FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)

C/C++ Runtime Error Detection Using Source Code Instrumentation

Martin Bauer

Bachelor Thesis

C/C++ Runtime Error Detection Using Source Code Instrumentation

Martin Bauer

Bachelor Thesis

Aufgabensteller:	Prof. Dr. U. Rüde
Betreuer:	M.Sc. K. Iglberger
	Dr. T. Panas
Bearbeitungszeitraum:	01.10.2009 - 19.3.2010

Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 19. März 2010 .....

Abstract

The detection and removal of software errors is an expensive component of the total software development cost. Especially hard to remove errors are those which occur during the execution of a program, so called runtime errors. Therefore tools are needed to detect these errors reliably and give the programmer a good hint how to fix the problem. In the first part of the thesis different types of runtime errors, which can occur in C/C++ are explored, and methods are shown how to detect them. The goal of this thesis is to present an implementation of a system which detects runtime errors using source code instrumentation. and to compare it to common binary instrumentation tools. First the ROSE source-to-source compiler is introduced which is used to do the instrumentation. Then different instrumentation techniques are outlined, and methods are shown how to insert check-functions in arbitrary position in the code, which do not change the behavior of the original program. These check functions are used to pass necessary information to an external library, which uses this information to detect runtime errors. The system also takes type information into account and has therefore more information available than a binary instrumentation tool. It is shown that by instrumenting on the source code stage more errors can be detected than by using binary instrumentation.

Zusammenfassung

Das Erkennen und Beheben von Fehlern nimmt einen großen Teil der gesamten Entwicklungskosten des Software Projektes ein. Besonders schwer zu beheben sind Fehler, die erst beim Ausführen des Programms auftreten, sogenannte Laufzeitfehler. Deswegen werden Tools benötigt, die solche Fehler zuverlässig erkennen können und dem Programmierer Hinweise geben, die diesem helfen, die Fehler schnell zu beheben. Im ersten Teil der Arbeit werden verschiedenen Typen von Laufzeitfehlern, die in C/C++ auftreten können, beschrieben und verschiedene Methoden zur Erkennung aufgezeigt. Das Ziel dieser Arbeit war es, ein System zu entwickeln, das Laufzeitfehler durch Instrumentierung des Quelltextes erkennen kann. Zuerst wird der ROSE source-to-source compiler vorgestellt, der für die Instrumentierung verwendet wird. Dann werden verschiedene Instrumentierungstechniken beschrieben, und gezeigt, wie Test-Funktionen an beliebigen Stellen im Quelltext eingefügt werden können, sodass das ursprüngliche Verhalten des Programms nicht verändert wird. Diese Testfunktionen werden verwendet um Informationen an eine externe Bibliothek zu übergeben, die diese Informationen verwendet um Laufzeitfehler zu erkennen. Das System verwendet insbesondere Typ-Informationen und hat somit mehr Informationen verfügbar als ein Binärinstrumentierungs-Werkzeug. Es wird gezeigt dass durch Instrumentierung auf der Quelltext Ebene mehr Fehler erkannt werden können, als durch Instrumentierung auf Binär Ebene.

Contents

1 Introduction

According to estimates of the National Institute of Standards and Technology, errors in software products cost the United States approximately $ 60 billion in 2002 [1]. According to their study "more than a third of these costs, or an estimated $22.2 billion, could be eliminated by an improved testing infrastructure that enables earlier and more effective identification and removal of software defects." One part of the testing infrastructure are tools, which can detect software faults automatically. A variety of such tools is already in existence. However, usually for different software errors, different tools are needed.

Software errors can be classified by the time of occurrence. Most errors are caught when the software is built, and can be removed immediately. But not all possible errors are detected during compilation and linking. The remaining errors only show up during execution of the program. Unfortunately, they can depend on input data and may be very hard to reproduce. But even if they can be reproduced the cause of the error and the exact position in the source code is often hard to find. One possible way to detect and remove these errors is by using runtime error detection systems. These systems are able to detect errors when they occur and to show the programmer where they are located in the source code, so that they can be fixed easily.

C/C++ is a very powerful and expressive language: It provides precise low-level control of the underlying hardware, which makes it suitable for performance critical applications and system programming. However, this expressive power is obtained by allowing the programmer direct access to the hardware, and therefore to overcome many build-in limitations of the language, like the type system. In C/C++ it is, for example, allowed to access any memory location with any type of pointer, and every type can be casted to every other type. Memory management is explicitly handled by the programmer, in contrast to automatic memory management, like it is implemented in languages like JAVA and C#.

JAVA and other managed languages also have a built-in runtime-system, which catches most of the errors immediately when they occur. Consider an array out of bounds error: The JAVA runtime environment terminates the program as soon as the wrong array access occurs. In C/C++ the memory behind the array is accessed, and other variables may be changed. This can lead to errors which occur at a source code location far from the position where the cause of the error lies, so that these errors are very hard to find.

A variety of different runtime checking tools are used for C/C++. The most common tools work on the binary level: the program is compiled and linked as usual, but instead of executing it directly on the hardware, it is run in a virtual machine, which emulates the program and monitors it. The probably most widely used tool of this class is `valgrind` [NS07]. It uses dynamic binary re-compilation: First a code block is disassembled into an intermediate representation, then instrumentation code is added and the block is transformed back into machine code. This translated code block is then executed. None of the program's original

[1] http://www.nist.gov/public_affairs/releases/n02-10.htm

code is run. There have also been approaches to develop compilers, which automatically add runtime checks to the executable [Ste92].

Another class of tools uses source code transformation. Instead of modifying the compiler, the checks are expressed in the same language as the original code. A transformation tool automatically adds check-functions at critical code locations like array accesses. Then the transformed source code is compiled with a normal compiler, and an additional library is linked to the program, which implements the check-functions. This method can for instance be used to track type-information of pointers [LYHR01].

The goal of this thesis is to develop a runtime checking tool, which uses source-code instrumentation, as described above. For the instrumentation step the `ROSE` [SQ03, ros10] source-to-source compiler is used. In the following chapters I will describe the instrumentation process and the implementation of the additional library in detail and will compare the source-code instrumentation approach to binary instrumentation tools.

In the next chapter I will describe different types of errors which are common in C/C++. In Chapter 3 I will show how source code instrumentation techniques, which are used to insert check functions at arbitrary locations in the code. In Chapter 4 the Runtime System is presented, which contains the implementation of the check functions. Chapter 5 then shows, how these methods are used to detect common C/C++ errors.

2 Problem Description - Types of Errors

The debugging process of a software program usually consists of ttwo phases. At first all compile time errors have to be removed. Usually this is not a hard task because the compiler issues good error messages and can in most cases show the programmer the location where the error occurs. After having fixed these errors, the program can be compiled and executed. However, this does not mean that the software is now free of errors. The remaining errors occur during the execution of the program and are called runtime errors. A very common runtime error in C/C++ is the segmentation fault.

A segmentation fault, or access violation, occurs when a memory address is accessed, which is not allocated, or when it is accessed in the wrong way, like for example writing to a read-only memory location. Modern operating systems prevent processes from accessing memory, which has not been allocated to them because this memory may belong to a different process or the operating system. This prevents erroneous programs from crashing the whole system. When a process attempts to access a protected memory region, the process is terminated with a segmentation fault. It is up to the programmer to find the position in the source code where the access violation occurred. Therefore the bug has to be reproduced, which is often a difficult task. Then the program can be executed in a debugger until the segmentation fault occurs, and the exact source code position of the error can be determined.

However, the location where the fault occurs is often not the same location where the cause of the error lies. Consider the case where a previously valid memory chunk is freed, and afterwards accessed again. The access produces a segmentation fault, but the cause of the problem is the premature free operation. So the interesting code location may be the allocation or deallocation, not the access itself. This is one reason why runtime error checking tools are needed: They are able to detect the underlying causes for segmentation faults. Software errors do not have to manifest themselves in faults which can be detected by the operating system, like access violations, but they simply may result in wrong behavior of the program. Also in this case runtime checking systems can be very helpful to detect these kinds of errors. They execute additional checks in order to detect invalid or undefined behavior as early as possible. They check, for example, if all read variables are initialized, or that arrays are not accessed out-of-bounds. Further reasons why runtime error checking systems are important and a methodology how to test them can be found in [LCH+]. In this Chapter I will discuss certain types of errors in more detail, and describe why they occur. A good overview of these error types can also be found in [LCH+06].

2.1 Illegal Memory Access

An important feature of C/C++ is the possibility to directly access the underlying memory system, which makes the language attractive for system programming and especially for high performance computing. The programmer has very fine grained control over his program, and

no automatic error checking is done during the execution of the software. Performance is gained at the cost of loosing automatic error detection. In JAVA, for example, an exception is thrown if an array is accessed out of bounds. In C/C++ however a value is returned which happens to be in the memory adjacent to this array. Often these errors are not detected immediately. Higher level languages detect these errors, by checking all memory accesses in software, which of course costs performance.

Not all C/C++ memory errors remain undetected. In most architectures there exists a special hardware component for controlling memory accesses: The memory management unit (MMU). The MMU maps virtual memory page numbers to physical ones. The purpose of this mechanism is to assign a separate address space to every process, and prevent that one process can read or modify data from another process. The MMU can also restrict access to certain virtual memory regions by using the read-write flag of a memory-page, so a bug in one process cannot influence other running processes or the operating system. However, this technique only detects illegal memory accesses if the address lies on a different memory page, which is usually far from any legally allocated memory. Illegal accessed memory addresses are typically not far from allocated regions, since they are usually accessed due to array-out-of-bounds errors or illegal pointer dereferentiation. This means that a lot of errors remain undetected by the MMU.

2.2 Unsafe Library Functions

Another class of errors which are closely related to the ones described in the previous section and in fact mostly lead to an illegal memory access, are due to wrong usage of functions in the C standard library. Consider the following code section:

```
1    char src[4];
2    char dest[4];
3    src = "abc";
4    src[3]='d';
5    strcpy(dest,src);
```

The problem in the example above is the missing terminating null byte, which is overwritten in line 4. The `strcpy` function then reads and writes beyond the legal allocated memory regions of 4 byte, until a zero character is found. So the error actually occurs inside the library function because of wrong parameters.

These kinds of errors require special treatment by source code instrumentation tools. Note however, that typically the library is just available in a non-instrumented form so only the code written by the user can be instrumented.

2.3 Uninitialized Memory

When a process is started by the operating system, a virtual memory region is assigned to it. This memory region is completely set to zero by the operating system, so that a newly started program cannot read data from the process to which this memory was assigned before. This means that global and static variables are automatically initialized with zero [Int99, 6.7.8 (10)] [Int03, 3.6.2 (1)]. But a variable, which is not initialized explicitly and has automatic storage

duration (i.e. is not global or static), has an indeterminate value [Int99, 6.7.8 (10)] [Int03, 8.5. (9)]. In most C implementations they have the value of the variable, which happened to be at that stack address before. If the value of an uninitialized variable is read and used later on, this is probably an error.

However, not every use of uninitialized memory has to be an error. Uninitialized memory can in fact be viewed as random data, which may be used for random number generation, as it is for example done in the *Open-SSL* library. This part of the code was reported erroneous by runtime error detection systems, since it made use of uninitialized memory, so people tried to "fix" it, which introduced a severe security leak because the removal of this line of code made the random number generation predictable. [1]

2.4 Memory Leaks

There's another group of errors related to wrong memory management: Memory Leaks. Memory leaks occur when allocated memory cannot be freed anymore, for example because the last pointer to the memory chunk was overwritten. So memory is allocated, which is useless to the program, since it cannot be accessed any more. This memory is still allocated and cannot be used for other processes either. In many cases memory leaks are not critical, especially for short running programms. Modern operating systems free all the memory when a process is terminated. However, in long running programs, like server applications or in programs which run on embedded systems where not much memory is available a careful memory handling is crucial. A steadily increasing memory consumption will often have no immediate effect on the system. Most systems use secondary storage devices, like the hard disk, when no more memory is available in the RAM-chip. This process is called swapping: Regions, which are often accessed, are held in main memory, others are swapped out. A process with a memory leak that constantly allocates new memory, which is not freed, pushes out memory of other processes to the slower secondary storage device. This effect is called thrashing and can massively impact the performance of the system. Even if the leaking process is terminated it takes some time until the system recovers and has swapped-in the memory of the other processes. Memory leaks can even lead to the termination of the program. Operating systems limit the total amount of memory a process can consume, at least the total amount of memory is limited by the physical address space, which is around 2 to 4 GB on 32-bit systems.

2.5 Illegal Access to Allocated Memory

Most array-out-of-bound errors lead to an invalid memory access and can be detected by checking if allocated memory is accessed. However, there are cases when the memory behind an array is legally allocated, as it is shown in the following code example.

```
1    struct {
2        int arr [2];
3        int i;
4    } s;,
5
```

[1] http://www.schneier.com/blog/archives/2008/05/random_number_b.html

```
6    s.arr[2]=42; // writes i
```

Here the (non existing) third element of the array would be stored, where now the variable `i` is located. When trying to access the third array element, the member behind the array is modified instead. Even if some programmers may write code like this deliberately, such constructs should be reported as errors. If the behavior described above is intended, a union could be used. This union would contain the struct `s` as a first member and an array of size 3 as a second member. This is then an explicit way to make memory accessible by means of an array and by a named member. The task of a good error detection system is, to detect array-out-of-bound errors, even if the memory after the end of the array can be accessed legally.

The same is true for arrays on the heap:

```
1  int * arr1 = (int*) malloc( 2*sizeof(int));
2  int * arr2 = (int*) malloc( 10*sizeof(int));
3
4  arr1[7]=42;
```

When by accident the out-of-bound access hits another allocated memory block, as it is very likely in the code example above, also an error should be detected.

2.6 File Handling Errors

The problems described above can in general occur with every resource, which has to be allocated. Illegal memory references correspond to unallowed operations on file handles. The programmer may access an incorrectly opened file handle or try to access it in the wrong way, like writing to a read-only file. These kinds of errors are less serious than the memory errors, since they are normally reported by the underlying library. However, the mechanisms of a runtime error detection system can also be used for these types of errors. It can detect the code location where the source for the error lies, and give the user a more expressive error message than the library function can do.

3 Instrumentation

3.1 Overview

The runtime error checking system described in this thesis was tested and evaluated with a test suite created by the High Performance Computing Group of the Iowa State University. They have written several test suites in different programming languages to compare different runtime error detection systems on the market. Their results can be found in [LCH+06].

Since the system was developed on the basis of their test suite for serial programs [1], the implementation of the runtime error detection sytem, which is described in this thesis, was named after the test suite. The implementation is called *ROSE-RTED* because RTED testcases are solved using the ROSE source-to-source compiler, which will be described later on.

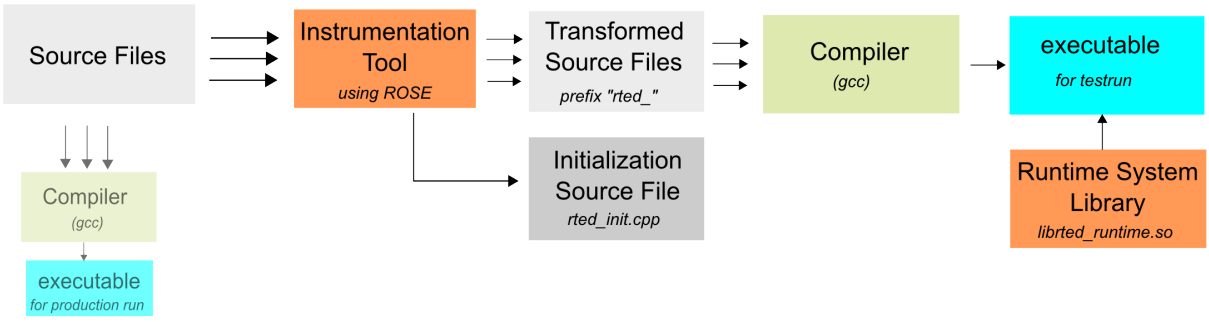

Figure 3.1: Structure of the implemented runtime error detection system

The system consists of two big components, the instrumentation (rewriting) tool and the runtime system library. First all source files of a project have to be processed by the instrumentation tool, which transforms the source code by adding check functions at appropriate locations. The instrumented code is pure C code, such that a project, which was written in C, can still be compiled with a C compiler, which is usually faster than a C++ compiler.

The instrumentation tool can be called with the same parameters as the GNU C/C++ compiler. This makes it easy to instrument a whole project, because just the name of the compiler has to be changed in the build-system, which can be easily done in common build-systems like cmake or automake. The instrumenter generates an instrumented copy of each source file. Additionally, a new source file, the so called initialization file, is created, which is modified each time a new source file is instrumented. This file is needed for handling type information and global variables. It consists of initialization code, which is run before any user written code is executed.

[1] http://rted.public.iastate.edu/Serial/homepage.html

The instrumented copies and the type file are now all compiled with a standard compiler, and the runtime-system library is linked to the executable. This library contains the implementation of all check functions, where the actual book-keeping of the allocation and initialization status takes place. The behavior of the runtime system can be adapted with a configuration file. For every class of error the user can decide what to do when an error of a certain type occurs: One can decide if the program should be terminated or if just a warning should be printed to a log file without interrupting the execution. When the executable is started, the program behaves just like it would, if it hadn't been instrumented. Only when the runtime system detects an error, the program is terminated and a message is printed with the code location where the error occurred. For further debugging there exists also the possibility to compile the runtime library with graphical user interface (GUI) support. If the GUI is enabled in the configuration file a window shows up, where breakpoints can be created. The program is then executed until it comes to a place with a breakpoint or where a fault is detected. Then the debugger window is shown again, where the memory status can be inspected and the values of all variables are shown.

3.2 Instrumenter: ROSE Source-to-Source Compiler

ROSE is a source-to-source compiler, which is developed at Lawrence Livermore National Laboratory. It is a framework for automatic source code rewriting and is used by ROSE-RTED to instrument the source code. For a detailed description of ROSE see [SQ03, ros10].

ROSE consists of frontends, a midend and backends. The frontend is responsible for parsing the source code, and generating a high level intermediate representation, the abstract syntax tree (AST). Each supported language needs its own frontend. For parsing C/C++ ROSE uses the Edison Design Group (EDG) frontend [edg10]. The AST representation is well suited for source-to-source compilation, since it also stores information which is discarded in normal compilers at this stage (like comments, preprocessor macros etc.). The AST is used in the midend to do source-code analysis and transformation. ROSE-RTED inserts its check-functions and other transformations at this stage. Finally the backend again generates C/C++ code from the AST.

3.3 Instrumentation Methods

The basic idea of source code instrumentation is to add additional check functions or other suitable constructs to all places in the source code, where the programmer might have made an error. Ideally the check function should be executed right before or after the critical source code section. Let's take as an example a simple read of a variable. To check if the read is valid, i.e., if the variable has been initialized, the runtime system needs to know the address of the variable being read. So the task is now to insert a check-function as close as possible to a read operation of a variable. The most common situation of a variable read is probably a simple assignment.

```
1    result = (a+b)*c;
```

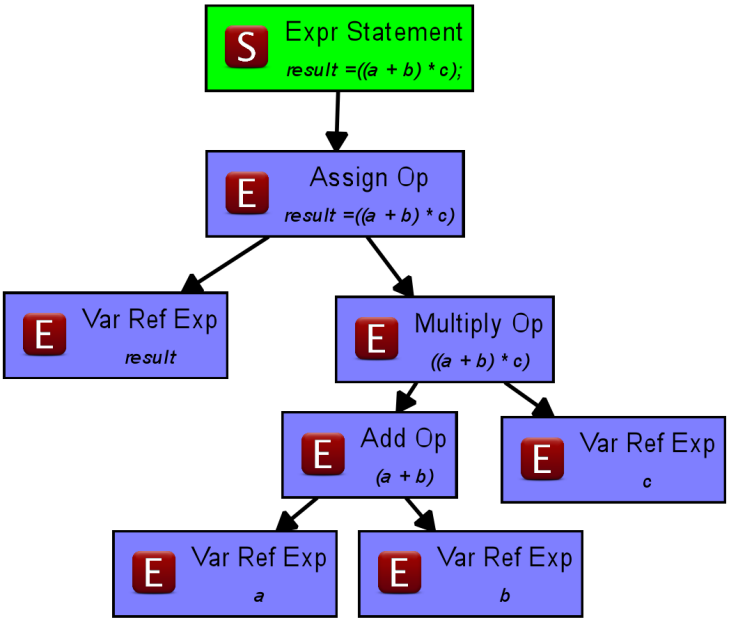

Figure 3.2: ROSE AST of `result=(a+b)*c;`

As can be seen in the figure above, four variable references (`VarRefExp`) occur in the AST of this statement. So the first idea is to traverse the AST, look for variable references and add a check-function for each variable, which is read before the whole statement. This results in a code similar to the following:

```
1    checkRead(&a);
2    checkRead(&b);
3    checkRead(&c);
4    r= (a+b)*c;
```

This method works fine in the given example, however problems occur in more complex language constructs, like logical operators. Consider the following code portion:

```
1    bool a = false;
2    bool r = a && b;
```

According to the language specification the variable b is never read, since its value is irrelevant for the evaluation of r. With the instrumentation method above problems would occur, since `checkRead` would also be called for b, even if b is never read. This would generate false positives in our runtime error checker. The critical point here is the location where the check-function is inserted. In the current method we look for the surrounding statement, which in this case is the complete second line, and insert our check before. However, the portions of code we have to check are not statements but expressions. This conclusion leads naturally to the next instrumentation method: instrumentation with the help of logical operators. Our check-function should only be executed if and only if the instrumented expression is executed, which means we have to insert the check-function directly in the (logical) expression. The goal is to replace the original expression by a new one, which contains the check-function, but evaluates to the same value and has the same side-effects as the original expression.

As a first try, the bitwise **or** operator "|" can be used. All check-functions are changed such that they always return zero, and are inserted in the following way:

```
1  bool r = ( checkRead(&a) | a) && ( checkRead(&b) | b );
```

The new expressions evaluate to the same value as the original ones, since $(a|0) = a$. Note that both operands are evaluated every time, so the check-function could be added either before or after the original expression.

It turns out that this is the main advantage of this instrumentation method. While this method works fine if the original expressions are of boolean or integer type, problems occur if class valued expressions are transformed. Normally the bitwise **or** operator is not defined on classes, or even worse, in C++ it may be overloaded and do some user-defined operation. So this type of instrumentations gets quite complex, since one would have to check if already an overloaded bit-or operator exists, if so rename it and all calls to it. Then one would have to insert a new one, which returns a reference to the original object, regardless of the second operand.

A better and much easier solution can be used if the check-function has to be inserted *before* a given expression. The comma-operator does exactly what is needed in this case: it executes its first operand, discards the value, then executes the second operand and returns this value. Unfortunately there is no `operator,` which works the other way round i.e. returns the value of the first operand. So this method can only be used, when the check-function has to be inserted before the original expression. For the example above the instrumentation would look like:

```
1  bool r = ( checkRead(&a) , a) && ( checkRead(&b) , b );
```

But how can check-functions be inserted after a given expression? We could instrument with the help of the bitwise **or** operator as described above, but since this only works with integral datatypes, another method is needed.

If the instrumented file is later on compiled with a C++ compiler we could insert a template function, which has a similar behavior as the comma operator, but returns the value of the first expression:

```
1  template< typename A, typename B>
2  A inverseCommaOp(A & first, B & second) {
3     return first;
4  }
```

This is a quite general method, however it still has some drawbacks: One disadvantage is, that can of course only be used in C++ files. Often the requirement is, that a C file should not be converted to C++ by the instrumentation, since C++ files usually take longer to compile. So a method is needed, which only makes use of C constructs, and is able to insert a function after a given expression. The only way to do that is to copy the whole expression, assign it to a temporal variable, call the check-function, and finally use the temporal variable in the original statement or expression. This does not change the semantics of the whole construct, since the compiler would generate a temporal object for the expression anyway. So this transformation behaves the same way as the original construct, even in terms of constructor calls in the case that the temporal variable is of class-type. Of course the name of the helper-variable has to be chosen such that no naming conflicts occur. There may already have been inserted helper-variable for other expressions or by coincidence the programmer may also have declared

a variable called `helpVar`. In this case a unique number is appended to the helper variable name.

```
1    int helpVar = a+b;
2    checkFctAfterExpr();
3    r = helpVar;
```

Now check-functions can be inserted before and after each expression, which will be used in the following, to monitor memory related activity of the program.

3.4 Instrumentation of important code constructs

3.4.1 Allocation Routines

Since the runtime system has to be able to check every memory access, it has to be informed of all memory allocations and deallocations. First the case of heap-memory allocation is described, stack "allocations", i.e. variable declarations are then described in Chapter 3.4.2. It is assumed that the program uses the allocation routines of the C standard library (`malloc`, `free`) or the allocation operators `new` and `delete`. As far as the runtime system is concerned these two types can theoretically also be mixed, i.e., some heap portions can be allocated in *C* style, others in *C++* style. Mixing these two methods is bad coding practice since it relies on the fact that the C++ allocation mechanism internally uses the `malloc`/`free` system. Thus this case is reported to the user as warning.

Let's take a look at the allocation routines: The runtime system needs to know the size and the address of the new memory region. Since the address is not known until after the call, the instrumentation has to be done after the call to the allocation routine. As pointed out above, instrumentation after a given expression is quite complex and hard to implement and another, simpler method is chosen: Every call to an allocation or deallocation routine is renamed and reimplemented in the runtime system. So instead of calling the memory handling library directly, an additional layer is introduced. These intermediate functions behave exactly the same as the original functions in terms of parameters and return values, but additionally perform the necessary checks and registrations.

Of course these methods only work under the assumption that the standard C/C++ memory allocation is used. If, for example, a different `malloc`-library is used, the runtime system would have to be adapted.

3.4.2 Variable Declarations

Having done the substitution of allocation functions and operators, only the heap layout is known. However, since the complete memory layout is needed to perform memory access checks, the "allocations" on the stack have to be instrumented too.

Every declaration of a variable or function parameter has to be reported to the runtime-system. Instrumentation of declarations is done after the complete declaration statement, not somewhere within the declaration statement. This is because the declared variable can't be legally accessed inside the statement. Consider the code portion `int a=a+5` and assume `a` was not declared before in a different scope. Though most compilers do not report an error when compiling this code fragment, this code results in undefined behavior, due to the uninitialized

use of `a` . Since the creation of `a` is registered after the statement, `a` is not yet known to the runtime-system when the initializer part is executed, and thus an error is reported. However, there exists a special case when a variable is used in its declaration statement: `int a=5, b=a ;` . This case is not a problem in the implementation since in the ROSE abstract syntax tree, this statement is internally represented with two single statement-nodes, one for `a` and one for `b`.

Since the C99 standard, variables do not have to be declared at the beginning of a scope, as it was the case in ANSI-C and C89 [Int99, 6.8.5.3]. For instance, they can be declared in the initialization part of a `for`-loop. To instrument these declarations correctly the loop has to be rewritten, as it is shown in the following code example.

```
1  // Before transformation
2  for(int i=0; i<N; ++i) {...}
3
4  // After transformation
5  { int i =0;
6    for(; i<N; ++i) {...}  }
```

Another special case are static variables, which are declared in a function. They are initialized when the function is called the first time, but the memory associated with them is not freed after leaving the function. They still have the same value when the function is called again. This is signaled to the runtime system when creating a static variable. Then the variable information structures are not deleted when the variable goes out of scope, and the variable has the same initialization status it had on a previous call of this function.

3.4.3 Global Variables

Memory for global variables is already valid at the beginning of the program. The functions, which register global variables, have to be called before code of the original program is executed. If the global variable registration would not be executed at the very beginning, all accessed to them would be lead to an error, because the runtime system does not know that the memory where the global variable is stored, is valid.

A method is needed to insert a function, which is called before any user written code is executed. This has to be handled differently for C and C++. The instrumentation should be done such that it does not insert C++ code in a C program, so that the (faster) C compiler can still be used. To inject code at the beginning of a C program, the `main` function is renamed, and a new `main` function is inserted. In this new `main` function the registration of all global variables is done first, and then the renamed original main function is called. For C++ programs this approach does not guarantee that the instrumentation is called before any user written code. Constructors of global or static objects are executed before the main function is called. If in one of these constructors global variables are accessed that are not yet known to the runtime system, so the memory where this global variable is stored is considered invalid, and the access to this variable would result in an error.

Since constructors of global objects are executed first in a C++ program, the initialization code has to be a constructor. A dummy class is instrumented, which consists only of a constructor, and a global instance is created at the beginning of the source file. The C++ standard guarantees that the constructors are executed in the order of appearance within a source file.

Initialization order across translation units is not specified in the standard. However, the GNU gcc compiler provides an attribute to specify the initialization order across translation units. Low values of the init_priority attribute indicate higher priority. The lowest possible value is 101 and is used for the initialization object [2]:

```
1  class Initializer {
2      Initializer() {
3          // Global Variable registration
4          // and other intitialization code
5      }
6  };
7
8  Initializer inst  __attribute__ ((init_priority (101)));
```

This attribute has to be used because the object of the initializer class will be in a separate source file. In the current version of the instrumentation tool, the constructor approach, which is necessary to handle C++ code correctly is not yet implemented. This means, that currently false positives are detected when global variables are accessed in the constructor of a global class instance.

All global variables from all source files have to be registered at the beginning. Since the instrumentation tool can be called like a compiler, with only one source file at a time, the problem arises where to add the initialization code. Lets assume the `main` renaming approach is used, then only one function exists in the program where initialization code can be instrumented: the new `main` function. This new `main` function is therefore written in a separate source file and every time a source file is processed, registration functions for the global variables are added to the startup code in the separate file.

Only global variables, which are declared without the extern modifier are registered, because for these variables no memory is reserved, they only refer to another variable in a different translation unit. For static member variables, the same method is used. They can be seen as global variables, which are only visible in a certain scope. The memory, which is allocated for them is never freed or used for other purposes during the program run.

The resulting relevant parts of the instrumentation are shown in the code example below:

```
1  struct C  { static int staticMember; };
2  int C::staticMember=42;
3
4  int globalInteger;
```

Listing 3.1: Original Code

```
1  // Registration code
2
3  struct C {  static int staticMember; };
4  extern int globalInteger;
5
6  void RTED_onStartup()
7  {
```

[2] http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/C_002b_002b-Attributes.html

```
8     RTED_registerType("C","C",sizeof(struct C ),...);
9     RTED_createGlobalVar(&globalInteger,"globalInteger",...);
10  }
```

Listing 3.2: Instrumentation in separate file

3.4.4 Scopes

In order to know when the memory for a stack variable becomes invalid, the runtime system needs to know when a scope is entered or left. Scopes occur in the ROSE intermediate representation as a "basic block" object. In each basic block a function is added in the beginning and at the end, to notify the runtime system that a scope is entered or left. This works well for loops and if-statements but a problem occurs when a function scope is entered or left:

```
1   // Transformed code of
2   // "int func(int a, int b) { return a+b; } "
3   int func(int a, int b)
4   {
5     RTED_enterScope("func");
6     RTED_createVar("a", ...);
7     RTED_createVar("b", ...);
8       return (checkRead(&a),a)+ (checkRead(&b),b);
9     RTED_exitScope();
10  }
```

In the code example above this problem is illustrated. First the runtime system is informed that a scope is entered. Then the two parameters are registered as stack variables. However, the exit-scope function is never called, because of the return statement. One could put an extra exit-scope call before every return, but then another problem occurs: After the exit-scope call the stack variables are no longer valid, and the access to the parameters in the return statement would cause an error. The exit-scope function has to be inserted after the variable accesses. Here the technique I've described above in the instrumentation methods chapter can be used. The return statement is replaced by the following code:

```
1   {
2      int RTED_helpVar = (checkRead(&a),a)+ (checkRead(&b),b);
3      RTED_exitScope();
4      return RTED_helpVar;
5   }
```

The newly introduced helper variable is accessed after the exit-scope function, but this generates no error, because the access to this helper variable is not checked by the runtime-system. There are more special cases when the above described instrumentation of scopes does not work, like `goto` statements or exceptions. These cases are not yet handled in the current version.

3.4.5 Instrumentation of Read and Writes

The runtime system has knowledge of all valid memory addresses on the stack and on the heap so that all the information is available to check if a memory access is legal. We just need to know the address and the size of the accessed memory region. The main task is therefore to find all locations in the source code, where memory is accessed, and call a function before the actual access, which passes the address and size to the runtime system. Memory accesses are expressed in C/C++ via expressions. The instrumentation is done with the comma-operator method as described above, so the function can be inserted directly into the expression, right before the memory access is executed.

Now all the source code positions have to be found where an actual memory access takes place. This is done with the ROSE intermediate representation, the abstract syntax tree. Each variable identifier, which occurs in the code is represented in the AST as `VariableReference` node.

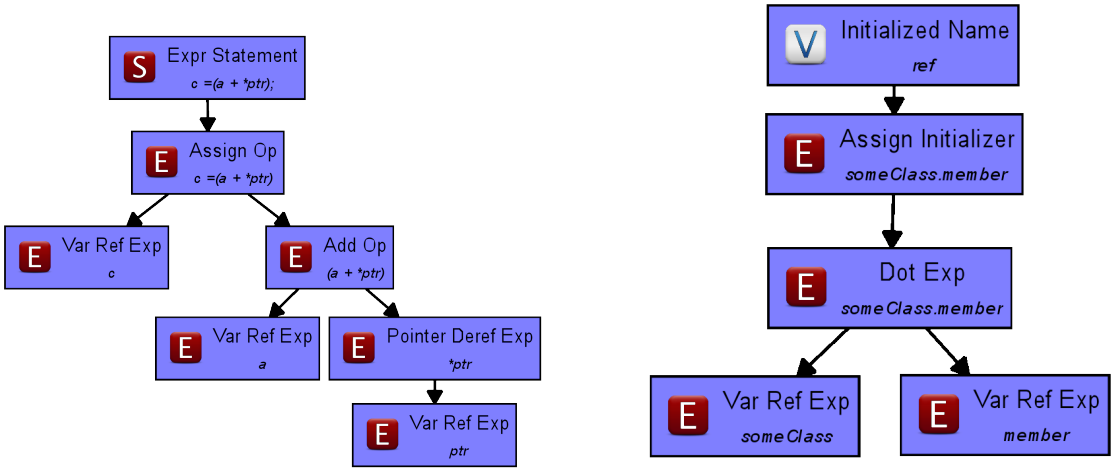

Figure 3.3: AST of expressions `c=a+ *ptr` (left) and `int & ref = someClass.member;` (right)

As an example how the instrumentation positions are found in the source code, let's have a look at the expression `c = a + *ptr` . In Figure 3.3 the according AST representation is shown: For each of the accessed variables a Variable Reference node occurs. The basic instrumentation algorithm traverses the AST. When it finds a variable reference, it first has to determine if this corresponds to a read or a write operation. Expressions, which correspond to write operations have an assignment operator node as ancestor. So every time a variable reference is found, all ancestors are checked. If an assignment operator is found, and we came from the left subtree, the variable reference is a write operation. In the example AST this would be the node corresponding to `c` . If no assignment operator is found (and the term 'assignment operator' also includes '`+=`', '`*=`' etc.), or the node is in the right subtree of the operator, the variable reference is a read.

With this method some important memory reads remain undetected: pointer dereferentiations. The expression `*ptr` is actually translated into two memory accesses: the pointer has to be read, and the location where it points to. By checking these two read-operations many

common programming errors are detected, like accessing already freed memory or array-out-of-bounds errors. To handle this case, an additional check function is inserted before every `PointerDerefExp`-node. After inserting all these check-functions the transformed expression looks like this:

```
1 (chW(&c,sizeof(c)), c) =
2        (chR(&a,sizeof(a)) , a) +
3       * (chR(&ptr,sizeof(ptr)), chR(ptr,sizeof(*ptr)) , ptr);
```

By using the method described above check-functions are added in cases where actually no memory access occurs. Consider the expression `res=&a`. The variable `a` is not accessed, but in the AST a `VariableReference` node occurs. To prevent these wrong instrumentations, we have to make sure, that the variable reference is not an ancestor of an `AddressOf` operator. A similar case is shown in Figure 3.3 on the right. The initialization of a reference does not translate to a memory access either. But even if, in this example, the left side of the assignment wouldn't be a reference, there would be too many check functions instrumented. In the AST two variable references occur, so two checks would be inserted: one at address `&someClass` and one at `&(someClass.member)`. Obviously only the second one is correct. The same is of course true for the operator `->` . But all these special cases can be detected by inspecting the AST around the variable reference, so that the instrumentation can be adapted accordingly.

3.4.6 Type Layout

As described in the chapter about global variables, an instrumented initialization function is available, which is called before any user written code is executed. This function can now also be used to pass the memory layout of all types to the runtime system. As we will see later, this knowledge can then be used by the runtime system for better error detection. For every type, the name, the size in bytes, and for compound types also the name and size of the members has to be determined and passed to the runtime system.

For member variables also the byte offset from the beginning of the class or the struct has to be calculated. The offset cannot be calculated from the size of previous members but has to be explicitly passed, since the compiler may insert extra space between members due to memory alignment constraints. Some processors require specific memory alignment, otherwise a bus error occurs, which leads to the termination of the program. Other processors are much faster when handling aligned data. For an integer, which is not 4-byte aligned on a 32 bit architecture usually two memory load operations are required instead of one. The alignment behavior of a compiler can be changed by directives like `pragma pack` and can be different for each type. In order to always know the correct memory layout each time, even in the presence of these compiler directives, the offset of each member has to be calculated at runtime.

The first approach to do that, is to create an instance of the class or struct and calculate the member-offsets using the address-of operator, at it is demonstrated in the next code example.

```
1 // #pragma pack (1)
2 struct S { char c; int i; };
3
4 // In init function:
5 S instance_S
6 size_t offset_i;
```

```
7   offset_i = (char*)&instance_S.i - (char*)&instance_S;
```

The offset of the second member in this example will on most architectures not be stored at offset 1, even if a char needs only one byte, but on offset 4, such that the integer is aligned correctly. If the pragma statement is uncommented the offset will change and the integer will be stored at offset 1.

While this approach works well for C-code, it can change the behavior of C++ programs. When creating an instance of a class, the constructor is called. The constructor, however, may have side-effects. It can for example modify a static or global variable. In order to keep the original behavior of the program no additional instance of a class can be created. So another approach is used here: The offset information can also be obtained from a pointer to this class. When using the arrow operator the compiler internally has to calculate the offset of the accessed member before it dereferentiates it. The offset can then be determined without creating an instance using the following construct:

```
1    struct S { char c; int i; };
2
3   // In init function:
4   size_t offset_i = (size_t) ( &((struct S *)0)->i) )
```

This constructs only works correctly under the assumption that the `&` operator has not been overloaded. To handle the general case the instrumenter would have to replace the overloaded operator with a member function, and adapt all calls accordingly. This is not yet implemented in the current version.

The type registration takes place in the initialization function, which is in a separate, newly generated source file, the so called initialization or type file. The offset calculation code does only work if the type is known in the initialization file, because a pointer to this type is created. Therefore all type definitions, including `typedef` statements, have to be copied to this file. Additionally, the definitions are modified, such that all member variables are made public. When the offset calculation code would be used with a private member, an error would occur when accessing the private member with the arrow operator. Therefore all classes are converted to structs and all access modifiers are left out. This does not make accesses to private members legal in user written code, only the copied definition in the separate source file is changed.

This separate source file is extended each time a file of the project is instrumented. When a new type definition is detected, it is checked if this type is already known from a previously instrumentation run. If not, the definition is copied and registration code is added to the init function. After having processed all source files of a project, every type, which occurs in this project, is registered in the initialization function. This approach can lead to a problem: It is allowed to have a type with the same name, but different definitions in different translation units. This means that it is allowed to have two totally different classes, which have the same name and are defined in different source files. As long as these two source files are not including each other, this does not result in an error. The two source files are compiled separately, and the linker does not produce an error either. The type registration method assumes, that identical type names imply also identical memory layout. Since this assumption is true in most cases, this method is still used in the current implementation of the runtime error detection tool. The solution for this problem would look like this: Not only type names should be compared when checking if two types are equal, but also the count and type of their members. When two types have the same name, but different memory layout, one type would have to be renamed, in the

instrumented source file. This approach is very complex to implement, and therefore still the type name comparison is used.

4 Runtime Library

Now that all the instrumentation is in place, I will describe the functions of the runtime system library, where all the instrumented check-functions are implemented. The basic structure of the runtime system is shown in Figure 3.3. In the following sections I will describe the different modules and their tasks in detail.

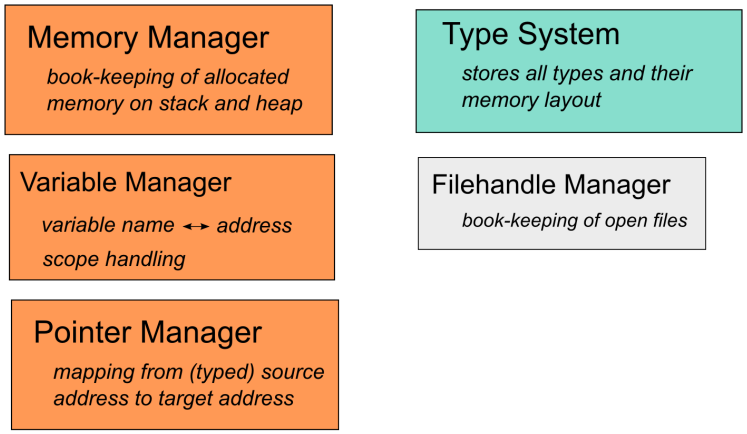

Figure 4.1: Structure of runtime-system library

4.1 Type System

For the detection of certain errors, type information is needed at runtime. Therefore functions have been instrumented, which report the memory layout of all known types to the runtime library. First, all basic types (`int`, `float` etc.) are registered, where only the name and the size is needed. The code, which registers them, does not need to be instrumented, because they are built into the language, and their size does not change on a fixed architecture.

A slightly more complicated case are `typedef`s. There are basically two methods to handle them: They could either be resolved by the instrumentation tool or by the runtime system. Let's first consider the case where they are resolved by the instrumentation. Since the instrumentation tool has parsed the whole source code it has the knowledge of all `typedef`s and thus, instead of passing type names directly to the runtime library (for example when registering types of member variables) it can resolve the `typedef`. The new type name, which has been introduced by the typedef, is never used in the runtime library. This improves performance, since no lookup has to be done at runtime. The drawback of this method is, that the runtime library has no knowledge of the new type names and thus cannot use them in messages to the user or in the debugger GUI. An error message may be more expressive when the `typedef`

name is used (for example `time_t` instead of `long int`). Therefore the decision was made to resolve `typedef`s in the runtime library.

During the startup of the program all types and their memory layout have been registered, and now type information for allocated memory can be gathered. The basic idea is, to know for each memory region which types are stored there. For the stack this information can be gathered easily: the type is part of the variable declaration and cannot be changed during the run of the program. Of course this memory region can be accessed with a pointer of different type, but I will describe this case later on. The type information for heap regions, which are allocated in the C++ way with `new` is also known, because the operator `new` takes as a first argument the name of the type, and returns a pointer of this type. This is not the case for C-style `malloc` allocations, since `malloc` just takes a size in bytes and returns a `void` pointer. There is no type information associated with this memory chunk. In this case a heuristic is used: The returned `void` pointer can never be dereferenced and therefore has to be casted to another pointer type somewhere. Every memory access is monitored, and type information for pointers is always known, since they have been explicitly declared. With this knowledge we can detect when a pointer changes the address it points to. Now we assume that the memory is of the same type as the basetype of the pointer which points to it. I will illustrate this on the following example in more detail:

```
1   void * voidPtr  = malloc(8);
2   int * intPtr    = (int*) voidPtr;
3   double * dblPtr = (double*) voidPtr; // heuristic fails
```

In the first line no type information is known yet, because the return value of malloc is assigned to a `void` pointer. In the second line the `void` pointer is casted to an `int` pointer and the runtime system assumes that this memory region will always be accessed as an integer. Of course this assumption is not always true. The pointer might be casted again, and used with this other type. As long as no integer has been previously written to this location, no problem arises. So instead of the heuristic described above, we could use pointer dereferences instead of pointer assignments to detect the type. However, it makes no sense to have a pointer pointing to a memory region of an incompatible type, which will never be dereferenced, as in the second line of the example. This might be a hint for an error, and therefore the decision was made to use pointer assignments instead of dereferences for the detection and verification of type information.

To create a C-style array the size and the number of dimensions have to be declared. This information is passed to the runtime system such that it can detect out-of-bounds errors. The same is true for C++ free store allocations: The size of an array is specified as a parameter of operator `new`. Again the harder case are C-style arrays on the heap. They are usually allocated this way:

```
1  unsigned int nrElements = 10;
2  int * arr = (int*) malloc(nrElements * sizeof(int));
```

In this example type information for the first integer is known, due to the cast operation. However for the remaining memory no information is stored. Since this is the usual way to create an array on the heap in C, one could come up with the idea to extend the heuristics described above in the following way: When only for the first part of an memory region the type is known, assume that this memory region is used to store an array, and extrapolate the

type information of the first part to the rest of the region. However, this is quite a strong assumption and this would lead to false positives in many cases:

```
1  struct S { int i; double b; }
2  int * firstMemberOfFirstStruct = (int*) malloc( 5*size(S));
3  *firstMemberOfFirstStruct = 1;
4  S * structPointer = (S*) firstMemberOfFirstStruct;
5  structPointer[4].b = 1.0;
```

If the array extrapolation heuristic would be used, an error would be reported in line 4 although the code would not lead to any errors. This is because in line 2 the assumption would be made that the complete memory region is an integer array. This would be possible since a double needs on most systems twice as much memory as an integer, so an integer array with 15 elements would be detected. When the same memory region is accessed as a `struct` (line 5), the runtime system would detect the incompatible types and terminate the program with an error. So the array extrapolation heuristics is not used, the type information is gathered later on, when the elements of the array are accessed. This leads to a different representation of stack arrays and heap arrays: the size and number of dimensions of stack arrays is explicitly known, whereas the type information for heap arrays is detected step by step, when the single elements are accessed.

The type system has functionality to merge type information. Memory may first be referenced as a basic type and later on with an enclosing struct or class, like in the following example.

```
1   struct S1 { int a; int b; bool c;} ;
2   struct S2 { int a; int b; double c; };
3   void * vPtr = malloc(sizeof(S1));
4   int * intPtr = vPtr;
5   *intPtr = 42;
6   S1  * ptr1 = vPtr;
7   ptr1->a = 42;
8   S2 * ptr2 = (S2*) ptr1;
9   ptr1->a = 42;
```

In line 3 memory is allocated, and the pointer is not casted to any type. So at this point the runtime system has no type information of the allocated memory at all. With the statement in line 4 the runtime system gets notified that an integer pointer is created, which points to the beginning of this region. Thus it is known that in the first bytes of this region an integer is stored. When in line 6 a `S1` pointer is created with the same target address, the runtime system has to check if this `struct` is compliant with the already known type information. The merging routine is called and with the memory layout information of the `struct`, it can be determined that the `struct` has an integer as first member, and therefore this operation is valid.

Before type information is merged, the consistency of the two types is checked. In line 8 the merging test fails and an error is reported, because the two `structs` have different memory layout. If the types of the structs would be the same and only the naming of the members would be different, the merging test would be passed. One could argue that the code construct above is legal as long as only the first two members are accessed. However, the decision was

made to be restrictive in this case, to detect errors as early as possible.

There exist some cases where the programmer deliberately uses different types to access a single memory address. This is often needed in system programming where the memory is for example accessed byte-wise with a `char` pointer. This is legal, because the C standard specifies, that every type may be copied into an `unsigned char[n]` array, where n is the byte-size of the type [Int99, 6.2.6.1 (4)]. The runtime system assumes that every memory location can only be accessed with one type. In some cases, like in the one above this may be too restrictive, and the runtime system has to be configured, to only report a warning in such cases.

4.2 Memory and Variable Manager

At the core of the runtime system is the Memory Manager. It keeps track of all allocated memory regions. The heap allocations are reported by the instrumented allocation functions (`malloc` calls and operator `new`). To know which addresses on the stack are valid, an allocation has to be registered each time a variable is declared. The task of the Variable Manager is to notify the Memory Manager each time a variable is declared, such that the according addresses can be marked as allocated. In order too detect stack deallocations the Variable Manager also keeps track of scopes where the variables are declared in. When a scope ends the Memory Manager is notified that this stack region is no longer valid. The Variable Manager stores in addition to the address also a map which connects the stack addresses to variable names. This information can also be directly accessed by the programmer for debugging purposes. The view shows the stack layout just like a normal debugger does.

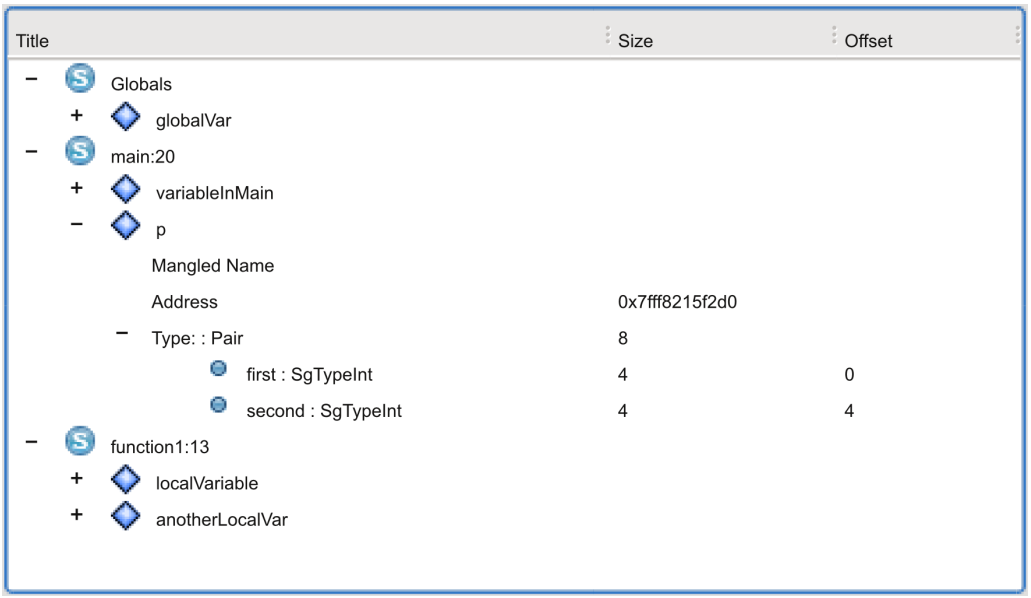

Figure 4.2: Stack view of the debugger included in the runtime system library

The Memory Manager now has a list of all allocation records. Internally there is no distinction made between stack and heap regions. In these allocation records the address, size and the initialization status of the memory chunk is stored.

The initialization status is a bit mask that stores if a value was already assigned to a variable. If the variable is not global or static, it is not zero initialized and has indeterminate value [Int99, 6.7.8 (10)]. For every allocated byte an initialization bit is stored, to indicate if this location has been initialized. The Memory Manager has no knowledge whether a certain memory region holds a stack variable or is a region on the heap.

The variable names and the stack layout are stored in the Variable Manager. The variable names are, strictly speaking, not required for the functionality of the runtime system because the address of a variable is a better identifier for a memory location. They are only stored because of display purposes: The variable names are shown in the graphical debugger, or in error messages. The more important task of the Variable Manager is to keep track of scope entries and exits and to notify the Memory-Manager when variables are declared or go out of scope.

4.3 Pointer Manager

The Pointer Manager module monitors memory locations, where addresses to other memory locations are stored. This is necessary for detecting memory leaks and out-of-bounds errors as I will show in Chapter 5.

The task of this module is to find these memory locations where pointers are stored and to initiate checks if a pointer changes its target address. Since we have the Variable Manager available, which knows all stack variables together with their type, all pointers, which are currently in scope, can be monitored. However, not all memory locations where other addresses are stored correspond to pointers on the stack. In the following code section two other cases are illustrated:

```
1  struct Element { int i; int * ptr; };
2
3  int ** arr = (int**) malloc(10 * sizeof(Element*));
4  for(int i=0; i < 10; ++i)
5      arr[i] = (int*) malloc(5 * sizeof(Element));
```

Here a two dimensional array is allocated. First a heap segment is created which holds pointers to the data segments. These pointer have to be detected and managed by the runtime system as well. They are detected in the `for`-loop, since type information is gathered when the assignment of the data segment pointers takes place. When the type system gathers new type information this is propagated to the Pointer Manager. Now the complete pointer information of the two dimensional array is known, but there are still other memory locations which store pointer values. The struct has a member of pointer type, which has to be registered in the pointer manager as well. Each time a compound type is created, all pointer members are registered. This has to be done recursively, since the `struct` may contain other `structs` or `classes` as members.

Now these pointer locations are monitored. When a memory write access takes place on such an address (which corresponds to a change of the pointer value) checks are initiated to detect memory leaks and other pointer related errors.

5 Results

5.1 Illegal Memory Access

With the Memory Manager module of the runtime-system we are now able to detect both cases of illegal memory accesses: accesses to unallocated memory and read operations on uninitialized memory. Before any variable is accessed, a check-function was instrumented, which reports the memory access to the runtime library. The called check function makes sure that the address lies in a previously allocated memory block. This is possible because all allocations have been instrumented before and are therefore all known. For each allocation block a bit mask is stored to keep track of the initialization status of the memory chunk. This makes it possible to report reads of uninitialized memory portions. When an uninitialized variable is read this is immediately reported as an error. An alternative to this approach is to only report an error, when an uninitialized variable is used at critical positions, like in a conditional clause, or as an argument to a system call. This approach is described in more detail in [BNK+07]. However, the decision was made to report the access to an uninitialized variable as an error immediately, since it makes no sense to propagate undefined values throughout the program.

To detect these kinds of errors the usual approach is to use binary instrumentation tools, like for instance `valgrind`. As we have seen above, the source code instrumentation tool is now able to perform the same task, however with a little more effort. Since binary instrumentation tools do not make any changes on the source code stage, the program does not have to be recompiled. Code in external libraries is automatically checked, too. Every machine code instruction is simulated and therefore it makes no difference if the instruction is part of a library or was written by the user. In order to use source code instrumentation reliably, the sources of all libraries have to be available, so that they can be instrumented and recompiled. To overcome this limitation there exist techniques, which make it possible to use non-instrumented libraries at the cost of less reliable error detection. I will discuss them in section 5.5.

5.2 Memory Leaks

The source code instrumentation is also able to detect memory leaks. The easiest way to do that is to check after the termination of the program, which memory regions have not been freed. Binary instrumentation tools have the same ability, since they can also monitor all allocation operations. If the program was compiled with debug information even the source code position of the allocation is known.

An advantage of the source instrumenter is that it is often able to detect a memory leak before the termination of the program. For each allocation a list of all the pointers is stored, which currently point into this allocated region. This is possible, because the runtime system is informed each time a memory access takes place, and can check with the help of the type information it has available, if this memory access resulted in the change of a pointer value.

If no pointer refers to the beginning or points into the allocated memory region anymore, it is very likely that the address of the memory region is lost. This means that it can not be deallocated any more. However, this system can fail, because a pointer can be changed, so that it does not point into the allocated region any more, but still holds the relative information of the allocation address. This can be best seen on an example:

```
1   int * ptr = (int*)malloc(sizeof(int));
2   ptr += 10;
3   ptr -= 10;
4   free(ptr);
```

The runtime system would report an error in the second line, since no pointer points to the allocated memory any more. But the memory can still be freed correctly. Basically the original address can still be recomputed, as long as the pointer is only changed in a relative way. But to detect this reliably a data flow analysis would be necessary, which would be very complicated to implement. So an easier solution was chosen: Instead of terminating the program in this case, just a warning is reported. If this warning really corresponds to a memory leak, can be decided after the termination of the program. The programmer then has the information, that a memory leak occurred, where the memory was allocated and where the last pointer to the memory region went of scope or was overwritten.

5.3 Access to Padded Memory Regions

As described in 4.1 the type system knows the memory layout of each type and therefore also which memory regions are not used due to alignment reasons. This padded memory is not valid and should not be accessed. However, since this memory is correctly allocated, these accesses are not recognized as errors by binary checkers. Consider the following code segment:

```
1   struct  { char arr[3]; int i; }  s1 ;
2   s1.arr[3];
```

In the second line an out-of-bounds error occurs, which might never be detected since the memory which corresponds to the fourth array element is normally unused, and thus no other member variable is overwritten. According to the C-Standard [Int99, Chapter 6.2.6.1 (6)] the padding bytes take unspecified value. This means, that padding bytes do not need to be copied on structure assignment. Every time the structure of the example above, is copied the fourth array element may take on unspecified values. This can lead to errors, which are very hard to find because the error probably manifests itself at a totally different position in the code. The runtime system, however, knows the memory layout of the structure and therefore knows of the padding bytes. So every access to a padding byte is recognized and can be immediately reported. Hence the programmer knows the cause of the error and can fix it easily.

5.4 Illegal Access to Allocated Memory

As discussed in Section 5.1 the runtime error detection system can detect accesses to non allocated memory. This catches most of the out-out-bound faults, because they often hit

unallocated memory. But this is not always the case, as can be seen in the following code example, which is part of the RTED test suite.

```
1    struct  { int arr[3]; int i; } s;
2    int * ptr = s.arr;
3    ptr += 3;
4    *ptr = 42;
```

Lets assume that the compiler generated a memory layout as it is shown in figure 5.1 on the left i.e. no padding areas have been inserted. Then, instead of writing to a fourth array element, as it was presumably intended by the programmer, the variable `i` is modified.

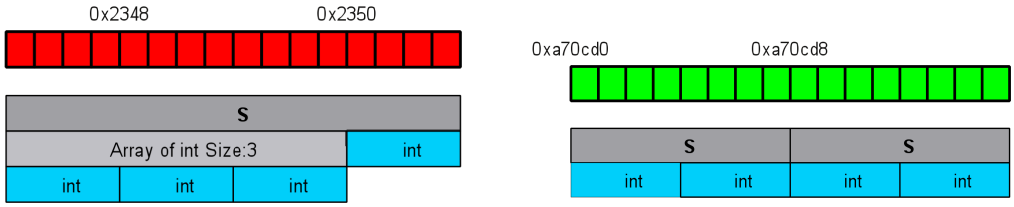

Figure 5.1: Memory layout of testcases

Code like this should be detected as an error, even if it would not lead to a segmentation fault in most cases. If the programmer would have intended to modify the variable `i`, he should have explicitly written that in the code.

To detect errors like this, the concept of a "memory chunk" is introduced. A memory chunk is a continuous memory region where a `struct`, a `class` or a stack-array is stored. Memory chunks can be nested, as it is the case in the code example above, where the array-chunk is contained by the `struct`-chunk. Now the cause of the error can be defined exactly: A pointer left the memory chunk, it previously pointed to, by using pointer arithmetic. The crucial memory chunk in this case consists of the array, where the size is explicitly known. The `arr` pointer first points to the beginning of the array, and by addressing the third element, the `arr` pointer leaves its memory chunk.

To illustrate the concept of the memory chunk more closely, I will use another code example:

```
1  struct S { int i1; int i2; };
2  S * sPtr = (S*) malloc(2*sizeof(S));
3  int * intPtr = (int*) sPtr;
4
5  intPtr += 2; // not legal
6  ++sPtr;      // legal
```

A pointer value change, which leaves a memory chunk is only invalid, if it is done using pointer arithmetic. Pointer arithmetic is considered to be every operation that modifies the pointer value in a relative way (i.e., addition or subtraction of an integer value). The assignment of a new value to a pointer is legal whether or not the new address lies in the same chunk. Relative modification of the pointer, however, is assumed to be an array-like access of the underlying memory, and bounds are checked in this case.

The size (granularity) of a memory chunk is defined by the type of the pointer. As demonstrated in the last code example, the integer pointer is not allowed to pass the struct boundary, a pointer to the struct may do so. These rules can be enforced with the information available to the runtime system. The same concept is used for detecting wrong indexing of static arrays.

```
1  int arr[3][3];
2  arr[1][4]=5;
```

In this code example, an array is indexed out-of-bounds, because the second dimension was declared to only have three entries, but the fifth element is accessed. The array elements lie consecutively in memory and when this code fragment is executed, no error will occur because the array access hits a valid memory location. For multidimensional arrays row-major storage order is used, so in the example above the element `arr[2][1]` is modified. This was presumably not the intend of the programmer, and at least a warning should be issued in this case. This is solved with the same mechanism as the `struct`-case above. Multidimensional arrays are considered to be built up out of memory chunks. Each row, column and in general every dimension is viewed as a memory chunk. Since array indexing in C/C++ is just pointer arithmetic, a simple array access could be rewritten in the following way:

```
1  int arr [3][3];
2  arr[1][4]=42;
3
4  int (* row_ptr)[3] = arr;
5  row_ptr += 1; // goto first row
6
7  int * col_ptr = (int*) row_ptr;
8  col_ptr += 4; // goto fourth element
9
10 *col_ptr=42;
```

In the fourth line a pointer to an array of size 3 is created. When incrementing this pointer the target address is incremented by the size of one row, in this case three times the size of an integer. After having reached the correct row, the right column is addressed. Of course this rewriting is not done explicitly in the instrumentation, but an array access check function is instrumented, which tells the runtime system the accessed indices. In the runtime library a test for each dimension takes place, to check if the corresponding pointer changes are legal.

Binary instrumentation tools cannot detect errors of this kind, since therefore type information is needed, which is not available in the executable. There exist approaches where these information is gathered from debug symbols. This is not always possible, and heuristics have to be used. For a detailed description see [NF].

5.5 Partially Instrumented Programs

Until now the assumption has been made, that the complete source code of the program is available and can be instrumented. This may not be the case, especially when closed source libraries are used. But even if the source code for the libraries is available, they still have to be instrumented and recompiled, which can be a time consuming task. In this section I will

discuss the problems, which occur when only parts of a program can be instrumented and how to handle this problems.

The first problem is, that the runtime system misses allocations and frees which leads to a wrong allocation status in the runtime system. To detect if a memory access is valid, it keeps a list of all allocated regions. This list is not correct any more, if allocations take place in non-instrumented source code. When memory is accessed, which has been allocated in a non instrumented code portion, the runtime system has no knowledge that this address is valid and incorrectly detects an error. It may also miss errors in case a free takes place in non-instrumented code, and afterwards the memory is accessed. So the basic problem here is that the heap allocation status cannot be tracked correctly due to non instrumented source code.

But there is a way to overcome this problem: The GNU C library provides a mechanism to modify the behavior of allocation functions by using so called hooks [1]. After having installed a hook, which is nothing more than passing a function pointer to a user defined function, this function is called instead of the allocation routine. In a partially instrumented program, the runtime system can now install hooks for heap memory handling functions to monitor all allocation operations. Also allocations with are done with the new-operator are covered by this method, because they are mapped to malloc calls, as long as the default new implementation is used, and the operator is not overloaded.

Installing hooks and monitoring the allocation behavior this way is not a good substitution for the instrumentation in the general case, because by instrumenting the allocation function calls, additional information like the exact code position (file name and line number) can be gathered.

The second class of problems which can occur when running partially instrumented programs are due to incomplete initialization information. Memory accesses which occur in non instrumented parts of the code cannot be monitored. Write operations can take place without being registered to the runtime-system. By tracking these write operations the initialization status of all variables and heap regions is updated. If a variable is initialized in a non-instrumented code section, the variable will still be flagged as uninitialized in the runtime-system. If this variable is read afterwards an error is detected. This problem cannot be overcome in general. Therefore, access to uninitialized memory is only reported as a warning instead of an error, when a program is partially instrumented. However, there are some techniques used to reduce these wrong warnings. A very effective, but tedious method is, to extend the instrumentation tool and runtime system, such that the behavior of all library functions is hard coded into them. This approach makes only sense for very common libraries and has been done for the C standard library.

With this method it is also possible to detect errors, which arise when library functions are called in an incorrect way. A good example for this is the `strcpy` function, which copies a null terminated C string. A call to this function is recognized by the instrumentation tool and is replaced by a wrapper function, which is part of the runtime library. This reimplemented `strcpy` function first checks if the source string can be correctly accessed, which means that the memory from the start address until the next null character is correctly allocated. Then it is verified that the source and target region do not overlap. If the target region also lies in a valid memory region, and is big enough to store the source string, it is marked as initialized and finally the original `strcpy` function is called. This is done for all functions of the C standard

[1] http://www.gnu.org/s/libc/manual/html_node/Hooks-for-Malloc.html

library.

Of course it is a tedious task to write wrapper implementations for all functions of a library, but this way also a clean interface can be defined and it is clearly specified what memory regions a function is allowed to access. Some functions may be very complex such that it is practically not possible to exactly define the memory access behavior. Thus the best solution still is to instrument the complete code of a project, including used libraries.

5.6 Results of RTED Test Suite

The prototype implementation described in the sections above was now tested with the RTED test suite of the Iowa State University. In the following tables the results for the serial (non-parallel) C and C++ test cases can be seen.

	tests	passed	detection rate
allocation deallocation errors	25	24	96 %
array-index out of bound	216	216	100 %
C99 specific errors	72	71	99 %
floating point errors	6	0	0 %
input output errors	10	10	100 %
memory leaks	15	15	100 %
pointer errors	120	119	99 %
string errors	40	40	100 %
subprogram call errors	19	18	95 %
uninitialized variables	87	81	93 %

Table 5.1: Results of RTED C test cases

	tests	passed	detection rate
allocation deallocation errors	109	104	95 %
array-index out of bound	332	329	99 %
input output errors	30	20	67 %
memory leaks	40	36	90 %
pointer errors	157	155	99 %
string errors	40	40	100 %
uninitialized variables	221	213	96 %

Table 5.2: Results of RTED C++ testcases

Most test cases of the RTED test suites are passed, however there is a small amount of test cases where the current implementation still fails. These failures are mostly due to some unhandled special cases, which have not been implemented yet. There is a class of errors which is not detected by the runtime-error checking system at all: floating point errors. These errors occur, when a division by zero, an underflow or overflow in a floating point variable happens

and are deliberately not handled by this tool, because there already exist good methods to detect and fix these errors. They are detected by the floating point unit of the processor, and the program can be configured such that a signal is sent to the process in this case. Depending on the architecture and the compiler, there are different ways to terminate a program when these errors occur. When using the GNU compiler on UNIX-systems, floating point exceptions can be enabled like this [2] :

```
1  #include <fenv.h>
2  // ...
3  feenableexcept (FE_INVALID|FE_DIVBYZERO|FE_OVERFLOW);
```

[2] http://gcc.gnu.org/onlinedocs/gcc-3.4.6/g77/Floating_002dpoint-Exception-Handling.html

6 Conclusion

This thesis demonstrates that runtime error checking systems, which use source code instrumentation, can detect all common runtime errors that occur in C/C++ programs. A prototype has been implemented and tested using the RTED test suite, where it passes the majority of the test cases.

Binary checkers have advantages when it comes to usability aspects. There is no need to recompile the program and an error detection run does not require more effort than a normal production run. Whereas, when using the source code instrumentation tool, all source files have to be processed first and then the whole project including the libraries, has to be recompiled. The developed source code instrumentation tool has to track more information (especially type information) and has to add additional function calls at each memory access. This impacts performance massively: The current implementation is much slower than comparable binary checkers. There is still room for performance improvements, especially in the memory-manager of the runtime-system library, where every memory access is checked. When implementing the runtime library, the focus was mainly put on functionality, not on performance.

Another possibility to improve performance is, to check only some parts of the project. For example, interfaces could be created for the used libraries (Section 5.5) and then only the user-written code is checked. Here further work is necessary to make the specification and implementation of these library interfaces easier and more intuitive for the user. Using this technique, only a small portion of the whole program can be instrumented and tested, whereas binary error checkers always have to check the whole program. Often a big amount of the total runtime is spent checking library code, which is assumed to be free of errors, and in most cases cannot be changed anyway.

Source code instrumentation tools have more information available from the original source code than binary checkers. Especially type information, which can be very useful for error detection, is lost during compilation. By using this additional information more errors can be detected. This makes it worth to improve the usability and performance of these tools, such that they become a serious alternative to binary instrumentation tools.

7 Bibliography

[BNK+07] M.D. Bond, N. Nethercote, S.W. Kent, S.Z. Guyer, and K.S. McKinley. Tracking bad apples: reporting the origin of null and undefined value errors. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, page 422. ACM, 2007.

[edg10] Edison design group, February 2010. `http://www.edg.com`.

[Int99] International Organization for Standardization. *ISO/IEC 9899-1999: Programming Languages—C*, December 1999.

[Int03] International Organization for Standardization. *ISO/IEC 14882:2003 : Programming Languages—C++*, October 2003.

[LCH+] G.R. Luecke, J. Coyle, J. Hoekstra, M. Kraeva, Y. Xu, M.Y. Park, E. Kleiman, O. Weiss, A. Wehe, and M. Yahya. The Importance of Run-time Error Detection.

[LCH+06] G.R. Luecke, J. Coyle, J. Hoekstra, M. Kraeva, Y. Li, O. Taborskaia, and Y. Wang. A survey of systems for detecting serial run-time errors. *Concurrency and Computation*, 18(15):1885, 2006.

[LYHR01] A. Loginov, S.H. Yong, S. Horwitz, and T. Reps. Debugging via Run-time type checking. *Lecture Notes in Computer Science*, pages 217–232, 2001.

[NF] N. Nethercote and J. Fitzhardinge. Bounds-checking entire programs without recompiling. In *Proceedings of the Second Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE 2004)*. Citeseer.

[NS07] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *ACM SIGPLAN Notices*, 42(6):100, 2007.

[ros10] Rose, February 2010. `http://www.rosecompiler.org`.

[SQ03] Markus Schordan and Daniel Quinlan. A source-to-source architecture for user-defined optimizations. In *JMLC'03: Joint Modular Languages Conference*, volume 2789 of *Lecture Notes in Computer Science*, pages 214–223. Springer Verlag, August 2003.

[Ste92] JL Steffen. Adding Run-time Checking to the Portable C compiler. *Software, practice & experience*, 22(4):305–316, 1992.