list:
```
   template <class T>
   class lstack: public stack <T>
   {
            // ...
   };
```
Now you can create and use stacks:
```
   void g ()
   {
            lstack <cat> s1 (100);
            astack <cat> s2 (100);
            cat Ginger;
            cat Snowball;
            some_function (s1, Ginger);
            some_function (s2, Snowball);
   }
```
Only the person who creates them (i.e. the g () function) should worry about how to represent stacks of different types, and the stack user (i.e. the author of some_function ()) is completely shielded from the details of their implementation. The price to pay for this flexibility is that all stack operations must be virtual functions.

1.5 Object-oriented programming support

Object-oriented programming is supported by classes together with an inheritance mechanism, as well as a mechanism for calling member functions depending on the true type of the object (the fact is that there are cases when this type is unknown at the translation stage). The mechanism for calling member functions is especially important. Equally important are tools that support data abstraction (we talked about them earlier). All the arguments in favor of data abstraction and methods based on it, which allow you to work with types naturally and beautifully, also apply to a language that supports object-oriented programming. The success of both methods depends on how types are constructed, how simple, flexible, and efficient they are. Object-oriented programming allows you to define more general and flexible user-defined types than you get if you only use data abstraction.

1.5.1 Calling mechanism

The main object-oriented programming support is the mechanism for calling a member function for a given object when its true type is unknown at the translation stage. For example, let there be a pointer p. How does the p-> rotate (45) call happen? Since C ++ is based on static type checking, the expression that specifies the call only makes sense if the rotate () function has already been described. Further, from the notation p-> rotate (), we see that p is a pointer to an object of a certain class, and rotate must be a member of this class. As with all static type checking, the call validation is necessary to make sure (as far as possible at the translation stage) that the types in the program are used in a consistent way. This ensures that the program is free from many types of errors.

So, the translator should know a class description similar to those given in $$ 1.2.5:

```
class shape
{
            // ...
            public:
                        // ...
                        virtual void rotate (int);
                        // ...
};
```

and the pointer p should be described, for example, like this:

```
T * p;
```

where T is the shape class or a class derived from it. Then the translator sees that the class of the object to which the pointer p is set does indeed have a rotate () function, and the function has a parameter of type int. So p-> rotate (45) is a valid expression.

Since shape :: rotate () was described as a virtual function, the mechanism for calling a virtual function must be used. To find out which of the rotate functions should be called, you need to get some service information from the object that was placed there when it was created before the call . Once it has been established which function to call, say circle :: rotate, it is called with the already mentioned type control. Usually , the function address table is used as the service information , and the translator converts the name rotate into the index of this table. With regard to this table-type object shape can be represented as follows:

```
center
vtbl:
            color & X :: draw
                            & Y :: rotate
...
...
```

Functions from the vtbl virtual function table allow you to work correctly with the object even in cases where neither the vtbl table nor the location of data in the part of the object designated ... is known in the calling function . Here, X and Y are the names of the classes that include the called functions. For a circle object, both the X and Y names are circle. A virtual function call can be inherently as efficient as a regular function call.

1.5.2 Type check

The need for type checking when accessing virtual functions can be a certain limitation for library developers. For example, it might be nice to provide the user with a "stack of anything" class. This cannot be done directly in C ++. However, by using type patterns and inheritance, you can get closer to the efficiency and ease of design and use of libraries that are inherent in dynamically typed languages . Such languages include, for example, Smalltalk, which can be used to describe a "stack of anything". Consider defining a stack using a pattern like:

```
template <class T> class stack
{
            T * p;
            int sz;
            public:
                        stack (int);
                        ~ stack ();
                        void push (T);
                        T & pop ();
   };
```
Without loosening static type control, you can use such a stack to store pointers to objects of type plane (airplane):
```
   stack <plane *> cs (200);
   void f ()
   {
              cs.push (new Saab900); // Error during broadcast:
                                                                   // requires plane *,
   but passed car *
              cs.push (new Saab37B);
                                                                   // great: Saab 37B -
   in fact
                                                                   // do the plane, i.e.
   plane type
              cs.pop () -> takeoff ();
              cs.pop () -> takeoff ();
   }
```
If there is no static typing, the above error will only be detected when the program is executed:
```
   // example of dynamic type checking
   // instead of static; this is not C ++
   Stack s; // the stack can store pointers to objects
                                                          // arbitrary type
   void f ()
   {
              s.push (new Saab900);
              s.push (new Saab37B);
              s.pop () -> takeoff (); // great: Saab 37B - plane
              cs.pop () -> takeoff (); // dynamic error:
                                                                   // the car cannot take
   off
   }
```
The way to determine if an operation on an object is usually more overhead than C ++ 's virtual function call mechanism .

By relying on static type checking and calling virtual functions, we end up in a different style of programming than relying only on dynamic type checking. A class in C ++ defines a well-defined interface for a set of objects of this and any derived class, while in Smalltalk a class defines only the minimum required number of operations, and the user has the right to use operations not defined in the class. In other words, a class in C ++ contains an accurate description of operations, and the user is guaranteed that only these operations are considered valid by the translator.

1.5.3 Multiple inheritance

If class A is the base class for B, then B inherits the attributes of A. i.e. B contains A plus something else. With this in mind, it becomes obvious that it is good when class B can inherit from the two base classes A1 and A2. This is called multiple inheritance.

Here's a typical example of multiple inheritance. Suppose there are two library classes displayed and task. The first represents tasks, information about which can be displayed on the screen using some monitor, and the second - tasks executed under the control of some dispatcher. The programmer can create his own classes, for example, the following:

```
class my_displayed_task: public displayed, public task
{
        // user text
};
class my_t ask: public task {
        // this task is not displayed
        // on the screen, because does not contain class displayed
        // user text
};
class my_displayed: public displayed
{
        // and this is not a task
        // since does not contain task class
        // user text
};
```

If only one class can be inherited, then only two of the three given classes are available to the user . The result is either duplication of parts of the program, or loss of flexibility, and, as a rule, both happens. The above example runs in C ++ without any additional time and memory overhead compared to programs in which no more than one class is inherited. Static type checking does not suffer from this either.

All ambiguities are revealed at the broadcast stage:

```
class task
{
        public:
                void trace ();
                // ...
};
class displayed
{
        public:
                void trace ();
                // ...
};
class my_displayed_task: public displayed, public task
{
        // trace () is not defined in this class
};
void g (my_displayed_task * p)
{
        p -> trace (); // error: ambiguity
}
```

This example shows the differences between C ++ and the object-oriented dialects of the Lisp language, which have multiple inheritance. In these dialects, ambiguity is resolved as follows: either the order of description is considered essential , or objects with the same name in different base classes are considered identical , or combined methods are used when the coincidence of objects is a fraction of base classes combined with a more complex way for derived classes. In C ++, ambiguity is generally resolved by introducing another function:

```
class my_displayed_task: public displayed, public task
{
        // ...
        public:
                void trace ()
                {
                        // user text
                        displayed :: trace (); // call trace () from
displayed
                        task :: trace (); // call trace () from task
                }
                // ...
};
void g (my_displayed_task * p)
{
        p -> trace (); // it's okay now
}
```

1.5.4 Encapsulation

Let a class member (whether a member function or a member representing the data) need to be protected from "unauthorized access". How to reasonably limit the set of functions that such a member will be available to? The obvious answer for languages that support object-oriented programming is that all the operations that are defined for that object have access , in other words, all member functions. For example:

```
class window
{
        // ...
        protected:
                Rectangle inside;
                // ...
};
class dumb_terminal: public window
{
                // ...
                public:
                        void prompt ();
                        // ...
};
```

Here, in the base class window, the inside member of type Rectangle is described as protected, but member functions of derived classes, such as dumb_terminal :: prompt (), can refer to it and figure out what kind of window they are working with. For all other functions, the window :: inside member is not available.

This approach combines a high degree of security (indeed, you are unlikely to "accidentally" define a derived class) with the flexibility required for programs that create classes and use their hierarchy (indeed, "for yourself" you can always provide access to protected members).

The implicit consequence of this is that you cannot make a complete and definitive list of all the functions that will have access to a protected member, because you can always add another one by defining it as a member function in a new derived class. For the data abstraction method, this approach is often not very acceptable. If the language focuses on the data abstraction method , then the obvious solution for it is the requirement to specify in the class description a list of all functions that need access to the member. In C ++ , the description of private members is used for this purpose. It was also used in the descriptions of the complex and shape classes.

The importance of encapsulation i.e. enclosing members in a protective shell increases sharply with the growth of the program size and the increasing spread of application areas. $$ 6.6 discusses the language 's encapsulation capabilities in more detail .

1.6 The Limits of Perfection

C ++ was designed to be the "best C" language to support data abstraction and object-oriented programming. However, it should be suitable for most basic system programming tasks.

The main challenge for a language that was built around data hiding, data abstraction, and object-oriented programming is that in order to be a general- purpose language , it must:

- go by traditional cars;
- coexist with traditional operating systems and languages;
- compete with traditional programming languages in the efficiency of program execution;
- be fit in all major areas of the application.

This means that there should be opportunities for efficient numeric operations (floating point arithmetic without much overhead, otherwise the user will prefer Fortran) and a means of such memory access that will allow writing device drivers in this language. In addition, you need to be able to write function calls in a rather unusual notation used for calls in traditional operating systems. Finally, it should be possible from a language that supports object-oriented programming to call functions written in other languages, and from other languages to call a function in this language that supports object-oriented programming.

Further, one cannot count on the widespread use of the desired programming language as a general-purpose language if its implementation relies entirely on capabilities that are absent in machines with traditional architecture.

If you do not introduce low-level capabilities into the language, then you will have to use some low-level languages, for example, C or assembler , for the main tasks of most areas of the application . But C ++ was designed with the expectation that you can do everything that is allowed in C in it, and without increasing the execution time. In general, C ++ was designed on the principle that there should not be any additional time and memory expenditures , unless the programmer himself explicitly wishes it.

The language was designed with modern translation methods in mind, which ensure program consistency, efficiency, and compact presentation. The main means of dealing with the complexity of programs is seen, first of all, strong type control and encapsulation. This is especially true for large programs created by many people. The user may not be one of the creators of such programs, and may not be a programmer at all. Since no real program can be written without the support of libraries created by other programmers, this last remark applies to almost all programs.

C ++ was designed to support the principle that every program is a model of some existing concepts in reality, and a class is a concrete representation of a concept taken from the application domain ($$ 12.2). Therefore, classes permeate the entire C ++ program, and strict requirements are imposed on the flexibility of the concept of a class, the compactness of class objects and the efficiency of their use. If it is inconvenient or too expensive to work with classes , then they simply will not be used, and programs will degenerate into programs in "better C". This means that the user will not be able to enjoy the opportunities for which, in fact, the language was created .

CHAPTER 2. DESCRIPTIONS AND CONSTANTS

"Perfection is achievable only at the moment of collapse."
(S.N. Parkinson)

This chapter describes the main types (char, int, float, etc.) and how to build new types (functions, vectors, pointers , etc.) based on them . The description introduces a name into the program, specifying its type and possibly an initial value. This chapter introduces concepts such as description and definition, types, scope of names, lifetime of objects. The notation of literal C ++ constants and ways of setting symbolic constants are given. Examples are provided that simply demonstrate the capabilities of the language. More meaningful examples illustrating the capabilities of expressions and operators in the C ++ language will be given in the next chapter. This chapter only mentions means for defining custom types and operations on them. These are discussed in Chapters 5 and 7.

2.1 DESCRIPTIONS

The name (identifier) must be described before it is used in a C ++ program. This means that you need to specify its type so that the translator knows what kind of objects the name refers to. Below are a few examples to illustrate the variety of descriptions:

```
char c h;
int count = 1;
char * name = "Njal";
struct complex {float re, im; };
complex cvar;
extern complex sqrt (complex);
extern int error_number;
typedef complex point;
float real (complex * p) {return p-> re; };
const double pi = 3.1415926535897932385;
struct user;
template <class T> abs (T a) {return a <0? -a: a; }
enum beer {Carlsberg, Tuborg, Thor};
```

From these examples, you can see that the role of descriptions is not limited to binding a type to a name. Most of these descriptions are at the same time definitions, i.e. they create an object that the name refers to. For ch, count, name and cvar, such an object is a memory element of the appropriate size. This element will be used as a variable and is said to have memory allocated for it. For real, such an object will be the given function. For the constant pi, the object will be the number 3.1415926535897932385. For complex, the object will be a new type. For point, the object is of type complex, so point becomes synonymous with complex. The following descriptions are no longer definitions:

```
extern complex sqrt (complex);
extern int error_number;
struct user;
```

This means that the objects introduced by them must be defined somewhere else in the program. The body of the sqrt function must be specified in some other description. Memory for an int error_number variable should be allocated as a result of a different error_number definition. There must be some other description of the user type, from which you can understand what this type is. A C ++ program should have only one definition for each name, but there can be many definitions. However, all descriptions must be consistent on the type of object entered in them. Therefore, the snippet below contains two errors:

```
int count;
int count; // error: override
extern int error_number;
extern short error_number; // error: type mismatch
```

But the following snippet does not contain a single error (see # 4.2 for using extern):

```
extern int error_number;
extern int error_number;
```

Some descriptions indicate the "values" of objects that they define:

```
struct complex {float re, im; };
typedef complex point;
float real (complex * p) {r eturn p-> re};
const double pi = 3.1415926535897932385;
```

For types, functions, and constants, the "value" remains unchanged; for data that is not constants, the initial value can subsequently change:

```
int count = 1;
char * name = "Bjarne";
// ...
coun t = 2;
name = "Marian";
```

Of all the definitions, only the following does not specify a value:

```
char ch;
```

Any description that gives a meaning is a definition.

2.1.1 Scope

The description defines the scope of the name. This means that the name can be used only in a certain part of the program text. If a name is described in a function (usually called a "local name"), then the scope of the name extends from the point of the description to the end of the block in which the description appears. If the name is not found in a function or class description (usually referred to as a "global name"), then the scope extends from the point of the declaration to the end of the file in which the description appears. The description of the name in the block can hide the description in the enclosing block or the global name; those. the name can be redefined to represent another object within the block. After exiting the block, the previous value of the name (if any) is restored. Let's give an example:

```
int x; // global x
void f ()
{
            int x; // local x hides global x
            x = 1; // assign to local x
            {
                        int x; // hides the first local x
                        x = 2; // assign to second local x
            }
            x = 3; // assign to the first local x
}
int * p = & x; // take the address of the global x
```

In large programs, name redefinition cannot be avoided. Unfortunately, such a redefinition can easily be overlooked by a person. The resulting errors are not easy to find, perhaps because they are rare enough. Therefore, name redefinition should be kept to a minimum. If you designate global variables or local variables in a large function with names such as i or x, then you yourself are asking for trouble. It is possible, using the scope resolution operation ::, to refer to the hidden global name, for example:

```
int x;
void f2 ()
{
        int x = 1; // hides the global x
        :: x = 2; // assignment to global x
}
```

There is no option to use a hidden local name.

The scope of a name begins at the point of its description (at the end of the descriptor, but even before the start of the initializer - see $$ R.3.2). This means that the name can be used even before its initial value is set. For example:

```
int x;
void f3 ()
{
        int x = x; // erroneous assignment
}
```

Such an assignment is invalid and meaningless. If you try to broadcast this program, you will receive a warning: "use before setting value". However, without using the :: operator, you can use the same name to refer to two different block objects. For example:

```
int x = 11;
void f4 () // perverted example
{
        int y = x; // global x
        int x = 22;
        y = x; // local x
}
```

The variable y is initialized to the value of the global x, i.e. 11, and then it is assigned the value of the local variable x, i.e. 22. The names of the formal parameters of the function are considered described in the largest block of the function, so there is an error in the description below:

```
void f5 (int x)
{
        int x; // mistake
}
```

Here x is defined twice in the same scope. This, although not too rare, is a rather subtle mistake.

2.1.2 Objects and Addresses

It is possible to allocate memory for unnamed "variables" and use those variables. It is even possible to assign such strange looking "variables", for example, * p [a + 10] = 7. Hence, there is a need to name "something stored in memory". You can give a suitable quote from the reference manual: "Any object is a certain area of memory, and an expression that refers to an object or function is called an address" ($$ R.3.7). The word address (lvalue - left value, ie the value on the left) was originally assigned the meaning "something that can be on the left in the assignment." An address can also refer to a constant (see $$ 2.5). An address that was not described with the const specification is called a mutable address.

2.1.3 Lifetime of objects

Unless the programmer explicitly intervenes, the object will be created when its definition appears and destroyed when it disappears from scope. Objects with global names are created, initialized (and only once) and exist until the end of the program. If local objects are described with the static service word, then they also exist until the end of the program. They are initialized when for the first time control "passes through" the description of these objects, for example:

```
int a = 1;
void f ()
{
     int b = 1; // initialized on every call to f ()
        static int c = a; // initialized only once
        cout << "a =" << a ++
                          << "b =" << b ++
                          << "c =" << c ++ << '\ n';
}
int main ()
{
        while (a <4) f ();
}
```

Here the program will produce this output:

```
a = 1 b = 1 c = 1
a = 2 b = 1 c = 2
a = 3 b = 1 c = 3
```

" The #include <iostream> macro has been removed from the examples in this chapter for brevity . It is needed only in those of them that give the result.

The operation "++" is an increment, that is, a ++ means: add 1 to the variable a.

A global or static local variable that has not been explicitly initialized is implicitly initialized to zero (# 2.4.5). Using the new and delete operations, the programmer can create objects, the lifetime of which he manages himself (see $$ 3.2.6).

2.2 NAMES

The name (identifier) is a sequence of letters or numbers. The first character must be a letter. The underscore _ is also considered a letter. The C ++ language does not limit the number of characters in a name. But the implementation includes software components that the creator of the translator cannot control (for example, the loader), and, unfortunately, they can set restrictions. In addition, some system programs required to run a C ++ program can expand or contract the set of characters allowed in an identifier. Extensions (for example, using $ in the name) can break the portability of the program. You cannot use C ++ service words as names (see $$ R.2.4), for example:

```
hello this_is_a_mo st_unusially_long_name
DEFINED foO bAr u_name HorseSense
var0 var1 CLASS _class ___
```

Now we will give examples of character sequences that cannot be used as identifiers:

```
012 a fool $ sys class 3var
pa y.due foo ~ bar .name if
```

Uppercase and lowercase letters are considered different, so Count and count are different names. But choosing names that are almost indistinguishable from each other is unwise. All names beginning with an underscore are reserved for use in the implementation itself or in programs that run in conjunction with the worker, so it is extremely frivolous to insert such names into your program. When parsing a program, the translator always tries to select the longest sequence of characters that form a name, therefore var10 is a name, not the name var and the number 10. For the same reason elseif is one name (service), not two service names else and if.

2.3 TYPES

Each name (identifier) in the program has a type associated with it. It specifies those operations that can be applied to a name (that is, to an object that denotes a name), as well as the interpretation of these operations. Here are some examples:

```
int error_number;
float real (complex * p);
```

Since the variable error_number is described as an int (integer), it can be assigned and its values can also be used in arithmetic expressions. The real function can be called with a parameter containing the address complex. You can get addresses of both a variable and a function. Some names, like int and complex in our example, are type names . Usually the type name is needed to give some other name in the type description . In addition, the type name can be used as an operand in the operations sizeof (it is used to determine the size of memory required for objects of this type) and new (it can be used to place an object of this type in free memory). For example:

```
int main ()
{
        int * p = new int;
        cout << "sizeof (int) =" << sizeof (int) '\ n';
}
```

Another type name can be used in the operation of explicit conversion of one type to another ($$ 3.2.5), for example:

```
float f;
```

```
char * p;
// ...
long ll = long (p); // converts p to long
int i = int (f); // converts f to int
```

2.3.1 Basic types

The basic C ++ types represent the most common units of machine memory and all the basic ways of working with them. It:

```
char
sho rt int
int
long int
```

The enumerated types are used to represent different sized integers. Floating point numbers are represented by types:

```
float
double
long double
```

The following types can be used to represent unsigned integers, booleans, bit arrays, and more:

```
unsigned char
unsigned short int
unsigned int
unsigned long int
```

The following are the types that are used to explicitly specify signed types:

```
signed char
signed short int
signed int
signed long int
```

Since by default values of type int are considered signed, the corresponding types with signed are synonyms of types without this special word. But the signed char type is of special interest: all 3 types - unsigned char, signed char and just char are considered different (see also $$ R.3.6.1).

For brevity (and this does not entail any consequences), the word int can be omitted in multi-word types, i.e. long means long int, unsigned means unsigned int. In general, if the type is not specified in the description, then it is assumed that it is int. For example, below are two definitions of an object of type int:

```
const a = 1; // carelessly, no type specified
static x; // same case
```

However , it is generally bad style to omit a type in a description in the hope that it will be int by default . It can cause subtle and undesirable effects (see $$ R.7.1).

The char type is most suitable for storing and working with characters. It usually represents an 8-bit byte. The sizes of all objects in C ++ are multiples of the size of a char, and by definition the value of sizeof (char) is the same as 1. Depending on the machine, a char value can be a signed or unsigned integer. Of course, an unsigned char value is always unsigned, and by explicitly specifying this type, we improve the portability of the program. However, using unsigned char instead of char can slow down program execution speed. Naturally, a signed char value is always signed.

Several integers, several unsigned types and several floating point types have been introduced into the language , so that the programmer can make fuller use of the command system capabilities. Many machines have significant differences in memory allocation, access time, and computation speed for values of various basic types. As a rule, knowing the features of a particular machine, it is easy to choose the optimal basic type (for example, one of the int types) for a given variable. However, writing a truly portable program that takes advantage of such low-level capabilities is not easy. For sizes of the main types, the following relationships are met:

```
1 == sizeof (char) <= sizeof (short) <= sizeof (int) <= sizeof (long)
sizeo f (float) <= sizeof (double) <= sizeof (long double)
sizeof (I) == sizeof (signed I) == sizeof (unsigned I)
```

Here I can be of type char, short, int, or long. In addition, it is guaranteed that char is represented by at least 8 bits, short by at least 16 bits, and long by at least 32 bits. The char type is sufficient to represent any character in the character set of the given machine. But this only means that the char type can represent integers in the range 0..127. To assume more is risky.

The unsigned integer types are best suited for programs that treat memory as an array of bits. But, as a rule, using unsigned instead of int does not do anything good, although in this way they hoped to win one more place for representing positive integers. By describing a variable as unsigned, you cannot guarantee that it will only be positive, since implicit type conversions are allowed, for example:

```
unsigned surprise = -1;
```

This definition is valid (although the compiler may issue a warning about it).

2.3.2 Implicit type conversion

In assignment and expression, basic types can be used together quite freely . The values are converted wherever possible so that information is not lost. The exact conversion rules are given in $$ R.4 and $$ R.5.4.

Still, there are situations when information can be lost or even distorted. A potential source of such situations is assignments, in which a value of one type is assigned to a value of another type, and the latter uses fewer bits in its representation . Suppose the following assignments are made on a machine that uses integers in two 's complement and a character is 8 bits long:

```
int i1 = 256 + 255;
char ch = i1 // ch == 255
in t i2 = ch; // i2 ==?
```

In the assignment ch = i1 one bit is lost (and the most important one!), And when we assign the value to the variable i2, the variable ch has the value "all ones", i.e. 8 unit digits. But what meaning will i2 take? On a DEC VAX machine, in which char represents signed values, it will be -1, and on a Motorola 68K machine, in which char is unsigned, it will be 255. In C ++ there are no dynamic controls for such situations, and control at the translation stage at all too complicated, so you have to be careful.

2.3.3 Derived types

Based on the basic (and user-defined) types, you can describe using the following operations:

```
* pointer
& link
[] array
() function
```

and also by defining structures, define other derived types. For example:

```
int * a;
float v [10];
char * p [20]; // array of 20 character pointers
void f (int);
struct str {short length; char * p; };
```

The rules for constructing types using these operations are explained in detail in $$ R.8. The key idea is that the description of an object of a derived type should reflect its use, for example:

```
int v [10]; // vector description
i = v [3]; // use vector element
int * p; // pointer description
i = * p; // use the specified object
```

The notation used for derived types is difficult to understand just because the operations * and & are prefix, and [] and () are postfix. Therefore, when specifying types, if the priorities of operations do not meet the purpose, parentheses should be used. For example, the priority of the operation [] is higher than that of *, and we have:

```
int * v [10]; // array of pointers
int (* p) [10]; // array pointer
```

Most people just remember what the most commonly used types look like . You can describe several names at once in one description. Then it contains instead of one name a list of names separated from each other by commas . For example, you can describe two variables of an integer type like this:

```
int x, y; // int x; int y;
```

When we describe derived types, remember that declaration operations apply only to the given name (and not at all to all other names of the same description). For example:

```
int * p, y; // int * p; int y; BUT NOT int * y;
int x, * p; // int x; int * p;
int v [10], * p; // int v [10]; int * p;
```

But such descriptions are confusing to the program and should probably be avoided.

2.3.4 The void type

The void type is syntactically equivalent to the base types, but it can only be used on a derived type. There are no void objects. It specifies pointers to objects of unknown type or functions that do not return a value.

```
void f (); // f returns no value
void * pv; // pointer to an object of unknown type
```

A pointer of an arbitrary type can be assigned to a variable of the void * type. At first glance, it is difficult to find a use for this, since indirection (dereferencing) is not allowed for void * . However, it is on this limitation that the void * type is based. It is ascribed to parameters of functions that do not need to know the true type of those parameters. Typeless objects returned by functions are also of type void * . To use such objects, you must perform an explicit type conversion operation . These functions are usually found at the lowest levels of the system, which control hardware resources. Let's give an example:

```
void * malloc (unsigned size);
void fre e (void *);
void f () // C style memory allocation
{
        int * pi = (int *) malloc (10 * sizeof (int));
        char * pc = (char *) malloc (10);
        // ...
        free (pi);
        free (pc);
}
```

Notation: (type) expression - used to specify the operation of converting an expression to a type, so before assigning pi, the void * type returned in the first call to malloc () is converted to int. The example is written in archaic style; a better style of managing free memory allocation is shown in $$ 3.2.6.

2.3.5 Pointers

For most types of T, a pointer to T is of type T *. This means that a variable of type T * can store the address of an object of type T. Pointers to arrays and functions, unfortunately, require a more complex notation:

```
int * pi;
```

```
char ** cpp; // pointer to pointer to char
int (* vp) [10]; // pointer to an array of 10 integers
int (* fp) (char, char *); // pointer to a function with parameters
                                                  // char and char *
returning int
```

The main operation on pointers is indirect reference (dereferencing), i.e. reference to the object to which the pointer is set . This operation is usually referred to simply as indirection. The indirection operator * is a prefix unary operator. For example:

```
char c1 = 'a';
char * p = & c1; // p contains address c1
char c2 = * p; // c2 = 'a'
```

The variable pointed to by p is c1, and the value stored in c1 is 'a'. Therefore, the value * p assigned to c2 is 'a'. Some arithmetic operations can also be performed on pointers. As an example, here is a function that counts the number of characters in a null-terminated string (which is ignored):

```
int strlen (char * p)
{
int i = 0;
        while (* p ++) i ++;
        return i;
}
```

You can determine the length of a string in another way: first find its end, and then subtract the address of the beginning of the string from the address of its end.

```
int strlen (ch ar * p)
{
        char * q = p;
        while (* q ++);
        return qp-1;
}
```

Function pointers are widely used; they are discussed specifically in $$
4.6.9

2.3.6 Arrays

For type TT, [size] is an "array of size elements of type T" type. Elements are indexed from 0 to size-1. For example:

```
float v [3]; // an array of three floating point numbers:
                                                    // v [0], v [1], v [2]
int a [2] [5]; // two arrays of five integers each
char * vpc; // array of 32 character pointers
```

You can write a loop like this, which prints the whole values of uppercase letters:

```
extern "C" int strlen (const char *); // from <string.h>
char alpha [] = "abcdefghijklmnopqrstuvwxyz";
main ()
{
        int sz = strlen (alpha);
        for (int i = 0; i <sz; i ++) {
                char ch = alpha [i];
                cout << '\' '<< ch <<' \ "
                                << "=" << int (ch )
                                << "= 0" << oct (ch)
                                << "= 0x" << hex (ch) << '\ n';
        }
}
```

Here the oct () and hex () functions return their integer parameter in octal and hexadecimal form, respectively. Both functions are described in <iostream.h>. The strlen () function from <string.h> is used to count the number of characters in alpha , but the size of the alpha array ($$ 2.4.4) could be used instead . For many ASCII characters, the result is:

```
'a' = 97 = 0141 = 0x61
'b' = 98 = 0142 = 0x62
'c' = 99 = 0143 = 0x63
...
```

Note that you do not need to specify the size of the alpha array: the translator will set it by counting the number of characters in the string specified as an initializer. Setting a character array as an initializer string is convenient, but unfortunately the only way to use strings in this way . Assigning a string to an array is unacceptable, since assignment to arrays is not defined in the language, for example:

```
char v [9];
v = "a string"; // mistake
```

Classes allow to implement string representation with a wide range of operations (see $$ 7.10).

Obviously, strings are only suitable for initializing character arrays; for other types, you have to use a more complex notation. However, it can also be used for character arrays. For example:

```
int v1 [] = {1, 2, 3, 4};
int v2 [] = {'a', 'b', 'c', 'd'};
char v3 [] = {1, 2, 3, 4};
char v4 [] = {'a', 'b', 'c', 'd'};
```

Here v3 and v4 are arrays of four (not five) characters; v4 is not null-terminated as required by the string convention and most library functions. By using such a char array, we ourselves are setting the stage for future mistakes.

Multidimensional arrays are represented as arrays of arrays. However, you cannot use a comma when setting the boundary values of indices, as is done in some languages. The comma is a special operation for listing expressions (see $$ 3.2.2). You can try setting this description:

```
int bad [5,2]; // mistake
```

or such

```
int v [5] [2];
int bad = v [4,1]; // mistake
int good = v [4] [1]; // right
```

An array of two elements is described below , each of which is, in turn, an array of 5 elements of type char:

```
char v [2] [5];
```

In the following example, the first array is initialized with the first five letters of the alphabet and the second with the least significant five digits.

```
char v [2] [5] = {
            {'a', 'b', 'c', 'd', 'e'},
            {'0', '1', '2', '3', '4'}
};
main () {
```

```
            for (int i = 0; i <2; i ++) {
                        for (int j = 0; j <5; j ++)
                                    cout << "v [" << i << "] [" << j
                                                            << "] =" << v [i] [j] << "";
                                    cout << '\ n';
            }
   }
```
As a result, we get:

```
   v [0] [0] = av [0] [1] = bv [0] [2] = cv [0] [3] = dv [0] [4] = e
   v [1] [0] = 0 v [1] [1] = 1 v [1] [2] = 2 v [1] [3] = 3 v [1] [4] = 4
```

2.3.7 Pointers and Arrays

Pointers and arrays in C ++ are closely related. The name of the array can be used as a pointer to its first element, so the example with the alpha array can be written like this:

```
   int main ()
   {
            char alpha [] = " abcdefghijklmnopqrstuvwxyz";
            char * p = alpha;
            char ch;
            while (ch = * p ++)
                        cout << ch << "=" << int (ch)
                                                << "= 0" << oct (ch) << '\ n';
   }
```
You can also define p as follows:

```
   char * p = α [0];
```
This equivalence is widely used when calling functions with an array parameter, which is always passed as a pointer to its first element. Thus, in the following example, both calls to strlen pass the same value:

```
   void f ()
   {
            extern "C" int strlen (const char *); // from <st ring.h>
            char v [] = "Annemarie";
            char * p = v;
            strlen (p);
            strlen (v);
```

```
}
```

But the catch is that there is no way to get around this: there is no way to describe the function in such a way that when it is called, the array v is copied ($$ 4.6.3).

The result of applying the arithmetic operations +, -, ++ or - to pointers depends on the type of the specified objects. If such an operation is applied to a pointer p of type T *, then p is considered to point to an array of objects of type T. Then p + 1 denotes the next element of this array, and p-1 denotes the previous element. It follows that the value (address) p + 1 will be sizeof (T) bytes larger than the value p. Therefore, in the next program

```
main ()
{
            char cv [10];
            int iv [10];
            char * pc = cv;
            int * pi = iv;
            cout << "char *" << long (pc + 1) -long (pc) << '\ n';
            cout << "int *" << long (pi + 1) -long (pi) << '\ n';
}
```

Taking into account the fact that on the author's machine (Maccintosh) a character occupies one byte, and an integer occupies four bytes, we get:

```
char * 1
int * 4
```

Before subtraction, the pointers were explicitly converted to the long type ($$ 3.2.5). It was used for conversion instead of the "obvious" int type, because in some C ++ implementations the pointer may not fit into int type (ie sizeof (int) <sizeof (char *)).

Pointer subtraction is only defined when they both point to the same array (although there is no way in the language to guarantee this). The result of subtracting one pointer from another is equal to the number (integer) of array elements between these pointers. You can add or subtract an integer value from a pointer ; in both cases, the result is a pointer. If a value is obtained that is not a pointer to an element of the same array to which the original pointer was set (or a pointer to the next element after the array), then the result of using such a value is undefined. Let's give an example:

```
void f ()
{
        int v1 [10];
        int v2 [10];
        int i = & v1 [5] - & v1 [3]; // 2
        i = & v1 [5] - & v2 [3]; // undefined result
        int * p = v2 + 2; // p == & v2 [2]
        p = v2-2; // * p is undefined
}
```

In general, complex pointer arithmetic is not required and is best avoided. It should be said that in most C ++ implementations there is no control over array boundaries. The description of an array is not self-sufficient, since it will not necessarily store the number of array elements. The notion of an array in C is essentially a low- level language notion . Classes help develop it (see $$ 1.4.3).

2.3.8 Structures

An array is a collection of elements of the same type, and a structure is a collection of elements of arbitrary (practically) types. For example:

```
struct address {
        char * name; // name "Jim Dandy"
        long number; // house number 61
        char * street; // "South Street"
        char * town; // city "New Providence"
        char * state [2]; // state 'N' 'J'
        int zip; // index 7974
};
```

This defines a new type called address that specifies the postal address. The definition is not general enough to cover all address cases, but it is a good example. Notice the semicolon at the end of the definition: this is one of the few cases in C ++ where a semicolon is required after the curly brace , so it's often overlooked.

Variables of type address can be described in the same way as any other variable, but using the operation. (dot) you can refer to individual members of the structure. For example:

```
address jd;
```

```
   jd.name = "Jim Dandy";
   jd.number = 61;
```

Variables of type struct can be initialized in the same way as arrays. For example:

```
   address jd = {
               "Jim Dandy",
               61, "South Street",
               "New Providence", { 'N', 'J'}, 7974
   };
```

But it is better to use the constructor for these purposes ($$ 5.2.4). Note that jd.state cannot be initialized with the string "NJ". After all, strings end with a null character '\ 0', which means there are three characters in the "NJ" line, and this is one more than it can fit in jd.state. Structured objects are often referenced with pointers using the -> operation. For example:

```
   void print_addr (address * p)
   {
               cout << p-> name << '\ n'
                                        << p-> number << '' << p-> street << '\ n'
                                        << p-> town << '\ n'
                                        << p-> state [0] << p-> sta te [1]
                                        << '' << p-> zip << '\ n';
   }
```

Objects of structured type can be assigned, passed as actual parameters to functions, and returned by functions as a result. For example:

```
   address current;
   address set_current (address next)
   {
               address prev = cur rent;
               current = next;
               return prev;
   }
```

Other valid operations, such as comparison (== and! =), Are undefined. However, the user can define these operations himself (see chapter 7).

The size of an object of a structured type is not necessarily the sum of the sizes of all its members. This is because on many machines it is required to place objects of certain types only by aligning them along some addressing system-dependent boundary (or simply because this alignment would be more efficient). A typical example is the alignment of a whole on a word boundary. As a result of alignment, "holes" may appear in the structure. So, on the author's machine already mentioned, sizeof (address) is 24, not 22, as one might expect.

It should also be mentioned that the type can be used immediately after it appears in the description, even before the entire description is completed. For example:

```
struct link {
        link * previous;
        link * successor;
};
```

However, new objects of a structure type cannot be described until a complete description is available. Therefore, the description

```
struct no_good {
        no_good member;
};
```

is erroneous (the translator is unable to set the size to no_good). To allow two (or more) structured types to refer to each other, you can simply describe the name of one of them as the name of some structured type. For example:

```
struct list; // will be defined later
struct link {
        link * pre;
        link * suc;
        list * member_of;
};
struct list {
        link * head;
};
```

Without the first description of list, the description of the link member would result in a syntax error. It is also possible to use the name of a structured type even before the type is defined, unless this usage requires knowing the size of the structure. For example:

```
class S; // 'S' is a name of some type
extern S a ;
S f ();
void g (S);
```

But the above descriptions can be used only after the type S has been defined:

```
void h ()
{
            S a; // error: S - unspecified
            f (); // error: S - unspecified
            g (a); // error: S - unspecified
}
```

2.3.9 Type equivalence

Two structural types are considered to be different even when they have the same members. For example, different types are defined below:

```
struct s1 {int a; };
struct s2 {int a; };
```

As a result, we have:

```
s1 x;
s2 y = x; // error: type mismatch
```

In addition, structural types differ from the main types, so we get:

```
s1 x;
int i = x; // error: type mismatch
```

It is, however, possible, without defining a new type, to specify a new name for the type. The description beginning with the typedef service word does not describe a variable of the specified type, but introduces a new name for the type. Let's give an example:

```
typedef char * Pchar;
Pchar p1, p2;
char * p3 = p1;
```

It is just a handy means of shortening the recording.

2.3.10 Links

A link can be thought of as another object name. Basically, references are used to set parameters and values returned by functions, as well as to overload operations (see $$ 7). The X & notation denotes a reference to X. For example:

```
int i = 1;
int & r = i;    // r and i refer to the same integer
int x = r; // x = 1
r = 2; // i = 2;
```

The link must be initialized, i.e. there must be something that it can mean. Remember that link initialization is completely different from assignment. Although you can specify operations on a link, none of them affects the link itself , for example,

```
int ii = 0;
int & rr = ii;
rr ++; // ii is incremented by 1
```

++ is allowed here, but rr ++ does not increment the rr itself ; instead ++ applies to integer, i.e. to variable ii. Therefore, after initialization, the value of the link cannot be changed: it always points to the object to which it was attached during its initialization. To get a pointer to the object denoted by the reference rr, you can write & rr. The obvious implementation of a reference is a constant pointer, which is used only for indirection. Then the initialization of the link will be trivial if the address is specified as the initializer (that is, the object whose address can be obtained; see $$ R.3.7). The initializer for type T must be an address. However, the initializer for & T may not be an address, or even a type T. In such cases , the following is done:

[1] firstly, if necessary, a type conversion is applied (see $$ R.8.4.3);

[2] then the resulting value is placed into a temporary variable;

[3] Finally, the address of this variable is used as the initializer of the link.

Let there be descriptions:

```
double & dr = 1; // error: address needed
const double & cdr = 1; // fine
```

This is interpreted like this:

```
double * cdrp; // link provided as a pointer
double temp;
temp = double (1);
cdrp = & temp;
```

References to variables and references to constants differ for the following reason: in the first case, creating a temporary variable is fraught with errors, since assigning to this variable means assigning a temporary variable, which may have disappeared by that time. Naturally, such problems do not exist in the second case. and constant references are often used as function parameters (see $$ R.6.3). A reference can be used for a function that changes the value of its parameter. For example:

```
void incr (int & aa) {aa ++; }
void f ()
{
        int x = 1;
        incr (x); // x = 2
}
```

By definition, parameter passing has the same semantics as initialization, so when the incr function is called, its parameter aa becomes a different name for x. It is better, however, to avoid functions that change their parameters so as not to confuse the program. In most cases, it is preferable for the function to return the result explicitly , or for a pointer type parameter to be used:

```
int next (int p) {return p + 1; }
void inc (int * p) {(* p) ++; }
void g ()
{
        int x = 1;
        x = next (x); // x = 2
        inc (& x); // x = 3
}
```

In addition to the above, using links, you can define functions used in both the right and left parts of the assignment. It usually finds its most interesting use when defining non-trivial user-defined types. Let's define a simple associative array as an example . Let's start by defining the structure

```
pair:
struct pair {
```

```
            char * name; // line
            int val; // integer
   };
```

The idea is that some integer value is associated with a string. It's not hard to write a find () search function that works with a data structure that represents an associative array. It contains a pair structure (pair: string and value) for each distinct line. In this example, it's just an array. To shorten the example, an extremely simple, albeit inefficient, algorithm is used:

```
   const int large = 1024;
   static pair vec [large + 1];
   pair * find (const char * p)
   / *
            // works with multiple "pair" pairs:
            // searches for p, if found, returns it "pair"
            // otherwise returns unused "pair"
   * /
   {
            for (int i = 0; vec [i] .name; i ++)
                     if (strcmp (p, vec [i] .name) == 0) return & vec [i];
            if (i == large) return & vec [large-1];
            return & vec [i];
   }
```

This function is used by the value () function, which implements an array of integers indexed by strings (although it's more common to index strings with integers):

```
   int & value (const char * p)
   {
            pair * res = find (p);
            if (res-> name == 0) { // until now the line has not been
   encountered,
                                                                      // means you need to
   initialize
                     res-> name = new char [strlen (p) +1];
                     strcpy (res-> name, p);
                     res-> val = 0; // initial value is 0
            }
            return res-> val;
}
```

For the given parameter (string), value () finds an object that represents an integer (not just the value of the corresponding integer) and returns a reference to it. These functions can be used like this:

```
const int MAX = 256; // longer than the length of the longest word
main ()            // counts the frequency of words in the input
stream
{
            char buf [MAX];
            while (cin >> buf) value (buf) ++;
            for (int i = 0; vec [i] .name; i ++)
                        cout << vec [i] .name << ":" << vec [i] .val << '\ n';
}
```

The while loop reads one word at a time from the standard input stream cin and writes it to buffer buf (see Chapter 10), incrementing the counter associated with the line being read each time. The counter is found in the associative array vec using the find () function. The for loop prints the resulting table of different words from cin along with their frequency. Having an input stream

```
aa bb bb aa aa bb aa aa
```

the program produces:

```
aa: 5
bb: 3
```

Using the template class and the [] ($$ 8.8) overloaded operator, it is easy to convert the array from this example to a real associative array.

2.4 LITERALS

In C ++, you can set values of all basic types: character constants, integer constants, and floating point constants. In addition, zero (0) can be used as a pointer value of arbitrary type, and character strings are constants of type char []. It is possible to define symbolic constants. A symbolic constant is a name that cannot be changed in its scope. In C ++, symbolic constants can be specified in three ways: (1) by adding the const in the definition, you can associate any value of an arbitrary type with the name; (2) a set of integer constants can be defined as an enumeration; (3) constant is the name of an array or function.

2.4.1 Integer constants

Integer constants can appear in four guises: decimal, octal, hexadecimal, and character constants. Decimal constants are used most often and look natural:

```
0 1234 976 12345678901234567890
```

A decimal constant is of type int if it fits into the memory allocated for an int, otherwise its type is long. The translator should warn about constants, the value of which exceeds the selected number format . A zero-starting constant followed by x (0x) is a hexadecimal number (base 16), and a zero-starting constant followed by a digit is an octal number (base 8). Here are some examples of octal constants:

```
0 02 077 0123
```

Their decimal equivalents are respectively: 0, 2, 63, 83. In hexadecimal notation, these constants look like this:

```
0x0 0x2 0x3f 0x53
```

The letters a, b, c, d, e, and f, or their equivalent capital letters, are used to represent the numbers 10, 11, 12, 13, 14, and 15, respectively. Octal and hexadecimal notation are most suitable for specifying a set of digits, and using them for ordinary numbers can have unexpected effects. For example, on a machine that represents an int as a 16-bit two's complement number, 0xffff is a negative decimal number -1. If more digits were used to represent an integer , then this would be the number 65535.

The U ending can be used to explicitly specify unsigned constants . Similarly, the ending L explicitly defines a constant of type long. For example:

```
void f (int);
void f (unsigned int);
void f (long int);
void g ()
{
        f (3); // call f (int)
        f (3U); // call f (unsigned int)
        f (3L); // call f (long int)
}
```

2.4.2 Floating point constants

Floating point constants are of type double. The translator should warn about constants whose value does not fit into the format chosen for representing floating point numbers. Here are some examples of floating point constants:

```
1.23 .23 0.23 1.1.1 1.2e10 1.23e-15
```

Note that there must be no spaces inside the floating point constant. For example, 65.43 e-21 is not a floating point constant, the translator recognizes this as four separate tokens:

```
65.43 e - 21
```

which will cause a syntax error. If you need a float constant, you can get it using the ending f:

```
3.14159265f 2.0f 2.997925f
```

2.4.3 Character constants

A character constant is a character enclosed in single quotes, such as 'a' or '0'. Character constants can be thought of as constants that give names to integer values of characters from the set accepted on the machine on which the program is running. This is not necessarily the same character set as on the machine where the program was broadcast. Thus, if you run the program on an ASCII machine, the value '0' is 48, and if the machine is using EBCDIC code, it will be 240. Using character constants instead of their decimal integer equivalent increases program portability. Some special combinations of characters that begin with a backslash have standard names:

End of line NL (LF) \ n
Horizontal tab HT \ t
Vertical tab VT \ v
Return BS \ b
Carriage return CR \ r
FF \ f format translation
BEL signal \ a
Backslash \ \\

Question mark ? \?
Single quote '\'
Double quote "\"
NUL character \ 0
Octal number ooo \ ooo
Hexadecimal number hhh \ xhhh

Despite their appearance, all these combinations define one symbol. The type of the character constant is char. You can also specify a character using an octal number, represented by one, two, or three octal digits (preceded by \), or by using a hexadecimal number (preceded by \ x). The number of hexadecimal digits in this sequence is unlimited. A sequence of octal or hexadecimal digits ends with the first non-digit character . Here are some examples:

'\ 6' '\ x6' 6 ASCII ack
'\ 60 ' '\ x30' 48 ASCII '0'
'\ 137' '\ x05f' 95 ASCII '_'

Any character from the machine character set can be represented this way . In particular, characters defined in this way can be included in character strings (see the next section). Note that if the numeric form of the task is used for symbols , then the portability of the program between machines with different character sets is violated .

2.4.4 Strings

A string is a sequence of characters enclosed in double quotes:

"this is a string"

Each line contains one more character than is explicitly specified: all lines are terminated with a null character ('\ 0') with the value 0. Therefore

sizeof ("asdf") == 5;

The string type is considered to be "an array of the corresponding number of characters", therefore the "asdf" type is char [5]. An empty string is written as "" and is of type char [1]. Note that strlen (s) == sizeof (s) -1 is executed for any string s , because the strlen () function ignores the trailing '\ 0' character.

Within a string, special combinations with \ can be used to represent invisible characters. In particular, you can specify the double quote "character" or the \ character in a string . Most often, of these characters, the end-of-line character '\ n' is needed, for example:

```
cout << "beep at end of message \ 007 \ n"
```

Here 7 is the ASCII value of the BEL (signal) character, which is portablely referred to as \ a. There is no way to set a "real" end-of-line character in a string:

```
"this is not a string,
and a syntax error "
```

For greater clarity of the program, long lines can be split with spaces, for example:

```
char alpha [] = "abcdefghijklmnopqrstuvwxyz"
            "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

Similar consecutive strings will be combined into one, so the alpha array can be equivalently initialized with a single string:

```
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXY
Z";
```

You can use the '\ 0' character in a string, but most programs do not expect any other characters after it. For example, the string "asdf \ 000hjkl" is treated as the string "asdf" by the standard functions strcpy () and strlen () .

If you specify a numeric constant in a string as a sequence of octal digits , then it is reasonable to specify all three digits. Writing this line is not too simple anyway, to even wonder whether the digit belongs to a number or is a separate symbol. Use two digits for hexadecimal constants. Consider the following examples:

```
char v1 [] = "a \ x0fah \ 0129"; // 'a' '\ xfa' 'h' '\ 12' '9'
char v2 [] = "a \ xfah \ 129"; // 'a' '\ xfa' 'h' '\ 12' '9'
char v3 [] = "a \ xfad \ 12 7"; // 'a' '\ xfad' '\ 127'
```

2.4.5 Zero

Zero (0) is of type int. Thanks to the standard conversions ($$ R.4), 0 can be used as a constant of an integer type, or a floating point type, or a pointer type. No object can be placed if 0 is specified instead of address. Which type of zero to use is determined by the context. Usually (but not necessarily) zero is represented by a sequence of all-zeros of suitable length.

2.5 Named constants

By adding the function word const to the description of the object, you can turn this object from a variable into a constant, for example:

```
const int model = 90;
const int v [] = {1, 2, 3, 4};
```

Since you cannot assign anything to a constant, it must be initialized. By describing an object as const, we guarantee that its value does not change in scope:

```
model = 200; // mistake
model ++; // mistake
```

Note that the const specification restricts the use of an object rather than specifying where to place the object. It might be quite reasonable and even useful to describe a function with a const return type :

```
const char * peek (int i) // return a pointer to a constant string
{
        return hidden [i];
}
```

The above function could be used to pass a write-protected string to another program where it will be read. Generally speaking, a translator can take advantage of the fact that an object is const for various purposes (of course, this depends on the "sanity" of the translator). The most obvious thing is that there is no need to allocate memory for the constant, since its value is known to the translator. Further, the initializer for a constant is usually (but not always) a constant expression that can be evaluated during translation. However, for an array of constants, you usually have to allocate memory, since in general the translator does not know which array element is used in the expression. But even in this case , optimization is possible on many machines by placing such an array in write-protected memory.

By defining a pointer, we are dealing with two objects: the pointer itself and the object being pointed to. If the pointer description contains a "prefix" const, then the object itself is declared as a constant, but not a pointer to it, for example:

```
const char * pc = "asdf"; // pointer to constant
pc [3] = 'a'; // mistake
pc = "ghjk"; // fine
```

To describe the pointer itself as a constant, and not the referenced object, you need to use the * operator before const. For example:

```
char * const cp = "asdf"; // constant pointer
cp [3] = 'a'; // fine
cp = "ghjk"; // mistake
```

To make both a pointer and an object constants, you must both declare const, for example:

```
const char * const cpc = "asdf"; // constant pointer to const
cpc [3] = 'a'; // mistake
cpc = "ghjk"; // mistake
```

An object can be declared constant when accessing it using a pointer, and at the same time being mutable when accessing it in another way. This is especially useful for function parameters . By describing the function pointer parameter as const, we prohibit changing the specified object in it, for example:

```
char * strcpy (char * p, const char * q); // cannot change * q
```

A pointer to a constant can be assigned the address of a variable, since it won't do any harm. However, the address of a constant cannot be assigned to a pointer without the const specification, otherwise it will be possible to change its value, for example:

```
in t a = 1;
const int c = 2;
const int * p1 = & c; // fine
const int * p2 = & a; // fine
int * p3 = & c; // mistake
* p3 = 7; // changes the value of c
```

2.5.1. Enumerations

There is a way to associate names with integer constants, which is often more convenient than describing them with const. For example:

```
enum {ASM, AUTO, BREAK};
```

There are three integer constants defined here, which are called enumeration items , and assigned values. Since by default the values of the enumeration elements start at 0 and go in ascending order, the given enumeration is equivalent to the definitions:

```
const ASM = 0;
const AUTO = 1;
const BREAK = 2;
```

An enumeration can have a name, for example:

```
enum keyword {ASM, AUTO, BREAK};
```

The enumeration name becomes the new type. Using standard conversions, the enumeration type can be implicitly cast to the int type. The reverse conversion (from type int to enumeration) must be specified explicitly. For example:

```
void f ()
{
        keyword k = ASM;
        int i = ASM;
        k = i // error
        k = keyword (i);
        i = k;
        k = 4; // mistake
}
```

The last conversion explains why there is no implicit conversion from int to enumeration: most int values have no representation in this enumeration. By describing a variable with type keyword instead of the obvious int, we have given both the user and the translator specific information about how this variable will be used. For example, for the following operator

```
keyword key;
switch (key) {
        case ASM:
                // do something
                break;
        case BREAK:
                // do something
                break;
}
```

the translator may issue a warning because only two of the three possible keyword values are used. The values of the enumeration elements can also be set explicitly. For example:

```
enum int16 {
        sign = 0100000,
        most_significant = 040000,
        least_significant = 1
};
```

The specified values do not have to be different, positive, or in ascending order.

2.6. Save memory

In the process of creating a non-trivial program, sooner or later a moment comes when more memory is required than can be allocated or requested. There are two ways to squeeze out some more memory:

[1] packing variables with small values into bytes;

[2] use the same memory to store different objects at different times.

The first method is implemented using fields, and the second is using unions. Both are described below. Since the purpose of these constructs is mainly related to program optimization, and since they are generally not portable, the programmer should think carefully before using them. It is often better to change the algorithm for working with data, for example, to use more dynamically allocated memory than pre-allocated static memory.

2.6.1 Fields

It seems wasteful to use char for a trait that only takes two values (eg: yes, no), but an object of type char is the smallest object in C ++ that can be independently allocated in memory. However, it is possible to collect variables with a small range of values together by defining them as structure fields. A member is a field if the number of digits it must occupy is specified in its definition after the name . Anonymous fields are allowed . They do not affect working with named fields, but they can improve the placement of fields in memory for a particular machine:

```
struct sreg {
        unsig ned enable: 1;
        unsigned page: 3;
        unsigned: 1; // not used
        unsigned mode: 2;
        unsigned: 4; // not used
        unsigned access: 1;
        unsigned length: 1;
        unsigned non_resident: 1;
};
```

The above structure describes the bits of the zero status register of the DEC PDP11 / 45 (it is assumed that the fields in the word are located from left to right). This example also shows another possible use of fields: to give names to those parts of an object whose placement is determined externally. The field must be of integer type ($$ R.3.6.1 and $$ R.9.6) and is used similarly to other integer objects. But there is an exception: you cannot take the field address. In the operating system kernel or debugger, the sreg type could be used as follows:

```
sreg * sr0 = (sreg *) 0777572;
// ...
if (sr0-> access) { // access violation
        // find a solution
        sr0-> access = 0;
}
```

However, by using fields to pack multiple variables into one byte, we don't necessarily save memory. Memory for data is saved, but on most machines the amount of instructions required to work with packed data increases at the same time . There are even known programs that were significantly reduced in size if the binary variables specified by the fields were converted to variables of the char type! Also, char or int access is usually much faster than field access. Fields are just a convenient short form for specifying logical operations for extracting or entering information in a part of a word.

2.6.2. Associations

Consider a table of names in which each element contains a name and its meaning. The value can be specified either as a string or as an integer:

```
struct entry {
        char * name;
        char type;
        char * string_value; // used if type == 's'
        int int_value; // used if type == 'i'
};
```

```
vo id print_entry (entry * p)
{
        switch (p-> type) {
                case 's':
                        cout << p-> string_value;
                        break;
                case 'i':
                        cout << p-> int_value;
                        break;
                default:
                        cerr << "type corrupted \ n";
                        break;
        }
}
```

Since the variables string_value and int_value can never be used at the same time, it is obvious that some memory is wasted. This can be easily fixed by declaring both variables as union members, like so:

```
struct entry {
        char * name;
        char type;
        union {
                char * string_value; // used if type == 's'
                int int_value; // used if type == 'i'
        };
};
```

It is now guaranteed that when memory is allocated for entry, the string_value and int_value members will be located from the same address, without having to change all the parts of the program that work with entry. This means that all the members of the union together occupy the same amount of memory as the largest member of the union.

The reliable way to work with union is to select the value using the same member that wrote it. However, in large programs it is difficult to ensure that a join is used only in this way, and using the wrong join member can lead to hard-to-find errors. However, you can embed a union in a structure that provides the correct association between the value of the type field and the current member type of the union ($$ 5.4.6).

Sometimes unions are used for "pseudo - conversions" of type (mostly programmers who are accustomed to languages that do not have type conversions, and as a result have to trick the translator). Here is an example of such a "conversion" of int to int * on a VAX machine, which is achieved by a simple match of the digits:

```
struct fudge {
            union {
                        int i;
                        int * p;
            };
};
f udge a;
ai = 4095;
int * p = ap; // incorrect use
```

In reality, this is not a type conversion at all; on some machines int and int * occupy different amounts of memory, while on others an integer cannot be located at an odd number. This use of unions is not portable, while there is a portable way of specifying an explicit type conversion ($$ 3.2.5).

Sometimes unions are used on purpose to avoid type conversion. For example, you can use fudge to find out how pointer 0 is represented:

```
fudge.p = 0;
int i = fudge.i; // i doesn't have to be 0
```

The union can be given a name, that is, you can make it a full-fledged type. For example, fudge can be described like this:

```
union fudge {
            int i;
            int * p;
} ;
```

and use (incorrectly) exactly the same as before. At the same time, named unions can be used in a completely correct and justified way (see $$ 5.4.6).

2.7 Exercises

1. (* 1) Run the "Hello, world" program (see $$ 1.3.1).
2. (* 1) For each description from $$ 2.1, do the following: if the description is not a definition, then write the corresponding

definition; if the description is a definition, write a description for it that would not be a definition at the same time.

3. (* 1) Write descriptions of the following objects: a pointer to a symbol; an array of 10 integers; references to an array of 10 integers; a pointer to an array of character strings; pointer to pointer to character; integer-constant; a pointer to an integer constant; a constant pointer to an integer. Provide descriptions with initialization.
4. (* 1.5) Write a program that prints the dimensions of the basic and pointer types. Use the sizeof operation.
5. (* 1.5) Write a program that prints the letters 'a' through 'z' and the numbers '0' through '9' and their integer values. Do the same for the other visible symbols. Do this using hexadecimal notation.
6. (* 1) Print the bit sequence of the 0 pointer representation on your machine. Hint: see $$ 2.6.2.
7. (* 1.5) Write a function that prints the order and mantissa of a double parameter .
8. (* 2) What are the largest and smallest values of the following types on the machine you are using : char, short, int, long, float, double, long double, unsigned, char *, int *, and void *? Are there any special restrictions on these values? For example, can int * be an odd integer? How are objects of these types aligned in memory? For example, can an integer have an odd address?
9. (* 1) What is the maximum length of a local name that can be used in your C ++ implementation? What is the maximum length of an external name? Are there any restrictions on the characters that can be used in the name?
10. (* 1) Write a function that swaps the values of two integers. Use int * as the parameter type. Write another function with the same purpose, using int & as the parameter type.
11. (* 1) What is the size of the str array in the following example: char str [] = "a short string"; How long is the string "a short string"?
12. (* 1.5) Make a table of the names of the months of the year and the number of days in each of them. Write a program that prints it. Do this twice: once using arrays for the month names and the

number of days, and the other time using an array of structures, each containing the name of the month and the number of days in it.

13. (* 1) Define types using typedef: unsigned char, constant unsigned char, pointer to integer, pointer to pointer to character, pointer to character array, array of 7 pointers to integer, pointer to array of 7 pointers to integer, and array of 8 arrays of 7 pointers to an integer.

14. (* 1) Define functions f (char), g (char &) and h (const char &) and call them using 'a', 49, 3300, c, uc, and sc as parameters , where c is char, uc - unsigned char and sc - signed char. Which challenge is legal? At what call will the translator have to set a temporary variable?

CHAPTER 3. EXPRESSIONS AND OPERATORS

"But on the other hand, we shouldn't forget about efficiency."
(John Bentley)

C ++ has a relatively small set of operators that allows you to create flexible control structures, and a rich set of operations for working with data. Their main capabilities are shown in this chapter with one complete example. It then summarizes the expressions and discusses in detail the type conversion operations and free memory allocation. The following is a summary of the operators, and at the end of the chapter, we discuss white space and comment usage.

3.1 Calculator

We will get acquainted with expressions and operators using the example of a calculator program . The calculator implements four basic arithmetic operations in the form of infix operations on floating point numbers. As an exercise, it is suggested to add variables to the calculator . Let's say the input stream is:

```
r = 2.5
area = pi * r * r
```

(pi has a predefined meaning here). Then the calculator program will give:

```
2.5
19.635
```

The result for the first input line is 2.5, and the result for the second line is 19.635.

The calculator program consists of four main parts: the analyzer, the input function, the name table, and the driver. In fact, it is a miniature translator in which the analyzer parses , the input function processes the input data and performs lexical analysis, the name table stores constant information needed for operation, and the driver performs initialization, outputting results, and handling errors. Many other useful features can be added to such a calculator , but its program is already quite large (200 lines), and the introduction of new features will only increase its volume, without providing additional information for learning C ++.

3.1.1 Analyzer

The grammar of the calculator language is determined by the following rules:

```
program:
    END // END is the end of the input
    expression-list END

    expression-list:
        expression PRINT // PRINT is '\ n' or ';'
        expression PRINT expression-list

    expression:
        expression + term
        expression - term
        term

    term:
        term / primary
        term * primary
        primary

    primary:
        NUMBER // floating point number in C ++
        NAME // name in C ++ language except '_'
        NAME = expression
        - primary
        ( expression )
```

In other words, a program is a sequence of lines, and each line contains one or more expressions, separated by semicolons. The main elements of an expression are numbers, names and operations *, /, +, - (unary and binary minus) and =. Names do not need to be described prior to use.

A technique commonly called recursive descent is used for parsing . This is a common and fairly obvious method. In languages such as C ++, that is, in which the call operation does not involve large overhead costs, this method is efficient.

Each grammar rule has its own function that calls other functions. Terminal characters (like END, NUMBER, + and -) are recognized by the get_token () lexical analyzer. Nonterminal characters are recognized by the parser functions expr (), term (), and prim (). As soon as both operands of an expression or subexpression are known, it is evaluated. In the real translator, at this moment, commands that evaluate the expression are created.

The analyzer uses the get_token () function for input. The value of the last call to get_token () is stored in the global variable curr_tok. The curr_tok variable accepts the values of the token_value enumeration elements:

```
enum token_value {
        NAME, NUMBER, END,
        PLUS = '+', MINUS = '-', MUL = '*', DIV = '/',
        PRINT = ';', ASSIGN = '=', LP = '(', RP = ')'
};
token_value curr_tok;
```

All parser functions assume that get_token () has already been called, and therefore curr_tok stores the next token to be parsed . This allows the parser to look one token ahead. Each parser function always reads one more token than is needed to recognize the rule for which it was called. Each analyzer function evaluates "its" expression and returns its result. The expr () function handles addition and subtraction. It consists of one cycle, in which the recognized terms are added or subtracted:

```
double expr () // add and subtract
{
        double left = term ();
        for (;;) // `` forever "
                switch (curr_tok) {
                        case PLUS:
                                get_token (); // case '+'
                                left + = term ();
                                break;
                        case MINUS:
                                get_token (); // case '-'
                                left - = term ();
                                break;
```

```
                    default:
                              return left;
          }
  }
```

By itself, this function does little. As is the case with high-level functions in large programs, it does the job by calling other functions. Note that expressions like 2-3 + 4 are calculated as (2-3) +4, which is predetermined by the grammar rules. The fancy for (;;) notation is the standard way to define an infinite loop, and can be denoted by the word "forever". This is a degenerate form of the for statement, and the while (1) statement is an alternative. The switch statement is executed repeatedly until the + or - operations no longer appear , at which point the return (default) statement is executed by default.

The + = and - = operations are used to perform addition and subtraction operations. You can write equivalent assignments: left = left + term () and left = left-term (). However, the options left + = term () and left- = term () are not only shorter, but more clearly define the action required. For a binary operation @, the expression x @ = y means x = x @ y, except that x is evaluated only once. This applies to binary operations:

```
  + - * /% & | ^ << >>
```

therefore, the following assignment operations are possible:

```
  + = - = * = / =% = & = | = ^ = << = >> =
```

Each operation is a separate token, so a + = 1 contains a syntax error (due to the space between + and =). The decoding of operations is as follows:% - taking the remainder, &, | and ^ - bit logical operations AND, OR and Exclusive OR; << and >> shift left and shift right. The term () and get_token () functions must be described before exp r () is defined . Chapter 4 discusses building a program as a collection of files. With one exception, all calculator programs can be designed so that they describe all objects only once and before using them . The exception is the expr () function, which calls the term () function , which , in turn, calls prim (), and that, finally,

calls expr (). This cycle needs to be broken somehow, for which the description given before prim () definition is quite suitable:

```
  double expr (); // this description is required
```

The term () function handles multiplication and division in the same way as expr () does addition and subtraction:

```
double term () // multiply and add
{
        double left = prim ();
        for (;;)
                switch (curr_tok) {
                        case MUL:
                                get_token (); // case '*'
                                left * = prim ();
                                break;
                        case DIV:
                                get_token (); // case '/'
                                double d = prim ();
                                if (d == 0) return error ("division by
0");
                                left / = d;
                                break;
                        default:
                                return left;
        }
}
```

A division by zero check is necessary because the result of division by zero is undefined and usually leads to disaster.

The error () function will be covered later. The variable d appears in the program where it is really needed, and is immediately initialized. In many languages, the description can only appear at the beginning of a block. But such a limitation can distort the natural structure of the program and contribute to the appearance of errors. More often than not, non-initialized local variables indicate bad programming style. The exceptions are those variables that are initialized with input operators, and variables of the array or structure type for which there is no traditional initialization with single assignments. It should be recalled that = is an assignment operation, while == is a comparison operation.

The prim function, which handles the primary, is a lot like the expr and term () functions. But since we got to the bottom of the call hierarchy, there is something to be done about it. No loop is needed for it:

```
double number_value;
char name_string [2 56];
double prim () // processes primary
{
        switch (curr_tok) {
                case NUMBER: // floating point constant
                        get_token ();
                        return number_value;
                case NAME:
                        if (get_token () == ASSIGN) {
                                name * n = insert (name_string) ;
                                get_token ();
                                n-> value = expr ();
                                return n-> value;
                        }
                        return look (name_string) -> value;
                case MINUS: // unary minus
                        get_token ();
                        return -prim ();
                case LP:
                        get_token ();
                        double e = expr ();
                        if (curr_tok! = RP) return error ("required)");
                        get_token ();
                        return e;
                case END:
                        return 1;
                default:
                        return error ("primary is required");
        }
}
```

When NUMBER appears (that is, a floating point constant), its value is returned. The get_token () input function puts the value of the constant into the global variable number_value. If the program uses global variables, then this often indicates that the structure is not fully worked out, and therefore some optimization is required . This is exactly the case in this case. Ideally, a token should consist of two parts: a value that determines the type of token (in this program, this is token_value), and (if necessary) the value of the token itself. There is only one simple variable curr_tok, so the global variable number_value is required to store the last read value of NUMBER . This solution works because the calculator in all calculations first selects one number, and then reads another from the input stream. As an exercise, it is proposed to get rid of this redundant global variable ($$ 3.5 [15]).

If the last NUMBER is stored in the global variable number_value, then the string representation of the last NAME is stored in name_string. Before doing anything with the name, the calculator must look ahead to see if it will be assigned a value or only use the existing value. In both cases, you must refer to the table of names. This table is discussed in $$ 3.1.3; but here it is enough just to know that it consists of records that look like:

```
struct name {
        char * string;
        name * next;
        double value;
};
```

The next member is used only by utility functions that work with the table:

```
name * look (const char *);
name * insert (con st char *);
```

Both functions return a pointer to the name record corresponding to their string parameter. The look () function "swears" if the name has not been entered into the table. This means that the calculator can use a name without a preliminary description, but the first time it may appear only on the left side of the assignment.

3.1.2 Input function

Getting input is often the most confusing part of the program. The reason lies in the fact that the program must interact with the user, that is, "put up" with his whims, take into account the accepted agreements and provide for seemingly rare errors. Attempts to force a person to behave in a more machine-friendly manner are usually considered unacceptable, which is fair. The input task for a low-level function is to sequentially read characters and compose them to a token, which is already used by higher-level functions. In this example , the get_token () function does the low-level input .

Fortunately, writing a low-level input function is a rare task. Good systems have standard functions for such operations.

The input rules for the calculator were specially chosen to be somewhat cumbersome for streaming input functions. Minor changes to token definitions would make get_token () a deceptively simple function.

The first tricky part is that the '\ n' end-of-line character is important to the calculator, but input streaming functions treat it as a generic space character. In other words, for these functions, '\ n' is only meaningful as a token-terminating character. Therefore, you have to analyze all generalized spaces (space, tabulation, etc.). This is done in the do statement, which is equivalent to the while statement, except that the body of the do statement is always executed at least once:

```
char ch;
do {// skip spaces except '\ n'
        if (! cin.get (ch)) return curr_tok = END;
} while (ch! = '\ n' && isspace (ch));
```

The cin.get (ch) function reads one character from the standard input stream into ch. The value of the condition if (! Cin.get (ch)) is false if no characters can be obtained from the cin stream . The END token is then returned to terminate the calculator. Operation ! (NOT) is needed because get () returns a nonzero value if read successfully.

The isspace () substitution function from <ctype.h> checks to see if its parameter is a generic space ($$ 10.3.1). It returns nonzero if it is, and zero otherwise. The check is implemented as a reference to the table, so for speed it is better to call isspace () than check yourself. The same can be said for the isalpha (), isdigit (), and isalnum () functions that are used in get_token ().

After skipping generic spaces, the next character read determines what the token that begins with it will be. Before giving the whole function, let's consider some cases separately. Expression-terminating tokens '\ n' and ';' are processed as follows:

```
switch (ch) {
        case ';':
        case '\ n':
                cin >> ws; // skip generic space
```

```
                    return curr_tok = PRINT;
```

It is not necessary to skip the space again, but by doing this we avoid repeated calls to get_token (). The ws variable, described in the <stream.h> file, is used only as a receiver for unnecessary spaces. An error in the input data, as well as the end of the input, will not be detected until the next call to get_token (). Notice how multiple selection labels mark the same sequence of statements defined for those choices. For both characters ('\ n' and ';'), the PRINT token is returned and placed in curr _tok.

The numbers are processed like this:

```
  case '0':
  case '1':
  case '2':
  case '3':
  case '4':
  case '5':
  case '6':
  case '7':
  case '8':
  case '9':
  case '.':
            cin.putback (ch);
            cin >> number_value;
            return curr_tok = NUMBER;
```

Placing variation labels horizontally rather than vertically is not the best way to do it, as it is harder to read; but writing a string for each digit is tedious. Since the >> operator can read a floating-point constant of type double, the program is trivial: first of all, the initial character (digit or dot) is returned back to cin, and then the constant can be read into number_value. Name, i.e. token NAME, defined as a letter, followed by several letters or numbers:

```
  if (isalpha (ch)) {
            char * p = name_string;
            * p ++ = ch;
            while (cin.get (ch) && isalnum (ch)) * p ++ = ch;
            cin.putback (ch);
            * p = 0;
            return curr_tok = NAME;
    }
```

This code fragment writes a null-terminated string to name_string . The isalpha () and isalnum () functions are defined in <ctype.h>. The result of isalnum (c) is nonzero if c is a letter or number, and zero otherwise.

Finally, here is the complete input function:

```
    token_value get_token ()
    {
            char ch;
            do {// skips generic spaces except '\ n'
                    if (! cin.get (ch)) return curr_tok = END;
            } while (ch! = '\ n' && isspace (ch));
            switch (ch) {
                    case ';':
                    case '\ n':
                            cin >> ws; // skip generic space
                            return curr_tok = PRINT;
                    case '*':
                    case '/':
                    case '+':
                    case '-':
                    case '(':
                    case ')':
                    case '=':
                            return curr_tok = t oken_value (ch);
                    case '0': case '1': case '2': case '3': case '4':
                    case '5': case '6': case '7': case '8': case '9':
                    case '.':
                            cin.putback (ch);
                            cin >> number_value;
                            return curr_tok = NUMBER;
                    default: // NAME, NAME = or error
                            if (isalpha (c h)) {
                                    char * p = name_string;
                                    * p ++ = ch;
```

```
                                while (cin.get (ch) && isalnum (ch)) *
p ++ = ch;
                                cin.putback (ch);
                                * p = 0;
                                return curr_tok = NAME;
                        }
                        error ("invalid token");
                        return curr_tok = PRINT;
        }
}
```

Converting an operation to a token value is trivial for it, because in the token_value enumeration, the operation token was defined as an integer (operation character code).

3.1.3 Name table

There is a search function in the name table:

```
name * look (char * p, int ins = 0);
```

Its second parameter shows whether the character string denoting the name was previously entered into the table. Initializer = 0 specifies the default parameter value, which is used if look () is called with only one parameter. This is convenient because you can write look ("sqrt2"), which means look ("sqrt2", 0), i.e. search, not entry into the table. To make it just as convenient to specify the operation of entering into the table, a second function is defined:

```
inline name * insert (const char * s) {return look (s, 1); }
```

As previously mentioned, the records in this table are of the following type:

```
struct name {
        char * string;
        name * next;
        double value;
};
```

The next member is used to link records in a table. The table itself is just an array of pointers to objects of type name:

```
const TBLSZ = 23;
name * table [TBLSZ];
```

Since by default all static objects are initialized to zero, this trivial description of the table table provides the required initialization as well.

The look () function uses a simple hash code to look up a name in a table (records where names have the same hash code are linked together):

```
int ii = 0; // hash code
const char * pp = p;
while (* pp) ii = ii << 1 ^ * pp ++;
if (ii <0) ii = -ii;
ii% = TBLSZ;
```

In other words, using the operator ^ ("exclusive OR") all characters of the input string p are added to ii in turn. The bit in the result x ^ y is 1 if and only if these bits in the operands x and y are different. Before performing the ^ operation, the value of ii is shifted one bit to the left so that more than one byte ii is used. These actions can be written as follows:

```
ii << = 1;
         ii ^ = * pp ++;
```

For a good hash code, it is better to use the ^ operator than +. The shift operation is important to get an acceptable hash code in both cases. Operators

```
if (ii <0) ii = -ii;
ii% = TBLSZ;
```

guarantee that the value of ii is in the range 0 ... TBLSZ-1. Recall that% is the operation of taking the remainder. The look function is shown below in full :

```
#include <string.h>
name * look (const char * p, int ins = 0)
{
            int ii = 0; // hash code
            const char * pp = p;
            while (* pp) ii = ii << 1 ^ * pp ++;
            if (ii <0) ii = -ii;
            ii% = TBLSZ;
            for (name * n = table [ii]; n; n = n-> next) // search
                        if (strcmp (p, n-> string) == 0) return n;
            if (ins == 0) error ("name not found");
            name * nn = new name; // entering
            nn-> string = new char [strlen ( p) +1];
            strcpy (nn-> string, p);
            nn-> value = 1;
            nn-> next = table [ii];
            table [ii] = nn;
            return nn;
   }
```

After calculating the hash code ii, a simple search for the name is performed by the next members . The names are compared using the standard string comparison function strcmp (). If the name is found, a pointer to the containing record is returned , otherwise a new record with this name is created.

Adding a new name means creating a new object name in free memory using the new operation (see $$ 3.2.6), initializing it and including it into the list of names. The latter is performed as entering a new name at the beginning of the list, since this can be done even without checking whether the list exists at all. The character string of the name is also allocated in free memory. The strlen () function specifies how much memory is needed for the string, the new operation allocates the needed memory, and the strcpy () function copies the string into it. All string functions are described in <string.h>:

```
   extern int strlen (const char *);
   extern int strcmp (const char *, const char *);
   extern char * strcp y (char *, const char *);
```

3.1.4 Error handling

Since the program is simple enough, you don't have to worry too much about error handling. The error function simply counts the number of errors, issues a message about them, and returns control back:

```
   int no_of_errors;
   double error (const char * s)
   {
            cerr << "error:" << s << "\ n";
            no_of_errors ++;
            return 1;
```

```
}
```

The unbuffered output of cerr is typically used for error reporting. Control returns from error () because errors tend to occur in the middle of evaluating an expression. This means that you need to either completely stop the calculation, or return a value that should not cause subsequent errors. For a simple calculator, the latter is more appropriate . If get_token () kept track of line numbers, then error () could tell the user the approximate location of the error. This would be useful for non-interactive work with the calculator. Often, after the appearance of an error, the program must terminate because it was not possible to offer a reasonable option for its further execution. It can be terminated by calling the exit () function, which ends work with output streams ($$ 10.5.1) and terminates the program, returning its parameter as its result. A more radical way to terminate a program is to call the abort () function, which interrupts program execution immediately or immediately after saving information for the debugger (resetting RAM). You can find details in your reference guide.

More subtle error handling techniques can be suggested for special situations (see $$ 9), but the proposed solution is perfectly acceptable for a toy calculator of 200 lines.

3.1.5 Driver

When all parts of the program are defined, only the driver is needed to initialize and start the process. In our example, the main () function will handle this :

```
int main ()
{
        // insert predefined names:
        insert ("pi") -> value = 3.1415926535897932385;
        insert ("e") -> value = 2.7182818284590452354;
        while (cin) {
                get_token ();
                if (curr_tok == END) break;
                if (curr_tok == PRINT) continue;
                cout << expr () << '\ n';
        }
        return no_of_errors;
}
```

It is assumed that the main () function returns zero if the program exits normally, and nonzero if otherwise. A nonzero value is returned as the number of errors. It turns out that all initialization is reduced to entering predefined names into the table.

In the main loop, expressions are read and the results returned. One line does this :

```
cout << expr () << '\ n';
```

Checking cin on each pass of the loop ensures that the program terminates even if something happens to the input stream, and checking for the END token is needed to complete the loop normally when get_token () detects the end of the file. The break statement is used to exit from the nearest enclosing switch or loop statement (that is, a for, while, or do statement). Checking for the PRINT token (i.e., '\ n' and ';') removes the obligation for expr () to process empty expressions. The continue statement is equivalent to jumping to the end of the loop, so in our case the fragment:

```
while (cin) {
        // ...
        if (curr_tok == PRINT) continue;
        cout << expr () << "\ n";
}
```

equivalent to snippet:

```
while (cin) {
        // ...
        if (curr_tok == PRINT) goto end_of_loop;
        cout << expr () << "\ n";
        end_of_loop:;
}
```

Loops are described in more detail in $$ R. 6

3.1.6 Command line parameters

When the calculator program was already written and debugged, it turned out that it was inconvenient to start it first, enter an expression, and then exit the calculator. Moreover, usually you just need to evaluate one expression. If you set this expression as a command line parameter for starting the calculator, you can save several keystrokes.

As already mentioned, program execution begins by calling main (). In this call, main () receives two parameters: the number of parameters (usually called argc) and an array of parameter strings (usually called argv). The parameters are character strings, so argv is of type char * [argc + 1]. The name of the program (as it was given on the command line) is passed to argv [0], so argc is always at least one. For example, for the command line

```
dc 150 / 1.1934
```

parameters matter:

```
argc 2
argv [0] "dc"
argv [1] "150 / 1.1934"
argv [2] 0
```

Getting to the command line parameters is easy; the problem is how to use them so as not to change the program itself. In this case, it turns out to be quite simple, since the input stream can be set to a character string instead of a file ($$ 10.5.2). For example, you can define cin so that characters are read from a string rather than from the standard input stream:

```
int main (int argc, char * argv [])
{
		switch (argc) {
				case 1: // read from standard input stream
						break;
				case 2: // read from parameter string
						cin = * new istream (argv [1], strlen (argv
[1]));
						break;
				default:
						error ("too many parameters");
						return 1;
		}
		// then the previous version of main
}
```

In this case, istrstream is an istream function that reads characters from the string that is its first parameter. To use istrstream, you need to include the <strstream.h> file in your program, not the regular <iostream.h>. The rest of the program remained unchanged, except for adding parameters to the main () function and using them in the switch statement. You can easily modify the main () function so that it can take multiple parameters from the command line. However, this is not very necessary, especially since you can pass several expressions as one parameter:

```
dc "rate = 1.1934; 150 / rate; 19.75 / rate; 217 / rate"
```

The quotes are required because the ';' serves as a UNIX command separator. Other systems may have different command line parameter conventions .

3.2 Summary of operations

A complete and detailed description of the C ++ language operations is given in $$ R.7. We advise you to read this section. It also provides a brief summary of the operations and a few examples. Each operation is accompanied by one or more specific names and an example of its use. In these examples, class_name is a class name, member is the name of a member, object is an expression that specifies a class object, pointer is an expression that specifies a pointer, expr is just an expression, and lvalue (address) is an expression that specifies a non-constant object. The designation (type) defines the name of the type in general form (with the possible addition of *, (), etc.). If it is specified without parentheses, there are restrictions.

The order of application of unary and assignment operations is "right to left", and all other operations - "from left to right". That is, a = b = c means a = (b = c), a + b + c means (a + b) + c, and * p ++ means * (p ++), not (* p) ++.

C ++ operations

: :	Scope resolution	class_name :: member
::	Global	:: name
...	Member selection	object. member
->	Member selection	pointer -> member
[]	Indexing	pointer [expr]
()	Function call	expr (expr_list)
()	Structural value	type (expr_list)
sizeof	Object size	sizeof expr

sizeof	Type size	sizeof (type)
++	Postfix increment	lvalue ++
++	Prefix increment	++ lvalue
-	Postfix decrement	lvalue -
-	Prefix decrement	- lvalue
~	Addition	~ expr
!	Logical NOT	! ex pr
-	Unary minus	- expr
+	Unary plus	+ expr
&	Taking an address	& lvalue
*	Indirection	* expr
new	Creation (placement)	new type
delete	Destruction (release)	delete pointer
delete []	Destroying an array	delete [] pointer
()	Casting (conversion) of the type	(type) expr
... *	Member selection indirect	object. pointer-to-member
-> *	Member selection indirect	pointer -> pointer-to-member
*	Multiplication	expr * expr
/	Division	expr / expr
%	Remainder of the division	expr% expr
+	Addition (plus)	expr + expr
-	Subtraction (minus)	expr - expr
<<	Add (plus) Shift left	expr << expr
>>	Shift right	expr >> expr
<	Smaller	expr <expr
<=	Less or equal	expr <= expr
>	More	expr> expr
> =	More or equal	expr> = expr
==	Equally	expr == expr
! =	Not equal	expr! = expr
&	Bitwise AND	expr & expr
^	Bitwise exclusive OR	expr ^ expr

|	Bitwise inclusive OR	expr | expr
&&	Logical AND	expr && expr
||	Logical OR	expr || expr
? :	Condition operation	expr? expr: expr
=	Simple assignment	lvalue = ex pr
* =	Assignment with multiplication	lvalue * = expr
/ =	Division assignment	lvalue / = expr
% =	Assignment with taking the remainder of division	lvalue% = expr
+ =	Addition assignment	lvalue + = expr
- =	Assignment Subtraction	lvalue - = expr
<< =	Left shift assignment	lvalue << = expr
>> =	Right Shift Assignment	lvalue >> = expr
& =	Bitwise AND assignment	lvalue & = expr
| =	Bitwise inclusive OR assignment	lvalue | = expr
^ =	Bitwise exclusive OR assignment	lvalue ^ = e xpr
,	Comma (sequence)	expr, expr

All table operations that are between two horizontal lines closest to each other have the same priority. The priority of operations decreases when moving from top to bottom. For example, a + b * c means a + (b * c) because * takes precedence over +; and the expression a + bc means (a + b) -c, because + and - have the same precedence, and the + and - operations are applied from left to right.

3.2.1 Brackets

C ++ syntax is overloaded with parentheses, and the variety of their uses can be confusing. They highlight the actual parameters when calling functions, the names of the types defining the functions, and also serve to resolve conflicts between operations with the same priority. Fortunately, the latter is not very common, since the precedence and order of application of operations are defined so that expressions are evaluated "naturally" (that is, the most common way). For example, the expression

```
if (i <= 0 || max <i) // ...
```

means the following: "If i is less than or equal to zero, or if max is less than i". That is, it is equivalent to

```
if ((i <= 0) || (max <i)) // ...
```

but not equivalent to a valid albeit meaningless expression

```
if (i <= (0 || max) <i) // ...
```

However, if the programmer is unsure of the above rules, parentheses should be used, with some preferring to write longer and less elegant expressions for reliability, such as:

```
if ((i <= 0) || (max <i)) // ...
```

When complicating subexpressions, parentheses are used more often. Don't forget , however, that complex expressions are a source of errors. Therefore, if you get the feeling that this expression needs parentheses, it is better to break it up into parts and introduce an additional variable. There are times when the priorities of the operations do not lead to a "natural" order of computation. For example, in the expression

```
if (i & mask == 0) // trap! & is applied after ==
```

not masking i (i & mask) and then checking the result against 0. Since y == has precedence over &, this expression is equivalent to i & (mask == 0). In this case, parentheses play an important role:

```
if ((i & mask) == 0) // ...
```

It makes sense to give one more expression, which is calculated in a completely different way than an inexperienced user might expect:

```
if (0 <= a <= 99) // ...
```

It is valid, but it is interpreted as (0 <= a) <= 99, and the result of the first comparison is either 0 or 1, but not the value of a (unless, of course, a is 1). You can check if a is in the 0 ... 99 range like this:

```
if (0 <= a && a <= 99) // ...
```

It is a common mistake for beginners to use = (assign) in a condition instead of == (equal):

```
if (a = 7) // error: constant assignment in the condition
// ...
```

It is quite understandable, since in most languages "=" means "equal". It will not be difficult for a translator to report errors of this kind.

3.2.2 Calculation order

The order in which subexpressions in an expression are evaluated is not always defined. For example:

```
int i = 1;
v [i] = i ++;
```

Here the expression can be evaluated either as v [1] = 1, or as v [2] = 1. If there are no restrictions on the order in which subexpressions are evaluated, then the translator can create more optimal code. The translator should have warned about ambiguous expressions, but unfortunately most of them don't. For operations

```
&& || ,
```

it is guaranteed that their left operand is evaluated before the right operand. For example, in the expression b = (a = 2, a + 1) b will be assigned the value 3. Example of the operation || was given in $$ 3.2.1, and an example of the && operation is in $$ 3.3.1. Note that the comma operation differs in meaning from the comma that is used to separate parameters when calling functions. Let there be expressions:

```
f1 (v [i], i ++); // two parameters
f2 ((v [i], i ++)) // one parameter
```

The function f1 is called with two parameters: v [i] and i ++, but the order of evaluation of the parameter expressions is undefined. The dependence of the computation of the values of the actual parameters on the order of computation is far from the best programming style. In addition, the program becomes non-portable. F2 is called with one parameter, which is an expression containing the comma operation: (v [i], i ++). It is equivalent to i ++.

The parentheses can force the order of the calculation. For example, a * (b / c) can be evaluated as (a * b) / c (if only the user sees some difference in this). Note that for floating point values, the results of evaluating the expressions a * (b / c) and (a * b) / can differ quite significantly.

3.2.3 Increment and decrement

The ++ operator explicitly sets the increment, as opposed to implicitly setting it using addition and assignment. By definition ++ lvalue means lvalue + = 1, which in turn means lvalue = lvalue + 1, provided that the contents of the lvalue do not cause side effects. The increment operand expression is evaluated only once. The decrement operation (-) is denoted similarly . ++ and - operations can be used as prefix and postfix operations. The value of ++ x is the new (i.e., increased by 1) value of x. For example, y = ++ x is equivalent to y = (x + = 1). In contrast, the value of x ++ is equal to the previous value of x. For example, y = x ++ is equivalent to y = (t = x, x + = 1, t), where t is a variable of the same type as x.

Recall that the operations of incrementing and decrementing a pointer are equivalent to adding 1 to a pointer or subtracting 1 from a pointer, and the calculation takes place in the elements of the array to which the pointer is set . So, the result of p ++ is a pointer to the next element . For a pointer p of type T *, the following is true by definition:

```
long (p + 1) == long (p) + sizeof (T);
```

The most common use of increment and decrement operations is to modify variables in a loop. For example, copying a null-terminated string is specified as follows:

```
inline void cpy (char * p, const char * q)
{
        while (* p ++ = * q ++);
}
```

The C ++ language (like C) has both supporters and opponents precisely because of such a concise, complex expression programming style . Operator

```
wh ile (* p ++ = * q ++);
```

most likely, it will seem incomprehensible to those unfamiliar with C. It makes sense to take a closer look at such constructs, since they are not uncommon for C and C ++.

Let's first look at a more traditional way of copying a character array :

```
int length = strlen (q)
for (int i = 0; i <= length; i ++) p [i] = q [i];
```

This is an inefficient solution: the string is null terminated; the only way to find its length is to read it all down to a null character; as a result, the string is read both to set its length and to copy, that is, twice. So let's try this option:

```
for (int i = 0; q [i]! = 0; i ++) p [i] = q [i];
p [i] = 0; // write a null character
```

Since p and q are pointers, you can do without the variable i used for indexing :

```
while (* q! = 0) {
        * p = * q;
        p ++; // pointer to next character

        q ++; // pointer to next character

}
* p = 0; // write a null character
```

Since the postfix increment operation allows you to first use a value and then increase it, you can rewrite the loop like this:

```
while (* q! = 0) {
        * p ++ = * q ++;
}
* p = 0; // write a null character
```

Note that the result of the expression * p ++ = * q ++ is * q. Therefore, we can rewrite our example like this:

```
while ((* p ++ = * q ++)! = 0) {}
```

This option takes into account that * q is zero only when * q has already been copied to * p, so you can eliminate the terminating null character assignment. Finally, we can shorten the writing of this example even more if we take into account that an empty block is not needed, and the operation "! = 0" is redundant, since the result of the conditional expression is always compared to zero. As a result, we come to the original version, which caused confusion:

```
while (* p ++ = * q ++);
```

Is this option more difficult to understand than the ones above? Only inexperienced C ++ or C programmers! Will the latter option be the most time and memory efficient? Apart from the first option with the strlen () function, this is not obvious. Which of the options will be more effective is determined by both the specifics of the command system and the capabilities of the translator. The most efficient copying algorithm for your machine can be found in the standard function for copying strings from file <string.h>:

```
int strcpy (char *, const char *);
```

3.2.4 Bitwise logical operations

Bitwise logical operations

```
& | ^ ~ >> <<
```

apply to integers, that is, to objects of type char, short, int, long and their unsigned counterparts. The result of the operation will also be whole.

Most often, bitwise logical operations are used to work with a small amount of data (an array of bits). In this case, each bit of an unsigned integer represents one element of the set, and the number of elements is determined by the number of digits. The binary operation & is interpreted as intersection of sets, the operation | as a union, and the operation ^ as a difference of sets. Using an enumeration, you can name the elements of a set. Below is an example borrowed from <iostream.h>:

```
class ios {
        public:
                enum io_state {
                        goodbit = 0, eof bit = 1, failbit = 2, badbit = 4
                };
                // ...
};
```

The state of a stream can be set by the following assignment:

```
cout.state = ios :: goodbit;
```

Clarification with the ios name is necessary because the definition of io_state is in the ios class, and also to avoid collisions if the user enters their names like goodbit.

The check for the correctness of the flow and the successful completion of the operation can be set as follows:

```
if (cout.state & (ios :: badbit | ios :: failbit)) // error in the stream
```

Additional parentheses are needed because the & operation has a higher precedence than the "|" operation.

A function that detects the end of an input stream can report it like this:

```
cin.state | = ios :: eofbit;
```

The | = operation is used because the stream could already have an error (i.e. state == ios :: badbit), and the assignment

```
cin.st ate = ios :: eofbit;
```

could erase its mark. You can establish differences in the state of two threads in the following way:

```
ios :: io_state diff = cin.state ^ cout.state;
```

For types such as io_state, finding differences is not a very useful operation, but for other similar types it can be quite useful. For example, it is useful to compare two bit arrays, one representing the set of all possible interrupts being processed, and the other representing the set of interrupts waiting to be processed.

Note that using fields ($$ R.9.6) can serve as a convenient and more concise way of working with word parts than shifts and masking. You can also work with parts of a word using bitwise logical operations. For example, you can isolate the middle 16 bits from the middle of a 32-bit integer:

```
unsigned short middle (int a) {return (a >> 8) & 0xffff; }
```

Just do not confuse bitwise logical operations with simple logical operations:

```
&& || !
```

The latter can result in 0 or 1, and they are mainly used in conditional expressions of if, while, or for ($$ 3.3.1) statements . For example,! 0 (non-zero) has the value 1, while ~ 0 (zero's complement) is the all-ones bit set, which is usually -1's two's complement value.

3.2.5 Type conversion

Sometimes it is necessary to explicitly convert the value of one type to the value of another. An explicit conversion will result in a value of the specified type obtained from a value of a different type. For example:

```
float r = float (1);
```

Here, the integer value 1 is converted to floating point value 1.0f before being assigned . The result of the type conversion is not an address, so it cannot be assigned (unless the type is a reference).

There are two types of notation for explicit type conversion: the traditional notation as a cast operation in C, such as (double) a, and functional notation, such as double (a). Functional notation cannot be used for types that do not have a simple name. For example, to convert a value to a pointer type, you either need to cast

```
char * p = (char *) 0777;
```

or define a new type name:

```
typedef char * Pchar;
char * p = Pchar (0777);
```

According to the author, the functional notation is preferable in non-trivial cases . Consider two equivalent examples:

```
Pname n2 = Pbase (n1-> tp) -> b_name; // functional record
Pname n3 = ((Pbase) n2-> tp) -> b_name; // record with casting
```

Since the -> operation takes precedence over the cast operation, the last expression is executed like this:

```
((Pbase) (n2-> tp)) -> b_name
```

Using an explicit conversion to the pointer type, you can pass this object as an object of an arbitrary type. For example, the assignment

```
any_type * p = (any_type *) & some_object;
```

will allow accessing some object (some_object) through the p pointer as an object of arbitrary type (any_type). However, if some_object is not actually of type any_type, strange and unwanted results may result.

If type conversion is not necessary, it should be avoided altogether . Programs that have these transformations are usually more difficult to understand than programs that do not. At the same time, programs with explicit type conversions are clearer than programs that do without such conversions because they do not introduce types to represent higher-level concepts. This is, for example, programs that manipulate a device register by shifting and masking integers, instead of defining a suitable struct and working with it directly (see $$ 2.6.1). The correctness of an explicit type conversion often significantly depends on how the programmer understands how the language works with objects of various types, and what is the specificity of the given language implementation . Let's give an example:

```
   int i = 1;
   char * pc = "asdf";
   int * pi = & i;
   i = (int) pc;
   pc = (char *) i; // beware: the pc value might change.
                                                  // On some machines
   sizeof (int)
            // less than sizeof (char *)
   pi = (int *) pc;
   pc = (char *) pi; // be careful: pc might change
            // On some machines, char * has something different
            // representation as int *
```

For many machines, these assignments are harmless, but for some the result can be disastrous. At best, such a program would be portable. Usually, it is safe to assume that pointers to different structures have the same representation. Further, an arbitrary pointer can be assigned (without explicit type conversion) to a void * pointer, and void * can be explicitly converted back to a pointer of an arbitrary type.

In C ++, explicit type conversions are redundant in many cases where C (and other languages) require them. In many programs, explicit type conversions can be dispensed with altogether, and in many others they can be localized in multiple subroutines .

3.2.6 Free memory

A named object is either static or automatic (see $$ 2.1.3). A static object is allocated in memory when the program is started and exists there until it finishes. An automatic object is allocated in memory whenever control enters a block containing an object definition, and exists only as long as control remains in that block. However, it is often convenient to create a new object that exists until it is no longer needed. In particular, it can be convenient to create an object that can be used after returning from the function where it was created. The new operation creates such objects, and the delete operation is used to destroy them later. Objects created by the operation n ew are said to be allocated in free memory. Examples of such objects are tree nodes or list items that are included in data structures whose size is unknown at the translation stage. Let's take as an example a sketch of a translator, which is built similarly to a calculator program. The parsing functions create a tree from the expression representations that will be used later to generate code. For example:

```
struct enode {
            token_value oper;
            enode * left;
            enode * right;
};
enode * expr ()
{
            enode * left = term ();
            for (;;)
                        switch (curr_tok) {
                                    case PLUS:
                                    case MINUS:
                                                get_token ();
                                                enode * n = new enode;
                                                n-> oper = curr_tok;
                                                n-> left = left;
                                                n-> right = term ();
                                                left = n;
                                                break;
                                    default:
                                                return left;
                        }
}
```

A code generator can use an expression tree, like this:

```
void generate (enode * n)
{
            switch (n-> oper) {
                        case PLUS:
                                    // appropriate generation
                                    delete n;
            }
}
```

An object created by the new operation exists until it is explicitly destroyed by the delete operation. After that, the memory it occupied can be used again by new. There is usually no garbage collector looking for objects that no one else is referring to and giving the memory they occupy to the new operation for reuse. The delete operand can only be a pointer that the new operation returns, or zero. Applying delete to zero has no effect.

The new operation can also create arrays of objects, for example:

```
char * save_string (con st char * p)
{
        char * s = new char [strlen (p) +1];
        strcpy (s, p);
        return s;
}
```

Note that in order to reallocate memory allocated by the new operation, the delete operation must be able to determine the size of the allocated object. For example:

```
int main (int argc, char * argv [])
{
        if (argc <2) exit (1);
        char * p = save_string (arg [1]);
        delete [] p;
}
```

To achieve this, you have to allocate a little more memory for the object allocated by the standard new operation than for the static one (usually, more by one word). The simple delete operator destroys individual objects, and the delete [] operation is used to destroy arrays.

Operations with free memory are implemented by functions ($$ R.5.3.3-4):

```
void * operator new (size_t);
void operator delete (void *);
```

Where size_t is the unsigned integer type as defined in <stddef.h>.

The standard implementation of operator new () does not initialize the provided memory.

What happens when new can no longer find free memory to allocate? Since even virtual memory is not infinite, this happens from time to time. So, a request of the form:

```
char * p = new char [100000000];
```

usually does not go well. When the new operation cannot complete the request, it calls the function that was specified as a parameter to the set_new_handler () function from <new.h>. For example, in the following program:

```
#include <iostream.h>
#include <new.h>
#include <stdlib.h>

void out_of_store ()
{
        cerr << "operator new failed: out of store \ n";
        exit (1);
}
int main ()
{
        set_new_handler (& out_of_store);
        char * p = ne w char [100000000];
        cout << "done, p =" << long (p) << '\ n';
}
```

most likely, not "done" will be printed, but the message:

```
operator new failed: out of store
// new operation failed: no memory
```

You can do something more complex with the new_handler function than just terminate the program. If the algorithm for new and delete operations is known (for example, because the user has defined his operator new and operator delete functions), then the new_handler may try to find free memory for new. In other words, the user can write his own "garbage collector", thus making the call to delete operation optional. However, such a task is certainly beyond the power of a beginner.

Traditionally, the new operation simply returns a 0 pointer if it can not find enough free memory. The reaction to this new_handler has not been set. For example, the following program:

```
#include <stream.h>
main ()
{
        char * p = new char [100000000];
```

```
            cout << "done, p =" << long (p) << '\ n';
   }
```
will issue

```
   done, p = 0
```

No memory allocated and you have been warned! Note that by setting a reaction to such a situation in the new_handler function, the user takes it upon himself to check whether the free memory is exhausted. It must be executed every time the program calls new (unless the user has defined his own functions for placing objects of user-defined types; see $$ R.5.5.6).

3.3 Statement of operators

For a complete and consistent description of C ++ operators, see $$ R.6. We advise you to read this section. It also provides a summary of the operators and some examples.

Operator syntax

```
   operator:
            description
            {opt-list}
            expression opt;

            if (expression) statement
            if (expression) statement else statement
            switch (expression) statement

            while (expression) statement
            do statement while (expression)
            for (start-for-statement opt   expression; opt expression)
   statement

            case constant expression: statement
            default: operator
            break;
            continue;

            return expression opt;
```

```
goto identifier;
identifier: operator

operator-list:
        operator
        operator-list operator

start-for-statement:
        description
        expression opt;
```

Note that the description is an operator, but there are no assignment or function calls (these are expressions).

3.3.1 Select Operators

The value can be checked using if or switch statements:

```
if (expression) statement
if (expression) statement else statement
switch (expression) statement
```

In the C ++ language, there is no separate boolean (type with values true, false) among the basic types . All relationship operations:

```
==! = <> <=> =
```

results in an integer 1 if the relation holds, and 0 otherwise . Usually the constants TRUE as 1 and FALSE as 0 are defined.

In an if statement, if the expression is nonzero, the first statement is executed, otherwise the second statement (if specified) is executed. Thus, any integer or pointer expression is allowed as a condition. Let a be an integer, then

```
if (a) // ...
```

equivalent to

```
if (a! = 0) ...
```

Logical operations

```
&& || !
```

commonly used in conditions. In operations && and || the second operand is not evaluated if the result is determined by the value of the first operand. For example, in the expression

```
    if (p && l <p-> count) // ...
```
first the value of p is checked, and only if it is not equal to zero, then the relation l <p-> count is checked. Some simple if statements are conveniently replaced with conditional expressions. For example, instead of the operator
```
    if (a <= b)
            max = b;
    else
            max = a;
```
better to use expression
```
    max = (a <= b)? b: a;
```
The condition in the condition expression does not have to be surrounded by parentheses, but if you use them, the expression becomes clearer.

A simple switch can be written using a series of if statements. For example,
```
    switch (val) {
            case 1:
                    f ();
                    break;
            case 2:
                    g ();
                    break;
            default:
                    h ();
                    break;
    }
```
can be equivalently set like this:
```
    if (val == 1)
            f ();
    else if (val == 2)
            g ();
    else
            h ();
```
The meaning of both constructions is the same, but nevertheless the first is preferable, since it more clearly shows the essence of the operation: checking for the coincidence of the value of val with a value from a set of

constants. Therefore, in non-trivial cases, the notation using the switch is clearer.

You need to take care of some kind of completion of the statement specified in the switch option, unless you want statements from the next option to be executed. For example, switch

```
switch (val) {// possible error
        case 1:
                cout << "case 1 \ n";
        case 2:
                cout << "case 2 \ n";
        default:
                cout << "default: case not found \ n";
}
```

for val == 1, it prints, much to the surprise of the uninitiated:

```
c ase 1
case 2
        default: case not found
```

It makes sense to note in the comments those rare cases when the standard transition to the next version is left on purpose. Then this transition in all other cases can be safely considered a mistake. To terminate a statement in a variant, break is most often used, but sometimes return and even goto are used. Let's give an example:

```
switch (val) {// possible error
        case 0:
                cout << "case 0 \ n";
        case1:
  case 1:
                cout << "case 1 \ n";
                return;
        case 2:
                cout << "case 2 \ n";
                goto case1;
        default:
                cout << "default: case not found \ n";
                return;
}
```

Here, with val equal to 2, we get:

```
case 2
case 1
```

Note that a variant label cannot be used in a goto statement:

```
goto case 2; // syntax error
```

3.3.2 The g oto operator

The despised goto operator still exists in C ++:

```
goto identifier;
identifier: operator
```

Generally speaking, it is little used in high-level languages, but it can be very useful if the text in C ++ is not created by a person, but automatically, i.e. using the program. For example, goto statements are used to create an analyzer for a given grammar of a language using software tools. In addition, goto statements can be useful when the speed of the program is at the forefront. One of them is when some calculations take place in the inner loop of the program in real time.

There are few situations in ordinary programs where goto is warranted. One is to exit a nested loop or switch. The fact is that the break statement in nested loops or switches only allows you to go one level higher. Let's give an example:

```
void f ()
{
            int i;
            int j;
            for (i = 0; i <n; i ++)
                        for (j = 0; j <m; j ++)
                                    if (nm [i] [j] == a) goto found;
            // a not found here
            // .. .
            found:
            // nm [i] [j] == a
}
```

There is also a continue statement that allows you to go to the end of the loop. What this means is explained in $$ 3.1.5.

3.4 Comments and text arrangement

A program is much easier to read, and it becomes much clearer if you use comments wisely and systematically separate the program text with spaces. There are several ways to arrange program text, but there is no reason to think that one is the best. Although everyone has their own taste. The same can be said for comments.

However, you can fill the program with comments that make it harder to read and understand. The translator cannot understand the comment, so it cannot verify that the comment:

[1] meaningful,

[2] really describes the program,

[3] is not deprecated.

Many programs come across incomprehensible, ambiguous and simply incorrect comments. It is better to do without them altogether than to give such comments.

If a fact can be directly expressed in language, then this should be done, and one should not assume that it is enough to mention it in the commentary. The last note refers to comments like the one below:

```
// variable "v" needs to be initialized.
// variable "v" can only be used in function "f ()".
// before calling any function from this file
// you need to call the "init ()" function.
// call the "cleanup ()" function at the end of your program.
// don't use the "weird ()" function.
// function "f ()" has two parameters.
```

When properly programmed in C ++, such comments are usually redundant. To make these comments unnecessary, you can use the rules of binding ($$ 4.2) and scopes, as well as the rules of initialization and destruction of class objects ($$ 5.5).

If a statement is expressed by the program itself, you don't need to repeat it in the comment. For example:

```
a = b + c; // a takes the value b + c
count ++; // increment the count
```

Such comments are worse than redundant ones. They inflate the volume of text, cloud the program, and may even be false. At the same time, comments of this kind are used for examples in programming language textbooks like this book. This is one of the many reasons the curriculum is different from the real one.

We can recommend this style of introducing comments into the program:

1] start with a comment for each program file: outline what is defined in it, provide links to reference manuals, general ideas for maintaining the program, etc .;

2] annotate each definition of a class or type template ;

3] comment on each non-trivial function, indicating: its purpose, the algorithm used (if only it is not obvious) and, possibly, assumptions about the environment in which the function operates ;

4] comment on the definition of each global variable;

5] to give a certain number of comments in those places where the algorithm is not obvious or not portable;

6] almost nothing else.

Let's give an example:

```
// tbl.c: Implementation of the name table.
/ *
        Gaussian method used
        see Ralston, "Getting Started in ..." p. 411.
* /
// swap () assumes AT&T stack starts at 3B20.
/ ***********************************
        Copyright (c) 1991 AT&T, Inc
        All rights reserved
************************************ /
```

Correctly selected and well-written comments play an important role in the program. Writing good comments is as difficult as writing the program itself, and it's an art worth improving.

Note that if the function uses only comments like //, then any part of it can be made a comment using / * * /, and vice versa.

3.5 Exercises

1. (* 1) Rewrite the following for loop using the while statement:

```
for (i = 0; i <max_length; i ++)
        if (input_line [i] == '?') quest_count ++;
```

Write a loop using a pointer as its control variable so that the condition looks like * p == '?'.

2. (* 1) Specify the order in which the following expressions are evaluated by specifying a complete bracket structure:

```
a = b + c * d << 2 & 8
a & 077! = 3
a == b || a == c && c <5
c = x! = 0
0 <= i <7
f (1,2) + 3
a = - 1 + + b - - 5
a = b == c ++
a = b = c = 0
a [4] [2] * = * b? c: * d * 2
ab, c = d
```

3. (* 2) List 5 different C ++ constructs that are undefined.

4. (* 2) Give 10 different examples of non-portable C ++ constructs .

5. (* 1) What happens when division by zero in your C ++ program? What happens in case of overflow or loss of significance?

6. (* 1) Specify the order in which the following expressions are evaluated by specifying their full parenthesis structure:

```
* p ++
* - p
++ a--
(int *) p-> m
* pm
* a [i]
```

7. (* 2) Write the following functions: strlen () - counting the length of a string, strcpy () - copying strings, and strcmp () - comparing strings. What should be the types of parameters and

function results? Compare these to the standard versions found in <string.h> and in your manual.

8. (* 1) Find out how your translator will respond to errors like this:

```
void f (int a, int b)
{
            if (a = 3) // ...
            if (a & 077 == 0) // ...
            a: = b + 1;
}
```

See what the reaction will be to simpler mistakes.

9. (* 2) Write a cat () function that takes two string parameters and returns a string that is their concatenation. For the resulting string, use the memory allocated with new. Write a function rev () to reverse the string passed to it as a parameter. This means that after calling rev (p), the last p will become the first, and so on.

10. (* 2) What does the following function do?

```
void send (register * to, register * from, register count)
// Pseudo device. All comments deliberately removed
{
            register n = (count + 7) / 8;
            switch (count% 8) {
                        case 0: do {* to ++ = * from ++;
                        case 7: * to ++ = * from + +;
                        case 6: * to ++ = * from ++;
                        case 5: * to ++ = * from ++;
                        case 4: * to ++ = * from ++;
                        case 3: * to ++ = * from ++;
                        case 2: * to ++ = * from ++;
                        case 1: * to ++ = * from ++;
                                                                } while (--n> 0);
            }
}
```

What could be the meaning of this function?

11. (* 2) Write a function atoi () that takes a string parameter and returns the corresponding integer. For example, atoi ("123") is equal to 123. Modify the atoi () function so that it can convert to a number a sequence of digits not only in decimal, but also in octal and hexadecimal notation accepted in C ++. Add the ability to translate C ++ symbolic constants. Write an itoa () function to convert an integer value to its string representation.

12. (* 2) Rewrite get_token () ($$ 3.12) so that it reads an entire line into a buffer, and then outputs tokens by reading one character at a time from the buffer.

13. (* 2) Enter functions such as sqrt (), log () and sin () into the calculator program from $$ 3.1 . Hint: give predefined names and call functions using an array of pointers to them. Remember to check the parameters passed to these functions.

14. (* 3) Enter the ability to define custom functions in the calculator . Hint: define a function as a sequence of statements, as if defined by the user himself. This sequence can be stored either as a character string or as a list of tokens. When a function is called, operations must be selected and executed. If user-defined functions can have parameters, then you will have to come up with a record form for them.

15. (* 1.5) Redesign the calculator program using the symbol structure instead of the static variables name_string and number_value:

```
struct symbol {
        token_value tok;
        union {
                double number_value;
                char * name_string;
        };
};
```

16. (* 2.5) Write a program that removes all comments from a C ++ program. This means that you need to read symbols from cin and delete comments of two kinds: // and / * * /. Write the resulting text to cout. Do not worry about the beautiful appearance of the resulting text (this is another, more difficult task). The correctness of the

programs is not important. Consider the possibility of the symbols //, / * and * / appearing in comments, strings, and symbolic constants.

17. (* 2) Explore various programs and find out which methods of highlighting text with spaces and which comments are used.

CHAPTER 4.

Iteration is human, and recursion is god.
- L. Deutsch

All non-trivial programs are composed of several separately broadcast units, traditionally called files. This chapter describes how separately translated functions can call each other, how they can share data, and how to achieve consistency of types used in different program files. Functions are discussed in detail, including: parameter passing, function name overloading, standard parameter values, function pointers, and, of course, function descriptions and definitions. The chapter ends with a discussion of the macro capabilities of the language.

4.1 Introduction

The role of a file in C ++ is reduced to the fact that it defines the file scope ($$ R.3.2). This is the scope of global functions (both static and substitutions), as well as global variables (both static and const). In addition, a file is a traditional storage unit in the system as well as a translation unit. Typically, systems store, translate, and present a C ++ program to the user as multiple files, although there are systems that work differently. This chapter will discuss mainly the traditional use of files.

It is usually impossible to put the entire program in one file, since programs of standard functions and programs of the operating system cannot be included in text form in a user program. In general, putting the entire user program in one file is usually inconvenient and impractical. Breaking a program into files can make it easier to understand the overall structure of the program and gives the translator the ability to maintain that structure. If the unit of translation is a file, then even with a slight change in it, it should be re-translated. Even for programs that are not too large , the time for rebroadcasting can be significantly reduced if it is split into files of a suitable size.

Let's go back to the calculator example. The solution was given as a single file. When you try to broadcast it, there will inevitably be some problems with the order of the descriptions. At least one "fake" description will have to be added to the text so that the translator can understand the expr (), term (), and prim () functions that use each other . The text of the program shows that it consists of four parts: the lexical analyzer (scanner), the analyzer itself, the table of names and the driver. However, this fact is not reflected in the program itself. In fact, the calculator was not programmed that way. You shouldn't write a program like that. Even if you do not take into account all the recommendations on programming, maintenance and optimization for such a "wasted" program, it should still be created from several files, at least for convenience.

To make separate translation possible, the programmer must provide descriptions from which the translator will get enough information about the types to translate a file that is only part of the program. The requirement of consistency in the use of all names and types for a program consisting of several separately translated parts is just as true as for a program consisting of a single file. This is only possible if the descriptions in different translation units are consistent. There are tools in your programming system that can determine if this is done. In particular, the link editor reveals many inconsistencies. A link editor is a program that links separately translated parts of a program by name. It is sometimes mistakenly referred to as the bootloader.

4.2 Binding

Unless explicitly defined otherwise, a name that is not local to some function or class must denote the same type, value, function, or object in all translation units of a given program. In other words, there can be only one non-local type, value, function, or object with a given name in a program . Consider two files as an example:

```
// file1.c
int a = 1;
int f () {/ * some operators * /}

// file2.c
extern int a;
int f ();
void g () {a = f (); }
```

The g () function uses the same a and f () that are defined in file1.c. The extern keyword indicates that the description for a in file2.c is only a description, not a definition. If an initialization of a was present, then extern would simply be ignored, since an initialized declaration is always considered a definition. Any object in the program can be defined only once. It can be described more than once, but all descriptions must be consistent in type. For example:

```
// file1.c:
int a = 1;
int b = 1;
extern int c;

// file2.c:
int a;
extern double b;
extern int c;
```

There are three errors here: the variable a is defined twice ("int a;" is a definition meaning "int a = 0;"); b described twice, with different types; c is described twice, but undefined. The translator, which processes files separately, cannot detect such errors (linking errors) , but most of them are detected by the linker.

The following program is valid in C but not C ++:

```
// file1.c:
int a;
int f () {return a; }

// file2.c:
int a;
int g () {r eturn f (); }
```

First, it is a mistake to call f () in file2.c, since f () is not described in that file. Second, the program files cannot be linked correctly because a is defined twice.

If a name is declared static, it becomes local to this file. For example:

```
// file1.c:
static int a = 6;
static int f () {/ * ... * /}

// file2.c:
```

```
static int a = 7;
static int f () {/ * ... * /}
```

The above program is correct because a and f are defined to be static. Each file has its own variable a and function f () .

If the variables and functions in this part of the program are described as static, then this part of the program is easier to understand, since you do not need to look into other parts. It is also useful to describe functions as static because the translator is given the opportunity to create a simpler version of the function call operation. If the name of an object or function is local in a given file, then the object is said to be bound internally. Conversely, if the name of an object or function is not local in a given file, then it is subject to external linking.

It is usually said that type names, i.e. classes and enumerations cannot be linked. The names of global classes and enumerations must be unique throughout the program and have a single definition. Therefore, if there are even two identical definitions of the same class, it is still an error:

```
// file1.c:
struct S {int a; char b; };
extern void f (S *);

// file2.c:
struct S {int a; char b; };
void f (S * p) {/ * ... * /}
```

But be careful: most C ++ programming systems are unable to recognize the identity of two class declarations. Such duplication can cause rather subtle errors (after all, classes in different files will be considered different).

Global lookup functions are bound internally, and the same is true for constants by default. Type synonyms, i.e. the typedef names are local to their file, so the descriptions in the two files below do not contradict each other:

```
// file1.c:
typedef int T;
const int a = 7;
inline T f (int i) {return i + a; }
```

```
// file2.c:
typedef v oid T;
const int a = 8;
inline T f (double d) {cout << d; }
```

A constant can only get external binding by using an explicit description:

```
// file3.c:
extern const int a;
const int a = 77;

// file4.c:
extern const int a;
void g () {cout << a; }
```

In this example, g () will print 77.

4.3 Header files

The types of one object or function must be consistent across all their descriptions. Both the input text processed by the translator and the associated program parts must be consistent in types . There is a simple, albeit imperfect, way to ensure consistency of descriptions in different files. These are: include in the input files containing statements and data definitions, header files that contain interface information.

The #include macro serves as a means of including texts , which allows you to collect several source files of the program into one file (translation unit) . Command

```
#include "include-file"
```

replaces the line in which it was specified with the contents of includefile. Naturally, this content must be C ++ text , since the translator will read it. Typically, the include operation is implemented in a separate program called the C ++ preprocessor . It is called by the programming system before the actual translation to process such commands in the input text. Another solution is also possible : the part of the translator that directly works with the input text processes the commands to include files as they appear in the text. In the programming system in which the author works, to see the result of the commands for including files, you need to set the command:

```
CC -E file.c
```

This command to process the file file.c starts the preprocessor (and nothing more!), Just like the CC command without the -E flag starts the translator itself .

To include files from standard directories (usually directories named INCLUDE), use the angle brackets <and> instead of quotation marks . For example:

```
#include <stream.h> // include from the standard directory
#include "myheader.h" // include from the current directory
```

Including from standard directories has the advantage that the names of these directories are not associated in any way with a specific program (usually , the include files are looked for in the / usr / include / CC directory and then in / usr / include). Unfortunately, the gaps in this command are significant:

```
#include <stream.h> // <stream.h> will not be found
```

It would be ridiculous if every time the file was required to be re-translated before it was included . Usually, include files contain only descriptions, not statements and definitions that require significant translational processing. In addition, the programming system can pre-translate the header files, if, of course, it is so advanced that it is able to do so without changing the semantics of the program.

Let's indicate what the header file can contain:

```
Type definitions struct p oint {int x, y; };
           Type templates template <class T>
                     class V {/ * ... * /}
           Function descriptions extern int strlen (const char *);
           Definitions inline char get () {return * p ++; }
           substitution functions
           Data descriptions extern int a;
 Constant definitions const float pi = 3.141593;
 Enumerations enum bool {false, true};
 Descriptions of names class Matrix;
 File Include Commands #include <si gnal.h>
 Macros #define Case break; case
 Comments / * check for end of file * /
```

Listing what to put in a header file is not a requirement of the language, it is simply advice for judicious use of include files. On the other hand, a header file should never contain:

```
Common function definitions char get () {return * p ++; }
Data definitions           int a;
Definitions of compound constants const tb [i] = {/ * ... * /};
```

Traditionally, header files have a .h extension, and files containing function or data definitions have a .c extension. These are sometimes referred to as "h-files" or "c-files" respectively. Use other extensions for these files: .C, cxx, .cpp and .cc. You will find the accepted extension in your reference manual. Macro tools are described in $$ 4.7. Note only that in C ++ they are not used as widely as in C, because C ++ has certain capabilities in the language itself: definitions of constants (const), substitution functions (inline), allowing a simpler call operation, and type templates to generate a family of types and functions ($$ 8).

The advice to put in the header file definitions only simple, but not compound, constants is explained by a completely pragmatic reason. It's just that most translators aren't smart enough to prevent unnecessary copies of a compound constant from being created. Generally speaking, the simpler version is always the more general one, which means that the translator must consider it first in order to create a good program.

4.3.1 Single header file

The easiest way is to split the program into several files as follows: place the definitions of all functions and data in a number of input files, and describe all the types necessary for communication between them in a single header file. All input files will include a header file. The calculator program can be broken down into four .c input files: lex.c, syn.c, table.c, and main.c. The dc.h header file will contain a description of each name that is used in more than one .c file:

```
// dc.h: general description for the calculator
#include <iostream.h>
enum token_value {
            NAME, NUMBER, END,
            PLUS = '+', MINUS = '-', MUL = '*', DIV = '/',
            PRINT = ';', ASSIGN = '=', LP = '(', RP = ')'
};
extern int no_of_errors;
extern double error (const char * s);
extern token_value get_token ();
extern token_value curr_tok;
extern double number_value;
extern char name_string [256];
extern double expr ();
extern double term ();
extern double prim ();
struct name {
            char * string;
            name * next;
            double value;
};
extern name * look (const char * p, int ins = 0);
inline name * insert (const char * s) {return look (s, 1); }
```

If you omit the operators themselves, lex.c should look like this:

```
// lex.c: input and lexical analysis
#include "dc.h"
#include <ctype.h>
token_value curr_tok;
double number_value;
char name_string [256];
token_value get_token () {/ * ... * /}
```

Using the compiled header file, we will ensure that the description of each object entered by the user will necessarily appear in the file where this object is defined. Indeed, when processing the lex.c file, the translator will encounter descriptions

```
extern token_value get _token ();
// ...
token_value get_token () {/ * ... * /}
```

This will allow the translator to detect any discrepancy in the types specified in the description of the given name. For example, if the get_token () function were declared as token_value, but defined as int, translation of the lex.c file would throw an error: type mismatch.

The syn.c file might look like this:

```
// syn.c: parsing and computation

#include "dc.h"

double prim () {/ * ... * /}
double term () {/ * ... * /}
double expr () {/ * ... * /}
```

The table.c file might look like this:

```
// table.c: name table and lookup function
#include "dc.h"
extern char * strcmp (const char *, const char *);
extern char * strcpy (char *, const char *);
extern int strlen (const char *);
const int TBLSZ = 23;
name * table [TBLSZ];
name * look (ch ar * p, int ins) {/ * ... * /}
```

Note that since the string functions are described in the table.c file itself, the translator cannot check for consistency of these descriptions by type. It is always better to include an appropriate header file than to describe some name as extern in the .c file. This can lead to the inclusion of "too much", but such inclusion is not scary, since it does not affect the speed of the program and its size, and saves the programmer time. Let's say the strlen () function is again described in the following main.c. file. This is just unnecessary input of characters and a potential source of errors. the translator will not be able to detect discrepancies in the two strlen () declarations (however, the linker can do this). This problem would not have arisen if the dc.h file contained all extern declarations as originally intended. This kind of negligence is present in our example because it is typical for C programs. It is very natural for a programmer, but often leads to errors and such programs that are difficult to maintain. So the warning has been made!

Finally, here's the main.c file:

```
// main.c: initialization, main loop, error handling
#include "dc.h"
double error (char * s) {/ * ... * /}
```

```
extern int strlen (const char *);
int main (int argc, c har * argv []) {/ * ... * /}
```

In one important case, header files are very inconvenient. Through a series of header files and a standard library, they extend the capabilities of the language by introducing many types (both generic and application-specific; see Chapters 5-9). In this case, the text of each translation unit can begin with thousands of lines of header files. The content of the library header files is usually stable and rarely changes. A pretranslator that handles it would be very useful here. In fact, you need a special-purpose language with its own translator. But there are no established methods for constructing such a pretranslator yet.

4.3.2 Multiple header files

Splitting a program into one header file is more suitable for small programs, some parts of which do not have an independent purpose. For such programs, it is permissible that the header file cannot be used to determine whose descriptions are there and for what reason. Only comments can help here. An alternative solution is possible: let each part of the program have its own header file, which defines the means provided to other parts. There will now be a .h file for each .c file that defines what the former can provide. Each .c file will include both its own .h file and some other .h files based on its needs.

Let's try to use such an organization for a calculator program. Note that the error () function is needed in almost all program functions, and it itself uses only <iostream.h>. This situation is typical for functions that handle errors. It should be separated from the main.c file:

```
// error.h: error handling
extern int no_of_errors;
extern double error (const char * s);

// error.c
#include <iostream.h>
#inc lude "error.h"
int no_of_errors;
double error (const char * s) {/ * ... * /}
```

With this approach to splitting a program, each pair of .c and .h files can be thought of as a module where the .h file defines its interface and the .c file defines its implementation.

The name table does not depend on any part of the calculator except for the error handling part. Now this fact can be expressed explicitly:

```
// table.h: description of the table of names
struct name {
            char * string;
            name * next;
            double value;
};
extern name * look (const char * p, int ins = 0);
inline name * insert (const char * s) {return look (s, 1); }

// table.h: name table definition
#include "error.h"
#include <string.h>
#include "table.h"
const int TBLSZ = 23;
name * table [TBLSZ];
name * look (const char * p, int ins) {/ * ... * /}
```

Note that now the descriptions of string functions are taken from the include file <string.h>. This removes another source of errors.

```
// lex.h: descriptions for input and lexical analysis
enum token_value {
            NAME, NUMBER, END,
            PLUS = '+', MINUS = '-',    MUL = '*',
            PRINT = ';', ASSIGN = '=', LP = '(', RP = ')'
};
extern token_value curr_tok;
extern double number_value;
extern char name_string [256];
extern token_value get_token ();
```

The interface with the lexical analyzer is quite confusing. Since there are not enough matching types for tokens, the user of get_token () is given the same number_value and name_string buffers that the lexical analyzer itself works with.

```
// lex.c: definitions for input and lexical analysis
#include <iostream.h>
#i nclude <ctype.h>
#include "error.h"
#include "lex.h"
token_value curr_tok;
double number_value;
char name_string [256];
token_value get_token () {/ * ... * /}
```

The parser interface is clearly defined:

```
// syn.h: descriptions for parsing and computation
extern double expr ();
extern double term ();
extern double prim ();

// syn.c: definitions for parsing and computation
#include "error.h"
#include "lex.h"
#include "syn.h"

double prim () {/ * ... * /}
double term () {/ * ... * / }
double expr () {/ * ... * /}
```

As usual, the definition of the main program is trivial:

```
// main.c: main program
#include <iostream.h>
#include "error.h"
#include "lex.h"
#include "syn.h"
#include "table.h"
int main (int argc, char * argv []) {/ * ... * /}
```

How many header files should be used for a given program depends on many factors. Most of them are determined by the way files are processed on your system, and not in C ++ itself. For example, if your editor cannot work with several files at the same time, dialog processing of several header files becomes difficult. Another example: it may turn out that opening and reading 10 files of 50 lines each takes significantly longer than opening and reading one file of 500 lines. As a result, you will have to think carefully before breaking up a small program using multiple header files. Caveat: you can usually handle a lot of about 10 header files (plus the standard header files). If you split the program into minimal logical units with header files (for example, creating its own header file for each structure), you can very easily get an unmanageable set of hundreds of header files.

4.4 Linking with programs in other languages

C ++ programs often contain parts written in other languages, and vice versa, often a C ++ fragment is used in programs written in other languages. It is quite difficult to put together into one program fragments written in different languages, or written in the same language, but in programming systems with different binding conventions , it is quite difficult. For example, different languages or different implementations of the same language can differ in the use of registers when passing parameters, the order in which parameters are placed on the stack, the packaging of built-in types such as integers or strings, the format of function names that the translator passes to the binder, the amount of type checking required from link editor. To simplify the task, you can specify a binding condition in the description of external ones . For example, the following description declares strcpy to be an external function and specifies that it should bind according to the binding order in C:

```
extern "C" char * strcpy (char *, const char *);
```

The result of this description is different from the result of a regular description

```
extern char * strcpy (char *, const char *);
```

only the binding order for functions calling strcpy (). The very semantics of the call and, in particular, the control of the actual parameters will be the same in both cases. It makes sense to use the extern "C" description also because the C and C ++ languages, as well as their implementations, are close to each other. Note that in the extern "C" description, the reference to C refers to the order of linking, not the language, and is often used to refer to Fortran or assembler. These languages obey to some degree the binding order for C.

It is tedious to add "C" to many external descriptions, and it is possible to specify such a specification at once for a group of descriptions. For example:

```
extern "C" {
        c har * strcpy (char *, const char);
        int strcmp (const char *, const char *)
        int strlen (const char *)
        // ...
}
```

You can include the entire C header file in such a construct to indicate that it obeys binding for C ++, for example:

```
extern "C" {
        #include <s tring.h>
}
```

Usually, using this technique, such a file for C ++ is obtained from the standard header file for C. A different solution is possible using conditional translation:

```
#ifdef __cplusplus
        extern "C" {
#endif
char * strcpy (char *, const char *);
int strcmp (const char *, const char *);
int strlen (const char *);
// ...
#ifdef __cplusplus
}
#endif
```

The predefined __cplusplus macro is needed to bypass the extern "C" {...} construct if the header file is used for C.

Because extern "C" {...} only affects the binding order, it can contain any description, for example:

```
extern "C" {
// arbitrary descriptions
// eg:
        static int st;
        int glob;
}
```

The memory class and scope of the described objects does not change in any way, so st is still subject to internal binding, and glob remains a global variable.

Again, the extern "C" declaration only affects the binding order and does not affect the function call order. In particular, a function described as extern "C" still obeys typing and actual parameter conversion rules, which are stricter in C ++ than in C. For example:

```
extern "C" int f ();
int g ()
{
        return f (1); // error: there should be no parameters
}
```

4.5 How to create a library

Such phrases are widespread (and in this book too): "put in a library", "search in such and such a library". What do they mean for C ++ programs? Unfortunately the answer depends on the system you are using. This section describes how to create and use a library for the tenth version of the UNIX system. Other systems should provide similar capabilities. The library consists of .o files, which are the result of translating .c files. Usually there is one or more .h files, which contain the descriptions necessary for calling .o files. Let us consider, as an example, how, for a clearly unspecified set of users, it is quite convenient to define a certain set of standard mathematical functions. The header file can look like this:

```
extern "C" { // standard math functions
                                                // usually written in
C
        double sqrt (double); // subset of <math.h>
        double sin (double);
        double cos (double);
        double exp (double);
        double log (double);
```

```
            // ...
    }
```
The definitions of these functions will be found in sqrt.c, sin.c, cos.c, exp.c, and log.c files, respectively.

A library named math.a can be created using the following commands:

```
$ CC -c sqrt.c sin.c cos.c exp.c log.c
$ ar cr math.a sqrt.o sin.o cos.o exp.o log.o
$ ranlib math.a
```

Here the $ symbol is a system prompt.

First, the sources are translated, and modules with the same names are obtained. The ar (archiver) command creates an archive named math.a. Finally, the archive is indexed for quick access to functions. If your system does not have a ranlib command (it may not be needed), then at least you can find a reference to the name ar in the reference manual. To use the library in your program, you need to set the translation mode as follows:

```
$ CC myprog.c math.a
```

The question arises: what does the math.a library give us? You could use .o files directly, like this:

```
$ CC myprog.c sqrt.o sin.o cos.o exp.o log.o
```

The point is, in many cases, it is difficult to correctly indicate which .o files are actually needed. The above command used all of them. If only sqrt () and cos () are called in myprog, then, apparently, it is enough to set the following command:

```
$ CC myprog.c sqrt.o cos.o
```

But this will not be true, since the cos () function calls sin ().

The linker, which is invoked by the CC command to process .a files (in our case, the math.a file), can extract only the necessary .o files from the set of files that make up the library. In other words, linking to a library allows programs to include many definitions of the same name (including definitions of functions and variables used only by internal functions that the user will never know about). At the same time, only the minimum required number of definitions will be included in the resulting program.

4.6 Functions

The most common way of doing things in C ++ is by calling a function that does those things. A function definition is a description of how to execute them. Undescribed functions cannot be called.

4.6.1 Function descriptions

The description of a function contains its name, the type of return value (if any), and the number and types of parameters that must be specified when calling the function. For example:

```
extern double sqrt (double);
extern elem * next_elem ();
extern char * strcpy (char * to, const char * from);
extern void exit (int);
```

The semantics of parameter passing is identical to the semantics of initialization: the types of the actual parameters are checked and, if necessary, implicit type conversions are performed. So, if we take into account the descriptions given, then in the following definition:

```
double sr2 = sqrt (2);
```

contains the correct call to sqrt () with a floating point value of 2.0. The control and type conversion of the actual parameter is of great importance in C ++.

Parameter names can be specified in the function description. This makes the program easier to read, but the translator simply ignores these names.

4.6.2 Function definitions

Each function called in a program must be defined somewhere in it, and only once. The definition of a function is its description, which contains the body of the function. For example:

```
extern void swap (int *, int *); // description
void swap (int * p, int * q) // definition
{
          int t = * p;
          * p = * q;
          * q = * t;
}
```

It is not uncommon for a function definition to not use some parameters:

```
void search (table * t, const char * key, const char *)
```

```
{
        // third parameter is not used
        // ...
}
```

As you can see from this example, the parameter is not used if its name is not specified. Such functions appear when the program is simplified or if one counts on its further expansion. In both cases, reserving space in the function definition for an unused parameter ensures that other functions containing a call to this one do not have to be changed.

It has already been said that a function can be defined as an inline. For example:

```
inline fac (int i) {return i <2? 1: n * fac (n-1); }
```

The inline specification serves as a hint to the translator that a call to the fac function can be implemented by substituting its body, and not using the usual function calling mechanism ($$ R.7.1.2). A good optimizing translator, instead of generating a call to fac (6), can simply use the constant 720. Because of the presence of mutually recursive calls to substitution functions, as well as substitution functions whose recursiveness depends on the input, it cannot be argued that every call to a substitution function is actually implemented. substitution of her body. The degree of optimization performed by the translator cannot be formalized, so some translators will create commands 6 * 5 * 4 * 3 * 2 * 1, others will create 6 * fac (5), and some will limit themselves to an unoptimized call to fac (6).

For the implementation of a call by substitution to become possible even for not too advanced programming systems, it is necessary that not only the definition, but also the description of the substitution function be in the current scope. Otherwise, the inline specification does not affect the call semantics.

4.6.3 Passing parameters

When a function is called, memory is allocated for its formal parameters, and each formal parameter is initialized with the value of the corresponding actual parameter. Parameter passing semantics are identical to initialization semantics. In particular, the types of the formal parameter and the corresponding actual parameter are checked, and all standard and custom type conversions are performed. There are special rules for passing arrays

($$ 4.6.5). It is possible to pass a parameter bypassing the type control ($$ 4.6.8), and the ability to set a standard parameter value ($$ 4.6.7). Consider the function:

```
void f (int val, int & ref)
{
        val ++;
        ref ++;
}
```

When f () is called in val ++, the local copy of the first actual parameter is incremented, whereas in ref ++, the second actual parameter itself is incremented. Therefore, in the function

```
void g ()
{
        int i = 1;
        int j = 1;
        f (i, j);
}
```

j will increase, but not i. The first parameter i is passed by value, and the second parameter j is passed by reference. In $$ 2.3.10 we said that functions that modify their by-reference parameter are harder to understand and therefore better avoided (see also $$ 10.2.2). But large objects are obviously much more efficient to pass by reference than by value. True, you can declare a parameter with the const specification to ensure that passing by reference is only used for efficiency, and the called function cannot change the value of the object:

```
void f (const large & arg)
{
        // "arg" value cannot be changed without explicit
        // type conversion operations
}
```

If const is not specified in the reference parameter description, then this is considered an intention to modify the passed object:

```
void g (large & arg); // it is assumed that in g () arg will change
```

Hence the moral: use const whenever possible.

Likewise, the const specification of a pointer parameter says that the referenced object will not change in the called function. For example:

```
extern int strlen (const char *); // from <string.h>
extern char * strcpy (char * to, const char * from);
ex tern int strcmp (const char *, const char *);
```

The value of this technique grows with the growth of the program.

Note that the semantics of parameter passing differs from the semantics of assignment. This difference is significant for parameters that are const or reference, and for parameters with a user-defined type ($ 1.4.2).

A literal, constant, and parameter to be converted can be passed as a parameter of type const &, but without a const specification you cannot pass it. By allowing conversions for a parameter of type const T &, we guarantee that it can take values from the same set as a parameter of type T, the value of which is passed, if necessary, using a temporary variable.

```
float fsqrt (const float &); // sqrt function in Fortran style
void g (double d)
{
            float r;
            r = fsqrt (2.0f); // passing a link to a temporary
                                                              // variable containing
2.0f
            r = fsqrt (r); // pass a link to r
            r = fsqrt (d); // passing a link to a temporary
                                                              // variable containing
float (d)
}
```

The prohibition on type conversions for reference parameters without the const specification was introduced in order to avoid ridiculous errors associated with the use of temporary variables when passing parameters:

```
void update (float & i);
void g (double d)
{
            float r;
            update (2.0f); // error: parameter-constant
            update (r); // ok: a reference to r is passed
            update (d); // error: you need to convert the type here
}
```

4.6.4 Return value

If a function is not declared void, it must return a value. For example:

```
int f () {} // error
void g () {} // ok
```

The return value is specified in the return statement in the body of the function. For example:

```
int fac (int n) {return (n> 1)? n * fac (n-1): 1; }
```

There can be several return statements in the body of a function:

```
int fac (int n)
{
            if (n> 1)
                        return n * f ac (n-1);
            else
                        return 1;
}
```

Like passing parameters, returning a function is equivalent to initializing. The return statement is considered to initialize a variable with a return type. The type of the expression in the return statement is checked against the type of the function, and all standard and custom type conversions are performed. For example:

```
double f ()
{
            // ...
            return 1; // implicitly converts to double (1)
}
```

Each time a function is called, a new copy of its formal parameters and automatic variables is created. The memory occupied by them after exiting the function will be used again, so it is unreasonable to return a pointer to a local variable. The contents of the memory to which such a pointer is configured can change in an unpredictable way:

```
in t * f ()
{
        int local = 1;
        // ...
        return & local; // mistake
}
```

This error is not as common as a similar error when the function type is a reference:

```
int & f ()
{
        int local = 1;
        // ...
        return local; // mistake
}
```

Fortunately, the translator warns that a local variable reference is being returned. Here's another example:

```
int & f () {return 1; } // mistake
```

4.6.5 Array parameter

If an array is specified as a function parameter, a pointer to its first element is passed. For example:

```
int strlen (const char *);
void f ()
{
        char v [] = "array";
        strlen (v);
        strlen ("Nikolay");
}
```

This means that the actual parameter of type T [] is converted to type T *, and then passed. Therefore, assigning a formal array parameter to an element changes that element. In other words, arrays differ from other types in that they are not passed and cannot be passed by value.

In the called function, the size of the passed array is unknown. This is frustrating, but there are several ways to get around this difficulty. First of all, all strings are null terminated, which means their size is easy to calculate. You can pass one more parameter that sets the size of the array. Another way is to define a structure that contains a pointer to the array and the size of the array, and pass it as a parameter (see also $$ 1.2.5). For example:

```
void compute1 (int * vec_ptr, int vec_size); // 1st way
struct vec { // 2nd way
        int * ptr;
        int size;
};
void compute2 (vec v);
```

It is more difficult with multidimensional arrays, but often you can use an array of pointers instead, reducing these cases to one-dimensional arrays. For example:

```
char * day [] = {
        "mon", "tue", "wed", "thu", "fri", "sat", "sun"
};
```

Now let's consider a function that works with a two-dimensional array - a matrix. If the sizes of both indices are known at the translation stage, then there are no problems:

```
void print_m34 (int m [3] [4])
{
        for (int i = 0; i <3; i ++) {
                for (int j = 0; j <4; J ++)
                        cout << '' << m [i] [j];
                cout << '\ n';
        }
}
```

Of course, the matrix is still passed as a pointer, and the dimensions are just for completeness.

The first dimension is not important for calculating the address of the element ($$ R.8.2.4), so it can be passed as a parameter:

```
void print_mi4 (int m [] [4], int dim1)
{
for (int i = 0; i <dim1; i ++) {
        for (int j = 0; j <4; j ++)
                cout << '' << m [i] [j];
        cout << '\ n';
```

```
            }
}
```

The most difficult case is when you need to transfer both dimensions. Here the "obvious" solution is simply not applicable:

```
void print_mij (int m [] [], int dim1, int dim2) // error
{
            for (int i = 0; i <dim1; i ++) {
                        for (int j = 0; j <dim2; j ++)
                                    cout << " << m [i] [j];
                        cout << '\ n';
            }
}
```

First, the description of the m [] [] parameter is invalid, because the second dimension must be known to calculate the address of an element of a multidimensional array. Secondly, the expression m [i] [j] is evaluated as * (* (m + i) + j), and this, apparently, is not what the programmer had in mind. Here's the correct solution:

```
void print_mij (int ** m, int dim1, int dim2)
{
            for (int i = 0; i <dim1; i ++) {
                        for (int j = 0; j <dim2; j ++)
                                    cout << " << ((int *) m) [i * dim2 + j]; //
confused
                        cout << '\ n';
            }
}
```

The expression used to select an element of the matrix is equivalent to the one that the translator creates for the same purpose when the last dimension is known. You can introduce an additional variable to make this expression clearer:

```
int * v = (int *) m;
// ...
v [i * dim2 + j]
```

It is better to hide such rather complicated places in the program. You can define the type of a multidimensional array with an appropriate indexing operation. Then the user may not know how the data is arranged in the array (see Exercise 18 in $$ 7.13).

4.6.6 Function name overloading

It usually makes sense to give different names to different functions. If several functions perform the same action on objects of different types, then it is more convenient to give the same names to all these functions. Name overloading refers to its use to indicate different operations on different types. Actually, overloading is used for basic C ++ operations. Indeed: there is only one name for addition operations, +, but it is used for addition of integers, floating point numbers, and pointers. This approach can easily be extended to user-defined operations, i.e. on the function. For example:

```
void print (int); // print whole
void print (const char *) // print a string of characters
```

For a translator in such overloaded functions, there is only one common thing - the name. Obviously, these functions are similar in meaning, but the language does not facilitate and does not prevent the allocation of overloaded functions. Thus, the definition of overloaded functions is primarily for convenience. But for functions with traditional names such as sqrt, print, or open, this convenience cannot be overlooked. If the name itself plays an important semantic role, for example, in operations such as +, * and << ($$ 7.2), or for a class constructor ($$ 5.2.4 and $$ 7.3.1), then such convenience becomes an essential factor. When calling a function named f, the translator must figure out which function f should be called. To do this, the types of actual parameters specified in the call are compared with the types of formal parameters of all descriptions of functions with the name f. As a result, the function is called, whose formal parameters match the call parameters in the best way, or an error is thrown if no such function is found. For example:

```
void print (double);
void print (long);
void f ()
{
        print (1L); // print (long)
        pr int (1.0); // print (double)
        print (1); // error, ambiguity: what to call
```

```
                                                    // print (long (1)) or
print (double (1))?
}
```

The rules for matching parameters are described in detail in $$ R.13.2. It is enough to give their essence here. Rules are applied in the following order, in descending order of priority:

[1] Exact matching: matching occurred without any type conversions or only with inevitable conversions (for example, array name to pointer, function name to function pointer, and type T to const T).

[2] Matching using the standard integer conversions defined in $$ R.4.1 (ie char to int, short to int and their unsigned doubles to int), as well as float to double conversions.

[3] Matching using standard conversions defined in $$ R.4 (eg int to double, derived * to base *, unsigned to int).

[4] Matching using custom transforms ($$ R.12.3).

[5] Matching using ellipsis ... in the function description .

If two matches are found by the highest priority rule, then the call is considered ambiguous, and therefore erroneous. These parameter matching rules work with the C and C ++ numeric conversion rules. Let there be such descriptions of the print function:

```
void print (int);
void print (const char *);
void print (double);
void print (long);
void print (char);
```

Then the results of the following calls to print () will be like this:

```
void h (char c, int i, short s, float f)
{
        print (c); // exact match: print (char ) is called
        print (i); // exact match: print (int) is called
        print (s); // standard integer conversion:
                                                    // print (int) is called
        print (f); // standard conversion:
                                                    // print (double) is
called
```

```
            print ('a'); // exact match: print (char) is called
            print (49); // exact match: print (int) is called
            print (0); // exact match: print (int) is called
            print ("a"); // exact match:
                                            // print (const char *)
   is called
   }
```
Calling print (0) results in a call to print (int), because 0 is int. Calling print ('a') results in a call to print (char), because 'a' is of type char ($$ R.2.5.2).

Note that the order of descriptions of the functions under consideration does not affect the resolution of ambiguity under overloading, and the types of values returned by the functions are not taken into account at all.

Based on these rules, it can be guaranteed that if the efficiency or accuracy of calculations differ significantly for the types in question, then a function is called that implements the simplest algorithm. For example:

```
   int pow (int, int);
   double pow (double, double); // from <math.h>
   complex pow (double, complex); // from <complex.h>
   complex pow (complex, int);
   complex pow (complex, double);
   complex pow (complex, complex);
   void k (complex z)
   {
            int i = pow (2,2); // pow (int, int) is called
            double d = pow (2.0,2); // pow (double, double) is called
            complex z2 = pow (2, z); // pow (double, complex) is called
            complex z3 = pow (z, 2); // pow (complex, int) is called
            complex z4 = pow (z, z); // pow (complex, complex) is called
   }
```

4.6.7 Default parameter values

In general, a function may have more parameters than in the simplest and most commonly used cases. In particular, this is inherent in functions that construct objects (for example, constructors, see $$ 5.2.4). For more flexible use of these functions, optional parameters are sometimes used. Let's take the function of printing an integer as an example. It is perfectly reasonable to use the base of the printed number as an optional parameter, although in most cases the numbers will be printed as decimal integers. Next function

```
void print (int value, int base = 10);
void F ()
{
            print (31);
            print (31,10);
            print (31,16);
            print (31,2);
}
```

will print numbers like this:

```
31 31 1f 11111
```

An overload of the print function could have been used instead of the default parameter value:

```
void print (int value, int base);
inline void print (int value) {print (value, 10); }
```

However, in the latter version, the text of the program does not so clearly demonstrate the desire to have one print function, but at the same time provide a convenient and concise notation.

The type of the standard parameter is checked against the type of the specified value during translation of the function description, and the value of this parameter is calculated at the time of the function call. You can set a standard value only for the final consecutive parameters:

```
int f (int, int = 0, char * = 0); // fine
int g (int = 0, int = 0, char *); // mistake
int h (int = 0, int, char * = 0); // mistake
```

Note that in this context, the presence of a space between the characters * and = is quite significant, since * = is an assignment operation:

```
int nasty (char * = 0); // syntax error
```

4.6.8 Undefined number of parameters

There are functions in whose description it is impossible to specify the number and types of all valid parameters. Then the list of formal parameters ends with an ellipsis (...), which means: "and possibly some more arguments." For example:

```
int printf (const char * ...);
```

When calling printf, a char * parameter must be specified, but there may or may not be other parameters. For example:

```
printf ("Hello, world \ n");
printf ("My name is% s% s \ n", first_name, second_name);
printf ("% d +% d =% d \ n", 2,3,5);
```

Such functions are used to recognize their actual parameters information inaccessible to the translator. In the case of printf, the first parameter is a string specifying the output format. It can contain special characters that allow you to correctly interpret subsequent parameters. For example,% s means - "there will be an actual parameter of type char *",% d means - "there will be an actual parameter of type int" (see $$ 10.6). But the translator does not know this, and therefore it cannot make sure that the declared parameters are actually present in the call and have the appropriate types. For example, the following call

```
printf ("My name is% s% s \ n", 2);
```

broadcasts normally, but will (at best) produce unexpected results. You can check it yourself.

Obviously, since the parameter is not described, the translator has no information for control and standard conversions of the type of this parameter. Therefore, char or short are passed as int and float as double, although the user may have meant otherwise.

A well-designed program may require, as an exception, only a few functions that do not specify all parameter types. To bypass parameter typing, it is better to use function overloading or default parameter values than parameters whose types have not been described. Ellipsis becomes necessary only when not only the types but also the number of parameters can change. Most often, ellipsis is used to define an interface with a library of standard functions in C, if there is no substitute for these functions:

```
extern "C" int fprintf (FILE *, const char * ...);
extern "C" int execl (const char * ...);
```

There is a standard set of macros, found in <stdarg.h>, to select unspecified parameters for these functions. Consider the error response function, the first parameter of which indicates the severity of the error. It can be followed by an arbitrary number of lines. It is necessary to compose an error message, taking into account that each word from it is transmitted as a separate line:

```
extern void error (int ...)
extern char * itoa (int);
main (int argc, char * argv [])
{
        switch (argc) {
                case 1:
                        error (0, argv [0], (char *) 0);
                        break;
                case 2:
                        error (0, argv [0], argv [1 ], (char *) 0);
                        break;
                default:
                        error (1, argv [0],
                        "With", itoa (argc-1), "arguments", (char *) 0);
        }
        // ...
}
```

The itoa function returns a character string representing its integer parameter. The error response function can be defined as follows:

```
#include <stdarg.h>
void error (int severity ...)
/ *
        "severity" is followed by
        null terminated list of strings
* /
{
        va_list ap;
        va_start (ap, severity); // start of parameters
        for (;;) {
                char * p = va_arg (ap, char *);
                if (p == 0) break;
                cerr << p << '';
```

```
            }
            va_end (ap); // clear parameters
            cerr << '\ n';
            if (severity) exit (severity);
}
```

First, when calling va_start (), the va_list is defined and initialized. The parameters of the va_start macro are the name of the va_list type and the last formal parameter. The macro definition va_arg () is used to select in order the undescribed parameters. Each call to va_arg must specify the type of the expected actual parameter. Va_arg () assumes that a parameter of this type is present in the call, but there is usually no way to check this. Before exiting the function in which va_start was called, you must call va_end. The reason is that there may be stack operations in va_start () that make it impossible to return correctly from the function. Va_end () removes any unwanted stack changes.

Casting 0 to (char *) 0 is necessary because sizeof (int) does not have to match sizeof (char *). This example demonstrates all the difficulties that a programmer has to face if he decides to bypass type checking using ellipsis.

4.6.9 Function Pointer

Only two operations with functions are possible: calling and taking an address. The pointer obtained by the last operation can later be used to call the function. For example:

```
void error (char * p) {/ * ... * /}
void (* efct) (char *); // function pointer
void f ()
{
            efct = & error; // efct is set to error function
            (* efct) ("error"); // call error via efct pointer
}
```

To call a function using a pointer (efct in our example), you must first apply the indirection operation to the pointer - * efct. Since the priority of the call operation () is higher than the priority of indirection *, you cannot just write * efct ("error"). This would mean * (efct ("error")), which is an error. For the same reason, parentheses are needed when describing a function pointer. However, you can simply write efct ("error"), since the translator understands that efct is a function pointer and creates commands that call the desired function.

Note that formal parameters in function pointers are described in the same way as in ordinary functions. When assigning to a function pointer, an exact match between the function type and the type of the value being assigned is required. For example:

```
void (* pf) (char *); // pointer to void (char *)
void f1 (ch ar *); // void (char *);
int f2 (char *); // int (char *);
void f3 (int *); // void (int *);
void f ()
{
 pf = & f1; // fine
        pf = & f2; // error: wrong return type
                                         // values
        pf = & f3; // error: wrong parameter type
        (* pf) ("asdf"); // fine
        (* pf) (1); // error: wrong parameter type
        int i = (* pf) ("qwer"); // error: void is assigned to int
}
```

The rules for passing parameters are the same for a regular call and for a pointer call.

It is often more convenient to designate the type of a function pointer by a name than to use a rather complex notation all the time. For example:

```
typedef int (* SIG_TYP) (int); // from <signal.h>
typedef void (SIG_ARG_TYP) (int);
SIG_TYP signal (int, SIG_ARG_TYP);
```

An array of function pointers is also often useful. For example, you can implement a menu system for an editor with mouse-driven input using an array of function pointers that implement commands. It is not possible here to describe such an editor in detail, but here is the most general outline of it:

```
typedef void (* PF) ();
PF edit_ops [] = {// editor commands
        & cut, & paste, & snarf, & search
};
```

```
PF file_ops [] = {// file management
        & open, & reshape, & close, & write
};
```

Next, you need to define and initialize the pointers, with the help of which the functions that implement the commands selected from the menu will be launched. The selection is made by pressing the mouse button:

```
PF * button2 = edit_ops;
PF * button3 = file_ops;
```

For a real editor program, more objects need to be defined to describe each menu item. For example, you need to store a string somewhere that specifies the text that will be displayed for each position. As you navigate the menu system, the mouse button assignments will constantly change. In part, these changes can be thought of as changes to the values of the pointer associated with the given key. If the user has selected a menu item, which is defined, for example, as position 3 for key 2, then the corresponding command is implemented by calling:

```
(* button2 [3]) ();
```

To fully appreciate the power of constructing a pointer to a function, it is worth trying to write a program without it. The menu can be changed over time by adding new functions to the command table.

It is quite easy to create dynamic new menus.

Function pointers help you implement polymorphic routines, i.e. such routines that can be applied to objects of various types:

```
typedef int (* CFT) (void *, void *);
void sort (void * base, unsigned n, unsigned int sz, CFT cmp)
/ *
        Sorting the "base" vector of n elements
        in ascending order;
        the comparison function pointed to by c mp is used.
        The size of the elements is "sz".
        Algorithm very inefficient: bubble sorting
* /
{
        for (int i = 0; i <n-1; i ++)
                for (int j = n-1; i <j; j--) {
                        char * pj = (char *) base + j * sz; // b [j]
                        char * pj1 = pj - sz; // b [j-1]
                        if ((* cmp) ( pj, pj1) <0) {
                                // swap b [j] and b [j-1]
                                for (int k = 0; k <sz; k ++) {
                                        char temp = pj [k];
                                        pj [k] = pj1 [k];
                                        pj1 [k] = temp;
                                }
                        }
                }
}
```

The sort routine does not know the type of objects being sorted; only their number (array size), the size of each element and the function that can compare objects are known . We have chosen the same header for sort () as qsort (), the standard sort function in the C library. Real programs use this function. Let's show how you can sort a table with the following structure using sort () :

```
struct user {
        char * name; // name
        char * id; // password
        int dept; // Department
};
typedef user * Puser;
user heads [] = {
        "Ritchie DM", "dmr", 11271,
        "Sethi R.", "ra vi", 11272,
        "SZYmanski TG", "tgs", 11273,
        Schryer NL, nls, 11274,
        "Schryer NL", "nls", 11275
        "Kernighan BW", "bwk", 11276
};
void print_id (Puser v, int n)
{
        for (int i = 0; i <n; i ++)
                cout << v [i] .name << '\ t'
                                << v [i] .id << '\ t'
                                << v [i] .dept << '\ n';
}
```

To be able to sort, you must first define suitable comparison functions. The comparison function must return a negative number if its first parameter is less than the second, zero if they are equal, and a positive number otherwise:

```
int cmp1 (const void * p, const void * q)
        // compare strings containing names
{
        return strcmp (Puser (p) -> name, Puser (q) -> name);
}
int cmp2 (const void * p, const void * q)
        // compare section numbers
{
        return Puser (p) -> dept - Puser (q) -> dept;
}
```

The following program sorts and prints the result:

```
int main ()
{
        sort (heads, 6, sizeof (user), cmp1);
        print_id (heads, 6); // In alphabet order
        cout << "\ n";
        sort (heads, 6, sizeof (user), cmp2);
        print_id (heads, 6); // by department numbers
}
```

The operation of taking the address is allowed for both the substitution function and the overloaded function ($$ R.13.3).

Note that the implicit conversion of a pointer to something to a pointer of type void * is not performed for a parameter of a function called through a pointer to it. Therefore the function

```
int cmp3 (const m ytype *, const mytype *);
```

cannot be used as a parameter to sort (). By doing otherwise, we violate the condition specified in the description that cmp3 () must be called with parameters of type mytype *. If you specifically want to violate this condition, you must use explicit type conversion.

4.7 Macro tools

Language macros are defined in $$ R.16. They play a much smaller role in C ++ than in C. One might even give this advice: use macros only when you cannot do without them. Generally speaking, virtually every occurrence of a macro name is considered to be evidence of some flaw in the language, program, or programmer. Macro tools pose a challenge for the system utilities because they process the code before translation. Therefore, if your program uses macro tools, then the service provided by programs such as the debugger, profiler, cross-reference program will be incomplete for it. If you do decide to use macros, then first carefully study the description of the C ++ preprocessor in your reference manual and do not try to be too smart.

A simple macro definition looks like:

```
#define name remainder-line
```

In the text of the program, the lexeme name is replaced with the remainder of the line. For example,

```
object = name
```

will be replaced by

```
object = remainder-line
```

A macro definition can have parameters. For example:

```
#define mac (a, b) argument1: a argument2: b
```

The mac call must be given two strings representing the parameters. When substituted, they replace a and b in mac (). Therefore the line

```
expanded = mac (foo bar, yuk yuk)
```

when substituted is converted to

```
expanded = argument1: foo bar argument2: yuk yuk
```

Macro names cannot be overloaded. Recursive macro calls are too complex a task for the preprocessor:

```
// mistake:
#define print (a, b) cout << (a) << (b)
#define print (a, b, c) cout << (a) << (b) << (c)

// too hard:
#define fac (n) (n> 1)? n * fac (n-1): 1
```

The preprocessor works with strings and knows almost nothing about C ++ syntax, language types, and scopes. The translator deals only with an already expanded macro definition, so an error in it can be diagnosed after substitution, and not when defining a macro name. This results in rather confusing error messages.

The following macros are allowed:

```
#define Case break; case
#define forever for (;;)
```

And here are completely superfluous macros:

```
#define PI 3.141593
#define BEGIN {
#define END}
```

The following macros can lead to errors:

```
#define SQUARE (a) a * a
#define INCR_xx (xx) ++
#define DISP = 4
```

To verify this, it is enough to try to make a substitution in this example:

```
int xx = 0; // global counter
void f () {
int xx = 0; // local variable
xx = SQUARE (xx + 2); // xx = xx + 2 * xx + 2;
INCR_xx; // increment local variable xx
if (a-DISP == b) { // a- = 4 == b
            // ...
            }
}
```

When referring to global names in macros, use the scope resolution operation ($$ 2.1.1), and enclose the name of the macro parameter in parentheses whenever possible . For example:

```
#define MIN (a, b) (((a) <(b))? (a) :( b))
```

If a macro definition is complex enough, and a comment is required on it, then it is more reasonable to write a comment of the form / * * /, since the C ++ implementation can use the C preprocessor, which does not recognize comments like //. For example:

```
#define m2 (a) something (a) / * thoughtful comment * /
```

Using macro tools, you can create your own language, however, most likely, it will not be understood by others. In addition, the C preprocessor provides fairly weak macro facilities. If your problem is not trivial, you will most likely find that it is either impossible or extremely difficult to solve with these tools. As an alternative to the traditional use of macros, const, inline and type templates have been introduced into the language . For example:

```
const int answer = 42;
template <class T>
inline T min (T a, T b) {return (a <b)? a: b; }
```

4.8 Exercises

1. (* 1) Make the following descriptions: a function with parameters of type pointer to symbol and reference to integer, which does not return values; a pointer to such a function; a function with a parameter of the type of such a pointer; a function that returns such a pointer. Write a definition of a function whose parameter and return value are of the type of such a pointer. Hint: use typedef.
2. (* 1) How to understand the following description? Where can it come in handy?

   ```
   typedef int (rifii &) (int, int);
   ```
3. (* 1.5) Write a program like the one that produces "Hello, world". It takes name as a command line parameter and outputs "Hello, name". Modify the program so that it receives an arbitrary number of names and displays its greeting to all of them: "Hello, ...".
4. (1.5) Write a program that, taking an arbitrary number of filenames from the command line , rewrites all these files one by one in cout. Since the program concatenates files, you can call it cat for concatenation).
5. (* 2) Translate a small program from C to C ++. Modify the header files so that they contain descriptions of all called functions and descriptions of the types of all parameters. If possible, replace all #define commands with enum, const or inline constructs . Remove all external descriptions from the .c files, and change the function definitions to the form corresponding to C ++. Replace calls malloc

() and free () with new and delete operations. Remove unnecessary casting operations .

6. (* 2) Write a sort () function ($$ 4.6.9) that uses a more efficient sorting algorithm.

7. (* 2) Look at the definition of tnode structure in $$ R.9.3. Write a function to store new words in the tnode tree. Write a function to display the nodes of the tnode tree. Write a function that produces such output in alphabetical order. Modify the tnode structure so that it only contains a pointer to an arbitrary length word that is allocated with new in free memory. Modify the function to work with the new tnode structure.

8. (* 1) Write the itoa () function used in the example in $$ 4.6.8.

9. (* 2) Find out what standard header files are on your system. Dig in the / usr / include or / usr / include / CC directories (or whatever directories your system 's standard headers are stored in). Read any file you find interesting .

10. (* 2) Write a function that will flip a two-dimensional array. (The first element of the array will be the last).

11. (* 2) Write an encryption program that reads characters from cin and writes them encrypted to cout. You can use the following simple encryption method: for the character s, the encrypted representation is obtained by the operation s ^ key [i], where key is an array of characters passed on the command line. The characters from the key array are used in a circular manner until the entire input stream has been read. The original text is obtained by repeated application of the same operation with the same key elements. If the key array is not specified (or an empty string is specified), no encryption occurs.

12. (* 3) Write a program that helps to decrypt text encrypted in the above manner when the key (i.e. the array key) is unknown. Hint: See D Kahn "The Codebreakers", Macmillan, 1967, New York, pp. 207-213.

13. (* 3) Write an error handling function, the first parameter is similar to the formatting parameter string printf () and contains the formats % s,% c and% d. It can be followed by an arbitrary number of numeric parameters. Don't use the printf () function. If you don't

know the meaning of % s and other formats, see $$ 10.6. Use <stdarg.h>.

14. (* 1) What name would you choose for the types of function pointers that are defined with a typedef?

15. (* 2) Explore different programs to get an idea of the different naming styles used in practice. How are capital letters used? How is the underline used? When are names like i or x used?

16. (* 1) What errors are contained in the following macros?

```
#define PI = 3.141593;
#define MAX (a, b) a> b? a: b
#define fac (a) (a) * fac ((a) -1)
```

17. (* 3) Write a macro processor with simple capabilities, like the C preprocessor. Read the text from cin, and write the result to cout. Implement macros with no parameters first. Hint: The calculator program has a table of names and a parser that you can use.

18. (* 2) Write a program that extracts the square root of two (2) using the standard sqrt () function, but do not include <math.h> in the program . Do this exercise with the sqrt () function in Fortran.

19. (* 2) Implement the print () function from $$ 4.6.7.

5. CLASSES

"These types are not abstract, they are as real as int and float."
- Doug McIlroy

This chapter describes the possibilities of defining new types for which access to data is limited to a given set of functions that perform it. Explains how you can use the members of a data structure, how to protect it, initialize it, and finally destroy it. The examples provide simple classes for managing the name table, working with the stack, multitude, and implementing discriminatory (i.e., reliable) joins. The next three chapters round out C ++ 's ability to construct new types, and they contain more interesting examples.

5.1 Introduction and overview

The concept of a class, which this chapter and the next three chapters are devoted to, serves in C ++ to give the programmer a tool for constructing new types. They are no less convenient to use than the built-in ones. Ideally, using a user-defined type should be the same as using built-in types. Differences are possible only in the way of construction.

A type is a very concrete representation of a certain concept. For example, in C ++, the type is float with operations +, -, *, etc. is, although limited, but a specific representation of the mathematical concept of a real number. A new type is created in order to become a special and concrete representation of a concept that does not find direct and natural reflection among the built-in types. For example, in a telephony program, you might enter the type trunk_module, in a video game, type explosion, and in a text-processing program, type list_of_paragraphs (list-of-paragraphs). It is usually easier to understand and modify a program in which the types represent the concepts used in the problem well. A well-chosen set of user-defined types makes the program clearer. It allows the translator to detect invalid object use that would otherwise go undetected until the program is debugged.

The key to defining a new type is to separate non-essential implementation details (for example, the location of data in an object of a new type) from those of its characteristics that are essential for its correct use (for example, a complete list of functions that have access to data). This separation is ensured by the fact that all work with the data structure and internal, service operations on it are available only through a special interface (through "one throat").

The chapter is divided into four parts:

$ 5.2 Classes and Members. This introduces the basic concept of a custom type called a class. Access to class objects can be limited by many functions, descriptions of which are included in the class description. These functions are called member and friend functions. Special member functions called constructors are used to create objects of a class . You can define a special member function to delete objects of a class when it is destroyed. This function is called a destructor.

$ 5.3 Interfaces and implementations. Here are two examples of designing, implementing, and using classes.

$ 5.4 Additional properties of classes. There are many more details about the classes here. It is shown how a non-member function can provide access to its private part. Such a function is called a friend of the class. The concepts of static class members and pointers to class members are introduced . It also shows you how to define a discriminatory union.

$ 5.5 Constructors and destructors. The object can be created as automatic, static, or as an object in free memory. In addition, an object can be a member of some aggregate (array or other class), which can also be placed in one of these three ways. Explains in detail the use of constructors and destructors, describes the use of user-defined allocation functions in free memory and free memory functions.

5.2 Classes and members

A class is a custom type. This section introduces the basic means of defining a class, creating its objects, working with such objects, and finally deleting those objects after use.

5.2.1 Member functions

Let's see how you can represent the concept of a date in the language using a structure type and a set of functions that work with variables of this type:

```
struct date {int month, day, year; };
```

```
            date today;
            void set_date (date *, int, int, int);
            void next_date (date *);
            void print_date (const date *);
            // ...
```
There is no explicit connection between functions and the date structure. It can be set by describing functions as members of a structure:

```
struct date {
            int month, day, year;
            void set (int, int, int);
            void get (int *, int * int *);
            void next ();
            void print ();
};
```
Functions described in this way are called member functions. They can be called only through variables of the corresponding type using the standard structure member call notation:

```
date today;
date my_birthday;
void f ()
{
            my_birthday.set (30,12,1950);
            today.set (18,1,1991);
            my_birthday.p rint ();
            today.next ();
}
```
Since different structures can have member functions with the same name, when defining a member function, you must specify the name of the structure:

```
void date :: next ()
{
            if (++ day> 28) {
                        // difficult option here
            }
}
```

Within the body of a member function, member names can be used without specifying an object name. In this case, the name refers to the member of the object for which the function was called.

5.2.2 Classes

We have defined several functions for working with the date structure, but it does not follow from its description that these are the only functions that provide access to objects of type date. You can set such a restriction by declaring a class instead of a structure:

```
class date {
        int month, day, year;
        public:
                void set (int, int, int);
                void get (int *, int *, int *);
                void next ();
                void print ()
};
```

The public keyword breaks the class description into two parts. The names described in the first private part of the class can only be used in member functions. The second - the general part - is an interface with class objects. Therefore, a structure is a class in which, by definition, all members are common. The member functions of a class are defined and used in the same way as shown in the previous section:

```
void date :: print () // print date in US format
{
        cout << month << '/' << day << '/' << year;
}
```

However, the private members of the date class are already shielded from non-member functions:

```
void backdate ()
{
        today.day--; // mistake
}
```

There are several advantages to restricting access to the data structure to an explicitly specified list of functions. Any error in the date (for example, December, 36, 1985) could only have been introduced by a member function, so the first stage of debugging - error localization - occurs even before the first program run. This is just a special case of the general rule: any change in the behavior of the date type can and should be caused by changes in its members. Another advantage is that a potential user of a class only needs to know the definitions of member functions to work with it .

Protection of private data is based only on restricting the use of class member names. Therefore, it can be circumvented by manipulating addresses or explicit type conversions, but this can already be considered a fraud.

5.2.3 Link to yourself

In a member function, you can directly use the member names of the object for which it was called:

```
class X {
            int m;
            public:
                        int readm () {return m; }
};
void f (X aa, X bb)
{
            int a = aa.readm ();
            int b = bb.readm ();
            // ...
}
```

On the first call to readm (), m stands for aa.m, and on the second call, bb.m.

A member function has an additional hidden parameter that is a pointer to the object on which the function was called. You can explicitly use this hidden parameter under the name this. It is believed that in each member function of class X, the this pointer is implicitly described as

```
X * const this;
```

and is initialized to point to the object for which the member function was called. This pointer cannot be changed because it is constant (* const). It is also impossible to describe it explicitly, since this is a service word. An equivalent description of class X can be given:

```
class X {
```

```
            int m;
            public:
                        int readm () {return this-> m; }
   };
```

It is unnecessary to use this to refer to members. This is mainly used in member functions that directly work with pointers. A typical example is a function that inserts an item into a double-linked list:

```
   class dlink {
            dlink * pre; // pointer to the previous element
            dlink * suc; // pointer to next element
            public:
                        void append (dlink *);
                        // ...
   };
   void dlink :: append (dlink * p)
   {
            p-> suc = suc; // i.e. p-> suc = this-> suc
            p-> pre = this; // explicit use of "this"
            suc-> pre = p; // i.e. this-> suc-> pre = p
            suc = p; // i.e. this-> suc = p
   }
   dlink * list_head;
   void f (dlink * a, dlink * b)
   {
            // ...
            list_head-> append (a);
            list_head-> append (b);
   }
```

Lists with this general structure are the foundation of the list classes described in Chapter 8. To attach a rung to a list, you need to change the objects pointed to by this, pre, and suc pointers. They are all of type dlink, so the dlink :: append () member function can access them. The protected unit in C ++ is a class, not an individual object of the class.

You can describe a member function in such a way that the object for which it is called will be read-only. The fact that the function will not change the object for which it is called (i.e. this *) is indicated by the const at the end of the parameter list :

```
class X {
            int m;
            public:
                        readme () const {return m; }
                        writeme (int i) {m = i; }
};
```

A member function with the const specification can be called on constant objects, but a member function without such a specification cannot:

```
void f (X & mutable, const X & constant)
{
            mutable.readme (); // fine
            mutable.writeme (7); // fine
            constant.readme (); // fine
            constant.writeme (7); // mistake
}
```

In this example, a sane translator could detect that the X :: writeme () function is trying to modify a persistent object. However, this is not an easy task for a translator. Because of the split translation, it generally cannot guarantee the "persistence" of an object if there is no corresponding declaration with the const specification . For example, the readme () and writeme () definitions could have been in another file:

```
class X {
            int m;
            public:
                        readme () const;
                        writeme (int i);
};
```

In such a case, the description of readme () with a const specification is essential.

The type of the this pointer in a constant member function of class X is const X * const. This means that without an explicit cast using this, you cannot change the value of an object:

```
class X {
```

```
            int m;
            public:
                        // ...
                        void implicit_cheat () const {m ++; } // mistake
                        void explicit_cheat () const {((X *) this) -> m ++; }
                        // fine
   };
```

You can drop the const specification because the notion of "persistence" of an object has two meanings. The first, called "physical persistence", is that the object is stored in write-protected memory. The second, called "logical persistence", is that the object acts as a constant (immutable) in relation to users. An operation on a logically constant object can change part of the object's data , if it does not violate its persistence from the user's point of view. Operations that do not violate the logical constancy of an object can be value buffering, statistics maintenance , changing counter variables in constant member functions.

Logical consistency can be achieved by casting that removes the const specification:

```
   class calculator1 {
            int cache_val;
            int cache_arg;
            // ...
            public:
                        int compute (int i) const;
                        // ...
   };

   int calculator1 :: compute (int i) const
   {
            if (i == cache_arg) return cache _val;
            // not the best way
            ((calculator1 *) this) -> cache_arg = i;
            ((calculator1 *) this) -> cache_val = val;
            return val;
   }
```

The same result can be achieved using a non-const data pointer :

```
struct cache {
        int val;
        int arg;
};

class calculator2 {
        c ache * p;
        // ...
        public:
                int compute (int i) const;
                // ...
        };

int calculator2 :: compute (int i) const
{
        if (i == p-> arg) return p-> val;
        // not the best way
        p-> arg = i;
        p-> val = val;
return val;
}
```

Note that const must be specified both in the description and in the definition of a constant member function. Physical persistence is ensured by placing the object in write-protected memory only if the class does not have a constructor ($$ 7.1.6).

5.2.4 Initialization

Initializing class objects using functions such as set_date () is inelegant and error-prone. Since it was not explicitly stated that the object needs to be initialized, the programmer can either forget to do it or do it twice, which can lead to equally disastrous consequences. It is better to give the programmer the opportunity to describe a function that is explicitly intended to initialize objects. Since such a function constructs a value of a given type, it is called a constructor. This function is easy to recognize - it has the same name as its class:

```
cla ss date {
        // ...
```

```
            date (int, int, int);
    };
```

If the class has a constructor, all objects of this class will be initialized. If the constructor requires parameters, they must be specified:

```
    date today = date (23,6,1983);
    date xmas (25,12,0); // short form
    date my_birthday; // wrong, needs an initializer
```

It is often convenient to specify several ways to initialize an object. To do this, you need to describe several constructors:

```
    class date {
            int month, day, year;
            public:
                    // ...
                    date (int, int, int); // day month Year
                    date (int, int); // day, month and current year
                    date (int); // day and current year and month
                    date (); // default value: current date
            date (const char *); // date in string representation
    };
```

Constructor parameters follow the same parameter type rules as all other functions ($$ 4.6.6). While constructors differ enough in the types of their parameters, the translator is able to choose the right constructor:

```
    date today (4);
    date july4 ("July 4, 1983");
    date guy ("5 Nov");
    date now; // initialization with default value
```

Reproduction of constructors in the example c date is typical. When developing a class, there is always a temptation to add one more feature, in case it may be useful to someone. It takes some thought to determine what you really need, but the result is usually a more compact and understandable program. You can reduce the number of similar functions by using a default parameter value. In the example with date for each parameter, you can set a standard value, which means: "take the value from the current date".

```
    class date {
            int month, day, year;
            public:
                    // ...
                    date (int d = 0, int m = 0, y = 0);
                    // ...
    };

    date :: date (int d, int m, int y)
    {
            day = d? d: today.day;
            month = m? m: today.month;
            year = y? y: toda y.year;
            // check if the date is correct
            // ...
    }
```

When a default parameter value is used, it must be different from all valid parameter values. In the case of month and day, it is obvious that a value of zero is true, but it is not obvious that zero is suitable for a value of a year. Fortunately, there is no year zero in the European calendar. immediately after 1 BC. (year == - 1) is 1 year of birth. (year == 1). However, for a normal program, this may be too subtle.

An object of a class without a constructor can be initialized by assigning another object of the same class to it. This is not prohibited even when constructors are described:

```
    date d = today; // initialization by assignment
```

In fact, there is a standard copy constructor, defined as element-wise copying of objects of the same class. If such a constructor is not needed for class X, you can override it as a copy constructor X :: X (const X &). We'll talk more about this in $$ 7.6.

5.2.5 Deletion

User-defined types are more likely than not to have constructors that initialize appropriately. For many types, an inverse operation is also required — a destructor to ensure that objects of that type are properly deleted. The destructor of class X is denoted ~ X ("constructor complement"). In particular, for many classes free memory is used (see $$

3.2.6), allocated by the constructor and released by the destructor. For example, here is the traditional definition of the stack type, from which error handling has been completely thrown out for brevity:

```
class char_stack {
            int size;
            char * top;
            char * s;
            public:
                        char_stack (int sz) {top = s = new char [size = sz]; }
                        ~ char_stack () {delete [] s; } // destructor
                        void push (char c) {* top ++ = c; }
                        void pop () {return * - top; }
};
```

When an object of type char_stack goes out of its current scope, the destructor is called:

```
void f ()
{
            char_stack s1 (100);
            char_stack s2 (200);
            s1.push ('a');
            s2.push (s1.pop ());
            char ch = s2.pop ();
            cout << ch << '\ n';
}
```

When f () starts executing, the char_stack constructor is called, which allocates an array of 100 characters s1 and an array of 200 characters s2. When f () returns, the memory that was occupied by both arrays will be freed.

5.2.6 Substitution

Programming with classes implies that many small functions will appear in the program. In fact, wherever a routine reference to a data structure would be in a program with a traditional organization, a function is used. What was the convention became the standard checked by the translator. As a result, the program can become extremely ineffective. Although a function call in C ++ is not that expensive compared to other languages, it still costs a lot more than a couple of memory accesses that make up the body of a trivial function.

Inline functions help overcome this difficulty. If a member function is defined in the class description, and not just described, then it is considered a substitution. This means, for example, that when translating functions using char_stack from the previous example, no function call operations will be used, except for the implementation of output operations! In other words, when designing such a class, you don't need to take into account the cost of calling functions. Any, even the smallest, action can be safely defined as a function without losing efficiency. This remark removes the most commonly used argument in favor of common data members.

A member function can be described with the inline specification and outside the class description:

```
class char_stack {
        int size;
        char * top;
        char * s;
        public:
                char pop ();
                // ...
};

inline char char_stack :: pop ()
{
        ret urn * - top;
}
```

Note that it is inadmissible to describe different definitions of a substitution member function in different source files ($$ R.7.1.2). This would violate the concept of a class as a whole type.

5.3 Interfaces and implementations

What constitutes a good class? It is something that has a well-defined set of operations. Something considered as a "black box" that can only be controlled through these operations. Something whose actual representation can be changed in any conceivable way, but without changing the way the operations are used. Something that may need multiple copies.

Obvious examples of good classes are given by containers of different kinds: tables, sets, lists, vectors, dictionaries, etc. Such a class has an operation of entering into a container. Usually there is also a check operation: was this member entered into the container? There can be operations to order all members and view them in a specific order. Finally, there can be a member removal operation. Usually container classes have constructors and destructors.

5.3.1 Alternative implementations

As long as the description of the general part of the class and member functions remains unchanged, you can change its implementation without affecting the users of the class. In support of this, consider the table of names from the calculator program given in Chapter 3. Its structure is as follows:

```
struct name {
        char * string;
        name * next;
        double value;
};
```

And here is a variant of the table class (table of names):

```
// file table.h
class table {
        name * tbl;
        public:
                table () {tbl = 0; }
                name * look (char *, i nt = 0);
                name * insert (char * s) {return look (s, 1); }
};
```

This table differs from the one defined in Chapter 3 in that it is a real type. You can describe several tables, create a pointer to a table, etc. For example:

```
#include "table.h"
table globals;
table key words;
table * locals;
```

```
main ()
{
            locals = new table;
            // ...
}
```

Here is the implementation of the table :: look () function, which uses a linear search in the list of table names:

```
#include <string.h>

name * table :: look (char * p, int ins)
{
            for (name * n = tbl; n; n = n-> n ext)
                        if (strcmp (p, n-> string) == 0) return n;
            if (ins == 0) error ("name not found");
            name * nn = new name;
            nn-> string = new char [strlen (p) +1];
            strcpy (nn-> string, p);
            nn-> value = 1;
            nn-> next = tbl;
            tbl = nn;
            return nn;
}
```

Now we will improve the class table so that the search for the name goes by the key (hash functions from the name), as it was done in the example with the calculator. This is harder to do if you adhere to the constraint that not all programs using the given version of the table class need to be changed:

```
class table {
            name ** tbl;
            int size;
            public:
                        table (int sz = 15);
                        ~ table ();
                        name * look (char *, int = 0);
            name * insert (char * s) {return look (s, 1); }
};
```

The changes to the data structure and the constructor occurred because the table must be of a certain size in order to be hashed. Setting the constructor with a default parameter value ensures that old programs that did not use the table size remain correct. The default parameter values are useful when you want to change the class without affecting the programs of the users of the class. Now the constructor and destructor create and destroy the hashed tables:

```
table :: table (int sz)
{
        if (sz <0) error ("table size is negative");
        tbl = new name * [size = sz];
        for (int i = 0 ; i <sz; i ++) tbl [i] = 0;
}

table :: ~ table ()
{
        for (int i = 0; i <size; i ++) {
                name * nx;
                for (name * n = tbl [i]; n; n = nx) {
                        nx = n-> next;
                        delete n-> string;
                        delete n;
                }
        }
        delete tbl;
}
```

By describing a destructor for the name class, you can get a clearer and simpler version of table :: ~ table (). The search function is almost the same as in the example calculator ($$ 3.13):

```
name * table :: look (const char * p, int ins)
{
        int ii = 0;
        char * pp = p;
        while (* pp) ii = ii << 1 ^ * pp ++;
        if (ii <0) ii = -ii;
        ii % = size;
```

```
            for (name * n = tbl [ii]; n; n = n-> next)
                    if (strcmp (p, n-> string) == 0) return n;
            name * nn = new name;
            nn-> string = new char [strlen (p) +1];
            strcpy (nn-> string, p);
            nn-> value = 1;
            nn-> next = tbl [ii];
            tbl [ii] = nn;
    return nn;
    }
```

Obviously, the member functions of a class must be retranslated whenever a change is made to the class description. Ideally, this change should not affect the users of the class in any way. Unfortunately, this is not usually the case. To place a variable of a class type, the translator must know the size of the class object. If the size of the object changes, you need to re-translate the files that used the class. It is possible to write a system program (and it has even been written) that will determine the minimum set of files to be re-translated after the class change. But such a program has not yet become widespread.

A possible question is: why was C ++ designed in such a way that after changing the private part of the class, re-translation of user programs is required? Why is the private part of the class even present in the class description? In other words, why are the descriptions of private members present in the header files available to the user, if they are still not available for him in the program? There is only one answer - efficiency. In many programming systems, the translation process and the sequence of instructions that make the function call will be easier if the size of the automatic (i.e., placed on the stack) objects is known at the translation stage.

You might not know the definition of the entire class if you think of each object as a pointer to a "real" object. This allows you to solve the problem, since all pointers will have the same size, and the placement of real objects will be carried out only in one file, in which the private parts of the classes are available. However, such a solution leads to additional memory consumption for each object and additional memory access each time the member is used. Even worse, every function call with an automatic class object requires calls to allocate and deallocate memory functions. In addition, it becomes impossible to implement by substitution of member functions that work with private members of the class. Finally, such a change will make it impossible to link C ++ and C programs, since the C translator will handle struct structures differently. Therefore, this solution was deemed unacceptable for C ++.

On the other hand, C ++ provides a facility for creating abstract types in which the relationship between the user interface and the implementation is rather weak. Chapter 6 introduces derived classes and describes abstract base classes, and $$ 13.3 explains how to implement abstract types using these tools. The purpose of this is to make it possible to define custom types as efficiently as possible.

and specific, as well as standard, and provide a basic means of defining more flexible options for types that may not be as effective.

5.3.2 Completed class example

Programming without data hiding (per structure) requires less prior thought than programming with data hiding (per class). The structure can be defined without much thought about how it will be used. When a class is defined, the focus is on providing a complete set of operations for the new type. This is an important shift in focus in software design. Usually, the time spent on developing a new type pays off many times in the process of debugging and developing a program.

Here is an example of a complete definition of type intset, representing the concept of "set of integers":

```
class intset {
        int cursize, maxsize;
        int * x;
        public:
                intset (int m, int n); // no more than m integers from 1..n
                ~ intset ();
                int member (int t) const; // is t a member?
                void insert (int t); // add to set t
                void start (int & i) const {i = 0; }
                void ok (int & i) const {return i <cursize; }
                void next (int & i) const {return x [i ++]; }
};
```

To test this class, first create and then print out a bunch of random integers. This simple set of integers can be used to check if there are repetitions in their sequence. But for most problems, of course, a more developed type of set is needed . As always, errors are possible, so we need a function:

```
#include <iostream.h>

v oid error (const char * s)
{
        cerr << "set:" << s << '\ n';
        exit (1);
}
```

The intset class is used in the main () function, which must be given two parameters: the first determines the number of random numbers generated , and the second determines the range of their values:

```
int main (i nt argc, char * argv [])
{
        if (argc! = 3) error ("two parameters must be specified");
        int count = 0;
        int m = atoi (argv [1]); // number of elements in the set
        int n = atoi (argv [2]); // from the range 1..n
        intset s (m, n);
        while (count <m) {
                int t = randint (n);
                if (s.member (t) == 0) {
                        s.insert (t);
                        count ++;
                }
        }
        print_in_order (& s);
}
```

The argc program parameter counter is 3, although the program has only two parameters. The point is that argv [0] is always passed an additional parameter containing the name of the program. Function

```
extern "C" int atoi (const char *)
```

is a standard library function that converts an integer from a string representation to an internal binary form. As usual, if you do not want to have such a description in your program, then you need to include in it the appropriate header file containing descriptions of standard library functions. Random numbers are generated using the standard rand function:

```
extern "C" int rand (); // be careful:
                                                // numbers are not
entirely random
int randint (int u) // range 1..u
{
        int r = rand ();
        if (r <0) r = -r;
        return 1 + r% u;
}
```

The implementation details of the class are of little interest to the user, but member functions will be used anyway. The constructor allocates an array of integers with a size equal to the specified maximum set size, and the destructor deletes this array:

```
intset :: intset (int m, int n) // no more than m integers in 1..n
{
        if (m <1 || n <m) error ("invalid intset size");
        cursize = 0;
        maxsize = m;
        x = new int [maxsize];
}

intse t :: ~ intset ()
{
        delete x;
}
```

Integers are added in such a way that they are stored in a set in ascending order:

```
void intset :: insert (int t)
{
        if (++ cursize> maxsize) error ("too many elements");
        int i = cursize-1;
```

```
            x [i] = t;
            while (i> 0 && x [i-1] > x [i]) {
                        int t = x [i]; // swap x [i] and x [i-1]
                        x [i] = x [i-1];
                        x [i-1] = t;
                        i--;
            }
}
```

To find an element, a simple binary search is used:

```
int intset :: member (int t) const // binary search
{
            int l = 0;
            int u = cursize-1;
            while (l <= u) {
                        int m = (l + u) / 2;
                        if (t <x [m])
                                    u = m-1;
                        else if (t> x [m])
                                    l = m + 1;
                        else
                                    return 1; // found
            }
            return 0; // not found
}
```

Finally, you need to provide the user with a set of operations with which he could iterate over the set in some order (after all, the order used in the intset view is hidden from him). A set is inherently not internally ordered, and you cannot just allow to select the elements of an array (what if tomorrow intset will be implemented as a linked list?).

The user receives three functions: start () to initialize the iteration, ok () to check if there is a next element, and next () to get the next element:

```
class intset {
            // ...
            void start (int & i) const { i = 0; }
            int ok (int & i) const {return i <cursize; }
            int next (int & i) const {return x [i ++]; }
};
```

To ensure that these three operations work together, you need to remember the element where the iteration stopped. To do this, the user must specify an integer parameter. Since our representation of the set is ordered, the implementation of these operations is trivial. Now we can define the print_in_order function:

```
void print_in_order (intset * set)
{
        int var;
        set-> sart (var);
        while (set-> ok (var)) cout << set-> next (var) << '\ n';
}
```

Another way to construct an iterator over a set is shown in $$ 7.8.

5.4 More about classes

This section describes the additional properties of the class. Described a way to provide access to private members in non-member functions ($$ 5.4.1). It describes how to resolve collisions of member names ($$ 5.4.2) and how to nest class descriptions ($$ 5.4.3), but avoid unwanted nesting ($$ 5.4.4). The concept of static members is introduced, which are used to represent operations and data related to the class itself, and not to its individual objects ($$ 5.4.5). The section ends with an example showing how a discriminatory (reliable) join ($$ 5.4.6) can be built.

5.4.1 Friends

Let two classes be defined: vector (vector) and matrix (matrix). Each of them hides its own view, but provides a complete set of operations for working with objects of its type. Let's say you need to define a function that multiplies a matrix by a vector. For simplicity, let's assume that the vector has four elements with indices from 0 to 3, and the matrix has four vectors also with indices from 0 to 3. Access to the elements of the vector is provided by the elem () function, and there is a similar function for the matrix. You can define a global multiply function like this:

```
vector multiply (const matrix & m, const vector & v);
{
        vector r;
```

```
            for (int i = 0; i <3; i ++) {// r [i] = m [i] * v;
                        r.elem (i) = 0;
                        for (int j = 0; j <3; j ++)
                                    r.elem (i) + = m.elem (i, j) * v.elem (j);
            }
            return r;
   }
```

This is a natural solution, but it can be very ineffective. Each time you call multiply (), the elem () function will be called 4 * (1 + 4 * 3) times. If elem () does real control of array boundaries, then such control will take much more time than the execution of the function itself, and as a result it will be unusable for users. On the other hand, if elem () is some kind of special access option without control, then by doing so we litter the interface with the vector and matrix with a special access function that is needed only to bypass control.

If we could make multiply a member of both vector and matrix, we could do without index control when accessing an element of the matrix, but at the same time not introduce the special function elem (). However, a function cannot be a member of two classes. The language needs to be able to provide a non-member function with access to private members of the class. A function that is not a member of a class - that has access to its private part is called a friend of this class. A function can become a friend of a class if it is described as friend in its description. For example:

```
   class matrix;
   class vector {
            float v [4];
            // ...
            friend vector multiply (const matrix &, const vector &);
   };
   class matrix {
            vector v [4];
            // ...
            friend vector multiply (const matrix &, co nst vector &);
   };
```

The friend function has no special features, except for the right to access the private part of the class. In particular, you cannot use the this pointer in such a function unless it really is a member of the class. The friend description is the real description. It injects the name of a function into the scope of the class in which it was declared, and it does the usual checks for other definitions of the same name in that scope. The friend description can be found in both the public and private parts of the class, it doesn't matter.

Now we can write the multiply function using the elements of the vector and matrix directly:

```
vector multiply (const matrix & m, const vector & v)
{
        vector r;
        for (int i = 0; i <3; i ++) {// r [i] = m [ i] * v;
                 rv [i] = 0;
                for (int j = 0; j <3; j ++)
                        rv [i] + = mv [i] [j] * vv [j];
        }
        return r;
}
```

Note that, like a member function, a friend function is explicitly described in the description of the friend class. Therefore, it is an integral part of the class interface along with the member function.

A member function of one class can be a friend of another class:

```
class x {
        // ...
        void f ();
};

class y {
        // ...
        friend void x :: f ();
};
```

It is possible that all functions of one class are friends of another class. There is a short form for this:

```
class x {
        friend class y;
        // ...
};
```

As a result of this description, all member functions of y become friends of class x.

5.4.2 Clarifying Member Name

It is sometimes useful to make an explicit distinction between class member names and other names. To do this, use the :: (scope permissions) operation:

```
class X {
        int m;
        public:
                int readm () const {return m; }
                void setm (int m) {X :: m = m; }
};
```

In X :: setm (), the m parameter hides the m member, so the member can only be accessed using the qualified name X :: m. The right operand of the :: operator must be a class name.

Name starting with :: must be a global name. This is especially useful when using such common names as read, put, open, which can be used to denote member functions without losing the ability to denote non-member functions with them. For example:

```
class my_file {
        // ...
        public:
                int open (const char *, const char *);
};

int my_file :: jpen (const char * name, const char * spec)
{
        // ...
        if (:: open (name, flag)) {// used by open () from UNIX (2)
                // ...
        }
        // ...
}
```

5.4.3 Nested classes

The class description can be nested. For example:

```
class set {
  struct setmem {
                int mem;
                setmem * next;
                setmem (int m, setmem * n) {mem = m; next = n; }
        };
        setmem * first;
        public:
                set () {first = 0; }
                insert (int m) {first = new setmem (m, first); }
                // ...
};
```

The nested class's accessibility is limited by the scope of the lexically enclosing class:

```
setmem m1 (1,0); // error: setmem is not found
                                              // in the global scope
```

If only the description of a nested class is not very simple, then it is better to describe this class separately, since nested descriptions can become very confusing:

```
class setmem {
        friend class set; // only available for set members
        int mem;
        setmem * next;
        setmem (int m, setmem * n) {mem = m; next = n; }
        // many other useful members
};

class set {
        setmem * first;
        public:
                set () {first = 0; }
                insert (int m) {first = new setmem (m, first); }
                // ...
};
```

A useful property of nesting is to reduce the number of globals, but the disadvantage is that it violates the freedom to use nested types (see $$ 12.3).

The name of a member class (nested class) can be used outside of the enclosing class description in the same way as the name of any other member:

```
class X {
            struct M1 {int m; };
            public:
                        struct M2 {int m; };
                        M1 f (M2);
};

void f ()
{M1 a; // error: name `M1 'out of scope
            M2 b; // error: name `M1 'out of scope
            X :: M1 c; // error: X :: M1 private member
            X :: M2 d; // fine
}
```

Note that access control also occurs for nested class names.

In a member function, the class scope begins after the X :: qualifier and extends to the end of the function declaration. For example:

```
M1 X :: f (M2 a) // error: name `M1 'out of scope
{/ * ... * /}

X :: M1 X :: f (M2 a) // normal
{/ * ... * /}

X :: M1 X :: f (X :: M2 a) // ok, but the third refinement X :: is
unnecessary
{/ * ... * /}
```

5.4.4 Static members

A class is a type, not some data, and a copy of the members representing the data is created for each class object. However, the most successful implementation of some types requires that all objects of this type have some common data. It is better if this data can be described as part of a class. For example, in operating systems or when modeling task management, a list of tasks is often needed:

```
class task {
        // ...
        static task * chain;
        // ...
};
```

By describing the chain member as static, we guarantee that it will be created in the singular, i.e. will not be created for every task object. But it is in the scope of the task class, and can be accessed outside of that scope if only described in the general part. In this case, the member name must be qualified with the class name:

```
if (task :: chain == 0) // some statements
```

In a member function, it can be simply referred to as chain. Using static class members can dramatically reduce the need for global variables.

By describing a member as static, we limit its scope and make it independent of the individual objects of its class. This property is useful for both member functions and members representing data:

```
class task {
        // ...
        static task * task_chain;
        static void shedule (int);
        // ...
};
```

But the description of a static member is only a description, and somewhere in the program there should be a single definition for the object or function being described, for example, this:

```
task * task :: task_chain = 0;
void task :: shedule (int p) {/ * ... * /}
```

Naturally, private members can be defined in the same way.

Note that the static keyword is not needed or even used in the definition of a static member of a class. If it were present, ambiguity would arise: does it indicate that a member of the class is static, or is it used to describe a global object or function?

The word static is one of the most overloaded control words in C and C ++. A static member representing data has both main meanings: "statically allocated", that is. the opposite of objects allocated on the stack or free memory, and "static" in the sense of scoped, ie. the opposite of objects subject to external binding. Only the second static is relevant to member functions.

5.4.5 Member pointers

You can take the address of a class member. The operation of taking the address of a member function is often useful, because the purpose and use of function pointers that we discussed in $$ 4.6.9 apply equally to such functions. A pointer to a member can be obtained by applying the address take operation & to the fully qualified name of a class member, for example, & class_name :: member_name. To describe a variable of type "pointer to a member of class X", you must use a descriptor of the form X :: *. For example:

```
#include <iostream.h>

struct cl
{
        char * val;
        void print (int x) { cout << val << x << '\ n'; }
        cl (char * v) {val = v; }
};
```

A pointer to a member can be described and used like this:

```
typedef void (cl :: * PMFI) (int);
int main ()
{
        cl z1 ("z1");
        cl z2 ("z2");
        cl * p = & z2;
        PMFI pf = & cl :: print;
        z1.print (1);
        (z1. * pf) (2);
        z2.prin t (3);
        (p -> * pf) (4);
}
```

Using a typedef to replace a hard-to-read C descriptor is fairly common. The. * And -> * operations set up a pointer to a specific object, resulting in a function that can be called. The () operation has a higher priority than the. * And -> * operations, so parentheses are needed.

In many cases virtual functions ($$ 6.2.5) successfully replace function pointers.

5.4.6 Structures and unions

By definition, a structure is a class all of whose members are common, i.e. description

```
struct s {...
```

this is just a short form of description

```
class s {public: ...
```

A named union is defined as a structure, all members of which have the same address ($$ R.9.5). If you know that only one member of the structure is used at a time, then declaring it as a union can save memory. For example, you can use concatenation to store C translator tokens:

```
union tok_val {
        char * p; // line
        char v [8]; // identifier (no more than 8 characters)
        long i; // integer values
        double d; // floating point values
};
```

The problem with unions is that the translator generally does not know which member is currently being used, and therefore type control is not possible. For example:

```
void strange (int i)
{
        tok_val x;
         if ( i)
                  xp = "2";
         else
                 xd = 2;
        sqrt (xd); // error if i! = 0
}
```

Also, a union defined this way cannot be initialized in this seemingly natural way:

```
tok_val val1 = 12; // error: int is assigned to tok_val
tok_val val2 = "12"; // error: char * assigned to tok_val
```

For proper initialization, you must use the constructors:

```
union tok_val {
        char * p; // line
        char v [8]; // identifier (no more than 8 characters)
        long i; // integer values
        double d; // floating point values
        tok_val (const char *); // need to choose between p and v
        tok_val (int ii) {i = ii; }
        tok_val (double dd) {d = dd; }
};
```

These descriptions allow you to resolve the ambiguity of function name overloading using member types (see $$ 4.6.6 and $$ 7.3). For example:

```
void f ()
{
        tok_val a = 10; // ai = 10
        tok_val b = 10.0; // bd = 10.0
}
```

If this is not possible (for example, for types char * and char [8] or int and char, etc.), then you can determine which member is initialized by examining the initializer during program execution, or by entering an additional parameter. For example:

```
tok_val :: tok_val (const char * pp)
{
 if (strlen (pp) <= 8)
                strncpy (v, pp, 8); // short string
        else
                p = pp; // long string
}
```

But it is better to avoid such ambiguity.

The standard strncpy () function, like strcpy (), copies strings, but it has an additional parameter that specifies the maximum number of characters to copy.

The fact that constructors are used to initialize a union does not yet guarantee against accidental errors when working with union, when a value of one type is assigned, but a value of another type is selected. Such a guarantee can be obtained by enclosing a union in a class in which the type of the value being inserted will be tracked:

```
class tok_val {
        pu blic:
                enum Tag {I, D, S, N};
        private:
                union {
                        const char * p;
                        char v [8];
                        long i;
                        double d;
                };
Tag tag;

void check (Tag t) {if (tag! = t) error (); }
        public:
                Tag get_tag () {return tag; }
                tok_val (const char * pp);
                tok_val (long ii) {i = ii; tag = I; }
                tok_val (double dd) {d = dd; tag = D; }
                long & ival () {check (I); return i; }
                double & fval () {check (D); return d; }
                const char * & sval () {check (S); return p; }
                char * id () {check (N); return v; }
};

tok _val :: tok_val (const char * pp)
{
        if (strlen (pp) <= 8) { // short string
                tag = N;
                strncpy (v, pp, 8);
        }
        else { // long string
```

```
                tag = S;
                p = pp; // only the pointer is written
            }
    }
```

You can use the tok_val class like this:

```
    void f ()
    {
            tok_val t1 ("short"); // assigned to v
            tok_val t2 ("long string"); // assigned to p
            char s [8];
            strncpy (s, t1.id (), 8); // fine
            strncpy (s, t2.id (), 8); // check () will throw an error
    }
```

By describing the Tag type and the get_tag () function in general, we ensure that the tok_val type can be used as a parameter type. Thus, a type-safe alternative to describing parameters with ellipsis appears. For example, here is a description of the error handling function, which can have one, two, or three parameters of types char *, int or double:

```
    extern tok_val no_arg;
    void error (
            const char * format,
            tok_val a1 = no_arg,
            tok_val a2 = no_arg,
            tok_val a3 = no_arg);
```

5.5 Constructors and destructors

If a class has a constructor, it is called whenever an object of that class is created. If a class has a destructor, it is called whenever an object of that class is destroyed. An object can be created as:

1] automatic, which is created every time its description is encountered during program execution, and is destroyed upon exiting the block in which it is described;

2] static, which is created once at the start of the program and destroyed at its termination;

3] an object in free memory, which is created by the new operation and destroyed by the delete operation;

4] a member object that is created when another class is created or when an array is created of which it is an element .

Besides, an object can be created if its constructor is explicitly used in the expression ($$ 7.3) or as a temporary object ($$ R.12.2). In both cases, such an object has no name. The following subsections assume that objects are of a class with a constructor and destructor. The table class from $$ 5.3.1 is used as an example.

5.5.1 Local variables

The local variable constructor is called every time its description is encountered during program execution. The destructor of a local variable is called every time it exits the block where it was declared. Destructors for local variables are called in the order that constructors are called when they are created:

```
void f (int i)
{
        table aa;
        table bb;
        if (i> 0) {
                table cc;
                // ...
        }
        // ...
}
```

Here aa and bb are created (in that order) every time f () is called, and they are destroyed when f () returns in reverse order - bb, then aa. If i is greater than zero in the current call to f (), then cc is created after bb and destroyed before it.

Since aa and bb are objects of class table, the assignment aa = bb means copying from bb to aa (see $$ 2.3.8). This interpretation of assignment can lead to unexpected (and usually undesirable) results if objects of the class in which the constructor is defined are assigned:

```
void h ()
{
        table t1 (100);
          table t2 = t1; // trouble
        table t3 (200);
        t3 = t2; // trouble
}
```

In this example, the table constructor is called twice: for t1 and t3. It is not called for t2, because this object is assignment-initialized. However, the destructor for table is called three times: for t1, t2, and t3! Further, the standard interpretation of assignment is copying by member, so before exiting from h () t1, t2 and t3 will contain a pointer to an array of names for which memory was allocated in free memory when t1 was created. A pointer to the memory allocated for the array of names when t3 is created will be lost. These troubles can be avoided (see $$ 1.4.2 and $$ 7.6).

5.5.2 Static memory

Consider this example:

```
table tbl (100);
void f (int i)
{
        static table tbl2 (i);
}

int main ()
{
        f (200);
        // ...
}
```

Here, the constructor defined in $$ 5.3.1 will be called twice: once for tbl and once for tbl2. The destructor table :: ~ table () will also be called twice: to destroy tbl and tbl2 upon exit from main (). Constructors of global static objects in the file are called in the same order as they appear in the object description file, and destructors for them are called in the opposite order. The constructor of a local static object is called when the object definition is first encountered during program execution.

Traditionally, executing main () was seen as executing the entire program. In fact, this is not the case even for C. Already placing a static object of a class with a constructor and / or a destructor allows the programmer to specify the actions that will be performed before calling main () and / or after exiting main ().

Calling constructors and destructors on static objects is extremely important in C ++. They can be used to ensure proper initialization and deletion of data structures used in libraries. Consider <iostream.h>. Where do cin, cout and cerr come from? When are they initialized? More important question: Since the output streams use internal character buffers, these buffers are popped, but when? There is a simple and obvious answer: all actions are performed by the corresponding constructors and destructors before running main () and after exiting it (see $$ 10.5.1). There are alternatives to using constructors and destructors to initialize and destroy library data structures, but they are all either very specialized, or clumsy, or both.

If the program terminates by calling the exit () function, then the destructors for all constructed static objects are called. However, if the program ends with an abort () call, this does not happen. Note that exit () does not terminate the program immediately. Calling exit () in the destructor can lead to infinite recursion. If you need a guarantee that both static and automatic objects will be destroyed, you can take advantage of special situations ($$ 9).

Sometimes, when developing a library, it is necessary or simply convenient to create a type with a constructor and a destructor for only one purpose: initialization and destruction of objects. This type is used only once to place a static object in order to invoke constructors and destructors.

5.5.3 Free memory

Let's consider an example:

```
main ()
{
        table * p = new table (100);
        table * q = new table (200);
        delete p;
        delete p; // will probably throw a runtime error
 }
```

The table :: table () constructor will be called twice, just like the table :: ~ table () destructor. But this does not mean anything, since in C ++ it is not guaranteed that the destructor will only be called on the object created by the new operation. In this example, q is not destroyed at all, but p is destroyed twice! Depending on the type of p and q, the programmer may or may not consider this an error. The fact that the object is not deleted is usually not an error, but simply a loss of memory. At the same time, removing p again is a serious mistake. Repeated use of delete on the same pointer can result in an infinite loop in the subroutine that manages free memory. But in the language, the result of repeated deletion is not defined, and it depends on the implementation.

The user can define his own implementation of the new and delete operations (see $$ 3.2.6 and $$ 6.7). In addition, it is possible to establish the interaction of the constructor or destructor with the new and delete operations (see $$ 5.5.6 and $$ 6.7.2). Free memory allocation of arrays is discussed in $$ 5.5.5.

5.5.4 Class Objects as Members

Let's consider an example:

```
class classdef {
        table mem bers;
        int no_of_members;
        // ...
        classdef (int size);
        ~ classdef ();
};
```

The purpose of this definition is obviously for the classdef to contain a member that is a table of size size, but there is a complication: you must ensure that the table :: table () constructor is called with the size parameter. This can be done, for example, like this:

```
classdef :: classdef (int size)
        : members (size)
{
        no_of_members = size;
        // ...
}
```

The parameter for the member constructor (that is, for table :: table ()) is specified in the definition (but not in the description) of the constructor of the class containing the member (that is, in the definition of classdef :: classdef ()). The constructor for a member will be called before the body of the constructor that specifies the parameter list for it is executed.

Similarly, you can set parameters for the constructors of other members (if there are still other members):

```
class classdef {
        table members;
        table friends;
        int no_of_members;
        // ...
        classdef (int size);
        ~ classdef ();
};
```

The parameter lists for members are separated from each other by commas (not colons), and the list of initializers for members can be specified in any order:

```
classdef :: classdef (int size)
        : friends (size), members (size), no_of_members (size)
{
        // ...
}
```

Constructors are called in the order in which they are specified in the class description.

Such constructor declarations are essential for types whose initialization and assignment are different from each other, in other words, for objects that are members of a class with a constructor, for constant members, or for members of a reference type. However, as shown by the no_of_members member in the above example, such constructor descriptions can be used for members of any type.

If the member's constructor requires no parameters, then no parameter lists need to be specified. So, since the constructor table :: table () was defined with a default parameter value of 15, this definition is sufficient:

```
classdef :: classdef (int size)
        : members (size), no_of_members (size)
{
```

```
            // ...
    }
```

Then the size of the friends table will be 15.

If an object of a class that itself contains class objects is destroyed (for example, classdef), then the body of the enclosing class's destructor is executed first, and then the destructors of the members in the reverse order of their description.

Consider, instead of entering class objects as members, a traditional alternative solution: have pointers to members in the class and initialize the members in the constructor:

```
    class classdef {
            table * members;
            table * friends;
            int no_of_members;
            // ...
    };

    classdef :: classdef (int size)
    {
            members = new table (size);
            friends = new table; // standard is used
            // table size
            no_of_members = size;
            // ...
    }
```

Since the tables were created using the new operation, they must be destroyed with the delete operation:

```
    classdef :: ~ classdef ()
    {
            // ...
            delete members;
            delete friends;
    }
```

These separately created objects can be useful, but note that members and friends point to independent objects, each of which must be explicitly placed and removed. In addition, the pointer and object in free memory together take up more space than the member object.

5.5.5 Arrays of class objects

To be able to describe an array of class objects with a constructor, this class must have a standard constructor, i.e. a constructor called without parameters. For example, according to the definition

```
table tbl [10];
```

an array of 10 tables will be created, each initialized with a call to table :: table (15), since the call to table :: table () will occur with the actual parameter 15.

In the description of an array of objects, it is not possible to specify parameters for the constructor. If the members of an array must be initialized with different values, then tricks with global or static members begin.

When an array is destroyed, the destructor must be called for each element of the array. For arrays that are not allocated with new, this is done implicitly. However, for arrays located in free memory, you cannot implicitly call the destructor, since the translator will not distinguish a pointer to a separate array object from a pointer to the beginning of the array, for example:

```
void f ()
{
            table * t1 = new table;
            table * t2 = new tabl e [10];
            delete t1; // one table is deleted
            delete t2; // trouble:
                                                      // actually deletes 10
tables
}
```

In this case, the programmer must indicate that t2 is a pointer to an array:

```
void g (int sz)
{
            table * t1 = new table;
            table * t2 = new table [sz];
            delete t1;
            delete [] t2;
   }
```
The allocation function stores the number of elements for each allocated array. The requirement to use only the delete [] operation to delete arrays relieves the allocation function of the obligation to store the element counts for each array. Fulfilling such a duty in C ++ implementations would cause significant waste of time and memory and break C compatibility.

5.5.6 Small objects

If your program has a lot of small objects allocated in free memory, then you may find that a lot of time is spent placing and deleting such objects. To get out of this situation, you can determine a more optimal general-purpose memory allocator, or you can delegate the responsibility for allocating free memory to the class creator, who will have to define the appropriate allocation and deletion functions.

Let's go back to the name class used in the examples with table. It could be defined like this:

```
   struct name {
            char * string;
            name * next;
            double value;
            name (char *, double, name *);
            ~ name ();
            void * operator new (size_t);
            void operator delete (void *, size_t);
            private:
                     enum {NALL = 128};
                     static name * nfree;
   };
```
The functions name :: operator new () and name :: operator delete () will be used (implicitly) instead of the global functions operator new () and operator delete (). A programmer can write allocation and deletion functions that are more efficient in terms of time and memory for a particular type than the generic functions operator new () and operator delete (). You can, for example, pre-allocate "chunks" of memory sufficient for objects of type name and link them into a list; then place and delete

operations are reduced to simple list operations. The variable nfree is used as the beginning of a list of unused chunks of memory:

```
void * name :: operator new (size_t)
{
            register name * p = nfree; // select first
            if (p)
                        nfree = p-> next;
            else {// select and link to a list
                        name * q = (name *) new char [NALL * sizeof (name)];
                        for (p = nfree = & q [NALL-1]; q <p; p--) p-> next = p-
1;
                        (p + 1) -> ne xt = 0;
            }
            return p;
}
```

The memory allocator called by new stores its size with the object so that the delete operation can be performed correctly. This additional memory consumption can be easily avoided by using a valve type specific. So, on the author's machine, the function name :: operator new () uses 16 bytes to store the name object, while the standard global function operator new () uses 20 bytes.

Note that in the function name :: operator new () itself, memory cannot be allocated in such a simple way:

```
name * q = new name [NALL];
```

This will cause infinite recursion, since new will call name :: name ().

Freeing memory is usually trivial:

```
void name :: operator delete (void * p, size_t)
{
            ((name *) p) -> next = nfree;
            nfree = (name *) p;
}
```

Casting a parameter of type void * to type name * is necessary because the release function is called after the object is destroyed, so there is no longer a real object of type name, but only a piece of memory of sizeof (name). The parameters of type size_t in the above functions name :: operator new () and name :: operator delete () were not used. How they can be used will be shown in $$ 6.7. Note that our placement and deletion functions are only used for objects of type name, not for arrays of names.

5.6 Exercises

1. (* 1) Modify the calculator program from Chapter 3 to use the table class.
2. (* 1) Define tnode ($$ R.9) as a class with constructors and destructors, etc., define a tree of tnode objects as a class with constructors and destructors, etc.
3. (* 1) Define class intset ($$ 5.3.2) as a set of strings.
4. (* 1) Define the intset class as a set of nodes of type tnode. Think of the tnode structure yourself.
5. (* 3) Define a class for parsing, storing, calculating, and printing simple arithmetic expressions consisting of integer constants and the +, -, *, and / operations. The general class interface should look something like this:

```
class expr {
            // ...
            public:
                        expr (char *);
                        int eval ();
                        void print ();
};
```

The expr :: expr () constructor has a string parameter that specifies an expression.

Expr :: eval () returns the value of the expression, and expr :: print () prints the cout representation of the expression. You can use these functions like this:

```
expr ("123/4 + 123 * 4-3");
cout << "x =" << x.eval () << "\ n";
x.print ();
```

Give two definitions for expr: the first uses a linked list of nodes to represent it , and the second uses a string of characters. Experiment with different formats for printing an expression, such as fully extended parentheses, postfix notation, assembly code, etc.

6. (* 1) Define a char_queue (character queue) class so that its generic interface is representation independent. Implement the class as: (1) a linked list and (2) a vector. Don't think about parallelism .

7. (* 2) Define a class histogram (histogram), which counts numbers in specific intervals, given as parameters to the constructor of this class. Define the function of outputting the histogram. Make handling of values out of range. Hint: refer to <task.h>.

8. (* 2) Identify several classes that generate random numbers with specific distributions. Each class must have a constructor that specifies the distribution parameters and a draw function that returns the "next" value. Hint: refer to <task.h> and the intset class .

9. (* 2) Rewrite the examples date ($$ 5.2.2 and $$ 5.2.4), char_stack ($$ 5.2.5), and intset ($$ 5.3.2) without using any member functions (not even constructors and destructors). Use only class and friend. Check out each of the new versions and compare them with the versions that use member functions.

10. 10. (* 3) For some language, compose class definitions for the name table and the class representing the entry in that table. Explore the translator for that language to see what a real table of names should look like .

11. 11. (* 2) Modify the expr class from Exercise 5 so that you can use variables and the assignment operator = in the expression . Use the class for the table of names from Exercise 10.

12. 12. (* 1) Let there be a program:

```
#include <iostream.h>
main ()
{
        cout << "Hello everyone \ n";
}
```

Change it so that it displays:

```
Initialization
Hello
Deleting
```

The main () function itself cannot be changed.

CHAPTER 6.

Do not produce objects unnecessarily.
- V. Okkam

This chapter focuses on the concept of a derived class. Derived classes are a simple, flexible, and efficient means of defining a class. New capabilities are added to an existing class without the need to reprogram or re-translate it. With derived classes, you can organize a common interface with several different classes so that other parts of your program can work consistently with objects of those classes. The concept of a virtual function is introduced, which makes it possible to use objects appropriately even in those cases when their type is unknown at the stage of translation. The main purpose of derived classes is to make it easier for the programmer to express the generality of classes.

6.1 Introduction and overview

Any concept does not exist in isolation, it exists in interconnection with other concepts, and the power of a given concept is largely determined by the presence of such connections. Since a class is used to represent concepts, the question arises of how to represent the relationship of concepts. The concept of a derived class and the language facilities that support it serve to represent hierarchical relationships, in other words, to express commonality between classes. For example, the concepts of a circle and a triangle are related to each other, since both of them also represent the concept of a figure, i.e. contain a more general concept. To represent circles and triangles in the program and at the same time not lose sight of the fact that they are figures, it is necessary to explicitly define the circle and triangle classes so that it is clear that they have a common class - a figure. The chapter explores what follows from this simple idea, which is essentially the foundation of what is commonly called object-oriented programming.

The chapter is divided into six sections:

$ 6.2 introduces the concept of derived class, class hierarchy, and virtual functions through a series of small examples .

$ 6.3 introduces the concept of purely virtual functions and abstract classes, gives small examples of their use.

$$ 6.4 derived classes are shown in the complete example

$ 6.5 introduces the concept of multiple inheritance as an opportunity for a class to have more than one direct base class, describes how to resolve name collisions arising from multiple inheritance.

$$ 6.6 discusses the access control mechanism.

$ 6.7 shows some techniques for managing free memory for derived classes.

Later chapters will also provide examples using these features of the language.

6.2 Derived classes

Let's discuss how to write a program for accounting for the employees of a certain company. It can use, for example, the following data structure:

```
struct employee { // employees
        char * name; // name
        short age; // age
        short department; // Department
        int salary; // salary
        employee * next;
        // ...
};
```

The next field is needed to link the list of records about employees of one department (employee). Now let's try to define the data structure for the manager:

```
struct manager {
        employee emp; // employee record for manager
        employee * group; // subordinate collective
        short level;
        // ...
};
```

The manager is also an employee, so the employee record is stored in the emp member of the manager object. To humans, this commonality is obvious, but to a translator, the emp member is no different from other class members. A pointer to a structure manager (manager *) is not a pointer to employee (employee *), so one cannot be freely used instead of the other. In particular, without special actions, the manager object cannot be included in the list of objects of type employee. You either have to use an explicit cast manager * or include the address of the emp member in the list of employee records. Both solutions are ugly and can be quite confusing. The correct solution is for the type manager to be of type employee with some additional information:

```
struct manager: employee {
            employee * group;
            short level;
            // ...
};
```

The manager class derives from employee and, conversely, employee is the base class for manager. Besides the group member, the manager class has members of the employee class (name, age, etc.). Graphically, the inheritance relationship is usually depicted as an arrow from derived classes to the base one:

```
employee
            ^
            |
manager
```

It is usually said that a derived class inherits from a base class, so the relationship between them is called inheritance. Sometimes the base class is called the superclass and the derived class is called the subclass. But these terms can be confusing because a derived class object contains an object of its own base class. In general, a derived class is larger than its base in the sense that it contains more data and more functions are defined.

With the definitions employee and manager, you can create a list of employees, some of which are also managers:

```
void f ()
{
            manager m1, m2;
            employee e1, e2;
            employee * elist;
            elist = & m1; // put m1 in elist
            m1.next = & e1; // put e1 in elist
            e1.next = & m2; // put m2 into elist
            m2.next = & e2; // put m2 into elist
            e2.next = 0; // end of list
   }
```

Since the manager is also an employee, the pointer manager * can be used as employee *. At the same time, the employee is not necessarily the manager, and therefore employee * cannot be used as manager *.

In general, if derived has a common base class base, then a pointer to derived can be assigned to a variable of the pointer type to base without explicit type conversions. The reverse conversion from a pointer to base to a pointer to derived can only be explicit:

```
   void g ()
   {
            manager mm;
            employee * pe = & mm; // fine
            employee ee;
            manager * pm = & ee; // mistake:
                                                    // not every
   employee is a manager
            pm-> level = 2; // disaster: when placing ee
                                                    // memory for
   member `level 'was not allocated
            pm = (manager *) pe; // ok: actually pe
                                                    // not configured for
   an mm object of type manager
            pm-> level = 2; // great: pm points to mm object
                                                    // type manager, and
   in it when placing
                                                    // allocated memory
   for the member `level '
   }
```

In other words, if working with an object of a derived class goes through a pointer, then it can be considered as an object of the base class. The converse is not true. Note that a typical C ++ implementation does not dynamically control that, after a type conversion such as that used in the assignment of pe to pm, the resulting pointer is actually set to an object of the required type (see $$ 13.5).

6.2.1 Member functions

Simple data structures like employee and manager are not very interesting by themselves and often not particularly useful. Therefore, let's add functions to them:

```
class employee {
            char * name;
            // ...
            public:
                        employee * next; // is in the general part so that
                                                                     // could work with
the list
                        void print () const;
            // ...
};

class manager: public employee {
            // ...
            public:
                        void print () const;
            // ...
};
```

Some questions need to be answered. How can a member function of the derived class manager use the members of the base class employee? Which members of base employee class can member functions of derived class manager use? What members of the base employee class can a function that is not a member of an object of type manager use? What answers to these questions should the implementation of the language give in order for them to best suit the programmer's task?

Let's consider an example:

```
void manager :: print () const
{
            cout << "name" << name << '\ n';
}
```

A member of a derived class can use a name from the common part of its base class as well as all other members, i.e. without specifying the name of the object. It is assumed that there is an object that this is set to, so the correct reference to name will be this-> name. However, when translating the manager :: print () function, an error will be recorded: a member of a derived class has not been granted access to private members of its base class, so name is not available in this function.

This may seem strange to many, but let's consider an alternative solution: a member function of a derived class has access to the private members of its base class. Then the very notion of a private (private) member loses all meaning, because to access it, you just need to define a derived class. It will no longer be enough to find out who is using the private members of a class by looking at all the member functions and friends of that class. You have to go through all the source files of the program, find the derived classes, then examine each function of those classes. Next, you need to search again for derived classes from those already found, etc. This is at least tiring, and most likely unrealistic. Wherever possible, you should use protected members instead of private members (see $$ 6.6.1).

Generally, the safest solution for a derived class is to only use the common members of its base class:

```
void manager :: print () const
{
        employee :: print (); // print employee data
        // print information about managers
}
```

Note that the :: operation is required because the print () function is overridden in the manager class. This reuse of names is typical of C ++. An unwary programmer would write:

```
void manager :: print () const
{
        print (); // print employee data
                                                      // print information
about managers
}
```

As a result, it would receive a recursive sequence of calls to manager :: print ().

6.2.2 Constructors and destructors

Some derived classes require constructors. If there is a constructor in the base class, then it is he who should be called with the parameters, if any:

```
class e mployee {
        // ...
        public:
                // ...
                employee (char * n, int d);
};

class manager: public employee {
        // ...
        public:
                // ...
                manager (char * n, int i, int d);
};
```

The parameters for the base class constructor are set in the definition of the derived class constructor. In this sense, the base class acts as a class that is a member of the derived class:

```
manager :: manager (char * n, int l, int d)
        : employee (n, d), level (l), group (0)
{
}
```

The base class constructor employee :: employee () can be defined like this :

```
employee :: employee (char * n, int d)
        : name (n), department (d)
{
        next = list;
        list = this;
}
```

Here list should be described as a static member of employee.

Class objects are created from the bottom up: first the base classes, then the members, and finally the derived classes themselves. They are destroyed in reverse order: first the derived classes themselves, then the members, and then the base ones. Members and bases are created in the order they are described in the class, and they are destroyed in the reverse order.

6.2.3 Class hierarchy

A derived class can itself be a base class:

```
class employee { / * ... * /};
class manager: public employee { / * ... * /};
class director: public manager { / * ... * /};
```

This set of related classes is commonly referred to as a class hierarchy. Usually it is represented by a tree, but there are hierarchies with a more general structure in the form of a graph:

```
class temporary { / * ... * /};
class secretary: public employee { / * ... * /};

class tsec
            : public temporary, public secretary {/ * ... * /};

class consultant
            : public tempora ry, public manager {/ * ... * /};
```

We see that classes in C ++ can form a directed acyclic graph (for more details see $$ 6.5.3). This graph for the given classes is:

6.2.4 Type fields

For derived classes to be more than just a convenient form of short description, the language implementation must resolve the question: which of the derived classes does the object pointed to by base * refer to? There are three main ways to answer:

1] Ensure that a pointer can refer to objects of only one type ($$ 6.4.2);

2] Put a type field in the base class that functions can check ;

3] use virtual functions ($$ 6.2.5).

Base class pointers are commonly used when designing container classes (set, vector, list, etc.). Then in case [1] we get homogeneous lists, that is, lists of objects of the same type. Methods [2] and [3] allow you to create heterogeneous lists; lists of objects of several different types (in fact, lists of pointers to these objects). Method [3] is a special version of method [2] that is reliable in the sense of the type. Combinations of methods [1] and [3] provide especially interesting and powerful options; these are discussed in Chapter 8.

First, let's discuss a simple way with a type field, i.e. method [2]. The example with classes manager / employee can be overridden like this:

```
struct employee {
            enum empl_type {M, E};
            empl_type type;
            employee * next;
            char * name;
            short department;
            // ...
};

struct manager: employee {
            employee * group;
            short level;
            // ...
};
```

With these definitions in place, we can write a function that prints data about an arbitrary employee:

```
void print_employee (const employee * e)
{
            switch (e-> type) {
                        case E:
                                    cout << e-> name << '\ t' << e-> department <<
'\ n';
                                    // ...
                                    break;
                        case M:
                                    cout << e-> name << '\ t' << e-> department <<
'\ n';
                                    // ...
                                    manager * p = (manager *) e;
                                    cout << "level" << p-> level << '\ n';
                                    // ...
```

```
                    break;
            }
    }
```
You can print the list of employees like this:
```
    void f (const employee * elist)
    {
            for (; elist; elist = elist-> next) print_em ployee (elist);
    }
```
This is quite a good solution, especially for small programs written by one person, but it has a significant drawback: the translator cannot check how correctly the programmer is handling types. In large programs, this leads to two kinds of errors. The first is when the programmer forgets to check the type field. The second is when not all possible values of the type field are specified in the switch. These errors are easy enough to avoid while writing a program, but it is not at all easy to avoid them when making changes to a non-trivial program, and especially if it is a large program written by someone else. It is even more difficult to avoid such errors because functions like print () are often written to take advantage of the generality of the classes:
```
    void print (const employee * e)
    {
            cout << e-> name << '\ t' << e-> department << '\ n';
            // ...
            if (e-> type == M) {
                    manager * p = (manager *) e;
                    cout << "level" << p-> level << '\ n';
                    // ...
            }
    }
```
If statements like the one in the example are hard to find in a large function that works with many derived classes. But even when they are found, it is not easy to understand what is really going on. In addition, every time a new type of employee is added, changes are required in all important functions of the program, i.e. functions that check the type field. As a result, you have to edit important parts of the program, thereby increasing the time for debugging these parts.

In other words, using the type field is fraught with errors and difficulties in program maintenance. Difficulties increase dramatically as the program grows, because using a type field is contrary to the principles of modularity and data hiding. Each function that works with a type field must know the representation and implementation specifics of any class that is derived from the class containing the type field.

6.2.5 Virtual functions

With virtual functions, you can overcome the difficulties of using a type field. The base class describes functions that can be overridden in any derived class. The translator and loader will ensure the correct correspondence between objects and the functions applied to them:

```
class employee {
        char * name;
        short department;
        // ...
        employee * next;
        static employee * list;
        public:
                employee (char * n, int d);
                // ...
                static void print_list ();
                virtual void print () const;
};
```

The virtual keyword indicates that the print () function can have different versions in different derived classes, and it is the compiler's job to select the correct version when calling print (). The function type is specified in the base class and cannot be overridden in a derived class. The definition of a virtual function must be given for the class in which it was first described (unless it is a purely virtual function, see $$ 6.3). For example:

```
void employee :: print () const
{
        cout << name << '\ t' << department << '\ n';
        // ...
}
```

We see that a virtual function can be used even if there are no derived classes from its class. In a derived class, it is not necessary to override a virtual function if it is not needed there. When constructing a derived class, you need to define only those functions that are really needed in it:

```
class manager: public employee {
        employee * group;
        short level;
        // ...
        public:
                manager (char * n, int d);
                // ...
                void print () const;
};
```

The print_employee () function has been replaced by the print () member functions and is no longer needed. The employee list builds the employee constructor ($$ 6.2.2). You can print it like this:

```
void employee :: print_list ()
{
        for (employee * p = list; p; p = p-> next) p-> print ();
}
```

The details of each employee will be printed according to the type of record about him. Therefore the program

```
int main ()
{
        employee e ("J. Brown", 1234);
        manager m ("J. Smith", 2.1234);
        employee :: print_list ();
}
```

will print

```
J.Smith 1234
level 2
J. Brown 1234
```

Note that the print function will work even if the employee_list () function was written and translated even before a specific derived class manager was conceived! Obviously, for the virtual function to work correctly, you need to store some type of service information in each object of the employee class. Typically, implementations simply use a pointer as such information. This pointer is stored only for objects of the class with virtual functions, but not for objects of all classes, and even for not all objects of derived classes. Additional memory is allocated only for classes that describe virtual functions. Note that when using a type field, it still needs additional memory.

If the function call explicitly specifies the scope resolution operation ::, for example, in the call to manager :: print (), then the mechanism for calling the virtual function has no effect. Otherwise, such a call would lead to infinite recursion. Qualifying the function name has another positive effect: if the virtual function is a substitution (there is nothing unusual in this), then the function body is substituted in the call with the :: operation. This is an efficient calling method that can be used in important cases where one virtual function calls another with the same object. An example of such a case is a call to the manager :: print () function. Since the type of the object is explicitly set in the call to manager :: print () itself, there is no need to define it dynamically for the employee :: print () function that will be called.

6.3 Abstract classes

Many classes are similar to employee in that they can be reasonably defined for virtual functions. However, there are other classes as well. Some, such as the shape class, represent an abstract concept (shape) for which objects cannot be created. The shape class only makes sense as a base class in some derived class. The reason is that it is impossible to give a meaningful definition of virtual functions of the shape class:

```
class shape {
        // ...
        public:
                virtual void rotate (int) {error ("shape :: rotate"); }
                virtual void draw () {error ("shape :: draw"):}
                // neither rotate nor draw abstract shape
                // ...
};
```

Creating an object of type shape (abstract shape) is legal, although completely pointless operation:

```
shape s; // nonsense: `` figure in general "
```

It is meaningless because any operation on s will result in an error.

It is better to describe the virtual functions of the shape class as purely virtual. You can make a virtual function purely virtual by adding an initializer = 0:

```
class shape {
        // ...
        public:
                virtual void rotate (int) = 0; // purely virtual
function
                virtual void draw () = 0; // purely virtual
function
};
```

A class that contains virtual functions is called abstract. Objects of this class cannot be created:

```
shape s; // error: variable of abstract class shape
```

An abstract class can only be used as a base for another class:

```
class circle: public shape {
        int radius;
        public:
                void rotate (int) {} // ok:
                                                // override shape ::
rotate
                void draw (); // fine:
                                                // override shape ::
draw
                circle (point p, int r);
};
```

If a pure virtual function is not defined in a derived class, then it remains so, which means that the derived class is also abstract. With this approach, you can implement classes in stages:

```
class X {
        public:
```

```
                    virtual void f () = 0;
                    virtual void g () = 0;
};

X b; // error: description of an object of abstract class X

class Y: public X {
          void f (); // override X :: f
};

Y b; // error: description of an object of abstract class Y

class Z: public Y {
          void g (); // override X :: g
};

Z c; // fine
```

Abstract classes are needed to define an interface without specifying any specific implementation details. For example, in the operating system, the implementation details of a device driver can be hidden by this abstract class:

```
class character_device {
          public:
                    virtual int open () = 0;
                    virtual int close (const char *) = 0;
                    virtual int read (c onst char *, int) = 0;
                    virtual int write (const char *, int) = 0;
                    virtual int ioctl (int ...) = 0;
                    // ...
};
```

Real drivers will be defined as derived from the character_device class.

With the introduction of the abstract class, we have all the basic tools to write a complete program.

6.4 Example of a completed program

Consider a program for drawing geometric shapes on the screen. It breaks down naturally into three parts:

1] screen monitor: a set of low-level functions and data structures for working with a screen; operates only with concepts such as points, lines;

2] shape library: many definitions of general shapes (eg, rectangle, circle) and standard functions for working with them;

3] Application: Specific shape definitions associated with a task and functions that work with them.

Typically, these three parts are programmed by different people in different organizations and at different times, and they are usually created in the order listed. In this case, difficulties naturally arise, since, for example, the developer of the monitor does not have an exact idea of what tasks it will ultimately be used for. Our example will reflect this fact. To keep the example sizeable, the shape library is quite limited, and the application is trivial. A completely primitive screen representation is used so that even a reader who has no graphics on his machine can work with this program. You can easily replace your screen monitor with a more advanced program without changing the shape library or application program.

6.4.1 Screen Monitor

In the beginning, there was a desire to write a screen monitor in C to further emphasize the separation between implementation layers. But this turned out to be tedious, and therefore a compromise solution was chosen: the programming style adopted in C (no member functions, virtual functions, user-defined operations, etc.), but constructors are used, function parameters are fully described and checked, etc. This monitor is very similar to a C program that was modified to take advantage of the capabilities of C ++, but was not completely redone.

The screen is represented as a two-dimensional array of characters and is controlled by the put_point () and put_line () functions. They use the point structure to communicate with the screen:

```
// screen.h file
const int XMAX = 40;
const int YMAX = 24;
struct point {
```

```
            int x, y;
            point () {}
            point (int a, int b) {x =; y = b; }
};

extern void put_point (int a, int b);
inline void put_point (point p) {put_point (px, py); }
extern void put_line (int, int, int, int);
extern void put_line (point a, point b)
          {put_line (ax, ay, bx, by); }
extern void screen_init ();
extern void screen_destroy ();
extern void screen_refresh ();
extern void screen_clear ();

#include <iostream.h>
```

Before calling the functions that display the image on the screen (put _...), you must call the screen initialization function screen_init (). Changes in the data structure describing the screen will become visible on it only after the call to the screen refresh function screen_refresh (). The reader can verify that updating the screen is simply by copying the new values into an array representing the screen. Here are the functions and data definitions for screen control:

```
#include "screen.h"
#include <stream.h>

enum color {blac k = '*', white = ''};
char screen [XMAX] [YMAX];
void screen_init ()
{
            for (int y = 0; y <YMAX; y ++)
                        for (int x = 0; x <XMAX; x ++)
                                    screen [x] [y] = white;
}
```

Function

```
void screen_destroy () {}
```

just for completeness. In real systems, similar functions of object destruction are usually needed.

Points are recorded only if they hit the screen:

```
inline int on_screen (int a, int b) // hit check
{
        return 0 <= a && a <XMAX && 0 <= b && b <YMAX;
}

void put_point (int a, int b)
{
        if (on_screen (a, b)) scree n [a] [b] = black;
}
```

The put_line () function is used to draw straight lines:

```
void put_line (int x0, int y0, int x1, int y1)
/ *
        Draw a line segment (x0, y0) - (x1, y1).
        Line equation: b (x-x0) + a (y-y0) = 0.
        The abs (eps) value is minimized,
        where eps = 2 * (b (x-x0)) + a (y-y0).
        See Newman, Sproull
        Principles of interactive Computer Graphics
        McGraw-Hill, New York, 1979. pp. 33-34.
* /
{
        register int dx = 1;
        int a = x1 - x0;
        if (a <0) dx = -1, a = -a;
        register int dy = 1;
        int b = y1 - y0;
        if (b <0) dy = -1, b = -b;
        int two_a = 2 * a;
        int two_b = 2 * b;
        int xcrit = -b + two_a;
        register int eps = 0;

  for (;;) {
```

```
                    put_point (x0, y0);
                    if (x0 == x1 && y0 == y1) break;
                    if (eps <= xcrit) x0 + = dx, eps + = two_b;
                    if (eps> = a || a <b) y0 + = dy, eps - = two_a;
            }
    }
```

There are functions to clear and refresh the screen:

```
void screen_clear () {screen_init (); }
void screen_refresh ()
{
            for (int y = YMAX-1; 0 <= y; y--) {// from the top line to the
bottom
                    for (int x = 0; x <XMAX; x ++) // from left column to
right
                            cout << screen [ x] [y];
                    cout << '\ n';
            }
}
```

But you need to understand that all these definitions are stored in some library as a result of the translator's work, and you cannot change them.

6.4.2 Shape library

Let's start by defining the general concept of a figure. The definition must be such that it can be used (as the base class of shape) in different classes representing all specific shapes (circles, squares, etc.). It should also allow you to work with any shape exclusively using the interface defined by the shape class:

```
struct shape {
            static shape * list;
            shape * next;
            shape () {next = list; list = this; }
            virtual point north () const = 0;
            virtual point south () const = 0;
            virtual point east () const = 0;
            virtual point west () const = 0;
             virtual point n east () const = 0;
            virtual point seast () const = 0;
            virtual point nwest () const = 0;
            virtual point swest () const = 0;
            virtual void draw () = 0;
            virtual void move (int, int) = 0;
};
```

The shapes are placed on the screen with the draw () function, and moved around it using move (). Shapes can be positioned relative to each other using the concept of contact points. To designate points of contact, the names of the cardinal points in the compass are used: north - north, ..., neast - north-east, ..., swest - south-west. The class of each particular figure itself determines the meaning of these points and determines how to draw the figure. The shape :: shape () constructor adds a shape to the shape :: list shape list. The next member of each shape object is used to build this list. Since there is no point in generic shape objects, the shape class is defined as an abstract class.

To define a line segment, you need to specify two points or a point and a whole. In the latter case, the segment will be horizontal, and the integer specifies its length. The integer sign indicates where the given point should be relative to the end point, i.e. to the left or right of it:

```
class line: public shape {
/ *
            line segment ["w", "e"]
            north () defines a point - `` above the center of the line segment
and
            as far north as its northernmost point "
* /
            point w, e;
            public:
                        point north () const {return point ((w.x + ex) / 2, ey
<wy? wy: e: y); }
                        point south () const {return point ((w.x + ex) / 2, ey
<wy? ey: wy); }
                        point east () const;
                        point west () const;
                        point neast () const;
```

```
                    point seast () const;
                    point nwest () const;
                    point swest () const;
                    void move (int a, int b)
                            {wx + = a; wy + = b; ex + = a; ey + = b; }
                            void draw () {put_line (w, e); }
                      line (point a, point b) {w = a; e = b; }
                            line (point a, int l) {w = point (a.x + l-1, ay); e
  = a; }
  };
```

The rectangle is defined similarly:

```
  class rectangle: public shape {
       / * nw ------ n ----- ne
           | |
           | |
           w c e
           | |
           | |
           sw ------ s ----- se
       * /
           point sw, ne;
           public:
                    point north () const {return point ((sw.x + ne.x) /2,ne.y);
  }
                    point south () const {return point ((sw.x + ne.x)
  /2,sw.y); }
                    point east () const;
                    point west () const;
                    point neast () const {return ne; }
                    point seast () const;
                    point nwest () const;
                    point swest () const {return sw; }
                    void move (int a, int b)
                            {sw.x + = a; sw.y + = b; ne.x + = a; ne.y + = b;
  }
                    void draw ();
                    rectangle (point, point);
```

```
            };
```

The rectangle is drawn from two points. The constructor becomes more complicated, since it is necessary to find out the relative position of these points:

```
   rectangle :: rectangle (point a, point b)
   {
            if (ax <= bx) {
                     if (ay <= by) {
                              sw = a;
                              ne = b;
                     }
                     else {
                              sw = point (ax, by);
                              ne = point (bx, ay);
                     }
            }
            else {
                     if (ay <= by) {
                              sw = point (bx, ay);
                              ne = point (ax, by);
                     }
                     else {
                              sw = b;
                              ne = a;
                     }
            }
   }
```

To draw a rectangle, you need to draw four lines:

```
   void rectangle :: draw ()
   {
            poin t nw (sw.x, ne.y);
            point se (ne.x, sw.y);
            put_line (nw, ne);
            put_line (ne, se);
            put_line (se, sw);
            put_line (sw, nw);
```

```
}
```

The shape library has shape definitions and functions for working with them:

```
void shape_refresh (); // draw all shapes
void stack (shape * p, const shape * q); // put p over q
```

The shape update function is needed to work with our primitive display; she just re-draws all the shapes. Note that this function has no idea what shapes it draws:

```
void shape_refresh ()
{
        screen_clear ();
        for (shape * p = shape :: list; p; p = p-> next) p-> draw ();
        screen_refresh ();
}
```

Finally, there is one really utility function that draws one shape on top of another. To do this, it defines the south (south ()) of one figure just above the north (north ()) of another:

```
void stack (shape * p, const shape * q) // place p over q
{
        point n = q-> north ();
        point s = p-> south ();
        p-> move (nx-sx, ny-s.y + 1);
}
```

Suppose now that this library is the property of some software vendor and that it only sells a header file with shape definitions and translated function definitions. You can still define new shapes using the functions you purchased.

6.4.3 Application program

The application program is extremely simple. A new myshape shape is defined (if drawn, it resembles a face), and then the main () function is given, in which it is drawn with a hat. First, let's describe the myshape shape:

```
#include "shape.h"
class myshape: public rectangle {
```

```
            line * l_eye; // left eye
            line * r_eye; // right eye
            line * mouth; // mouth
            public:
                        myshape (point, point);
                        void draw ();
                        void move (int, int);
   };
```

The eyes and mouth are separate independent objects that the myshape class constructor creates:

```
   my shape :: myshape (point a, point b): rectangle (a, b)
   {
            int ll = neast (). x-swest (). x + 1;
            int hh = neast (). y-swest (). y + 1;
            l_eye = new line (point (swest (). x + 2, swest (). y + hh * 3/4),
   2);
            r_eye = new line (point (swest (). x + ll-4, swest (). y + hh *
   3/4), 2);
            mouth = new line (point (swest (). x + 2, swest (). y + hh / 4), ll-
   4);
   }
```

The objects representing the eyes and mouth are emitted separately by shape_refresh (). In principle, you can work with them independently of the my_shape object to which they belong. This is one of the ways to set facial features for a hierarchically constructed myshape object. How it can be done differently can be seen from the task of the nose. No nose type is defined, it is simply drawn in the draw () function:

```
   void myshape :: draw ()
   {
            rectangle :: draw ();
            int a = (swest (). x + neast (). x) / 2;
            int b = (swest (). y + neast (). y) / 2;
            put_p oint (point (a, b));
   }
```

The movement of the myshape shape is reduced to the movement of the object of the base rectangle class and to the movement of the secondary objects (l_eye, r_eye and mouth):

```
void myshape :: move (int a, int b)
{
            rectangle :: move (a, b);
            l_eye-> move (a, b);
            r_eye-> move (a, b);
            mouth -> move (a, b);
}
```

Finally, let's define some shapes and move them:

```
int main ()
{
            screen_init ();
            shape * p1 = new rectangle (point (0,0), point (10,10));
            shape * p2 = new line (point (0,15), 17);
            shape * p3 = new myshape (point (15,10), point (27,18));
            shape_refre sh ();
            p3-> move (-10, -10);
            stack (p2, p3);
            stack (p1, p2);
            shape_refresh ();
            screen_destroy ();
            return 0;
}
```

Note again that functions like shape_refresh () and stack () work with objects whose types were known after the definition of these functions (and, probably, after their translation).

Here's the resulting face with a hat:

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

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

To simplify the example, copying and deleting shapes has not been discussed.

6.5 Multiple inheritance

In $$ 1.5.3 and $$ 6.2.3, it was already stated that a class can have multiple direct base classes. This means that more than one class can be specified after: in the class description. Consider a modeling problem in which parallel actions are represented by the standard task class library, and the displayed library class provides information collection and output. Then the class of modeled objects (let's call it satellite) can be defined as follows:

```
class satellite: public task, public displayed {
        // ...
};
```

This definition is commonly referred to as multiple inheritance. Conversely, the existence of only one direct base class is called single inheritance.

All the operations defined in the satellite class are appended with the union of the operations of the task and displayed classes:

```
void f (satellite & s)
{
        s.dr aw (); // displayed :: draw ()
        s.delay (10); // task :: delay ()
        s.xmit (); // satellite :: xmit ()
}
```

Alternatively, a satellite object can be passed to functions with a task or displayed parameter:

```
void highlight (displayed *);
void suspend (task *);
void g (satellite * p)
{
          highlight (p); // highlight ((displayed *) p)
          suspend (p); // suspend ((task *) p);
}
```

Obviously, the implementation of this feature requires some (simple) trick from the translator: you need to pass functions with task and displayed parameters to different parts of the satellite object.

For virtual functions, of course, the call will be executed correctly anyway:

```
class task {
          // ...
          virtual pending () = 0;
};

class displayed {
          // ...
          virtual void draw () = 0;
};

class satellite: public task, public displayed {
          // ...
          void pending ();
          void draw ();
};
```

Here, the satellite :: draw () and satellite :: pending () functions on a satellite object will be called as if it were a displayed or task object, respectively.

Note that targeting only single inheritance limits the implementation of the displayed, task, and satellite classes. In such a case, the satellite class could be task or displayed, but not both together (unless, of course, task is derived from displayed or vice versa). In any case, flexibility is lost.

6.5.1 Multiple occurrences of a base class

The ability to have more than one base class entails the possibility of multiple occurrences of the class as base. Let's say the classes task and displayed are derived from the link class, then satellite will appear twice:

```
class task: public link {
        // link is used to link all
        // tasks to the list (manager list)
        // ...
};

class displayed: public link {
        // link is used to link all
        // displayed objects (list of images)
        // ...
};
```

But there are no problems. Two different link objects are used for different lists, and these lists do not conflict with each other. Of course, without the risk of ambiguity, you cannot access the members of the link class, but how to do this correctly is shown in the next section. Graphically, the satellite object can be represented as follows:

However, examples can be given when a common base class should not be represented by two different objects (see $$ 6.5.3).

6.5.2 Disambiguation

Naturally, two base classes can have member functions with the same name:

```
class task {
        // ...
        virtual debug_info * get_debug ();
};

class displayed {
        // ...
        virtual debug_info * get_debug ();
};
```

When using the satellite class, such function ambiguity should be resolved:

```
void f (satellite * sp)
{
        debug_info * dip = sp-> get_debug (); // error: ambiguity
```

```
            dip = sp-> task :: get_debug (); // fine
            dip = sp-> displayed :: get_debug (); // fine
  }
```
However, explicitly resolving the ambiguity is troublesome, so the best way to resolve it is to define a new function in a derived class:
```
  class satellite: public task, public derived {
   // ...
            debug_info * get_debug ()
            {
                        debug_info * dip1 = task: get_debug ();
                          debug_inf o * dip2 = displayed :: get_debug ();
                        return dip1-> merge (dip2);
            }
  };
```
Thus, information from the base satellite classes is localized. Since satellite :: get_debug () is an override of the get_debug () functions from both base classes, it is guaranteed to be called on every call to get_debug () on an object of type satellite.

The translator detects name collisions that occur when the same name is defined in more than one base class. Therefore, the programmer does not need to specify which name is used, unless its use is really ambiguous. In general, the use of base classes does not result in name collisions. In most cases, even if the names are the same, no collision occurs because names are not used directly for derived class objects.

A similar problem when two classes have functions with the same name but different purposes is discussed in $$ 13.8 using the draw () function for the Window and Cowboy classes as an example.

Unless ambiguity arises, it is unnecessary to specify the name of the base class when explicitly referring to a member. In particular, if multiple inheritance is not used, it is sufficient to use the "somewhere in base class" notation. This allows the programmer not to remember the name of the direct base class and saves him from errors (though rare) that arise when rebuilding the class hierarchy. For example, in the function from $$ 6.2.5
```
  void manager :: print ()
  {
          employee :: print ();
          // ...
  }
```

employee is assumed to be the direct base class for manager. The result of this function does not change if employee happens to be an indirect base class for manager, but the print () function is not in the direct base class. However, someone could restructure the classes as follows:

```
  class employee {
          // ...
          virt ual void print ();
  };

  class foreman: public employee {
          // ...
          void print ();
  };

  class manager: public foreman {
          // ...
          void print ();
  };
```

Now the foreman :: print () function will not be called, although this function was almost certainly intended to be called. With a little trick, you can overcome this difficulty:

```
  class foreman: public employee {
          typedef employee inherited;
          // ...
          void print ();
  };

  class manager: public foreman {
          typedef foreman inherited;
          // ...
          void print ();
  };
```

```
void manager :: pr int ()
{
        inherited :: print ();
        // ...
}
```

Scope rules, particularly those related to nested types, ensure that multiple inherited types that arise do not conflict with each other. In general, it's a matter of taste whether a solution with inherited type should be considered visual or not.

6.5.3 Virtual base classes

In the previous sections, multiple inheritance was considered as an essential factor allowing, through merging of classes, to painlessly integrate independently created programs. This is the most basic use of multiple inheritance, and fortunately (but not coincidentally) it is the simplest and most reliable way to use it.

Sometimes the use of multiple inheritance implies a fairly close relationship between the classes, which are considered "sibling" base classes. These sibling classes usually need to be designed together. In most cases, this does not require a special programming style that is significantly different from the one we just looked at. It's just that the derived class has some extra work to do. It usually boils down to overriding one or more virtual functions (see $$ 13.2 and $$ 8.7). In some cases, sibling classes need to share information. Since C ++ is a strongly typed language, common information is only possible by explicitly specifying what is common in these classes. A virtual base class can be used for this indication.

A virtual base class can be used to represent a "head" class, which can be instantiated in different ways:

```
class window {
        // head information
        virtual void draw ();
};
```

For simplicity, let's look at just one kind of general information from the window class — the draw () function. Various more advanced classes can be defined to represent windows. Each defines its own (more advanced) drawing function (draw):

```
class window_w_border: public virtual window {
            // class "window with frame"
            // definitions associated with the frame
            void draw ();
};

class window_w_m enu: public virtual window {
            // class "window with menu"
            // definitions associated with the menu
            void draw ();
};
```

Now I would like to define a window with a frame and a menu:

```
class window_w_border_and_menu
            : public virtual window,
            public window_w_border,
            public window_w_ menu {
            // class "window with frame and menu"
            void draw ();
};
```

Each derived class adds new window properties. To take advantage of a combination of all of these properties, we must ensure that the same window object is used to represent occurrences of the base class window in those derived classes. This is what the definition of window in all derived classes provides as a virtual base class.

You can depict the composition of an object of class window_w_border_and_menu as follows:

To see the difference between regular and virtual inheritance, compare this figure with the figure from $$ 6.5 showing the composition of a satellite object. In the inheritance graph, each base class with a given name that was specified as virtual will be represented by a single object of that class. On the contrary, each base class that was not specified as virtual in the description of inheritance will be represented by its own object.

Now we need to write all these draw () functions. It's not too difficult, but there is a catch for the unwary programmer. First, let's go the simplest way, which just leads to it:

```
void window_w_border :: draw ()
{
        window :: draw ();
        // draw the frame
}

void window_w_menu :: draw ()
{
        window :: draw ();
        // draw the menu
}
```

So far so good. This is all obvious, and we follow the pattern of defining such functions under the condition of single inheritance ($$ 6.2.1), which worked fine. However, a trap appears in the derived class of the next level:

```
void window_w_border_and_menu :: draw () // trap!
{
        window_w_border :: draw ();
        window_w_menu :: draw ();
        // now only related operations
        // to window with frame and menu
}
```

At first glance, everything is quite normal. As usual, all the operations required for the base classes are performed first, and then those related to the derived classes themselves. But as a result, the window :: draw () function will be called twice! For most graphics programs, this is not just an unnecessary challenge, but damage to the picture on the screen. Typically, the second display overwrites the first.

To avoid the trap, you need to be slow to act. We will separate the actions performed by the base class from the actions performed from the base class. To do this, in each class, we introduce the _draw () function, which performs the actions necessary only for it, and the draw () function will perform the same actions, plus the actions needed for each base class. For the window class, changes are reduced to the introduction of an unnecessary function:

```
class window {
        // head information
```

```
            void _draw ();
            void draw ();
   };
```

For derived classes, the effect is the same:

```
   class window_w_border: public virtual window {
            // class "window with frame"
            // definitions associated with the frame
            void _draw ();
            void draw ();
   };

   void window_w_border :: draw ()
   {
            window :: _ draw ();
            _draw (); // draws frame
   };
```

Only for the derived class of the next level does the difference of the function appear, which allows you to bypass the trap of calling window :: draw () again, since now window :: _ draw () is called and only once:

```
   class window_w_border_and_menu
            : public virtual wi ndow,
            public window_w_border,
            public window_w_menu {
                     void _draw ();
                     void draw ();
   };

   void window_w_border_and_menu :: draw ()
   {
            window :: _ draw ();
            window_w_border :: _ draw ();
            window_w_menu :: _ draw ();
            _draw (); // now only related operations
                                                      // to window with
   frame and menu
   }
```

It is not necessary to have both window :: draw () and window :: _ draw (), but having them avoids a variety of simple cliches.

In this example, the window class serves as a repository for information common to window_w_border and window_w_menu, and defines an interface for communication between the two classes. If only inheritance is used, then the generality of information in the class tree is achieved by the fact that this information is moved to the root of the tree until it becomes available to all the node classes interested in it. As a result, an unpleasant effect easily arises: the root of the tree or classes close to it are used as the global namespace for all classes of the tree, and the class hierarchy degenerates into a set of unrelated objects.

It is essential that each of the sibling classes overrides the functions defined in the common virtual base class. Thus, each of the brothers can get his own version of operations, different from the others. Let the window class have a general input function get_input ():

```
class window {
        // head information
        virtual void draw ();
        virtual void get_input ();
};
```

In one of the derived classes, you can use this function without worrying about where it is defined:

```
class window_w_banner: public virtual window {
        // class "window with title"
        void draw ();
        void update_banner_text ();
};

void window_w_banner :: update_banner_text ()
{
        // ...
        get_input ();
        // change the title text
}
```

In another derived class, you can define the get_input () function without worrying about who will use it:

```
class window_w_menu: public virtual window {
        // class "window with menu"
        // definitions associated with the menu
        void draw ();
        void get_input (); // overrides window :: get_input ()
};
```

All of these definitions come together in a derived class at the next level:

```
class window_w_banner_and_menu
        : public virtual window,
                public window_w_banner,
                public window_w_menu
{
        void draw ();
};
```

Ambiguity control makes sure that different functions are defined in sibling classes:

```
c lass window_w_input: public virtual window {
        // ...
        void draw ();
        void get_input (); // overrides window :: get_input
};

class window_w_input_and_menu
        : public virtual window,
                public window_w_input,
                public window_w_menu
                {// error: both classes window_w_input and
             // window_w_menu override function
             // window :: get_input
        void draw ();
};
```

The translator detects such an error, and the ambiguity can be resolved in the usual way: in the window_w_input and window_w_menu classes, a function that overrides the offending function, and somehow disambiguate:

```
class window_w_input_and_menu
        : public virtual window,
                    public window_w_input,
                    public window_w_menu
{
        void draw ();
        void get_input ();
};
```

In this class window_w_input_and_menu :: get_input () will override all get_input () functions. The disambiguation mechanism is described in detail in $$ R.10.1.1.

6.6 Access Control

A class member can be private, protected, or public:

The private member of class X can only be used by member functions and friends of class X.

Protected member of class X can only be used by member functions and friends of class X, as well as member functions and friends of everyone derived from X classes (see $$ 5.4.1).

The common member can be used in any function.

These rules correspond to the division of functions that refer to a class into three kinds: functions that implement the class (its friends and members), functions that implement the derived class (friends and members of the derived class), and all other functions.

Access control is applied uniformly to all names. Access control is not affected by which entity the name represents. This means that member functions, constants, etc. can be private. along with private members representing data:

```
class X {
        private:
                enum {A, B};
                void f (int);
                int a;
};

void X :: f (int i)
{
```

```
            if (i <A) f (i + B);
            a ++;
}

void g (X & x)
{
            int i = X :: A; // error: X :: A private member
            xf (2); // error: X :: f private member
            x.a ++; // error: X :: a private member
}
```

6.6.1 Protected members

Let's give an example of protected members, returning to the window class from the previous section. Here, the _draw () functions were intended only for use in derived classes because they provided an incomplete set of capabilities, and therefore were not convenient and reliable enough for general use. They were like a building material for more advanced functions. On the other hand, the draw () functions were intended for general use. This difference can be expressed by breaking the interfaces of the window classes into two parts - the protected interface and the generic interface:

```
class window {
            public:
                        virtual void draw ();
                        // ...
            protected:
                        void _draw ();
                        // other functions that serve as building materials
            private:
                        // class view
};
```

This partitioning can also be done in derived classes such as window_w_border or window_w_menu.

The _ prefix is used in the names of protected functions that are part of a class implementation, as a general rule: names beginning with _ should not be present in parts of the program that are open to public use. Names starting with double underscores are best avoided altogether (even for members).

Here's a less practical but more detailed example:

```
class X {
        // by default the private part of the class
        int priv;
        protected:
                int prot;
        public:
                int publ;
                void m ();
};
```

For member X :: m, access to members of the class is unlimited:

```
void X :: m ()
{
        priv = 1; // fine
        prot = 2; // fine
        publ = 3; // fine
}
```

A derived class member only has access to public and protected members:

```
class Y: public X {
        void mderived ();
};

Y :: mderived ()
{
        priv = 1; // error: priv private member
        prot = 2; // ok: prot is a protected member, but
                                                    // mderived ()
member of derived class Y
        publ = 3; // ok: publ common member
}
```

Only common members are available in a global function:

```
void f (Y * p)
{
```

```
            p-> priv = 1; // error: priv private member
            p-> prot = 2; // error: prot is a protected member, and f ()
                                                        // not a friend or
member of classes X and Y
            p-> publ = 3; // ok: publ common member
}
```

6.6.2 Access to base classes

Like a member, a base class can be described as private, protected, or public:

```
class X {
            public:
                        int a;
                        // ...
};

class Y1: public X {};
class Y2: protected X {};
class Y3: private X { };
```

Since X is a common base class for Y1, in any function, if necessary, you can (implicitly) convert Y1 * to X *, and moreover, the common members of the X class will be available in it:

```
void f (Y1 * py1, Y2 * py2, Y3 * py3)
{
            X * px = py1; // ok: X is the generic base class Y1
            py1-> a = 7; // fine
            px = py2; // error: X is a protected base class Y2

            py2-> a = 7; // mistake
            px = py3; // error: X is a private base class Y3
            py3-> a = 7; // mistake
}
```

Now let them describe

```
class Y2: protected X {};
class Z2: p ublic Y2 {void f (); };
```

Since X is the protected base class of Y2, only friends and members of Y2, as well as friends and members of any classes derived from Y2 (in particular Z2), can (implicitly) convert Y2 * to X * if necessary. In addition, they can access the public and protected members of class X:

```
void Z2 :: f (Y1 * py1, Y2 * py2, Y3 * py3)
{
        X * px = py1; // ok: X is the generic base class Y1
        py1-> a = 7; // fine
        px = py2; // ok: X is the protected base class Y2,

                                        // and Z2 is a derived
class Y2
        py2-> a = 7; // fine
        px = py3; // error: X is a private base class Y3
        py3-> a = 7; // mistake
}
```

Finally, consider:

```
class Y3: private X {void f (); };
```

Since X is Y3's private base class, only friends and members of Y3 can (implicitly) convert Y3 * to X * as needed. In addition, they can access the public and protected members of class X:

```
void Y3 :: f (Y1 * py1, Y2 * py2, Y3 * py3)
{
        X * px = py1; // ok: X is the generic base class Y1
        py1-> a = 7; // fine
        px = py2; // error: X is a protected base class Y2

        py2-> a = 7; // mistake
        px = py3; // ok: X is the private base class Y3,
                                        // and Y3 :: f is a
member of Y3
        py3-> a = 7; // fine
}
```

6.7 Free memory

By defining operator new () and operator delete (), you can take control of the memory management for the class. It is also possible (and often more useful) to do this for a class that serves as the base for many derived classes. Let's say we need our own functions for allocating and freeing memory for the employee class ($$ 6.2.5) and all its derived classes:

```
class employee {
        // ...
        public:
                void * operator new (size_t);
                void operator delete (void *, size_t);
};

void * employee :: operator new (size_t s)
{
        // allocate memory in `s' bytes
        // and return a pointer to it
}

void employee:: operator delete (void * p, size_t s)
{
        // `p 'must point to memory in` s' bytes,
        // assigned by the function employee :: operator new ();
        // free this memory for reuse
}
```

The purpose of the hitherto mysterious parameter of type size_t becomes obvious. This is the size of the object to be deallocated. When a simple employee is removed, this parameter is set to sizeof (employee), and when a manager is removed, it is set to sizeof (manager). Therefore, the native functions of the hosting classes may not store the size of each hosted object. Of course, they can store those sizes (like general purpose allocation functions) and ignore the size_t parameter in the operator delete () call, but then they are unlikely to be better than general purpose allocation and deallocation functions.

How does the translator determine the required size to be passed to operator delete ()? As long as the type specified in operator delete () matches the true type of the object, it's simple; but consider this example:

```
class manager: public employee {
        int level;
        // ...
};

void f ()
{
        employee * p = new manager; // problem
        delete p;
}
```

In this case, the translator will not be able to correctly determine the size. As with deleting an array, you need a programmer's help. It should define a virtual destructor in the employee base class:

```
class employee {
        // ...
        public:
                // ...
                void * operator new (size_t);
                void operator delete (void *, size_t);
                virtual ~ employee ();
};
```

Even an empty destructor will solve our problem:

```
employee :: ~ employee () {}
```

Now, memory will be freed in the destructor (and the size is known in it), and any class derived from employee will also be forced to define its own destructor (thereby setting the desired size), unless the user himself defines it. Now the following example will run correctly:

```
void f ()
{
        employee * p = new manager; // no problem now
        delete p;
}
```

The placement is done with a (compiler-generated) call

```
employee :: operator new (sizeof (manager))
```

and release by calling

```
employee :: operato r delete (p, sizeof (manager))
```

In other words, if you need to have correct allocation and release functions for derived classes, you must either define a virtual destructor in the base class, or not use the size_t parameter in the release function. Of course, when designing the language, it was possible to provide means to free the user from this problem. But then the user would "be freed" from certain advantages of a more optimal, albeit less reliable system.

In general, it always makes sense to define a virtual destructor for all classes that are actually used as base classes, i.e. they work with objects of derived classes and, possibly, delete them through a pointer to the base class:

```
class X {
            // ...
            public:
                        // ...
                        virtual void f (); // X has a virtual function, so
                                                                        // define a virtual
destructor
                        virtual ~ X ();
};
```

6.7.1 Virtual Constructors

Having learned about virtual destructors, it is natural to ask: "Can constructors be virtual in the same way?" The short answer is no. You can give a longer answer: "No, but you can easily get the desired effect."

A constructor cannot be virtual, because in order to construct an object correctly, it must know its true type. Moreover, the constructor is not an ordinary function. It can interact with memory management functions, which is not possible with normal functions. It also differs from ordinary member functions in that it is not called on existing objects. Hence, you cannot get a pointer to the constructor.

But these limitations can be worked around by defining a function that contains a constructor call and returns a constructed object. This is fortunate because it is often necessary to create a new object without knowing its true type. For example, during translation, it is sometimes necessary to make a copy of the tree representing the parsed expression. The tree can contain nodes of expressions of different kinds. Suppose that nodes that contain repeated operations in an expression need to be copied only once. Then we need a virtual propagation function for the expression node.

Typically, "virtual constructors" are standard parameterless constructors or copy constructors that take the type of the result as a parameter:

```
class expr {
        // ...
        public:
                expr (); // standard constructor
                virtual expr * new_expr () {return new expr (); }
};
```

The virtual function new_expr () simply returns a standard-initialized object of type expr allocated in free memory. In a derived class, you can override the new_expr () function to return an object of that class:

```
class conditional: public expr {
        // ...
        public:
                conditional (); // standard constructor
                expr * new_expr () {return new conditional (); }
};
```

This means that, given an object of class expr, the user can create an object of "exactly the same type":

```
void user (expr * p1, expr * p2)
{
        expr * p3 = p1-> new_expr ();
        expr * p4 = p2-> new_expr ();
        // ...
}
```

The variables p3 and p4 are assigned pointers of an unknown but suitable type.

In the same way, you can define a virtual copy constructor called a multiplication operation, but you need to be more careful about the specifics of the copy operation:

```
class expr {
        // ...
```

```
            expr * left;
            expr * right;
            public:
                        // ...
                        // copy `s 'to` this'
                        inline void copy (expr * s);
                        // create a copy of the object this is looking at
                        virtual expr * clone (int deep = 0);
   };
```

The deep parameter shows the difference between copying the object itself (shallow copy) and copying the entire subtree rooted by the object (deep copy). The default value of 0 means shallow copy.

The clone () function can be used like this:

```
   void fct (expr * root)
   {
            expr * c1 = root-> clone (1); // deep copy
            expr * c2 = root-> clone (); // shallow copy
            // ...
   }
```

Being virtual, clone () can propagate objects of any expr-derived class. Real copying can be defined like this:

```
   void expr :: copy (expression * s, int deep)
   {
            if (deep == 0) {// copy only members
                        * this = * s;
            }
            else {// go through the pointers:
                        left = s-> clone (1);
                        right = s-> clone (1);
                        // ...
            }
   }
```

The expr :: clone () function will only be called for objects of type expr (but not for classes derived from expr), so you can simply put in and return an object of type expr that is its own copy:

```
expr * expr :: clone (int deep)
{
        expr * r = new expr (); // build a standard expression
        r-> copy (this, deep); // copy `* this 'to` r'
        return r;
}
```

This clone () function can be used for classes derived from expr if no data members appear in them (which is a typical case):

```
class arithmetic: public expr {
        // ...
        // no new data members =>
        // you can use the already defined clone function
};
```

On the other hand, if you have added data members, you need to define your own clone () function:

```
class conditional: public expression {
        expr * cond;
        public:
                inline void copy (cond * s, int deep = 0);
                expr * clone (int deep = 0);
                // ...
};
```

The copy () and clone () functions are defined similarly to their counterparts from expression:

```
expr * conditional :: clone (int deep)
{
        conditional * r = new conditional ();
        r-> copy (this, deep);
        return r;
}

void conditional :: copy (expr * s, int deep)
{
        if (deep == 0) {
                * this = * s;
        }
```

```
            else {
                        expr :: copy (s, 1); // copy part of expr
                        cond = s-> cond-> clone (1);
            }
   }
```

The definition of the last function shows the difference between true copying in expr :: copy () and full copying in expr :: clone () (i.e. creating a new object and copying into it). Simple copy is useful for defining more complex copy and duplicate operations. The difference between copy () and clone () is equivalent to the difference between the assignment operator and the copy constructor ($$ 1.4.2) and is equivalent to the difference between the _draw () and draw () functions ($$ 6.5.3). Note that the copy () function is not virtual. It doesn't need to be, because the clone () function that calls it is virtual. Obviously, simple copy operations can also be defined as substitution functions.

6.7.2 Specifying placement

By default, the new operation creates the specified object in free memory. What if you need to place an object in a certain place? This can be achieved by overriding the placement operation. Consider a simple class:

```
   class X {
            // ...
            public:
                        X (int);
                        // ...
   };
```

An object can be placed anywhere by entering additional parameters into the placement function:

```
   // operation of placing in the specified location:
   void * operator new (size_t, void * p) {return p; }
```

and setting these parameters for the new operation as follows:

```
   char buffer [sizeof (X)];
   void f (int i)
   {
            X * p = new (buffer) X (i); // place X in buffer
            // ...
```

```
}
```

The operator new () function used by the new operation is selected according to the parameter matching rules ($$ R.13.2). All operator new () functions must have size_t as their first parameter. The size specified by this parameter is implicitly passed by the new operation.

The operator new () function we have defined with a given placement is the simplest of its kind. Another example of an allocation function that allocates memory from some given area can be given:

```
class Arena {
        // ...
        virtual void * alloc (size_t) = 0;
        virtual void free (void *) = 0;
};

void operator new (size_t sz, Arena * a)
{
        return a.alloc (sz) ;
}
```

Now you can allocate memory for objects of arbitrary types from different areas (Arena):

```
extern Arena * Persistent; // persistent memory
extern Arena * Shared; // shared memory
void g (int i)
{
        X * p = new (Persistent) X (i); // X in persistent memory
        X * q = new (Shared) X (i); // X in shared memory
        // ...
}
```

If we place an object in a memory area that is not directly controlled by the standard free memory allocation functions, then we must take care of the correct destruction of the object. The main remedy here is to explicitly call the destructor:

```
void h (X * p)
{
        p-> ~ X (); // call the destructor
```

```
            Persistent-> free (p); // free memory
   }
```

Note that explicit calls to destructors, as well as special-purpose global placement functions, should be avoided whenever possible. There are times when it is difficult to do without them, but a beginner should think three times before using an explicit destructor call, and should first consult with a more experienced colleague.

6.8 Exercises

1. (* 1) Let there be a class

```
class base {
        public:
                virtual void iam () {cout << "base \ n"; }
};
```

Define two base-derived classes, and in each define an iam () function that returns the name of your class. Create objects of these classes and call iam () on them. Assign the addresses of derived class objects to a base * pointer and call iam () with those pointers.

2. (* 2) Implement screen control primitives ($$ 6.4.1) in a way that is reasonable for your system.
3. (* 2) Define the classes triangle and circle .
4. (* 2) Define a function that draws a line segment connecting two shapes. First you need to find the closest points of the shapes, and then connect them.
5. (* 2) Modify the shape example so that line is derived from rectangle, or vice versa.
6. (* 2) Let there be a class

```
class char_vec {
        int sz;
        char element [1];
        public:
                static new_char_vec (int s);
                char & operator [] (int i) {return element [i]; }
                // ...
};
```

Define the new_char_vec () function to allocate a contiguous chunk of memory for char_vec objects so that elements can be indexed as an array element []. When will this feature cause serious difficulties?

7. (* 1) Describe the data structures needed for the example with the shape class from $$ 6.4 and explain how a virtual call can be made.

8. (* 1.5) Describe the data structures needed for the $$ 6.5 satellite example and explain how a virtual call can be made.

9. (* 2) Describe the data structures needed for the window class example from $$ 6.5.3 and explain how a virtual call can be made.

10. (* 2) Describe a class of graphic objects with a set of possible operations that will be common base in the library of graphic objects. Explore any graphics libraries to understand what operations are needed. Define a database object class with a set of possible operations that will be a common base object class, stored as a sequence of database fields. Explore some databases to understand what operations are needed. Define a graphical database object using or not using multiple inheritance. Discuss the relative pros and cons of both solutions.

11. (* 2) Write a variant of the clone () function from $$ 6.7.1, in which the object being propagated can be placed in the Arena region ($$ 6.7.2), passed as a parameter. Implement the simple Arena class as derived from Arena.

12. (* 2) Let there are classes Circle, Square, and Triangle derived from the shape class. Define a function intersect () with two parameters of type Shape * that calls an appropriate function to find out if the given two shapes intersect . To do this, in the specified classes, you need to define the corresponding virtual functions. Don't waste your energy on a function that actually sets the shapes to intersect, just make sure the function calls are in the correct sequence.

13. (* 5) Design and implement a library for event driven modeling . Hint: use <task.h>. There are already outdated functions and you can write better. There should be a class task (task). The task object must be able to save its state and restore it (for this you can define the functions task :: save () and task :: restore ()) and then it can act

as a coroutine. Special tasks can be defined as objects of classes derived from task. Define the program that the task is executing as a virtual function. It should be possible to pass parameters to a new task as parameters to its constructor or constructors. There must be a dispatcher who implements the concept of virtual time. Define a function task :: delay (long) that will "eat" virtual time. An important development question: Is the dispatcher part of the task class, or should it be independent? Tasks should be able to communicate with each other. Develop a queue class for this purpose. Think of a way for the task to wait for input from multiple queues. All dynamic errors must be handled consistently. How to organize debugging of programs written using such a library?

CHAPTER 7.

If I choose a word, it only means
what I decide, nothing more and nothing less.
- Humpty Dumpty

The chapter contains a description of the operation overloading mechanism in C ++. The programmer can specify the interpretation of operations when they are applied to objects of a particular class. In addition to arithmetic, logical, and relational operations, you can override function calls (), indexing [], indirection ->, and assignment and initialization. You can define explicit and implicit conversions between custom and base types. Shows how to define a class whose object can only be copied and destroyed using special user-defined functions.

7.1 Introduction

Typically, programs use objects that are concrete representations of abstract concepts. For example, in C ++, the data type is int along with the operations +, -, *, /, etc. realizes (albeit limitedly) the mathematical concept of the whole. Usually, a concept is associated with a set of actions that are implemented in the language in the form of basic operations on objects, specified in a concise, convenient and familiar form. Unfortunately, only a small number of concepts are directly represented in programming languages. Thus, the concepts of complex numbers, matrix algebra, logical signals, and strings in C ++ do not have a direct expression. The ability to specify the representation of complex objects along with a set of operations performed on such objects is implemented in C ++ classes. By letting the programmer define operations on class objects, we get a more convenient and traditional notation for working with these objects, compared to the one in which all operations are defined as ordinary functions. Let's give an example:

```
class complex {
        double re, im;
        public:
                complex (double r, double i) {re = r; im = i; }
                friend complex operator + (complex, complex);
                friend complex operator * (complex, complex);
};
```

Here is a simple implementation of the concept of a complex number when it is represented by a pair of double-precision floating-point numbers that can only be manipulated with the + and * operations. The interpretation of these operations is set by the programmer in the definitions of the functions named operator + and operator *. So, if b and c are of type complex, then b + c means (by definition) operator + (b, c). Now you can get closer to the usual notation of complex expressions:

```
void f ()
{
          complex a = complex (1,3.1);
          complex b = complex (1.2,2);
          complex c = b;
          a = b + c;
          b = b + c * a;
          c = a * b + complex (1,2);
}
```

The normal precedences of the operations are preserved, so the second expression is executed as b = b + (c * a) rather than b = (b + c) * a.

7.2 Operator functions

You can describe functions that determine the interpretation of the following operations:

```
+ - * /% ^ & | ~!
= <> + = - = * = / =% = ^ = & =
| = << >> >> = << = ==!   = <=> = &&
|| ++ - -> *, -> [] () new delete
```

The last five operations mean: indirection ($$ 7.9), indexing ($$ 7.7), function call ($$ 7.8), allocation in free memory and deallocation ($$ 3.2.6). You cannot change the priorities of these operations, as well as the syntax rules for expressions. So, you cannot define a unary operation%, as well as a binary operation!. You cannot introduce new lexemes to denote operations, but if the set of operations does not suit you, you can use the usual notation for a function call. So use pow () rather than **. These restrictions may be considered draconian, but looser rules easily lead to ambiguity. Let's say we define the operation ** as exponentiation, which at first glance seems like an obvious and simple task. But if you think about it, then questions arise: should the operations ** be performed from left to right (as in Fortran) or from right to left (as in Algol)? How to interpret the expression a ** p as a * (* p) or as (a) ** (p)?

The name of an operator function is the service word operator followed by the operation itself, for example, operator <<. An operator function is described and called as an ordinary function. Using the operation symbol is just a shorthand notation for calling an operator function:

```
void f (complex a, complex b)
{
        complex c = a + b; // short form
        complex d = operator + (a, b); // explicit call
}
```

Given the above description of the type complex, the initializers in this example are equivalent.

7.2.1 Binary and unary operations

A binary operation can be defined as a member function with one parameter, or as a global function with two parameters. Therefore, for any binary operation @, the expression aa @ bb is interpreted as either aa.operator (bb) or operator @ (aa, bb). If both functions are defined, then the choice of interpretation occurs according to the rules of parameter matching ($$ R.13.2). A prefix or postfix unary operation can be defined as a member function without parameters, or as a global function with only parameters. For any prefixed unary operator @, the @aa expression is interpreted as either aa.operator @ () or operator @ (aa). If both functions are defined, then the choice of interpretation occurs according to the rules of parameter matching ($$ R.13.2). For any postfix unary operator @, the @aa expression is interpreted as either aa.operator @ (int) or operator @ (aa, int). This is explained in detail in $$ 7.10. If both functions are defined, then the choice of interpretation occurs according to the rules of parameter matching ($$ 13.2). An operation can only be defined in accordance with the syntax rules available for it in the C ++ grammar. In particular, you cannot define% as a unary operation, but + as a ternary operation. Let us illustrate what has been said with examples:

```
class X {
```

```
            // members (implicitly using the `this' pointer):
            X * operator & (); // prefix unary operation &
                                                           // (take the address)
            X operator & (X); // binary operation & (AND bitwise)
            X operator ++ (int); // postfix increment
            X operator & (X, X); // error: & cannot be ternary
            X operator / (); // error: / cannot be unary
    };

    // global functions (usually friends)

    X operator- (X); // prefix unary minus
    X operator- (X, X); // binary minus
    X operator - (X &, int); // postfix increment
    X operator- (); // error: no operand
    X operator- (X, X, X); // error: ternary operation
    X operator% (X); // error: unary operation%
```

The [] operation is described in $$ 7.7, the () operation in $$ 7.8, the -> operation in $$ 7.9, and ++ and operations in $$ 7.10.

7.2.2 Predefined Operation Properties

Only a few assumptions are made about the properties of custom operations. In particular, operator =, operator [], operator (), and operator-> must be non-static member functions. This ensures that the first operand of these operations is an address.

For some built-in operations, their interpretation is defined as a combination of other operations performed on the same operands. So, if a is of type int, then ++ a means a + = 1, which in turn means a = a + 1. Such relationships are not stored for custom operations unless the user specifically defines them for such purpose. Thus, the definition of operator + = () for type complex cannot be inferred from the definitions of complex :: operator + () and complex operator = ().

By historical coincidence, it turned out that the operations = (assignment), & (taking an address) and, (comma operation) have predefined properties for class objects. But you can close these properties from an arbitrary user if you describe these operations as private:

```
class X {
            // ...
            private:
                        void operator = (const X &);
                        void operator & ();
                        void operator, (const X &);
                        // ...
};

void f (X a, X b)
{
            a = b; // error: operation = private
            & a; // error: operation & private

            a, b // error: operation, private
}
```

On the other hand, you can, on the contrary, assign a different meaning to these operations with the help of the corresponding definitions.

7.2.3 Operator functions and user-defined types

An operator function must either be a member or have at least one parameter, which is an object of the class (for functions that override the new and delete operations, this is optional). This rule ensures that the user cannot change the interpretation of expressions that do not contain objects of the user-defined type. In particular, you cannot define an operator function that only works with pointers. This ensures that extensions are possible in C ++, but not mutations (except for the operations =, &, and, for class objects).

An operator function that has the first parameter of the main type cannot be a member function. So, if we add the complex variable aa to the integer 2, then with a suitable description of the member function aa + 2 can be interpreted as aa.operator + (2), but 2 + aa cannot be interpreted that way, since there is no int class for which + is defined like 2.operator + (aa). Even if it were possible, the interpretation of aa + 2 and 2 + aa had to deal with two different member functions. This example is trivially written with non-member functions.

Each expression is checked for ambiguities. If user-defined operations specify a possible interpretation of an expression, it is checked according to the rules of $$ R.13.2.

7.3 Custom type conversion operations

The implementation of a complex number described in the introduction is too limited to satisfy anyone and needs to be extended. This is done by simply repeating descriptions of the same kind that have already been applied:

```
class complex {
        double re, im;
        public:
                complex (double r, double i) {re = r; im = i; }
                friend complex operator + (complex, complex);
                friend complex operator + (complex, double);
                friend complex operator + (double, complex);
                friend complex operator- (complex, double);
                friend complex operator- (complex, double);
                friend complex operator- (double, complex);
                complex operator- (); // unary -
                friend complex operator * (complex, complex);
                friend complex operator * (complex, double);
                friend complex operator * (double, complex);
                // ...
};
```

With this definition of a complex number, we can write:

```
void f ()
{
        comp lex a (1.1), b (2.2), c (3.3), d (4.4), e (5.5);
        a = -bc;
        b = c * 2.0 * c;
        c = (d + e)   * a;
}
```

Still, it's tedious, as we just did for operator * () to write its own function for each combination of complex and double. Moreover , a reasonable means of complex arithmetic should provide dozens of such functions (see, for example, how the complex type is described in <complex.h>).

7.3.1 Constructors

Instead of describing several functions, you can describe a constructor that creates a complex from the double parameter:

```
class complex {
        // ...
        complex (double r) {re = r; im = 0; }
};
```

This determines how to get complex if double is given. This is the traditional way of extending the real line to the complex plane.

A constructor with a single parameter does not need to be called explicitly:

```
complex z1 = complex (23);
complex z2 = 23;
```

Both z1 and z2 will be initialized by calling complex (23).

A constructor is an algorithm for creating a value of a given type. If a value of some type is required and there is a constructor building it , the parameter of which is this value, then this constructor will be used. So, the complex class could be described as follows:

```
class complex {
        double re, im;
        public:
                complex (double r, double i = 0) {re = r; im = i; }
                friend complex operator + (complex, complex);
                friend complex operator * (complex, complex);
                complex operator + = (complex);
                complex operator * = (complex);
                // ...
};
```

All operations on complex variables and integer constants with this description become legal. An integer constant will be interpreted as a complex number with an imaginary part equal to zero. So, a = b * 2 means

```
a = operator * (b, complex (double (2), double (0)))
```

It only makes sense to define new versions of operations such as + if practice shows that the efficiency gains by eliminating type conversions are worth it. For example, if it turns out that the operation of multiplying a complex variable by a real constant is critical, then op erator * = (double) can be added to the set of operations :

```
class complex {
        double re, im;
        public:
                complex (double r, double i = 0) {re = r; im = i; }
                friend complex operator + (complex, complex);
                friend complex operator * (complex, complex);
                complex & operator + = (complex);
                complex & operator * = (complex);
                complex & operator * = (double);
                // ...
};
```

Assignments like * = and + = can be very useful for working with user-defined types, since they are usually shorter than their usual "doubles" * and +, and they can also speed up program execution by eliminating temporary variables :

```
inline complex & complex :: operator + = (complex a)
{
        re + = a.re;
        im + = a.im;
        return * this;
}
```

This function does not require a temporary variable to store the result, and it is simple enough for the translator to "perfectly" perform body substitution. Such simple operations as addition of complex ones are also easy to define directly:

```
inline complex operator + (complex a, complex b)
{
        return complex (a.re + b.re, a.im + b.im);
}
```

Here, a constructor is used in the return statement, which gives the translator valuable optimization clues. But for more complex types and operations, for example, such as matrix multiplication, the result cannot be specified as a single expression, then the * and + operations are easier to implement using * = and + =, and they will be easier to optimize:

```
matrix & matrix :: operator * = (const matrix & a)
{
        // ...
        return * this;
}

matrix operator * (const matrix & a, const matrix & b)
{
        matrix prod = a;
        prod * = b;
        return prod;
}
```

Note that an operation defined in this way does not need any special access rights to the class to which it is applied, i.e. this operation must not be a friend or member of this class.

Custom type conversion is applied only if it is unique ($$ 7.3.3).

Constructed as a result of an explicit or implicit call to a constructor, an object is automatic, and is destroyed as soon as possible, usually immediately after the statement in which it was created is executed .

7.3.2 Conversion operations

The constructor is convenient to use for type conversion, but undesirable consequences are possible:

[1] Implicit conversions from a user-defined type to a base type are not possible (since base types are not classes).

[2] You cannot specify a conversion from a new type to an old one without changing the description of the old type.

[3] You cannot define a constructor with one parameter without defining the type conversion.

The latter is not a big problem, and the first two can be overcome by defining an operator conversion function for the original type. The member function X :: operator T (), where T is a type name , defines a type conversion from X to T. For example, you can define a tiny type whose values are in the range 0..63, and this type can be in arithmetic operations are almost free to mix with wholes:

```
class tiny {
        char v;
        void assign (int i)
        {if (i> 63) {error ("out of range"); v = i & ~ 63; }
                v = i;
        }
        public:
                tiny (int i) {assign (i)}
                tiny (const tiny & t) {v = tv; }
                tiny & operator = (const tiny & t) {v = tv; return * this;
}
                tiny & operator = (int i) {assign (i); return * this; }
                operator int () {return v; }
};
```

The range is checked both when the tiny object is initialized and when it is assigned an int. One tiny object can be assigned to another without spanning. To perform normal integer operations on variables of type tiny, a tiny :: operator int () function is defined to perform an implicit conversion from tiny to int. Where an int is required and a variable of type tiny is specified, the value converted to int is used:

```
void main ()
{
        tiny c1 = 2;
        tiny c2 = 62;
        tiny c3 = c2 -c1; // c3 = 60
        tiny c4 = c3; // no range control (not needed)
        int i = c1 + c2; // i = 64
        c1 = c2 + 2 * c1; // out of range: c1 = 0 (not 66)
        c2 = c1 - i; // out of range: c2 = 0
        c3 = c2; // no range control (not needed)
}
```

A vector of tiny objects might be more useful because it saves memory. To make this type convenient to use, you can use the indexing operation [].

Custom type conversion operations can be useful for working with types that implement non-standard number representations (base-100 arithmetic, fixed-point arithmetic, BCD representation , etc.). This usually requires overriding operations such as + and *.

Conversion functions are especially useful for working with data structures for which reading (implemented as a conversion operation) is trivial, and assignment and initialization are much more complex operations. Conversion functions are needed for the istream and ostream types to make possible, for example, the following operators:

```
while (cin >> x) cout << x;
```

The input operation cin >> x returns istream &. It is implicitly converted to a value indicating the state of the cin stream, which is then checked in a while statement (see $$ 10.3.2). Still, defining an implicit type conversion in which you can lose the converted value is usually a bad decision.

In general, it is best to use transform operations sparingly. The excess of such operations can cause a large number of ambiguities. The translator detects these ambiguities, but they can be tricky to resolve. Perhaps in the beginning it is better to use named functions for conversions, for example, X :: intof (), and only after such a function has been tried out as follows, and an explicit type conversion is considered inelegant, it can be replaced with an operator conversion function X :: operator int ( ).

7.3.3 Ambiguities

The assignment or initialization of an object of class X is legal if the value being assigned is of type X, or if there is only one conversion to a value of type X.

In some cases, a value of the desired type is constructed by repeated use of constructors or conversion operations. This must be specified explicitly, implicit custom conversion of only one nesting level is allowed. In some cases, there are several ways to construct a value of the desired type, but this is illegal. Let's give an example:

```
class x {/ * ... * / x (int); x (char *); };
        class y {/ * ... * / y (int); };
        class z {/ * ... * / z (x); };
        xf (x);
        yf (y);
```

```
            zg (z);
            void k1 ()
            {
                        f (1); // invalid, ambiguity: f (x (1)) or f (y (1))
            f (x (1));
                        f (y (1));
                        g ("asdf"); // invalid, g (z (x ("asdf"))) not used
   }
```

Custom type conversions are considered only when, without them, the called function cannot be unambiguously selected:

```
   class x {/ * ... * / x (int); };
            void h (double);
            void h (x);
            void k2 ()
            {
                        h (1);
            }
```

The call to h (1) can be interpreted as either h (double (1)) or h (x (1)), so it might be illegal due to the unambiguity requirement. But since in the first interpretation only the standard conversion is used, it is selected according to the rules specified in $$ 4.6.6 and $$ R.13.2.

The rules for type conversions are not too easy to formulate and implement, and they do not have sufficient generality. Consider the requirement of uniqueness of a legal transformation. The easiest way is to let the translator apply any transform it can find. Then, to find out the correctness of the expression, it is not necessary to consider all existing transformations. Unfortunately, in this case the behavior of the program will depend on what kind of transformation is found. As a result, the behavior of the program will depend on the order of the transformation descriptions. Since these descriptions are often scattered across different source files (possibly created by different programmers), the result of the program will depend on the order in which these files are merged into the program. On the other hand, implicit conversions can be prohibited altogether, and this is the simplest solution. But the result will be a poor user-defined interface, or an explosive growth of overloaded functions and operations, as we saw with the complex class in the previous section.

The most general approach takes into account all type information and considers all existing conversions. For example, taking into account the above descriptions in the assignment aa = f (1), you can deal with the call f (1), since the type aa defines a single conversion. If aa is of type x, then the only transformation is f (x (1)), since it alone gives the type x needed for the left-hand side. If aa is of type y, f (y (1)) will be used. With the most general approach, it is possible to deal with the call to g ("asdf"), since g (z (x ("asdf))) is its only interpretation. The difficulty with this approach is that it requires a thorough analysis of the entire expression to establish the interpretation of each operations and function calls As a result, translation slows down, expression evaluation can happen in a strange way and cryptic error messages appear when the translator takes into account the transformations defined in the libraries, etc. As a result, the translator has to take into account more information than the programmer himself knows! , in which the verification is a strictly bottom-up process, when only one operation with operands whose types have already passed the verification is considered at a time.

The strict bottom-up parsing of the expression assumes that the return type is not considered when resolving the overload:

```
class quad {
            // ...
            pu blic:
                        quad (double);
                        // ...
};
quad operator + (quad, quad);
void f (double a1, double a2)
{
            quad r1 = a1 + a2; // double precision addition
            quad r2 = quad (a1) + a2; // forces to use
                                                    // operations with
quad types
}
```

The design of the language was based on strictly bottom-up parsing, since it is more understandable, and besides, it is not the job of the translator to solve such issues as the programmer wants for addition.

However, it should be noted that if the types of both parts are defined in the assignment and initialization, then both are used to resolve them:

```
class real {
        // ...
        public:
                operator double ();
                operator int ();
                // ...
};

void g (real a)
{
        double d = a; // d = a.double ();
        int i = a; // i = a.int ();
        d = a; // d = a.double ();
        i = a; // i = a.int ();
}
```

In this example, expressions are still parsed in a strictly bottom-up manner, with only one operation and the types of its operands considered at a time.

7.4 Literals

You cannot define literal values for classes, like 1.2 and 12e3 are double literals. However, basic type literals can be used instead of member functions to interpret class values. A common tool for constructing such values is single parameter constructors. If a constructor is simple enough to be implemented by substitution, it makes sense to think of its call as a literal. For example, given the description of the complex class in <complex.h>, there will be two function calls in the expression zz1 * 3 + zz2 * complex (1,2), not five. The two * operations will result in a function call, and the + operation and constructor calls to construct complex (3) and complex (1,2) will be implemented by substitution.

7.5 Large objects

When performing any binary operation for the complex type, the functions that implement this operation will be passed as parameters of a copy of both operands. The overhead of copying two doubles is noticeable, although it appears to be acceptable. Unfortunately, the presentation of not all classes is so conveniently compact. To avoid redundant copying, you can define functions with reference type parameters:

```
class matrix {
            double m [4] [4];
            public:
                        matrix ();
                        friend matrix operator + (const matrix &, const matrix
&);
                        friend matrix operator * (const matrix &, const matrix
&);
};
```

References allow you to use expressions with common arithmetic operations and for large objects without unnecessary copying. You cannot use pointers for this purpose, since it is not possible to override the interpretation of an operation if it is applied to a pointer. The plus operation for matrices can be defined as follows:

```
matrix operator + (const matrix & arg1, const & arg2)
{
            matrix sum;
            for (int i = 0; i <4; i ++)
                        for (int j = 0; j <4; j ++)
                                    sum.m [i] [j] = arg1.m [i] [j] + arg2.m [i] [j];
            return sum;
}
```

Here, in the operator + () function, the operands are selected by reference, and the value of the object itself is returned. A more efficient solution would be to return the link too:

```
class matrix {
            // ...
            friend matrix & operator + (const matrix &, const matrix &);
            friend matrix & operator * (const matrix &, const matrix &);
};
```

This is okay, but there is a memory allocation problem. Since a reference to the result of an operation will be passed as a reference to a function's return value, it cannot be an automatic variable of that function. Since the operation can be used multiple times in the same expression, the result cannot be a local static variable either. As a rule, the result will be written to the object allocated in free memory. It is usually cheaper (in terms of execution time and data and instruction memory) to copy the resulting value than to allocate it to free memory and then eventually free the allocated memory. In addition, this method is easier to program.

7.6 Assignment and Initialization

Consider a simple string class:

```
struc t string {
            char * p;
            int size;                  // the size of the vector pointed to by
p
            string (int size) {p = new char [size = sz]; }
            ~ string () {delete p; }
};
```

A string is a data structure that contains a pointer to a character vector and the size of that vector. The vector is created by the constructor and removed by the destructor. But as we saw in $$ 5.5.1, problems can arise here:

```
void f ()
{
            string s1 (10);
            string s2 (20)
            s1 = s2;
}
```

Two character vectors will be placed here, but as a result of the assignment s1 = s2, the pointer to one of them will be destroyed and replaced with a copy of the second. Upon exiting f () , the destructor will be called for s1 and s2, which will delete the same vector twice, the results of which will most likely be deplorable. To solve this problem, you need to define the appropriate assignment of objects of type string:

```
struct string {
            char * p;
            int size; // the size of the vector pointed to by p
            string (int size) {p = new char [size = sz]; }
            ~ string () {delete p; }
            string & operator = (const string &);
};

string & string :: operator = (const string & a)
{
        if (this! = & a) {// dangerous when s = s
                delete p;
                p = new char [size = a.size];
                strcpy (p, ap);
        }
        return * this;
}
```

With this definition of string, the previous example will proceed as intended. But after a slight change in f (), the problem arises again, but in a different guise:

```
void f ()
{
        string s1 (10);
        string s2 = s1; // initialization, not assignment
}
```

Now only one object of type string is constructed by the string :: string (int) constructor, and two strings will be destroyed. The point is that a custom assignment operation does not apply to an uninitialized object. It is enough to look at the function string :: operator () to understand the reason for this: the pointer p will then have an undefined, in fact, random value. As a rule, an assignment operation assumes that its parameters are initialized. For an initialization of the type shown in this example, this is not the case by definition. Therefore, to cope with initialization, you need a similar, but its own function:

```
struct string {
char * p;
        int size;               // the size of the vector pointed to by
p
        string (int size) {p = new char [size = sz]; }
        ~ string () {delete p; }
        string & operator = (const string &);
        string (const string &);
};

string :: string (const string & a )
{
            p = new char [size = sz];
            strcpy (p, ap);
}
```

An object of type X is initialized using the X (const X &) constructor. We keep repeating that assignment and initialization are different operations. This is especially important in cases where a destructor is defined. If class X has a non-trivial destructor, for example, freeing an object in free memory, most likely, this class will need a full set of functions to avoid copying objects by member:

```
class X {
// ...
            X (something); // constructor that creates the object
            X (const X &); // copy constructor
            operator = (const X &); // assignment:
                                                            // delete and copy
            ~ X (); // destructor that removes the object

};
```

There are two more cases when you have to copy an object: passing a parameter to a function and returning a value. When passing a parameter, an uninitialized variable, i.e. the formal parameter is initialized. The semantics of this operation are identical to other types of initialization. The same happens when a function returns a value, although this case is not so obvious. In both cases, the copy constructor is used:

```
string g (string arg)
{
            return arg;
}

main ()
{
            string s = "asdf";
            s = g (s);
}
```

Obviously, after calling g (), s must be "asdf". It is not difficult to write a copy of the s value into the s parameter, for this you need to call the copy constructor for string. To get another copy of the value of s upon exiting g (), you need another call to the string (const string &) constructor. This time, a temporary variable is initialized, which is then assigned to s. For optimization, one, but not both, of such copy operations can be removed. Naturally, temporary variables used for such purposes are destroyed appropriately by the destructor string :: ~ string () (see $$ R.12.2).

If in class X the assignment operation X :: operator = (const X &) and the copy constructor X :: X (const X &) are not explicitly specified by the programmer, the missing operations will be created by the translator. These generated functions will copy member by member for all members of class X. If the members are simple, as in the case of complex numbers, this is what you need, and the generated functions will turn into simple and optimal bitwise copying. If custom copy operations are defined for the members themselves, they will be called accordingly:

```
class Record {
            string name, address, profession;
            // ...
};

void f (Record & r1)
{
            Record r2 = r1;
}
```

Here, string :: operator = (const string &) will be called to copy each member of type string from r1. In our first and inferior version, the string class has a pointer member and a destructor. So the standard member copying is almost certainly wrong for it. The translator can warn about such situations.

7.7 Indexing

The operator function operator [] sets the indexing interpretation for class objects. The second parameter of this function (index) can be of any type.

This allows, for example, to define associative arrays. As an example, you can rewrite the definition from $$ 2.3.10, where an associative array was used in a small program that counts the number of occurrences of words in a file. There a function was used for this. We'll define the real type of the associative array:

```
class assoc {
        struct pair {
                char * name;
                int val;
        };
        pair * vec;
        int max;
        int free;
        assoc (const assoc &); // prevents copying
        assoc & operator = (const assoc &); // prevents copying
        public:
                assoc (int);
                int & operator [] (const char *);
                void print_all ();
};
```

The assoc object stores a vector of pair structures of size max . The variable free stores the index of the first free element of the vector.

To prevent copying of assoc objects, the copy constructor and the assignment operation are described as private. The constructor looks like this:

```
assoc :: assoc (int s)
{
        m ax = (s <16)? 16: s;
        free = 0;
        vec = new pair [max];
}
```

The implementation uses the same inefficient search algorithm as in $$ 2.3.10. But now, if the vector overflows, the assoc object grows:

```
#include <string.h>
int & assoc :: operator [] (cons t char * p)
/ *
```

```
            works with multiple pairs (pair structures):
            searches for p, returns a reference to an integer value from the
found pair,
            creates a new pair if p is not found
* /
{
            register pair * pp;
            for (pp = & vec [free-1]; vec <= pp; pp--)
                        if (strcmp (p, pp-> na me) == 0) return pp-> val;
            if (free == max) { // overflow: vector increases
                        pair * nvec = new pair [max * 2];
                        for (int i = 0; i <max; i ++) nvec [i] = vec [i];
                                    delete vec;
                        vec = nvec;
                        max = 2 * max;
            }
            pp = & vec [free ++];
            pp-> name = new char [strlen ( p) +1];
            strcpy (pp-> name, p);
            pp-> val = 0; // initial value = 0
            return pp-> val;
}
```

Since the representation of the assoc object is hidden from the user, you need to be able to print it in some way. The next section will show you how to define a real iterator for such an object. Here we restrict ourselves to a simple print function:

```
void assoc :: print_all ()
{
            for (int i = 0; i <free; i ++)
                        cout << vec [i] .name << ":" << vec [i] .val << '\ n';
}
```

Finally, a trivial program can be written:

```
main () // count the number of entries in the input
                                                // stream of each word
{
            const MAX = 256; // longer than the length of the longest word
```

```
        char buf [MAX];
        assoc vec (512);
        while (cin >> buf) vec [buf] ++;
        vec.print_all ();
}
```

Experienced programmers will notice that the second comment can be easily refuted. To solve the problem arising here, see Exercise $$ 7.14 [20]. The concept of an associative array will be further developed in $$ 8.8.

The operator [] () function must be a member of the class. It follows that the equivalence x [y] == y [x] may fail if x is an object of the class. The usual equivalence relations, which are true for operations on built-in types, may not hold for user-defined types ($$ 7.2.2, see also $$ 7.9).

7.8 Calling a function

Function call, i.e. the expression (expression-list) construct can be viewed as a binary operation in which expression is the left operand and expression-list is the right one. The call operation can be overloaded like other operations. In operator () (), the list of actual parameters is computed and type-checked according to the usual rules for passing parameters. Overloading a call operation makes sense primarily for types with which only one operation is possible, as well as for those types, one of the operations on which is so important that all the others can be ignored in most cases.

We have not provided an iterator definition for an assoc array. For this purpose, you can define a special class, assoc_iterator, whose task is to emit elements from assoc in some order. The iterator needs to have access to the data stored in the assoc, so it must be described as a friend:

```
class assoc {
        friend class assoc_iterator;
        pair * vec;
        int max;
        int free;
        public:
                assoc (int);
                int & opera tor [] (const char *);
};
```

An iterator can be defined like this:

```
class assoc_iterator {
            const assoc * cs; // assoc array
            int i; // current index
            public:
                        assoc_iterator (const assoc & s) {cs = & s; i = 0; }
                        pair * operator () ()
                                    {return (i <cs-> free)? & cs-> vec [i ++]: 0; }
};
```

The assoc array of the assoc_iterator object must be initialized, and each time it is accessed using the operator function (), a pointer to a new pair (pair structure) from this array will be returned . When the end of the array is reached, 0 is returned:

```
main () // count the number of entries in the input

                                                                        // stream of each
word
{
            const MAX = 256; // longer than the length of the longest
word
            char buf [MAX];
            assoc vec (512);
            while (cin >> buf) vec [buf] ++;
            assoc_iterator next (vec);
            pair * p;
            while (p = ne xt (vec))
                        cout << p-> name << ":" << p-> val << '\ n';
}
```

An iterator of this kind has an advantage over a set of functions that solve the same problem: an iterator can have its own private data, in which information about the progress of the iteration can be stored. It is usually important that you can run several iterators of the same type at the same time .

Of course, using objects to represent iterators has nothing to do with overloading operations directly. Some people prefer to use the iterator type with operations such as first (), next (), and last (), while others prefer the

++ operator overloading , which allows you to get an iterator used as a pointer (see $$ 8.8). In addition, the operator function operator () is actively used to extract substrings and index multidimensional arrays.

The operator () function must be a member function.

7.9 Indirect appeal

The member indirection operation -> can be defined as a unary postfix operation. This means if there is a class

```
class Ptr {
        // ...
        X * operator -> ();
};
```

objects of class Ptr can be used to access members of class X in the same way as pointers are used for this purpose:

```
void f (Ptr p)
{
        p-> m = 7; // (p.operator -> ()) -> m = 7
}
```

The transformation of an object p into a pointer p.operator -> () does not depend in any way on the member m to which it points. It is for this reason that operator -> () is a unary postfix operation. However, we are not introducing new syntactic conventions, so the member name must still come after ->:

```
void g (Ptr p)
{
        X * q1 = p->; // syntax error
        X * q2 = p.operator -> (); // fine
}
```

Operation overloading -> is primarily used to create "tricky pointers", i.e. objects that, in addition to being used as pointers, allow some operations to be performed with each call to the indicated object with their help. For example, you can define the RecPtr class to provide access to Rec objects stored on disk. The parameter to the RecPtr constructor is the name that will be used to locate the object on disk. When accessing an object using the RecPtr :: operator -> () function, it is rewritten to the main memory, and at the end of the work, the RecPtr destructor writes the changed object back to disk.

```
class RecPtr {
            Rec * in_core_address;
            const char * identifier;
            // ...
            public:
                        RecPt r (const char * p)
                                    : identifier (p) {in_core_address = 0; }
                        ~ RecPtr ()
                                    {write_to_disc (in_core_address, identifier); }
                        Rec * operator -> ();
};

Rec * RecPtr :: operator -> ()
{
            if (in_core_address == 0)
                        in_core_address = read_from_disc (identifier);
            return in_core_address;
}
```

You can use it like this:

```
main (int argc, const char * argv)
{
            for (int i = argc; i; i--) {
                        RecPtr p (argv [i]);
                        p-> update ();
            }
}
```

In fact, the RecPtr type must be defined as a type template (see $$ 8), and the type of the Record structure will be its parameter. In addition, the real program will contain error handling and interaction with the disk will not be so primitive.

For ordinary pointers, the -> operation is equivalent to the operations using * and []. So, if described

```
Y * p;
```

then the relation

```
p-> m == (* p) .m == p [0] .m
```

As always, such ratios are not guaranteed for user-defined operations . Where such equivalence is nevertheless required, it can be ensured:

```
class X {
        Y * p;
        public:
                Y * operator -> () {return p; }
                Y & operat or * () {return * p; }
                Y & operator [] (int i) {return p [i]; }
};
```

If your class has more than one such operator defined, it is reasonable to provide equivalence, just as it is reasonable to provide for a simple variable x with some class that contains ++, + = =, and + so that ++ x and x + = 1 were equivalent to x = x + 1.

Overloading -> like [] overloading can play an important role for a whole class of real programs, and is not just an experiment for the sake of curiosity. The point is that in programming the concept of indirection is key, and the overloading -> provides a clear, direct, efficient way of representing this concept in a program. There is another view of the -> operation as a means of defining a limited but useful version of the concept of delegation in C ++ (see $$ 12.2.8 and 13.9).

7.10 Increment and decrement

If we have thought of "tricky pointers", then it is logical to try to override the increment ++ and decrement - operations in order to get the capabilities for classes that these operations give for built-in types. Such a task is especially natural and necessary if the goal is to replace the type of ordinary pointers with the type of "tricky pointers", for which the semantics remains the same, but some dynamic control actions appear . Let's say there is a program with a common error:

```
void f1 (T a) // traditional use
{
        T v [200];
         T * p = & v [10];
        p--;
```

```
            * p = a; // Arrived: `p 'is configured outside the array,
                                                         // and it was not
   found
            ++ p;
            * p = a; // fine
   }
```

Naturally, there is a desire to replace the pointer p with an object of the CheckedPtrToT class , according to which an indirect call is possible only if it really points to an object. It will be possible to apply increment and decrement to such a pointer only if the pointer is set to an object within the boundaries of the array, and as a result of these operations, an object within the boundaries of the same array will be obtained :

```
   class CheckedPtrToT {
            // ...
   };

   void f2 (T a) // variant with control
   {
            T v [200];
            CheckedPtrToT p (& v [0], v, 200);
            p--;
            * p = a; // dynamic error:
                                                         // `p 'out of bounds
   of array
            ++ p;
            * p = a; // fine
   }
```

Increment and decrement are the only operations in C ++ that can be used as postfix and prefix operations. Therefore, in the definition of the CheckedPtrToT class, we must provide separate functions for the prefix and postfix increment and decrement operations:

```
   class CheckedPtrToT {
            T * p;
            T * array;
            int size;
            public:
                                                        // initial value `p '
                                                        // bind to array `a 'of
   size` s'
                        CheckedPtrTo T (T * p, T * a, int s);
                                                        // initial value `p '
                                                        // bind to a single
   object
                        CheckedPtrToT (T * p);
                        T * operator ++ (); // prefix
                        T * operator ++ (int); // postfix
                          T * operator - (); // prefix
                        T * operator - (int); // postfix
                        T & operator * (); // prefix
   };
```

A parameter of type int serves as an indication that the function will be called for a postfix operation. In fact, this parameter is artificial and never used, but only serves to distinguish between postfix and prefix operations. To remember which version of the operator ++ function is used as a prefix operation, it is enough to remember that the prefix is the version without an artificial parameter, which is true for all other unary arithmetic and logical operations. The artificial parameter is used only for "special" postfix operations ++ and -.   Using the CheckedPtrToT class, an example can be written like this:

```
   void f3 (T a) // variant with control
   {
            T v [200];
            CheckedPtrToT p (& v [0], v, 200);
            p.operator - (1);
            p.operator * () = a; // dynamic error:
                                                        // `p 'out of bounds
   of array
            p.operator ++ ();
            p.operator * () = a; // fine
   }
```

Exercise $$ 7.14 [19] proposes to complete the definition of the CheckedPtrToT class , and another exercise ($$ 9.10 [2]) is to convert it to a type template that uses exceptions to report dynamic errors. Examples of using ++ and - for iterations can be found in $$ 8.8.

7.11 String class

Now we can bring a more meaningful version of the string class. It counts the number of references per string to minimize copying and uses standard C ++ strings as constants.

```
#include <iostream.h>
#include <string.h>
class string {
            struct srep {
                        char * s; // pointer to string
                        int n; // reference count
                        srep () {n = 1; }
            };
            srep * p;
            public:
                        string (const char *); // string x = "abc"
                        string (); // string x;
                        string (const string &); // string x = string ...
                        string & operator = (const char *);
                        string & operator = (const string &);
                        ~ string ();
                        char & operator [] (int i);
                        friend ostream & operator << (ostream &, const string
&);
                        friend istream & operator >> (istream &, string &);
                        friend int operator == (const string & x, const char * s)
                                    {return strcmp (xp-> s, s) == 0; }
                        friend int operator == (const string & x, const string &
y)
                                    {return strcmp (xp-> s, yp-> s) == 0; }
                        friend int operator! = (const string & x, const char * s)
                                    {return strcmp (xp-> s, s)! = 0; }
                        friend int operator! = (const string & x, const string &
y)
                                    {return strcmp (xp-> s, yp-> s)! = 0; }
};
```
Constructors and destructors are trivial:
```
string :: string ()
{
            p = new srep;
            p-> s = 0;
}

string :: string (const string & x)
{
            xp-> n ++;
            p = xp;
}

string :: string (const char * s)
{
            p = new srep;
            p-> s = new char [strlen (s) +1];
            strcpy (p- > s, s);
}

string :: ~ string ()
{
            if (--p-> n == 0) {
                        delete [] p-> s;
                        delete p;
            }
}
```
As always, assignment operations are similar to constructors. In them, you need to take care of removing the first operand that sets the left side of the assignment:
```
string & string :: oper ator = (const char * s)
{
```

```
            if (p-> n> 1) { // detach from the old line
                        p-> n--;
                        p = new srep;
            }
            else // free the line with the old value
                        delete [] p-> s;
            p-> s = new char [strlen (s) +1];
            strcpy (p-> s, s);
            return * this;
   }

   string & string :: o perator = (const string & x)
   {
            xp-> n ++; // protection against the case of `` st = st "

            if (--p-> n == 0) {
                        delete [] p-> s;
                        delete p
            }
            p = xp;
            return * this;
   }
```

The output operation shows how the reference count is used. It echoes each line that is entered (input is done using the << operation below):

```
   ostream & operator << (ostream & s, const string & x)
   {
            return s << xp-> s << "[" << xp-> n << "] \ n";
   }
```

The input operation takes place using the standard function for entering a character string ($$ 10. 3.1):

```
   istream & operator >> (istream & s, string & x)
   {
            char buf [256];
            s >> buf; // unreliable: buf may overflow
                                                          // see $$ 10.3.1 for
   the correct solution
```

```
            x = buf;
            cout << "echo:" << x << '\ n';
            return s;
   }
```

The indexing operation is needed to access individual characters. The index is controlled by:

```
   void error (const char * p)
   {
            cerr << p << '\ n';
            exit (1);
   }

   char & string :: operator [] (int i)
   {
   if (i <0 || strlen (p-> s) <i) error ("invalid index value");
            return p-> s [i];
   }
```

The main program just gives a few examples of using string operations. Words from the input stream are read into lines, and then the lines are printed. This continues until the done line is found, or there are no more lines to write words, or the input stream ends. Then all lines are printed in reverse order and the program ends.

```
   int main ()
   {
            string x [100];
            int n;
            cout << "start here \ n";
            for (n = 0; cin >> x [n]; n ++) {
                     if (n == 100) {
                              error ("too many words");
                              return 99;
                     }
                     string y;
                     cou t << (y = x [n]);
                     if (y == "done") break;
            }
            cout << "now we go backward by words \ n";
            for (int i = n-1; 0 <= i; i--) cout << x [i];
                        return 0;
   }
```

7.12 Friends and Members

Finally, we can discuss when to use member functions when accessing the private part of a user-defined type, and when to use friend functions. Some functions, such as constructors, destructors, and virtual functions ($$ R.12), must be members, but others have a choice. Because we are not introducing a new global name when describing a function as a member, member functions should be used unless there are other reasons.

Consider a simple class X:

```
   class X {
            // ...
            X (int);
            int m1 ();
            int m2 () const;
            friend int f1 (X &);
            friend int f2 (const X &);
            friend int f3 (X);
   };
```

First, we point out that the X :: m1 () and X :: m2 () members can be called only for objects of class X. The X (int) transformation will not be applied to the object for which X :: m1 () or X :: m2 ():

```
   void g ()
   {
            1.m1 (); // error: X (1) .m1 () not used
            1.m2 (); // error: X (1) .m2 () not used
   }
```

The global function f1 () has the same property ($$ 4.6.3), since its parameter is a reference without a const specification. The situation is different with the functions f2 () and f3 ():

```
   void h ()
   {
            f1 (1); // error: f1 (X (1)) not used
```

```
        f2 (1); // ok: f2 (X (1));
        f3 (1); // ok: f3 (X (1));
}
```

Therefore, an operation that changes the state of a class object must be a member or a global function with a reference parameter without a const specification. Operations on basic types that require addresses (=, *, ++, etc.) as operands are naturally defined as members for user-defined types.

Conversely, if an implicit type conversion is required for all operands of an operation, then the function that implements it must be not a member, but a global function and have a reference type parameter with the const specification or a nonreference parameter. This is usually the case with functions that implement operations that, for basic types, do not require addresses as operands (+, -, ||, etc.).

If the type conversion operations are not defined, then there is no compelling reason for a member function over a friend function with a reference parameter, and vice versa. It happens that a programmer just likes one form of call recording more than another. For example, many people prefer inv (m) rather than m.inv () to denote the function of inverting a matrix m. Of course, if inv () inverts the matrix m itself, and does not return the new inverse of m, then inv () must be a member.

All other things being equal, it is best to stick with a member function. Such arguments can be made. There is no guarantee that the call operation will not be defined someday. It cannot be guaranteed in all cases that future changes will not entail changes in the state of the object. Writing a member function call makes it clear to the programmer that an object can be modified, while writing with a reference parameter is far from obvious. Further, the expressions allowed in a member function can be substantially shorter than the equivalent expressions in a global function. A global function must use explicitly specified parameters, while a member function can implicitly use the this pointer. Finally, because member names are not global names, they tend to be shorter than global function names.

7.13 Cautions

Like any other language facility, operation overloading can be used wisely and unwisely. In particular, the opportunity to give new meaning to ordinary operations can be used in such a way that the program is completely incomprehensible. Imagine how the reader would feel if you redefined the + operation in your program to denote subtraction. The overloading mechanism described here will protect the programmer and user from such recklessness. Therefore, the programmer cannot change the meaning of operations on basic data types such as int, nor the syntax of expressions and the priority of operations for them.

It seems like it makes sense to use operation overloading to emulate the traditional use of operations. Recording with a regular function call can be used in cases when a traditional recording with a basic operation does not exist, or when the set of operations that can be overloaded is not sufficient to record the required actions with it.

7.14 Exercises

1. (* 2) Define an iterator for the string class. Define a concatenation operator + and an operator + = meaning "append to end of string". What other operations would you like and be able to define for this class?
2. (* 1.5) Define a substringing operation for a string class using overloading ().
3. (* 3) Define the string class so that the substring operation can be applied to the left side of the assignment. First, write a variation in which a string can be assigned to a substring of the same length, and then a variation with different string lengths.
4. (* 2) Design the string class in such a way that its objects are treated as values when passing parameters and assigning them , i.e. so that the string representations themselves are copied in the string class , not just control structures.
5. (* 3) Modify the string class from the previous exercise so that strings are copied only when needed. This means that you need to keep one common representation of two identical strings until one of them changes. Do not try to define a substring operation that can be applied to the left side of the assignment at the same time .
6. (* 4) Define a string class that has the properties listed in the previous exercises: its objects are treated as values, copying is deferred (that is, it happens only when necessary) and the substring operation can be applied to the left side of the assignment.

7. (* 2) What type conversions are used in the expressions of the following program?

```
struct X {
        int i;
        X (int);
        operator + (int);
};

struct Y {
        int i;
        Y (X);
        operator + (X);
        operator int ();
};

extern X operator * (X, Y);
extern int f (X);

X x = 1;
Y y = x;
int i = 2;

int main ()
{
        i + 10; y + 10; y + 10 * y;
        x + y + i; x * X + i; f (7);
        f (y); y + y; 106 + y;
}
```

Define X and Y as integer types. Modify the program so that it can be executed and it will print the values of all the correct expressions.

8. (* 2) Define an INT class that will be equivalent to int. Hint: define a function INT :: operator int ().

9. (* 1) Define a RINT class that is equivalent to int, except that only the following operations are allowed: + (unary and binary), - (unary and binary), *, / and%. Hint: You don't need to define RINT :: operator int ().

10. (* 3) Define a LINT class equivalent to the RINT class, but it must use at least 64 bits to represent an integer .

11. (* 4) Define a class that implements arbitrary precision arithmetic . Hint: You will have to use memory like you do in the string class.

12. (* 2) Write a program that, thanks to macros and overloading, makes it impossible to understand. Tip: define INT + as - and vice versa; use a macro to specify int as INT. In addition, a lot of confusion can be created by overriding well-known functions and using link type parameters and providing misleading comments.

13. (* 3) Exchange the solutions for exercise [12] with your friend. Try to understand what his program is doing without launching it. If you do this exercise, it will become clear to you what to avoid.

14. (* 2) Rewrite examples with classes complex ($$ 7.3), tiny ($$ 7.3.2) and string ($$ 7.11) without using friendly functions. Use only member functions. Check out the new versions of these classes. Compare these to the versions that use friendly functions. Refer to Exercise 5.3.

15. (* 2) Define type vec4 as a vector of four floating point numbers . Define the operator [] function for it. For combinations of vectors and floating point numbers, define the operations: +, -, *, /, =, + =, - =, * =, and / =.

16. (* 3) Define the class mat4 as a vector of four elements of type vec4. Define a function operator [] for it that returns vec4. Define normal matrix operations for this type. Define a function in mat4 that produces a Gaussian transform with a matrix.

17. (* 2) Define a vector class similar to vec4, but here the vector size must be specified as a parameter to the vector :: vector (int) constructor .

18. (* 3) Define a matrix class similar to mat4, but here the dimensions of the matrix must be specified as parameters to the matrix :: matrix (int, int) constructor .

19. (* 3) Complete the CheckedPtrToT class definition from $$ 7.10 and test it. For the definition of this class to be complete, it is necessary to define at least the following operations: *, ->, =, ++,

and -. Do not throw a dynamic error until you actually access a null pointer.

20. (* 1.5) Rewrite the $$ 7.7 word counting example so that it does not have a predefined maximum word length .

CHAPTER 8. TYPE TEMPLATES

Here is your quote
- Bjorn Stroustrup

This chapter introduces the concept of a type template. It makes it easy to define and implement without sacrificing program execution efficiency and, without abandoning static type checking, container classes such as lists and associative arrays. In addition, type templates allow you to define generic (generic) functions for an entire family of types at once , such as sort. The family of list classes is given as an example of a type template and its relationship with other language constructs . Several variants of the sort () boilerplate function are provided to show how to derive a program from largely independent parts . Finally, a simple type template for an associative array is defined and shown in two small demo programs how to use it.

8.1 Introduction

One of the most useful kinds of classes is the container class, i.e. a class that stores objects of some other types. Lists, arrays, associative arrays, and sets are all container classes. Using the facilities described in Chapters 5 and 7, you can define a class as a container of objects of a single, known type. For example, $$ 5.3.2 defines a set of integers. But container classes have the interesting property that the type of objects they contain does not really matter for the creator of the container, but for the user of a particular container, this type is essential. Therefore, the type of the contained objects must be a parameter of the container class, and the creator of such a class will define it using the type-parameter. For each specific container (i.e. an object of a container class), the user will specify what type of objects it contains. An example of such a container class was the Vector template from $$ 1.4.3.

This chapter explores a simple stack pattern and introduces the concept of a templated class. Then, more complete and plausible examples of several related type patterns for a list are considered. Template functions are introduced and rules are formulated that can be a parameter of such functions. Finally, a type template for an associative array is provided.

8.2 Simple type pattern

The type template for a class specifies how individual classes are constructed, just as a class definition specifies how its individual objects are constructed. You can define a stack containing elements of arbitrary type:

```
template <class T>
class stack {
        T * v;
        T * p;
        int sz;
        public:
                stack (int s) {v = p = new T [sz = s]; }
                ~ stack () {delete [] v; }
                void push (T a) {* p ++ = a; }
                T pop () {return * - p; }
                int size () const {return pv; }
};
```

For simplicity, dynamic error control has not been considered. Apart from this, the example is complete and quite plausible. The template <class T> prefix indicates that a type template with a type parameter T is being described, and that this notation will be used in the following description. Once the identifier T is specified in the prefix, it can be used like any other type name. The scope of T continues until the end of the declaration, which begins with the template <class T> prefix. Note that in the prefix, T is declared a type and does not have to be a class name. So, below in the description of the object sc, the type T turns out to be just char.

The template class name followed by the type enclosed in angle brackets <> is the class name (defined by the type template) and can be used like all class names. For example, the following defines an object sc of class stack <char>:

```
stack <char> sc (100); // character stack
```

Apart from the special notation of the name, the stack <char> class is completely equivalent to the class defined like this:

```
class stack_char {
        char * v;
        char * p;
        int sz;
        public:
```

```
                         stack_char (int s) {v = p = new char [ sz = s]; }
                        ~ stack_char () {delete [] v; }
                        void push (char a) {* p ++ = a; }
                        char pop () {return * - p; }
                        int size () const {return pv; }
   };
```

You might think that a type template is a tricky macro definition that obeys C ++ naming, type, and scoping rules. This is, of course, a simplification, but it is such a simplification that it helps to avoid big misunderstandings. In particular, the use of a type template does not imply any dynamic support beyond those used for normal "manual" classes. Nor should we think that it will shorten the program.

It usually makes sense to first debug a specific class, such as stack_char, before building a template like stack <T> from it. On the other hand, to understand the type pattern, it is useful to imagine its action on a particular type, such as int or shape *, before trying to represent it in general terms.

With the template class stack defined, you can define and use different stacks as follows:

```
   stack <shape *> ssp (200); // stack of pointers to shapes
   stack <Point> sp (400); // stack of Point structures
   void f (stack <complex> & sc) // parameter of type `reference to
                                                                          // complex '
   {
            sc.push (complex (1,2));
            complex z = 2.5 * sc.pop ();
            stack <int> * p = 0; // pointer to the stack of integers
            p = new stack <int> (800); // the stack of integers is
   allocated
                                                                          // in free memory
            for (int i = 0; i <400; i ++) {
                        p-> push (i);
                        sp.push (Point (i, i + 400));
            }
            // ...
   }
```

Since all member functions of the stack class are substitutions, and in this example, the translator creates function calls only for allocation in free memory and freeing.

Functions in a type template may or may not be substitutions, the stack template class can rightfully be defined like this:

```
template <class T> class stack {
            T * v;
            T * p;
            int sz;
            public:
                        stack (int);
                        ~ stack ();
                        void push (T);
                        T pop ();
                        int size () const;
};
```

In this case, the definition of the stack member function must be given elsewhere, as was the case for the member functions of regular, non-template classes. Such functions are also parameterized with a type that serves as a parameter to their template class, so they are defined using a type template for the function. If this happens outside the template class, it must be done explicitly:

```
template <class T> void stack <T> :: push (T a)
{
            * p ++ = a;
}

template <class T> stack <T> :: stack (int s)
{
            v = p = new T [sz = s];
}
```

Note that within the scope of the name stack <T>, the qualification <T> is redundant, and stack <T> :: stack is the name of the constructor.

It is the programming system's job, not the programmer's, to provide versions of the template functions for each actual parameter of the type template. Therefore, for the above example, the programming system must create constructor definitions for the classes stack <shape *>, stack <Point> and stack <int>, destructors for stack <shape *> and stack <Point>, versions of the push () functions for stack < complex>, stack <int>, and stack <Point>, and the stack <complex> version of pop (). Such generated functions will be perfectly normal member functions, for example:

```
void stack <complex> :: push (complex a) {* p ++ = a; }
```

Here the difference from a regular member function is only in the form of the class name. Just as there can be only one class member function definition in a program, there can be only one type template definition for a template class member function. If you need to define a member function of a template class for a specific type, then the task of the programming system is to find a type template for this member function and create the required version of the function. In general, the programming system can rely on instructions from the programmer to help find the type template you want.

It is important to structure the definition of a type template in such a way that its dependence on global data is minimal. The point is, the type template will be used to generate functions and classes based on a previously unknown type and in unknown contexts. Almost any, even weak, dependence on context can manifest itself as a problem when debugging a program by a user who, most likely, was not the creator of the type template. The advice to avoid using global names as much as possible should be taken especially seriously when designing a type template.

8.3 Type templates for a list

In practice, when designing a class that serves as a collection of objects, you often have to consider the relationship of the classes used in the implementation, memory management, and the need to define an iterator over the contents of the collection. It often happens that several related classes are developed together ($$ 12.2). As an example, we propose a family of classes representing singly linked lists and type templates for them.

8.3.1 List with forced link

Let's start by defining a simple list that assumes that every object that is listed has a link field. Then this list will be used as a building material for creating more general lists, in which the object does not need to have a link field. At first, only the general part will be given in the descriptions of the classes, and the implementation will be given in the next section. This is done so that class design issues are not obscured by their implementation details.

Let's start with a slink type that defines a link field in a singly linked list:

```
struct slink {
            slink * next;
            slink () {next = 0; }
            slink (slink * p) {next = p; }
};
```

Now you can define a class that can contain objects of any slink-derived class:

```
class slist_base {
            // ...
            public:
                        int insert (slink *); // add to the beginning of the
list
                        int append (slink *); // add to the end of the list
                        slink * get (); // remove and return the beginning of the
list
                        // ...
};
```

This class can be called a forced link list because it can only be used when all elements have a slink field that is used as a pointer to slist_base. The very name slist_base (base singly linked list) says that this class will be used as the base for singly linked list classes. As usual, when designing a family of related classes, the question arises of how to choose names for the various members of the family. Since class names cannot be overloaded, as is done for function names, overloading will not help us to curb name multiplication.

The slist_base class can be used like this:

```
void f ()
{
            slist_base slb;
            slb.insert (new slink);
```

```
        // ...
        slink * p = slb.get ();
        // ...
        delete p;
}
```
But since the slink structure cannot contain any information other than a link, this example is not very interesting. To use slist_base, you must define a useful slink-derived class. For example, the translator uses the program tree nodes name, which have to be linked into a list:
```
class name: public slink {
        // ...
};

void f (const char * s)
{
        slist_base slb;
        slb.insert (new name (s));
        // ...
        name * p = (name *) slb.get ();
        // ...
        delete p ;
}
```
Everything is fine here, but since the definition of the slist_base class is given through the slink structure, you have to use an explicit cast to convert the slink * value returned by slist_base :: get () to name *. It's not beautiful. For a large program with many lists and slink-derived classes, this is also error prone. We could use a type-safe version of the slist_base class:
```
template <class T>
class Islist: private slist_base {
        public:
                void insert (T * a) {slist _base :: insert (a); }
                T * get () {return (T *) slist_base :: get (); }
                // ...
};
```
Casting in the Islist :: get () function is perfectly justified and reliable, because the Islist class guarantees that every object in the list is indeed of

type T or a type derived from the class T. Note that slist_base is a private Islist base class. We don't want the user to accidentally come across unreliable implementation details.

The name Islist (intrusive singly linked list) denotes an intrusive singly linked list. This type template can be used like this:

```
void f (const char * s)
{
            Islist <name> ilst;
            ilst.insert (new name (s));
            // ...
            name * p = ilst.get ();
            // ...
            delete p
}
```

Attempts to use it incorrectly will be revealed at the broadcast stage:

```
class exp r: public slink {
            // ...
};

void g (expr * e)
{
            Islist <name> ilst;
            ilst.insert (e); // error: Islist <name> :: insert (),
                                                            // but you need name
*
            // ...
}
```

There are several important points to note about our example. First, the solution is type-safe (trivial errors are prevented in a very limited part of the program, namely, in the access functions from Islist). Second, type reliability is achieved without increasing time and memory costs, since Islist accessors are trivial and are implemented by substitution. Third, since all the real work with the list is done in the implementation of the slist_base class (not yet presented), there is no duplication of functions, and the source code of the implementation, i.e. the slist_base function should not be accessible to the user at all. This can be significant in commercial use of list utilities. In addition, a separation between the interface and its implementation is achieved, and it becomes possible to change the implementation without re-translating user programs. Finally, a simple forced-link list is close in memory and time usage to an optimal solution. In other words, this approach is close to optimal in terms of time, memory, data hiding and type control, and at the same time, it provides greater flexibility and compactness of expressions.

Unfortunately, an object can only be added to Islist if it derives from slink. It means that you cannot have a list Islist of values of type int, you cannot create a list of values of some previously defined type that is not derived from slink. In addition, you will have to try to include the object in the two Islists ($$ 6.5.1).

8.3.2 List without forced link

After a "digression" into the issues of building and using a list with forced linkage, let's move on to building lists without forced linking. This means that the elements of the list do not have to contain additional information to assist in the implementation of the list class. Since we can no longer expect that the object in the list has a link field, such a link must be provided in the implementation:

```
template <class T>
struct Tlink: public slink {
            T info;
            Tlink (const T & a): info (a) {}
};
```

The Tlink <T> class stores a copy of objects of type T in addition to the link field that comes from its base class slink. Note that the initializer of the form info (a) is used, not the assignment info = a. This is essential for the efficiency of the operation in the case of types with non-trivial copy constructors and assignment operations ($$ 7.11). For such types (for example, for String), defining the constructor as

```
Tlink (const T & a) {info = a; }
```

we get that a standard String object will be built, and only then it will be assigned a value. With a class defining a link and a class Islist, getting the definition of a list without forcing a link is quite simple:

```
template <class T>
```

```
class Slist: private slist_base {
        public:
                void insert (const T & a)
                        {slist_base :: insert (new Tlink <T> (a)); }
                void append (const T & a)
                        {slist_base :: append (new Tlink <T> (a)); }
                T get ();
                // ...
};

template <class T>
T Slist <T> :: get ()
{
        Tlink <T> * lnk = (Tlink <T> *) slist_base :: get ();
        T i = lnk-> info;
        delete lnk;
        return i;
}
```

Working with a Slist is as easy as working with an Ilist. The difference is that you can include an object in a Slist whose class is not derived from slink, and you can also include one object in two lists:

```
void f (int i)
{
        Slist <int> lst1;
        Slist <int> lst2;
        lst1.insert (i);
        lst2.insert (i);
        // ...
        int i1 = lst1.get ();
        int i2 = lst2.get ();
        // ...
}
```

However, a forced-link list, such as Islist, allowed for a much more efficient program and gave a more compact presentation. Indeed, every time an object is included in the Slist, a Tlink object must be placed, and every time an object is removed from a Slist, a Tlink object must be deleted, each time copying an object of type T. When this overhead problem arises, two tricks can help. First, Tlink is a direct candidate for placement with a near-optimal special-purpose placement function (see $$ 5.5.6). Then the additional costs of the program will be reduced to the usually acceptable level. Secondly, such a technique is useful when objects are stored in a "primary" list that has a forced link, and lists without a forced link are used only when it is required to include an object in several lists:

```
void f (name * p)
{
        Islist <name> lst1;
        Slist <name *> lst2;
        lst1.insert (p); // communication via object `* p '
        lst2.insert (p); // used to store `p '
                                        // separate object of
type list
                                        // ...
}
```

Of course, such tricks can only be done in a separate component of the program to avoid confusion of list types in the interfaces of various components. But this is exactly the case when, for the sake of efficiency and compactness of the program, they are worth going for.

Because the Slist constructor copies the parameter to insert (), the Slist is only suitable for objects as small as integers, complex numbers, or pointers. If for objects copying is too expensive or unacceptable for semantic reasons, usually the solution is to put pointers to them in the list instead of objects. This is done in the f () function above for lst2.

Note that once the parameter for Slist :: insert () is copied, passing an object of a derived class to an insert () function expecting a base class object will not go as smoothly as one might (naively) think:

```
class smiley: public circle { / * ... * /};
void g1 (Slist <circle> & olist, const smiley & grin)
{
        olist.insert (grin); // trap!
}
```

Only the circle portion of the smiley object will be included in the list. Note that this trouble will be detected by the translator in the most likely case.

So, if the base class in question were abstract, the translator would prohibit "truncation" of the derived class object:

```
void g2 (Slist <shape> & olist, const circle & c)
{
            olist.insert (c); // error: trying to create an object
                                                          // abstract class
}
```

To avoid "truncating" the object, you need to use pointers:

```
void g3 (Slist <shape *> & plist, const smiley & grin)
{
            olist.insert (& grin); // perfectly
}
```

You don't need to use a reference parameter for the template class:

```
void g4 (Slist <shape &> & rlist, const smiley & grin)
{
            rlist.insert (grin); // error: commands will be created,
                                                          // containing a link to
the link (shape &&)
}
```

When generated from a type template, references used in this manner will result in type errors. Generating from a template type for a function

```
Slist :: insert (T &);
```

will result in an invalid function

```
Slist :: insert (shape &&);
```

A link is not an object, so you cannot have a link to a link.

Since a pointer list is a useful construct, it makes sense to give it a special name:

```
template <class T>
class Splist: private Slist <void *> {
            public:
                        void insert (T * p) {Slist <void *> :: insert (p); }
                        void append (T * p) {Slist <void *> :: append (p); }
                        T * get () {return (T *) Slist <void *> :: get (); }
};
```

```
class Isplist: private slist_base {
        public:
                void insert (T * p) {slist_base :: insert (p); }
                void append (T * p) {slist_base :: append (p); }
                T * get () {return (T *) slist_base :: get (); }
};
```

These definitions also improve type checking and further reduce the need to duplicate functions.

It is often useful for the type of element specified in a type template to be itself a templated class. For example, a sparse matrix containing dates can be defined like this:

```
typedef Slist <Slist <date>> dates;
```

Note the presence of spaces in this definition. If there are no spaces between the first and second angle brackets>, a syntax error occurs because the >> in the definition

```
typedef Slist <Slist <date >> dates;
```

will be treated as a right shift operation. As usual, the name you enter in a typedef is a synonym for the type it denotes, rather than a new type. The typedef construct is useful for naming long template class names as well as it is useful for any other long type name.

Note that a template parameter of a type that can be used in different ways in its definition must still be specified once among the list of template parameters. Therefore, a type template that uses an object T and a list of T elements must be defined like this:

```
template <class T> class mytemplate {
        T ob;
        Slist <T> slst;
        // ...
};
```

but not at all:

```
template <class T, class Slist <t>> class mytemplate {
        T obj;
        Slist <T> slst;
        // ...
};
```

$$ 8.6 and $$ R.14.2 give rules on what can be a type template parameter.

8.3.3 List implementation

The implementation of the slist_base functions is straightforward. The only difficulty comes with error handling. For example, what to do if the user tries to take an element from an empty list using the get () function. Similar situations are dealt with in the error handling function slist_handler (). A more advanced method tailored to specific situations will be discussed in Chapter 9.

Here is a complete description of the slist_base class:

```
class slist_base {
        slink * last; // last-> next is the start of the list
        public:
                void insert (slink * a); // add to the beginning of the
list
                void append (slink * a); // add to the end of the
list
                slink * get (); // delete and return
                                                // start of the list
                void clear () {last = 0; }
                slist_base () {last = 0; }
                slist_base (slink * a) {last = a-> next = a; }
                friend class slist_base_iter;
};
```

To simplify the implementation of both insert and append, a pointer to the last element of the closed list is stored:

```
void slist_base_insert (slink * a) // add to the beginning of the list
{
        if (last)
                a-> next = last-> next;
        else
                last = a;
                last-> next = a;
}
```

Note that last-> next is the first item in the list.

```
void slist_base :: append (slink * a) // add to the end of the list
{
        if (last) {
                a-> next = last-> next;
                last = last-> next = a;
        }
        else
                last = a-> next = a;
}

slist * slist_base :: get () // remove and return the beginning of the list
{
        if (last == 0)
                slist_handler ("cannot be taken from an empty list");
        slink * f = last-> next;
        if (f == last)
                last = 0;
        else
                last-> next = f-> next;
        return f ;
}
```

A more flexible solution is possible when slist_handler is a function pointer rather than the function itself. Then call

```
slist_handler ("cannot be taken from an empty list");
```

will be asked like this

```
(* slist_handler) ("cannot be taken from an empty list");
```

As we have already done for the new_handler function ($$ 3.2.6), it is useful to create a function that will help the user create their own error handlers:

```
typedef void (* PFV) (const char *);
Pfv set_slist_handler (pfv a)
{
        PFV old = slist_handler;
        slist_handler = a;
        return old;
}
```

```
PFV slist_ handler = & default_slist_handler;
```

The special situations discussed in Chapter 9 not only provide an alternative way to handle errors, but also a way to implement slist_handler.

8.3.4 Iteration

The slist_base class does not have any functions for viewing the list, you can only insert and delete elements. However, it describes the slist_base_iter class as a friend, so you can define an iterator suitable for the list. Here is one possible, given in the style shown in $$ 7.8:

```
class slist_base_iter {
        slink * ce; // current item
        slist_base * cs; // current list
        public:
                inline slist_base_iter (slist_base & s);
                inline slink * operator () ()
};

slist_base_iter :: slist_base_iter (slist_base & s)
{
        cs = & s;
        ce = cs-> last;
}

slink * slist_base_iter :: operator () ()
// returns 0 when the iteration ends
{
        slink * ret = ce? (ce = ce-> next): 0;
        if (ce == cs-> last) ce = 0;
        return ret;
}
```

Based on these definitions, it is easy to get iterators for Slist and Islist. First, you need to define friendly classes for iterators for the corresponding container classes:

```
template <class T> class Islist_iter;
template <class T> class Islist {
            friend class Islist_iter <T>;
            // ...
};

template <class T> class Slist_iter;

template <class T> class Slist {
            friend class Slist_iter <T>;
            // .. .
};
```
Note that the names of the iterators appear without defining their template class. It is a way of defining a type template under interdependency.

Now you can define the iterators themselves:
```
template <class T>
class Islist_iter: private slist_ base_iter {
            public:
                        Islist_iter (Islist <T> & s): slist_base_iter (s) {}
                        T * operator () ()
                                    {return (T *) slist_base_iter :: operator () (); }
};
template <class T>
class Slist_iter: private slist_base_iter {
            public:
                        Slist_iter (Slist <T> & s): slist_base_iter (s) {}
                        inline T * operator () ();
};
T * Slist_iter :: operator () ()
{
            return ((Tlink <T> *) slist_base_iter :: operator () ()) -> info;
}
```
Note that we again used the technique of building a family of derived classes from a single base class (namely, a template class). We use inheritance to express the generality of classes and avoid unnecessary duplication of functionality. It is difficult to overestimate the desire to avoid

duplication of functions when implementing such simple and frequently used classes as lists and iterators. You can use these iterators like this:

```
void f (name * p)
{
            Islist <name> lst1;
            Slist <name> lst2;
            lst1.insert (p);
            lst2.in sert (p);
            // ...
            Islist_iter <name> iter1 (lst1);
            const name * p;
            while (p = iter1 ()) {
                        list_iter <name> iter2 (lst1);
                        const name * q;
                        while (q = iter2 ()) {
                                    if (p == q) cout << "found" << * p << '\ n';
                        }
            }
}
```

There are several ways to define an iterator for a container class. The developer of a program or library should choose one and stick to it. This method may seem too tricky. In a simpler version, one could simply rename operator () () as next (). Both assume a relationship between the container class and the iterator for it, so that when the iterator is executed, you can handle cases where items are added or removed from the container. This and some other ways of specifying iterators would not be possible if the iterator depended on a user function that has pointers to elements from the container. Typically, a container or its iterators implement the concept of "set iteration to start" and the concept of "current item".

If the concept of the current element is provided not by an iterator, but by the container itself, iteration is forced in relation to the container, in the same way as the link fields are forced to be stored in objects from the container. This means that it is difficult to simultaneously conduct two iterations for one container, but the memory and time costs for such an organization of iteration are close to optimal. Let's give an example:

```
class slist_base {
```

```
            // ...
            slink * last; // last-> next head of the list
            slink * current; // current item
            public:
                        // ...
                        slink * head ( ) {return last? last-> next: 0; }
                        slink * current () {return current; }
                        void set_current (slink * p) {current = p; }
                        slink * first () {set_current (head ()); return current; }
                        slink * next ();
                        slink * prev ();
   };
```

Just as for the sake of efficiency and compactness of the program, you can use both a forced and unlinked list for a single object, you can use forced and non-forced iteration for a single container:

```
   void f (Islist <name> & ilst)
   // slow search for duplicate names
   {
            list_iter <name> slow (ilst); // using an iterator
            name * p;
            while (p = slow ()) {
                        ilst.set_current (p); // count on the current item
                        name * q;
                        while (q = ilst.next ())
                                    if (strcmp (p-> string, q-> string) == 0)
                                                cout << "duplicate" << p << '\ n';
            }
   }
```

Another kind of iterators is shown in $$ 8.8.

8.4 Type templates for functions

Using templated classes means having templated member functions. In addition, you can define global template functions, i.e. type templates for functions that are not members of the class. A type template for functions generates a family of functions just as a type template for a class generates a family of classes. We will discuss this feature in a sequence of examples that show variants of the sort () function. Each of the options in the following sections will illustrate a general method.

As usual, we will focus on organizing the program, not developing its algorithm, so a trivial algorithm will be used. All variants of the type template for sort () are needed to show the capabilities of the language and useful programming techniques. The options are not ordered according to how good they are. In addition, you can discuss traditional options without type templates, in particular, passing a pointer to a function that does the comparison.

8.4.1 Simple type template for a global function

Let's start with the simplest template for sort ():

```
template <class T> void sort (Vector <T> &);
void f (Vector <int> & vi,
                              Vector <String> & vc,
                              Vector <int> & vi2,
                              Vector <char *> & vs)
{
        sort (vi); // sort (Vector <int> & v);
        sort (vc); // sort (Vector <String> & v);
        sort (vi2); // sort (Vector <int> & v);
        sort (vs); // sort (Vector <char *> & v);
}
```

Which sort () function will be called is determined by the actual parameter. The programmer defines the type template for the function, and the task of the programming system is to ensure that the correct variants of the function are created from the template and the corresponding variant is called. For example, a simple bubble-sort pattern could be defined like this :

```
template <class T> void sort (Vector <T> & v)
/ *
        Sort items in ascending order
        Sorting by the bubble method is used
* /
{
unsigned n = v.size ();
for (int i = 0; i <n-1; i ++)
```

```
            for (int j = n-1; i <j; j--)
                    if (v [j] <v [j-1]) {// swap v [j] and v [j-1]
                            T temp = v [j];
                            v [j] = v [j-1];
                            v [j-1] = temp;
                    }
  }
```

We advise you to compare this definition with a sorting function with the same algorithm from $$ 4.6.9. The essential difference between this option is that all the necessary information is transferred in a single parameter v. Since the type of the elements to be sorted is known (from the type of the actual parameter, you can compare the elements directly, rather than pass a pointer to the comparison function. Also, there is no need to fiddle with the sizeof operation. This solution seems prettier and more efficient than the usual However, it does run into a difficulty: for some types the <is undefined, while for others, such as char *, its definition contradicts what is required in the above definition of the template function. (Indeed, we need to compare not pointers to strings, but In the first case, an attempt to create a variant of sort () for such types will fail (which is to be expected), and in the second, a function appears that produces unexpected results.

To correctly sort a vector of char * elements, we can simply define ourselves a suitable function definition

```
  sort (Vector <char *> &):
  void sort (Vector <char *> & v)
  {
            unsigned n = v.size ();
                    for (int i = 0; i <n-1; i ++)
                            for (int j = n-1; i <j; j--)
                                    if (strcmp (v [j], v [j-1]) <0) {
                                            // swap v [j] and v [j-1]
                                            char * temp = v [j];
                                            v [j] = v [j-1];
                                            v [j-1] = temp;
                                    }
  }
```

Since for vectors from pointers to strings the user has given his own special definition of the sort () function, it will be used, and there is no need to create a definition for it using a template with a parameter of the Vector <char *> & type. The ability to define a templated function for critical or "unusual" types provides a valuable coding flexibility and can be an important means of improving program performance.

8.4.2 Derived classes allow you to introduce new operations

In the previous section, the comparison function was "built in" in the body of sort () (just using the <operation). Another solution is possible when the Vector templated class itself provides it. However, such a solution only makes sense if a meaningful notion of comparison is possible for the element types. Typically, in this situation, sort () is only defined for vectors on which the <operation is defined:

```
template <class T> void sort (SortableVector <T> & v)
{
        unsigned n = v.size ();
        for (int i = 0; i <n-1; i ++)
                for (int j = n-1; i <j; j--)
                        if (v.lessthan (v [j], v [j-1])) {
                                // swap v [j] and v [j-1]
                                T temp = v [j];
                                v [j] = v [j-1];
                                v [j-1] = temp;
                        }
}
```

The SortableVector class can be defined like this:

```
template <class T> class SortableVector
        : public Vector <T>, public Comparator <T> {
        public:
                SortableVector (int s): Vector <T> (s) {}
};
```

For this definition to make sense, you also need to define the Comparator template class:

```
template <class T> class Comparator {
        public:
```

```
                        inline static lessthan (T & a, T & b) // function "less"
                                {return strcmp (a, b) <0; }
                        // ...
    };
```

To eliminate the effect that, in our case, the <operation gives the wrong result for the char * type, we define a special version of the comparator class:

```
    class Comparator <char *> {
            public:
                        inline static lessthan (const char * a, const char * b)
                        // function "less"
                        {return strcmp (a, b) <0; }
                        // ...
    };
```

The description of a special templated class for char * is completely similar to how in the previous section we defined a special templated function for the same purpose. For the description of a custom templated class to work, the translator must detect it before using it. Otherwise the class that is generated from the template will be used. Since a class must have exactly one definition in the program, it would be a mistake to use both the custom version of the class and the version generated from the template.

Since we already have a special version of the Comparator class for char *, a special version of the SortableVector class for char * is not needed, and we can finally try sorting:

```
    void f (SortableVector <int> & vi,
                                So rtableVector <String> & vc,
                                SortableVector <int> & vi2,
                                SortableVector <char *> & vs)
    {
            sort (vi);
            sort (vc);
            sort (vi2);
            sort (vs);
    }
```

It is possible to have two kinds of vectors and not very good, but at least SortableVector derives from Vector. This means that if the function does not need sorting, then you do not need to know about the SortableVector class in it, and where necessary, the implicit conversion of a reference to a derived class into a reference to a common base class will work. We introduced a Vector and Comparator-derived SortableVector class (instead of adding functions to a Vector-derived class) simply because the Comparator class suggested itself in the previous example. This approach is typical when building large libraries. The Comparator class is a natural candidate for a library because it can specify different requirements for comparison operations for different types.

8.4.3 Passing operations as function parameters

Instead of specifying the comparison function as part of the Vector type, you can pass it as the second parameter to sort (). This parameter is an object of the class in which the implementation of the comparison operation is defined:

```
template <class T> void sort (Vector <T> & v, Comparator <T> & cmp)
{
        unsigned n = v.size ();
        for (int i = 0; i <n-1; i ++)
                for (int j = n-1; i <j; j--)
                        if (cmp.lessthan (v [j], v [j-1])) {
                                // swap v [j] and v [j-1]
                                T temp = v [j];
                                v [j] = v [j-1];
                                v [j-1] = temp;
                        }
}
```

This option can be viewed as a generalization of the traditional technique, when the comparison operation is passed as a pointer to a function. You can use it like this:

```
void f (Vector <int> & vi,
                        Vector <String> & vc,
                        Vector <int> & vi2,
                        Vector <char *> & vs)
{
        Comparator <int> ci;
        Comparator <char *> cs;
        Comparator <String> cc;
        sort (vi, ci); // sort (Vector <int> &);
        sort (vc, cc); // s ort (Vector <String> &);
        sort (vi2, ci); // sort (Vector <int> &);
        sort (vs, cs); // sort (Vector <char *> &);
}
```

Note that including the Comparator class as a parameter in the template ensures that the lessthan function will be implemented by substitution. This is particularly useful when a template function uses multiple functions rather than a single comparison operation, and is especially useful when those functions depend on data stored in the same object.

8.4.4 Implicit transfer of operations

In the example in the previous section, Comparator objects weren't actually used in calculations. These are just "artificial" parameters needed for proper type checking. The introduction of such parameters is a fairly general and useful technique, although not very nice. However, if the object is used only to transfer the operation (as it was in our case), i.e. neither the value nor the address of the object is used in the called function, then you can instead implicitly pass the operation:

```
template <class T> void sort (Vector <T> & v)
{
        unsigned n = v.size ();
        for (int i = 0; i <n-1; i ++)
                for (int j = n-1; i <j; j--)
                        if (Comparator <T> :: lessthan (v [j], v [j-1])) {
                                // swap v [j] and v [j-1]
                                T temp = v [j];
                                v [j] = v [j-1];
                                v [j-1] = temp;
                        }
}
```

As a result, we come to the original use case for sort ():

```
void f (Vector <int> & vi,
                        Vector <String> & vc,
                        Vector <int> & vi2,
                              Vector <char *> & vs)
   {
             sort (vi); // sort (Vector <int> &);
             sort (vc); // sort (Vector <String> &);
             sort (vi2); // sort (Vector <int> &);
             sort (vs); // sort (Vector <char *> &);
   }
```

The main advantage of this version, like the two previous ones, compared to the original version, is that the part of the program that is actually sorting is separated from the parts that contain operations that work with elements, such as lessthan. The need for such a separation grows with the growth of the program, and this separation is of particular interest when designing libraries. Here, the creator of the library cannot know the types of template parameters, and users do not know (or do not want to know) the specifics of the algorithms used in the template. In particular, if the sort () function were to use a more complex, optimized, and commercialized algorithm, the user would not be too eager to write his own special version for char *, as was done in $$ 8.4.1. Although implementing the Comparator class for the special char * case is trivial and can be used in other situations as well.

8.4.5 Injecting Operations Using Template Class Parameters

There are situations where the implicit relationship between the sort () templated function and the Comparator templated class creates difficulties. The implicit relationship is easy to overlook and at the same time, it can be difficult to understand. Also, because this relationship is built into the sort () function, it is not possible to use this function to sort vectors of one type if the comparison operation is calculated on a different type (see Exercise 3 in $$ 8.9). By placing the sort () function in a class, we can explicitly associate with the Comparator class:

```
   template <class T, class Comp> c lass Sort {
             public:
                       static void sort (Vector <T> &);
   };
```

You don't want to repeat the element type, and you can avoid it if you use typedef in the Comparator template:

```
   template <class T> class Comparator {
```

```
            public:
                        typedef TT; // definition of Comparator <T> :: T
                        static int lessthan (T & a, T & b) {
                                    return a <b;
                        }
            // ...
   };
```
In a special version for pointers to strings, this definition looks like this:
```
   class Comparator <char *> {
            public:
            typedef char * T;
            static int lessthan (T a, T b) {
                        return strcmp (a, b) <0 ;
            }
            // ...
   };
```
After these changes, you can remove the parameter specifying the element type from the Sort class:
```
   template <class T, class Comp> class Sort {
            public:
                        static void sort (Vector <T> &);
   };
```
Now you can use sorting like this:
```
   void f (Vector <int> & vi,
                                    V ector <String> & vc,
                                    Vector <int> & vi2,
                                    Vector <char *> & vs)
   {
            Sort <int, Comparator <int>> :: sort (vi);
            Sort <String, Comparator <String>>: sort (vc);
            Sort <int, Comparator <int>> :: sort (vi2);
            Sort <char *, Comparator <char *>> :: sort (vs);
   }
```
and define the sort () function like this:
```
   template <class T, class Comp>
```

```
void Sort <T, Comp> :: sort (Vector <T> & v)
{
        for (int i = 0; i <n-1; i ++)
                for (int j = n-1; i <j; j--)
                        if (Comp :: lessthan (v [j], v [j-1])) {
                                T temp = v [j];
                                v [j] = v [j-1];
                                v [j-1] = temp;
                        }
}
```

The last option clearly demonstrates how you can combine its individual parts into one program. This example can be simplified even further by using a comparator (Comp) class as the only template parameter. In this case, in the definitions of the Sort class and the Sort :: sort () function, the element type will be denoted as Comp :: T.

8.5 Overload Resolution for Template Function

No type conversion can be applied to the parameters of a template function . Instead, new function variants are created as needed:

```
template <class T> T sqrt (t);

void f (int i, double d, complex z)
{
        complex z1 = sqrt (i); // sqrt (int)
        complex z2 = sqrt (d); // sqrt (double)
        complex z3 = sqrt (z); // sqrt (complex)
        // ...
}
```

Here, for all three types of parameters, a custom sqrt function will be created according to the template. If the user wants something different, for example, call sqrt (double) with an int parameter, an explicit type conversion must be used:

```
template <class T> T sqrt (T);

void f (int i, double d, complex z)
{
```

```
            complex z1 = sq rt (double (i)); // sqrt (double)
            complex z2 = sqrt (d); // sqrt (double)
            complex z3 = sqrt (z); // sqrt (complex)
            // ...
}
```

In this example, only definitions for sqrt (double) and sqrt (complex) will be generated from the template.

A template function can be overloaded with a simple or a template function of the same name. Overloading both template and regular functions with the same name is resolved in three steps . These rules are too strict and are likely to be relaxed to allow for reference and pointer conversions, and possibly other standard conversions. As usual, unambiguity control will be in effect with such conversions.

1] Find a function with exact parameter matching ($$ R.13.2); if there is one, call it.

2] Find a type pattern by which you can create a callable function with exact parameter matching; if there is one, call it.

3] Try permission rules for ordinary functions ($$ r13.2); if the function is found according to these rules, call it, otherwise the call is an error.

In any case, if more than one function is found in the first step, the call is considered ambiguous and an error. For example:

```
template <class T>
            T max (T a, T b) {return a> b? A: b; };

void f (int a, int b, char c, char d)
{
            int m1 = max (a, b); // max (int, int)
            char m2 = max (c, d); // max (char, char)
            int m3 = max (a, c); // error: impossible
                                                    // create max (int,
char)
}
```

Since no type conversions are applied before generating the function from the template (rule [2]), the last call in this example cannot be resolved as max (a, int (c)). This can be done by the user himself by explicitly describing the function max (int, int). Then the rule [3] comes into force:

```
template <class T>
T max (T a, T b) {return a> b? A: b; }

int max (int, int);

void f (int a, int b, char c, char d )
{
        int m1 = max (a, b); // max (int, int)
        char m2 = max (c, d); // max (char, char)
        int m3 = max (a, c); // max (int, int)
}
```

The programmer does not need to define the function max (int, int), it will be created from a template by default.

You can define the max template to make the original example of our example work:

```
template <class T1, class T2>
T1 max (T1 a, T2 b) {return a> b? A: b; };

void f (int a, int b, char c, char d)
{
        int m1 = max (a, b); // int max (int, int)
        char m2 = max (c, d); // char max (char, char )
        int m3 = max (a, c); // max (int, char)
}
```

However, in C and C ++, the rules for built-in types and operations on them are such that it can be quite difficult to use such a template with two parameters. So, it may be wrong to set the function result type as the first parameter (T1), or at least it may lead to unexpected results, for example, to call

```
max (c, i); // char max (char, int)
```

If two parameters are used in the template for a function that can have many parameters with different arithmetic types, then the result will be too many definitions of different functions. It makes more sense to achieve type conversion by explicitly describing the function with the required types.

8.6 Type template parameters

The type template parameter does not have to be a type name (see $$ R.14.2). In addition to type names, you can specify strings, function names, and constant expressions. Sometimes you need to specify an integer as a parameter:

```
template <class T, int sz> class buffer {
        T v [sz]; // buffer of objects of arbitrary type
        // ...
};

void f ()
{
        buffer <char, 128> buf1;
        buffer <complex, 20> buf2;
        // ...
}
```

We made sz a parameter of the buffer template, and not of its objects, which means that the size of the buffer must be known at translation stage so that its objects can be allocated without using free memory. Because of this property, templates such as buffer are useful for implementing container classes, since for the latter, the primary factor determining their effectiveness is the ability to allocate them out of free memory. For example, if the implementation of the string class places short strings on the stack, this provides a significant benefit to the program, since in most tasks almost all strings are very short. To implement these types, the buffer template can be useful.

Each type template parameter for a function must affect the type of the function, and this influence is expressed in the fact that it participates in at least one of the types of formal parameters of functions created from the template. This is necessary so that functions can be selected and created based only on their parameters:

```
template <class T> void f1 (T); // fine
template <class T> void f2 (T *); // fine
template <class T> T f3 (int); // mistake
template <int i> void f4 (int [] [i]); // mistake
template <int i> void f5 (int = i); // mistake
template <class T, class C> void f6 (T); // mistake
template <class T> void f7 (const T &, complex); // fine
template <class T> void f8 (Vector <List <T>>); // fine
```

All errors here are caused by the fact that the type parameter of the template has no effect on the formal parameters of the functions.

There is no such restriction in type templates for classes. The fact is that the parameter for such a template must be specified every time an object of the template class is described. On the other hand, for template classes, the question arises: when can two types created from a template be considered the same? Two template class names denote the same class if their template names match, and the parameters used in these names have the same meaning (taking into account possible typedef definitions, evaluation of constant expressions, etc.). Let's go back to the buffer template:

```
template <class T, int sz>
class buffer {
            T v [sz];
            // ...
};

void f ()
{
            buffer <char, 20> buf1;
            buffer <complex, 20> buf2;
            buffer <char, 20> buf3;
            buf fer <char, 100> buf4;
            buf1 = buf2; // error: type mismatch
            buf1 = buf3; // fine
            buf1 = buf4; // error: type mismatch
            // ...
}
```

If the type template for a class uses parameters that specify non-types, it is possible that constructs that look ambiguous may appear:

```
template <int i>
```

```
class X {/ * ... * /};

void f (int a, int b)
{
         X <a> b>; // How to understand this: X <a> b and then
                                              // invalid token, or X
<(a> b)>; ?
}
```

This example is syntactically incorrect because the first angle bracket> terminates the template parameter. In the unlikely event that you need a template parameter that is a greater than expression, use parentheses: X <(a> b)>.

8.7 Type templates and derived classes

We have already seen that combining derived classes (inheritance) and type templates can be powerful. A type template expresses commonality between all types that are used as its parameters, and a base class expresses commonality between all representations (objects) and is called an interface. There are some simple misunderstandings here that should be avoided.

Two types created according to the same template will be different and there is no inheritance relationship between them, except for the only case when these types have identical template parameters. For example:

```
template <cla ss T>
class Vector { / * ... * /}

Vector <int> v1;
Vector <short> v2;
Vector <int> v3;
```

Here v1 and v3 are of the same type, and v2 is of a completely different type. The fact that short is implicitly convertible to int does not imply that there is an implicit conversion from Vector <short> to Vector <int>:

```
v2 = v3; // type mismatch
```

But this is to be expected, since there is no built-in conversion from int [] to short [].

A similar example:

```
class circle: public shape {/ * ... * /};
Vector <circle *> v4;
Vector <shape *> v5;
Vector <circle *> v 6;
```

Here v4 and v6 are of the same type, and v5 is of a completely different type. The fact that there is an implicit conversion from circle to shape and circle * to shape * does not imply that there are implicit conversions from Vector <circle *> to Vector <shape *> or Vector <circle *> * to Vect or <shape *> *:

```
v5 = v6; // type mismatch
```

The point is that, in the general case, the structure (representation) of a class created from a type template is such that inheritance relations are not assumed for it. Thus, a class created from a template can contain an object of the type specified in the template as a parameter, and not just a pointer to it. In addition, allowing such conversions leads to a violation of type checking:

```
void f (Vector <circle> * pc)
{
        Vector <shape> * ps = pc; // error: type mismatch
        (* ps) [2] = new square; // put a round leg into a square
                                                  // hole (memory
allocated for
                                                  // square, but used
for circle
}
```

With the Islist, Tlink, Slist, Splist, Islist_iter, Slist_iter, and SortableVector templates as examples, we have seen that type templates provide a convenient way to create entire families of classes. Without templates, creating such families with just derived classes can be tedious and error prone. On the other hand, if you abandon derived classes and use only templates, then there are many copies of the member functions of the template classes, many copies of the descriptive part of the template classes, and many repeats functions using type templates.

8.7.1 Specifying Implementation Using Template Parameters

In container classes, you often need to allocate memory. Sometimes it is necessary (or just convenient) to give the user the opportunity to choose from several memory allocation options, and also allow him to set his own option. This can be done in several ways. One way is to define a type template to create a new class, the interface of which includes a description of the corresponding container and a class that performs memory allocation in the manner described in $$ 6.7.2:

```
template <cla ss T, class A> class Controlled_container
            : public Container <T>, private A {
            // ...
            void some_function ()
            {
                        // ...
                        T * p = new (A :: operator new (sizeof (T))) T;
                        // ...
            }
            // ...
};
```

The type template is needed here because we are creating the container class. Inheriting from Container <T> is required so that the Controlled_container class can be used as a container class. A type template with an A parameter will allow us to use various placement functions:

```
class Shared: public Arena {/ * ... * /};
class Fast_a llocator {/ * ... * /};

Controlled_container <Process_descriptor, Shared> ptbl;
Controlled_container <Node, Fast_allocator> tree;
Controlled_container <Personell_record, Persistent> payroll;
```

This is a versatile way to provide meaningful implementation information to derived classes. Its positive qualities are its consistency and the ability to use substitution functions. This method is characterized by unusually long names. However, as usual, typedef allows you to define synonyms for overly long type names:

```
typedef
Controlled_container <Personell_record, Persistent> pp_record;
pp_record payroll;
```

Typically, a type template for creating a class such as pp_record is used only when the added implementation information is significant enough not to be manually programmed into the derived class. An example of such a template can be a common (perhaps standard for some libraries) template class Comparator ($$ 8.4.2), as well as non-trivial (possibly standard for some libraries) Allocator classes (classes for memory allocation). Note that the construction of derived classes in such examples follows the "main avenue" that defines the interface with the user (in our example, this is Container). But there are also "side streets" that define implementation details.

8.8 Associative array

Of all the generic non-built-in types, the associative array is probably the most useful. It is often called a map, and sometimes a dictionary, and stores pairs of values. Having one of the values, called the key, you can access the other, called simply the value. An associative array can be thought of as an array, in which the index does not have to be integer:

```
template <class K, class V> class Map {
        // ...
        public:
                V & operator [] (const K &); // find V corresponding to
K
                                                          // and return a link to
it
                // ...
};
```

Here, a key of type K denotes a value of type V. It is assumed that the keys can be compared using the == and <operations, so that the array can be stored in an ordered manner. Note that the Map class differs from the $$ 7.8 assoc type in that it requires a less than operation rather than a hash function.

Here's a simple word counting program that uses the Map pattern and the String type:

```
#include <String.h>
#include <iostream .h>
#include "Map.h"
```

```
int main ()
{
        Map <String, int> count;
        String word;
        while (cin >> word) count [word] ++;
                for (Mapiter <String, int> p = count.first (); p; p ++)
                        cout << p.value () << '\ t' << p.key () << '\ n';
        return 0;
}
```

We use the String type in order not to worry about memory allocation and overflow, which we have to remember about using the char * type. The Mapiter iterator is used to select in order all the values in an array. Iteration in Mapiter is defined as simulating pointers. If the input stream is

```
It was new. It was singular. It was simple. It must succeed.
```

the program will issue

```
4 It
1 must
1 new.
1 simple.
1 singular.
1 succeed.
3 was.
```

Of course, there are many ways to define an associative array, and given the definition of Map and its associated iterator class, we can offer many ways to implement them. The trivial way of implementation is chosen here. Linear search is used and is not suitable for large arrays. Naturally, a commercial implementation will be built with fast search and compact presentation requirements (see Exercise 4 of $$ 8.9).

We use a double-linked list Link:

```
template <class K, class V> class Map;
temp late <class K, class V> class Mapiter;
template <class K, class V> class Link {
        friend class Map <K, V>;
        friend class Mapiter <K, V>;
```

```
            private:
                        const K key;
                        V value;
                        Link * pre;
                        Link * suc;
                        Link (const K & k, const V & v): key (k), value (v) {}
                        ~ Link () {delete suc; } // recursively delete all
                                                                    // objects in the list
   };
```

Each Link object contains a pair (key, value). Classes are described in Link as friends, and this ensures that Link objects can only be created, manipulated, and destroyed using the appropriate iterator and Map classes. Take a look at the preliminary descriptions of the Map and Mapiter template classes.

The Map template can be defined like this:

```
   template <class K, class V> class Map {
            friend class Mapiter <K, V>;
            Link <K, V> * head;
            Link <K, V> * current;
            V def_val;
            K def_key;
            int sz;
            void find (const K &);
            void init () {sz = 0; head = 0; current = 0; }
            public:
                        Map () {init (); }
                        Map (const K & k, const V & d)
                        : def_key (k), def_val (d) {init (); }
                        ~ Map () {delete head; } // recursive delete
                                                      // all objects in the list
            Map (const Map &);
            Map & operator = (const Map &);
            V & operator [] (const K &);
            int size () const {return sz; }
            void clear () {delete head; init (); }
            void remove (const K & k);
            // functions for iteration
            Mapiter <K, V> element (const K & k)
            {
                        (void) operator [] (k); // make k the current element
                        return Mapiter <K, V> (this, current);
            }
            Mapiter <K, V> first ();
            Mapiter <K, V> last ();
   };
```

Items are stored in an ordered list with a cash relationship. For simplicity, nothing is done to speed up the search (see Exercise 4 of $$ 8.9). The key here is the operator [] () function:

```
   template <class K, class V>
   V & Map <K, V> :: operator [] (const K & k)
   {
            if (head == 0) {
                        current = head = new Link <K, V> (k, def_val);
                        current-> pre = current-> suc = 0;
                        return current-> valu e;
            }

     Link <K, V> * p = head;
            for (;;) {
                        if (p-> key == k) {// found
                                    current = p;
                                    return current-> value;
                        }
                        if (k <p-> key) {// insert before p (at the beginning)
                                    current = new Link <K, V> (k, def_val);
                                    current-> pre = p-> pre;
                                    current-> suc = p;
                                    if (p == head) // current element becomes head
                                                head = current;
                                    else
                                                p-> pre-> suc = current;
                                    p-> pre = current;
                                    return current-> value;
```

```
                    }
                    Link <K, V> * s = p-> suc;
                    if (s == 0) {// insert after p (at the end)
                              current = new L ink <K, V> (k, def_val);
                              current-> pre = p;
                              current-> suc = 0;
                              p-> suc = current;
                              return current-> value;
                    }
                    p = s;
          }
}
```

The indexing operation returns a reference to the value that matches the key specified as a parameter. If no such value is found, a new item with a standard value is returned. This allows the indexing operation to be used on the left side of the assignment. The default values for keys and values are set by the Map constructors. The indexing operation defines the current value used by the iterators.

The implementation of the rest of the member functions is left as an exercise:

```
template <class K, class V>
void Map <K, V> :: remove (const K & k)
{
          // see exercise 2 of $$ 8.10
}

template <class K, class V>
Map <K, V> :: Map (const Map <K , V> & m)
{
          // copy the Map table and all its elements
}

template <class K, class V>
Map & Map <K, V> :: operator = (const Map <K, V> & m)
{
          // copy the Map table and all its elements
}
```

Now we just have to define the iteration. The Map class has member functions first (), last (), and element (const K &), which return an iterator set to the first, last, or parameter key, respectively. This can be done because the items are stored in a key-ordered fashion. A Mapiter iterator for a Map is defined like this:

```
template <class K, class V> class Mapiter {
        friend class Map <K, V>;
        Map <K, V> * m;
        Link <K, V> * p;
        Mapiter (Map <K, V> * mm, Link <K, V> * pp)
                {m = mm; p = pp; }
        public:
                Mapiter () {m = 0; p = 0; }
                Mapiter (Map <K, V> & mm);
                operator void * () {return p; }
                const K & key ();
                V & value ();
                Mapiter & operator - (); // prefix
                void operator - (int); // postfix
                Mapiter & operator ++ (); // prefix
                void operator ++ (int); // postfix
};
```

After positioning the iterator, the key () and value () functions from Mapiter return the key and value of the element that the iterator is set to.

```
template <class K, class V> const K & Mapiter <K, V> :: key ()
{
        if (p) return p-> key; else return m-> def_key;
}

template <class K, class V> V & Mapiter <K, V> :: value ()
{
        if (p) return p-> value; else return m-> def_val;
}
```

By analogy with pointers, ++ and - are defined to move forward and backward through Map elements:

```
Mapiter <K, V> & Mapiter <K, V> :: operator - () // prefix decrement
{
        if (p) p = p-> pre;
        retu rn * this;
}

void Mapiter <K, V> :: operator - (int) // postfix decrement
{
        if (p) p = p-> pre;
}

Mapiter <K, V> & Mapiter <K, V> :: operator ++ () // prefix increment
{
        if (p) p = p-> suc;
        return * this;
}

void Mapiter <K, V> :: operator ++ (int) // postfix increment
{
        if (p) p = p-> suc;
}
```

Postfix operations are defined so that they do not return any value. The fact is that the cost of creating and passing a new Mapiter object at each iteration step is significant, and the benefit from it will not be great.

The Mapiter object can be initialized to be set to the beginning of the Map:

```
template <class K, class V> Mapiter <K, V> :: Mapiter (Map <K, V> &
mm)
{
        m == & mm; p = m-> head;
}
```

The conversion operator void * () returns zero if the iterator is not set to the Map element, and nonzero otherwise. So you can check the iterator iter, for example, like this:

```
void f (Mapiter <const char *, Shape *> & iter)
{
        // ...
        if (iter) {
                // set to table element
        }
        else {
                // not set to table element
        }
        // .. .
}
```

A similar trick is used to control streaming I / O in $$ 10.3.2.

If the iterator is not set to a table element, its key () and value () functions return references to standard objects.

If, after all these definitions, you forget their purpose, you can bring up another small program that uses the Map table. Let the input stream be a list of value pairs of the following form:

```
hammer 2
nail 100
saw 3
saw 4
hammer 7
nail 1000
nail        250
```

You need to sort the list so that the values corresponding to one subject add up, and print the resulting list along with the total value:

```
hammer 9
nail 1350
saw 7
-------------------
total 1366
```

First, let's write a function that reads input lines and enters items with their number into the table. The key in this table is the first word of the row:

```
template <class K, class V>
void readlines (Map <K, V> & key)
{
        K word;
        while (cin >> word) {
```

```
                    V val = 0;
                    if (cin >> val)
                              key [word] + = val;
                    else
                              return;
          }
}
```

Now we can write a simple program that calls the readlines () function and prints the resulting table:

```
main ()
{
          Map <String, int> tbl ("nil", 0);
          readlines (tbl);
          int total = 0;
          for (Mapiter <St ring, int> p (tbl); p; ++ p) {
                    int val = p.value ();
                    total + = val;
                    cout << p.key () << '\ t' << val << '\ n';
 }
          cout << "------------------- \ n";
          cout << "total \ t" << total << '\ n';
}
```

8.9 Exercises

1. (* 2) Define a family of double-linked lists that will be counterparts to the single-linked lists defined in $$ 8.3.

2. (* 3) Define a template of type String, whose parameter is the type of the character. Show how it can be used not only for ordinary characters, but also for the hypothetical class lchar, which represents non-English characters or an extended character set. You should try to define String in such a way that the user does not notice the degradation of the program's performance in terms of memory, time, or convenience compared to a regular string class.

3. (* 1.5) Define a Record class with two data members: count (quantity) and price (price). Order the vector of such records for each

of the members. However, you cannot change the sort function and Vector template.

4. (* 2) Complete the definitions of the template class Map by writing the missing member functions.

5. (* 2) Define a different Map implementation from $$ 8.8 using a double-linked list class.

6. (* 2.5) Define another Map implementation from $$ 8.8 using a balanced tree. Such trees are described in $$ 6.2.3 of D. Knuth's book "The Art of Computer Programming" v.1, "World", 1978 [K].

7. (* 2) Compare the quality of the two Map implementations. The first uses the Link class with its own placement function, while the second does not.

8. (* 3) Compare the performance of the $$ 8.8 word counting program with a similar program that does not use the Map class. I / O operations must be used in the same way in both programs. Compare several such programs that use different variants of the Map class, including the class from your library, if there is one.

9. (* 2.5) Use the Map class to implement a topological sort. It is described in [K] volume 1, pp. 323-332. (see exercise 6).

10. (* 2) Modify the $$ 8.8 program so that it works correctly for long names and for names containing spaces (eg "thumb back").

11. (* 2) Define a type pattern to read different kinds of strings , such as (item, quantity, price).

12. (* 2) Define the Sort class from $$ 8.4.5 using Shell sort . Show how you can set the sorting method using a template parameter. The sorting algorithm is described in [K] v.3, $$ 5.2.1 (see exercise 6).

13. (* 1) Modify the Map and Mapiter definitions so that the ++ and - postfix operations return a Mapiter object.

14. (* 1.5) Use type templates in a modular programming style , as shown in $$ 8.4.5, and write a sort function that evaluates directly to Vector <T> and T [].

CHAPTER 9.

I interrupted you, so don't interrupt me.
- Winston Churchill

This chapter describes the exception handling mechanism and some of the error handling techniques based on it. The mechanism consists in triggering an exception, which must be intercepted by a special handler. The rules for catching exceptions and the rules for reacting to uncaught and unexpected exceptions are described. Entire groups of exceptions can be defined as derived classes. Describes a method using destructors and exception handling , which provides reliable and hidden resource management.

9.1 Error handling

The library creator is able to detect dynamic errors, but has no idea what the general response should be. The user of the library is able to write a reaction to such errors, but cannot detect them. If he could, he would have dealt with errors in his program himself, and they would not have to be detected in library functions. To solve this problem, the concept of a special situation is introduced into the language .

It is only recently that exceptions have been included in the language standard by the C ++ Standardization Committee , but at the time of this writing they have not yet been included in most implementations.

The essence of this concept is that a function that has detected an error and can not cope with it triggers a special situation, hoping that the problem can be eliminated in the function that directly or indirectly called the first one. If a function is designed to handle some kind of error , it can indicate this explicitly as a willingness to catch the exception.

As an example, consider how, for the Vector class, you can represent and handle exceptions caused by an array out of bounds:

```
class Vector {
        int * p;
        int sz;
        public:
                class Range {}; // class for an exception
                int & operator [] (int i);
                // ...
};
```

It is assumed that objects of the Range class will be used as exceptions, and you can run them like this:

```
int & Vector :: operator [] (int i)
{
            if (0 <= i && i <sz) return p [i];
            throw Range ();
}
```

If a function provides a response to an error with an invalid index value , then the part of the function in which these errors will be caught must be placed in a try statement. It should also contain an exception handler:

```
void f (Vector & v)
{
            // ...
            try {
                        do_something (v); // the content part dealing with v
            }
            catch (Vector :: Range) {
                        // exception handler Vector :: Range
                        // if do_something () fails,
                        // need to somehow react to this
                        // we will get here only if
                        // calling do_something () will call Vector :: operator []
()
                        // due to invalid index value
            }
            // ...
}
```

The exception handler is the construction

```
catch (/ * ... * /) {
            // ...
}
```

It can be used only immediately after a block that begins with the try service word, or immediately after another exception handler. The catch word is also useful. It is followed by a description in brackets, which is used similarly to the description of the formal parameters of the function, namely, it specifies the type of objects for which the handler is designed and, possibly, the names of the parameters (see $$ 9.3). If an index error occurs in do_something () or in any function called from it (on any Vector object), then the handler will catch the exception and the part that handles the error will be executed. For example, defining the following functions will trigger a handler in f ():

```
void do_something ()
{
        // ...
        crash (v);
        // ...
}

void crash (Vector & v)
{
        v [v.size () + 10]; // artificially cause an index error
}
```

The process of triggering and catching an exception involves traversing the chain of calls from the point of triggering the exception to the function in which it is being intercepted. This restores the state of the stack corresponding to the function that caught the error, and destructors are called for the local objects of functions from this chain while traversing the entire chain of calls . This is described in detail in $$ 9.4.

If, when looking through the entire chain of calls, starting with the function that triggered the exception, no suitable handler is found, the program terminates. This is described in detail in $$ 9.7.

If a handler has caught an exception, it will be processed and other handlers designed for this situation will not be considered. In other words, only the handler that is in the most recently called function containing the corresponding handlers will be activated . In our example, the f () function will intercept Vector :: Range, so this exception cannot be caught in any function calling f ():

```
int ff (Vector & v)
{
        try {
            f (v); // Vector :: Range will be intercepted in f ()
        }
        catch (Vector :: Range) {// so we will never get here
            // ...
        }
}
```

9.1.1 Exceptions and traditional error handling

Our way of handling errors compares favorably with more traditional methods in many respects . Let's list what the Vector :: operator [] () indexing operation can do when an invalid index value is encountered :

[1] complete the program;

[2] return the value treated as "error";

[3] return to normal and leave the program in an undefined state;

[4] call the function specified to respond to such an error.

Option [1] ("terminate the program") is implemented by default when the exception was not caught. For most errors, you can and should provide a better response.

Option [2] ("return the value" error "") can not always be implemented , since it is not always possible to determine the value "error". So, in our example, any integer is a valid value for the result of the indexing operation. If you can highlight such a special value, then often this option is still inconvenient, since you have to check for this value with each call. This way you can easily double the size of the program. Therefore, this option is rarely used sequentially to find all errors.

Option [3] ("leave the program in an undefined state") has the disadvantage that the calling function might not notice the abnormal state of the program. For example, many C standard library functions set the global variable errno to signal an error . However, user programs usually do not have sufficient sequential control of errno, and the result is hover errors caused by standard functions returning the wrong value. Moreover, if the program has parallel computations, using one global variable to signal different errors will inevitably lead to disaster.

Exception handling was not intended for those cases for which option [4] is designed ("call the error response function "). Note, however, that if special situations are not provided, then instead of the error response function, you can just use only one of the three listed options. The discussion of reaction functions and special situations will continue in $$ 9.4.3.

The exception mechanism successfully replaces traditional error handling in cases where the latter is incomplete, ugly, or error-prone. This mechanism allows you to explicitly separate the part of the program in which errors are handled from the rest of it, thereby making the program more understandable and easier for various service programs to work with. This mechanism's inherent error handling makes it easier to communicate between separately written parts of a program.

There is a new point in this way of handling errors for C programmers : the standard response to an error (especially an error in a library function) is to terminate the program. The traditional reaction was to continue the program in the hope that it would somehow end on its own. Therefore, the method based on special situations makes the program more "fragile" in the sense that more effort and attention is required to execute it normally. But this is still better than getting incorrect results at a later stage in the development of the program (or getting them even later, when the program is considered complete and handed over to an unsuspecting user). If the termination of the program is an unacceptable reaction, it is possible to simulate the traditional reaction by catching all exceptions or all exceptions belonging to a special class ($$ 9.3.2).

The exception mechanism can be viewed as a dynamic analogue of the translation-stage type checking and ambiguity checking mechanism . With this approach, the design stage of the program becomes more important , and more support for the program execution process is required than for C programs.However, the result will be a more predictable program, it will be easier to integrate it into a software system, it will be easier for other programmers to understand and use it. will work with various service programs. We can say that the mechanism of exceptions supports, like other C ++ tools, a "good" programming style, which in languages such as C, can only be applied incompletely and at an informal level.

It must be recognized, however, that error handling remains a difficult task, and although the exception mechanism is stricter than traditional methods, it is still not sufficiently structured compared to constructs that only allow local control transfers .

9.1.2 Other perspectives on special situations

"Special situation" is one of those concepts that have different meanings for different people. In C ++, the exception mechanism is designed to handle errors. In particular, it is designed to handle errors in programs consisting of independently generated components.

This mechanism is designed for special situations that arise only during sequential program execution (for example, control of array boundaries). Asynchronous exceptions, such as keyboard interrupts , cannot be handled directly with this mechanism. Different systems have other mechanisms, such as signals, but they are not covered here as they are system specific.

The exception mechanism is a non-local transfer construct and can be thought of as a variant of the return statement. Therefore, exceptions can be used for purposes other than error handling ($$ 9.5). After all, the main purpose of the exception mechanism and the topic of this chapter will be error handling and the creation of error-resistant programs.

9.2 Distinguishing special situations

Naturally, several different dynamic errors are possible in the program . These errors can be associated with special situations that have different names. For example, in the Vector class, you usually have to identify and report two types of errors: range errors and errors caused by an inappropriate parameter for the constructor:

```
class Vector {
        int * p;
        int sz;
        public:
                enum {max = 32000};
                class Range {}; // index exception
                class Size {}; // exception "invalid size"
                Vector (int sz);
                int & operator [] (int i);
                // ...
};
```

As mentioned, an index operation triggers a Range exception if an out-of-range index is specified for it. The constructor throws a Size exception if it is given an invalid vector size:

```
Vector :: Vector (int sz)
{
        if (sz <0 || max <sz) throw Size ();
        // ...
}
```

A user of the Vector class can distinguish between these two special situations by specifying handlers for both situations in the block being checked (i.e., in the try statement block) :

```
void f ()
{
        try {
                use_vectors ();
        }
        catch (Vector :: Range) {
                // ...
        }
        catch (Vector :: Size) {
                // ...
        }
}
```

Depending on the exception, the appropriate handler will be executed . If control reaches the end of the handler statements, the next statement will be executed after the list of handlers:

```
void f ()
{
        try {
                use_vectors () ;
        }
        catch (Vector :: Range) {
                // fix the index and
                // try again:
                f ();
        }
        catch (Vector :: Size) {
                cerr << "Error in Vector :: Size constructor";
                exit (99);
```

```
            }
            // we will get here, if there were no special situations at all
            // or after handling a Range exception
}
```

The list of handlers resembles a switch, but here no break statements are needed in the body of the handler. The syntax of the handler list differs from the syntax of the switch case variants, partly for this reason, and partly to show that each handler defines its own scope (see $$ 9.8).

It is not necessary to intercept all special situations in one function:

```
void f1 ()
{
            try {
                        f2 (v);
            }
            catch (Vector :: Size) {
                        // ...
            }
}

void f2 (Vector & v)
{
            try {
                        use_vectors () ;
            }
            catch (Vector :: Range) {
                        // ...
            }
}
```

Here, f2 () will catch the Range exception raised in use_vectors (), and the Size exception will be left for f1 ().

From the point of view of the language, an exception is considered handled immediately upon entering the body of its handler. Therefore, all exceptions triggered by this handler must be handled in the functions that called the function that contains the block being checked. This means that the following example will not create an infinite loop:

```
try {
```

```
            // ...
}
catc h (input_overflow) {
            // ...
            throw input_overflow ();
}
```

Here input_overflow (input overflow) is the name of the global class.

Exception handlers can be nested:

```
try {
            // ...
}
catch (xxii) {
            try {
                        // complex reaction
            }
            catch (xxii) {
                        // error during complex reaction
            }
}
```

However, such nesting is rarely needed in regular programs, and more often than not, it is evidence of bad style.

9.3 Exception names

The special situation is caught by its type. However, it is not the type that is launched, but the object. If we need to pass some information from the launch point to the handler, then for this it should be placed in the launched object. For example, let's say you need to know an index value that is out of range:

```
class V ector {
            public:
                        class Range {
                                    public:
                                                int index;
                                                Range (int i): index (i) {}
                        };
                        // ...
```

```
                    int & operator [] (int i)
};

int Vector :: operator [] (int i)
{
          if (o <= i && i <sz) return p [i];
          throw Range (i);
}
```

To investigate an invalid index value, you need to name the exception object in the handler :

```
void f (Vector & v)
{
          // ...
          try {
                    do_something (v);
          }
          catch (Vector :: Range r) {
                    cerr << "Invalid index" << r.index << '\ n';
                    // ...
          }
          // ...
}
```

The construction in parentheses after the catch word is essentially a description and it is similar to the description of a formal parameter of a function. It specifies what the type of the parameter (i.e., exception) can be and the name for the actual one, i.e. neglected, special situation. Recall that in type templates we had a choice for naming exceptions. Each template-generated class had its own exception class:

```
template <class T> class Allocator {
          // ...
          class Exhausted {}
          // ...
          T * get ();
};

void f (Al locator <int> & ai, Allocator <double> & ad)
```

```
{
        try {
                // ...
        }
        catch (Allocator <int> :: Exhausted) {
                // ...
        }
        catch (Allocator <double> :: Exhausted) {
                // ...
        }
}
```

On the other hand, the exception can be common for all classes created from the template:

```
c lass Allocator_Exhausted {};

template <class T> class Allocator {
        // ...
        T * get ();
};

void f (Allocator <int> & ai, Allocator <double> & ad)
{
        try {
                // ...
        }
        catch (Allocator_Exhausted) {
                // ...
        }
}
```

It is difficult to say which way of specifying a special situation is preferable. The choice depends on the purpose of the template in question.

9.3.1 Grouping exceptions

Special situations are naturally broken down into families. Indeed, it is logical to represent the Matherr family, which includes Overflow, Underflow, and some other special situations. The Matherr family form exceptions that can trigger math functions in the standard library.

One way to define such a family is to define Matherr as a type whose possible values include Overflow and all the others:

```
enum {Overflow, Underflow, Zerodivide, / * ... * /};
try {
        // ...
}
catch (Matherr m) {
        switch (m) {
                case Overflow:
                        // ...
                case Underflow:
                        // ...
                        // ...
        }
        // ...
}
```

Another way is to use inheritance and virtual functions to avoid introducing a switch by type field value. Inheritance helps describe families of exceptions:

```
class Matherr {};
class Overflow: public Matherr {};
cl ass Underflow: public Matherr {};
class Zerodivide: public Matherr {};
// ...
```

It often happens that you need to handle a Matherr exception regardless of which situation from this family occurred. Inheritance makes it easy:

```
try {
        // ...
}
catch (Overflow) {
        // handle Overflow or any derived situation
}
catch (Matherr) {
        // handle any situation other than Overflow
}
```

In this example, Overflow is dealt with separately, and all other exceptions from Matherr are dealt with as one general case. Of course, the function containing catch (Matherr) will not know which exception it is catching. But whatever it is, upon entering the handler, the copy passed to it will be Matherr. This is usually just what you need. If it is not, the exception can be caught by reference (see $$ 9.3.2).

The hierarchical ordering of exceptions can play an important role in creating a clear program structure. Indeed, suppose there is no such ordering, and you need to handle all the special situations of the standard library of mathematical functions. To do this, you will have to list all the possible special situations ad infinitum:

```
try {
        // ...
}
catch (Overflow) {/ * ... * /}
catch (Underflow) {/ * ... * /}
catch statement (Zerodivide) {/ * .. . * /}
// ...
```

This is not only tiring, but also dangerous, as it can forget a special situation. In addition, having to list all exceptions in the block under test virtually guarantees that when the library's exception family expands, an error will occur in the user program. This means that when a new exception is introduced into the library of mathematical functions, you will have to re-translate all parts of the program that contain handlers for all exceptions from Matherr. In general, such retransmission is unacceptable. Often it is not even possible to find all the parts of the program that require re-translation. If this is possible, it cannot be required that the source code of any part of a large program is always available, or that we have the right to modify any part of a large program whose source code we have. In fact, the user doesn't have to think about the internals of the libraries. All of these re-translation and maintenance issues can result in no new exceptions being introduced after the first release of the library. But this solution is not suitable for almost all libraries.

All of these arguments suggest that exceptions should be defined as a class hierarchy (see also $$ 9.6.1). This, in turn, means that exceptions can be members of several groups:

```
class network_file_err // file system errors on the network
        : public network_err, // network errors
        public file_system_err { // file system errors
        // ...
};
```

The network_file_err exception can be caught in functions that handle network exceptions:

```
void f ()
{
        try {
                // some operators
        }
        catch (network_err) {
                // ...
        }
}
```

It can also be intercepted in functions that handle file system exceptions:

```
voi d g ()
{
        try {
                // some other operators
        }
        catch (file_system_err) {
                // ...
        }
}
```

This is an important point because a system service such as networking must be transparent, which means that the creator of the g () function may not even know that this function will be executed in network mode.

Note that there is currently no standard set of exceptions for the standard math and I / O libraries. It is up to the ANSI and ISO C ++ committees to decide whether such a set is needed and what names and classes should be used in it.

Since it is possible to intercept all exceptions at once (see $$ 9.3.2), there is no urgent need to create a generic, base class for all exceptions for this purpose. However, if all exceptions derive from the empty class Exception, then their use becomes more regular in interfaces (see $$ 9.6). If you are using a generic base Exception class, make sure there is nothing in it other than a virtual destructor. Otherwise, such a class may conflict with the intended standard.

9.3.2 Derived exceptions

If we use a class hierarchy to handle exceptions, then, of course, each handler should only deal with a part of the information transmitted in special situations. We can say that, as a rule, an exception is caught by the handler of its base class, and not by the handler of the class corresponding to this particular exception. Naming and catching an exception handler is semantically equivalent to naming and getting a parameter in a function. Simply put, a formal parameter is initialized with the value of the actual parameter. This means that a triggered exception is "relegated" to the exception expected by the handler. For example:

```
class Matherr {
        // ...
        virtual void debug_print ();
};

class Int_overflow: public Matherr {
        public:
                char * op;
                int opr1, opr2 ;;
                int_overflow (const char * p, int a, int b)
                        {cerr << op << '(' << opr1 << ',' << opr2 << ')';
}
};

void f ()
{
        try {
                g ();
        }
        catch (Matherr m) {
                // ...
```

```
            }
    }
```

When entering a Matherr handler, exception m is a Matherr object, even though Int_overflow was triggered when calling g (). This means that additional information passed to Int_overflow is not available.

As usual, pointers or links can be used to access additional information. Therefore, one could write like this:

```
    int add (int x, int y) // add x and y with control
    {
            if (x> 0 && y > 0 && x> MAXINT - y
                    || x <0 && y <0 && x <MININT + y)
                    throw Int_overflow ("+", x, y);
                    // This is where we get, either when checking
                    // gave a negative result for overflow,
                    // either when x and y have different signs
            return x + y;
    }

    void f ()
    {
            try {
                    add (1,2);
                    add (MAXINT, -2);
                    add (MAXINT, 2); // and then overflow
            }
            catch (Matherr & m) {
                    // ...
                    m.debug_print ();
            }
    }
```

Here, the last call to add would trigger an exception, which in turn would trigger Int_overflow :: debug_print (). If the exception were passed by value rather than by reference, then Matherr :: debug_print ( ).

It often happens that, having intercepted a special situation, the handler decides that there is nothing it can do about this error. In this case, the most natural thing to do is to run the exception again, hoping that another handler can handle it:

```
void h ()
{
        try {
                // some operators
        }
        catch (Matherr) {
                if (can_handle_it) { // if processing is possible,
                        // make it
                }
                else {
                        throw; // restart the intercepted
                        // special situation
                }
        }
}
```

The rerun is written as a throw statement with no parameters. This triggers the original exception that was intercepted again, and not the part of it that the Matherr handler was designed to handle. In other words, if Int_overflow was started, the function calling h () could intercept it as Int_overflow, even though it was intercepted in h () as Matherr and started again:

```
void k ()
{
        try {
                h () ;
                // ...
        }
        catch (Int_overflow) {
                // ...
        }
}
```

The degenerate restart case is useful. As with functions, ellipsis ... for a handler means "any parameter", so the catch (...) statement means catching any exception:

```
void m ()
```

```
{
            try {
                        // some operators
            }
            catch (...) {
                        // get everything in order
                        throw;
            }
}
```

This example should be understood as follows: if an exception occurs during the execution of the main part of m (), a handler is executed that performs general actions to eliminate the consequences of the exception; after these actions, the exception that caused them is restarted.

Since a handler can catch multiple types of derived exceptions, the order in which the handlers go in the block being checked is significant. Handlers try to catch exceptions in the order they are described. Let's give an example:

```
try {
            // ...
}
catch (ibuf) {
            // handle input buffer overflow
}
catch (io) {
            // handle any I / O error
}
catch (stdlib) {
            // handle any exception in the library
}
catch (...) {
            // handle all other exceptions
}
```

The exception type in the handler corresponds to the triggered exception type in the following cases: if these types match, or the second type is the type of the available base class of the triggered situation, or it is a pointer to such a class, and the expected situation type is also a pointer ($$ R.4.6).

Since the translator knows the class hierarchy, it is able to detect such ridiculous errors when the catch (...) handler is not specified last, or when the base class situation handler precedes the situation handler derived from this class ($$ R15.4). In both cases, the subsequent handler (or handlers) cannot be run because they are "masked" by the first handler.

9.4 Resource Requests

If a function requires certain resources, for example, you need to open a file, allocate a block of memory in the free memory area, set exclusive access rights, etc., for the further operation of the system it is usually extremely important that the resources are released properly. Typically, this "proper way" is implemented by a function that requests resources and releases them before exiting. For example:

```
void use_file (const char * fn)
{
        FILE * f = fopen (fn, "w"); // work with f
        fclose (f);
}
```

This all looks quite normal until you realize that any error after calling fopen () and before calling fclose () will result in an exception that will cause us to exit use_file () without calling fclose. (). It should be said that the same problem occurs in languages that do not support special situations. For example, calling the longjump () function from the C standard library can have the same unpleasant consequences.

If you are building fault-tolerant systems, this problem will have to be addressed. A primitive solution can be given:

```
void use_file (const char * fn)
{
        FILE * f = fopen (fn, "w");
        try {
                // work with f
        }
        catch (...) {
                fclose (f);
                throw;
        }
        fclose (f);
}
```

The entire part of the function that works with the file f is placed in a block under test, in which all exceptions are caught, the file is closed, and the exception is re-run.

The disadvantage of this solution is its verbosity, cumbersomeness and potential wastefulness. In addition, any verbose and cumbersome solution is fraught with errors, if only due to programmer fatigue. Fortunately, there is a better solution. In general terms, the problem can be formulated as follows:

```
void acquire ()
{
        // request for resource 1
        // ...
        // request for resource n
        // resource usage
        // free resource n
        // ...
        // free resource 1
}
```

As a rule, it is important that resources are released in the reverse order of the requests. This is very similar to the way you work with local objects created by constructors and destroyed by destructors. Therefore, we can solve the problem of requesting and freeing resources if we use appropriate class objects with constructors and destructors. For example, you can define a FilePtr class that acts as a FI LE * type:

```
class FilePtr {
        FILE * p;
        public:
                FilePtr (const char * n, const char * a)
                        {p = fopen (n, a); }
                FilePtr (FILE * pp) {p = pp; }
                        ~ FilePtr () {fclose (p); }
                operator FILE * () {return p; }
};
```

You can construct a FilePtr object either by having an object of type FILE *, or by receiving the parameters required for fopen (). In any case, this object will be destroyed when it leaves its scope, and its destructor will close the file. Our example now compresses to a function like this:

```
void use_file (const char * fn)
{
        FilePtr f (fn, "w" );
        // work with f
}
```

The destructor will be called regardless of whether the function ended normally or an exception was triggered.

9.4.1 Constructors and destructors

The described resource management method is commonly referred to as "requesting resources by initialization". This is a generic trick that takes into account the properties of constructors and destructors and their interaction with the exception mechanism.

An object is not considered built until its constructor has finished executing. Only after this is it possible to unwind the stack accompanying the call of the object's destructor. An object consisting of nested objects is built to the extent that the nested objects are built.

A well-written constructor must ensure that the object is built completely and correctly. If he fails to do so, he should, as far as possible, restore the state of the system, which was before the start of construction. For simple constructors, it would be ideal to always satisfy at least one condition - correctness or completeness of objects, and never leave an object in a "half-built" state. This can be accomplished by using the "request resources by initialization" technique when constructing the members.

Consider a class X whose constructor requires two resources: a file x and a lock y (i.e., exclusive access to something). These requests may be rejected and trigger an exception. To simplify the programmer's work, you can require that the constructor of class X never ends with the request for the file being satisfied, but not for the lock. To represent two types of resources, we will use objects of two classes FilePtr and LockPtr (naturally, one class would be enough if x and y are resources of the same type). Requesting a resource looks like initializing an object representing the resource:

```
class X {
		FilePtr aa;
		LockPtr bb;
		// ...
		X (const char * x, const char * y)
				: aa (x), // request `x '
		bb (y) // request `y '
		{}
		// ...
};
```

Now, as was the case for local objects, all the overhead related to resources can be left to the implementation. The user is not obliged to follow the progress of such work. For example, if an exception occurs after constructing aa and before constructing bb, only the destructor aa will be called, but not bb.

This means that if you strictly adhere to this simple scheme for requesting resources, then everything will be fine. More importantly, the creator of the constructor does not need to write exception handlers himself.

For requests to allocate a block in free memory, the most arbitrary order of resource requests is characteristic. Examples of such queries have already been encountered many times in this book:

```
class X {
		int * p;
		// ...
		public:
				X (int s) {p = new int [s]; init (); }
				~ X () {delete [] p; }
				// ...
};
```

This is a typical example of free memory usage, but combined with special situations, it can lead to memory depletion. Indeed, if an exception is triggered in init (), then the allocated memory will not be freed. The destructor will not be called because the construction of the object has not been completed. There is a more reliable version of this example:

```
template <class T> class MemPtr {
		public:
```

```
                    T * p;
                    MemPtr (size_t s) {p = new T [s]; }
                    ~ MemPtr () {delete [] p; }
                    operator T * () {return p; }
}
class X {
            MemPtr <int> cp;
            // ...
            public:
                    X (int s): cp (s) {init (); }
                    / / ...
};
```

Now the destruction of the array pointed to by p is implicit in MemPtr. If init () raises an exception, the allocated memory will be freed by implicitly calling the destructor on a fully constructed nested cp object.

Note also that the standard C ++ memory allocation strategy ensures that if operator new () fails to allocate memory for an object, then its constructor will never be called. This means that the user does not have to worry about the constructor or destructor being called on a non-existent object.

In theory, the additional cost required to handle exceptional situations when none of them actually occurred can be reduced to zero. However, this is unlikely to be true for early implementations of the language. Therefore, it would be wise not to use local class variables with destructors in critical internal program loops.

9.4.2 Precautions

Not all programs need to be robust against all kinds of errors. Not all resources are critical enough to justify trying to protect them with the described "request for resources by initialization" method. There are many programs out there that simply read the input and execute to the end. For them, the most appropriate response to a dynamic error would be to simply terminate the count (after issuing the appropriate message). It is the responsibility of the system to release all the requested resources, and the user must rerun the program with more appropriate input. Our scheme is intended for tasks in which such a primitive response to a dynamic error is unacceptable. For example, the developer of a library is usually not in a position to make assumptions about how robust a program that uses the library should be. Therefore, it must account for all dynamic errors and free all resources before returning from the library function to the user program. The "request resources by initialization" method, in conjunction with special error situations, can be useful when building many libraries.

9.4.3 resource exhaustion

There is one of the eternal problems in programming: what to do if a request for a resource has failed? For example, in the previous example, we calmly opened files using fopen () and requested a block of free memory using the new operation, without thinking that such a file might not exist, and free memory might be exhausted. To solve this kind of problem, programmers have two ways:

1. Re-request: The user must change his request and repeat it.
2. Completion: request additional resources from the system, if not available, trigger an exception.

The first method assumes the user's assistance to set an acceptable request; in the second, the user must be ready to correctly respond to the refusal to allocate resources. In most cases, the latter method is much simpler and allows the system to keep the different levels of abstraction separate.

In C ++, the first way is supported by the function call mechanism, and the second by the exception mechanism. Both methods can be demonstrated by an example of the implementation and use of the new operation:

```
#include <stdlib.h>
extern void * _last_allocation;
extern void * operator new (size_t size)
{
        void * p;
        while ((p = malloc (size)) == 0) {
                if (_new_handler)
                        (* _new_handler) (); // ask for help
                else
                        return 0;
        }
        return _last_allocation = p;
}
```

If the new () operation cannot find free memory, it calls the _new_handler () control function. As long as sufficient memory can be allocated in _new_handler (), everything is fine. If not, you cannot return from the control function to the new operation, since an endless cycle will arise. Therefore, the control function can trigger an exception and leave the program calling new to correct the situation:

```
void my_new_handler ()
{
        try_find_some_memory (); // try to find
                                        // free memory
        if (found_some ()) return; // if found, everything is fine
        throw Memory_exhausted (); // otherwise run a special
                                        // "Out of memory"
situation
}
```

Somewhere in the program there must be a block to be checked with the appropriate handler:

```
try {
        // ...
}
catch (Memory_exhausted) {
        // ...
}
```

The operator new () function used a pointer to the control function _new_handler, which is set by the standard function set_new_handler (). If you need to tune in to your own control function, you need to apply like this

```
set_new_handler (& my_new_handler);
```

You can intercept the Memory_exhausted situation as follows:

```
void (* oldnh) () = set_new_handler (& my_new_ handler);
try {
        // ...
}
catch (Memory_exhausted) {
        // ...
}
```

```
   catch (...) {
             set_new_handler (oldnh); // restore pointer to
                                                           // control function
             throw (); // re-run the exception
   }

   set_new_handler (oldnh); // restore pointer to
                                                           // control function
```

You can do even better if you apply the "request resources by initialization" method described in $$ 9.4 to the control function and remove the catch (...) handler.

In a solution using my_new_handler (), no information is passed from the point of error detection to the function where it is handled. If you need to transfer some data, then the user can include his control function in the class. Then, in the function that detected the error, the necessary data can be placed in an object of this class. A similar method using function objects was used in $$ 10.4.2 to implement manipulators. The way in which a pointer to a function or a function object is used in order to call back the function that requested this resource from a control function serving a certain resource is usually called simply a callback.

It should be understood that the more information is transferred from the function that detected the error to the function trying to fix it, the greater the relationship between these two functions. In general, it is better to minimize such dependencies, since any change in one of the functions will have to be done with the other function in mind, and it may also have to be changed. In general, it is best not to mix the individual components of the program. The exception mechanism allows you to keep separate components better than the usual mechanism for calling control functions, which is set by the function that requested the resource.

In general, a reasonable approach is to have multiple levels of resource allocation (according to the levels of abstraction). In doing so, you should avoid allowing functions at one level to depend on a control function called at another level. The experience of creating large software systems shows that over time, successful systems develop in this direction.

9.4.4 Exceptions and constructors

Exceptions provide a means of signaling errors occurring in the constructor. Since the constructor does not return a value that the calling function could check, there are the following common (i.e., no exceptions) signaling methods:

1] Return an object in an abnormal state, expecting the user to check its state.

2] Set the value of a non-local variable that signals that the object could not be created.

Special situations allow the fact that the object could not be created to be passed from the constructor to the outside:

```
Vector :: Vector (int size)
{
        if (sz <0 || max <sz) throw Size ();
        // ...
}
```

In the function that creates vectors, you can catch errors caused by an invalid size (Size ()) and try to react to them:

```
Vector * f (int i)
{
        Vector * p;
        try {
                p = new Vector v (i);
        }
        catch (Vector :: Size) {
                // reaction to invalid vector size
        }
        //. ..
        return p;
}
```

The function that controls the creation of the vector is capable of correctly responding to an error. In the exception handler itself, you can use any of the standard methods for diagnosing and recovering from an error. Each time an exception is caught, the control function may have its own view of the cause of the error. If data describing it is transmitted with each special situation, then the amount of data that needs to be analyzed for each error grows. The main goal of error handling is to provide a reliable and convenient way to transfer data from the initial point of error detection to the point where meaningful recovery is possible after it.

The "requesting resources by initialization" method is the most reliable and beautiful solution when there are constructors that require more than one resource. In essence, it allows you to reduce the task of allocating multiple resources to a reusable, simpler, single resource method.

9.5 Exceptional situations may not be errors

If a special situation was expected, was intercepted and did not have a bad effect on the program flow, is it worth calling it an error? They say that only because the programmer thinks of it as an error, and the exception mechanism is a means of handling errors. On the other hand, exceptional situations can be viewed simply as another governance structure. Let's confirm this with an example:

```
class message {/ * ... * /}; // message

class queue {// queue
            // ...
            message * get (); // return 0 if the queue is empty
            // ...
};

void f1 (queue & q)
{
            message * m = q.get ();
            if (m == 0) {// queue is empty
                        // ...
            }
            // use m
}
```

This example can be written like this:

```
class Empty {} // exception type "Empty_queue"
class queue {
            // ...
            message * get (); // start Empty if the queue is empty
```

```
        // ...
};

void f2 (queue & q)
{
         try {
                message * m = q.get ();
                // use m
        }
        catch (Empty) {// queue is empty
                // ...
        }
}
```

There is even some charm in the variant with a special situation. This is a good example of when it is difficult to say if such a situation can be considered a mistake. If the queue should not be empty (i.e. it is empty very rarely, say once in a thousand), and actions in the case of an empty queue can be considered as restoration, then in the function f2 () the view of the exception will be the same as we so far and have stuck (i.e. exception handling is error handling). If the queue is often empty, and the actions taken in this case form one of the branches of the normal program flow, then you will have to abandon this view of the special situation, and the function f2 () must be rewritten:

```
class queue {
        // ...
        message * get (); // start Empty if the queue is empty
        int empty ();
        // ...
};

void f3 (queue & q)
{
        if (q.empty ()) {// queue is empty
                / / ...
        }
        else {
                message * m = q.get ();
```

```
                    // use m
            }
    }
```

Note that it is possible to remove the emptyness check from the get () function only if there are no parallel calls to the queue.

It is not easy to abandon the view that exception handling is error handling. As long as we hold this point of view, the program is clearly divided into two parts: the usual part and the error handling part. Such a program is more understandable. Unfortunately, it is impossible to make a clear division in real problems, therefore the structure of the program should (and will) reflect this fact. Suppose the queue is empty only once (this may be the case if the get () function is used in a loop, and the empty queue indicates the end of the loop). Then the emptiness of the queue is not something strange or erroneous. Therefore, by using exceptions to denote the end of a queue, we expand the concept of exceptions as errors. On the other hand, the actions taken in the case of an empty queue are clearly different from the actions taken during the loop (that is, in the normal case).

The exception mechanism is less structured than local control structures such as if or for statements. It is also usually not as effective if an exceptional situation has arisen. Therefore, special situations should only be used when there is no good solution with more traditional control structures, or it is not possible at all. For example, in the case of an empty queue, you can perfectly use the value to signal this, namely the zero value of the pointer to the message string, which means that a special situation is not needed here. However, if from the queue class we received an int value instead of a pointer, then there might not be such a value denoting an empty queue. In this case, the get () function becomes equivalent to the indexing operation of $$ 9.1, and it is more attractive to represent an empty queue with an exception. The last consideration suggests that in the most general type template for a queue, you will have to use an exception to denote an empty queue, and the function working with the queue will be like this:

```
    void f (Queue <X> & q)
    {
            try {
                    for (;;) { // `` endless loop "
```

```
                                                    // interrupted by
exception
                        X m = q.get ();
                        // ...
                }
        }
        catch (Queue <X> :: Empty) {
                return;
        }
}
```

If the above loop is executed thousands of times, then it is likely to be more efficient than a regular loop checking the empty queue condition. If it runs only a few times, then a regular loop is almost certainly more efficient.

In a generic queue, an exception is used as a way to return from get (). Using exceptions as a return method can be an elegant way to complete search functions. This is especially suitable for recursive tree search functions. However, when applying special situations for such purposes, it is easy to go beyond reason and get an unintelligible program. Still, wherever this is really justified, one must adhere to the point of view that handling a special situation is handling an error. Error handling is by its very nature complex, so any methods that provide a clear representation of errors in the language and a way to handle them are valuable.

9.6 Setting the interface

Triggering or catching an exception affects the function relationship. Therefore, it makes sense to set in the function description a lot of special situations that it can trigger:

```
void f (int a) throw (x2, x3, x4);
```

This description states that f () can trigger exceptions x2, x3, and x4, as well as situations of all types derived from them, but it does not trigger any other situations. If a function lists its exceptions, then it gives some kind of guarantee to any function that calls it, namely, if it tries to trigger another exception, it will result in a call to unexpected (). The standard use of unexpected () is to call terminate (), which in turn usually calls abort (). Details are given in $$ 9.7.

Essentially the definition

```
void f () throw (x2, x3, x4)
{
            // some operators
}
```

is equivalent to this definition

```
void f ()
{
            try {
                        // some operators
            }
            catch (x2) {// restart
                        throw;
            }
            catch (x3) {// restart
                        throw;
            }
            catch (x4) {// restart
                        throw;
            }
            catch (...) {
                        unexpected ();
            }
}
```

The advantage of explicitly specifying special situations of a function in its description over an equivalent method, when checking for exceptions in the body of the function, is not only in a shorter notation. The main thing here is that the description of the function is included in its interface, which is visible to all calling functions. On the other hand, a function definition may not be universally available. Even if you have the source code for all the library functions, you usually don't want to study them often.

If a function does not specify its exceptions in the description, it is considered capable of triggering any exception.

```
int f (); // can trigger any exception
```

If the function will not trigger any special situations, it can be described by explicitly specifying an empty list:

```
int g () throw (); // does not trigger any special situations
```

It would seem logical for the function not to trigger any special situations by default. But then you would have to describe your own special situations for almost every function. This, as a rule, would require its re-translation, and besides, it would prevent communication with functions written in other languages. As a result, the programmer would try to disable the exception mechanism and write redundant statements to get around them. The user would consider such programs to be reliable, since he might not notice the substitution, but this would be completely unjustified.

9.6.1 Unexpected exceptions

If exceptions are not taken seriously enough, unexpected () may result, which is undesirable for all but debugging. You can avoid the unexpected () call if you organize the exception structure and interface description well. Alternatively, the unexpected () call can be intercepted and rendered harmless.

If the Y component is well designed, all of its exceptions can only be derived from one class, say Yerr. Therefore, if there is a description

```
class someYerr: public Yerr {/ * ... * /};
```

then the function described as

```
void f () throw (Xerr, Yerr, IOerr);
```

will pass any Yerr exception to the calling function. In particular, handling an exception like someYerr in f () will be reduced to passing it to a function calling f ().

There are times when terminating a program when an unexpected exception occurs is too strict a decision. Suppose function g () is written for non-network mode in a distributed system. Naturally, g () doesn't know anything about special network situations, so unexpected () is called whenever any of them occurs. So to use g () in a distributed system, you need to provide a network exception handler or rewrite g (). Assuming that it is impossible or undesirable to rewrite g (), the problem can be resolved by redefining the unexpected () function. The set_unexpected () function is used for this. First, we'll define a class that allows us to use the "request resources by initialization" method for the unexpected () functions:

```
typedef void (* PFV) ();
```

```
PFV set_unexpected (PFV);

class STC { // class for saving and restoring
        PFV old; // unexpected () functions
        public:
                STC (PFV f) {old = set_unexpected (f); }
                ~ STC () {set_unexpected (old); }
};
```

We will now define a function to replace unexpected () in our example:

```
void rethrow () {throw; } // restart all network
                                                    // special situations
```

Finally, we can give a variant of the g () function designed to work in network mode:

```
void networked_g ()
{
        STC xx (& rethrow); // now unexpected () calls rethrow ()
        g ();
}
```

In the previous section, you saw that unexpected () is potentially called from a catch (...) handler. This means that in our case there will be a repeated triggering of the special situation. Restarting when no exception was triggered results in a call to terminate (). Since the catch (...) handler is out of scope in which the network exception was triggered, an infinite loop cannot occur.

There is another, rather dangerous, solution, when they simply "close their eyes" to an unexpected special situation:

```
void muddle_on () {cerr << "we don't notice the special situation \ n"; }
        // ...
        STC xx (& muddle_on); // now the unexpected () action is rolled
up
                                                    // just print the
message
```

Overriding the unexpected () action in this way allows a function to return normally when it encounters an unexpected exception. Despite its obvious danger, this solution is used. For example, you can "close your eyes" to special situations in one part of the system and debug other parts of it. This approach can be useful in the process of debugging and development of a system ported from a programming language without special situations. Still, it is usually best if errors appear as early as possible.

Another solution is possible where the unexpected () call translates to triggering a Fail exception:

```
void fail () {throw Fail; }
        // ...
        STC yy (& fail);
```

With this solution, the calling function does not have to understand in detail the possible result of the called function: this function will either succeed (i.e., return normally) or fail (i.e., run Fail). The obvious disadvantage of this solution is that it does not take into account additional information that may accompany the exception. However, if necessary, it can be taken into account by transmitting information along with Fail.

9.7 Uncaught exceptions

If an exception is triggered and not caught, the terminate () function is called. It is also called when the exception handling system detects that the stack structure is broken, or when the destructor is called during the exception handling while unwinding the stack, and it tries to complete its work by triggering the exception.

The action of terminate () boils down to executing the very last function given as a parameter to set_terminate ():

```
typedef void (* PFV) ();
PFV set_terminate (PFV);
```

The set_terminate () function returns a pointer to the function that was specified as a parameter in the previous call to it.

The need for such a function as terminate () is explained by the fact that sometimes, instead of a mechanism for exceptions, more crude techniques are required. For example, terminate () can be used to terminate a process and possibly restart the system. This function serves as an emergency tool to apply when an exception handling strategy has failed and it is high time to apply a lower-level strategy.

The unexpected () function is used in a similar but less severe case, namely when a function triggered an exception not listed in its description. The unexpected () function is reduced to executing the very last function specified as a parameter to set_unexpected ().

By default, unexpected () calls terminate (), which in turn calls abort (). Most users are expected to be satisfied with such an agreement.

The terminate () function is not expected to return in the calling function.

Recall that a call to abort () indicates an abnormal program termination. The exit () function is used to exit the program normally. It returns a value that indicates to the surrounding system how correctly the program ended.

9.8 Other ways to handle errors

The mechanism of special situations is needed so that from one part of the program it is possible to report to another about the occurrence in the first "special situation". This assumes that the parts of the program are written independently of each other, and in the part that handles the exception, a meaningful reaction to the error is possible.

How should an exception handler be arranged? Here are several options:

```
int f (int arg)
{
        try {
                g (a rg);
        }
        catch (x1) {
                // fix the error and repeat
                g (arg);
        }
        catch (x2) {
                // perform calculations and return the result
                return 2;
        }
        catch (x3) {
                // pass the error
                throw;
        }
        catch (x4) {
            // fire another exception instead of x4
            throw xxii;
        }
        catch (x5) {
            // fix the error and continue with the next statement
        }
        catch (...) {
            // refuse to handle the error
            terminate ();
        }
        // ...
}
```

We indicate that variables from the scope containing the checked block of this handler are available in the handler. Variables described in other handlers or other checked blocks are of course not available:

```
void f ()
{
        int i1;
        // ...
        try {
            int i2;
            // ...
        }
        catch (x1) {
            int i3;
            // ...
        }
        catch (x4) {
            i1 = 1; // fine
            i2 = 2; // error: i2 is invisible here
            i3 = 3; // error: i3 is invisible here
        }
}
```

You need a general strategy to effectively use handlers in your program. All program components must use exceptions consistently and have a common part for error handling. The exception handling mechanism is non-local in nature, so it is important to stick to a general strategy. This suggests that an error handling strategy should be developed at the earliest stages of projects. In addition, this strategy should be simple (compared to the complexity of the entire program) and clear. It will be simply impossible to consistently pursue a complex strategy in such an inherently complex field of programming as error recovery.

First of all, you should immediately abandon the fact that one tool or one technique can be used to handle all errors. This will only complicate the system. A successful error-tolerant system must be built in a multi-tiered manner. At each level, you should handle as many errors as possible without breaking the structure of the system, leaving the handling of other errors to higher levels. The purpose of terminate () to support this approach, providing an emergency exit from a situation when the exception handling mechanism itself is violated, or when it is fully used, but the exception has not been caught. The unexpected () function is designed to get out of a situation where a protection based on the description of all exceptions did not work. Think of this as a firewall, i.e. a wall that surrounds each function and prevents the spread of the error. An attempt to carry out complete control in each function in order to ensure that the function will either succeed or fail, but in one of the defined and correct ways, cannot bring success. The reasons for this may vary for different programs, but for large programs, the following can be named:

1] the work that needs to be done to ensure the reliability of each function is too great and therefore cannot be done consistently enough;

2] there will be too large additional costs of memory and time, which will be unacceptable for the normal operation of the system (there will be a tendency to repeatedly check for the same error, which means that variables with correct values will be constantly checked);

3] functions written in other languages will not obey such restrictions ;

4] this concept of reliability is purely local and it complicates the system so much that it becomes an additional burden for its overall reliability.

However, breaking the program into separate subsystems that either complete successfully or fail, but in one of the definite and correct ways, is quite possibly important and even beneficial. Basic libraries, subsystems or key functions should have this property. The description of special situations should be included in the interfaces of such libraries or subsystems.

Sometimes you have to switch from one style of reaction to a mistake to another. For example, after calling the standard C function, you can check the value of errno and, possibly, trigger an exception, or, on the contrary, you can catch the exception and set the value of errno before exiting the standard function into the C program:

```
void callC ()
{
        errno = 0;
        cfunction ();
        if (errno) throw some_exception (errno);
}

void fromC ()
{
        try {
                c_pl_pl_function ();
        }
        catch (...) {
                errno = E_CPLPLFCTBLEWIT;
        }
}
```

With this style change, it is important to be consistent so that the change in error response is complete.

Error handling should be as strictly hierarchical as possible. If a dynamic error is detected in a function, then you do not need to contact the calling function for help to recover or allocate resources. With such calls, cyclical dependencies arise in the structure of the system, as a result of which it is more difficult to understand, and infinite loops may occur during processing and recovery from an error.

To keep the error-handling part of the program more streamlined, it is worth using simplistic techniques such as “requesting resources by initialization” and using the simplifying assumptions that “exceptions are errors”.

9.9 Exercises

1. (* 2) Generalize the STC class to a type template that allows you to store and set functions of different types.
2. (* 3) Extend the CheckedPtrToT class from $$ 7.10 to a type template where exceptions signal dynamic errors.
3. (* 3) Write a find function to search a binary tree for nodes by the value of a char * field. If a node is found with a field that has the value "hello", it should return a pointer to it. Use an exception to indicate a failed search.
4. (* 1) Define an Int class that is all the same as the built-in int type except that exceptions are thrown in this class instead of overflow or underflow . Hint: see $$ 9.3.2.
5. (* 2) Transfer the basic file operations from the standard C interface to your operating system: open, close, read and write. Implement them as C ++ functions with the same purpose as C functions, but trigger exceptions in case of errors .
6. (* 1) Write a complete template definition of the Vector type with the exceptions Range and Size. Hint: see $$ 9.3.
7. (* 1) Write a loop to calculate the sum of the elements of the vector defined in Exercise 6, and do not check the vector size. Why is this a bad decision?
8. (* 2.5) Suppose the Exception class is used as the base for all classes that define exceptions . What should it look like? What good could it be? What inconvenience can be caused by the requirement to use this class?
9. (* 2) Write a class or type template to help implement the callback.
10. (* 2) Write a Lock class for some system that allows parallel execution.
11. (* 1) Let the function

```
int main () {/ * ... * /}
```

Modify it to catch all exceptions, convert to error messages, and call abort (). Hint: The fromC () function from $$ 9.8 does not cover all cases.

CHAPTER 10. FLOWS

"Only what is visible is available"

B. Kernighan

There is no I / O facility in C ++. They are not needed, since such tools can be easily and elegantly created in the language itself. The stream I / O library described here implements a strict, generic, yet flexible and efficient way of character input and output of integers, real numbers, and character strings, and is also the basis for an extension designed to work with user-defined data types . The library user interface is located in the <iostream.h> file . This chapter focuses on the streaming library itself, some of the ways to work with it, and some techniques for implementing the library.

10.1 INTRODUCTION

The difficulty of designing and implementing standard I / O facilities for programming languages is widely known . Traditionally, I / O has relied solely on a small number of built-in data types. However, there are many user-defined data types in non-trivial C ++ programs, so it is necessary to provide the ability to I / O values of such types. Obviously, I / O should be simple, convenient, reliable, and, most importantly, adequate. So far, no one has found a solution that would satisfy everyone; therefore, it is necessary to enable the user to create other I / O means, as well as expand the standard I / O means, counting on a specific application.

The goal of creating C ++ was that the user could define new data types that would be as convenient and efficient to work with as with built-in types. Thus, it seems reasonable to require that C ++ I / O be programmed using the C ++ capabilities available to everyone. The streaming I / O facilities presented here are an attempt to meet these requirements.

The main task of streaming I / O is the process of converting objects of a certain type to a sequence of characters and vice versa. There are other I / O schemes, but this is the main one, and if we consider a character as just a set of bits, ignoring its natural relationship with the alphabet, then many binary I / O schemes can be reduced to it. Therefore, the programmer's essence of the problem is reduced to describing the connection between an object of a certain type and a typeless (which is essential) string.

The following sections describe the main parts of the C ++ streaming library:

0.2 Output: What appears to be output to an application is actually a conversion of objects such as int, char *, complex, or Employee_record to a sequence of characters. Describes the means for writing objects of built-in and user-defined data types.

0.3 Input: Functions for inputting characters, strings and values of built-in and user-defined data types are described .

0.4 Formatting: Often there are certain requirements for the type of output, for example, int must be printed in decimal digits, pointers in hexadecimal notation, and real numbers must be with an explicitly specified precision of a fixed size. The formatting functions and certain programming techniques for their creation, in particular, manipulators, are discussed .

0.5 Files and Streams: Every C ++ program can use three streams by default — standard output (cout), standard input (cin), and standard error (cerr). To work with any devices or files, you need to create streams and bind them to these devices or files. The mechanism of opening and closing files and linking files to streams is described .

0.6 I / O for C: Discusses the printf function in <stdio.h> for C and the relationship between the library for C and <iostream.h> for C ++.

Note that there are many independent implementations of the streaming I / O library, and the set of tools described here will only be a subset of the tools available in your library. They say that within any large program there is a small program that tends to break out. This chapter attempts to describe just a small stream I / O library that will allow you to understand the basic concepts of streaming I / O and introduce the most useful tools. Many programs can be written using only the tools described here; if there is

need for more complex tools, consult your C ++ manual for details . The header file <iostream.h> defines the stream library interface . Earlier versions of the streaming library used the <stream.h> file. If both files exist, <iostream.h> defines the complete

toolkit, and <stream.h> defines a subset that is compatible with earlier, less rich streaming libraries.

Naturally, to use the streaming library, you do not need to know the technique of its implementation, especially since the technique may be different for different implementations. However, the I / O implementation

is a task that dictates certain conditions, which means that the techniques found in the process of solving it can be applied to other tasks, and this solution itself is worth studying.

10.2 CONCLUSION

Strict typed and consistent handling of both built-in and user-defined types can be achieved by using a single overloaded function name for different inference operations. For example:

```
put (cerr, "x ="); // cerr - output error stream
put (cerr, x);
put (cerr, '\ n');
```

The type of the argument determines which function to call in each case. This approach is used in several languages, however, this is too long a record. By overloading the << operator to mean "put to", you can get a simpler notation and allow the programmer to output a sequence of objects in a single statement, like this:

```
cerr << "x =" << x << '\ n';
```

Here cerr stands for standard error stream. So, if x is of type int with the value 123, then the above operator will produce

```
x = 123
```

and another line terminator to standard error. Similarly, if x is a user-defined type complex with the value (1,2.4), then the specified operator will produce

```
x = (1,2.4)
```

to the cerr stream. This approach is easy to use as long as x is of the type for which the operator << is defined , and the user can simply redefine << for new types.

We used the inference operation to avoid the verbosity inevitable when using the inference function. But why exactly the symbol <<? It is impossible to invent a new token (see 7.2). An assignment operation was a candidate for input and output, but most people prefer input and output to be different. Moreover, the order of execution of the operation = is not

suitable, so cout = a = b means cout = (a = b). We tried to use the operations <and>, but the concept of "less than" and "more than" is so tightly attached to them that I / O operations with them in almost all cases were unreadable.

The operations << and >> do not seem to create such problems. They are asymmetric, which allows us to attribute to them the meaning of "in" and "from". They are not among the most commonly used built-in type operations, and the priority << is low enough to write arithmetic expressions as an operand without parentheses:

```
cout << "a * b + c =" << a * b + c << '\ n';
```

The parentheses are needed if the expression contains lower priority operations :

```
cout << "a ^ b | c =" << (a ^ b | c) << '\ n';
```

The left shift operation can be used in an output operation, but of course it must be in parentheses:

```
cout << "a << b =" << (a << b) << '\ n';
```

10.2.1 Inference of built-in types

To control the inference of built-in types, an ostream class is defined with the << (output) operation:

```
class ostream: public virtual ios {
        // ...
        public:
                ostream & operator << (const char *); // lines
                ostream & operator << (char);
                ostream & operator << (short i)
                        {return * this << int (i); }
                ostream & operator << (int);
                ostream & operator << (long );
                ostream & operator << (double);
                ostream & operator << (const void *); // pointers
                // ...
};
```

Naturally, the ostream class must have a set of operator << () functions for working with unsigned types.

The operator << function returns a reference to the ostream class from which it was called so that operator << can be reapplied to it . So, if x is of type int, then

```
cerr << "x =" << x;
```

understood as

```
(cerr.operator << ("x =")). operator << (x);
```

In particular, this means that if several objects are displayed using a single output operator, then they will be displayed in a natural order: from left to right.

The ostream :: operator << (int) function outputs integer values, and the ostream :: operator << (char) function outputs character values . Therefore the function

```
void val (char c)
{
        cout << "int ('" << c << "') =" << int (c) << '\ n';
}
```

prints integer character values and using the program

```
main ()
{
        val ('A');
        val ('Z');
}
```

will be printed

```
int ('A') = 65
int ('Z') = 90
```

This assumes ASCII character encoding, your machine may have a different result. Note that the character constant is of type char, so cout << 'Z' will print the letter Z, not the integer 90.

The ostream :: operator << (const void *) function will print the value of the pointer in a record that is more appropriate for the addressing system being used . Program

```
main ()
{
        int i = 0;
```

```
            int * p = new int (1);
            cout << "local" << & i
                                    << ", free store" << p << '\ n';
   }
```
will issue on the machine used by the author,

```
   local 0x7fffead0, free store 0x500c
```

For other addressing systems, there may be different conventions for the representation of pointer values.

We'll postpone the discussion of the base ios class until 10.4.1.

10.2.2 Inference of user-defined types

Consider a custom data type:

```
   class complex {
            double re, im;
            public:
                        complex (double r = 0, double i = 0) {re = r; im = i; }
                        f riend double real (complex & a) {return a.re; }
                        friend double imag (complex & a) {return a.im; }
                        friend complex operator + (complex, complex);
                        friend complex operator- (complex, complex);
                        friend complex operator * (complex, complex);
                        friend complex operator / (complex, complex);
                        // ...
   };
```

For the new type complex, the operation << can be defined as follows:

```
   ostream & operator << (ostream & s, complex z)
   {
            return s << '(' real (z) << ',' << imag (z) << ')';
   };
```

and use like operator << for built-in types. For example,

```
   main ()
   {
            complex x (1,2);
            cout << "x =" << x << '\ n';
   }
```

will issue

```
x = (1,2)
```

To define an inference operation on user-defined data types, you do not need to modify the description of the ostream class, nor do you need access to data structures hidden in the class description. The latter is very useful, since the description of the ostream class is among the standard header files, write access to which is closed to most users, and which they would hardly want to change, even if they could. This is also important for the reason that it provides protection against accidental corruption of these data structures. In addition, it is possible to change the ostream implementation without affecting user programs.

10.3 ENTER

Input is very similar to output. There is an istream class that implements the >> ("input from") operation for a small set of standard types. For custom types, you can define the operator >> function .

10.3.1 Introducing built-in types

The istream class is defined as follows:

```
class istream: public virtual ios {
        // ...
        public:
                istream & operator >> (char *); // line
                istream & operator >> (char &); // symbol
                istream & operator >> (short &);
                istream & operator >> (int &);
                istream & operator >> (long &);
                istream & operator >> (float &);
                istream & operator >> (double &);
                // ...
};
```

The operator >> input functions are defined like this:

```
istream & istream :: operator >> (T & tvar)
{
        // skip generic spaces
        // somehow read T in `tvar '
```

```
            return * this;
}
```

Now you can enter a sequence of integers, separated by spaces, into VECTOR using the function:

```
int readints (Vector <int> & v)
// return the number of integers read
{
            for (int i = 0; i <v.size (); i ++)
            {
                        if (cin >> v [i]) continue;
                        return i;
             }
            // too many integers for Vector size
            // need appropriate error handling
}
```

The appearance of a value with a type other than int terminates the input operation, and the input loop ends. So if we enter

```
1 2 3 4 5.6 7 8.
```

then the readints () function will read five integers

```
1 2 3 4 5
```

The dot remains the first character to be entered. Whitespace as defined in the C standard refers to a generic whitespace, i.e. space, tab, end of line, line feed, or carriage return. Checking for a generic space is possible using the isspace () function from the <ctype.h> file.

Alternatively, you can use the get () functions:

```
class istream: public virtual ios {
            // ...
            istream & get (char & c); // symbol
            istream & get (char * p, int n, char = 'n'); // line
};
```

They treat the generalized space as any other character and are intended for input operations where no assumptions are made about the characters being entered.

The istream :: get (char &) function takes one character into its parameter. Therefore, a character-by-character copy program can be written like this:

```
main ()
{
        char c;
        while (cin.get (c)) cout << c;
}
```

This notation looks asymmetrical, and the >> operation has a double named put () for printing characters , so you can write it like this:

```
main ()
{
        char c;
        while (cin.get (c)) cout.put (c);
}
```

A function with three parameters istream :: get () enters at least n characters into a character vector , starting at address p. Each time get () is called, all characters placed in the buffer (if any) are 0 terminated, so if the second parameter is n, then no more than n-1 characters are entered. The third parameter defines the character that ends the input. A typical use of the get () function with three parameters is to read a string into a buffer of a given size for further parsing, for example:

```
void f ()
{
        char buf [100];
        cin >> buf; // suspicious
        cin.get (buf, 100, '\ n'); // reliable
        // ...
}
```

The cin >> buf operation is suspicious because a string of more than 99 characters will overflow the buffer. If a terminating character is found, it remains the first character to be entered in the stream. This allows you to check the buffer for overflow:

```
void f ()
{
        char buf [100 ];
        cin.get (buf, 100, '\ n'); // reliable

  char c;
  if (cin.get (c) && c! = '\ n') {
                        // input string is larger than expected
            }
            // ...
   }
```
Naturally, there is an unsigned char version of get ().

The standard header file <ctype.h> defines several functions that are useful for handling on input:

```
   int isalpha (char) // 'a' .. 'z' 'A' .. 'Z'
   int isupper (char) // 'A' .. 'Z'
   int islower (char) // 'a' .. 'z'
   int isdigit (char) // '0' .. '9'
   int isxdigit (char) // '0' .. '9' 'a' .. 'f' 'A' .. 'F'
   int isspace (char) // '' '\ t' returns the end of the string
                                                   // and translation of
   the format
   int iscntrl (char) // control character in the range
                                                   // (ASCII 0..31 and
   127)
   int ispunct (char) // punctuation mark, other than
                                                   // above
   int isalnum (char) // isalpha () | isdigit ()
   int isprint (char) // visible: ascii '' .. '~'
   int isgraph (char) // isalpha () | isdigit () | ispunct ()
   int isascii (char c) {return 0 <= c && c <= 127; }
```

All of them, except for isascii (), work with a simple lookup, using the symbol as an index in the symbol attribute table. Therefore, instead of an expression like

```
   (('a' <= c && c <= 'z') || ('A' <= c && c <= 'Z')) // letter
```

which is not only tedious to write, but it can also be erroneous (on a machine with EBCDIC encoding, it specifies not only letters), it is better to use the call to the standard function isalpha (), which is also more efficient. As an example, here's the eatwhite () function, which reads generic whitespace from a stream:

```
   istream & eatwhite (istream & is)
```

```
{
        char c;
        while (is.get (c)) {
                if (isspace (c) == 0) {
                        is.putback (c);
                        break;
                }
        }
        return is;
}
```

It uses the putback () function, which returns a character to the stream, and it becomes the first to be read.

10.3.2 Stream states

Each stream (istream or ostream) has a specific state associated with it. Abnormal situations and errors are handled by checking and setting the status appropriately. You can find out the state of a thread using operations on the ios class:

```
class ios {     // ios is the base for ostream and istream
        // ...
        public:
                int eof () const; // reached the end of the file
                int fail () const; // next operation will fail
                int bad () const; // the stream is corrupted
                int good () const; // next operation will be
successful
                // ...
};
```

The last input operation is considered successful if the state is set to good () or eof (). If the state is good (), then the subsequent input operation may succeed, otherwise it will fail. Applying an input operation to a stream in a non-good () state is considered an empty operation. If the attempt to read into v fails, then the value of v has not changed (it will not change if v is of a type controlled by member functions from istream or ostream). The distinction between states specified as fail () or as bad () is difficult to grasp and only makes sense to input developers . If the state is fail (), then it is considered that the stream is not damaged, and no characters are missing; nothing can be said about the bad () state .

The values denoting these states are defined in the ios class:

```
class ios {
        // ...
        public:
                enum io_state {
                goodbit = 0,
                eofbit = 1,
                filebit = 2,
                badbit = 4,
        };
        // ...
};
```

The actual values of the states are implementation dependent, and the values shown are provided only to avoid syntactically incorrect constructs.

You can check the status of a stream as follows:

```
switch (cin.rdstate ()) {
        case ios :: goodbit:
                // last cin operation was successful
                break;
        case ios :: eofbit:
                // at the end of the file
                break;
        case ios :: filebit:
                // some analysis of the error
                // maybe not bad
                break;
        case ios :: badbit:
                // cin is possibly corrupted
                break;
}
```

Earlier implementations used global names for state values . This resulted in unwanted cluttering of the namespace, so new names are only available within the ios class. If you need to use the old names in combination with the new library, you can use the following definitions:

```
const int _good = ios :: goodbit;
const int _bad = ios :: badbit;
const int _file = ios :: filebit;
const int _eof = ios :: eofbit;
typedef ios :: io_state state_value;
```

Library developers should take care not to add new names to the global namespace. If enumeration members are part of the general library interface, they should always be used in the class with prefixes such as ios :: goodbit and ios :: io_state.

For a variable of any type for which the operations << and >> are defined , the copy cycle is written as follows:

```
while (cin >> z) cout << z << '\ n';
```

If a thread appears in a condition, then the state of the thread is checked, and the condition is satisfied (that is, its result is not 0) only for the state good (). It is in the above loop that the status of the istream is checked, which is the result of the cin >> z operation. To find out why a loop or condition failed, you need to check the state. This check for a stream is implemented using the cast operation (7.3.2).

Thus, if z is a character vector, then the above loop reads standard input and outputs one word for each line of standard output (that is, a sequence of characters that are not generic spaces). If z is of type complex, then complex numbers will be copied in this loop using the operations defined in 10.2.2 and 10.2.3 . A templated copy function for streams with arbitrary values can be written as follows:

```
complex z;
iocopy (z, cin, cout); // copy complex
double d;
iocopy (d, cin, cout); // copy double
char c;
iocopy (c, cin, cout); // copy char
```

Since it gets boring to check for the correctness of every I / O operation, then a common source of errors is precisely those places in the program where such control is essential. Inference operations are usually not validated, but sometimes they may fail. Stream I / O was designed with the principle of making exceptions easily accessible and thereby simplifying error handling in the I / O process.

10.3.3 Entering custom types

You can define an input operation for a user-defined type in exactly the same way as an output operation, but for an input operation it is essential that the second parameter has a reference type, for example:

```
istream & operator >> (istream & s, complex & a)
/ *
        input format is for complex; "f" stands for float:
                f
                (f)
                (f , f)
* /
{
        double re = 0, im = 0;
         char c = 0;
        s >> c;
        if (c == '(') {
                s >> re >> c;
                if (c == ',') s >> im >> c;
                if (c! = ')') s.clear (ios :: badbit); // set the state
        }
        else {
                s.putback (c);
                s >> re;
        }
        if (s) a = complex (re, im);
                return s;
}
```

While the error handling code is very compact , most errors are actually counted. The initialization of the local variable c is necessary so that a random value does not get into it, for example '(', in case of an unsuccessful operation. The last check of the flow state ensures that the parameter a will receive a value only on successful input.

The operation that sets the state of the stream is called clear () (here clear is clear, correct), because it is most often used to restore the state of the stream as good (); the default value for the ios :: clear () parameter is ios :: goodbit.

10.4 Formatting

All examples from 10.2 contained unformatted output, which was the transformation of an object into a sequence of characters specified by standard rules, the length of which is also determined by these rules. Programmers often need more advanced features. Thus, there is a need to control the amount of memory required for an output operation and the format used to output numbers. Likewise, you can control some aspects of the input.

10.4.1 The ios class

Most of the I / O controls are concentrated in the ios class , which is the base for ostream and istream. Essentially, this is where you control the communication between istream or ostream and the buffer used for I / O operations. It is the ios class that controls: how characters enter the buffer and how they are fetched from there. So, the ios class has a member that contains information about the number system used when reading or writing integers (decimal, octal or hexadecimal), the precision of real numbers, etc., as well as functions for checking and setting the values of variables that control the flow ...

```
class ios {
        // ...
        public:
                ostream * tie (ostream * s); // link input and
output
                ostream * tie (); // return "tie"
                int width (int w); // set field width
                int width () const;
                char fill (char); // set fill character
                char fill () const; // return the fill character
                long flags (long f);
                lon g flags () const;
                long setf (long setbits, long field);
                long setf (long);
```

```
                    long unsetf (long);
                    int precision (int); // set precision for float
                    int precision () const;
                    int rdstate (); const; // thread states, see $$
10.3.2
                    int eof () const;
                    int fail () const;
                    int bad () const;
                    int good () const;
                    void clear (int i = 0);
                    // ...
};
```

10.3.2 describes the functions that work with the state of the thread, the rest are given below.

10.4.1.1 Binding streams

The tie () function can establish and break the link between ostream and istream. Let's consider an example:

```
main ()
{
          String s;
          cout << "Password:";
          cin >> s;
          // ...
}
```

How can you ensure that the Password: prompt appears on the screen before the read operation is performed? Output to cout and input from cin are buffered independently, so Password: appears only when the program ends, when the output buffer is closed.

The solution is to bind cout and cin using the cin.tie (cout) operation. If ostream is associated with an istream, an output buffer is issued on every input to istream. Then the operations

```
cout << "Password:";
cin >> s;
```

are equivalent

```
cout << "Password:";
```

```
cout.flush ();
cin >> s;
```

Calling is.tie (0) breaks the connection between the is stream and the thread it was associated with, if any. Like other threading functions that set a specific value, tie (s) returns the previous value, i.e. the value of the associated thread before the call, or 0. Calling without the tie () parameter returns the current value.

10.4.1.2 Output fields

The width () function sets the minimum number of characters used in a subsequent operation to output a number or string. So as a result of the following operations

```
cout.width (4);
cout << '(' << 12 << ')';
```

we get the number 12 in a field of 4 characters, i.e.

```
( 12)
```

Filling the field with specified characters or alignment can be set using the fill () function, for example:

```
cout.width (4);
cout.fill ('#');
cout << '(' << "ab" << ')';
```

will print

```
(## ab)
```

By default, the field is filled with spaces, and the default field size is 0, which means "as many characters as needed". You can return the field size to its default value by calling

```
cout.width (0); // `` as many characters as needed "
```

The width () function sets the minimum number of characters. If more characters appear , they will all be printed, so

```
cout.width (4);
cout << '(' << "121212" << ") \ n";
```

will print

```
(121212)
```

The reason field overflow is allowed, rather than output truncation , is to avoid hanging on output. It's better to get a correct look and feel than a pretty one that is wrong.

The call to width () only affects one subsequent output operation , so

```
cout.width (4);
cout.fill ('#');
cout << '(' << 12 << "), (" << '(' << 12 << ") \ n";
```

will print

```
(## 12), (12)
```

but not

```
(## 12), (## 12)
```

as you might expect. Note, however, that if the effect was extended to all number and string output operations, the result would be even more unexpected:

```
(## 12 #), (## 12 #
)
```

By using the standard manipulator shown in 10.4.2.1, you can more elegantly set the size of the output field.

10.4.1.3 Format status

The ios class contains the format state, which is controlled by the flags () and setf () functions. In fact, these functions are needed to set or uncheck the following flags:

```
class ios {
            public:
                                                             // format control
flags:
                        enum {
                                    skipws = 01, // skip generic spaces for
input
                                                             // alignment field:
                                    left = 02, // add before value
                                    right = 04, // add after value
                                    internal = 010, // add between sign and
value
```

```
                                                  // base of integer:
                                  dec = 020, // octal
                                  oct = 040, // decimal
                                  hex = 0100, // hex
                                  showbase = 0200, // show base of
   integer
                                  showpoint = 0400, // print trailing
   zeros
                                  uppercase = 01000, // 'E', 'X', not 'e',
   'x'
                                  showpos = 02000, // '+' for positive
   numbers
                                                  // write a number of
   float type:
                                  scientific = 04000, // .dddddd Edd
                                  fixed = 010000, // dddd.dd
                                                  // dump to output
   stream:
                                  unitbuf = 020000, // after each
   operation
                                  stdio = 040000 // after each character
          } ;
          // ...
   };
```

The meaning of the flags will be explained in the following sections. The exact meaning of the flags is implementation dependent and is given here only to avoid syntactically incorrect constructs.

Defining an interface as a collection of flags and the operations to set or clear them is a time-honored, albeit somewhat outdated, technique. Its main advantage is that the user can put together a set of flags, for example, like this:

```
   const int my_io_options =
   ios :: left | ios :: oct | ios :: showpoint | ios :: fixed;
```

Such a set of flags can be set as a parameter of one operation

```
   cout.flags (my_io_options);
```

and also just pass between functions of one program:

```
void your_function (int ios_options);

void my_function ()
{
        // ...
        your_function (my_io_options);
        // ...
}
```

Many flags can be set using the flags () function, for example:

```
void your_function (int ios_options)
{
        int old_options = cout.flags (ios_options);
        // ...
        cout.flags (old_options); // reset options
}
```

The flags () function returns the old value of the set of flags. This allows you to reset the values of all flags as shown above, and also set the value of an individual flag. For example call

```
myostream.flags (myostream.flags () | ios :: showpos);
```

causes the myostream class to return positive numbers with a + sign and, at the same time, does not change the values of other flags. The old value of the set of flags is obtained , to which it is added using the operation | showpos flag. The setf () function does the same, so the equivalent notation is

```
myostream.setf (ios :: showpos );
```

Once set, the flag is retained until explicitly canceled.

Still, controlling I / O by setting and unsetting flags is a crude and error-prone decision. Unless you study your reference manual carefully and use flags only in simple cases, as you do in the following sections, then it is better to use manipulators (described in 10.4.2.1). It is better to learn techniques for working with thread state by implementing a class than by studying the interface of the class.

10.4.1.4 Displaying integers

Reception of setting a new value of a set of flags using the operation | and the flags () and setf () functions only work when one bit determines the value of the flag. The situation is different when specifying the number system of integers or the type of issue of real ones. Here, the value that determines the type of dispensing cannot be set with one bit or a combination of individual bits.

The solution in <iostream.h> is to use a version of the setf () function that takes a second "pseudo-parameter" that indicates which flag we want to add to the new value.

Therefore, the appeal

```
cout.setf (ios :: oct, ios :: basefield); // octal
cout.setf (ios :: dec, ios :: basefield); // decimal
cout.setf (ios :: hex, ios :: basefield); // hexadecimal
```

set the number system without affecting the other components of the thread state . If the number system is installed, it is used before explicit reinstallation, so

```
cout << 1234 << "; // decimal by default
cout << 1234 << ";
cout.setf (ios :: oct, ios :: basefield); // octal
cout << 1234 << ' ';
cout << 1234 << ";
cout.setf (ios :: hex, ios :: basefield); // hexadecimal
cout << 1234 << ";
cout << 1234 << ";
```

will print

```
1234 1234 2322 2322 4d2 4d2
```

If it becomes necessary to specify the base system for each returned number, set the showbase flag. Therefore, adding before the above calls

```
cout.setf (ios :: showbase);
```

we'll get

```
1234 1234 02322 02322 0x4d2 0x4d2
```

The standard manipulators shown in $$ 10.4.2.1 offer a more elegant way of defining the numeral system when printing integers.

10.4.1.5 Aligning margins

With calls to setf (), you can control the arrangement of characters within the field:

```
cout.setf (ios :: left, ios :: adjustfield); // left
cout.setf (ios :: right, ios :: adjustfield); // right
cout.setf (ios :: internal, ios :: adjustfield); // internal
```

The alignment will be set on the output field defined by the ios :: width () function , without affecting other components of the flow state.

The alignment can be set as follows:

```
cout.width (4);
cout << '(' << -12 << ") \ n";
cout.width (4);
cout.setf (ios :: left, ios :: adjustfield);
cout << '(' << -12 << ") \ n";
cout.width (4);
cout.setf (ios :: internal, ios :: adjustfield);
cout << '(' << -12 << "\ n";
```

what will give

```
( -12)
(-12 )
(- 12)
```

If the internal alignment flag is set, characters are appended between sign and value. As you can see, right alignment is standard .

10.4.1.6 Output of floating numbers.

The output of real values is also controlled by functions that work with the state of the thread. In particular, appeals:

```
cout.setf (ios :: scientific, ios :: floatfield);
cout.setf (ios :: fixed, ios :: floatfield);
cout.setf (0, ios :: floatfield); // go back to standard
```

set the type of printing real numbers without changing other components of the flow state. For example:

```
cout << 1234.56789 << '\ n';
cout.setf (ios :: scientific, ios :: floatfield);
cout << 1234.56789 << '\ n';
cout.setf (ios :: fixed, ios :: floatfield);
```

```
cout << 1234.56789 << '\ n';
```

will print

```
1234.57
1.234568e + 03
1234.567890
```

After the dot, n digits are printed, as specified in the call

```
cout.precision (n)
```

The default n is 6. A call to precision affects all floating point I / O operations until the next call to precision, so

```
cout.precision (8);
cout << 1234.56789 << '\ n';
cout << 1234.56789 << '\ n';
cout.precision (4);
cout << 1234.56789 << '\ n';
cout << 1234.56789 << '\ n';
```

will issue

```
1234.5679
1234.5679
1235
1235
```

Note that rounding is done, not decimal discarding.

The standard manipulators, introduced in $$ 10.4.2.1, offer a more elegant way of specifying the output format of real.

10.4.2 Manipulators

These include a variety of operations that must be applied immediately before or immediately after an I / O operation. For example:

```
cout << x;
cout.flush ();
cout << y;
cin.eatwhite ();
ci n >> x;
```

If you write individual operators as above, then the logical connection between the operators is not obvious, and if the logical connection is lost, the program is more difficult to understand.

The idea of manipulators allows operations such as flush () or eatwhite () to be inserted directly into the list of I / O operations. Consider the flush () operation. You can define a class with operator << () that calls flush ():

```
class Flushtype {};

ostream & operator << (ostream & os, Flushtype)
{
        return flush (os);
}
```

define an object of this type

```
Flushty pe FLUSH;
```

and get the buffer to be issued by including FLUSH in the list of objects to be displayed :

```
cout << x << FLUSH << y << FLUSH;
```

An explicit link has now been established between the output and flush operations . However, it will quickly get boring to define a class and object for each operation that we want to apply to a streamed output operation. Fortunately, you can do better. Consider a function like this:

```
typedef ostream & (* Omanip) (ostream &);

ostream & operator << (ostream & os, Omanip f)
{
        return f (os);
}
```

Here, the output operation uses parameters of type "a pointer to a function that takes an ostream & argument and returns ostream &". Noticing that flush () is a function of type "a function with an argument ostream & and returning ostream &", we can write

```
cout << x << flush << y << flush;
```

receiving a call to the flush () function. In fact, in the <iostream.h> file , the flush () function is described as

```
ostream & flush (ostream &);
```

and the class has an operator << operation that uses a function pointer as above:

```
class ostream: public virtual ios {
            // .. .
            public:
                        ostream & operator << (ostream & ostream & (*)
(ostream &));
                        // ...
};
```

In the line below, the buffer is pushed to the cout stream twice at the appropriate time:

```
cout << x << flush << y << flush;
```

Similar definitions exist for the istream class:

```
istr eam & ws (istream & is) {return is.eatwhite (); }

class istream: public virtual ios {
            // ...
            public:
                        istream & operator >> (istream &, istream & (*)
(istream &));
                        // ...
};
```

so in the line

```
cin >> ws >> x;
```

really generalized spaces will be stripped before attempting to read into x. However, since by default for the >> operation, spaces are "eaten" anyway , this use of ws () is redundant.

Parameter manipulators are also used. For example, a desire may appear using

```
cout << setprecision (4) << angle;
```

print the value of the real variable angle to four decimal places after the dot.

To do this, you need to be able to call a function that will set the value of a variable that controls the precision of real ones in the stream. This is achieved by defining setprecision (4) as an object that can be "inferred" using operator << ():

```
class Omanip_int {
```

```
            int i;
            ostream & (* f) (ostream &, int);
            public:
                        Omanip_int (ostream & (* ff) (ostream &, int), int ii)
                                    : f (ff), i (ii) {}
                        friend ostream & operator << (ostream & os, Oma nip
   & m)
                                    {return mf (os, mi); }
   };
```

The Omanip_int constructor stores its arguments in i and f, and operator << calls f () with the i parameter. Objects of such classes are often called function object. To result in a string

```
   cout << setprecision (4) << angle
```

was what we wanted, it is necessary that the call to setprecision (4) creates an unnamed object of the Omanip_int class containing the value 4 and a pointer to a function that sets the value of the variable specifying the precision of real ones in the ostream :

```
   os tream & _set_precision (ostream &, int);

   Omanip_int setprecision (int i)
   {
            return Omanip_int (& _ set_precision, i);
   }
```

Given the definitions made, operator << () will result in a call to precision (i).

It's tedious to define classes like Omanip_int for all argument types, so let's define a template like:

```
   template <class T> class OMANIP {
            T i;
            ostream & (* f) (ostream &, T);
            public:
                        OMANIP (ostream (* ff) (ostream &, T), T ii)
                                    : f (ff), i (ii) {}
                        friend ostream & operator << (ostream & os, OMANIP
   & m)
```

```
                                    {return mf (os , mi)}
};
```
With OMANIP, the precision setting example can be shortened as follows:
```
ostream & precision (ostream & os, int)
{
        os.precision (i);
        return os;
}

OMANIP <int> setprecision (int i)
{
        return OMANIP <int> (& precision, i);
}
```
The file <iomanip.h> contains a template of type OMANIP, its counterpart for istream is a template of type SMANIP, and SMANIP is a twin for ioss. Some of the standard handles offered by the streaming library are described below. Note that the programmer can define new manipulators he needs without affecting the istream, ostream, OMANIP, or SMANIP definitions.

The idea of manipulators was proposed by A. Koenig. It was inspired by the layout routines of the Algol68 I / O system. This technique has many interesting applications beyond I / O. Its essence is that an object is created that can be passed anywhere and which is used as a function. Passing an object is more flexible because the execution details are partly determined by the creator of the object and partly by the person who is accessing it.

10.4.2.1 Standard I / O Handlers

These are the following manipulators:
```
// Simple manipulators:
ios & oct (ios &); // in octal notation
ios & dec (ios &); // in decimal notation
ios & hex (ios &); // in hexadecimal notation
ostream & endl (ostream &); // add '\ n' and print
ostream & ends (ostream &); // add '\ 0' and print
ostream & flush (ostream &); // issue a stream
istream & ws (istream &); // remove generic spaces
```

```
// Manipulators have parameters:

SMANIP <int> setbase (int b);
SMANI P <int> setfill (int f);
SMANIP <int> setprecision (int p);
SMANIP <int> setw (int w);
SMANIP <long> resetiosflags (long b);
SMANIP <long> setiosflags (long b);
```

For example,

```
cout << 1234 << "
              << hex << 1234 << "
              << oct << 1234 << endl;
```

will print

```
1234 4d2 2322
```

and

```
cout << setw (4) << setfill ('#') << '(' << 12 << ") \ n";
cout << '(' << 12 << ") \ n";
```

will print

```
(##12)
(12)
```

Don't forget to include the <iomanip.h> file if you are using parameter manipulators .

10.4.3 ostream members

There are only a few functions in the ostream class for controlling the output, most of these functions are in the ios class.

```
class ostream: public virtual ios {
        // ...
        public:
                ostream & flush ();
                ostream & seekp (streampos);
                ostream & seekp (streamoff, seek_dir);
                streampos tellp ();
                // ...
};
```

As we said, the flush () function empties the buffer onto the output stream. The rest of the functions are used to position the ostream when writing. The ending with the letter p indicates that exactly the position is used when writing characters to the given stream. Of course, these functions only make sense if the stream is attached to something that can be positioned, such as a file. The streampos type represents the position of a character in the file, and the streamoff type represents an offset from the position specified by seek_dir. They are all defined in the ios class:

```
class ios {
        // ...
        enum seek_dir {
                beg = 0, // from the beginning of the file

                cur = 1, // from the current position in the file

                end = 2 // from the end of the file
        };
        // ...
};
```

Positions in the stream are counted from 0, as if the file were an array of n characters:

```
ch ar file [n-1];
```

and if fout is attached to file then

```
fout.seek (10);
fout << '#';
```

will place # in file [10].

10.4.4 istream Members

As with ostream, most of the formatting and input manipulation functions are not in the iostream class, but in the base ios class.

```
class istream: public virtual ios {
        // ...
        public:
                int peek ()
                istream & putback (char c);
                istream & seekg (streampos);
```

```
                    istream & seekg (streamoff, seek_dir);
                    streampos tellg ();
                    // ...
};
```

The positioning functions work like their ostream counterparts. The ending with the letter g indicates that exactly the position is used when entering characters from the given stream. The letters p and g are needed because we can derive iostreams from ostream and istream, and we need to keep track of the input and output positions in it.

By using the peek () function, the program can find out the next character to be input without affecting the result of the subsequent reading. With the putback () function, as shown in $$ 10.3.3, you can put an unneeded character back into the stream so that it can be read at a different time.

10.5 Files and streams

Below is a program to copy one file to another. The file names are taken from the command line of the program:

```
#include <fstream.h>
#include <libc.h>

void error (char * s, char * s2 = "")
{
        cerr << s << " << s2 << '\ n';
        exit (1);
}

int main (int argc, char * argv [])
{
        if (argc! = 3) error ("wrong number of arguments");
        ifstream from (argv [1]);
        if (! from) error ("cannot open input file", argv [1]);
        ostream to (argv [2]);
        if (! to) error ("cannot open output file", argv [2]);
        char ch;
        while (from.get (ch)) to.put (ch);
        if (! from.eof () || to.bad ())
```

```
                error ("something strange happened");
        return 0;
}
```

To open the output file, an object of the ofstream class is created - the output file stream, using the file name as an argument. Likewise, to open an input file, an object of class ifstream is created — the input file stream, which also takes the filename as an argument. In both cases, you should check the state of the created object to make sure that the file was opened successfully, and if it is not, the operations will not complete successfully, but correctly.

By default, ifstream is always opened for reading, and ofst ream is always opened for writing. In ostream and istream, an optional second argument can be used to specify different opening modes:

```
class ios {
        public:
                // ...
                enum open_mode {
                        in = 1, // open for reading
                        out = 2, // open as holiday
                        ate = 4, // open and move to the end of the
file
                        app = 010, // add
                        trunc = 020, // truncate file to zero
length
                        nocreate = 040, // fail if file doesn't
exist
                        noreplace = 0100 // fail if file exists
                };
                // ...
};
```

The actual values for open_mode and their meaning are likely to be implementation dependent . Please see your library manual for details, or experiment. The comments provided may clarify their purpose. For example, you can open a file with the condition that the open operation will not be performed if the file does not already exist:

```
void f ()
```

```
{
            ofstream mystream (name, ios :: out | ios :: nocreate);
            if (ofstream.bad ()) {
                        // ...
            }
            // ...
}
```

You can also open the file immediately for reading and writing:

```
fstream dictionary ("concordance", ios :: i n | ios :: out);
```

All ostream and ostream operations can be applied to fstream. In fact, the fstream class derives from iostream, which in turn derives from istream and ostream. The reason the buffering and formatting information for ostream and istream is in the ios virtual base class is to make this whole sequence of derived classes work. For the same reason, the positioning operations in istream and ostream have different names - seekp () and seekg (). The iostream has separate read and write positions.

10.5.1 Closing Streams

A file can be closed explicitly by calling close () on its stream:

```
mystream.close ();
```

But this is implicitly done by the stream destructor, so an explicit call to close () may be necessary if the file needs to be closed before reaching the end of the stream's scope.

This raises the question of how an implementation can ensure that the predefined cout, cin and cerr streams are created before they are first used and only closed after the last use. Of course, different implementations of the stream library from <iostream.h> may solve this problem in different ways. In the end, the solution is up to the implementation and should be hidden from the user. There is only one method shown here , used in only one implementation, but it is general enough to ensure the correct order of creation and destruction of global objects of different types.

The basic idea is to define a helper class that essentially acts as a counter keeping track of how many times <iostream.h> has been included in separately compiled program files:

```
class Io_init {
```

```
            static int count;
            // ...
            public:
                        Io_init ();
                        ^ Io_init ();
};

static Io_init io_init;
```

Each program file has its own object named io_init. The constructor for io_init objects uses Io_init :: count as the first indication that the actual initialization of the I / O stream library global objects has been done exactly once:

```
Io_init :: Io_init ()
{
            if (count ++ == 0) {
                        // initialize cout
                        // initialize cerr
                        // initialize cin
                        // etc.
            }
}
```

Conversely, the destructor for io_init objects uses Io_count as a final indication that all streams are closed:

```
Io_init :: ^ Io_init ()
{
            if (--count == 0) {
                        // clear cout (reset, etc.)
                        // clear cerr (reset, etc.)
                        // clear cin
                        // etc.
            }
}
```

This is a common technique for working with libraries that require initialization and deletion of global objects. For the first time in C ++ it was applied by D. Schwartz. In systems where all programs are located in main memory during execution , this technique is not interfered with. If this is not the case, then the overhead associated with calling each program file into memory to perform initialization functions will be noticeable. As always, it is best to avoid global objects whenever possible. For classes in which each operation is significant in terms of the amount of work to be done, to guarantee initialization, it would be wise to check these first signs (like Io_init :: count) with each operation. However, for streams, this would be unnecessarily wasteful.

10.5.2 String streams

As shown, a stream can be linked to a file, i.e. an array of characters that is not stored in main memory, but, for example, on disk. Similarly , a stream can be bound to an array of characters in main memory. For example, you can use the output string stream ostrstream to format messages that should not be printed immediately:

```
char * p = new char [message_size];
ostrstream ost (p, message_size );
do_something (arguments, ost);
display (p);
```

Using standard output operations, the do_something function can write ost to the stream, pass ost to subordinate functions, and so on. Overflow control is not needed, because ost knows its size and, when filled, will go to the state defined by fail (). The display function can then send the message to the "real" output stream. This technique is most appropriate when the final output operation is intended to be written to a device that is more complex than the traditional line- oriented output device. For example, text from ost can be placed in a fixed area on the screen.

Likewise, istrstream is an input string stream that reads from a null-terminated sequence of characters:

```
void word_per_line (char v [], int sz)
/ *
        print "v" with size "sz" one word per line
* /
{
        istrstream ist (v, sz); // create an istream for v
        char b2 [MAX]; // longer than longest word
        while (ist >> b2) cout << b2 << "\ n";
}
```

A trailing zero is considered the end of the file.

String streams are described in the <strstream.h> file.

10.5.3 Buffering

All I / O operations have been defined without any connection to the file type , but you cannot work the same with all devices without considering the buffering algorithm. Obviously, an ostream bound to a character string needs a different buffer than an ostream bound to a file. Such issues are solved by creating different buffers for streams of different types during initialization . But there is only one set of operations on these types of buffers, so there are no functions in ostream whose code takes into account the difference in buffers. However, the functions that monitor overflow and access to an empty buffer are virtual. This is a good example of using virtual functions to work consistently with logically equivalent but differently implemented structures, and they do quite well with the required buffering algorithms. The stream buffer description in the <iostream.h> file may look like this:

```
class streambuf { // stream buffer management
        protected:
                char * base; // start of buffer
                char * pptr; // next free byte
                char * gptr; // next filled byte
                char * eptr; // one of the pointers to the end of the
buffer
                char alloc; // buffer allocated with "new"
                // ...
                // Empty the buffer:
                // Return EOF on error, 0 - luck
                virtual int overflow (int c = EOF);
                // Fill the buffer:
                // Return EOF on error or end of input stream,
                // otherwise return the next character
                virtual int underflow ();
                // ...
        public:
                streambuf ();
```

```
                    streambuf (char * p, int l);
                    virtual ~ streambuf ();
                    int snextc () // get the next character
                    {
                              return (++ gptr == pptr)? underflow (): * gptr
& 0377;
                    }
                    int allocate (); // allocate memory for the buffer
                    // ...
};
```

The implementation details of the streambuf class are provided here for completeness only . It is not assumed that there are publicly available implementations using these particular names. Notice the pointers defined here that control the buffer; with their help, simple character-by-character stream operations can be defined as efficiently as possible (and moreover, once) as substitution functions. Only the overflow () and underflow () functions require their own implementation for each buffering algorithm, for example:

```
cl ass filebuf: public streambuf {
          protected:
                    int fd; // file descriptor
                    char opened; // sign of file opening
          public:
                    filebuf () {opened = 0; }
                    filebuf (int nfd, char * p, int l)
                              : streambuf (p, l) {/ * ... * /}
                    ~ filebuf () { close (); }
                    int overflow (int c = EOF);
                    int underflow ();
                    filebuf * open (char * name, ios :: open_mode om);
                    int close () {/ * ... * /}
                              // ...
};

int filebuf :: underflow () // fill the buffer from "fd"
{
          if (! opened || allocate () == EOF) return EOF;
```

```
            int cou nt = read (fd, base, eptr-base);
            if (count <1) return EOF;
            gptr = base;
            pptr = base + count;
            return * gptr & 0377; // & 0377 prevents sign multiplication
    }
```

See the streambuf implementation manual for further details .

10.6 I / O in C

Since the text of C and C ++ programs is often confused, sometimes C ++ stream I / O and the printf family I / O functions for the C language are confused . Since C functions can be called from a C ++ program, many people prefer to use the more familiar C I / O functions.

For this reason, a basic C I / O function will be given here. Typically, C and C ++ I / O operations can occur alternately at the row level. Shuffling them at the I / O level is possible for some implementations, but such a program may not be portable. Some C ++ streaming library implementations require a call to the static member function ios :: sync_with_stdio () when allowing I / O in C.

In general, stream output functions have the advantage over the standard C printf () function that stream functions have a certain type of reliability and uniformly define the output of objects of predefined and user-defined types.

The main function of C output is

```
int printf (const char * format, ...)
```

and it outputs an arbitrary sequence of parameters in the format specified by the format string. The formatting string consists of two types of objects: simple characters that are simply copied into the output stream, and conversion specifications, each of which converts and prints the next parameter. Each conversion specification starts with a% character, for example

```
printf ("there were% d members present.", no_of_members);
```

Here,% d indicates that no_of_members should be treated as an integer and printed as an appropriate sequence of decimal digits. If no_of_members == 127 then it will print

```
there were 127 members present.
```

The set of conversion specifications is large enough to provide more printing flexibility. The% character can be followed by:

an optional minus sign specifying left justification in the specified field for the converted value;

l is an optional string of numbers that specifies the width of the field; if the converted value contains fewer characters than the line width, then it will be padded to the field width with spaces on the left (or on the right, if the left justification specification is given); if the field width line starts with zero, then padding will be carried out with zeros, not spaces;

.. the optional dot character is used to separate the field width from the following string of numbers;

l is an optional string of digits that specifies a precision that specifies the number of digits after the decimal point for values in an e or f specification, or specifies the maximum number of printable characters in a string;

: can be used to specify field width or precision * instead of a string of numbers. In this case, there must be an integer parameter that contains the field width or precision value;

ı optional character h indicates that the subsequent specification d, o, x, or u refers to a short integer type parameter;

the optional l character indicates that the subsequent specification d, o, x, or u refers to a long integer parameter;

⁄o means that you want to print the% character itself; the parameter is not needed;

: is a character indicating the type of conversion required. Conversion symbols and their meanings are as follows:

d Integer parameter is output in decimal notation;

o Integer parameter is given in octal notation;

x Integer parameter is output in hexadecimal notation;

f A real or double precision parameter is returned in decimal notation like [-] ddd.ddd, where the number of digits after the dot is equal to the precision specification for the parameter. If no precision is specified, six digits are printed; if precision is explicitly set to 0, the period and numbers after it are not printed;

e Real or double precision parameter is given in decimal notation like [-] d.ddde + dd; there is one digit before the dot, and the number of digits after the dot equals the precision specification for the parameter; if it is not specified six digits are printed;

g A real or double precision parameter is printed according to the d, f, or e specification that gives more precision with a smaller field width;

c The character parameter is printed. Null characters are ignored;

s The parameter is considered a string (character pointer), and characters are printed from the string to a null character or until the number of characters is equal to the precision specification; but if the precision is 0 or unspecified, all characters up to zero are printed;

p The parameter is considered a pointer and its appearance on printing depends on the implementation;

u An unsigned integer parameter is printed in decimal notation.

A non-existent field or a field with a smaller width than the real one will truncate the field. Padding with spaces occurs only if the field width specification is greater than the actual width. A more complex example follows:

```
char * src_file_na me;
int line;
char * line_format = "\ n # line% d \"% s \ "\ n";
main ()
{
        line = 13;
        src_file_name = "C ++ / main.c";
        printf ("int a; \ n");
        printf (line_format, line, src_file_name);
        printf ("int b; \ n");
}
```

which is printed

```
int a;
#line 13 "C ++ / main.c"
```

```
int b;
```

Using printf () is unreliable in the sense that there is no type checking. So, below is a known way to get an unexpected result - printing a junk value or worse:

```
char x;
// ...
printf ("bad input char:% s", x);
```

However, these functions provide a lot of flexibility and are familiar to C programmers.

As usual, getchar () allows you to read characters from the input stream in a familiar way :

```
int i ;:
while ((i = getchar ())! = EOF) {// character input C
          // use i
}
```

Note that for a legal comparison with an EOF of type int when checking for the end of the file, the result of getchar () must be placed in a variable of type int, not char.

For details on C I / O refer to your C manual or Kernighan and Ritchie's book The C Programming Language.

10.7 Exercises

1. (* 1.5) When reading a file of real numbers, make complex numbers from pairs of read numbers, write down complex numbers.
2. (* 1.5) Define the type name_and_address (type_and_address). Define << and >> for it. Write a program for copying stream objects name_and_address.
3. (* 2) Develop several functions for requesting and reading data of different types. Sentences: integer, real number, file name, mailing address, date, personal information, etc. Try to make them robust.
4. (* 1.5) Write a program that prints: (1) lowercase letters, (2) all letters, (3) all letters and numbers, (4) all characters included in an identifier in your version of C ++, (5) all punctuation marks, (6) integer values for all control characters, (7) all generalized spaces,

(8) integer values for all generalized spaces, and finally (9) all displayed characters.

5. (* 4) Implement the C Standard I / O Library (<stdio.h>) with the C ++ Standard I / O Library (<iostream.h>).

6. (* 4) Implement the C ++ Standard I / O Library (<iostream.h>) with the C Standard I / O Library (<stdio.h>).

7. (* 4) Implement the C and C ++ libraries so that they can be used simultaneously.

8. (* 2) Implement the class overloaded with the [] operation to allow arbitrary reading of characters from the file.

9. (* 3) Repeat Exercise 8, but make sure that operation [] is applicable for reading and writing. Hint: let [] return a "type descriptor" object, for which assignment means: assign via the file descriptor, and implicit casting to the type char means reading the file using the descriptor.

10. (* 2) Repeat Exercise 9, allowing the [] operation to index objects of arbitrary types, not just symbols.

11. (* 3.5) Think over and implement the format input operation. Use a spec string to specify the format as in printf (). It should be possible to try to apply multiple specifications to the same input to find the required format. The formatted input class must derive from the istream class.

12. (* 4) Come up with (and implement) better input formats.

13. (** 2) Define a based manipulator for output with two parameters: number system and an integer value, and print the integer in the representation determined by the number system. For example, based (2.9) will print 1001.

14. (** 2) Write a "miniature" I / O system that implements the classes istream, ostream, ifstream, ofstream and provides functions such as operator << () and operator >> () for integers, and operations such as open () and close () for files. Use exceptions , not state variables, to report errors.

15. (** 2) Write a manipulator that enables and disables character echo .

CHAPTER 11. DESIGN AND DEVELOPMENT

"There is no silver bullet."
- F. Brooks

This chapter discusses approaches to software development. The discussion touches on both the technical and sociological aspects of the software development process. The program is viewed as a model of reality in which each class represents a specific concept. A key design challenge is to define the accessible and secure portions of the class interface from which the various parts of the program are defined . Defining these interfaces is an iterative process, usually requiring experimentation. The emphasis is on the important role of design and organizational factors in the software development process.

11.1 Introduction

Building any non-trivial software system is a difficult and often exhausting task. Even for an individual programmer, the actual recording of program statements is only part of the whole work. Usually, analyzing the entire problem, designing the program as a whole, documentation, testing, maintaining and managing all of this overshadows the task of writing and debugging individual parts of the program. Of course, you can designate all these activities as "programming" and then quite reasonably say: "I do not design, I only program." But whatever the name of the individual activities, it is sometimes important to focus on them separately, just as sometimes it is important to consider the whole process as a whole. In an effort to quickly bring the system to delivery, neither the details nor the big picture should be overlooked, although this is quite often the case. This chapter focuses on those parts of the program development process that are not related to writing and debugging individual pieces of software . The discussion here is less precise and detailed than in all other parts of the book, which deal with specific features of the language or specific programming techniques. This is inevitable, since there are no ready-made recipes for creating good programs. Detailed "how" recipes may only exist for certain well-developed applications, not for broad enough application areas. In programming, there are no substitutes for intelligence, experience, and taste. Therefore, in this chapter you will find only general recommendations, alternative approaches and cautious conclusions.

The complexity of this topic stems from the abstract nature of programs and the fact that techniques applicable to small projects (say, a 10,000 line program created by one or two people) do not apply to medium or large projects. For this reason, we sometimes provide examples from less abstract engineering disciplines, not just programming. Let us remind you that "proof by analogy" is a fraudulent practice and analogies are only used as an example here. Design concepts, formulated using certain C ++ constructs and illustrated by examples, will be discussed in Chapters 12 and 13. The recommendations proposed in this chapter are reflected both in the C ++ language itself and in solving specific programming problems throughout the book. ...

Again, because of the tremendous variety of applications, programmers, and environments in which a software system develops , you cannot expect every conclusion drawn here to be directly applicable to your task. These conclusions are applicable in many very different cases, but they cannot be considered universal laws. Look at them with a healthy dose of skepticism.

C ++ can simply be used as the best option for C. However, in doing so, we are not using the most powerful features of C ++ and certain programming techniques in it, so we realize only a small fraction of the potential benefits of C ++. This chapter introduces a design approach that takes full advantage of the abstract data and object programming capabilities of C ++. This approach is commonly referred to as object-oriented design. Chapter 12 discusses basic C ++ programming techniques, warns against dubious ideas that there is only one "correct" way to use C ++, and that every C ++ tool should be used in any program to get the maximum benefit ($$ 12.1).

Here are some of the basic principles discussed in this chapter:

- Of all the issues related to the software development process , the most important is to be clearly aware of what you are trying to create.
- A successful software development process is a long process.
- The systems that we create tend to the limit of complexity in relation to both the creators themselves and the tools used.
- An experiment is a necessary part of a project to develop all non-trivial software systems.

- Design and programming are iterative processes.
- The different stages of a software project, such as design, programming and testing, cannot be strictly separated.
- Design and programming cannot be considered in isolation from the management of these activities.

It is very easy to underestimate any of these principles, but usually costly. At the same time, it is difficult to translate these abstract ideas into practice. Some experience is required here. Like building a boat, riding a bicycle, or programming, design is an art that cannot be mastered through theory alone.

Maybe all these capacious principles can be compressed into one: design and programming are types of human activity; forget about it - and everything is gone. Too often we forget about this and consider the software development process simply as "a sequence of well- defined steps, at each of which, according to given rules , some actions are performed on the input data in order to obtain the desired result." The very style of the previous sentence betrays the presence of human nature! This chapter is about projects that can be considered ambitious given the resources and expertise of the people building the system. It seems that it is in the nature of both the individual and the organization to take on projects at the limit of their capabilities. If the task does not contain a specific challenge, there is no point in paying special attention to its design. Such tasks are solved within the framework of an already established structure that should not be destroyed. Only if they are aiming at something ambitious, there is a need for new, more powerful tools and techniques. In addition, there is a tendency for those who "know how to do it" to outsource the project to beginners who do not have such knowledge.

There is no "one right way" to design and build an entire system. I would consider the belief in "the only correct way" to be a childhood disease, if experienced programmers did not get sick too often with this disease . Recall again: just because a technique has been successfully used for a year for one project, it does not follow that it will be equally useful for another person or another task without any changes . It is always important not to be prejudiced.

The belief that there is no single right solution permeates the entire design of C ++, and, in general, for this reason, the first edition of the book did not have a section on design: I did not want to be seen as a "manifesto" of my personal sympathy. For the same reason, here, as in Chapters 12 and 13, there is no clearly defined view of the process of software development, rather, it simply discusses a certain range of frequently asked questions and offers some solutions that turned out to be useful in certain conditions.

This introduction is followed by a brief discussion of the goals and means of software development in $$ 11.2, and then the chapter breaks down into two main parts:

- $$ 11.3 contains a description of the software development process.
- $$ 11.4 contains some practical recommendations for organizing this process.

The relationship between design and a programming language is discussed in Chapter 12, and Chapter 13 is devoted to the design of C ++ libraries.

Obviously, most of the reasoning relates to large software projects . Readers who do not participate in such developments can sit quietly and rejoice that all these horrors have passed them, or they can choose questions that concern only their interests. There is no lower bound on the size of a program from which it makes sense to start designing before starting to write a program. However, there is still a lower limit from which any design methods can be used . Size issues are discussed in $$ 11.4.2.

The hardest part of software projects is to deal with complexity. There is only one general way to deal with complexity: divide and conquer. If the problem can be divided into two subtasks that can be solved separately, then we can consider it solved by dividing more than half. This simple principle applies to a surprisingly large number of situations. In particular, the use of modules or classes in the development of software systems allows you to split the program into two parts: the implementation part and the user part - which are interconnected (ideally) by a well-defined interface. This is a basic, intrinsic programming principle of dealing with complexity. Similarly, the software design process can be broken down into distinct activities with well-defined (ideally) interactions between the people involved. It is a fundamental, inherent design principle of dealing with complexity and an approach to managing the people involved in a project.

In both cases, isolating the parts and defining the interface between the parts is where the maximum experience and flair is required. This selection is not a purely mechanical process, it usually requires discernment, which can only come from a thorough understanding of the system at various levels of abstraction (see $$ 11.3.3, $$ 12.2.1 and $$ 13.3). A shortsighted look at a program or software development process often leads to a defective system. Note that it is easy to separate both programs and programmers . It is more difficult to achieve effective interaction between participants on both sides of the border without breaking it or making the interaction too rigid.

It proposes a specific design approach rather than a complete formal description of the design method. Such a description is outside the subject area of the book. The approach proposed here can be applied with varying degrees of formalization and can serve as the basis for different formal specifications. At the same time, this chapter cannot be considered an abstract, and it does not attempt to address every topic related to the software development process or present every point of view. This also goes beyond the subject area of the book. An abstract on this topic can be found in [2]. This book uses fairly general and traditional terminology. The most "interesting" terms, such as: design, prototype, programmer - have several definitions in the literature that often contradict each other, so we warn you against the fact that, based on the definitions of terms accepted in your environment, you do not take from the book what that the author did not expect at all.

11.2 Objectives and means

The goal of programming is to create a product that satisfies the user. The most important means to achieve this goal is to create a program with a clear internal structure and educate a team of programmers and developers who have sufficient experience and motivation to quickly and effectively respond to all changes.

Why is this so? After all, the internal structure of the program and the process by which it is obtained, ideally, do not affect the end user in any way . Moreover, if the end user is for some reason interested in how the program is written, then something is wrong with this program . Why, in spite of this, are the structure of the program and the people who created it so important ? After all, the end user doesn't have to know anything about this .

The clear internal structure of the program makes it easy to:

- testing,
- portability,
- accompaniment,
- extension,
- reorganization and
- understanding.

The main thing here is that any successful large program has a long life, during which generations of programmers and developers work on it , it is transferred to a new machine, it adapts to unforeseen requirements, and is rebuilt several times. During the entire lifetime, it is necessary to issue versions of the program at an acceptable time and with an acceptable number of errors. Not planning all this is like planning failure.

Note that while users ideally should not know the internal structure of the system, in practice they usually want to know it. For example, a user may want to become familiar with the details of system design in order to learn how to control the capabilities and reliability of the system in the event of rework and expansion. If the software product in question is not a complete system, but a set of libraries for obtaining software systems, then the user will want to know more "details" so that they serve as a source of ideas and help to better use the library.

You need to be able to very accurately determine the scope of program design. Insufficient volume leads to endless cutting of sharp corners ("we will quickly transfer the system, and the error will be eliminated in the next version"). Excessive volume leads to a complicated description of the system, in which the essential is lost in the formalities, as a result of which, when the program is reorganized, getting a working version is delayed ("the new structure is much better than the old one, the user agrees to wait for it"). In addition, resource needs arise that are beyond the reach of most potential users. Choosing the scope of design is the most difficult moment in development, this is where talent and experience manifests itself. The choice is difficult for one programmer or developer, but it is even more difficult for large tasks, where many people of different skill levels are employed .

The organization must create and maintain the software product despite changes in staffing, direction of work, or management structure. A common way of solving these problems was to try to reduce the process of creating a system to a few relatively simple tasks that fit into a rigid structure. For example, create a group of easy-to-learn (cheap) and interchangeable low-level programmers ("coders") and a group of not so cheap, but interchangeable (and therefore also not unique) developers. It is believed that coders do not make design decisions , and developers do not bother with the "dirty" coding details. Typically, this approach fails , and where it works, the system is overly cumbersome with poor performance.

The disadvantages of this approach are as follows:

- weak interaction between programmers and developers leads to inefficiency, procrastination, missed opportunities and repetition of mistakes due to poor accounting and lack of exchange of experience;

- the narrowing of the area of creativity of developers leads to poor professional growth, lack of initiative, negligence and high staff turnover.

In fact, such systems are a waste of rare human talent. Creating a structure within which people can find application for different talents, master new occupations and participate in creative work is not only a noble cause, but also a practical, commercially profitable enterprise.

On the other hand, you cannot create a system, provide documentation for it and maintain it indefinitely without some rigid organizational structure. For a purely innovative project, a good place to start is simply to find the best people and let them solve the problem according to their ideas. But as the project develops, more planning, specialization and strictly defined interaction between the people involved in it are required . A strictly defined notation is not meant to be a mathematical or automatically verified record (although this is certainly good where possible and applicable), but rather a set of guidelines for recording, naming, documentation, testing, etc. But even here a sense of proportion is needed. A structure that is too rigid can hinder growth and make it difficult to improve. This is where the manager's talent and experience are put to the test. For the individual worker, a similar problem comes down to determining where to be smart and where to follow the recipe.

You can recommend planning not for the period until the next version of the system is issued , but for a longer period. To make plans only before the next version is released is to plan for failure. You need to have a software organization and strategy that aims to create and maintain many versions of different systems, i.e. you need multiple planning for success.

The goal of design is to develop a clear and relatively simple internal structure of a program, sometimes called architecture, in other words, a framework in which individual program fragments fit and which helps to write these fragments.

A project is the end result of the design process (if only there is an end product for an iterative process). It is the focal point for interactions between developer and programmer and between programmers. A sense of proportion must be observed here. If I, as an individual programmer, design a small program that I am going to write tomorrow, then the accuracy and completeness of the description of the project can be reduced to a few scribbles on the back of the envelope. At the other extreme is a system that hundreds of programmers and developers are working on, and this may require volumes of carefully written project specifications in a formal or semi-formal language. Determining the required degree of accuracy, detail and design formality is in itself a non-trivial technical and administrative task.

In what follows, it will be assumed that the design of the system is written as a series of class definitions (in which particular descriptions are omitted as unnecessary details) and the relationships between them. This is a simplification because a particular project can take into account: concurrency issues, the use of a global namespace, the use of global functions and data, building a program to minimize re-translation, robustness, multi-machine mode, etc. But when discussed at this level of detail, simplification is essential , and classes in the context of C ++ are a key design concept. Some of these issues will be discussed below, and those that directly affect the design of C ++ libraries will be discussed in Chapter 13. For a more detailed discussion and examples of certain object-oriented design techniques , see [2].

We have deliberately not made a clear distinction between analysis and design, since a discussion of their differences is outside the scope of this book, and it depends on the design methods used. The key is to choose an analysis method that suits your design method and choose a design method that suits your programming style and language.

11.3 Development process

The software development process is an iterative and incremental process. As development progresses, each stage is repeated many times, and with each return to a certain stage of the process, the final product obtained at this stage is specified . In the general case, the process has no beginning or end, because when designing and implementing the system, you start using other projects, libraries and application systems as a base , at the end of the work, you are left with a description of the project and a program that others can refine, modify. , expand and transfer. Naturally, a specific project has a definite beginning and end, and it is important (although often surprisingly difficult) to clearly and strictly limit the time and scope of the project. But claiming that you are starting with a blank slate can lead to serious problems for you, as well as the position that once the final version is handed over - even a deluge - will cause serious problems for your followers (or for you in your new role).

It follows that the following sections can be read in any order, since design and implementation issues can be intertwined almost arbitrarily in a real project. Namely, the "project" almost always undergoes redesign based on the previous project, certain implementation experience, time constraints , skill of workers, compatibility issues, etc. Here the main challenge for a manager or developer or programmer is to create an order in this process that does not hinder improvements and does not prohibit the repeated passes necessary for successful development.

The development process has three stages:

- Analysis: defining the scope of the problem.
- Design: creation of the overall structure of the system.
- Implementation: programming and testing.

Do not forget about the iterative nature of these processes (it is not for nothing that the stages were not numbered), and note that no important aspects of the program development process are distinguished into separate stages, since they must allow:

- Experimentation.
- Testing.
- Design and implementation analysis.

- Documentation.
- Escort.

Software maintenance is viewed simply as a few more passes through the stages of the development process (see also $$ 11.3.6).

It is very important that analysis, design and implementation are not too disconnected from each other, and that the people involved are of the same skill level to establish effective contacts.

It is all too often different in large projects. Ideally, in the process of project development, workers themselves should move from one stage to another: the best way to convey subtle information is to use the worker's head. Unfortunately, organizations often place barriers to such transitions, for example, a developer may have a higher status and / or a higher salary than a “simple” programmer. It is not customary for employees to go to departments in order to gain experience and knowledge, but at least let there be regular interviews of employees employed at different stages of the project. For medium and small projects, there is usually no distinction between analysis and design - these stages merge into one. For small projects, design and programming are also not separated. Of course, this solves the problem of interaction. It is important for this project to find the appropriate degree of formalization and maintain the required degree of separation between stages ($$ 11.4.2). There is no single right way to do this.

The software development process model presented here is radically different from the traditional waterfall model. In the latter, the development process proceeds linearly from the analysis stage to the testing stage. The main disadvantage of the cascade model is that information moves in only one direction. If a downstream problem is identified , there is strong methodological and organizational pressure to address the problem at this level without affecting the previous stages of the process. Failure to repeat passes results in a defective design, and local troubleshooting results in a flawed implementation. In those inevitable cases when information must be transferred back to the source of its receipt and cause changes in the project, we will get only a weak "wobble" at all levels of the system, seeking to suppress the change, which means that the system is poorly adapted to changes. The argument for "no change" or "only local change" often boils down to the fact that one department does not want to shift a lot of work to another department "for their own good." It often happens that by the time a bug is found, there has been so much paper on the erroneous decision that the effort required to correct the documentation overshadows the effort to fix the program itself. Thus, paperwork can become a major problem in the system creation process. Of course, such problems can and do arise in the development of large systems. After all, some paperwork is necessary. But the choice of a linear development model (cascade) greatly increases the likelihood that this problem will get out of control. The disadvantage of the cascade model is the lack of repeated passes and the inability to respond to changes. The danger of the iterative model proposed here is the temptation to replace thinking and real development with a sequence of endless changes. It is easier to point out both shortcomings than to eliminate, and it is easy for the organizer of the work to mistake simple activity for real progress.

You can pay close attention to detail, use smart management techniques, advanced technology, but nothing will save you if you don't have a clear understanding of what you are trying to create. Most projects have failed because of the lack of well- formulated realistic goals, and not for any other reason. Whatever you do and whatever you do, you need to clearly understand the means you have, set achievable goals and benchmarks, and not look for technical solutions to sociological problems. On the other hand, only adequate technology should be used, even if it requires costs - people work better with adequate funds and an acceptable environment. Don't be fooled into thinking these guidelines are easy to follow.

11.3.1 Development cycle

The system development process is an iterative activity. The main loop is reduced to steps repeated in the following sequence:

[1] Create a general description of the project.

[2] Select standard components.

  [a] Adjust the components for the given project.

[3] Create new standard components.

  [a] Adjust the components for the given project.

[4] Prepare an updated description of the project.

Consider a car factory as an example. The project should start with the most general description of the new car. This first step is based on some analysis and description of the machine in the most general terms that relate to the intended use rather than the characteristics of the desired capabilities of the machine. Often the most difficult part of a project is choosing the desired features, or more precisely, defining a relatively simple criterion for selecting the desired features. Luck here is usually the result of the work of an individual discerning person and is often called foresight. The lack of clear goals is all too typical , which leads to uncertainly developing or simply failing projects.

So, let's say you want to create a medium-sized car with four doors and a fairly powerful motor. Obviously, in the first phase of a project, you should not start designing a machine (and all of its components) from scratch. Although a programmer or software developer would do just that in such circumstances .

The first step is to find out which components are available in your own warehouse and which can be obtained from reliable suppliers. Components found in this way do not necessarily fit the new machine exactly. Adjustment of the components is always required . It may even be necessary to change the characteristics of the "next version" of the selected components to make them suitable for the project. For example, there may be a perfectly acceptable motor that produces slightly less power, then either you or the motor supplier should offer, without changing the general description of the project, an additional charging generator as compensation . Note that doing this "without changing the general description of the project" is unlikely, unless the description itself is adapted to some fit. This usually requires cooperation between you and the motor supplier. Similar questions arise for a programmer or software developer. Here, fitting usually makes it easier to use derived classes effectively . But do not expect to carry out arbitrary extensions in a project without some foresight or cooperation with the creator of such classes.

When the set of suitable standard components has been exhausted, the machine designer is in no hurry to design new optimal components for his machine. It would be too wasteful. Let's say there is no suitable air conditioning unit, but there is an L - shaped free space in the engine compartment. A solution is possible to develop a conditioning unit of the specified form. But the likelihood that a block of this strange shape will be used in a different type of machine (even after significant adjustment) is extremely low. This means that our machine designer will not be able to share the costs of producing such a block with the creators of other types of machines, which means that the lifetime of this block is short. Therefore, it is worth designing a block that will find wider application, i.e. develop a sensible block design more fit for fit than our L-shaped beast. This may require a lot of effort, and even the general description of the machine design will have to be changed to accommodate a more universal unit . Since the new block was designed for a more general purpose than our L-shaped beast, presumably it will require some tweaking to fully meet our revised requests. A similar alternative arises for a programmer or software developer: instead of creating a program that is tied to a specific project, the developer can design a new sufficiently universal program that has a good chance of becoming standard in a certain area.

Finally, when we have gone through all the standard components, a "final" general project description is drawn up. Several specially designed tools are listed as possible. Probably next year we will have to repeat our steps for the new model, and it is precisely these special tools that will have to be redone or thrown away. Sadly, the experience of traditionally designed programs shows that only a few parts of the system can be separated into separate components, and only a few of them are suitable outside the given project.

We are not trying to argue that all machine designers act as intelligently as in the above example, and that software developers make all these mistakes. It is argued that the specified machine development methodology is applicable to software. So, in this and the next chapters, techniques for using it for C ++ are given. Nevertheless, we can say that the very nature of programming is conducive to committing these errors ($$ 12.2.1 and $$ 12.2.5). Section 11.4.3 refutes professional bias against using the design model described here. Note that the software development model is well applicable only in the long term. If your horizon is narrowing to the time of the next release, there is no point in creating and maintaining the functioning of standard components. This will simply lead to unnecessary overhead. Our model is designed for organizations with a lifetime, for which several projects take place , and with dimensions that allow for additional costs for design tools, programming, and project support, and for advanced training of developers, programmers and managers. In fact, this is a sketch of some factory for the production of programs. Surprisingly, it only differs in scale from the actions of the best programmers, who over the years have accumulated a stock of design techniques and methods , created tools and libraries to increase their productivity . It seems that most organizations simply do not know how to take advantage of the achievements of the best employees, both due to a lack of foresight and an inability to apply these achievements in a sufficiently wide range .

It is still unreasonable to require "standard components" to be universally standard. There are few international standard libraries, and most of the components will only be standard within a country, industry, company, production chain, department or application area, etc. It's just that the world is too big for a universal standard of all components and tools to be a real or desired goal of a project.

11.3.2 Design objectives

What are the most general design goals? Of course, simplicity, but what is the criterion for simplicity? Since we believe that the project should develop over time, i.e. the system will expand, move, customize and, in general, change in a variety of ways that cannot be foreseen, it is necessary to strive for a design and implementation system that would be simple, taking into account that it will change in many ways. In fact, it is practical to assume that the system requirements themselves will change repeatedly over the period from the initial project to the release of the first version of the system.

The conclusion is that the system should be designed as simple as possible , provided that it will undergo a series of changes. We must design for change, i.e. aspire to

- flexibility,
- extensibility and
- portability

The best solution is to separate the parts of the system that are most likely to change into independent units, and give the programmer or developer the flexibility to modify such units. This can be done by highlighting the key concepts for a given task and providing a class responsible for all the information related to a particular concept (and only to it). Then the change will only affect a specific class. Naturally, this ideal way is much easier to describe than to implement.

Consider an example: in the task of modeling meteorological objects, you need to represent a rain cloud. How to do it? We do not have a general method for depicting a cloud, since its appearance depends on the internal state of the cloud, and it can only be specified by the cloud itself.

First solution: let the cloud represent itself. It is suitable for many limited applications. But it is not general enough , since there are many ways to represent a cloud: a detailed picture, a sketch of an outline, a pictogram, a map, etc. In other words, the type of cloud is determined both by itself and by its environment.

The second solution is to provide the cloud itself with information about its environment for its image. It is suitable for more cases. However, this is not a general solution either. If we provide the cloud with information about its environment, then we violate the basic postulate, which requires a class to be responsible for only one concept, and each concept is embodied by a specific class. It may not be possible to come up with a consistent definition of "cloud environment" because, generally speaking, what a cloud looks like depends on the cloud itself and the observer. How the cloud appears to me depends a lot on how I look at it: with the naked eye, with the help of a polarizing filter, with the help of a weather radar, etc. In addition to the observer and the cloud, the "general background" should also be considered, for example the relative position of the sun. The addition of new objects such as other clouds and airplanes leads to a further complication of the picture . To make the developer's task virtually impossible , you can add the ability to coexist multiple observers.

The third solution is for the cloud, as well as other objects, such as airplanes or the sun, to describe themselves in relation to the observer. This approach is general enough to satisfy most requests. However, it can lead to significant complexity and large overhead in execution. How, for example, can you make the observer understand the descriptions produced by the cloud or other objects?

Even this model is likely to be insufficient for such limiting cases as high-resolution graphics. I think a different level of abstraction is needed to get a very detailed picture.

Rain clouds are not an object that you often see in programs, but objects that participate in various input and output operations are common. Therefore, we can consider the example with the cloud suitable for programming in general and for developing libraries in particular. A logically similar example in C ++ is provided by manipulators, which are used to format the output in stream I / O ($$ 10.4.2). Note that the third solution is not the "right solution", it is just a more general solution. The designer must balance the various requirements of the system to find a level of generality and abstraction suitable for a given task in a given area. Rule of thumb: For a program with a long lifespan, the most general level of abstraction that you can understand and can afford is right, but not necessarily completely general. A generalization that goes beyond the scope of this project and the concept of the people involved in it can be harmful, i.e. lead to delays, unacceptable performance, unmanageable projects and simple failure.

For these methods to be economical and manageable, design and management must be reusable, as discussed in $$ 11.4.1, and efficiency should not be forgotten altogether (see $$ 11.3.7).

11.3.3 Design steps

Consider designing a separate class. This is usually not the best method. Concepts do not exist in isolation, on the contrary, a concept is defined in relation to other concepts. Similarly, a class does not exist in isolation, but is defined together with a set of related classes. This set is often referred to as a class library or component. Sometimes all component classes form a single hierarchy, sometimes they are not (see $$ 12.3).

Many component classes are united by some logical condition, sometimes it is a general style of programming or description, sometimes it is a provided service. A component is a unit of design, documentation, ownership, and often reuse. This does not mean that if you use one component class, then you must understand all and be able to apply all component classes, or you must load modules of all component classes to your program. The exact opposite is usually strive to ensure that the use of the class led to minimizing overhead costs: how computer resources, and human effort. But to use any component class, you need to understand the logical condition that defines it (hopefully, it is very clearly stated in the documentation), understand the conventions and style applied in the process of designing and describing the component, and the available service (if any).

So, let's move on to how to design a component. Since this is often not an easy task, it makes sense to break it down into steps and focus on the subtasks to provide a complete and consistent description. There is usually no single correct way to split. However, the following is a sequence of steps that came in handy in several cases:

1] Define the concept / class and establish the main connections between them.

2] Clarify class definitions by specifying a set of operations for each.

[a] Classify transactions. In particular, clarify the need for building, copying and destroying.

[b] Make sure that it is minimal, complete and comfortable.

3] Clarify the definitions of classes by indicating their dependence on other classes.

[a] Inheritance.

[b] Using dependencies.

4] Define class interfaces.

[a] Divide functions into general and protected.

[b] Determine the exact type of class operations.

Note that these are steps in an iterative process. Usually, to get a project that you can confidently use for initial implementation or re-implementation, you need to go through a series of steps several times . One of the benefits of deep analysis and the data abstraction suggested here is the relative ease with which you can rebuild class relationships even after programming each class. It's never easy, though .

The next step is to start implementing the classes, and then return to evaluate the project based on the implementation experience. Let's take a look at these steps separately.

11.3.3.1 Step 1: define classes

Define concepts / classes and establish basic connections between them. The main thing in a good project is to directly reflect any concept of "reality", ie. grasp a concept from the scope of a class application, represent the relationship between classes in a strictly defined way, for example, using inheritance, and repeat these actions at different levels of abstraction. But how can we grasp these concepts? How do we decide in practice which classes we need?

It is better to look for the answer in the application domain itself than to rummage through the programmer's repository of abstractions and concepts. Reach out to someone who has become an expert at working in a once-made system, as well as someone who has become a critic of the system that replaced it. Memorize the expressions of both.

It is often said that nouns play the role of classes and objects used in a program, this is true. But this is just the beginning. Further, verbs can represent operations on objects or ordinary (global) functions that generate new values based on their parameters, or even classes. The functional objects described in $$ 10.4.2 can be considered as an example . Verbs such as "to repeat" or "commit" can be represented by an iterative object or an object representing the operation of program execution in databases. Even adjectives can be successfully represented using classes, such as "stored", "parallel", "register", "limited". These can be classes that help a developer or programmer, by defining virtual base classes, to specify and select the desired properties for classes that are designed later.

The best way to find these concepts / classes is a slate board, and the best method of first clarification is by talking to application experts or just friends. Discussion is needed to create an initial viable vocabulary and conceptual framework. Few can do it alone. Refer to [1] for methods of such refinements.

Not all classes conform to application scope concepts. Some may represent system resources or implementation period abstractions (see $$ 12.2.1).

The relationships we are talking about naturally arise in the application domain or (in the case of repeated iterations of the design steps) arise from the subsequent work on the structure of the classes. They reflect our understanding of the basics of the application area. They are often a classification of basic concepts. An example of this attitude: a car with a retractable ladder is a truck, there is a fire engine, there is a moving vehicle.

$$ 11.3.3.2 and $$ 11.3.3.5 offer some perspective on classes and the class hierarchy if you need to improve their structure.

11.3.3.2 Step 2: Define the set of operations

Refine your class definitions by specifying a set of operations for each. In reality, you cannot separate the processes of defining classes and figuring out what operations they need. However, in practice they differ, because when defining classes, attention is focused on the basic concepts, not dwelling on the programming issues of their implementation, while when defining operations, it primarily focuses on defining a complete and convenient set of operations. It is often too difficult to combine both approaches, especially since related classes must be designed at the same time.

There are several possible approaches to the process of defining a set of operations.

We suggest the following strategy:

1] Consider how the class object will be created, copied (if necessary) and destroyed.

2] Determine the minimum set of operations required for the concept represented by the class.

3] Consider the operations that can be added for ease of recording, and include only a few that are really important.

4] Consider what operations can be considered trivial, ie those for which the class acts as an interface for the derived class implementation.

5] Consider how common naming and functionality can be achieved for all component classes.

Obviously this is a minimalist strategy. It is much easier to add any function that brings tangible benefits and make all operations virtual. But, the more functions, the more likely they will not be used, will impose certain restrictions on the implementation and will complicate the evolution of the system. For example, functions that can directly read and write to an object's state variable from a class are forced to use a single implementation method and significantly reduce the possibility of redesign. Such functions reduce the level of abstraction from a concept to its specific implementation. Plus, adding functionality adds work to the programmer and even the developer when he gets back to designing. It is much easier to include another function in the interface as soon as the need for it is established, than to remove it from there when it has already become familiar.

The reason why we require an explicit decision about the virtuality of a given function, without leaving it to the implementation stage , is that by declaring a function virtual, we will significantly affect the use of its class and the relationship of this class with others. Objects from a class that has at least one virtual function require non-trivial memory allocation when compared with objects from languages such as C or Fortran. A class with at least one virtual function essentially acts as an interface to classes that "can still be defined", and a virtual function assumes dependence on classes that "can still be defined" (see $$ 12.2.3)

Note that the minimalist strategy requires, perhaps, a lot of efforts on the part of the developer.

When defining a set of operations, focus more on what needs to be done rather than how to do it.

It is sometimes useful to classify the operations of a class by how they work with the internal state of objects:

- Basic operations: constructors, destructors, copy operations.
- Selectors: operations that do not change the state of the object.
- Modifiers: operations that change the state of an object.
- Transformation operations, i.e. operations that generate an object of another type, based on the value (state) of the object to which they are applied.
- Repeaters: operations that open access to objects of a class or use a sequence of objects.

This is not a partition into orthogonal groups of operations. For example, a repeater can be designed as a selector or modifier. Highlighting these groups is simply intended to aid in the design process of the class interface. Of course, another classification is also permissible . This classification is especially useful for maintaining consistency between classes within the same component.

C ++ has a construct to help you define selectors and modifiers as a member function with and without a const specification. In addition, there are facilities for explicitly defining constructors, destructors, and conversion functions. The copy operation is implemented using assignment operations and copy constructors.

11.3.3.3 Step 3: specifying dependencies

Refine the definition of the classes by specifying their dependencies on other classes. The different kinds of dependencies are discussed in $$ 12.2. The main relationship with respect to design should be considered the relationship of inheritance and use. Both involve an understanding of what it means for a class to be responsible for a specific property of the system. Being responsible for something does not mean that the class must contain all the information, or that its member functions must perform all the necessary operations themselves. Quite the opposite, each class with a certain level of responsibility organizes the work, delegating it in the form of subtasks to other classes that have a lower level of responsibility. But we must warn you that abuse of this technique leads to inefficient and poorly understood projects, since classes and objects are multiplied to such an extent that instead of real work, only a series of requests are made to execute it. What can be done in a given location should be done.

The need to take into account inheritance and use relationships at design time (and not just during implementation) is a direct consequence of the fact that classes represent certain concepts. It also follows that it is a component (i.e., a set of related classes), and not a separate class, that is a design unit .

11.3.3.4 Step 4: define the interfaces

Define class interfaces. Private functions do not need to be considered at this stage of design Implementation issues that arise during the design phase are best discussed in Step 3 when looking at the various dependencies. Moreover, there is a golden rule: if a class does not allow at least two significantly different implementations, then something is clearly wrong with this class, it is just a disguised implementation, not a representation of an abstract concept. In many cases, to answer the question: "Is the interface of a class implementation independent enough?" - it is necessary to indicate whether a lazy evaluation scheme is possible for the class.

Note that common base classes and friends are part of the class's common interface (see $$ 5.4.1 and $$ 12.4). A useful exercise can be to define a separate interface for derived classes and all other classes by splitting the interface into public and private parts.

It is at this step that the precise definitions of the argument types should be thought through and described . Ideally, it is desirable to have the maximum number of interfaces with static types that are application-specific (see $$ 12.1.3 and $$ 12.4).

When defining interfaces, you should pay attention to those classes where the set of operations is presented at more than one level of abstraction. For example, in the file class, some member functions have arguments of type file_descriptor, and others have a character string that denotes the file name. The file_descriptor operations operate on a different (less) abstraction level than the file name operations, so it is even strange that they belong to the same class. It might be better to have two classes, one representing the concept of a file descriptor and the other representing the concept of a file name. Typically, all operations in a class should represent concepts at the same level of abstraction. If this is not the case, then it is worth thinking about reorganizing both it and its associated classes.

11.3.3.5 Rebuilding the class hierarchy

Steps 1 and 3 require researching the classes and their hierarchy to make sure they adequately meet our requirements. This is usually not the case and needs to be rebuilt to improve structure, design, or implementation.

The most typical restructuring of a class hierarchy consists of separating the common part of two classes into a new class or splitting a class into two new ones. In both cases, the result is three classes: a base class and two derivatives. When should such a restructuring take place? What are the general indications that such a restructuring would be beneficial?

Unfortunately, there is no simple and universal answer to these questions. This is not surprising, since what is proposed is not a trifle in implementation, but changes the basic concepts of the system. An important and non-trivial task is to find commonality among the classes and highlight the common part. There is no precise definition of generality, but attention should be paid to generality for system concepts, not just for ease of implementation. The indications that two classes have something in common that can be separated into a common base class are similar uses, similarity of sets of operations, similarity of implementations, and simply the fact that often during the discussion of a project, both classes appear at the same time. On the other hand, if there are multiple sets of operations for a class with different uses, if those sets provide access to separate subsets of implementation objects , and if the class arises from discussion of unrelated topics, then that class is a clear candidate for chunking.

Due to the close relationship between concepts and classes, the problems of rebuilding the hierarchy of classes are highlighted on the surface of the problems of naming classes and the use of class names during the discussion of the project. If the class names and their ordering as defined by the class hierarchy seem awkward when discussing a project, then there is likely room for improvement in the hierarchy. Note that the implication is that the analysis of the class hierarchy is not best done alone. If you find yourself in a position where there is no one to discuss the project with, it is a good idea to try to write a training description of the system using class names.

11.3.3.6 Using models

When you write an article, you try to find a model that fits the topic. You need not rush to print the text right away, but look for articles on similar topics, suddenly there is one that can serve as a starting point. If it turns out to be my own article, then it will be possible to use even parts of it, changing other parts as necessary, and introduce new information only where the logic of the subject requires. Thus , based on the first edition, this book is written. An extreme case of this approach is writing a form postcard, where you just need to provide a name and perhaps add a couple of lines to give a "personal" attitude. In fact, such postcards are written with an indication of the difference from the standard.

In all creative endeavors, using existing systems as models for new projects is the rule rather than the exception. Whenever possible, design and programming should build on previous work. This reduces the degrees of freedom for the developer and allows you to focus on fewer questions at a given time. Starting a big project "practically from scratch" can be exciting, but it would be better to use the term "drunkenness", which will lead to "drunken wandering" in many variations. Building a model does not impose any restrictions and does not mean obediently following it, it just frees the developer from some questions.

Note that in fact, the use of models is inevitable, since each project is synthesized from the experience of its developers. Better when using the model is an explicit decision, then all assumptions are made explicitly, a common vocabulary of terms is defined , an initial project skeleton appears, and it is more likely that developers have a common approach.

Naturally, the choice of the initial model is an important decision, and it is usually only made after searching for potential models and carefully evaluating the options. Moreover, in many cases, a model is appropriate only if it is understood that significant changes will be required to translate its ideas into another application area. But designing software is hard work and any help should be used. You should not abandon the use of models because of undue disregard for imitation. Imitation is nothing more than a form of genuine admiration, and with ownership and copyright in mind, using models and prior artwork as a source of inspiration is an acceptable way for all innovative work in all activities. What Shakespeare was allowed to do works for us too. Some refer to the use of models in the design process as “reusable design”.

11.3.4 Experiment and Analysis

At the start of an ambitious project, we don't know the best way to build a system. It often happens that we do not even know exactly what the system should do, since specific facts will become clear only in the process of building, testing and operating the system. How long before the creation of a complete system can be obtained the information necessary to understand what design decisions will be significant, and what consequences they will lead to?

You need to experiment. Of course, an analysis of the project and its implementation is needed as soon as there is food for it. Much of the discussion revolves around design and implementation alternatives . Except in rare cases, design is a social activity that leads along the path of presentation and discussion. Often the most important design tool is a simple slate; without it, project ideas that are in embryo cannot develop and become common property among developers and programmers.

It seems that the most popular way to conduct an experiment is to build a prototype, i.e. reduced version of the system. The prototype does not have to satisfy the characteristics of real systems, there is usually an abundance of machine resources and software support, and in such conditions programmers and developers become unusually experienced, well educated and active. The goal appears - to make a working prototype as soon as possible in order to start exploring project options and ways of implementation.

This approach, if applied wisely, can lead to success. But it can also serve as an excuse for failing systems. The point is that by paying special attention to the prototype, you can come to a shift of effort from "exploring project options " to "getting a working version of the system as soon as possible. Then interest in the internal structure of the prototype will quickly fade away ("after all, this is only a prototype"), and design work will be supplanted by manipulation of the prototype implementation. The miscalculation is that such an implementation can easily lead to a system that looks like "almost complete", but in fact is a resource eater and a nightmare for those who maintain it. In this case, time and energy is wasted on the prototype, which is better reserved for the real system. There is a temptation for developers and managers to remake a prototype into a final software product, and postpone the "art of system tuning" until the next version is released. If you go this way, then prototypes deny all design fundamentals.

A similar problem arises when researchers become attached to the tools they created when building a prototype, and forget that they may not be suitable for a working system, and that the freedom from the restrictions and formalities to which they are used to working in a small group can be impossible in a large team struggling to remove a long chain of obstacles.

And at the same time, prototyping can play an important role. Consider, for example, designing a user interface. For this task, the internal structure of that part of the system that does not directly communicate with the user is usually not important, and using prototypes is the only way to find out what the user's reaction will be when working with the system. Another example is prototypes that are specifically designed to study the internal structure of a system. Here, the interface with the user can be primitive, it is possible to work with the user model.

Using prototypes is a way of experimenting. The expected result is a deeper understanding of the goals, not the prototype itself. Perhaps the essence of the prototype is that it is so incomplete that it can only serve as a tool for experiment, and it cannot be turned into a final product without the large expense of redesign and other implementation. By leaving the prototype "incomplete", we thereby shift attention to the experiment and reduce the danger of turning the prototype into a finished product. It also almost eliminates the temptation to take the prototype design as the basis for the system design, while forgetting or ignoring the constraints that are inherent in the prototype. After the experiment, the prototype should simply be thrown away.

Do not forget about other experimental methods, which can serve as alternatives to prototyping in many cases, and where applicable, they are preferred because they are more accurate and require less developer time and system resources. Examples include mathematical models and various forms of modeling. In fact, there is an endless increasing sequence, starting from mathematical models, to more and more detailed methods of modeling, then to prototypes, to partial realizations of the system, right up to the complete system.

This leads to the idea of building a system from the initial design and implementation, and moving through the design and implementation stages over and over. This is an ideal strategy, but it places high demands on the design and implementation tools , and it carries a certain risk that the software volume that implements the decisions made during the initial design will grow to such a size during the development process that it is simply impossible to significantly improve the project. ...

It seems that at least now this strategy is being used in small to medium-sized projects, i.e. where alterations of the general project are unlikely, or for redesign and other implementation after the release of the initial version of the system, where the specified strategy becomes inevitable.

In addition to experiments designed to evaluate decisions made at the design stage, an analysis of the design and / or implementation itself can be a source of useful information . For example, it can be useful to study various dependencies between classes (see $$ 12.2), and do not forget about such traditional implementation aids as function call graph, performance evaluation, etc.

Note that the specification (result of the system analysis) and the design may contain errors, as well as the implementation, and perhaps they are even more error-prone, since are less accurate, cannot be tested in practice, and are usually not surrounded by sophisticated tools such as those used to analyze and validate the implementation. Introducing more formalization into the language or notation with which the design is presented makes it somewhat easier to use these tools for design. But, as $$ 12.1.1 says, this cannot be done at the cost of degrading the language used for the implementation. Moreover , the formal entry can itself become a source of difficulties and problems. This happens when the chosen degree of formalization is poorly suited for specific tasks, when the rigor of formalization exceeds the mathematical basis of the system and the qualifications of developers and programmers, and when the formal description of the system begins to diverge from the real system for which it was intended.

The conclusion that experience is needed, and that design is inevitably buggy and poorly supported by software , is the main argument for an iterative design and implementation model . The alternative is a linear model of the development process, starting with analysis and ending with testing, but it is significantly defective, since it does not allow repeated passes, based on the experience gained at various stages of the development of the system.

11.3.5 Testing

A program that has not been tested does not work. The ideal that, after design and (or) verification, the program works the first time, is unattainable for everyone, except for the most trivial programs. You should strive for the ideal, but do not be mistaken that testing is a simple matter.

"How to test?" - this question cannot be answered in the general case. However, the question "When to start testing?" has such an answer - at the earliest stage where possible. The testing strategy should be developed as part of the project and included in the implementation, or at least developed in parallel with them. As soon as a working system appears, testing should begin . Delaying testing until "full implementation" is done is a surefire way to get out of schedule or submit a buggy version .

Wherever possible, design should be done so that testing the system is straightforward. In particular, it makes sense to build testing tools directly into the system. This is sometimes not done because of fear of too much run -time validation, or fear that the redundancy required for full testing would unnecessarily complicate the data structures. Usually, such fears are unjustified, since the actual verification programs and additional constructions required for them can, if necessary, be removed from the system before it is delivered to the user. Sometimes statements about program properties can come in handy (see $$ 12.2.7).

More important than a set of tests is the approach when the structure of the system is such that there is a real chance to convince yourself and users that errors can be eliminated using a certain set of static checks, static analysis and testing. If a strategy for building a fault-tolerant system has been developed (see $$ 9.8), the testing strategy is usually designed as an auxiliary one.

If testing issues are completely ignored during the design phase , there will be problems with testing, delivery times and system maintenance. The best place to start working on your testing strategy is with the class interfaces and their interdependencies (as suggested in $$ 12.2 and $$ 12.4).

It is difficult to determine the amount of testing required. However, it is clear that the problem is lack of testing, not too much testing . How much resources should be allocated for testing versus design and implementation depends on the nature of the system and how it is built. However, we can suggest the following rule: devote more resources, time and human effort to testing the system than getting its first implementation.

11.3.6 Escort

Software maintenance is an unfortunate term. The word "accompaniment" offers a false analogy to hardware. Programs do not require lubrication, have no moving parts that wear out and need to be replaced, and they have no cracks that water gets into , causing rust. Programs can be played accurately and transmitted over long distances within a minute. In short, software is not at all the same as hardware. (Original: "Software is not hardware").

The activity that we refer to as software maintenance actually consists of redesign and re-implementation and is therefore part of the normal software development cycle. If the project takes into account the issues of extensibility, flexibility and portability, then the usual maintenance tasks are solved in a natural way.

Similar to testing, maintenance tasks should not be tackled outside the mainstream of the project and should not be postponed .

11.3.7 Efficiency

D. Knuth owns the statement "Poor optimization is the root of all troubles." Some are all too convinced that this is true and consider all worries about optimization harmful. In fact, efficiency issues need to be kept in mind all the time during design and implementation. This does not mean that the developer should deal with local optimization problems, only the optimization problem at the most global level should worry him.

The best way to be effective is to create a clear and simple design. Only such a project can remain relatively stable for the entire development period and serve as the basis for tuning the system in order to increase productivity. It is important here to avoid the "gargantualism" that is the bane of big projects. Too often people add certain system features "just in case" (see $$ 11.3.3.2 and $$ 11.4.3), doubling, quadrupling the size of the program being executed for the sake of curl. Even worse, such complex systems are difficult to analyze , making it difficult to distinguish excess overhead from necessary overhead and to analyze and optimize at a general level. Optimization should be the result of analyzing and evaluating the performance of the system, rather than arbitrary manipulation of the program code, and this is especially true for large systems where the intuition of the developer or programmer cannot serve as a reliable indicator of efficiency.

It is important to avoid inherently ineffective designs, as well as designs that can be brought to an acceptable level of performance only with a lot of time and effort. For the same reason, it is important to minimize the use of inherently unportable structures and tools, since their presence prevents the system from operating on other machines (less powerful, less expensive).

11.4 Project management

If it only makes sense, most people do what they are encouraged to do. So, in the context of a software project, if a manager encourages certain behaviors and punishes others, rare programmers or developers will risk their position, meeting resistance or indifference from the administration, to do as they see fit.

An organization that thinks their programmers are half-witted will very soon have programmers who are happy and able to act only as half-witted.

It follows that the manager should encourage structures that are consistent with the stated project and implementation objectives. However, in practice it is too often the case. A significant change in programming style is achievable only with a corresponding change in the design style, in addition, usually both require a change in the management style. Thought and organizational inertia too simply reduces everything to local changes, although only global changes can bring success. A perfect illustration is the transition to a language with object-oriented programming, for example, to C ++, when it does not entail appropriate changes in design methods to take advantage of the new features of the language (see $$ 12.1), and, conversely, when the transition to " object-oriented design "is not accompanied by a transition to an implementation language that supports this style.

11.4.1 Reuse

Often the main reason for moving to a new language or new design method is that it makes it easier to reuse programs or projects. However, many organizations reward an employee or group when they choose to reinvent the wheel. For example, if a programmer's productivity is measured by the number of lines of code, will he write small programs that work with the standard libraries at the expense of his income and maybe position? And the manager, if he is paid in proportion to the number of people in his group, will he use programs made by other teams if he can just hire a couple more programmers to his group? A company can get a government contract in which its income is a fixed percentage of the project costs, will it reduce its income by using the most efficient means? It is difficult to provide rewards for reuse, but if the administration does not find ways to encourage and reward, then there simply will not be.

Reuse is primarily a social factor. Reuse of the program is possible provided that

1] she works; cannot be reused if it is not possible for the first time;

2] it is understandable; here the structure of the program, the presence of comments, documentation, manuals matter ;

3] it can work together with programs that were not specially created with such a condition;

4] you can count on her support (or you have to do it yourself, which you usually don't want);

5] it is beneficial (although it is possible to share development and maintenance costs with other users) and finally;

6] it can be found.

Added to this is that a component is not reusable until someone actually does it. Usually the task of adapting a component to the existing environment leads to a more precise set of operations, generalization of its behavior, and an increase in its ability to adapt to other programs. Until all this is done at least once, unexpected sharp corners are found even in components that have been carefully designed and implemented.

Personal experience suggests that conditions for reuse arise only when there is a specific person involved in this issue. In small groups, this is usually someone who happens to be the custodian of shared libraries or documentation by accident or by design. In large organizations, this is a group or department that has the privilege of collecting, documenting, promoting, and maintaining software used by various groups.

Such groups of "standard components" should not be underestimated. We point out that, as a first approximation, the system reflects the organization that created it. If the organization has no means of encouraging and rewarding cooperation and division of labor, then in practice they will be an exception. The standard components group should actively offer their components. Plain traditional documentation is important, but not sufficient. In addition, the specified group should provide guidelines and other information that will allow a potential user to find the component and understand how he can help him. This means that this group should take actions that are usually associated with the education and marketing system. Component team members should, whenever possible, work closely with developers in the application areas. Only then will they be aware of user requests and will be able to sense the possibilities of using the standard component in various fields. This is the rationale for using such a group as a consultant and in favor of internal software supply so that information from the component group can be freely shared.

Note that not all programs need to be reusable; in other words, reuse is not a universal property. To say that a component can be reused means that within a certain structure, its reuse does not require significant effort. But in most cases, moving to another structure can be a lot of work. In this sense, reuse strongly resembles portability. It is important to understand that reuse is the result of design with this goal in mind, modification of components based on experience, and special efforts made to find candidates for reuse among existing components . Unconscious use of language or programming techniques cannot miraculously guarantee reuse. Such means of language C ++ as the classes, virtual functions and the type of pattern to help design, makes it easier to reuse (hence make it more likely), but by themselves these funds do not guarantee re-use.

11.4.2 Size

Individuals and organizations tend to be overly happy that they are "doing the right thing." In an institution, this often sounds like "development according to strict regulations." In both cases, common sense is the first victim of a passionate and often sincere desire to make improvements. Unfortunately, if common sense is lacking, then the damage caused by unreasonable actions can be unlimited.

Let's go back to the development process steps listed in $$ 11.3 and the design steps listed in $$ 11.3.3. It is relatively easy to transform these steps into a precise design method when the step is well- defined, has well-defined inputs and outputs, and a semiformal record for specifying inputs and outputs. You can draw up a protocol to which the design must obey, create tools that provide certain convenience for recording and organizing the process. Further, by examining the classification of dependencies given in $$ 12.2, one can determine that certain dependencies are good and others should be considered bad, and provide analysis tools that will ensure that such assessments are carried out at all stages of the project. To complete such a "standardization" of the process of creating programs, it would be possible to introduce standards for documentation (including rules for spelling and grammar and conventions on the format of documentation), as well as standards for the general appearance of programs (including instructions which language means should use and which not, enumeration of valid libraries and those that do not need to be used, naming conventions for functions, types, variables, rules for the arrangement of program text, etc.).

All of these can contribute to the success of the project. At the very least, it would be sheer foolishness to take on the design of a system that is supposed to have on the order of ten million lines of text, on which hundreds of people will work, and which will be accompanied by thousands of people for decades, without a well-defined and rigorous plan for all of the above items.

Fortunately, most systems do not fall into this category. However, once it is decided that a given design method, or following the specified patterns in programming and documentation, is "correct," pressure begins to be applied to apply it universally. On small projects, this leads to ridiculous restrictions and a lot of overhead. In particular, this can lead to the fact that the measure of development and success is not productive work, but sending papers and filling out various forms. If this happens, then in such a project real programmers and developers will be replaced by bureaucrats.

When there is such a ridiculous abuse of design methods (apparently perfectly reasonable), then the failure of the project becomes an excuse for abandoning almost all formalization of the software development process. This, in turn, leads to the kind of confusion and failure that proper design method should have prevented.

The main problem is to determine the degree of formalization suitable for the development process of a specific project. Don't expect to find a solution easily. Basically, for a small project, every method can work. Worse, just about every method looks like, even if it's poorly thought out and harsh on the performers, can work for a large project if you're willing to put in a lot of time and money.

In the process of software development, the main task is to maintain the integrity of the project. The difficulty of this task depends non-linearly with the size of the project. Only one person or a small group can formulate and save the basic settings in a large project . Most people spend so much time solving subtasks, technical details, day-to-day administrative work, that the general goals of the project are easily forgotten or replaced with more local and close goals. A sure- fire path to failure when there is no person or group with a direct task of overseeing the integrity of the project. A sure way to fail is when such a person or group has no means to influence the project as a whole.

The absence of agreed long-range goals is far more dangerous to the project and organization than the absence of any one particular property. A small group of people should formulate such general goals, constantly keep them in mind, draw up documents containing the most general description of the project, compose explanations for the basic concepts, and in general, help everyone else remember the purpose of the project.

11.4.3 Human Factors

The design method described here is designed for skilled developers and programmers, so the success of the organization depends on their selection .

Managers often forget that an organization is made up of individuals. It is widely believed that programmers are equal and interchangeable. This misconception can destroy the organization by crowding out many of the most active employees and forcing others to work on tasks well below their level. Individuals are interchangeable only if they are not allowed to use their talent, which raises them above the general minimum level necessary to solve a given problem. Therefore, the myth of fungibility is inhuman and inherently wasteful.

Many systems for evaluating programmer performance encourage extravagance and fail to account for significant personal input . The most obvious example is the widespread practice of evaluating success in terms of the number of lines programmed, documentation pages returned, tests missed, etc. Such figures look impressive on the diagrams, but have the most distant relation to reality. For example, if performance is measured by the number of lines programmed, then successful reuse will degrade the programmer's score. Usually, the same effect will have the same effect as applying the best practices in the process of redesigning a large part of the system.

The quality of the result is much more difficult to measure than the quantity, and the reward of the performer or group follows the quality of their work, and not on the basis of rough quantitative estimates. Unfortunately, as far as is known, the practical development of quality assessment methods has not yet begun. In addition, estimates that do not fully describe the state of the project can distort the process of its development. People adjust to meet the deadline and redesign their work according to performance estimates, with the result that overall system integrity and performance suffers . For example, if a deadline is set aside for detecting a certain number of errors, then in order to meet it, they actively use checks at the execution stage , which degrades the performance of the system. Conversely, if only the characteristics of the system at the stage of execution are taken into account, then the number of undetected errors will grow, provided there is a lack of time for the executors. The lack of good and reasonable quality ratings raises the demands on the technical qualifications of managers, otherwise there will be a constant tendency to reward voluntary activity rather than real progress. Remember that managers are people too, and they should be at least as knowledgeable about new technologies as the people they manage.

Here, as in other aspects of the software development process , long time frames should be considered. In fact, it is impossible to quantify a person's performance based on their work per year. However, many employees have a record of their achievements over a long period, and this can serve as a reliable guide for predicting their performance. If such cards are ignored , which is what is done when employees are considered interchangeable spokes in the wheel of the organization, the manager is left with only misleading quantitative estimates.

If we consider only sufficiently long time frames and abandon management methods designed for "interchangeable idiots", then we must admit that an individual (both a developer or programmer, and a manager) needs a long time to grow to more interesting and important work. This approach discourages both "jumping" from place to place, and transferring work to another for career reasons. The goal should be low turnover of key specialists and key managers. No manager will be successful without the right technical knowledge and rapport with core developers and programmers. At the same time, ultimately no group of developers or programmers will succeed without the support of competent managers and without understanding at least basic non-technical issues related to the environment in which they work.

When it is required to offer something new, the main specialists come to the fore - analysts, developers, programmers. It is they who must solve the difficult and critical task of introducing new technology. These are the people who must learn new methods and, in many cases, forget old habits. It’s not easy. After all, these people have made a great personal contribution to the creation of the old methods and their reputation as a specialist is justified by the successes obtained using the old methods. The same is the case with many managers.

Naturally, these people have a fear of change. It can lead to exaggeration of the problems of change and an unwillingness to acknowledge the problems caused by old methods. Naturally, on the other hand, proponents of change may overestimate the benefits that change will bring and underestimate the challenges that arise. These two groups of people need to communicate, they need to learn to speak the same language, and they need to help each other develop a suitable transition scheme. The alternative would be organizational paralysis and the departure of the most capable people from both groups. Both should be aware that the most fortunate of the "old grumblers" could have been "young lions" last year, and if a person was given the opportunity to learn without any bullying, then he can become the most persistent and reasonable advocate of change. It will have invaluable properties of healthy skepticism, user knowledge and understanding of organizational barriers. Advocates of immediate and radical change need to realize that more often a transition is needed that involves the gradual introduction of new methods. On the other hand, those who do not want change should look for areas where it is possible, rather than waging fierce rearguard battles in an area where new requirements have already set completely different conditions for a successful project.

11.5 Code of practice

We have covered many topics in this chapter, but generally did not provide strong and specific design recommendations. This is consistent with my belief that there is no "one right solution". The principles and techniques should be applied in a way that is best suited to specific tasks. This requires taste, experience and intelligence. Still, you can specify some set of rules that the developer can use as guidelines until they gain enough experience to work out the best ones. Below is a set of such rules.

These rules can be used as a starting point in establishing guidelines for a project or organization, or as a checklist. Let me emphasize again that they are not universal rules and cannot replace reflection.

- Find out what you have to create.
- Set definite and tangible goals.
- Don't try to solve social problems with techniques .
- Count on long term

- in design, and
- people management.

- Use existing systems as models, inspiration, and starting point.
- Design for change:

  - flexibility,
  - extensibility,
  - portability, and
  - reuse.
- Document, propose, and maintain reusable components.
- Encourage and reward reuse

  - projects,
  - libraries, and
  - classes.
- Concentrate on component design.

  - Use classes to represent concepts.
  - Define interfaces to make the minimum amount of information required for the interface open .
  - Do strong typing of interfaces whenever possible.
  - Use application-scoped types in interfaces whenever possible.
- Research and refine both the design and implementation many times.
- Use the best verification and analysis tools available

  - the project, and
  - implementation.
- Experiment, analyze and test as early as possible.
- Strive for simplicity, maximum simplicity, but not beyond that.
- Don't go big, don't add features "just in case."
- Don't forget about efficiency.
- Maintain a level of formalization appropriate for the size of the project.
- Don't forget that developers, programmers, and even managers are still human.

Some more rules can be found in $$ 12.5

11.6 References with comments

In this chapter, we have only scratched the surface of the design and management of software projects. For this reason, a commented bibliography is provided below . A much more extensive list of references with comments can be found in [2].

1] Bruce Anderson and Sanjiv Gossain: An Iterative Design Model for Reusable Object-Oriented Softwa re. Proc. OOPSLA'90. Ottawa, Canada. pp. 12-27.

Description of the iterative design and redesign model with some examples and discussion of the results.

2] Grady Booch: Object Oriented Design. Benjamin Cummings. 1991.

This book contains a detailed design description, a specific design method with a graphical notation, and several large design examples recorded in various languages. This is an excellent book that influenced this chapter in many ways. It goes into more depth on many of the issues raised here.

3] Fred Brooks: The Mythical Man Month. Addison Wesley. 1982.

Everyone should reread this book every couple of years. A warning against arrogance. It is somewhat outdated in technical matters, but not outdated at all in terms of individual worker, organization, and size.

4] Fred Brooks: No Silver Bullet. IEEE Computer, Vol.20 No.4. April 1987.

A summary of different approaches to the development of large software systems, with a very useful warning against belief in magic recipes ("golden bullet").

5] De Marco and Lister: Peopleware. Dorset House Publishing Co. 1987.

One of the few books devoted to the role of the human factor in software production. A must for every manager. Calming enough to read before bed. A cure for many nonsense.

6] Ron Kerr: A Materialistic View of the Software "Engineering" Analogy. in SIGPLAN Notices, March 1987. pp 123-125.

The use of analogy in this and the next chapters is largely due to the observations from this article, as well as conversations with R. Kerr that preceded this.

7] Barbara Liskov: Data Abstraction and Hierarchy. Proc. OOPSLA'87 (Addendum). Orlando, Florida. pp 17-34.

Explores how the use of inheritance can damage the concept of abstract data. We point out that C ++ has special language tools that help to avoid most of these problems ($$ 12.2.5).

8] CN Parkinson: Parkinson's Law and other Studies in Administration. Houghton- Mifflin. Boston. 1957.

One of the funniest and most caustic descriptions of the ills that the administration process leads to.

9] Bertrand Meyer: Object Oriented Software Construction. Prentice Hall. 1988.

Pages 1-64 and 323-334 contain a good description of one look at object-oriented programming and design, as well as a lot of sound, practical advice. The rest of the book describes the Eiffel language.

10] Alan Snyder: Encapsulation and Inheritance in Object-Oriented Programming L anguages. Proc. OOPSLA'86. Portland, Oregon. pp. 38-45.

Perhaps the first good description of how wrapper interacts with inheritance. The article also discusses at a good level some concepts related to multiple inheritance.

11] Rebecca Wirfs -Brock, Brian Wilkerson, and Lauren Wiener: Designing Object-Oriented Software. Prentice Hall. 1990.

An anthropomorphic design method based on special CRC (Classes, Responsibilities, Collaboration) cards (i.e. Classes, Responsibilities, Collaboration) is described . The text, and maybe the method itself, gravitates towards the Smalltalk language.

CHAPTER 12. DESIGN AND C ++

Strive for simplicity, maximum simplicity, but not beyond that.
- A. Einstein

This chapter focuses on the relationship between design and the C ++ programming language. It explores the use of classes in design and identifies certain types of dependencies that should be distinguished both within a class and between classes. The role of static typing is explored. The application of inheritance and the relationship between inheritance and ownership are investigated . The concept of a component is discussed and some examples for interfaces are provided.

12.1 Design and programming language.

If I had to build a bridge, I would seriously think about what material to build it from, and the design of the bridge would be highly dependent on the material chosen, and therefore, reasonable stone bridge designs are different from reasonable metal bridge designs or reasonable wooden bridge designs etc. You should not rely on choosing a material suitable for a bridge without some knowledge of the materials and their use. Of course, you do not need to be a specialist carpenter to design a wooden bridge, but you do need to know the basics of wood construction in order to choose it over metal as a bridge material. What's more, while you don't have to be a specialist carpenter to design a wooden bridge , you need to know the properties of wood in sufficient detail and know even more about carpenters.

Likewise, when choosing a programming language for a certain software, you need to know several languages, and to successfully design a program, you need to know the chosen implementation language in sufficient detail , even if you personally do not have to write a single line of the program. A good bridge designer appreciates the properties of the materials they use and applies them to improve the design. Likewise, a good software developer takes advantage of the strengths of the implementation language and, as much as possible, tries to avoid using it in a way that would cause implementation difficulties.

You might think that this happens naturally if only one developer or programmer is involved in the design, but even in this case, the programmer, due to lack of experience or due to unjustified adherence to a programming style designed for completely different languages, can get lost for incorrect use. language. If the developer is significantly different from the programmer, especially if they have different programming cultures, the possibility of errors, ineffective and inelegant solutions in the final version of the system will almost certainly become inevitable.

So how can a programming language help a developer? It can provide such language tools that will allow you to express the basic concepts of the project directly in the programming language. This makes it easier to implement, easier to maintain consistency with the design, easier communication between developers and programmers, and the opportunity to create better tools for both developers and programmers.

For example, many design techniques place considerable emphasis on dependencies between different parts of a program (usually with the aim of reducing them and ensuring that those parts are understood and well defined). A language that allows explicit specification of interfaces between parts of a program can help developers in this matter . It can guarantee that only supposed dependencies will actually exist. Since most dependencies are explicitly expressed in a program in such a language, it is possible to develop tools that read the program and return dependency graphs . In this case, it is easier for the developer and other executors to understand the structure of the program. Programming languages such as C ++ help narrow the gap between project and program, and therefore reduce the possibility of confusion and misunderstandings.

The basic concept of C ++ is a class. A class is of a specific type. In addition, the class is the primary means of hiding information. You can describe programs in terms of user-defined types and hierarchies of these types. Both built-in and user-defined types are subject to static typing rules. Virtual functions provide a runtime binding mechanism without violating static type rules . Type templates allow you to create parameterized types. Special situations allow you to make a regular reaction to mistakes. All of these C ++ features can be used without additional overhead compared to a C program. These are the most important C ++ features that a developer should be aware of and consider. In addition, the availability of large libraries for the following purposes can significantly affect decision making at the design stage : for working with matrices, for communicating with databases, for supporting parallel programming, graphic libraries, etc.

Fear of novelty, unsuitable experience in other languages, in other systems or areas of application, poor design tools - all this leads to sub-optimal use of C ++. There are three things to note when a developer fails to capitalize on the capabilities of C ++ and accommodate the limitations of the language:

1] Ignoring classes and designing in such a way that programmers have to limit themselves to only C.

2] Ignoring derived classes and virtual functions, using only a subset of abstract data.

3] Ignore static type checking and design in such a way that programmers are forced to apply dynamic type checking. Usually, these moments arise from developers associated with:

1] C, or traditional CASE system or structural design methods ;

2] Ada or design methods using data abstraction;

3] languages close to Smalltalk or Lisp. In each case, it is necessary to decide: the implementation language was chosen incorrectly (assuming that the design method was chosen correctly), or the developer failed to adapt and evaluate the language (assuming that the implementation language was chosen correctly).

It should be said that there is nothing unusual or shameful about such a discrepancy. It's just that this discrepancy, which will lead to a non-optimal project, will impose additional work on the programmers, and in the case when the structure of the project's concepts is much poorer than the structure of the C ++ language, then on the developers themselves.

Note that not necessarily all programs should be structured based on the concepts of classes and / or class hierarchies, and not necessarily every program should use all the facilities provided by C ++. Quite the opposite, for the success of the project it is necessary that people are not forced to use the language they have just become acquainted with. The purpose of the following discussion is not to impose dogmatic use of classes, hierarchies, and strongly typed interfaces, but to show how they can be used wherever the scope of the application, the limitations of C ++, and the experience of the performers allow . $$ 12.1.4 discusses approaches to different uses of C ++ in a project titled "Hybrid Project".

12.1.1 Ignoring classes

Consider the first of these points - ignoring classes. In this case, the resulting C ++ program will be approximately equivalent to a C program developed according to the same project, and we can say that they will be approximately equivalent to programs in Ada or Cobol developed according to the same project . In fact, the project is structured as independent of the implementation language, which forces the programmer to limit himself to a common subset of the C, Ada or Cobol languages. There are advantages here. For example, the resulting strict separation of data and program code makes it easy to use traditional databases that are designed for such programs. Since a limited programming language is used , less experience (or at least a different level of experience) is required from programmers . For many applications, for example, for traditional databases that work with a file sequentially, this approach is quite reasonable, and traditional techniques, worked out over decades, are quite adequate for the task.

However, where the scope of the application is significantly different from the traditional sequential processing of records (or characters), or the complexity of the task is higher, as, for example, in the CASE dialog system , the lack of language support for abstract data due to the abandonment of classes (if they are not taken into account) will hurt project. The complexity of the task will not decrease, but since the system is implemented in a lean language, the structure of the program will not respond well to the project. It is too large, lacks type checking, and is generally ill-suited to the use of various aids. This is a path leading to nightmares when accompanying her .

Usually, to overcome these difficulties, special tools are created that support the concepts used in the project. Thanks to them, higher- level constructs are created and checks are organized in order to compensate for defects (or deliberate impoverishment) of the implementation language. So the design method becomes an end in itself, and a special programming language is created for it . Such programming languages are in most cases a poor substitute for widely used general-purpose programming languages that are accompanied by suitable design tools. It makes no sense to use C ++ with such a limitation, which should be compensated for when designing with special means. Although the mismatch between the programming language and the design tools may be just a stage in the transition process, and therefore a temporary phenomenon.

The most common reason for ignoring classes in design is simple inertia. Traditional programming languages do not provide the concept of a class, and traditional design methods reflect this flaw. Typically, in the design process, most attention is paid to breaking down a task into procedures that perform the required actions. In Chapter 1, this concept was referred to as procedural programming, and in the design field, it was referred to as functional decomposition. A typical question arises: "Can C ++ be used in conjunction with a design method based on functional decomposition?" Yes, you can, but most likely, as a result, you will end up using C ++ as just improved C with all the problems mentioned above. This may be acceptable during the transition to a new language, or for already completed design, or for subtasks in which the use of classes does not provide significant benefits (if we take into account the experience of programming in C ++ at this point), but in general over a large segment time, the abandonment of the free use of classes associated with the method of functional decomposition is in no way compatible with the effective use of C ++.

Procedural and object-oriented programming approaches are different in nature and usually lead to completely different solutions to the same problem. This conclusion is true for both the implementation stage and the design stage: you focus on either the actions you take or the entities represented, but not both at the same time.

So why is object-oriented design preferred over functional decomposition? The main reason is that functional decomposition does not provide sufficient data abstraction. And from this it already follows that the project will be

- less susceptible to changes,
- less adapted for the use of various aids ,
- less suitable for parallel development and
- less suitable for parallel execution.

The point is that functional decomposition forces us to declare “important” data as global, because if the system is structured as a tree of functions, any data available to two functions must be global in relation to them. This causes “important” data to “float” to the top of the tree as more and more functions require access to it.

This is exactly the case for a single- root class hierarchy , where "important" data floats towards the base class.

When we focus on class descriptions that enclose certain data in a shell, then the dependencies between different parts of the program are explicitly expressed and can be traced. More importantly, this approach reduces the number of dependencies in the system due to better placement of data references.

However, some tasks are better solved using a set of procedures. The point of "object-oriented" design is not to remove all global procedures from the program or to have no procedural parts in the system. Rather, the main idea is that classes, rather than global procedures, become the main focus of attention during design. Using a procedural style should be a deliberate decision, not a default. Both classes and procedures should be applied according to the scope of the application, not just as immutable design techniques.

12.1.2 Ignoring inheritance

Consider option 2 - a project that ignores inheritance. In this case, the final program simply does not use the capabilities of the main C ++ tool, although there are certain benefits when using C ++ compared to using languages C, Pascal, Fortran, Cobol, etc. The usual arguments for this, in addition to inertia, are that "inheritance is an implementation detail" or "inheritance prevents information being hidden" or "inheritance makes it difficult to interact with other programming systems."

Treating inheritance as just an implementation detail is to ignore a class hierarchy that can directly model relationships between concepts in the application domain. Such relationships must be explicitly expressed in the project to enable the developer to think through them.

There is a strong case for eliminating inheritance from parts of a C ++ program that directly interact with programs written in other languages. But this is not a good enough reason to abandon inheritance in the system as a whole, it is simply an argument for carefully defining and encapsulating the programming interface with the "outside world". Likewise, to get rid of the anxiety caused by the confusion about information hiding in the presence of inheritance, one must use virtual functions and private members carefully , but not abandon inheritance.

There are many situations where there are no clear benefits to using inheritance, but a "no inheritance" policy will lead to a less understandable and less flexible system in which inheritance is "faked" by more traditional language and design constructs. This is essential for large projects. Moreover, it is possible that despite this policy, inheritance will still be used, since C ++ programmers will find compelling reasons for designing with inheritance in different parts of the system. Thus, the policy of "no inheritance" will only lead to the absence of a coherent overall structure in the system , and the use of the class hierarchy will be limited to certain subsystems.

In other words, be open-minded. A class hierarchy is not a required part of every good program, but there are many situations where it can help both in understanding the scope of the application and in formulating solutions. The assertion that inheritance can be misused or overused serves only as an argument for caution, not at all in favor of abandoning it.

12.1.3 Ignoring Static Type Checking

Consider option 3 for a project that ignores static typing. Common arguments for avoiding static typing at design time are that "types are a product of programming languages," or that "it's more natural to think about objects without worrying about types", or "static typing forces us to think about implementation too early. " This approach is perfectly valid as long as it works and is not harmful. It is perfectly reasonable not to worry about the details of type checking during the design phase , and it is often perfectly acceptable to completely forget about the type issues during the analysis and early design phases . At the same time, classes and class hierarchies are very useful at the design stage, in particular, they give us more definition of concepts, allow us to accurately define relationships between concepts, and help to reason about concepts. As the project evolves, this certainty and precision translates into more and more concrete statements about classes and their interfaces.

It is important to understand that well-defined and strongly typed interfaces are a fundamental design tool. The C ++ language was created with this in mind. A strongly typed interface ensures that only compatible parts of a program can be compiled and linked together, and thus allows relatively strong assumptions to be made about those parts. These assumptions are enforced by the language's type system. As a result, run-time checks are minimized , which increases efficiency and leads to a significant reduction in the integration phase of parts of the project implemented by different programmers. The real positive experience of integrating a system with strongly typed interfaces has led to the fact that integration issues do not figure among the main topics of this chapter at all.

Consider the following analogy: In the physical world, we are constantly connecting different devices, and there is a seemingly endless number of connection standards. The main feature of these connections: they are specially designed in such a way as to make it impossible to connect two devices that are not designed for it, that is, the connection must be made in the only correct way. You cannot connect the shaver to a high voltage outlet. If you could do this, you would burn the razor or burn yourself. A lot of ingenuity was shown to make it impossible to connect two incompatible devices. An alternative to the simultaneous use of several incompatible devices can be a device that protects itself from incompatible devices that connect to its input. A voltage regulator is a good example . Since perfect device compatibility cannot be guaranteed only at the "connection level", sometimes more expensive protection in the electrical circuit is required , which allows the speaker to adapt and / and protect against voltage surges.

This is almost a direct analogy: static type checking is equivalent to connection-level compatibility, and dynamic type checking corresponds to protection or adaptation in the chain. The result of a failed control both physically and in the software world is serious damage. In large systems, both types of control are used. At an early stage of design, a simple statement is sufficient: "These two devices need to be connected"; but it soon becomes essential how to connect them: "What guarantees does the connection give regarding the behavior of the devices?" or "What errors are possible?", or "What is the approximate cost of such a connection?"

The use of "static typing" is not limited to the programming world. In physics and engineering, units of measurement (meters, kilograms, seconds) are ubiquitous in order to avoid mixing incompatible entities.

In our description of the design steps in $$ 11.3.3, types appear on the scene already at step 2 (obviously, after some artificial consideration of them at step 1) and become the main topic of step 4.

Statically controlled interfaces are the main means of interaction between software parts of a C ++ system created by different groups, and the description of the interfaces of these parts (taking into account the exact type definitions) becomes the main way of cooperation between individual groups of programmers. These interfaces are the main output of the design process and serve as the main means of communication between developers and programmers.

Failure to do this leads to projects in which the structure of the program is unclear , error control is postponed to the execution stage , which are difficult to implement well in C ++.

Consider an interface described using "objects" that define themselves. Perhaps, for example, this description: "The function f () has an argument that must be an airplane" (which is checked by the function itself at runtime), as opposed to the description "The function f () has an argument, the type of which is an airplane" (which checked by the translator). The first description is a significantly insufficient description of the interface, since results in dynamic checking instead of static checking. A similar conclusion from the example with an airplane is made in $$ 1.5.2. It uses more precise specifications, and uses type pattern and virtual functions instead of unlimited dynamic checks to move error detection from runtime to translation. The difference in the running times of programs with dynamic and static control can be quite significant, usually in the range from 3 to 10 times.

But one should not go to the other extreme. You cannot detect all errors using static inspection. For example, even programs with the most extensive static monitoring are vulnerable to hardware failures . Still, ideally, you should have a wide variety of statically typed interfaces using application- scoped types , see $$ 12.4.

It can happen that a project that is perfectly reasonable at an abstract level will run into serious problems if it does not take into account the limitations of the basic facilities, in this case C ++. For example, using names rather than types to structure a system would lead to unnecessary problems for the C ++ type system, and thus can cause errors and runtime overhead. Consider three classes:

```
class X { // pseudo code, not C ++
        f ()
        g ()
}
```

```
class Y {
        g ()
        h ()
}

class Z {
        h ()
        f ()
}
```

used by some of the functions of a typeless project:

```
k (a, b, c) // pseudo code, not C ++
{
        af ()
        bg ()
        ch ()
}
```

Here appeals

```
X x
Y y
Z z

k (x, y, z) // ok
k (z, x, y) // ok
```

will be successful because k () simply requires its first parameter to have an f () operation, the second parameter to a g () operation, and its third parameter to have an h () operation. On the other side of the appeal

```
k (y, x, z); // fail
k (x, z, y); // fail
```

will fail. This example allows perfectly reasonable implementations in languages with full dynamic control (such as Smalltalk or CLOS), but it has no direct representation in C ++ because the language requires that type generality be implemented as a relation to the base class. Typically, examples like this can be imagined in C ++ by writing generic statements using explicit class definitions, but this requires a lot of cleverness and aids. You can do, for example, like this:

```
class F {
        virtual void f ();
};

class G {
        virtual void g ();
};

class H {
        virtual void h ();
};

class X: public virtual F, public virtual G {
        void f ();
        void g ();
};

class Y: public virtual G, public virtual H {
        void g ();
        void h ();
};

class Z: public virtual H, public virtual F {
        void h ();
        void f ();
};

k (const F & a, const G & b, const H & c)
{
        af ();
        bg ();
        ch ();
}

main ()
```

```
{
            X x;
            Y y;
            Z z;
            k (x, y, z); // ok
            k (z, x, y); // ok
            k (y, x, z); // error F required for first argument
            k (x, z, y); // error G required for second argument
}
```

Note that by making k ()'s assumptions about its arguments explicit, we have moved error control from runtime to translation. Complicated examples like this one arise when trying to implement projects in C ++ based on experience with other type systems. This is usually possible, but the result is an unnatural and ineffective program. This mismatch between design techniques and a programming language can be compared to mismatch in word-by-word translation from one natural language to another. After all, English with German grammar looks as awkward as German with English grammar, but both languages can be understood by someone who is fluent in one of them.

This example confirms the conclusion that the classes in the program are a concrete embodiment of the concepts used in the design, therefore fuzzy relations between the classes lead to the fuzziness of the basic design concepts.

12.1.4 Hybrid project

The transition to new ways of working can be painful for any organization. The divisions within it and the differences between employees can be significant. But an abrupt, decisive transition that can overnight transform effective and skilled “old school” supporters into ineffective newcomers to “new school” is usually unacceptable. At the same time, one cannot reach great heights without changes, and significant changes are usually associated with risk.

C ++ was designed to reduce this risk through the gradual introduction of new methods. While it is clear that the greatest benefits to using C ++ come from data abstraction , object-oriented programming, and object-oriented design, it is far from clear that the fastest way to achieve this is by breaking with the past. It is unlikely that such a clear gap will be possible, usually the desire for improvement is constrained or should be constrained in order for the transition to them to be controlled. Consider the following:

- Developers and programmers need time to master new techniques.
- New programs should interact with old programs.
- Old programs need to be maintained (often indefinitely).
- Work on current projects and programs must be completed on time.
- Tools designed for new methods need to be adapted to the local environment.

Here are just the situations associated with the listed requirements. It is easy to underestimate the first two requirements.

Since several programming schemes are possible in C ++, the language allows for a gradual transition to it, taking advantage of the following advantages of such a transition:

- By learning C ++, programmers can keep working.
- In a software-poor environment, using C ++ can bring significant benefits.
- Programs written in C ++ can interface well with programs written in C or other traditional languages.
- The language has a large subset compatible with C.

The idea is to gradually switch the programmer from a traditional language to C ++: first, he programs in C ++ in the traditional procedural style, then using data abstraction methods, and finally, when he has mastered the language and related tools, he completely switches to object- oriented programming. Note that a well-designed library is much easier to use than it is to design and implement, so even from the first steps a beginner can benefit from using more advanced C ++ tools.

The idea of gradual, step-by-step mastery of C ++, as well as the ability to mix C ++ programs with programs written in languages that do not have data abstraction and object-oriented programming, naturally leads to a project that has a hybrid style. Most of the interfaces can be left at the procedural level for now, since anything more complex will not bring immediate benefits. For example, a call to the math standard library from C is defined in C ++ like this:

```
extern "C" {
```

```
            #include <math.h>
}
```
and the standard math functions from the library can be used in the same way as in C. For all major libraries, this inclusion must be done by the vendor of the libraries, so that the C ++ programmer does not even know which language the library function is implemented in . Using libraries written in languages like C is the first and most important way to reuse in C ++ at the beginning .

In the next step, when more sophisticated techniques become necessary , tools implemented in languages such as C or Fortran are represented as classes by encapsulating data structures and functions in a C ++ class interface. A simple example of introducing a higher semantic level by moving from the procedure plus data structures level to the data abstraction level is the string class of $$ 7.6. Here, by encapsulating character strings and standard C string functions, you get a new string type that is much easier to use.

Similarly, you can include any built-in or separately defined type in the class hierarchy . For example, the int type can be included in the class hierarchy like this:

```
class Int: public My_object {
            int i;
            public:
                        // definition of operations
                        // see exercises [8] - [11] in section 7.14 for ideas
                        // operation definitions are obtained in exercises [8] -
[11]
                        // see section 7.14 for ideas
};
```

This should be done if there really is a need to include such types in the hierarchy.

Conversely, C ++ classes can be represented as functions and data structures in a C or Fortran program . For example:

```
class myclass {
            // representation
            pub lic:
                    void f ();
                    T1 g (T2);
                    // ...
};

extern "C" {// map myclass into C callable functions:
          void myclass_f (myclass * p) {p-> f (); }
          T1 myclass_g (myclass * p, T2 a) {return p-> g (a); }
          // ...
};
```

In a C program, you define these functions in the header file as follows:

```
// in C header file

extern void myclass_f (struct myclass *);
extern T1 myclass_g (struct myclass *, T2);
```

This approach allows a C ++ developer, if he already has a stock of programs written in languages that lack the concepts of data abstraction and class hierarchy, gradually become familiar with these concepts, even though the requirement that the final version of the program can be called from traditional procedural languages.

12.2 Classes

The basic premise of object-oriented design and programming is that the program serves as a model for some concepts of reality. The classes in the program represent the basic concepts of the application area and, in particular, the basic concepts of the reality modeling process itself. Class objects represent real-world objects and products of the implementation process .

We will consider the structure of the program in terms of the following relationships between classes:

- inheritance relations,
- the relationship of belonging,
- the relationship of use and
- programmed relationships.

When considering these relations, it is implicitly assumed that their analysis is a key moment in the design of the system. $$ 12.4 explores the properties that make a class and its interface useful for representing concepts. Generally speaking, ideally, the dependence of the class on the rest of the world should be minimal and clearly defined, and the class itself should open only a minimal amount of information through the interface to the rest of the world.

We emphasize that a class in C ++ is a type, therefore the classes themselves and the relationships between them are provided with significant support from the translator and, in the general case, lend themselves to static analysis.

12.2.1 What are classes?

In fact, there are two types of classes in the system:

1] classes that directly reflect the concepts of the application scope, ie. the concepts that the end user uses to describe their tasks and possible solutions; and

2] classes that are a product of the implementation itself, ie. reflect concepts used by developers and programmers to describe implementation methods.

Some of the classes that are products of the implementation may also represent real world concepts. For example, the software and hardware resources of a system are good candidates for classes that represent the scope of an application. This reflects the fact that a system can be viewed from multiple perspectives, and what is an implementation detail on one can be an application domain concept on the other. A well- designed system should contain classes that allow you to view the system from logically different points of view. Let's give an example:

1] classes representing user-defined concepts (eg cars and trucks),

2] classes representing generalizations of user-defined concepts (moving vehicles),

3] classes that represent hardware resources (eg, a memory management class),

4] classes representing system resources (eg output streams), 5] classes used to implement other classes (eg lists, queues, blockers) and

6] built-in data types and control structures.

In large systems, it is very difficult to maintain a logical separation of the types of different classes and maintain such separation between different levels of abstraction. The above listing provides three levels of abstraction:

1 + 2] represents a custom reflection of the system,

3 + 4] represents the machine on which the system will run,

5 + 6] represents a low-level (from the programming language side) implementation reflection.

The larger the system, the more levels of abstraction are needed to describe it, and the more difficult it is to define and maintain these levels of abstraction. Note that such levels of abstraction have a direct correspondence in nature and in various constructions of human intelligence. For example, you can consider a house as an object consisting of

1] atoms,

2] molecules,

3] planks and bricks,

4] floors, ceilings and walls;

5] rooms.

As long as the representations of these levels of abstraction can be kept separate, a holistic view of the house can be maintained. However, if you mix them, nonsense will arise. For example, the sentence "My house is made up of several thousand pounds of carbon, some complex polymers, 5000 bricks, two bathrooms and 13 ceilings" is clearly absurd. Due to the abstract nature of programs, such a statement about a complex software system is not always perceived as nonsense.

In the design process, isolating concepts from application domain to class is by no means a simple mechanical operation. This task usually requires a lot of discernment. Note that the concepts of application scope themselves are abstractions. For example, "taxpayers", "monks" or "employees" do not exist in nature. These concepts are nothing more than labels that designate a poor person in order to classify him in relation to a certain system. Often the real or imaginary world (for example, literature, especially fiction) serves as a source of concepts that are radically transformed when they are translated into classes. So, the screen of my computer (McKintosh) does not at all resemble the surface of my desk, although the computer was created with the aim of realizing the concept of "desktop", and the windows on my display have very little to do with the devices for presenting drawings in my room. I couldn't stand such a mess on my screen.

The essence of modeling reality is not obediently following what we see, but using reality as a beginning for design, a source of inspiration and as an anchor that holds when the element of programming threatens to deprive us of the ability to understand our own program.

A good word of caution here: it is usually difficult for beginners to "find" the classes, but this is soon overcome without any hassle. The next stage usually comes when classes and inheritance relationships between them multiply uncontrollably. There are already problems with the complexity, efficiency and clarity of the resulting program. Not every single detail should be represented as a separate class, and not every relationship between classes should be represented as an inheritance relationship . Try to remember that the goal of a project is to model a system with a suitable level of detail and a suitable level of abstraction. For large systems, finding a compromise between simplicity and generality is far from easy.

12.2.2 Class hierarchies

Let us consider the simulation of traffic flow in a city, the purpose of which is to accurately determine the time it takes for emergency vehicles to reach their destination. Obviously, we need to have ideas about cars and trucks, ambulances, all kinds of fire and police cars, buses, etc. Since any concept of the real world does not exist in isolation, but is connected by numerous connections with other concepts, such a relationship arises as inheritance. Without understanding the concepts and their interrelationships, we are not able to comprehend any separate concept. Also, a model, if it does not reflect the relationship between concepts, cannot adequately represent the concepts themselves. So, our program needs classes to represent concepts, but this is not enough. We need ways to represent relationships between classes. Inheritance is a powerful way to represent hierarchical relationships directly . In our example, we would most likely consider the emergency vehicles to be special moving vehicles and, in addition, would allocate funds represented by cars and trucks. Then the class hierarchy would look like this:

```
moving vehicle
passenger car
emergency facility
truck
police car
ambulance
fire engine
ladder machine
```

Here, the Emergency class represents all the information necessary to model emergency vehicles, for example: an emergency vehicle may violate certain traffic rules, it has priority at intersections, is under the control of a dispatcher , etc.

In C ++, it can be set like this:

```
class Vehicle {/*...*/};
class Emergency {/ * * /};
class Car: public Vehicle {/*...*/};
class Truck: public Vehicle {/*...*/};
class Police_car: public Car, public Emergency {
        // ...
};
class Ambulance: public Car, public Emergency {
        // ...
};
class Fire_engine: public Truck, Emergency {
        // ...
};
class Hook_and_ladder: public Fire_engine {
        // ...
};
```

Inheritance is the highest order relationship that is directly represented in C ++ and is used primarily in the early design stages. There is often a problem of choice: whether to use inheritance to represent the relationship, or prefer ownership. Let us consider another definition of the concept of emergency means: a moving vehicle is considered emergency if it carries a corresponding light signal. This will simplify the class hierarchy by replacing the Emergency class with a member of the Vehicle class :

```
vehicle (Vehicle {eptr})
passenger car (Car)
truck
police car (Police_car)
ambulance (Ambulance)
fire truck (Fire_engine)
retractable ladder machine (Hook_and_ladder)
```

The Emergency class is now used simply as a member in those classes that represent emergency vehicles:

```
class Emergency {/*...*/};
class Vehicle {public: Emergency * eptr; /*...*/};
class Car: public Vehicle {/*...*/};
class Truck: public Vehicle {/*...*/};
c lass Police_car: public Car {/*...*/};
class Ambulance: public Car {/*...*/};
class Fire_engine: public Truck {/*...*/};
class Hook_and_ladder: public Fire_engine {/*...*/};
```

Here, the vehicle is considered emergency if Vehicle :: eptr is nonzero. "Simple" cars and trucks are initialized with Vehicle :: eptr equal to zero, and for other Vehicle :: eptr should be set to a non-zero value, for example:

```
Car :: Car () // Car constructor
{
        eptr = 0;
}
Police_car :: Police_car () // Police_car constructor
{
        eptr = new Emergency;
}
```

Such definitions simplify the conversion of an emergency facility to a conventional facility and vice versa:

```
void f (Vehicle * p)
{
        delete p-> eptr;
```

```
            p-> eptr = 0; // no more emergency mover
            // ...
            p-> eptr = new Emergency; // it appeared again
    }
```

So what is the best class hierarchy? In general, the answer is: "The best is the program that most directly reflects the real world." In other words, when choosing a model, we should strive for its greater "reality", but taking into account the inevitable limitations imposed by the requirements of simplicity and efficiency. Therefore, despite the simplicity of converting an ordinary moving vehicle into an emergency one, the second solution seems to be impractical. Fire trucks and ambulances are special moving vehicles with specially trained personnel, they operate under the control of dispatcher commands , requiring special equipment for communication. This situation means that belonging to emergency vehicles is a basic concept that should be directly represented in the program to improve type control and use of various software tools . If we were to simulate a situation in which the destination of a moving vehicle is not so definite, say, a situation in which private vehicles are periodically used to bring special personnel to the scene of an accident, and communication is provided using portable receivers, then another method of modeling might be appropriate. systems.

For those who find the traffic simulation example exotic, it makes sense to say that in the design process , this kind of choice between inheritance and ownership almost constantly arises . There is a similar example in $$ 12.2.5, which describes a scrollbar - scrolling information in a window.

12.2.3 Dependencies within a class hierarchy.

Naturally, the derived class depends on its base classes. It is much less often taken into account that the opposite may also be true.

This thought can be expressed in this way: "Madness is inherited, you can get it from your children."

If a class contains a virtual function, derived classes are free to decide whether to implement some of the operations of that function every time it is overridden in the derived class. If a member of a base class calls one of the virtual functions of a derived class itself, then the implementation of the base class depends on the implementations of its derived classes. Likewise, if a class uses a protected member, its implementation will depend on derived classes. Consider the definitions:

```
class B {
        // ...
        protected:
                int a;
        public:
                virtual int f ();
                int g () {int x = f (); return xa; }
};
```

What is the result of running g ()? The answer essentially depends on the definition of f () in some derived class. Below is an option where g () will return 1:

```
class D1: public B {
        int f () {return a + 1; }
};
```

and under the following definition, g () will print "Hello, World" and return 0:

```
class D1: public {
        in t f () {cout << "Hello, World \ n"; return a; }
};
```

This example demonstrates one of the most important points related to virtual functions. Although you might say that this is nonsense, and a programmer will never write anything like this. The point here is that the virtual function is part of the interface with the base class, and that this class will most likely be used without information about its derived classes. Therefore, you can describe the behavior of a base class object in such a way that you can write programs later without knowing anything about its derived classes.

Any class that overrides a derived function must implement a variant of that function. For example, the virtual function rotate () from the Shape class rotates a geometric shape, and the rotate () functions for derived classes such as Circle and Triangle must rotate objects of the corresponding types, otherwise the basic premise of the Shape class will be violated . But no provisions have been formulated about the behavior of class B or its derived classes D1 and D2, so the example given seems unreasonable. When constructing a class, the main focus should be on describing the expected actions of the virtual functions.

Should the dependence on unknown (possibly still undefined) derived classes be considered normal ? The answer naturally depends on the goals of the programmer. If the goal is to isolate the class from all external influences and thereby prove that it behaves in a certain way, then it is better to avoid virtual functions and protected members. If the goal is to develop a structure into which subsequent programmers (or you yourself in a week) can embed their programs, then it is virtual functions that offer an elegant way of solving, and protected members can be useful in implementing it.

As an example, consider a simple type template that defines a buffer:

```
template <class T>
class buffer {
        // ...
        void put (T);
        T get ();
};
```

If the response to overflow and access to an empty buffer is "soldered" into the class itself, its use will be limited. But if the functions put () and get () refer to the virtual functions overflow () and underflow (), respectively, then the user can, satisfying his needs, create buffers of various types:

```
template <class T>
class buffer {
        // ...
        virtual in t overflow (T);
        virtual int underflow ();
        void put (T); // call overflow (T) when the buffer is full
        T get (); // call underflow (T) when the buffer is empty
};

template <class T>
class circular_buffer: public buffer <T> {
        // ...
        int overflow (T); // go to the beginning of the buffer if it is full
```

```
            int underflow ();
};

template <class T>
class expanding_buffer: public buffer <T> {
            // ...
            int overflow (T); // increase buffer size if full
            int underflow ();
};
```

This method was used in the streaming I / O libraries ($$ 10.5.3).

12.2.4 Membership relations

If a membership relation is used, then there are two main ways to represent an object of class X:

l] Describe a member of type X.

2] Describe a member of type X * or X &.

If the value of the pointer will not change and you don't care about efficiency, these methods are equivalent:

```
class X {
            // ...
            public:
                        X (int);
                        // ...
};

class C {
            X a;
            X * p;
            public:
                        C (int i, int j): a (i), p (new X (j)) {}
                        ~ C () {delete p; }
};
```

In such situations, direct object membership, like X :: a in the example above, is preferable because it saves time, memory, and input. See also $$ 12.4 and $$ 13.9.

The method using a pointer should be used in cases where you have to rebuild a pointer to an "element object" during the life of the "owner object". For example:

```
   class C2 {
             X * p;
             public:
                       C (int i): p (new X (i)) {}
                       ~ C () {delete p; }
                       X * change (X * q)
                       {
                                 X * t = p;
                                 p = q;
                                 return t;
                       }
   };
```

A pointer member can also be used to allow an "item-object" to be passed as a parameter:

```
   class C3 {
             X * p;
             public:
                       C (X * q): p (q) {}
                       // ...
   }
```

By allowing objects to contain pointers to other objects, we create what is commonly called an "object hierarchy". This is an alternative and auxiliary way of structuring in relation to the class hierarchy . As shown with the emergency mover in $$ 12.2.2, it is often a rather delicate design issue to represent a property of a class as another base class or as a member of a class. The need for an override should be considered an indication that the first option is better. But if you need to be able to represent some property using different types, then it is better to stop at the second option. For example:

```
   class XX: public X {/*...*/};
   class XXX: public X {/*...*/};

   void f ()
```

```
{
            C3 * p1 = new C3 (new X); // C3 "contains" X
            C3 * p2 = new C3 (new XX); // C3 "contains" XX
            C3 * p3 = new C3 (new XXX); // C3 "contains" XXX
            // ...
}
```

The above definitions cannot be modeled with either a C3- derived class of X, nor with C3 having a member of type X, since the exact member type must be specified. This is important for classes with virtual functions, such as the Shape class ($$ 1.1.2.5), and for the abstract set class ($$ 13.3).

Note that references can be used to simplify classes that use pointer members if, during the lifetime of the owner object, the reference is configured on only one object, for example:

```
class C4 {
            X & r;
            public:
                        C (X & q): r (q) {}
                        // ...
};
```

12.2.5 Ownership and Inheritance

Given the complexity of the importance of inheritance relationships, it is not surprising that they are often misunderstood and overused. If class D is described as generically derived from class B, then it is often said that D is B:

```
class B {/ * ... * /;
class D: public B / * ... * /}; // D grade B
```

Otherwise, it can be formulated as follows: inheritance is the relationship "is", or, more precisely for classes D and B, inheritance is a relationship D of grade B. In contrast, if class D contains another class B as a member, then they say that D "has" B:

```
class D {// D has B
            // ...
            public:
                        B b;
                        // ...
};
```

In other words, belonging is a "have" relationship or, for classes D and B, simply: D contains B.

Given two classes B and D, how do you choose between inheritance and ownership? Consider the airplane and engine classes, newbies usually ask if it would be a good idea to make the airplane class derived from the engine class. This is a bad decision, since the plane does not "have" a motor, the plane "has" a motor. One should approach this question by considering whether an airplane can "have" two or more engines. Since this seems to be quite possible (even if we are dealing with a program in which all aircraft will be with the same motor), ownership should be used, not inheritance. The question "Can he have two ..?" turns out to be surprisingly useful in many questionable cases. As always, our presentation touches upon the elusive nature of programming. If all classes were as easy to imagine as an airplane and a motor, then it would be easy to avoid trivial mistakes like the one when an airplane is defined as a derivative of the class motor. However, such errors are quite common, especially among those who consider inheritance to be another mechanism for combining programming language constructs. Despite the convenience and conciseness of notation that inheritance provides, it should be used only to express those relationships that are clearly defined in the project. Consider the definitions:

```
class B {
        public:
                virtual void f ();
                void g ();
};

class D1 { // D1 contains B
        public:
                B b;
                void f (); // does not override bf ()
};

void h1 (D1 * pd)
{
        B * pb = pd; // error: unable to convert D1 * to B *
```

```
            pb = & pd-> b;
            pb-> q (); // call B :: q
            pd-> q (); // error: D1 has no member q ()
            pd-> bq ();
            pb-> f (); // call B :: f (here D1 :: f does not override)

            pd-> f (); // call D1 :: f
   }
```

Note that this example does not implicitly convert a class to one of its members, and that a class that contains another class as a member does not override the virtual functions of that member. Here's a clear difference from the example below:

```
   class D2: public B { // D2 is B
            public:
                        void f (); // override B :: f ()
   };

   void h2 (D2 * pd)
   {
            B * pb = pd; // ok: D2 * implicitly converts to B *
            pb-> q (); // call B :: q
              pd-> q (); // call B :: q
            pb-> f (); // call virtual function: call to D2 :: f
            pd-> f (); // call D2 :: f
   }
```

The writeability demonstrated in the D2 example versus the D1 example is the reason why such inheritance is overused. Keep in mind, however , that there is a price to pay for ease of use in the form of an increased relationship between B and D2 (see $$ 12.2.3). In particular, it's easy to forget about the implicit conversion of D2 to B. Unless such conversions are relevant to the semantics of your classes, you should avoid describing the derived class in the generic part. If a class represents a specific concept, and inheritance is used as an "is" relationship, then such transformations are usually just what you need.

However, there are situations where it is desirable to have inheritance but cannot be converted. Consider defining a controled field (cfield) class that, among other things, allows you to control access to another field class at runtime. At first glance, it seems perfectly correct to define the cfield class as deriving from the field class:

```
class cfield: public field {
            // ...
};
```

This expresses the fact that cfield is indeed a sort of field, makes it easier to write a function that uses a member of the field part of the cfield class , and, most importantly, allows cfi eld to override field virtual functions. The catch here is that the conversion from cfield * to field * found in the definition of the cfield class allows you to bypass any access controls on field:

```
void q (cfield * p)
{
            * p = "asdf"; // field access is controlled
                                                        // cfield assignment
function:
                                                        // p-> cfield ::
operator = ("asdf")
            field * q = p; // implicit conversion of cfield * to field *
            * q = "asdf"; // arrived! control bypassed
}
```

It would be possible to define the cfield class so that field is a member, but then cfield cannot override the virtual functions of field. The best solution here is to use inheritance with the private specification (private inheritance):

```
class cfield: private field {/ * ... * /}
```

From a design perspective, aside from the (sometimes important) override issues , private inheritance is equivalent to ownership. In this case, a method is used in which the class is defined in the general part as derived from the abstract base class by specifying its interface, and also defined using private inheritance from the concrete class that specifies the implementation ($$ 13.3). Since inheritance used as private is implementation specific and is not reflected in the type of the derived class, it is sometimes referred to as "implementation inheritance" and is a contrast to inheritance in the general part when the base class interface is inherited and implicit conversions to base type. The latter inheritance is sometimes called subtype definition or "interface inheritance".

For further discussion of the possibility of choosing inheritance or ownership, we will consider how to present a scroll (an area for scrolling information in it) in an interactive graphics system , and how to bind a scroll to a window on the screen. You will need two types of scrolls: horizontal and vertical. This can be represented with the two types horizontal_scrollbar and vertical_scrollbar, or with a single scrollbar that takes an argument that specifies whether the layout is vertical or horizontal. The first solution assumes that there is a third type that simply defines a scrollbar, and this type is the base class for the two defined scrolls. The second solution assumes an additional argument for the scrollbar type and the presence of values that specify the type of the scroll. For example, like this:

```
enum orientation {horizontal, vertical};
```

As soon as we focus on one of the solutions, the amount of changes that will have to be made to the system will be determined . Let's say in this example we need to enter scrolls of the third kind. Initially, it was assumed that there could be only two types of scrolls (after all, every window has only two dimensions), but in this example, as in many others, extensions are possible that arise as redesign issues. For example, you might want to use a "control button" (like a mouse) instead of two kinds of scrolls . Such a button would set scrolling in different directions depending on which part of the window the user clicked on. Pressing in the middle of the top line should cause "scrolling up", pressing in the middle of the left column - "scrolling left", pressing in the upper left corner - "scrolling up and left". Such a button is not unusual, and it can be seen as a refinement of the concept of a scroll, which is especially suitable for areas of the application that involve more complex information rather than plain text .

To add a control button to a program that uses a hierarchy of three scrolls, you need to add another class, but you do not need to change the program that works with old scrolls:

```
scroll
horizontal_scroll
vertical_scroll
control_button
```

This is the positive side of the "hierarchical solution".

Setting the orientation of the rollout as a parameter causes the type fields to be set in the rollout objects and the use of radio buttons in the body of the rollout member functions. In other words, we are faced with a common dilemma: to express this aspect of the system structure using definitions or to implement it in the operator part of the program. The first solution increases the amount of static checks and the amount of information that various aids can work on. The second solution defers checks to the execution stage and allows changing the bodies of individual functions without changing the overall structure of the system as it appears from the point of view of static control or auxiliary means. In most cases, the first solution is preferable.

The positive side of the solution with a single scroll type is that it is easy to transfer information about the type of scroll we need to another function:

```
void helper (orientation oo)
{
            // ...
            p = new scrollbar (oo);
            // ...
}

void me ()
{
            helper (horizontal);
}
```

This approach makes it easy to readjust the scroll to a different orientation at runtime . This is unlikely to be very important in the scrolls example, but it can be significant in similar examples. The bottom line is that you always have to make certain choices, and this is often not easy.

Now let's look at how to bind a scroll to a window. If we consider window_with_scrollbar (window_with_scrollbar) to be something that is window and scrollbar, we get something like this:

```
class window_with_scrollbar
            : public window, public scrollbar {
                        // ...
};
```

This allows any object of type window_with_scrollbar to act as both window and scrollbar, but we are required to decide to use only a single type of scrollbar.

If, on the other hand, we consider window_with_scrollbar to be an object of type window that has a scrollbar, we get this definition:

```
class window_with_scrollbar: public window {
            // ...
            scrollbar * sb;
            pu blic:
                        window_with_scrollbar (scrollbar * p, / * ... * /)
                                    : window (/ * ... * /), sb (p)
                        {
                                    // ...
                        }
};
```

Here we can use a solution with three types of scrolls. Passing the rollout itself as a parameter allows the window (window) not to remember the type of its rollout. If you want an object of type window_with_scrollbar to act like a scrollbar, you can add a transform operation:

```
window_with_scrollbar :: operator scrollbar & ()
{
            return * sb;
}
```

12.2.6 Usage relationships

To compose and understand a project, it is often necessary to know which classes and in what way a given class is used. Such class relationships in C ++ are implicitly expressed. A class can only use names that are defined somewhere, but there is no part in a C ++ program that contains a list of all the names used. To obtain such a list, aids (or, if they are not available, careful reading) are required . The ways in which class X can use class Y can be classified as follows :

- X uses the name Y
- X uses Y

- X calls the member function Y

- X reads member Y

- X writes to member Y
- X creates Y
- X allocates an auto or static variable from Y
- X creates Y with new
- X uses Y dimension

We have attributed the use of the object size to its creation, as it requires knowledge of the complete class definition. On the other hand, we singled out the use of the name Y as a separate relation, since, by specifying it in the description of Y * or in the description of an external function, we do not need access to the definition of Y at all:

```
class Y; // Y - class name
Y * p;
extern Y f (const Y &);
```

We have decoupled the creation of Y with new from the case of declaring a variable, since it is possible for a C ++ implementation in which to create Y with new it is not necessary to know the size of Y. This can be essential to limit all dependencies in the project and minimize re-translations after the introduction changes.

The C ++ language does not require the class creator to define exactly which classes and how they will use. One of the reasons for this is that the most important classes depend on so many other classes that to make the program look better, it needs an abbreviated notation of the list of classes used, for example, using the #include command. Another reason is that it is not the responsibility of the programming language to classify these dependencies, and in particular to combine some dependencies . Rather, the goals of the developer, programmer, or tool determine how the use relationship should be viewed. Finally, which dependencies are of greater interest may depend on the specifics of the language implementation.

12.2.7 Relationships within a class

Until now, we have only discussed classes, and although operations were mentioned, apart from the discussion of the steps of the software development process ($$ 11.3.3.2), they were in the background, and objects were practically not mentioned at all. This is easy to understand: in C ++, a class, not a function or an object, is the basic concept of system organization.

A class can hide in itself any implementation specifics, along with "dirty" programming techniques, and sometimes it is forced to do this. At the same time, objects of most classes themselves form a regular structure and are used in such ways that they are easy enough to describe. A class object can be a collection of other nested objects (often called members), many of which, in turn, are pointers or references to other objects. Therefore, a single object can be viewed as the root of the object tree, and all objects included in it as an " object hierarchy " that complements the class hierarchy discussed in $$ 12.2.4. Consider the string class from $$ 7.6 as an example:

```
class String {
        int sz;
        char * p;
        public:
                String (const char * q);
                ~ String ();
                // .. .
};
```

An object of type String can be represented like this:

12.2.7.1 Invariants

The value of the members or objects accessed by class members is called the state of the object (or simply the value of the object). The main thing when building a class is: to bring an object into a fully defined state (initialization), to maintain a fully defined state of the object in the process of performing various operations on it, and at the end of the work, destroy the object without any consequences. The property that makes the state of an object completely specific is called an invariant.

Therefore, the purpose of initialization is to set specific values at which the invariant of the object is executed. For each operation of the class, it is assumed that the invariant must take place before the operation is performed and must be preserved after the operation. At the end of the work, the destructor violates the invariant, destroying the object. For example, the constructor String :: String (const char *) ensures that p points to an array of at least sz elements, with sz having a meaningful value and v [sz-1] == 0. Any string operation must not violate this statement.

It takes a lot of skill in designing a class to make the implementation of the class simple enough to have useful invariants that are easy to define. It is easy to require a class to have an invariant; it is more difficult to propose a useful invariant that is understandable and does not impose hard restrictions on the actions of the class designer or on the efficiency of the implementation. Here "invariant" is understood as a program fragment, by executing which, you can check the state of the object. It is quite possible to give a more rigorous and even mathematical definition of an invariant, and in some situations it may be more appropriate. Here, the invariant is understood as a practical, which means, usually economical, but incomplete check of the object state.

The notion of invariant originated in the pre- and post-condition work of Floyd, Naur, and Hoare, and has been found in all important papers on abstract data types and program verification over the past 20 years. It is also the main subject of debugging in C ++.

Usually, the invariant is not preserved during the operation of the member function. Therefore, functions that can be called at those moments when the invariant does not apply should not be included in the general interface of the class. Such features must be private or secure.

How can an invariant be expressed in a C ++ program? A simple solution is to define a function that checks the invariant and insert calls to this function in general operations. For example:

```
class String {
        int sz;
        int * p;
        public:
                class Range {};
                class Invariant {};
                void check ();
                String (const char * q);
                ~ String ();
                char & operator [] (int i);
                int size () {return sz; }
        // ...
};
```

```
void String :: check ()
{
        if (p == 0 || sz <0 || TOO_LARGE <= sz || p [sz-1])
                throw Invariant;
}

char & String :: operator [] (int i)
{
        check (); // check on input
        if (i <0 || i <sz) throw Range; // acts
        check (); // check on exit
        return v [i];
}
```

This option works great and does not complicate the programmer's life. But for a class as simple as String, checking the invariant will take most of the computation time. Therefore, programmers usually only check the invariant when debugging:

```
inline void String :: check ()
{
        if (! NDEBUG)
                if (p == 0 || sz <0 || TOO_LARGE <= sz || p [sz])
                        throw Invariant;
}
```

We chose the name NDEBUG because it is a macro that is used for similar purposes in the standard C macro assert (). Traditionally, NDEBUG is installed to indicate that there is no debugging. By specifying that check () is a substitution, we ensured that no program is created until the constant NDEBUG is set to a value that denotes debugging. Using a pattern like Assert (), you can specify less regular assertions, for example:

```
template <class T, class X> inline void Assert (T expr, X x)
{
        if (! NDEBUG)
                if (! expr) throw x;
}
```

will throw exception x if expr is false and we haven't disabled NDEBUG checking. Assert () can be used like this:

```
class Bad_f_arg {};

void f (String & s, int i)
{
        Assert (0 <= i && i <s.size (), Bad_f_arg ());
        // ...
}
```

The Assert () pattern mimics the C assert () macro. If i is not in the required range, the Bad_f_arg exception occurs.

It is a trivial matter to check such statements or invariants using a separate constant or a constant from a class . If you need to check invariants using an object, you can define a derived class in which they are checked by operations from a class where there is no check, see Exercise 8 in $$ 13.11.

For classes with more complex operations, the overhead of checks can be significant, so checks can be left only to "catch" hard-to-find errors. It is usually useful to leave at least a few checks, even in a very well-debugged program. Under all conditions, the very fact of defining invariants and using them in debugging provides invaluable help in getting a correct program and, more importantly, makes the concepts represented by classes more regular and well-defined. The point is that when you create invariants, you look at the class from a different point of view and introduce some redundancy into the program. Both increase the likelihood of finding errors, contradictions and oversights. We pointed out in $$ 11.3.3.5 that the two most general forms of transforming a class hierarchy are to split a class into two and to isolate the common part of the two classes into a base class. In both cases, a well thought out invariant can suggest the possibility of such a transformation. If, comparing the invariant with the operation programs, you can find that most of the checks of the invariant are unnecessary, then the class is ripe for splitting. In this case, the subset of operations only has access to a subset of the object's states. Conversely, classes are ripe for merging if they have similar invariants, even with some difference in their implementation.

12.2.7.2 Encapsulation

Note that in C ++, a class, not a single object, is the unit that should be encapsulated (wrapped). For example:

```
class list {
            list * next;
            public:
                        int on (list *);
};

int list :: on (list * p)
{
            list * q = this;
            for (;;) {
                        if (p == q) return 1;
                        if (q == 0) return 0;
                        q = q-> next;
            }
}
```

Here, referring to the private pointer list :: next is legal, since list :: on () has access to any object of the list class to which it has a reference. If this is inconvenient, the situation can be simplified by eliminating the possibility of access through a member function to representations of other objects, for example:

```
int list :: on (list * p)
{
            if (p == this) return 1;
            if (p == 0) return 0;
            return next-> on (p);
}
```

But now iteration turns into recursion, which can greatly slow down program execution, unless the translator can convert recursion back to iteration.

12.2.8 Programmable relationships

A particular programming language cannot directly support any concept of any design method. If the programming language is not able to directly represent the concept of design, you should establish a convenient mapping of the constructs used in the project to the language constructs. For example, a design method might use the concept of delegation, which means that any operation that is not defined on class A must be performed on it using a pointer p to the corresponding member of class B in which it is defined. You cannot express this directly in C ++. However, the implementation of this concept is so much in the spirit of C ++ that it is easy to imagine an implementation program:

```
class A {
            B * p;
            // ...
            void f ();
            void ff ();
};

class B {
            // ...
            void f ();
            void g ();
            void h ();
};
```

The fact that B is delegating A using the pointer A :: p is expressed in the following notation:

```
class A {
            B * p; // delegate with p
            // ...
            void f ();
            void ff ();
            void g () {p-> g (); } // delegation q ()
            void h () {p-> h (); } // delegation h ()
};
```

For the programmer it is quite obvious what is happening here, but the principle of one-to-one correspondence is clearly violated here . Such "programmable" relationships are difficult to express in programming languages, and therefore it is difficult to apply various auxiliary means to them. For example, such a facility might not distinguish "delegation" from B to A with A :: p from any other use of B *.

Still, wherever possible, one should strive for a one- to -one correspondence between the concepts of the project and the concepts of the programming language. It provides a degree of simplicity and ensures that the project is adequately displayed in the program, which simplifies the work of the programmer and tools. Type conversion operations are a mechanism by which a class of programmable relations can be represented in the language, namely: the conversion operation X :: operator Y () ensures that wherever Y is allowed, X can also be used. The same relation is defined by the constructor Y :: Y (X). Note that a type conversion operation (like a constructor) creates a new object rather than changes the type of an existing object. To assign a conversion operation to a function Y is simply to require an implicit application of a function that returns Y. Since implicit conversions and operations defined by constructors can get in trouble, it is useful to analyze them separately in the project.

It is important to ensure that the application graph of type conversion operations does not contain loops. If they are, an ambiguous situation arises in which the types participating in the loops become incompatible in combination. For example:

```
class Big_int {
        // ...
        friend Big_int operator + (Big_int, Big_int);
        // ...
        operator Rational ();
        // ...
};

class Rational {
        // ...
        friend Rational operator + (Rational, Rational);
        // ...
        operator Big_int ();
};
```

The Rational and Big_int types don't interact as smoothly as one might think:

```
void f (Rational r, Big_int i)
{
        // ...
        g (r + i); // error, ambiguity:
```

```
                                                       // operator + (r,
Rational (i)) or
                                                        // operator +
(Big_int (r), i)
        g (r, Rational (i)); // explicit ambiguity resolution
        g (Big_int (r), i); // Another
}
```

Such "reciprocal" transformations could have been avoided by making some of them explicit. For example, a conversion from Big_int to a Rational type could be specified explicitly using the make_Rational () function instead of a conversion operation, then addition in the above example would be resolved as g (BIg_int (r), i). If you cannot avoid "mutual" type conversion operations, then you need to overcome the resulting collisions either by using explicit conversions (as shown), or by defining several different versions of the binary operation (in our case +).

12.3 Components

There are no constructs in C ++ that can express the concept of a component directly in a program , i.e. many related classes. The main reason for this is that many classes (possibly with corresponding global functions, etc.) can be combined into a component in a variety of ways. The absence of the explicit representation of the concepts in the language complicates the border between the information (names) , used within the component, and the information (names) transmitted from the component users. Ideally, a component is defined by many interfaces used to implement it, plus many user-supplied interfaces, and everything else is considered "implementation specific" and should be hidden from the rest of the system. This may actually be the developer's view of the component. The programmer must come to terms with the fact that C ++ does not provide a general concept of a component namespace, so it has to be "modeled" using the concepts of classes and translation units , ie. the same tools that C ++ has to limit the scope of non-local names.

Consider two classes that must share the f () function and the v variable. It is easiest to describe f and v as global names. However, any experienced programmer knows that this "clogging" of a namespace can lead to trouble in the end : someone can inadvertently use the names f or v for other purposes or deliberately refer to f or v, directly using the "implementation specifics" and thus bypassing the explicit interface of the component. Three solutions are possible here:

l] Give "unusual" names to objects and functions that are not intended for the user.

2] Objects or functions not intended for the user should be described in one of the program files as static.

3] Place objects and functions that are not intended for the user into a class whose definition is closed to users.

The first solution is primitive and rather inconvenient for the creator of the program, but it works:

```
// don't use compX implementation specifics,
// unless you are a compX developer:
extern void compX_f (T2 *, const char *);
extern T3 compX_v;
// ...
```

Names such as compX_f and compX_v are unlikely to lead to a collision, and to the argument that a user can be an attacker and use these names directly, you can answer that the user can be an attacker anyway , and that the language protection mechanisms protect against accident , not from malice. The advantage of this solution is that it is always applicable and well known. At the same time, it is ugly, unreliable, and complicates text entry. The second solution is more reliable, but less versatile:

```
// specifics of compX implementation:
static void compX_f (T2 * a1, const char * a2) {/ * ... * /}
static T3 compX_v;
// ...
```

It is difficult to guarantee that the information used in the classes of one component will be available in only one translation unit, since the operations working on this information must be available everywhere. This solution can also lead to huge translation units, and some C ++ debuggers do not provide access to the names of static functions and variables. At the same time, this solution is reliable and often optimal for small components. The third solution can be seen as a formalization and generalization of the first two:

```
class compX_details {// specifics of compX implementation
```

```
          public:
                    static void f (T2 *, const char *);
                    static T3 v;
                    // ...
   };
```

The compX_details description will only be used by the creator of the class, others should not include it in their programs.

Of course, a component can have many classes that are not intended for general use. If their names are also designed only for local use, then they can also be "hidden" inside classes containing implementation specifics:

```
   class compX_details {// implementation specifics compX.
          public:
                    // ...
                    class widget {
                              // ...
                    };
                    // ...
   };
```

We point out that nesting creates a barrier to the use of the widget in other parts of the program. Typically, classes that represent clear concepts are considered prime candidates for reuse , and therefore form part of a component's interface rather than an implementation detail. In other words, while nested objects used to represent some object of a class are best considered hidden implementation details to maintain an appropriate level of abstraction , the classes defining such nested objects are best not hidden if they have sufficient generality . So, in the following example, hiding is perhaps unnecessary:

```
   class Car {
          class Wheel {
                    // ...
          };
          Wheel flw, frw, rlw, rrw;
          // ...
   };
```

In many situations, to maintain the level of abstraction of the concept of a car (Car), you should hide the real wheels (class Whe el), because when you work with a car, you cannot use wheels independently of it . On the other hand, the Wheel class itself is quite suitable for widespread use, so it's better to take its definition out of the Car class:

```
class Wheel {
        // ...
};
class Car {
        Wheel flw, frw, rlw, rrw;
        // ...
};
```

Should you use nesting? The answer to this question depends on the goals of the project and the generality of the concepts used. Both nesting and its absence can be quite acceptable solutions for a given project. But because nesting prevents the common namespace from clogging up , it is recommended to use nesting in the rulebook below , unless there is a reason not to.

Note that header files provide a powerful means for different views of a component to different users, and they also allow you to remove implementation-specific classes from the view of a component to the user .

Another means of building a component and presenting it to the user is a hierarchy. Then the base class is used as a repository of common data and functions. In this way, the problem associated with global data and functions designed to implement common requests of the classes of this component is eliminated . On the other hand, with such a solution, the component classes become too closely related to each other, and the user becomes dependent on all the base classes of those components that he really needs. There is also a tendency here for members representing "useful" functions and data to "float" to the base class, so that if the class hierarchy is too large, problems with global data and functions will manifest themselves within this hierarchy. This will most likely happen for a single-root hierarchy, and virtual base classes ($$ 6.5.4) can be used to combat this phenomenon . Sometimes it is better to choose a hierarchy to represent the component, and sometimes not. As always, the developer has to make a choice.

12.4 Interfaces and implementations

An ideal interface should

- present a complete and consistent set of concepts to the user,
- be consistent across all parts of a component,
- hide the specifics of the implementation from the user,
- allow multiple implementations,
- have a static type system,
- defined using types from the application scope,
- to depend on other interfaces only partially and in a very specific way.

Having noted the need for consistency for all classes that form a component's interface to the rest of the world, we can simplify the interface issue by looking at just one class, for example:

```
class X { // example of bad interface definition
        Y a;
        Z b;
        public:
                void f (const char * ...);
                void g (int [], int);
                void set_a (Y &);
                Y & get_a ();
};
```

There are a number of potential problems with this interface:

- -Types Y and Z are used so that the definitions of Y and Z must be known at the time of broadcast.
- The X :: f function can have an arbitrary number of parameters of unknown type (perhaps they are somehow controlled by the "format string" that is passed as the first parameter).
- The X :: g function has a parameter of type int []. This may be fine, but it usually indicates that the definition is too low an abstraction level. An array of integers is not a sufficient definition, since it is not known how many elements it can consist of.
- The set_a () and get_a () functions seem to expose the representation of objects of class X, allowing direct access to X :: a.

Here, member functions form an interface at a very low level of abstraction. Typically, classes with an interface of this level are specific to the implementation of a large component, if they can be related to anything at all. Ideally, a function parameter from an interface should be accompanied by enough information to understand it. You can formulate the following rule: you must be able to transfer service requests to a remote server over a narrow channel.

The C ++ language exposes the class representation as part of the interface. This view can be hidden (using private or protected), but it must be accessible to the translator so that it can place automatic (local) variables, substitute the function body , etc. The negative consequence of this is that the use of class types in the class view can lead to unwanted dependencies. Whether using members like Y and Z will lead to problems depends on what types Y and Z really are. If they are simple enough types, like complex or String, then their use will be fine in most cases. Such types can be considered persistent, and the need to include their class definitions would be a perfectly acceptable load on the translator. If Y and Z are themselves classes of the interface of a large component (for example, such as a graphics system or a bank account support system), then a direct dependence on them may be considered unreasonable. In such cases, it is preferable to use a pointer or reference member :

```
class X {
            Y * a;
            Z & b;
            // ...
};
```

In this way, the definition of X is separated from the definitions of Y and Z, i.e. now the definition of X only depends on the names Y and Z. The implementation of X will of course still depend on the definitions of Y and Z, but this will no longer adversely affect users X.

The above illustrates an important point: An interface that hides a significant amount of information (which a useful interface should do) must have significantly fewer dependencies than an implementation that hides them. For example, the definition of class X can be translated without access to the definitions of Y and Z. However, in the definitions of member functions of class X that work with object references Y and Z, access to the definitions of Y and Z is required. When analyzing dependencies, you should consider separately the dependencies in the interface and in the implementation. Ideally, for both kinds of dependencies, the system's dependency graph should be a directed non-cyclic graph, making it easier to understand and test the system. However, this goal is more important and more often achievable for implementations than for interfaces.

Note that the class defines three interfaces:

```
class X {
        private:
                // only available to members and friends
        protected:
                // only available to members and friends, and
                // for members and friends of derived classes
        public:
                // publicly available
};
```

Members should form the most restricted interface possible. In other words, a member should be described as private unless there is a reason for wider access to it; if there are any, then the member should be declared protected unless there is an additional reason to declare it public. In most cases, it is bad practice to set all member data as public. The functions and classes that make up a common interface should be designed so that the presentation of the class matches its role in the project as a means of representing concepts. As a reminder, friends are part of the overall interface.

Note that abstract classes can be used to represent a higher-level concealment concept ($$ 1.4.6, $$ 6.3, $$ 13.3).

12.5 Code of practice

We have touched on many topics in this chapter, but have generally avoided giving strong and specific advice on the issues under consideration . This is consistent with my conviction that there is no "one right solution." The principles and techniques should be applied in the manner most appropriate for the task at hand. Taste, experience and intelligence are required here . However, a set of rules can be proposed that a developer can use as guidelines until they gain enough experience to come up with better ones. This set of rules is given below.

It can serve as a starting point in the development of project guidelines for a specific task, or it can be used by an organization as a checklist. Let me emphasize again that these rules are not universal and cannot replace reflection.

- Target the user with data abstraction and object-oriented programming.
  - Gradually switch to new methods, do not rush.
  - Use C ++ features and object-oriented programming techniques only as needed.
- Match the style of the project and the program.
- Concentrate on component design.
- Use classes to represent concepts.

- Use general inheritance to represent is relationship.
- Use affiliation to represent the has relationship.
- Ensure that the relationships of use are clear, do not cycle, and that the number is minimal.
- Actively seek commonality between application domain and implementation concepts, and represent the resulting more general concepts as base classes.

• Define the interface to reveal the minimum amount of information required:

- Use private data and member functions wherever possible.
- Use public or protected descriptions to distinguish requests from a derived class developer from requests from regular users.
- Minimize dependencies of one interface on others.
- Maintain strong typing of interfaces.
- Define interfaces in terms of application - scoped types.

Additional rules can be found $$ 11.5.

CHAPTER 13. DESIGNING LIBRARIES

A library project is a language project,
(folklore by Bell Laboratories)

... and vice versa.
- A. Koenig

This chapter contains a description of various techniques that have proven useful in creating libraries for the C ++ language. In particular, it covers concrete types, abstract types, node classes, management classes, and interface classes. It also discusses the concepts of broad interface and application scope structure, the use of dynamic type information, and memory management techniques. Attention is focused on what properties library classes should have, and not on the specifics of the language tools that are used to implement such classes, and not on certain useful functions that the library should provide.

13.1 Introduction

Developing a general-purpose library is a much more difficult task than creating a regular program. A program is a solution to a specific problem for a specific area of an application, whereas a library should provide a solution for a variety of problems related to many areas of an application. In a typical program, strong assumptions about its environment are allowed, while a good library can be used successfully in a variety of environments created by many different programs. The more general and useful a library turns out to be, the more environments it will be tested, and the more stringent the requirements for its correctness, flexibility, efficiency, extensibility, portability, consistency, simplicity, completeness, ease of use, etc. Yet the library cannot give you everything, so some compromise is needed. A library can be thought of as a special, interesting variation on what we called a component in the previous chapter. Every piece of advice about designing and maintaining a component becomes critical to libraries, and conversely, many methods of building libraries find their way into the design of various components.

It would be too presumptuous to tell you how libraries should be designed. Several different methods have proven successful in the past, and the subject remains a field of intense discussion and experimentation. It only discusses some important aspects of this task and suggests some techniques that have proven useful in building libraries. Keep in mind that libraries are designed for completely different areas of programming, so you don't have to rely on one method to be the most appropriate for all libraries. Indeed, there is no reason to believe that the techniques that have proven useful in implementing parallel programming tools for the kernel of a multiprocessor operating system seem to be the most appropriate for creating a library for solving scientific problems or a library that provides a graphical interface.

The concept of a C ++ class can be used in many different ways, so the variety of programming styles can lead to confusion. A good library for keeping this mess to a minimum provides a consistent programming style, or at least a few. This approach makes the library more "predictable" and therefore makes it easier and faster to learn and use it correctly. The following describes the five "archetypal" classes, and discusses their inherent strengths and weaknesses: concrete types ($$ 13.2), abstract types ($$ 13.3), node classes ($$ 13.4), interface classes ($$ 13.8), management classes ($$ 13.9). All of these kinds of classes are conceptual, not language constructs. Each concept is embodied using the main construct - a class. Ideally, you should have a minimal set of simple and orthogonal class views from which any useful and reasonably defined class can be built. We have not achieved the ideal and, perhaps, unattainable at all. It is important to understand that any of the listed types of classes play a role in the design of the library and, if you count on general use, none of them is inherently better than the others.

This chapter introduces the concept of a broad interface ($$ 13.6) to highlight some general case of all these kinds of classes. It defines the concept of an application scope framework ($$ 13.7).

Here, first of all, classes are considered that belong strictly to one of the listed types, although, of course, classes of a hybrid type are also used. But the use of a hybrid species class should be the result of a deliberate decision made in evaluating the pros and cons of different species, and not the result of a pernicious tendency to shy away from choosing a kind of class (too often "postponing the choice" means simply unwillingness to think). It is best for inexperienced library developers to stay away from hybrid-looking classes. They may be advised to follow the programming style of the existing library that has the capabilities required for the library being designed. Only a sophisticated programmer can dare to create a general-purpose library, and each library creator will subsequently be "condemned" to many years of using, documenting, and maintaining their own creation.

C ++ uses static types. However, sometimes it becomes necessary to obtain dynamic information about types in addition to the capabilities directly provided by virtual functions. How to do this is described in $$ 13.5. Finally, any non-trivial library is faced with the task of memory management. Techniques for solving it are discussed in $$ 13.10. Naturally, this chapter cannot cover all the methods that have proven useful in creating the library. Therefore, you can refer to other places in the book, where the following topics are covered: handling errors and resilience to errors ($$ 9.8), using function objects and callbacks ($$ 10.4.2 and $$ 9.4.3), using type templates to build classes ($$ 8.4).

Many of the topics in this chapter are related to container classes (such as arrays and lists). Of course, such container classes are type templates (as discussed in $$ 1 and 4.3 $$ 8). But here, to simplify the presentation, the examples use classes that contain pointers to objects of the class type. To get a real program, you need to use type templates, as shown in Chapter 8.

13.2 Concrete types

Classes such as vector ($$ 1.4), Slist ($$ 8.3), date ($$ 5.2.2), and complex ($$ 7.3) are specific in the sense that each of them represents a fairly simple concept and has the necessary set of operations. There is a one-to-one correspondence between the interface of a class and its implementation. None of them were (originally) intended as a base for derived classes. Usually in a class hierarchy, specific types stand out. Each particular type can be understood in isolation, out of relation to other classes. If the implementation of a particular type is successful, then the programs working with it are comparable in size and speed to hand-made programs that use some special version of the general concept. Further, if there is a significant change in implementation, the interface is usually modified to reflect these changes. The interface, in its essence, is obliged to show what changes were significant in this context. The higher-level interface leaves more freedom to change the implementation, but may degrade the performance of the program. Moreover, a good implementation depends only on the minimum number of really significant classes. Any of these classes can be used without the translation or runtime overhead caused by adapting to other "similar" program classes. To summarize, you can specify the conditions that a particular type must satisfy:

1] fully reflect this concept and the method of its implementation;

2] with the help of substitutions and operations that fully use the useful properties of the concept and its implementation, to provide efficiency in terms of speed and memory, comparable to "manual programs";

3] have minimal dependence on other classes;

4] be understandable and useful even in isolation.

All of this should lead to a close relationship between the user and the program that implements the particular type. If there are changes in the implementation, the user program will have to be re-translated, since it probably contains function calls implemented by substitution, as well as local variables of a particular type.

For some areas of the application, concrete types are provided by basic types not directly represented in C ++, for example: complex numbers, vectors, lists, matrices, dates, associative arrays, character strings, and symbols from a non-English alphabet. In a world of concrete concepts, there really is no such thing as a list. Instead, there are many list classes, each specialized in representing a version of the list. There are a dozen list classes, including: a one-way link list; bi-directional list; a one-way link list in which the link field does not belong to the object; a two-way link list in which link fields do not belong to the object; a one-way link list for which you can simply and efficiently determine whether a given object is included in it; a bi-directional list for which you can easily and efficiently determine whether a given object is included in it, and so on.

The name "concrete type" (CDT - concrete data type) was chosen in contrast to the term "abstract data type" (ADT - abstract data type). The relationship between CDT and ADT is discussed in $$ 13.3.

It is essential that concrete types are not intended to explicitly express some generality. Thus, the types slist and vector can be used as an alternative implementation of the concept of a set, but this is not explicitly reflected in the language. Therefore, if a programmer wants to work with a set, uses specific types and does not have a class definition set, then he must choose between the types slist and vector. Then the program is written in terms of the selected class, say, slist, and if later they choose to use another class, the program will have to be rewritten.

This potential inconvenience is compensated by the presence of all "natural" operations for this class, such as indexing for an array and deleting an element for a list. These operations are presented in the best possible way, without "unnatural" operations such as indexing a list or deleting an array, which could cause confusion. Let's give an example:

```
void my (slist & sl)
{
        for (T * p = sl.first (); p; p = sl.next ())
        {
                // my code
        }
        // ...
}

void your (vector & v)
{
        for (int i = 0; i <v.size (); i ++)
        {
                // your code
        }
        // ...
}
```

The existence of such "natural" operations for the chosen method of implementation ensures the efficiency of the program and greatly facilitates its writing. In addition, although the implementation of a call by substitution is usually only possible for simple operations such as indexing an array or getting the next item in a list, it has a significant effect on the speed of program execution. The catch here is that program fragments that use inherently equivalent operations, such as the two loops above, may look different from each other, and program fragments that use different concrete types for equivalent operations cannot replace each other. Usually, it is generally impossible to combine similar program fragments into one.

A user accessing a function must specify exactly the type of object the function is working with, for example:

```
void user ()
{
        slist sl;
        vector v (100);
        my (sl);
        your (v);
        my (v); // error: type mismatch
        your (sl); // error: type mismatch
}
```

To compensate for the rigidity of this requirement, the developer of a useful function must provide several versions of it so that the user has a choice:

```
void my (slist &);
void my (vector &);
void your (slist &);
void your (vector &);
void user ()
{
        slist sl;
        vector v (100);
        my (sl);
        your (v);
        my (v); // ok now: call my (vector &)
        your (sl); // ok now: call your (slist &)
}
```

Since the body of a function essentially depends on the type of its parameter, you need to write each version of the my () and your () functions independently of each other, which can be troublesome.

All things considered, a concrete type can be said to be like built-in types. The positive side of this is the close relationship between the user of a type and its creator, as well as between users who create objects of a given type, and users who write functions that work with these objects. To use a particular type correctly, the user must understand it in detail. Usually, there are no universal properties that all specific types of a library would have, and that would allow the user, relying on these properties, not to waste energy on studying individual classes. This is the price to pay for the compactness of the program and the efficiency of its execution. Sometimes this is a reasonable price, sometimes not. In addition, it is possible that a particular concrete class is easier to understand and use than a more general (abstract) class. This is the case with classes that represent well-known data types such as arrays or lists.

However, we point out that ideally, implementation details should be hidden as much as possible until it degrades the performance of the program. Substitution functions are of great help here. Making member variables public by declaring them public, or working directly with them using functions that set and retrieve the values of those variables, almost always leads to bad results. Concrete types should still be real types, not just a software heap with a few features added for convenience.

13.3 Abstract types

The easiest way to loosen the connection between the user of a class and its creator, and between the programs in which objects are created and the programs in which they are used, is to introduce the concept of abstract base classes. These classes represent an interface with many implementations of the same concept. Consider a set class containing a set of objects of type T:

```
class set {
        public:
                virtual void insert (T *) = 0;
                virtual void remove (T *) = 0;
                virtual int is_member (T *) = 0;
                virtual T * first () = 0;
                virtual T * next () = 0;
                virtual ~ set () {}
};
```

This class defines an interface with an arbitrary set, based on the built-in concept of iterating over the elements of a set. Typical here are the absence of a constructor and the presence of a virtual destructor, see also $$ 6.7. Let's consider an example:

```
class slist_set: public set, private slist {
        slink * current_elem;
        public:
                void insert (T *);
```

```
                    void remove (T *);
                    int is_member (T *);
                    virtual T * first ();
                    virtual T * next ();
                    slist_set (): slist (), current_elem (0) {}
};

class vector_set: public set, private vector {
          int current_index;
          public:
                    void insert (T *);
                    void remove (T *);
                    int is_member (T *);
                    T * first () {current_index = 0; return next (); }
                    T * next ();
                    vector_set (int initial_size)
                              : array (initial_size), current_index (0) {}
};
```

A concrete type implementation is used as a private base class, not a member of the class. This is both for convenience of notation and because some concrete types may have a protected interface in order to provide more direct access to their members from derived classes. In addition, in a similar way, the implementation may use some classes that have virtual functions and are not concrete types. Only through the formation of derived classes is it possible to gracefully redefine (suppress) the virtual function of the implementation class in the new class. An interface is defined by an abstract class.

Now the user can write their functions from $$ 13.2 like this:

```
void my (set & s)
{
          for (T * p = s.first (); p; p = s.next ())
          {
                    // my code
          }
          // ...
}
```

```
void your (set & s)
{
        for (T * p = s.first (); p; p = s.next ())
        {
                // your code
        }
        // ...
}
```

The similarity between the two functions became apparent, and now it is sufficient to have only one version for each of the my () or your () functions, since both versions use the interface defined by the set class to communicate with slist_set and vector_set:

```
void user ()
{
        slist_set sl;
        vector_set v (100);
        my (sl);
        your (v);
        my (v);
        your (sl);
}
```

Moreover, the creators of the my () and your () functions are not required to know the descriptions of the slist_set and vector_set classes, and the my () and your () functions are in no way dependent on these descriptions. They do not need to be re-translated or changed in any way, neither if the slist_set or vector_set classes have changed, nor even if a new implementation of these classes is proposed. The changes are reflected only in functions that directly use these classes, for example vector_set. In particular, you can take advantage of the traditional use of header files and include a set.h definition file in programs with my () or your () functions, rather than slist_set.h or vector_set.h files.

Ordinarily, the operations of an abstract class are defined as pure virtual functions, and such a class has no data members (apart from a hidden pointer to a virtual function table). This is because adding a non-virtual function or data member would require certain class assumptions that would constrain possible implementations. The approach to abstract classes presented here is similar in spirit to traditional methods based on strict separation of the interface and its implementations. An abstract type serves as an interface, and concrete types represent its implementations.

This separation of the interface and its implementations implies that operations that are "natural" for any one implementation, but not general enough to enter the interface, are not available. For example, because there is no ordering in an arbitrary set, an indexing operation cannot be included in the set interface, even if an array is used to implement the particular set. This leads to a deterioration in program performance due to the lack of manual optimization. Further, it becomes, as a rule, impossible to implement functions by substitution (except for some specific situations when the real type is known to the translator), therefore, all useful interface operations are specified as calls of virtual functions. As with concrete types, the fees for abstract types are sometimes acceptable, sometimes too high.

To summarize, let's list what purposes an abstract type should serve :

1] to define some concept in such a way that several implementations for it can coexist in the program;

2] using virtual functions to ensure a sufficiently high degree of compactness and efficiency of program execution;

3] to minimize the dependence of any implementation on other classes;

4] represent a meaningful concept in itself.

This is not to say that abstract types are better than concrete types, they are just other types. Which one to prefer is usually a difficult and important question for the user. The library creator can shy away from answering it and provide options with both types, thereby leaving the choice to the user. But here it is important to clearly understand what kind of class you are dealing with. An attempt to limit the generality of an abstract type usually fails so that the speed of programs working with it approaches the speed of programs designed for a specific type. In this case, you cannot use interchangeable implementations without large re-translation of the program after the changes are made. An attempt to give "generality" in concrete types is equally unsuccessful, so that they could approach abstract types in terms of the power of concepts. This reduces the efficiency and applicability of simple classes. Classes of these two kinds can coexist, and they must coexist peacefully in the program. A concrete class embodies the implementation of an abstract type and should not be confused with an abstract class.

Note that neither concrete nor abstract types are created initially as base classes for further constructing derived classes. Derivatives to abstract class types are more needed to define implementations than to develop the very concept of an interface. Any concrete or abstract type is designed to clearly and efficiently represent a separate concept in a program. Classes that do this are rarely good candidates for creating new but related classes from them. Indeed, attempts to construct derived, "more advanced" classes based on concrete or abstract types such as strings, complex numbers, lists, or associative arrays usually lead to cumbersome constructions. In general, these classes should be used as members or private base classes so that they can be effectively used without causing confusion or conflict in the interfaces and implementations of these and new classes.

When creating a concrete or abstract type, the emphasis should be on offering a simple interface that implements a well thought out concept. Attempts to expand the scope of a class by loading its description with all sorts of "useful" properties only lead to confusion and inefficiency. This also ends a vain effort to guarantee class reuse, when each member function is declared virtual without thinking about why and how these functions will be overridden.

Why didn't we define the slist and vector classes as directly derived from the set class, thus dispensing with the slist_set and vector_set classes? In other words, why do we need concrete types when abstract types are already defined? Three answers can be suggested:

1] Efficiency: types such as vector or slist should be created without the overhead of moving implementations away from interfaces (the concept of an abstract type requires separation of interface and implementation).

2] Multiple interface: often different concepts are best implemented as derived from the same class.

3] Reusability: We need a mechanism that allows us to accommodate types designed "elsewhere" for our library.

Of course, all of these answers are related. As an example [2], consider the notion of an iteration generator. It is required to define an iteration generator (hereinafter an iterator) for any type so that it can be used to generate a sequence of objects of this type. Naturally, for this you need to use the already mentioned slist class. However, you cannot simply define a generic iterator over a slist, or even over a set, since a generic iterator must also iterate over more complex non-set objects, such as input streams or functions, which, when called again, yield the next iteration value. This means that we need both a set and an iterator, and at the same time, it is undesirable to duplicate specific types, which are obvious implementations of various kinds of sets and iterators. You can graphically represent your desired class structure like this:

Here, the set and iter classes provide interfaces, while slist and stream are private classes and represent implementations. Obviously, you cannot reverse this class hierarchy and, by providing common interfaces, build derived concrete types from abstract classes. In such a hierarchy, every useful operation on every useful abstract concept must be represented in a common abstract base class. See $$ 13.6 for further discussion of this topic.

Here's an example of a simple abstract type that is an iterator of objects of type T:

```
class iter {
        virtual T * first () = 0;
        virtual T * next () = 0;
        virtual ~ iter () {}
};

class slis t_iter: public iter, private slist {
        slink * current_elem;
        public:
                T * first ();
                T * next ();
                slist_iter (): current_elem (0) {}
};

class input_iter: public iter {
        isstream & is;
        public:
                T * first ();
                T * next ();
                input_iter (istream & r): is (r) {}
};
```
You can use the types we have defined in this way:
```
void user (const iter & it)
{
        for (T * p = it.first (); p; p = it.next ()) {
                // ...
        }
}

void caller ()
{
        slist_iter sli;
        input_iter ii (cin);
        // fill in sli
        user (sli);
        user (ii);
}
```
We have used a concrete type to implement an abstract type, but you can use it independently of abstract types, or just introduce such types to improve the efficiency of the program, see also $$ 13.5. In addition, one concrete type can be used to implement multiple abstract types.

Section $$ 13.9 describes a more flexible iterator. For it, the dependency on the implementation that supplies the objects to be iterated is determined at the time of initialization and can change during program execution.

13.4 Node classes

In reality, the class hierarchy is built from a very different concept of derived classes than the interface-implementation concept that was used for abstract types. The class is regarded as the foundation of the building. But even if an abstract class is at the base, it allows some representation in the program and itself provides some useful functions for derived classes. Examples of node classes are rectangle ($$ 6.4.2) and satellite ($$ 6.5.1).

Typically, in a hierarchy, a class represents some general concept, and derived classes represent specific variants of that concept. The node class is an integral part of the class hierarchy. It uses the service provided by the base classes, provides a specific service itself, and provides virtual functions and / or a secure interface to allow further granularity of its operations in derived classes.

A typical node class not only provides an implementation of the interface specified by its base class (as an implementation class does with respect to an abstract type), but also extends the interface itself by adding new functionality. As an example, consider the dialog_box class, which represents a window of some kind on the screen. In this window, questions appear to the user and in it he asks his answer by pressing a key or "mouse":

```
class dialog_box: public window {
            // ...
            public:
                        dialog_box (const char * ...); // null terminated
list
                                                                   // key labels
                        // ...
                        vi rtual int ask ();
};
```

Here an important role is played by the ask () function and the constructor, with which the programmer specifies the keys to be used and sets their numerical values. The ask () function displays a window and returns the number of the key pressed in response. You can imagine this use case:

```
void user ()
{
            for (;;) {
                        // some commands
                        dialog_box cont ("continue",
                                    "try again",
                                    "abort",
                                    (char *) 0);
                        switch (cont.ask ()) {
                                    case 0: return;
                                    case 1: break;
```

```
                    case 2: abort ();
                }
            }
    }
```
Let's pay attention to the use of the constructor. A constructor is usually needed for a node class and is often a non-trivial constructor. This distinguishes node classes from abstract classes, which rarely require constructors.

The user of the dialog_box class (not just the creator of the class) relies on the service provided by its base classes. This example assumes that there is some standard placement of the new window on the screen. If the user wants to control the placement of the window, the dialog_box base class window (window) should provide such an opportunity, for example:

```
    dialog_box cont ("continue", "try again", "abort", (char *) 0);
    cont.move (some_point);
```
Here, the window motion function move () relies on certain base class functions.

The dialog_box class itself is a good candidate for building derived classes. For example, it is quite reasonable to have a window in which, in addition to pressing a key or typing with a mouse, you can specify a character string (say, a file name). Such a dbox_w_str window is built as a class derived from the simple dialog_box window:

```
    class dbox_w_str: public dialog_box {
            // ...
            public:
                dbox_w_str (
                    const char * sl, // user query string
                    const char * ... // list of key symbols
                );
                int ask ();
                virtual char * get_string ();
                // ...
    };
```
The get_string () function is the operation by which the programmer gets a user-specified string. The ask () function from the dbox_w_str class

guarantees that the line is entered correctly, and if the user did not enter the line, then the corresponding value (0) is returned to the program.

```
void user2 ()
{
        // ...
        dbox_w_str file_name ("please enter file name",
                "done",
                (char *) 0);
        file_name.ask ();
        char * p = file_name.get_string ();
        if (p) {
                // use the filename
        }
        else {
                // file name not specified
        }
        //
}
```

To summarize, the node class should:

1] rely on their base classes both to implement them and to present the service to users of these classes;

2] present a more complete interface (ie, an interface with more member functions) to users than base classes;

3] base primarily (but not exclusively) their common interface on virtual functions;

4] depend on all of its (direct and indirect) base classes;

5] make sense only in the context of their base classes;

6] serve as a base class for building derived classes;

7] to be embodied in the object.

Not all, but many, node classes will satisfy conditions 1, 2, 6, and 7. A class that does not satisfy condition 6 is like a specific type and can be named a specific node class. A class that does not satisfy Condition 7 is like an abstract type and can be called an abstract node class. Many node classes have protected members to provide a less constrained interface for derived classes.

Let us point out the corollary of condition 4: in order to broadcast his program, the user of the node class must include descriptions of all its direct and indirect base classes, as well as descriptions of all those classes on which, in turn, base classes depend. Here again, the node class represents a contrast to the abstract type. The user of an abstract type does not depend on all the classes used to implement the type and should not include their descriptions for translating his program.

13.5 Dynamic type information

It is sometimes useful to know the true type of an object before using it in any operation. Consider the my (set &) function from $$ 13.3.

```
void my_set (set & s)
{
        for (T * p = s.fir st (); p; p = s.next ()) {
                // my code
        }
        // ...
}
```

It is good in the general case, but imagine - it became known that many parameters of the set are objects of the slist type. It is also possible that an algorithm for enumerating elements has also become known, which is much more efficient for lists than for arbitrary sets. As a result of the experiment, it was possible to find out that this particular search is a bottleneck in the system. Then, of course, it makes sense to take into account the slist option separately in the program. Assuming the possibility of determining the true type of the parameter defining the set, the function my (set &) can be written as follows:

```
void my (set & s)
{
        if (ref_type_info (s) == static_type_info (slist_set)) {
                                                        // compare two
views of the type
                                                        // s of type slist
                slist & sl = (slist &) s;
                for (T * p = sl.first (); p; p = sl.next ()) {
                        // efficient option based on list
                }
            }
            else {
                for (T * p = s.first (); p; p = s.next ()) {
                    // the usual option for an arbitrary set
                }
            }
            // ...
    }
```

Once a particular slist type was known, certain list operations became available, and even basic substitution operations became possible.

The above version of the function works great, since slist is a concrete class, and it really makes sense to separately parse the option when the parameter is slist_set. Let us now consider such a situation when it is desirable to separately parse the variant both for the class and for all its derived classes. Let's say we have a dialog_box class from $$ 13.4 and want to know if it is a dbox_w_str class. Since there can be many derived classes from dbox_w_str, a simple match against it is not a good solution. Indeed, derived classes can represent a wide variety of string queries. For example, one dbox_w_str-derived class might prompt the user for string choices, another might provide directory searches, and so on. This means that you need to check for coincidence with all classes derived from dbox_w_str. This is as typical of node classes as checking for a well-defined type is typical of abstract classes implemented by concrete types.

```
    void f (dialog_box & db)
    {
            dbox_w_str * dbws = ptr_cast (dbox_w_str, & db);
            if (dbws) {// dbox_w_str
                    // dbox_ w_str :: get_string () can be used here
            }
            else {
                     // `` normal '' dialog_box
            }
            // ...
    }
```

Here the "operation" of casting ptr_cast () casts its second parameter (pointer) to its first parameter (type), provided that the pointer is set to an object of type that matches the given one (or is a class derived from the given type). To check the dialog_box type, a pointer is used so that after casting it can be compared to zero.

Possibly an alternative solution using the dialog_box link:

```
void g (dialog_b ox & db)
{
        try {
                dbox_w_str & dbws = ref_cast (dialog_box, db);
                // dbox_w_str :: get_string () can be used here
        }
        catch (Bad_cast) {
                // `` normal " dialog_box
        }
        // ...
}
```

Since there is no acceptable representation of a null reference to compare to, a cast error exception is used (that is, the case where the type is not dbox_w_str). Sometimes it is better to avoid comparison with the cast result.

The difference between ref_cast () and ptr_cast () serves as a good illustration of the difference between references and pointers: a reference necessarily refers to an object, whereas a pointer may or may not be referenced, so a check is often needed for a pointer.

13.5.1 Type information

In C ++, there is no other standard means of obtaining dynamic type information other than calls to virtual functions. Although there have been several proposals to extend C ++ in this direction.

It is quite easy to model such a facility, and most large libraries have dynamic type query capabilities. A solution is proposed here that has the useful property that the amount of information about the type can be arbitrarily expanded. It can be implemented using virtual function calls, and it can be included in extended C ++ implementations.

A reasonably convenient interface with any tool that provides type information can be specified using the following operations:

```
typeid static_type_info (type) // get the typeid for the type name
typeid ptr_type_info (pointer) // Get the typeid for the pointer
typeid ref_type_info (reference) // get the typeid for the link
pointer ptr_cast (type, pointer) // pointer conversion
reference ref_cast (type, reference) // link conversion
```

The user of the class can get by with these operations, and the creator of the class must provide certain "fixtures" in the class descriptions to match the operations with the library implementation.

Most users who need dynamic type identification at all can limit themselves to casting operations ptr_cast () and ref_cast (). This removes the user from further complications associated with dynamic type identification. In addition, limited use of dynamic type information is the least error prone.

If it is not enough to know that the casting operation was successful, and you need a true type (for example, object-oriented I / O), then you can use the dynamic type query operations: static_type_info (), ptr_type_info (), and ref_type_info (). These operations return an object of the typeid class. As shown in the set and slist_set example, typeid objects can be compared. For most tasks, this information about the typeid class is sufficient. But for tasks that need more complete information about the type, the typeid class has a get_type_info () function:

```
class typeid {
        friend class Typ e_info;
        private:
                const Type_info * id;
        public:
                typeid (const Type_info * p): id (p) {}
                const Type_info * get_type_info () const {return id; }
                int operator == (typeid i) const;
};
```

The get_type_info () function returns a pointer to a non-variable (const) object of the Type_info class from typeid. It is essential that the object does not change: this is to ensure that the dynamic type information reflects the static types of the original program. It is bad if some type can change during program execution.

With the help of a pointer to an object of the Type_info class, the user gains access to information about the type from the typeid, and now his program begins to depend on the specific system of dynamic queries about the type and on the structure of dynamic information about it. But these tools are not part of the language standard, and it is not easy to define them using well-thought-out macros.

13.5.2 The Type_info class

The Type_info class contains the minimum amount of information to implement the ptr_cast () operation; it can be defined as follows:

```
class Type_in fo {
        const char * n; // name
        const Type_info ** b; // list of base classes
        public:
                Type_info (const char * name, const Type_info * base
[]);
                const char * name () const;
                Base_iterator bases (int direct = 0) const;
                int same (const Type_info * p) const;
                int has_base (const Type_info *, int direct = 0) const;
                int can_cast (const Type_info * p) const;
                static const Type_info info_obj;
                virtual typeid get_info () const;
                static typeid info ();
};
```

The last two functions must be defined in every class derived from Type_info.

The user should not care about the structure of the Type_info object, and it is included here for completeness only. The string containing the name of the type was introduced to enable searching for information in name tables, for example, in the debugger table. Using it, as well as information from the Type_info object, you can issue more meaningful diagnostic messages. In addition, if there is a need to have several objects of type Type_info, then the name can serve as a unique key for these objects.

```
const char * Type_info :: name () const
{
```

```
            return n;
    }

    int Type_info :: same (const Type_info * p) const
    {
            return this == p || strcmp (n, p-> n) == 0;
    }

    int Type_info :: can_cast (const Type_info * p) const
    {
            return same (p) || p-> has_base (this);
    }
```

Access to information about base classes is provided by the bases () and has_base () functions. The bases () function returns an iterator that produces pointers to the base classes of Type_info objects, and using the has_base () function, you can determine whether a given class is base for another class. These functions have an optional parameter direct, which indicates whether to consider all base classes (direct = 0), or only direct base classes (direct = 1). Finally, as described below, you can use the get_info () and info () functions to obtain dynamic type information for the Type_info class itself.

Here, the dynamic type query facility is deliberately implemented with very simple classes. This avoids linking to a specific library. Implementation per library may vary. You can, as always, advise users to avoid being overly dependent on implementation details.

The has_base () function searches for base classes using the list of base classes provided in Type_info. It is not necessary to store information about whether the base class is private or virtual, since any errors related to access restrictions or ambiguities will be detected during the broadcast.

```
    class base_iterator {
            short i;
            short alloc;
            const Type_info * b;
            public:
                    const Type_info * operator () ();
                    void reset () {i = 0; }
                    base_iterator (const Type_info * bb, int direct = 0);
                    ~ base_iterator () {if (alloc) delete [] (Type_info *) b; }
};
```

The following example uses an optional parameter to indicate whether to consider all base classes (direct == 0) or only direct base classes (direct == 1).

```
base_iterator :: base_iterator (const Type_info * bb, int direct)
{
          i = 0;
          if (direct) {// use a list of direct base classes
                    b = bb;
                    alloc = 0;
                    return;
          }
          // create a list of direct base classes:
          // int n = number of base
          b = new const Type_info * [n + 1];
          // add base classes to b
          alloc = 1;
          return;
}

const Type_info * base_iterator :: operator () ()
{
          const Type_info * p = & b [i];
          if (p) i ++;
          return p;
}
```

Now you can specify type query operations using macros:

```
#define static_type_info (T) T :: info ()
#define ptr_type_info (p) ((p) -> get_info ())
#define ref_type_info (r) ((r) .get_info ())
#define ptr_cast (T, p) \
          (T :: info () -> can_cast ((p) -> ge t_info ())? (T *) (p): 0)
#define ref_cast (T, r) \
          (T :: info () -> can_cast ((r) .get_info ()) \
          ? 0: throw Bad_cast (T :: info () -> name ()), (T &) (r))
```

The exception type is assumed to be Bad_cast:

```
class Bad_cast {
            const char * tn;
            // ...
            public:
            Bad_cast (const char * p): tn (p) {}
            const char * cast_to () {return tn; }
              // ...
};
```

In section $$ 4.7, it was said that the appearance of macros is a signal of problems. The problem here is that only the translator has direct access to literal types, and macros hide implementation specifics. In fact, a virtual function table is intended to store information for dynamic type queries. If an implementation directly supports dynamic type identification, then the operations in question can be implemented more naturally, efficiently, and elegantly. In particular, it is very easy to implement the ptr_cast () function, which converts a pointer to a virtual base class into a pointer to its derived classes.

13.5.3 How to Create a Dynamic Type Query System

It shows how you can directly implement dynamic type queries when the translator doesn't have such capabilities. This is a rather tedious task and you can skip this section as it only contains the details of a specific solution.

The set and slist_set classes from $$ 13.3 should be modified to work with type query operations. First of all, we need to introduce member functions into the base set class that use type query operations:

```
class set {
            public:
                        static const Type_info info_obj;
                        virtual typeid get_info () const;
                        static typeid info ();
                        // ...
};
```

When executing a program, the only representative of an object of type set is set :: info_obj, which is defined as follows:

```
c onst Type_info set :: info_obj ("set", 0);
```

Given this definition, the functions are trivial:

```
typeid set :: get_info () const {return & info_obj; }
typeid set :: info () {return & info_obj; }
typeid slist_set :: get_info () const {return & info_obj; }
typeid slist_set :: inf o () {return & info_obj; }
```

The get_info () virtual function will provide ref_type_info () and ptr_type_info () operations, and the info () static function will provide the static_type_info () operation.

With such a construction of the type query system, the main difficulty in practice is that for each class an object of type Type_info and two functions returning a pointer to this object are defined only once.

The slist_set class needs to be slightly modified:

```
class slist_set: public set, private slist {
        // ...
        public:
                st atic const Type_info info_obj;
                virtual typeid get_info () const;
                static typeid info ();
                // ...
};

static const Type_info * slist_set_b []
        = {& set :: info_obj, & slist :: info_obj, 0};
const Type_info slist_set :: info_obj ("slist_set", slist_set_b);
typeid slist_set :: get_info () const {return & info_obj; }
typeid slist_set :: info () {return & info_obj; }
```

13.5.4 Extended dynamic type information

The Type_info class contains only the minimum information required for type identification and safe casting operations. But since the Type_info class itself has member functions info () and get_info (), you can derive classes from it to dynamically determine which Type_info objects these functions return. Thus, without changing the Type_info class, the user can obtain more information about the type using objects returned by the dynamic_type () and static_type () functions. In many cases, additional information needs to include a table of object members:

```
struct Member_info {
            char * name;
            Type_info * tp;
            int offset;
};

class Map_info: public Type_info {
            Member_info ** mi;
            public:
                        stati c const Type_info info_obj;
                        virtual typeid get_info () const;
                        static typeid info ();
                        // access functions
};
```

The Type_info class is fine for the standard library. It is a base class with a minimum of required information, from which derived classes can be derived to provide more information. These derived classes can be defined either by the users themselves, or by some kind of utilities that work with C ++ text, or by the language translators themselves.

13.5.5 Correct and Incorrect Use of Dynamic Type Information

Dynamic type information can be used in many situations, including for: object I / O, object-oriented databases, debugging. At the same time, there is a high probability of misuse of such information. It is known that in the Simula language, the use of such facilities, as a rule, leads to errors. Therefore, these facilities were not included in C ++. It is too tempting to use dynamic type information when it is more correct to call a virtual function. Consider the Shape class from $$ 1.2.5 as an example. The rotate function could have been defined like this:

```
void rotate (const Shape & s)
            // incorrect use of dynamic
            // type information
{
            if (ref_type_info (s) == static_type_info (Circle)) {
                        // nothing is needed for this shape
            }
            else if (ref_type_info (s) == static_type_info (Triangle)) {
                // rotate the triangle
        }
        else if (ref_type_info (s) == static_type_info (Square)) {
                // rotate the square
        }
        // ...
}
```

If we use dynamic type information for the field type switch, we violate the principle of modularity in the program and deny the very goals of object-oriented programming. In addition, this solution is fraught with errors: if an object of a class derived from Circle is passed as a parameter to a function, then it will work incorrectly (indeed, it makes no sense to rotate the circle, but for an object representing a derived class, it may be required). Experience shows that programmers brought up in languages like C or Pascal find it difficult to avoid this trap. The programming style of these languages requires less foresight, and when creating a library, this style can simply be considered careless.

You might wonder why the ptr_cast () conditional cast operation is included in the interface with the dynamic type information system, and not the is_base () operation, which is directly defined using the has_base () operation from the Type_info class. Consider this example:

```
void f (dialog_box & db)
{
        if (is_base (& db, dbox_w_str)) { // is db base
                                              // for dbox_w-str?
                dbox_w_str * dbws = (dbox_w_str *) & db;
                // ...
        }
        // ...
}
```

The solution using ptr_cast ($$ 13.5) is shorter, besides, here the explicit and unconditional casting operation is separated from the check in the if statement, which means there is a possibility of error, inefficiency, and even an incorrect result. An incorrect result can occur in those rare cases when the dynamic type identification system recognizes that one type is derived from another, but the translator does not know this fact, for example:

```
class D;
class B;

void g (B * pb)
{
            if (is_base (pb, D)) {
                        D * pb = (D *) pb;
                        // ...
            }
            // ...
}
```

If the translator does not yet know the following description of class D:

```
class D: public A, public B {
            // ...
};
```

then an error occurs because correct casting of the pointer pb to D * requires changing the pointer value. The solution with the ptr_cast () operation does not face this difficulty, since this operation is applicable only if the descriptions of both of its parameters are in scope. The above example shows that the casting operation for undescribed classes is inherently unreliable, but its prohibition significantly impairs compatibility with the C language.

13.6 Extensive interface

When abstract types ($$ 13.3) and node classes ($$ 13.4) were discussed, it was emphasized that all the functions of a base class are implemented in the base class itself or in a derived class. But there is another way to build classes. Consider, for example, lists, arrays, associative arrays, trees, etc. It is natural to desire for all these types, often called containers, to create a generalizing class that can be used as an interface with any of the listed types. Obviously, the user does not need to know the details of a particular container. But the task of defining an interface for a generic container is not trivial. Assuming such a container is defined as an abstract type, then what operations should it provide? You can provide only those operations that are in each container, i.e. intersection of many operations, but such an interface would be too narrow. In fact, in many meaningful cases, such an intersection is empty. As an alternative solution, you can provide a union of all sets of operations and provide for a dynamic error when a "nonexistent" operation is applied to an object in this interface. The union of the interfaces of classes that represent many concepts is called a broad interface. Let's describe a "common" container of objects of type T:

```
   class container {
            public:
                     struct Bad_operation {// exception class
                              const char * p;
                              Bad_operation (const char * pp): p (pp) {}
                     };
                     virtual void put (const T *)
                              {throw Bad_operation ("container :: put"); }
                     virtual T * get ()
                              {throw Bad_o peration ("container :: get"); }
                     virtual T * & operator [] (int)
                              {throw Bad_operation ("container :: [] (int)");
   }
                     virtual T * & operator [] (const char *)
                              {throw Bad_operation ("container :: [] (char
   *)"); }
            // ...
   };
```

Still, there are few implementations where both indexing and list operations are well represented, and it may not be worth combining them in the same class.

Note that the difference is that pure virtual functions are used in an abstract type to guarantee translation-stage validation, and rich interface functions are used to detect errors at run-time, which trigger exceptions.

A container implemented as a simple one-way list can be described as follows:

```
   class slist_container: public container, private slist {
            public:
                     void put (const T *);
                     T * get ();
                     T * & operator [] (int)
```

```
                        {throw Bad_operation ("slist :: [] (int)"); }
                T * & operator [] (const * char)
                        {throw Bad_operation ("slist :: [] (char *)"); }
                // ...
};
```
Indexing operations have been introduced to simplify the handling of dynamic errors for the list. It was possible not to enter these unimplemented operations for the list and restrict ourselves to less complete information provided by the exceptions launched in the container class:
```
class vector_container: public containe r, private vector {
        public:
                T * & operator [] (int);
                T * & operator [] (const char *);
                // ...
};
```
To be careful, everything works fine:
```
void f ()
{
        slist_container sc;
        vector_container vc;
        // ...
}

void user (container & c1, container & c2)
{
        T * p 1 = c1.get ();
        T * p2 = c2 [3];
        // you can't use c2.get () or c1 [3]
        // ...
}
```
However, to avoid errors during program execution, it is often necessary to use dynamic type information ($$ 13.5) or special situations ($$ 9). Let's give an example:
```
void u ser2 (container & c1, container & c2)
/ *
        error detection is easy, recovery is difficult
```

```
   * /
   {
               try {
                           T * p1 = c1.get ();
                           T * p2 = c2 [3];
                           // ...
               }
               catch (container :: Bad_operation & bad) {
                           // Arrived!
                           // What to do now?
               }
   }
```

or another example:

```
   void user3 (container & c1, container & c2)
   / *
               error detection is not easy,
               and recovery is still a challenge
   * /
   {
               slist * sl = ptr_cast (slist_container, & c1);
               vector * v = ptr_cast (vector_container, & c2);
               if (sl && v) {
                             T * p1 = c1.get ();
                           T * p2 = c2 [3];
                           // ...
               }
               else {
                           // Arrived!
                           // What to do now?
               }
   }
```

Both of the error detection methods shown in these examples result in a program with bloated code and slow execution speed. Therefore, they usually simply ignore possible errors in the hope that the user will not come across them. But this does not make the task easier, because full testing is difficult and requires a lot of effort.

Therefore, if the goal is a program with good characteristics, or high guarantees of program correctness are required, or, in general, there is a good alternative, it is better not to use extensive interfaces. In addition, the use of a broad interface breaks the one-to-one correspondence between classes and concepts, and then new derived classes are introduced simply for ease of implementation.

13.7 Application area framework

We have listed the kinds of classes from which you can create libraries aimed at designing and reusing applications . They provide certain "building blocks" and explain how to build from them. The application developer creates a framework into which the generic building blocks must fit. The application design problem can have a different, more binding solution: write a program that itself will create the overall framework of the application area. The application developer will build application programs into this framework as building blocks . The classes that form the skeleton of the application area have such an extensive interface that they can hardly be called types in the usual sense. They are approaching the limit when they become purely application classes, but at the same time they actually have only descriptions, and all actions are specified by functions written by application programmers.

For example, consider a filter, i.e. a program that can do the following: read an input stream, perform some operations on it, issue an output stream, and determine the final result. The primitive skeleton for the filter will consist of defining a set of operations that the application programmer must implement :

```
class filter {
            public:
                        class Retry {
                        public:
                                    virtual const char * message () {return 0; }
};

            virtual void start () {}
            virtua l int retry () {return 2; }
            virtual int read () = 0;
            virtual void write () {}
```

```
        virtual void compute () {}
        virtual int result () = 0;
};
```

The functions needed for derived classes are described as pure virtual, the rest of the functions are simply empty. The framework contains the main processing loop and rudimentary error handling tools:

```
int main_loop (filter * p)
{
        for (;;) {
                try {
                        p-> start ();
                        while (p-> read ()) {
                                p-> compute ();
                                p-> write ();
                        }
                        return p-> result ();
                }
                catch (filter :: Retry & m) {
                        cout << m.message () << '\ n';
                        int i = p-> retry ();
                        if (i) return i;
                }
                catch (...) {
                        cout << "Fatal filter error \ n";
                        return 1;
                }
        }
}
```

The application can now be written like this:

```
class myfilter: public filter {
        istream & is;
        ostream & os;
        char c;
        int nchar;
        public:
                int read () {is.get (c); return is.good (); }
```

```
                    void compute () {nchar ++; };
                    int result ()
                    {os << nchar
                                    << "characters read \ n";
                         return 0;
   }

   myfilter (istream & ii, ostream & oo)
          : is (ii), os (oo), nchar (0) {}
   };
```

and call it like this:

```
   int main ()
   {
          myfilter f (cin, cout);
          return main_loop (& f);
   }
```

A real framework, in order to be useful in real-world tasks, must create more advanced structures and provide more useful functions than in our simple example. Typically, the framework forms a tree of node classes. The application programmer only supplies the classes that serve as leaves in this multilevel tree, thereby achieving commonality among the various applications and making it easier to reuse useful functions provided by the framework. The framework can be facilitated by libraries that define some useful classes such as scrollbar ($$ 12.2.5) and dialog_box ($$ 13.4). After defining his application classes, the programmer can use these classes.

13.8 Interface classes

One of the most important kinds of classes is usually overlooked - the "humble" interface classes. Such a class does not do any big work, because otherwise, it would not be called an interface class . The task of an interface class is to adapt some useful function to a specific context. The beauty of interface classes is that they allow you to share a useful function without making it rigid. Indeed, the function itself can not be expected to satisfy a wide variety of needs equally well.

An interface class in its pure form does not even require code generation. Recall the description of a template of type Splist from $$ 8.3.2:

```
template <class T>
        class Splist: private Slist <void *> {
        public:
                void insert (T * p) {Slist <void *> :: insert (p); }
                void append (T * p) {Slist <void *> :: append (p); }
                T * get () {return (T *) Slist <void *> :: get (); }
};
```

The Splist class converts a list of unsafe generic void * pointers into a more convenient family of safe classes that represent lists. To avoid using interface classes too much, you need to use substitution functions. In examples like this one, where the task of substitution functions is only to fit the type, there is no overhead in memory and program execution speed .

Naturally, you can think of an abstract base class as an interface , which represents an abstract type implemented by concrete types ($$ 13.3), just like the control classes in section 13.9. But here we are considering classes that have no other purpose - only the task of adapting the interface.

Consider the problem of merging two class hierarchies using multiple inheritance. What to do in case of a collision of names, i.e. situations when two classes use virtual functions with the same name that perform completely different operations? Let there be a video game called "Wild West", in which a dialogue with the user is organized using a general view window (Window class):

```
class Win dow {
        // ...
        virtual void draw ();
};

class Cowboy {
        // ...
        virtual void draw ();
};

class CowboyWindow: public Cowboy, public Window {
        // ...
};
```

In this game, the CowboyWindow class represents the movement of the cowboy on the screen and controls the interaction of the player with the cowboy. Obviously, there will be many useful functions defined in the Window and Cowboy classes , so it is preferable to use multiple inheritance rather than declaring Window or Cowboy as members. I would like to pass an object of the CowboyWindow type to these functions as a parameter, without requiring the programmer to specify any object specifications. This raises the question of which function to choose for the CowboyWindow: Cowboy :: draw () or Window :: draw ().

There can only be one function called draw () in the CowboyWindow class , but since this useful function works with Cowboy or Window objects and knows nothing about CowboyWindow, the CowboyWindow class should suppress (override) both the Cowboy :: draw () function and the Window_draw (). It is wrong to suppress both functions with one - draw (), because although the same name is used, all draw () functions are different and cannot be overridden by one.

Finally, it is desirable that the inherited functions Cowboy :: draw () and Window :: draw () have different unique names in the CowboyWindow class .

To solve this problem, you need to introduce additional classes for Cowboy and Window. Two new names are introduced for the draw () functions and it is guaranteed that calling them in the Cowboy and Window classes will result in the new named functions being called :

```
class CCowboy: public Cowboy {
        virtual int cow_draw (int) = 0;
        void draw () {cow_draw (i); } // override Cowboy :: draw
};

class WWindow: public Window {
        virtual int win_draw () = 0;
        void draw () {win_draw (); } // override Window :: dr aw
};
```

Now, using the CCowboy and WWindow interface classes, you can define the CowboyWindow class and make the required overrides of the cow_draw () and win_draw functions:

```
class CowboyWindow: public CCowboy, public WWindow {
```

```
            // ...
            void cow_draw ();
            void win_d raw ();
   };
```

Note that in reality the difficulty arose only because both draw () functions have the same parameter type. If the parameter types were different, then the usual rules for resolving ambiguity on overloading would ensure that there were no difficulties despite the existence of different functions with the same name.

For each use case of an interface class, you can propose a language extension such that the required adaptation is more efficient or specified in a more elegant way. But such cases are quite rare, and there is no point in overloading the language by providing special tools for each case. In particular, the case of name collisions when merging class hierarchies is quite rare, especially when compared to how often the programmer creates classes. Such cases can arise when merging class hierarchies from different areas (as in our example: games and operating systems). Merging such disparate class structures is always challenging, and resolving name collisions is far from the hardest part. Problems arise here due to different error handling , initialization, memory management strategies . The naming collision example was given because the proposed solution — introducing interface classes with transition functions — has many other uses. For example, they can be used to change not only names, but also types of parameters and return values, insert certain dynamic checks, etc.

The adapter functions CCowboy :: draw () and WWindow_draw are virtual, and simple substitution optimization is not possible. However, it is possible that the translator recognizes such functions and removes them from the call chain.

Interface functions are used to tailor the interface to user requests. Thanks to them, operations scattered throughout the program are collected in the interface . Let's turn to the vector class from $$ 1.4. For vectors like arrays, the index is zero-based. If the user wants to work with a range of indices other than the range 0..size-1, you need to make appropriate adjustments, for example, the following:

```
   void f ()
   {
```

```
            vector v (10); // range [0: 9]
            // as if v is in the range [1:10]:
            for (int i = 1; i <= 10; i ++) {
                        v [i-1] = ... // remember to recalculate the index
            }
            // ...
   }
```

The best solution is given by the vec class with arbitrary index bounds:

```
   class vec: public vector {
            int lb;
            public:
                        vec (int low, int high)
                        : vector (high-low + 1) {lb = low; }
                        int & operator [] (int i)
                                    {return vector :: ope rator [] (i-lb); }
                        int low () {return lb; }
                        int high () {return lb + size () - 1; }
   };
```

The vec class can be used without additional operations required in the first example:

```
   void g ()
   {
            vec v (1.10); // range [1:10]
            for (int i = 1; i <= 10; i ++) {
                        v [i] = ...
            }
            // ...
   }
```

Obviously, the vec class is clearer and safer.

Interface classes also have other important uses, such as the interface between C ++ programs and programs in another language ($$ 12.1.4), or an interface with C ++ specific libraries.

13.9 Control classes

The concept of an abstract class provides an effective means of separating an interface from its implementation. We applied this concept and got a persistent link between the interface given by the abstract type and the implementation represented by the concrete type. So, it is impossible to switch an abstract iterator from one source class to another, for example, if the set is exhausted (set class), it is impossible to switch to streams.

Further, as long as we work with objects of an abstract type using pointers or references, all the advantages of virtual functions are lost . The user program begins to depend on specific implementation classes . Indeed, without knowing the size of the object, even with an abstract type, you cannot place an object on the stack, pass it as a parameter by value, or place it as static. If work with objects is organized through pointers or references, then the task of memory allocation is transferred to the user ($$ 13.10).

There is another limitation related to the use of abstract types. An object of such a class always has a certain size, but classes that reflect the real concept may require memory of different sizes.

There is a common technique for overcoming these difficulties, namely, to split a separate object into two parts: the control part, which defines the object's interface, and the content, which contains all or most of the object's attributes. The connection between the two parts is realized using a pointer in the control part to the content part. Usually the control part contains other data besides the pointer , but there are not many of them. The bottom line is that the composition of the control part does not change when the content part changes, and it is so small that you can freely work with the objects themselves, and not with pointers or references to them.

control part content part

A simple example of a control class is the string class from $$ 7.6. It contains the interface, access control and memory management for the content part. In this example, the control and content parts are represented by concrete types, but more often the content part is represented by an abstract class.

Now, back to the abstract set type from $$ 13.3. How can you define a control class for this type, and what are the pros and cons? For a given set class, you can define a control class simply by overloading the -> operation:

```
class set_handle {
        set * rep;
        public:
                set * operator -> () {return rep; }
        set_handler (set * pp): rep (pp) {}
};
```
This does not affect working with sets too much, just objects of type set_handle are passed instead of objects of type set & or set *, for example:
```
void my (set_handle s)
{
        for (T * p = s-> first (); p; p = s-> next ())
        {
                // ...
        }
        // ...
}

void your (set_handle s)
{
        for (T * p = s-> first (); p; p = s-> next ())
        {
                // ...
        }
        // ...
}

void user ()
{
        set_handle sl (new slist_set);
        set_handle v (new vector_set v (100));
        my (sl);
        your (v);
        my (v);
        your (sl);
}
```
If the set and set_handle classes were co-developed, it is easy to count the number of sets created:
```
class set {
        fri end class set_handle;
```

```
            protected:
                        int handle_count;
            public:
                        virtual void insert (T *) = 0;
                        virtual void remove (T *) = 0;
                        virtual int is_member (T *) = 0;
                        virtual T * first () = 0;
                        virtual T * next () = 0;
                        set (): handle_count (0) {}
   };
```

To count the number of objects of a given set type, increase or decrease the value of the set_handle counter in the managing class :

```
   class set_handle {
            set * rep;
            public:
                        set * operator -> () {return rep; }
                        set_handle (set * pp)
                                    : rep (pp) {pp-> handle_count ++; }
                        s et_handle (const set_handle & r)
                                    : rep (r.rep) {rep-> handle_count ++; }
                        set_handle & operator = (const set_handle & r)
                        {
                                    rep-> handle_count ++;
                                    if (--rep-> handle_count == 0) delete rep;
                                    rep = r.rep;
                                    return * this;
                        }
                        ~ set_handle ()
                        {if (--rep-> handle_count == 0) delete rep; }
   };
```

If all calls to the set class necessarily go through set_handle, the user does not have to worry about the memory allocation for objects of the set type.

In practice, sometimes it is necessary to retrieve the pointer to the content part from the managing class and use it directly. You can, for example, pass such a pointer to a function that knows nothing about the controlling class. If the function does not destroy the object it received the pointer to, and if it does not store the pointer for later use after returning, there should be no errors . It might be useful to switch the controlling class to a different content part:

```
class set_handle {
        set * rep;
        public:
                // ...
                set * get_rep () {return rep; }
                void bind (set * pp)
                {
                        pp-> handle_ count ++;
                        if (--rep-> handle_count == 0) delete rep;
                        rep = pp;
                }
};
```

Creating new classes that derive from set_handle usually doesn't make much sense, since this is a concrete type with no virtual functions. Another thing is to build a control class for a family of classes defined by one base class . A useful trick would be to derive from such a control class. This technique can be used for both node classes and abstract types.

It is natural to define the controlling class as a template like:

```
template <class T> class handle {
        T * rep;
        public:
                T * operator -> () {return rep; }
                // ...
};
```

But this approach requires interaction between the manager and the "managed" class. If the managed and managed classes are developed together, for example, during the creation of a library, then this may be acceptable. However, there are other solutions ($$ 13.10).

By overloading the operation ->, the managing class gets the ability to control and perform some operations each time the object is accessed. For example, you can count how often objects are used through a control class:

```
template <class T>
```

```
class Xhandle {
        T * rep;
        int count;
        public:
                T * operator -> () {count ++; return rep; }
                // ...
};
```
A more complex technique is needed if you want to perform operations both before and after accessing an object. For example, you might need a set with a lock when performing add to and remove operations from a set. Here, in fact, in the control class, you have to duplicate the interface with the objects of the content part:
```
class set_controller {
        set * rep;
        // ...
        public:
                lock ();
                unlock ();
                virtual void insert (T * p)
                        {lock (); rep-> insert (p); unlock (); }
                virtual void remove (T * p)
                        {lock (); rep-> remove (p); unlock (); }
                virtual int i s_member (T * p)
                        {return rep-> is_member (p); }
                virtual T * first () {return rep-> first (); }
                virtual T * next () {return rep-> next (); }
                // ...
};
```
Writing adapter functions for the entire interface is tedious (which means errors can appear), but it is not difficult and does not degrade the program 's performance.

Note that not all set functions should be locked. As the author's experience shows, a typical case is when operations before and after accessing an object need to be performed not for all, but only for some member functions. Blocking all operations, as is done in the monitors of some operating systems, is redundant and can significantly degrade parallel execution.

By overriding all the interface functions in the control class, we got the advantage over the -> operation overloading technique that now we can build classes derived from set_controller. Unfortunately, we can lose some of the benefits of the managing class if we add data members to the derived classes . We can say that the software space that is shared between the managed classes decreases as the software space of the managed class grows.

13.10 Memory Management

When designing a library or just a program with a long computing time , one of the key issues is related to memory management. In general, the creator of a library does not know in what environment it will work. Will the memory resource be so critical that memory shortages become a serious problem, or will the overhead of memory management become a serious obstacle ?

One of the main questions of memory management can be formulated as follows: if the function f () passes or returns a pointer to an object, then who should destroy that object? It is also necessary to answer the related question: at what point can an object be destroyed? The answers to these questions are especially important for creators and users of containers such as lists, arrays, and associative arrays. From the point of view of the library creator, the ideal answers would be "System" and "At the moment when the object is not used by anyone else." When the system destroys an object, it is usually said to be garbage collected, and the part of the system that determines that the object is no longer used by anyone and destroys it is called the garbage collector.

Unfortunately, the use of a garbage collector can incur overhead costs of computation time and memory, interruptions of useful functions, certain hardware support, difficulties in linking parts of a program in different languages, or simply complicating the system. Many users cannot afford this. They say that Lisp programmers know how important memory management is, and therefore cannot give it to the user. C programmers also know how important memory management is, and therefore can not leave it to the system.

Therefore, in most C ++ programs, you do not have to rely on the garbage collector and you need to propose your own strategy for allocating objects in free memory without accessing the system. But C ++ implementations with a garbage collector do exist.

Let's consider the simplest memory management scheme for C ++ programs . To do this, replace operator new () with a trivial placement function , and operator delete () with an empty function:

```
inline size_t align (size_t s )
/ *
        Even a simple allocation function needs memory alignment so
that
        the object could be set up a pointer of arbitrary type
* /
{
        union Word {void * p; long double d; long l; }
        int x = s + sizeof (Word) - 1;
        x - = x% sizeof (Word);
        return x;
}

static void * freep; // set start to free memory
void * operator new (size_t s) // simple linear placement function
{
        void * p = freep;
        s = align (s);
        freep + = s;
        return p;
}
void operator delete (void *) {} // empty
```

If the memory is infinite, then our solution provides a garbage collector without any complexity and overhead. This approach is not applicable for libraries when it is not known in advance how the memory will be used, and when the program using the library will have a long computation time. This way of allocating memory is ideal for programs that require a limited amount of memory or a space proportional to the size of the input data stream.

13.10.1 Garbage collector

Garbage collection can be thought of as simulating infinite memory in memory of a limited size. With this in mind, a common question can be answered: Should the garbage collector call the destructor for the objects it uses memory? The correct answer is no, because if an object allocated in free memory has not been deleted, then it will not be destroyed either. Based on this, the delete operation can be viewed as a request to call the destructor (and it is also a message to the system that the object's memory can be used). But what if you really want to destroy a free- memory object that hasn't been deleted? Note that this question does not arise for static and automatic objects - destructors for them are always implicitly called. Further, the destruction of an object "during garbage collection" is essentially an operation with unpredictable results. It can occur at any time between the last use of the object and the "end of the program", which means that the state of the program at this moment is unknown. Quotes are used here because it is difficult to determine exactly what the end of the program is. (approx. transl.)

It is difficult to program such operations correctly and they are not as useful as they seem.

The problem of destroying objects, if the time of this operation is not precisely specified, can be solved using the program for servicing requests for destruction. Let's call it a claim server. If an object needs to be destroyed at the end of the program, then its address and a pointer to the "cleanup" function must be written into the global associative array . If the object is deleted by an explicit operation, the application is canceled. When the server itself is destroyed (at the end of the program), the cleanup functions are called for all remaining applications. This solution is also suitable for garbage collection, since we see it as a simulation of infinite memory. For the garbage collector, you need to choose one of two solutions: either delete the object when the only remaining reference to it is a reference located in the array of the server itself , or (the standard solution) do not delete the object until the end of the program, since there is still a reference to it.

The claim server can be implemented as an associative array ($$ 8.8):

```
class Register {
        Map <void *, void (*) (void *)> m;
        public:
```

```
                        insert (void * po, void (* pf) ()) {m [po] = pf; }
                        remove (void * po) {m.remove (po); }
  };
  Register cleanup_register;
```

A class constantly accessing the server might look like this:

```
  class X {
            // ...
            static void cleanup (void *);
            public:
                        X ()
                        {
                                    cleanup_register.insert (this, & cleanup);
                                    // ...
                        }
                        ~ X () {cleanup (this); }
                        // ...
  };

  void X :: cleanup (void * pv)
  {
            X * px = (X *) pv;
            cleanup_ register.remove (pv);
            // clean up
  }
```

To avoid having to deal with types in the Register class, we used a static member function with a pointer of type void *.

13.10.2 Containers and disposal

Let's say we don't have infinite memory and no garbage collector. What kind of memory management can a container creator , such as a Vector class, rely on ? For simple elements like int, obviously, you just need to copy them to the container. It is equally obvious that for other types, such as the abstract class Shap e, a pointer should be stored in the container. The library creator must consider both options. Here's a sketch of an obvious solution:

```
  template <class T> Vector {
```

```
            T * p;
            int sz;
            public:
                        Vector (int s) {p = new T [sz = s]; }
                        // ...
   };
```

If the user will not enter objects of the Shape type into the container instead of pointers to objects, then this solution is suitable for both options.

```
   Vector <Shape *> vsp (200); // fine
   Vector <Shape> vs (200); // error during broadcast
```

Fortunately, the compiler keeps track of an attempt to create an array of objects of the abstract base class Shape.

However, if pointers are used, the creator of the library and the user must agree on who will delete the objects stored in the container. Let's consider an example:

```
   void f ()
            // inconsistent use of funds
            // memory management
   {
            Vector <Shape *> v (10);
            Circle * cp = new Circle;
            v [0] = cp;
            v [1] = new Triangle;
            Square s;
            v [2] = & s;
            delete cp; // does not delete the objects that are
   configured
                                                            // pointers located in
   the container
   }
```

If you use the implementation of the Vector class from $$ 1.4.3, the Triangle object in this example will forever remain suspended (there are no pointers to it), unless there is a garbage collector. The key to memory management is correctness. Consider this example:

```
   void g ()
```

```
// correct use of memory management tools
{
            Vector <Shape *> v (10);
            Circle * cp = new Circle;
            v [0] = cp;
            v [1] = new Triangle;
            Square s;
            v [2] = & s;
            delete cp;
            delete v [1];
}
```

Consider now such a vector class that monitors the deletion of pointers entered into it:

```
template <class T> MVector {
            T * p;
            int sz;
            public:
                        MVector (int s);
                        ~ MVector ();
                        // ...
};

template <class T> MVector <T> :: MVector (int s)
{
            // check s
            p = new T [sz = s];
            for (int i = 0; i <s; i ++) p [i ] = 0;
}

template <class T> MVector <T> :: ~ MVector ()
{
            for (int i = 0; i <s; i ++) delete p [i];
            delete p;
}
```

The user can expect the pointers contained in the MVector to be deleted. It follows from this that after the MVector is removed, the user should not use pointers to access the objects that were stored in this container. When the MVector is destroyed, it should not contain pointers to static or automatic objects, for example:

```
void h ()
// correct use of memory management tools
{
            MVector <Shape *> v (10);
            Circle * cp = new circle ();
            v [0] = cp;
            v [1] = new Triangle;
            Square s;
            v [2] = & s;
            v [2] = 0; // prevents s being removed
                                                            // all remaining
pointers
                                                            // automatically
removed on exit
}
```

Naturally, this solution is only suitable for containers that do not contain copies of objects, and for the Map class ($$ 8.8), for example, it is not suitable. Here is a simple version of the destructor for MVector, but there is an error because the same pointer, entered twice into the container, will be deleted twice too.

Building and destroying containers that keep track of the disposal of the objects they contain is a costly operation. Copying of these containers should be prohibited, or at least strongly restricted (really, who will be responsible for deleting a container or a copy of it?):

```
template <class T> MVector {
            private:
                        MVector (const MVector &); // prevents
copying
                        MVector & operator = (const MVector &); //same
                        // ...
};
```

It follows that such containers must be passed by reference or pointer (if at all, it should be done), but then another kind of difficulty arises in memory management.

It is often useful to reduce the number of pointers that the user must follow. Indeed, it is much easier to keep track of 100 first-level objects, which, in turn, manage 1000 zero-level objects, than directly working with 1100 objects. Actually, the techniques given in this section, like other techniques used for memory management, boil down to standardization and universalization through the use of constructors and destructors. This allows you to reduce the memory management problem for an almost unimaginable number of objects, say 100,000, to a manageable number, say 100.

Is it possible to define a container class in this way so that the programmer creating an object of the container type can choose a memory management strategy from several possible ones, although only one container type is defined ? If this is possible, will it be justified? The answer to the second question is yes, since most of the functions in the system shouldn't care about memory allocation at all. The existence of several different types for each container class is an unnecessary complication for the user. The library must contain either one kind of containers (Vector or MVector), or both, but presented as variants of the same type, for example:

```
template <class T> PVector {
        T ** p;
        int sz;
        int managed;
        public:
                PVector (int s, int managed = 0 );
                ~ PVector ();
                // ...
};

template <class T> PVector <T> :: PVector (int s, int m)
{
        // check s
        p = new T * [sz = s];
        if (managed = m)
                for (int i = 0; i <s; i ++) p [i] = 0;
}
```

```
template <class T> PVector <T> :: ~ PVector ()
{
            if (managed) {
                        for (int i = 0; i <s; i ++) delete p [i];
            }
            delete p;
}
```

An example of a class the library has to offer to facilitate memory management is the management class from $$ 13.9. Since references to it are counted in the managing class, you can safely transfer an object of this class, without thinking about who will delete objects accessible through it. This is done by the object of the managing class itself. However, this approach requires that the managed objects have a field to count the number of uses. By introducing an additional object, you can simply remove this strict requirement:

```
template <class T>
class Handle {
            T * rep;
            int * pcount;
            public:
                        T * operator -> () {return rep; }
                        Handle (const T * pp)
                                    : rep (pp), pcount (new int) {(* pcount) = 0; }
                        Handle (const Handle & r)
                                    : rep (r.rep), pcount (r .count) {(* pcount) ++; }
                        void bind (const Handle & r)
                        {
                        if (rep == r.rep) return;
                        if (- (* pcount) == 0) {delete rep; delete pcount; }
                        rep = r.rep;
                        pcount = r.pcount;
                        (* pcount) ++;
}
            Handle & operator = (const Handle & r)
            {
                        bind (r);
```

```
                return * this;
        }
        ~ Handle ()
        {
                if (- (* pcount) == 0) {delete rep; delete pcount; }
        }
};
```

13.10.3 Allocation and release functions

In all the examples given, memory was seen as something given. However, the general purpose function for allocating free memory is surprisingly less efficient than the special purpose function . A degenerate case of such functions can be considered the given example with allocation in "infinite" memory and with an empty release function. The library can have more meaningful placement functions, and sometimes they can double the speed of the program. But before attempting to use them to optimize your program, run a profiler for it to identify the memory allocation overhead.

In sections $$ 5.5.6 and $$ 6.7, you have shown how you can use the placement function for objects of class X by defining the functions X :: operator new () and X :: operator delete () . There is a certain difficulty here. For two classes X and Y, the placement functions may be so similar that it is desirable to have one such function. In other words, it is desirable to have a class in the library that provides placement and release functions suitable for placing objects of this class. If there is such a class, then the allocation and release functions for this class are obtained by binding the general allocation and release functions to it :

```
class X {
static Pool my_pool;
        // ...
        public:
                // ...
                void * operator new (size_t) {return my_pool.alloc (); }
                void operator delete (void * p) {my_pool.free (p); }
};

P ool X :: my_pool (sizeof (X));
```

The Pool class allocates memory in blocks of the same size. In the above example, the my_pool object allocates memory in blocks of sizeof (X).

A description of the class X is compiled and Pool is used taking into account the optimization of the program speed and the compactness of the presentation. Note that the size of the allocated memory blocks is "built-in" for the class, therefore the size parameter of the X :: operator new () function is not used. A variant of the X :: operator delete () function without a parameter is used. If class Y is derived from class X, and sizeof (Y)> sizeof (X), then class Y must have its own allocation and deallocation functions . Inheriting functions from class X will lead to disaster. Fortunately, defining such functions for Y is easy.

The Pool class provides a linked list of items of the required size. Items are allocated from a fixed-size block of memory and new blocks of memory are requested as needed. Items are grouped in large chunks to minimize memory accesses to the general allocation function. Until the PooL object itself is destroyed, the memory is never returned to the general allocation function.

Here is a description of the Pool class:

```
class Pool {
        struct Link {Link * next; }
        cons t unsigned esize;
        Link * head;
        Pool (Pool &); // copy protection
        void operator = (Pool &); // copy protection
        void grow ();
        public:
                Pool (unsigned n);
                ~ Pool ();
                void * alloc ();
                void free (void * b);
};

inline void * Pool :: alloc ()
{
        if (head == 0) grow ();
        Link * p = head;
```

```
            head = p-> next;
            return p;
}

inline void Pool :: free (void * b)
{
            Link * p = (Link *) b;
            p-> next = head;
            head = p;
}
```

It is logical to place the above descriptions in the header file Pool.h. The following definitions can appear anywhere in the program and complete our example. The Pool object must be initialized with the constructor:

```
Pool :: Pool (unsigned sz): esize (sz)
{
            head = 0;
}
```

The Pool :: grow () function will bind all the items into the head list of free quanta , forming a new block from them. The definitions of the remaining member functions are left as Exercises 5 and 6 in $$ 13.11.

```
void Pool :: grow ()
{
            const int overhead = 12;
            const int chunk_size = 8 * 1024-overhead;
            const int nelem = (chunk_size-esize) / esize;
            char * start = new char [chunk_size];
            char * last = & start [(nelem-1) * esize];
            for (char * p = start; p <last; p + = esize)
                        ((Link *) p) -> next = ((Link *) p) +1;
            ((Link *) last) -> next = 0;
            head = (Link *) start;
}
```

13.11 Exercises

1. (* 3) Complete the definitions of the member functions of the Type_i nfo class .

2. (* 3) Suggest the structure of the Type_info object to make Type_info :: get_info () redundant, and rewrite the Type_info member functions with this in mind.

3. (* 2.5) How clear can you write examples with Dialog_box without using macros (as well as language extensions)? How vividly can you write them using language extensions?

4. (* 4) Explore two widely used libraries. Classify all library classes by breaking them down into: concrete types, abstract types, node classes, management classes, and interface classes. Are abstract node classes and concrete node classes used? Can anyone suggest a more appropriate class breakdown for these libraries? Is the extensive interface used? What dynamic type information tools are available (if any)? What is the memory management strategy?

5. (* 3) Define a templated version of the Pool class from $$ 13.10.3. Let the size of the allocated memory element be a parameter of the type template , not the constructor.

6. (* 2.5) Improve the Pool pattern from the previous exercise so that some elements are positioned while the designer is running. Formulate what the portability problem will be if you use Pool with the element type char, show how to fix it.

7. (* 3) If your version of C ++ does not directly support dynamic type queries, please refer to your main library. It implemented if there is a mechanism of dynamic type queries? If so, define the operations from $$ 13.5 as a superstructure on top of this mechanism.

8. (* 2.5) Define a string class that does not have any dynamic control, and a second derived inline class that only performs dynamic control and refers to the first. Indicate the pros and cons of such a solution versus a solution that does sample dynamic control, versus the invariants approach as suggested in $$ 12.2.7.1. How can these approaches be combined?

9. (* 4) Define the Storable class as an abstract base class with writeout () and readin () virtual functions. For simplicity, let's say that a string of characters is sufficient to specify the desired address space . Use the Storable class to implement object exchange with disk. Check it out on objects of several classes as you see fit.

10. (* 4) Define a base Persistent class with save () and nosave () operations that will check that the destructor has created an object in a specific memory. What other useful operations can you suggest? Check out the Persistent class on multiple classes of your choice. Is Persistent a node class, concrete type or abstract type? Argument your answer.

11. (* 3) Only describe the stack class, which implements the stack using create () (create a stack), delete () (destroy the stack), push () (write to the stack), and pop () (read from the stack). Use only static members. Define an id class to bind and label stacks. Ensure that the user can copy stack :: id objects, but cannot otherwise manipulate them. Compare this stack definition with the stack class from $$ 8.2.

12. (* 3) Describe the stack class, which is an abstract type ($$ 13.3). Suggest two different implementations for the interface given by stack. Write a small program that works with these classes. Compare this solution with the stack- defining classes from the previous exercise and from $$ 8.2.

13. (* 3) Make a description of the stack class for which you can change the implementation over time. Hint: "Any problem can be solved by introducing one more indirection."

14. (* 3.5) Define a class Oper containing an identifier (of some suitable type) and an operation (some pointer to a function). Define a cat_object class containing a list of Oper objects and a void * object. In the cat_object class, set the operation: add_oper (), which adds the object to the list; remove_oper (id), which removes the Oper object with id from the list; operator () (id, arg), which calls a function from the Oper object with the id id. Implement a stack of Oper objects using the cat_object class . Write a small program that works with these classes.

15. (* 3) Define a template of type Object that serves as the base class for cat_object. Use Object to implement a stack for String objects . Write a small program that uses this type pattern.

16. (* 3) Define a variant of the Object class called Class, in which objects with the same identifier have a common list of operations. Write a small program that uses this type pattern.

17. (* 3) Define a template of type Stack that defines a traditional and reliable interface with a stack implemented by a template object of type Object. Compare this with the definition of the stack classes defined stack of previous exercises. Write a small program that uses this type pattern.

f