Advanced Programming Languages

Authors

Tibor Ásványi
Ákos Balaskó
Iván József Balázs
Balázs Csizmazia
Péter Csontos
Szabina Fodor
Attila Góbi
Hajnalka Hegedűs†
Zoltán Horváth
András Juhász
Attila Kispitye
Tamás Kozsik
Lehel István Kovács D.
Richárd Legéndi
Tamás Marcinkovics
Attila Rajmund Nohl
Judit Nyéky-Gaizler
Gábor Páli
Zoltán Porkoláb
Gábor Pécsy
Máté Tejfel
Szabolcs Sergyán
Balázs Zaicsek
Viktória Zsók

Advanced Programming Languages

Editor
Judit Nyéky-Gaizler

The book is supported by the TÁMOP-4.1.2.A/11-1/1 project.

Editor:
Judit Nyéky-Gaizler

Layout editor: Attila Kispitye
ISBN:

Contents at a Glance

Table of Contents

Introduction

Programming languages are thought by many to provide as a notation form for program description. This view does not take into account – or does not even know –, how *high level* or *user-centered languages* can also aid in managing program complexity. Different languages with their possibilities suggest different programming approaches, so the common practice, which is still used nowadays in many places, is highly dangerous, when programming methodology is taught through particular programming languages, not independently from them – this could only lead to narrow concerning all the programming possibilities.

The goal of programming is to produce a *good quality software product*, so the education of programming must start with the general definition of the task and its solving program [ÁF83]. Then based on this principle, the different concrete language tools should be acquainted to the programmers, which support the implementation. However, as it is questionable to teach the methodology through particular concrete programming languages, it also leads to a dead end, if the used programming language is said to be not important for the sake of the methodology. This is – as described by Bertrand Meyer [Mey00] – like "a bird without wings". The idea is inseparable from the possibilities of formulation. It is not a coincidence that in programming no single language has become dominant, nor that always newer programming languages are designed, which support even more the adaptation of different methodological concepts and requirements into practice.

Designers of programming languages must deal with three problems [Hor94]:

- The representation provided by the language must fit the hardware and the software at the same time.
- The language must provide a good nomenclature for the description of algorithms.
- The language must serve as a tool to manage program complexity.

Aspects of software quality

The software is a product, and as for every product, it has – as defined by many ([Mey00, Hor94, Lis96]) – different quality characteristics and requirements. One of the most important goals of the programming methodology is to specify a theoretical approach for creating good quality program products. The design and the evaluation of already existing programming languages are definitely influenced by methodological considerations.

In the following, characteristics of "good" software will be discussed according to the work of Bertrand Meyer [Mey00]. Only after that will be dealt with the language features supporting the methodology – through numerous programming languages.

Software quality is influenced by many factors. One part of these – such as reliability, speed, or ease of use – are basically perceived by the user of the program. Others – such as how easy it is to reuse some parts of it for a different, but similar problem – affect program developers.

Correctness • *Correctness* of the program product means that the program solves exactly the problem and fits the desired specification. This is the first and most important criterion, since if a program is not working like it should, other requirements do not really count.

The elementary basis for this is the precise and the most complete specification.

Reliability • A program is called *reliable* if it is correct, and abnormal – not described in the specification – circumstances do not lead to catastrophe, but are handled in some "reasonable" way.

This definition shows, that reliability is by far not as a precise notion as correctness. One could say, of course with a more specific specification reliability would mean correctness exactly, but in practice there are always cases which are not covered by specification explicitly. That is why reliability is of high priority for the program product quality.

Maintainability • *Maintainability* refers to how easy it is to adjust the program product to specification changes.

The users often demand further development, modification, adjustment of the program product to new external conditions. According to some surveys 70 % of program product costs are spent on maintenance, so it is understandable that this requirement significantly affects the quality of the program. (This is relevant especially if developing big programs and program systems, since for small programs usually no change is too complex.)

To increase maintainability, design simplicity and decentralization (to have more independent modules) can be seen as the two most important basic principles.

Reusability • *Reusability* is the feature of the software products, that they can be partly or as a whole reused in new applications.

This is different to maintainability, since the same specification was modified there, but now the experience should be utilized, that many elements of software systems follow common patterns, and reimplementing already solved problems should be avoided.

This question is particularly important, not only when producing individual program products, but for a global optimization of software development, as the more reusable components are available to help problem solving, the more energy remains to improve other quality characteristics (at the same costs).

Compatibility • *Compatibility* shows how easy it is to combine the software products with each other.

Programs are not developed isolated, so efficiency can go up by orders of magnitude, if ready software can be simply connected to other systems. (Communication between programs is based on some standards, such as, for example, in Unix.)

Other characteristics • From the quality characteristics of the program product, portability, efficiency, user friendliness, testability, clarity etc. are also important to pay attention to.

Portability regards how easy it is to port the program to another machine, configuration or operating system – usually to have it run in different runtime environments.

The *efficiency* of a program is proportional to the running time and used memory size – the faster, or the less memory is used, the more efficient it is. (These requirements often contradict each other, a faster run is often set off by bigger memory requirements, and vice versa.)

The *user friendliness* is very important for the user: this requires data input to be logical and simple, the output of the results must be clearly formatted.

Testability and *clarity* are important for the developers and maintainers of the program, without these the reliability of the program cannot be guaranteed.

Aspects of software design

Some of these former requirements – the improvement of correctness and reliability – require primarily the development of specification tools. The easier it is to verify if a piece of program code is really an implementation according to the specification, the easier it will be to developed correct and reliable programs. The main role here have programming language features for specification (type invariant, pre- and postconditions) descriptions – this is supported for example by Eiffel [Mey00], by Ada 2012 [Nyék98] etc.

Implementation of another group of requirements – mainly reusability, maintainability and compatibility – can be best supported by designing the programs

as independent program units having well defined interconnections. This is the basis of the so called *modular design*. (A module here is not a programming language concept, but a unit of the design.) This question will be handled in more detail in Chapter **??**.

Our goal is to show what features the designers of different programming languages have chosen to support professional programmers in developing reliable software of good quality.

Study of the tools of programming languages

It is a natural question, why it is not enough to know *one* programming language, for what purpose it is good to deal with all the possible features of different programming languages. In the following – primarily based on the work of Robert W. Sebesta [Seb13] – we will try to summarize the advantages coming from this:

Increase of the expressive power • Our thinking and even abstraction skills are strongly influenced by the possibilities of the language used. Only that can be expressed, for which there are words. Likewise during program development and designing the solution, the knowledge of diverse programming language features can help programmers to widen their horizon. This is also true if a particular language must be used, since good principles can be applied in any environments.

Choosing the appropriate programming language • Many programmers have learnt programming through one or two languages. Others know older programming languages which are now considered obsolete, and they are not familiar with the features of modern languages. This could result in not selecting the most appropriate language if there would be more programming languages as options to choose from for a new task – since they do not know the possibilities the other languages could offer. If these programmers would know the unique features of the available tools, they could make considerably better decisions.

Better attainment of new tools • Quality programming requires continuous learning. Newer and newer programming languages will appear. The more the basic elements of the programming languages are known, the easier it will be to learn and keep up with progress.

In our book most examples are in Ada, C/C++ or Java language for certain language constructs, there are only a few chapters (except of course those about logical and functional programming) where these languages are not referenced in almost every paragraph.

Our book is aimed at facilitating primarily, the studies of university and college students to learn about programming languages, and to help the work of IT and computer specialists. Some degree of knowledge of informatics is a

prerequisite to fully understand our book: readers must have already solved some programming tasks on some programming languages.

Acknowledgements

The authors wish to thank for the support of TÁMOP tender on developing teaching materials.

We also would like to thank Zoltán Horváth, the dean of Faculty of Informatics at University Eötvös Loránd for permitting the usage of the infrastructure of the Faculty of Informatics. Without his kind contribution this work could not have been completed.

We would like to thank the generous assistance of the PhD students who helped us with their feedback to improve this new edition of the book.

In such a voluminous book – despite all the best efforts of the authors and the editor – there could be errors. We would like to ask You, dear reader, if such an error is found, please notify us via email addressed to *proglang@elte.hu*. We also welcome every kind of constructive criticism.

1 Scope and lifespan

Whichever programming language and environment we use, whichever paradigms they adhere to, whichever possibilities and features they offer, the scope and lifespan of the variables, the functions and the types are important notions.

The scope of an identifier is the part of the source code from where a language element (an object, a function, a type, etc.) is accessible with the given name.

The lifespan of an object is the part of the program's running time between the creation and the disposal of the given object, that is, when the object is live, is usable.

Scope and lifespan are related concepts, but their meanings do not overlap.

The principle of information hiding (so that the parts optimally connect) is essential in bigger projects, which involves the close cooperation of a number of programmers. The writer of a part should know about the part written by the other programmers only to the necessary extent: interface. There should be no unnecessary or accidental dependence between the parts as it encumbers human work, causes complications and possibly errors. When working on a program in collaboration with others, it would be annoying to find out that the name i (our favorite one for a loop index) can not be used any more, because another programmer was faster and already used it inside another module for some other purpose. Our own program can become unfamiliar after some while, so we can become "another programmer" in this sense.

The principle of information hiding is supported to different extent with different degrees of sophistication by the different programming languages.

A program uses different elements, e.g. types to describe common data structure and behavior, functions to encapsulate and reuse execution, and "objects" – things stored in the memory. These elements may have zero or more names: identifiers which can be used to refer to them. Usually such elements have names which are directly used to access them, but it is also possible that an object has no name at all and is accessible only indirectly (that is, with the help of some language construction, like pointers), or it can be accessible in both ways. Examples for only indirectly accessible objects are dynamically (that is, in runtime) allocated objects and elements of arrays. Dynamically allocated objects are usually accessed through pointers or references, an element of an array is usually accessed with an expression combined from the array name and the actual index.

Identifiers must follow specific syntactical rules, the exact details depend on the language, environment, and on compilation flags. In most programming languages, however, it is safe to use alphanumeric characters, usually started by a letter and continued by letters or numbers. In many programming languages identifiers are case sensitive, and the number of significant characters in an identifier may be limited. Thus, it is always always useful to check for the exact rules of identifiers in the specific programming language.

Identifiers are important resources for the programmer. A well-chosen identifier helps to understand the role of the programming element, like an interface function or a type name. A wrong identifier can be misleading and may deceive the reader of the code. Therefore for those identifiers which are accessible in many places of the program it is worth choosing longer, and telling names. Only for identifiers used locally do we tend to choose short names. A typical loop variable for example is called i – its usage context explains its role. A possible rule of thumb can be the broader the scope, the longer the name.

Identifiers can be reused, i.e. the same identifier may refer to different elements of the program in different parts of the source code. The part of the source where from an identifier can refer to a particular element of the program is called the *scope* of the identifier. The scope – or visibility – rules of an identifier are language specific; however, modern programming languages share many common patterns when defining scope rules.

During program execution we use different memory locations to store our objects. These memory locations are not necessarily needed for the whole execution time and modern programming languages can reuse unused memory parts. The allocation of the memory should precede the use of the object and we we must not refer to the memory after we have abandoned it. There is a specific time interval of the program execution when we rightfully refer to that memory area as the storage place of a certain object. This interval is called the *life* of the object. Referring to a memory area out of the object's life time may lead to invalid results or even run-time errors.

1.1 The types of memory storage

Programming languages typically define some abstract model to describe the proper behavior of the language. The description includes the memory storage model, i.e. how objects are mapped to the physical implementation, and how long they are accessible for use. The memory storage model is an important aspect of the scope and life of the objects.

1.1.1 The static memory

Static memory is sized and allocated at compile time, when also an initial value (defined by the programmer or default) is assigned. This memory will be available during the whole execution of the program.

The main advantages of the static memory is its simplicity. Static memory usually requires no runtime maintenance (object construction and destruction in object-oriented languages may be exceptions). However, this simplicity is also a drawback. Static memory is allocated for the whole execution time, even if its contents are used only in a fraction of the execution – this is rather uneconomical.

Besides this, they have certain drawbacks in a multithreaded environment as well – as they are shared between threads their access must be protected.

Static memory is primarily used for global variables – thus, sharing information between modules or subprograms during most of the execution time.

The executable code of the program may also be considered a static value, even if it is not manipulated in the same way as the data. The environment (the hardware and the operating system) might support the separation of the read-only and the read-write parts. It is of advantage to store the code of the program in a read-only part, so that an already running copy of a program may share it with a newly started instance, albeit at the price of preventing the self-modification of the code.

1.1.2 The automatic memory

We often use objects with a restricted lifetime: a loop variable is usually required only during the execution of the loop; math computations apply variables to store the temporary results before further use, etc. It would be a serious waste of resources to use static memory for such purposes. Instead, block-oriented languages use *automatic* memory to reserve storage when entering a block and keep that storage valid until leaving the block. The name automatic – or auto in short – comes from the fact that no programmer action is required to mark the beginning and the end of the life of such objects.

Automatic storage is usually implemented by a *stack*, a LIFO (last-in, first-out) data structure. When entering a block, the stack pointer identifying the top of the stack is incremented by the size of the required automatic variables - thus "allocating" memory. When leaving the block, the stack pointer is set back to the initial value to "free" the automatic storage. This position is also stored in the stack. As usually no other action happens simultaneously automatic variables are uninitialized – they may contain the "garbage" of the earlier content of the memory. In some object-oriented languages, however, constructor and destructor calls may be associated with these actions.

In many cases the function call mechanism are also implemented by using this stack. If recursion (that is, the ability of a function to directly or indirectly call itself) is not supported by a language and an environment, maximum one copy of a function may be active at any time. This may it is possible to store the data necessary for the calling of the function in the static, (that is, at compile time) preallocated memory. If, however, recursion is to be supported, as is generally the case, the data necessary at runtime for the calling of the function (parameter values, return address, register values) are stored in the stack. The stack area corresponding to a function call is called an *activation record*. As functions call each other their activation records build up on top of each other.

In multithreaded execution environments, each thread has its own stack, meaning, automatic storage variables are thread locals.

It is especially dangerous to refer to an object with automatic life after its lifetime – i.e. after leaving the block. The memory address will be still valid (therefore, no run-time error will occur), but it may happen that the value of that memory area could be already modified – perhaps an other block is using that activation record and our value has been modified by then.

1.1.3 Dynamic memory

There are situations when neither the static nor the automatic storage suits our purpose. This is the a case is when we need a storage with a shorter lifetime which still has to leave the block of creation, or when we do not know the size of the object in compilation time. In these situations we use *dynamic* memory.

In the case of dynamic memory the programmer is fully responsible for the lifetime control of the object. Dynamic memory handling is either supported directly by language features (e.g. the **new** operator of C++ and Java, and *delete* in C++), or indirectly by (standard) library functions (e.g. C's *malloc* and *free*). The allocation of such memory is based on the explicit request of the programmer: they typically use a **new** expression or initiate a specific function call. Allocation happens in a data structure called *heap* or *free memory*. The allocation process looks up an unused continuous area, administers its use and returns a pointer to it. The deallocation process takes the pointer to an allocated area on the heap and marks it free with some additional actions (like concatenating free neighboring areas).

In some implementations frequent allocations and deallocations may fragment the heap. Moving allocated fragments would invalidate pointers or references referring to the area, thus it is impossible in many languages. In the *managed heap* of the .NET system, however, allocated fragments can be moved and references are updated. A Java Virtual Machine can be implemented in many ways, and memory handling and garbage collection are eminent subjects to steady research, development and refinements. Oracle's HotSpot Virtual Machine includes several generational garbage collectors, which rank the objects into different generations and store them in several heap areas and move them around. The gory details actually do not have to do with the core language features and the themes discussed here. Due to them being (more or less) implementation details, we, as programmers are not affected by them and we are protected from their complexity. However, when we want to fine-tune the JVM to improve performance and to eliminate possible bottlenecks, and pondering about picking a garbage collector and customizing its parameters, we still need to deal with them.

While heap allocation actions are fully controlled by the programmer, deallocation may happen either upon the programmer's request (*free* function call in C, *delete* expression in C++), or can be automatically controlled by the *garbage collector* (ADA, Java, C#).

Heap allocation and deallocation are expensive operations in execution time and in possible lock conflicts. Therefore, heap allocations should be minimized in performance critical programs.

Although dynamic memory is quite flexible, the handling of dynamical data may lead to some errors still:

- memory is allocated but never eventually freed (memory leakage);
- several attempts are performed to free the same memory area;
- a pointer or reference is held to an already freed and possibly (for other purposes already) reused memory;
- everything is done correctly but the memory still gets exhausted.

Dynamic memory handling is not supported by every language and every environment. For example COBOL was not designed with it in mind, and even if some versions and implementations support it, it is a foreign feature to its logic and its culture.

1.1.4 A simple example

Let us consider a simple and admittedly contrived example to compute the third member of the Fibonacci series in a highly inefficient way, which however illustrates recursion and variables in different storage areas. The Fibonacci series was invented by Leonardo Fibonacci in 1202. For purposes of demonstration let us assume the rabbits are immortal (at least in the interval being discussed), and a pair of rabbits produces (the female gives birth to) a pair every month after the age of one month. From these assumptions it follows that the number of rabbit pairs in a given month is the sum of the number of the rabbits in the previous month plus the number of the newly born pairs, with the latter being equal to the ones living two months ago. If $F(n)$ denotes the number of the rabbit pairs after n months, then $F(n) = F(n-1)+F(n-2)$. An interesting series (occurring at several places in mathematics including the analysis of algorithms [Knu81]) is defined this way.

The series is defined in itself: a value can be computed from the previous ones. A widely used representation of this idea is computing the *fib* function in a recursive way. The example seen in Figure 1.1 is written in C++, but it is in a procedural, – in fact – C style. We chose the language C++ rather than C, because of the relatively easy programming of the output due to *std::cout*.

In a given environment the following output is generated:

```
1 fib ++ 3 0xbffff708
2 fib ++ 2 0xbffff6d8
3 fib ++ 1 0xbffff6a8
3 fib -- 1=1
4 fib ++ 0 0xbffff6a8
4 fib -- 0=1
4 fib -- 2=2
```

```
#include <iostream>
int ct;
int fib (int what)
{
    int ret = 1;
    ct++;
    std::cout << ct << " fib ++ " << what << " " << (void *)&ret << std::endl;
    if (what > 1) {
        ret = fib (what - 1) + fib (what - 2);
    }
    std::cout << ct << " fib - " << what << "=" << ret << std::endl;
    return ret;
}
int main ()
{
    int x = fib (3);
    std::cout << "the result is " << x << " in " << ct << " step(s)." << std::endl;
    return 0;
}
```

Figure 1.1: Fibonacci series in C++

```
5 fib ++ 1 0xbffff6d8
5 fib -- 1=1
5 fib -- 3=3
the result is 3 in 5 step(s).
```

In order to compute *fib(3)*, we need the values of *fib(2)* and *fib(1)*, so there are several active calls to *fib* with different parameters, and local *ret* variables. The address of *ret* is output so as to demonstrate it. The calls of the function *fib* are counted in the global variable *ct*.

1.2 Scope

The scope of an identifier is that part of the source code, where the denoted element (variable, process, type etc.) can be accessed with the given name.

Scopes include the following areas:

- the entire program;
- a compilation unit (see below);
- a subprogram;
- a block of code;
- a type (a class) or a namespace.

Different scopes may of course, overlap: a given point of the program can (and often does) belong to more scopes. Scopes usually contain each other: a point in a function belongs to the scope of every containing block, to that of the function, and to the global one; in object-oriented environments the scope of a type (a class) can be part of those of others due to inheritance.

```
extern int x;                    /* global x is defined somewhere else */
#ifdef __cplusplus
namespace N {
    int x;
    void f();
}
#endif
int foo(int y)
{
    if (y == 0) return x;        /* global x. */
    {
        int x=2;
#ifdef __cplusplus
        if (y > 0) return ::x;   /* C++: global x */
        N::x++;                  /* C++: x from namespace N. */
#endif
        if (y < 0) return x;     /* 2 */
        {
            int x = 3;
            return x;            /* 3 */
        }
    }
}
#ifdef __cplusplus
void N::f()
{
    x++;                         /* denotes N::x. */
}
#endif
```

The resolution of a name, that is, the search for the element denoted by a name (identifier) is performed among the scopes beginning at the narrowest proceeding towards the broadest one. So a declaration of a name in a closer scope hides the names declared in a further, wider scope.

However, names can be overloaded, that is, it is possible for a name to denote several different things at the same time. Obviously, this only applies when the denoted language elements have features that set them apart, apart from the name itself.Languages which support static (compile-time) type checking, functions may be distinguished by the signature, that is, the number and types

of the arguments, so functions with the same name but with different signatures can be handled:

```
void print(int i);
void print(char *s);
void print(char c);
```

ANSI C performs type checking, but does not support the name overloading in the above sense. During the resolution of an overloaded function name ADA takes the return type into consideration, but C++, C#, and Java do not.

C++ supports the scope operator ⟨*<name space name>*⟩*::*⟨*<object>*⟩ or ⟨*<class name>*⟩*::*⟨*<object>*⟩. We have seen the example *std::out* denoting the *out* object (handling the standard output) of the namespace *std* of the standard library objects and functions. Special version of scope operator with empty left operands *::x* denotes the global *x* variable.

1.2.1 Global scope

Global variables are accessible from every point of the program and reside in static storage. They enable easy communication among different parts (modules, functions) but can cause difficult dependencies among them. If a function modifies global variables, these changes are called the function's side effect. A function can be considered as an encapsulation, a "short hand" for an arbitrarily long and complicated sequence of instructions. Side effects contradict this to some extent.

In C and C++ the global variables are the ones defined outside of every function, namespace or class, and are not marked with the **static** keyword. In FORTRAN the directive *COMMON* can be used to define global memory areas which can be broken into variables in different ways in the different modules.

1.2.2 Compilation unit as a scope

It is important to break programs of at least middle size into independently compilable parts, so that after a change the whole program need not be compiled. Different languages support this with the introduction of different units: partitions, modules, compilation units, library modules, submodules. In large projects it is an explicitly stated design goal to brake programs down into such smaller units[Lak96].

A compilation unit (if the programming language supports it) is the set of the functions and variables put into the same source file and compiled together. (Java does not support this notion: the source of a public class or interface can be only in the source file named after it, and the byte codes of the classes are written into their own **.class** files, even those of the anonymous inner classes.) It is language-dependent how the functions and variables in a compilation unit are accessible from outside.

The variables and functions in the compilation unit deemed only locally accessible, that is, accessible only within the compilation unit, may be put into its scope.

Some languages (Pascal, Ada, C and C++) distinguish syntactically between declarative and implementing parts, others including Java do not. While in the language family of Pascal (Pascal, Ada) and in Java the compiler takes the necessary information from the object or byte codes of the referred units, in the languages C and C++ special source (header) files serve this purpose.

In C and in its successor C++, the keyword **static** denotes which variables declared outside every function make up the scope of the compilation unit. The storage class will be static, and the scope will be confined to the compilation unit. This heritage from C is a way of modularization. It is a rather unfavourable situation as it ties the question of accessibility (design level) to the question what is put together into the same compilation unit (implementational level).

C++ introduces the notion of the unnamed namespace to define a scope local to a compilation unit. Its usage is equivalent but still it is preferred to the static variables.

The variables in the compilation unit serve as a storage for the state of the module (that is, the set of functions in the compilation unit), since the functions share the variables preserving their values across the function calls. The names of the static variables and functions cannot be seen from outside the compilation unit, meaning they are reusable.

1.2.3 Functions and code blocks as scope

Static or automatic variables can have a function (method, subprogram) or in the case of block structured languages a code block as their scope. (The body of a function can be regarded as a code block anyway.) These variables are called local to the function or to the code block. Variables declared in a block are accessible from further blocks inside. From a block of course the locally declared and the global variables are also accessible, and the ones declared in the containing blocks and in the containing function, compilation unit too.

The static variables declared inside a function can be used as storage retaining the value across function calls with the restricted scope of the code block or function. A similar role is played in ALGOL 60 and SIMULA 67 by the local variables marked with the keyword *own*.

If not declared as **static**, the variables declared in a function or a code block become automatic, that is, they are put into the storage automatically allocated at entering and destroyed and possibly reallocated at leaving. The parameters of a function also belong to its scope.

1.2.4 A type as scope

Types and classes are scopes as well. Here the keyword **static** has a special meaning: static members belong to the class and not to the instances. Static data members are of static storage. Instance (non-static) functions access both static and instance functions and data members simply by the name, whereas static functions access only the static ones.

In C++ and in Java both functions and data members can have access modifiers:

- Private data members and functions can only be accessed from within the same class;
- Protected data members and functions can be accessed from within deriving types;
- Public data members and functions can be accessed from everywhere.

Java also applies the default semi-public accessibility without a keyword, meaning accessibility from the same package.

In C++ the private data members and functions of a type can however be accessed from a type declared to be its **friend**. In Java there is no friendship: private members are accessed only from the given class and its internal classes (besides reflection magic).

Inheritance has consequences regarding the scope: the functions of a derived type access both the data members and the functions of the base type as if they had been declared in the derived type, with the exception of the private ones.

1.3 Lifespan

One of the most basic properties of a variable (a memory object) in any programming language is its *lifespan*. The lifespan of an object is the part of the program's running time between the creation and the disposal of the given object, that is, when the memory allocated to the object stores its value. Lifespan is an important notion in traditional (procedural, structured) and in object oriented languages as well.

1.3.1 Creation and destruction of objects

When an object is created, as a first step, memory is allocated to the object. Then this memory area must be initialized: it must be prepared for use, that is, the object's invariants must be set, the object must be given a consistent internal state. In object-based languages there is a special function for this latter purpose, i.e. the constructor belonging to the given type, whose name is usually identical to that of the type. Due to overloading there can be several constructors with different parameter lists. Initializing functions may, of course, be used

in procedural languages: the API may require that a structure be initialized before the first use, but if there is no language-imposed guarantee, the compiler cannot enforce this, so erroneous usage is possible. Object-based languages, however, guarantee that an object is created only in a controlled manner, that is, essentially with the assistance of an appropriate constructor.

Allocating memory is the duty of the runtime environment, searching for and calling the appropriate constructor is up to the compiler, and the correct coding of constructor (and the destructor) is the responsibility of the programmer.

The constructor is neutral with regard to the storage type: it is not aware where the memory area to be initialized was allocated. A constructor is used to create an instance of a given (exactly known) type. In an object-oriented environment (that is, supporting inheritance) it is possible that only a parent type (implementing some given interface) is interesting, not the exact type of the instance. So some object (of the "factory" type) might be able to provide an instance of the given interface, which might actually be of an inheriting, derived type. The very method of the factory, however, cannot be storage-type neutral: it must create the object in order to able to return a reference or a pointer to it.

A typical example for this situation is provided by the class *DriverManager* in the package *java.sql* of the Java JDBC API. Its method

public static *Connection getConnection(String url)* **throws** *SQLException*

returns an object implementing the interface *Connection*, if it finds a dynamically registered driver class which is willing and able to provide one. If a non-static factory method is used, it can be polymorphic, making room for more complicated design patterns.

Constructors are special ones among the methods due to their relation to memory allocation. Therefore in C++, where there generally exist pointers to the methods, there are none of the constructors.

When an object is destroyed, and if external resources (files, locks, descriptors, handles, dynamic memory) have to be freed, first some house-keeping activities might be necessary. This is followed by the freeing of the memory allocated to the object. The latter is the chore of the runtime environment, but the former is the duty of a special method pertinent to the type, the so called destructor. In C++ its name is that of the type prefixed with a tilde (~) to allude to its being complementary to the constructor.

The Java language was designed with garbage collection in mind, therefore there is no destructor mechanism, but a *finalize* mechanism. The Java virtual machine stores the objects – that is, those of class (non-primitive) types – on a *garbage collected heap*, even if the reference variable is static. Memory eligible to *garbage collection* (that is, not reachable through any active path of reference chains) is detected, than finalized (that is, the *finalize* method is run) and freed typically in a low priority thread. The *finalize* method can be used to free resources before the object is lost and its memory is freed. Java does not define, however, when the virtual machine will perform garbage collection,

and finalizing. This is in contrast to the C++ language, where the destructor is immediately executed at the end of the object's life span. As a long waited enhancement, in its version 1.7, Java introduced the try-with-resources statement, which guarantees that for every resource declared within the statement, the appropriate *close* method will be executed upon leaving the statement.

1.3.2 Static

Static variables are usually created at program start and persist while the program runs. The order of the creation of the variables inside a module (compilation unit) follows naturally from their declaration order.

In Java, variables of primitive types and object references can be static, but the objects themselves are created only dynamically on the heap. In Java the variable called *staticObject* of the following class is assigned a value in the static block running at class load time:

```
public class ClassWithStaticObject {
    static int staticObject[] = { 123, 456 };
}
```

In C++ static variables declared within a compilation unit are initialized in the declaration order and destructed in the opposite order at the end of the program. Many C++ programmers (mis-)translate this as "constructors run at the very beginning of the *main* function and destructors run at the end of *main*". This is a serious (and potentially) dangerous oversimplification.

Firstly, local static variables have static storage, but their constructor runs only when the program execution reaches their declaration for the first time. This can happen well after *main* started. It is even possible that the control *never* reaches the declaration – meaning, the object will never be constructed. Such a non-constructed object naturally will never execute its destructor, which reveals that the C++ language should register the construction of local statics under runtime.

Secondly, there are serious problems with the initialization order of global static variables. Although the order of construction of static variables declared within a compilation unit is well-defined, the C++ standard says nothing about the order *between* compilation units. Unfortunately, many C++ programmers ignore this problem, naïvely thinking that static objects are not accessible before the *main* function. However, it may happen (and normally *does* happen) that one of the static object's constructor tries to access an uninitialized static object from an other compilation unit. This can lead to nasty, hard to debug errors.

This static initialization problem can be avoided using the *singleton* pattern: we encapsulate our statics into dynamically created objects.

1.3.3 Automatic

Automatic variables are stored on the stack along with the data needed for the runtime support of procedure calling in an automatically allocated area – hence the name. At leaving the function, this area is automatically freed. In most cases it is dangerous if a variable containing a pointer to some area has a broader scope than that of the area it points to. In the following example, the function returns a pointer to an area which is freed at the moment of the function's returning and just waits to be overwritten:

```
/**** BAD CODE! ****/
char *gettime()
{
    char wtime[24];
    /* fill wtime */
    return wtime;
}
```

C++ keeps track of the automatic variables and calls the destructors (in the opposite order of creation) if control is about to leave a function or a block for any reason like reaching a **return** statement, falling through the closing brace or due to an exception being thrown.

1.3.4 Dynamic

There are languages where the programmer can control the allocation of memory. The lifespans of these dynamical memory areas stretch either until the programmer explicitly frees them (irrespective of the overall structure of the program) or until all references go away and the areas became unreachable. This latter requires support from the runtime environment: "garbage collection". Some languages and environments were designed to include garbage collection (SIMULA 67, Ada, Java, C#), others either do not have it at all or have it as an optional feature, not as part of the specification. Garbage collection not only requires runtime efforts, but it can pose a principal problem. If the language supports pointers which can be manipulated, the value of a pointer to dynamically allocated memory can be hidden in the program. The following code snippet illustrates a situation when the address of a newly allocated piece of memory (originally stored in the variable *pt*) is no longer available in our program, and thus the garbage collection takes this memory as unreachable and as eligible to garbage collection, nevertheless the pointer is clearly reproducible from the values in *pointer0* and *pointer1*:

```
char *pt = new char();
char pointer0[sizeof(pt)];
char pointer1[sizeof(pt)];
*(char **)pointer0 = pt;   // Bitwise copy of pt.
*(char **)pointer1 = pt;   // Bitwise copy of pt.
pointer0[0] = 0xab;
pointer1[1] = 0xcd;
pt = 0;
```

Garbage collection frees the programmer from the burden of freeing the memory by detecting unreachable objects. However, keeping references to an object does prevent it from becoming unreachable, that it, eligible to garbage collection. Unintentional references can be kept in callback handlers, queues etc. under the hood. This innocently looking code runs to memory exhaustion (to an *OutOfMemoryError*) for example in SUN's JVM v1.4 (but fortunately no longer in v1.5):

```
public class a {
  public static void main( String args[] ) {
    while ( true ) {
      Thread t = new Thread();
    }
  }
}
```

In the infinite loop *Thread* objects are just created (but not started) and the variable holding the reference is reused, and the Threads become seemingly unreachable, so in theory they could be garbage collected and the memory reclaimed. But the particular *Thread* implementation seems to keep track of the unstarted threads obviously retaining references to them, hindering their becoming unreachable and garbage collected. Similar unintentional retention of references can occur in purely user-supplied code as well.

If a reachable reference exists to an object, this makes it reachable and not eligible to garbage collection. The so called *weak references* (as opposed to the common, strong ones) yield a weaker level of reachability not preventing the object's becoming unreachable in the common sense and garbage collected, in which case, of course, the encapsulated strong reference can not be used any longer. For example, Java (since v1.2) defines three levels of weak reachability listed in diminishing order of strength: *soft*, *weak*, and *phantom*.

1.4 Examples

Usage of a static buffer • Let us assume an operating system storing the creation and last access dates of the files in some machine- but not user-friendly form, the

number of the elapsed seconds since some given starting date. For this purpose let *time_t* denote the integral type[1]. Obviously a function will be necessary to convert a given *time_t* value into some human-readable form. Let us call this function *ctime*. When using this function, the question arising who will allocate the memory for the human-readable form. The caller cannot know how much space will be required, so the simple solution is the called function allocating it in a static area. The C standard library *ctime* function does exactly this: it returns a pointer to its static buffer. The function declaration as seen in the header file *time.h*:

```
char * ctime(const time_t *tp);
```

This is the simplest solution but it comes at a price. If we want to print the time before and after a long-running portion of code, the following "solution" will not work. A hint: *time(NULL)* returns a *time_t* value representing the current time.

```
char *before;   char *after;   time_t bt, at;
bt = time(NULL);
before = ctime(&bt);     /* 1 */
long_running_function();
at = time(NULL);
after = ctime(&at);      /* 2 */
printf("%s %s\n",before,after);
```

This "solution" does not work, because the second call to *ctime* reuses the same buffer, so the value stored during the first call is overwritten. Copying this value before the second call would alleviate the problem.

The usage of static buffers is even more problematic in *multithreaded* environments because a function might be called by several threads at the same time.

Therefore, the usage of static buffers is better avoided. A possible solution is the caller' supplying the necessary buffer and taking care of the allocation and freeing:

```
void ctime1(char *buffer, const time_t *tp);
```

Or we may want to pass along the buffer size:

```
void ctime2(char *buffer, int bufflen, const time_t *tp);
```

The most comfortable solution would be the object-oriented approach, provided it is supported by the given environment.

[1] An example of it is the UNIX operating system, where the starting time, the so called Epoch, is 1 January 1970. The readers might remember the "Y2K bug" from year 2000. A similar critical situation will arrive on 29 January 2038, when the maximal signed 32bit integer values will reach their upper limits in UNIX systems. Hopefully, many of our readers will experience the "Epoch 0x7fffffff bug".

Resource management through objects • Object-based management of resources is well supported by the constructor-destructor mechanism of C++. This is the idiomatic (that is, best fitting the language C++) way of creating and freeing of resources. Let us consider this example:

```
void bar();
void foo()
{
    char *pt = new char[1024];
    bar();
    delete[] pt;
}
```

It seems to be correct: at the beginning of the process, a char array is allocated, and before returning it is carefully freed. (The pair of brackets `[]` after **delete** denotes our intention to free an array, not a single object.) What happens however if the called function throws an exception? The flow of control leaves the function *foo* without the array's being freed. Since *pt* is the only pointer to this memory area, it cannot be freed any longer. The unexpected exception disturbs our program's behavior: it is not *exception-safe*. We can patch our program by catching the exception:

```
void bar();
void foo()
{
    char *pt = new char[1024];
    try {
        bar();
        delete[] pt;
    } catch (...) {
        delete[] pt;
        throw;
    }
}
```

A similar example in Java is more elegant due to the try-finally pair of blocks. Code written in the *finally* block is always executed, regardless of whether the exception has been thrown.

```
void foo() {
    try {
        get_resource();          // allocation of the resource
        bar();                   // may throw exception
    } finally {
        release_resource();      // releasing of the resource
    }
}
```

An even better solution of the C++ example is to encapsulate the resource into a single object, making use of the automatic insertion of the destructor calls for the automatic objects falling out of scope:

```
class mypuffer {
    char *pt;
    mypuffer(const mypuffer&);          // forbid copying
    mypuffer& operator=(const mypuffer&); // forbid assigning
public:
    mypuffer(int size) { pt = new char[size]; }
    ~mypuffer() { delete [] pt; }
};
void bar();
void foo()
{
   mypuffer a(1024);
   bar();
}
```

This technique is called *Resource Allocation Is Initialization* or *RAII*.

Fortunately, we do not have to reinvent the wheel. In C++ a whole branch of *smart pointers* of the standard library (e.g. the class templates *unique_ptr*, *shared_ptr* and others) serves the purpose of the exception-safe handling of automatic pointers. A smart pointer wraps a raw pointer, and it can be used similarly to pointers. When destroyed, it also destroys the object pointed to.

The *auto_ptr* defined in the earlier C++ standard has a number of issues, thus its usage is better avoided.

1.5 Summary

Scope and lifespan are two related but not overlapping major concepts in programming languages. The scope of an identifier defines the section of the source code from where a named language element (an object, a function, a type, etc.) is accessible using that identifier. Scope categories exist from local scope to class, namespace, compilation unit, and all program wide visibility with many variations in different programming languages. As well-chosen names are essential

resources, programmers should select the adequate scope category to manage the program's identifiers and to help the compiler detecting possible errors.

The lifespan (or simply: life) of an object is the part of the program's running time between the creation and the disposal of the given object. Lifespan categories are typically determined by the storage types used by the programming language. Most programming languages use automatic, static, and dynamic storage. Automatic objects are constructed when the program control entering the block where they have been declared and are disposed when the control leaves the block. Static storage (with global, namespace or even local scope) is allocated at the beginning of the program (although details vary in different languages) and remains available during the whole run of the program. In case of dynamic storage, allocation and deallocation are under the control of the programmer, although proper resource handling is sometimes supported by either a garbage collection mechanism or by language elements, like smart pointers. Modern object-oriented languages support the object's creation and disposal with user defined constructors and destructors.

1.6 Exercises

1. The languages mentioned in this chapter are compiled ones. Consider interpreted (script) languages, for example AWK, Perl, JavaScript, or Unix-shell (sh, ksh, csh, bash, zsh and so on) and examine to what extent the contents of the chapter apply to them. Check if, for example, there are non-global variables in them at all.

2. In the example on page 20 find a way for regaining the value of the pointer *pt* in the code example. Rewrite the example to avoid the use of casting.

3. Write the *ctime1* and *ctime2* functions alluded to in the *ctime* example on page 22. Give an object-based solution too.

4. Rewrite the "resource allocation and freeing" example on page 23 to include not only one, but three resource-objects in C++, and in Java before and after version 1.7, that is, without or with a **try**–*with–resources* statement.

5. Locate the code block initializing the static variables in the example *ClassWithStaticObject* on page 19.

```
javac ClassWithStaticObject.java
javap -private -c ClassWithStaticObject
```

6. Test the following C language example with various numbers of command line parameters and discuss the output:

```
#include <stdio.h>
int foo (int i)
{
    if (i <= 0) {
        /* deliberately with no initial value */
        int j;
        return j;
    }
    return foo(i - 1);
}

int main (int argc, char *argv[])
{
    printf ("%3d %8x\n", argc, foo (argc));
    return 0;
}
```

1.7 Useful Tips

1. As the authors do not know the Reader's preferences in the script languages, it is left to the Reader to pick one language and analyze it from this point of view.
2. There are three different solutions to this problem.
   - Using cast as above;
   - Copying raw bytes using the standard library *memcopy*function;
   - Using union.
3. In the object oriented version you can implement the buffer as the member variable of the class.
4. Do not forget to forbid the copy of RAII object in the C++ solution. In java use *AutoCloseable* interface.
5. Compile the class and then use the Java class file disassembler tool *javap* with the *-private* option.
6. Recall the different memory models and initialization strategies.

1.8 Solutions

1. Left to the reader.
2. This file contains a solution for casting, copy memory and using union.

```
#include <stdio.h>
#include <string.h>

void foo(void); // test funcion with casting
void bar(void); // test funcion using memcpy
void baz(void); // test funcion using union

int main(int argc,char *argv[] )
{
    foo();
    bar();
    baz();
}
void foo() // with casting as in the book
{
    char *pt = new char();
    char pointer0[sizeof(pt)];
    char pointer1[sizeof(pt)];
    *(char **)pointer0 = pt; // Bitwise copy of pt.
    *(char **)pointer1 = pt; // Bitwise copy of pt.
```

```
    printf("%d\r\n",__LINE__);
    printf("%p\r\n",pt);
    pointer0[0] = 0xab;
    pointer1[1] = 0xcd;

    pt = 0;

    printf("%p\r\n",pt);
    pointer0[0] = pointer1[0];
    pt = *(char **)pointer0;
    printf("%p\r\n",pt);
#ifdef FREE
    delete pt;
#endif
}
void bar() // using memcpy
{
    char *pt = new char();
    char pointer0[sizeof(pt)];
    char pointer1[sizeof(pt)];
    printf("%d\r\n",__LINE__);

    memcpy(pointer0,&pt,sizeof(pt));
    memcpy(pointer1,&pt,sizeof(pt));
    printf("%p\r\n",pt);
    pointer0[0] = 0xab;
    pointer1[1] = 0xcd;
    pt = 0;

    printf("%p\r\n",pt);
    pointer0[0] = pointer1[0];
    memcpy(&pt,pointer0,sizeof(pt));
    printf("%p\r\n",pt);
#ifdef FREE
    delete pt;
#endif
}

void baz() // using union
{
    char *pt = new char();
    union
    {
        char *pt0;
        char pointer0[sizeof(pt)];
    };
    union
```

```
    {
        char *pt1;
        char pointer1[sizeof(pt)];
    };

    printf("%d\r\n",__LINE__);
    printf("%p\r\n",pt);
    pt0 = pt;
    pt1 = pt;

    pointer0[0] = 0xab;
    pointer1[1] = 0xcd;
    pt = 0;

    printf("%p\r\n",pt);
    pointer0[0] = pointer1[0];
    pt = pt0;
    printf("%p\r\n",pt);
#ifdef FREE
    delete pt;
#endif
}


/* compilation and results:

$ g++ -dumpversion
3.4.6

$ g++ *.cpp

$ ./a.out
19
0x99e8008
(nil)
0x99e8008
37
0x99e8018
(nil)
0x99e8018
63
0x99e8028
(nil)
0x99e8028

$ g++ -DFREE *.cpp
```

```
$ ./a.out
19
0x83b6008
(nil)
0x83b6008
37
0x83b6008
(nil)
0x83b6008
63
0x83b6008
(nil)
0x83b6008

$ g++ -dumpversion
4.1.2

$ g++ a.cpp
$ ./a.out
19
0xe843010
(nil)
0xe843010
37
0xe843030
(nil)
0xe843030
63
0xe843050
(nil)
0xe843050

$ g++ -DFREE a.cpp

$ ./a.out
19
0x1b79c010
(nil)
0x1b79c010
37
0x1b79c010
(nil)
0x1b79c010
63
0x1b79c010
(nil)
0x1b79c010
```

```
//// ------------

bcc a.cpp
Borland C++ 5.5.1 for Win32 Copyright (c) 1993, 2000 Borland
a.cpp:
Warning W8057 a.cpp 11: Parameter 'argc' is never used in function main(int,char * *)
Warning W8057 a.cpp 11: Parameter 'argv' is never used in function main(int,char * *)
Turbo Incremental Link 5.00 Copyright (c) 1997, 2000 Borland

a.exe
19
00902C88
00000000
00902C88
37
00902C98
00000000
00902C98
63
00902CA8
00000000
00902CA8

bcc -DFREE a.cpp

Borland C++ 5.5.1 for Win32 Copyright (c) 1993, 2000 Borland
a.cpp:
Warning W8057 a.cpp 11: Parameter 'argc' is never used in function main(int,char * *)
Warning W8057 a.cpp 11: Parameter 'argv' is never used in function main(int,char * *)
Turbo Incremental Link 5.00 Copyright (c) 1997, 2000 Borland

a.exe
19
00902C88
00000000
00902C88
37
00902C88
00000000
00902C88
63
00902C88
00000000
00902C88
```

*

3. The C-like solution is as follows:

```
/*
We make use of the standard C functions in (time.h)
in particular of this:
size_t strftime (char* ptr, size_t maxsize, const char* format,const struc
*/
#include <stdio.h>
#include <time.h>

void long_running_function(void);
void foo(void);
void bar(void);
void baz(void);
void ctime1(char *buffer, const time_t *tp)
{
    struct tm * p = localtime(tp);
    strftime(buffer,24,"%a %b %d %H:%M:%S %Y",p);
}
void ctime2(char *buffer, int bufflen,const time_t *tp)
{
    struct tm * p = localtime(tp);
    strftime(buffer,bufflen,"%a %b %d %H:%M:%S %Y",p);
}
int main(int argc,char *argv[])
{
    foo();
    bar();
    baz();
}
void foo(void)
{
    char *before; char *after; time_t bt, at;
    bt = time(NULL);
    before = ctime(&bt); /* 1 */
    long_running_function();
    at = time(NULL);
    after = ctime(&at); /* 2 */
    printf ("%s %s\n",before,after);
}
void bar(void)
{
    char before[32];
    char after[32];
    time_t bt, at;
    bt = time(NULL);
    ctime1(before,&bt); /* 1 */
```

```
    long_running_function();
    at = time(NULL);
    ctime1(after,&at); /* 2 */
    printf ("%s %s\n",before,after);
}
void baz(void)
{
    char before[20]; // deliberately too small in order to demonstrate
    char after[20];  // the usefulness of the size parameter.
    time_t bt, at;
    bt = time(NULL);
    ctime2(before,sizeof(before),&bt); /* 1 */
    long_running_function();
    at = time(NULL);
    ctime2(after,sizeof(after),&at); /* 2 */
    printf ("%s %s\n",before,after);
}
void long_running_function(void) {
    clock_t when = clock() + CLOCKS_PER_SEC; // a second
    while( when >= clock());
}
```

The object-oriented solution in C++ contains:

- The class definition:

```
#include <time.h>

class Ctime
{
private:
   char buffer[25];
public:
   Ctime(const time_t *tp);
   const char *getBuffer();
};
```

- The class implementation:

```
#include "Ctime.h"
#include <time.h>

Ctime::Ctime(const time_t *tp)
{
    struct tm * p = localtime(tp);
    strftime(buffer,sizeof(buffer),"%a %b %d %H:%M:%S %Y",p);
}
```

```
const char * Ctime::getBuffer()
{
    return buffer;
}
```

- The test program:

```
#include "Ctime.h"
#include <time.h>
#include <stdio.h>

void long_running_function(void);

int main(int argc,char *argv[])
{
    char before[32];
    char after[32];
    time_t bt, at;
    bt = time(NULL);

    Ctime ct1(&bt);
    long_running_function();
    at = time(NULL);
    Ctime ct2(&at);
    printf("%s %s\n",ct1.getBuffer(),ct2.getBuffer());
}

void long_running_function(void)
{
    clock_t when = clock() + CLOCKS_PER_SEC; // a second
    while( when >= clock());
}
```

4. The C++ solution without RAII:

```
void bar();
void foo()
{
    char *pt = new char[1024];
    char *pt1 = new char[1024];
    char *pt2 = new char[1024];
```

```
    try
    {
        bar();
        delete[ ] pt;
        delete[ ] pt1;
        delete[ ] pt2;
    }
    catch (...)
    {
        delete[ ] pt2;
        delete[ ] pt1;
        delete[ ] pt;
        throw;
    }
}
```

The C++ solution with RAII:

```
class mypuffer
{
    char *pt;
    mypuffer(const mypuffer&);              // forbid copy
    mypuffer& operator=(const mypuffer&),; // forbid assignment
public:
    mypuffer(int size) { pt = new char[size]; }
    ~mypuffer() { delete [ ] pt; }
};
void bar();
void foo()
{
    mypuffer a(1024);
    mypuffer b(1024);
    mypuffer c(1024);
    bar();
}
```

The java solution before version 1.7

```
/*
An exception can be thrown during the allocation of a resource
We declared random checked exceptions from the java.lang package
*/
public class ex4a
{
    // this is a method that must be called on the object
    // before it gets garbage collected
    public void close()
    {
    }
```

```
// in a production code you might want to be more specific
// about the possible Exceptions
void doit()  throws Exception
{
}

// resource allocation
ex4a getResource1() throws IllegalAccessException
{
    return new ex4a();
}

ex4a getResource2() throws ClassNotFoundException
{
    return new ex4a();
}

ex4a getResource3() throws InstantiationException
{
    return new ex4a();
}

// in a production code you might want to be more specific
// about the possible Exceptions
public void foo() throws Exception
{
    ex4a res1 = null;
    ex4a res2 = null;
    ex4a res3 = null;

    try
    {
        res1 = getResource1();
        res2 = getResource2();
        res3 = getResource3();
        doit(); // do something here potentially throwing an exception
    }
    finally
    {
        if ( res3 != null )
        {
            res3.close();
        }
        if ( res2 != null )
        {
            res2.close();
        }
```

```
            if ( res1 != null )
            {
                res1.close();
            }
        }
    }
}
```

The java solution using version 1.7 features

```
public class ex4b implements AutoCloseable
{
    public void close() throws Exception
    {
    }

    public void doit( ) throws Exception
    {
    }

    ex4b getResource1() throws IllegalAccessException
    {
        return new ex4b();
    }

    ex4b getResource2() throws ClassNotFoundException
    {
        return new ex4b();
    }

    ex4b getResource3() throws InstantiationException
    {
        return new ex4b();
    }

    void foo() throws Exception
    {
        try
        (
            ex4b  res1   = getResource1();
            ex4b  res2   = getResource2();
            ex4b  res3   = getResource3();
        )
        {
            doit( ); // do something here potentially throwing an exception
        }
    }
}
```

5.
```
public class ClassWithStaticObject {
    static int staticObject[] = { 123, 456 };
}
```

```
javap -private -c ClassWithStaticObject

Compiled from "ClassWithStaticObject.java"
public class ClassWithStaticObject extends java.lang.Object{
static int[] staticObject;

public ClassWithStaticObject();
  Code:
   0: aload_0
   1: invokespecial #1; //Method java/lang/Object."<init>":()V
   4: return

static {}; // static class initializer block
  Code:
   0: iconst_2
   1: newarray int
   3: dup
   4: iconst_0
   5: bipush 123
   7: iastore
   8: dup
   9: iconst_1
   10: sipush 456
   13: iastore
   14: putstatic #2; //Field staticObject:[I
   17: return

}
```

6.
```
#include <stdio.h>

int foo (int i)
{
    if (i <= 0)
    {
        /* deliberately with no initial value */
        int j;
        return j;
    }
    return foo(i - 1);
}
int main (int argc, char *argv[])
```

```
{
    printf ("%3d %8x\n", argc, foo (argc));
    return 0;
}
```

A possible output looks like:

```
$ ./a.out
  1 bffa4270

$ ./a.out 0
  2        0

$ ./a.out 0 1
  3        0

$ ./a.out 0 1 2
  4   5b4e70

$ ./a.out 0 1 2 3
  5   2ac6a4

$ ./a.out 0 1 2 3 4
  6   11ca08

$ ./a.out 0 1 2 3 4 5
  7   e98aa7

$ ./a.out 0 1 2 3 4 5 6
  8   5b8573
```

The core of the example is this function:

```
int foo (int i)
{
    if (i <= 0) {
        /* deliberately with no initial value */
        int j;
        return j;
    }
    return foo(i - 1);
}
```

It calls recursively itself diminishing the parameter with each iteration, and returning a random value from the stack when bottoming out. Depending on the argument, these random values will be different, due to their being grabbed from different (uninitialized) locations of the stack.

Bibliography

[Hor94] E. Horowitz. *Fundamentals of Programming Languages*. Computer Science Press, Rockville, Maryland, 2nd ed. edition, 1994.

[Knu81] D. E. Knuth. *The Art of Computer Programming*, volume 1-3. Addison-Wesley, Reading, Massachusetts, 1981.

[Lak96] J. Lakos. *Large-Scale C++ Software Design*. Addison-Wesley, Reading, Mass., 1996.

[Lis96] J. Liskov, B. Guttag. *Abstraction and Specification in Program Development*. MIT Press, Cambridge, Massachusetts, 1996.

[Mey00] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, New York, 2nd ed. 2000 edition, 2000.

[Nyék98] Nyékyné Gaizler Judit (ed.) et al. *Az Ada95 programozási nyelv*. ELTE Eötvös Kiadó, Budapest, 1998.

[Seb13] Robert W. Sebesta. *Concepts of Programming Languages*. Pearson, Boston, 10th ed. edition, 2013.

[ÁF83] Ákos Fóthi. Bevezetés a programozáshoz. Technical report, ELTE TTK, Budapest, 1983.

Index