Fighting off memory leaks and errors (C++11)

Programming (//kacperkolodziej.com/programming) 09/02/2015 10:04

C++ (//kacperkolodziej.com/tag/cpp) C++11 (//kacperkolodziej.com/tag/cpp11)

valgrind (//kacperkolodziej.com/tag/valgrind) programming (//kacperkolodziej.com/tag/programming)

Modern tools provided with C++11's standard library make fight with memory leaks and errors easier and more effective. Sometimes problems which are seemingly not dangerous might put an end to your application. We are going to learn how to find and avoid them.

What is a memory leak?

C++ language let us reserve and release memory by ourselves. Each variable has its own area in memory. This area is released when application goes out of scope of its availability which is limited by `{` and `}` characters. There is no risk that memory leaks will raise, because memory is released automatically. However, C++ allow us also to reserve memory dynamically. It means that you can receive the amount of memory which you need, and it would not be released automatically by system. So it means that you are obliged to release this area when you don't need it. If you forget about it, you will cause memory leak.

Effects of memory leaks

Each memory leak affect in bigger memory usage than its needed. Depending on how much memory we lose leaks are more or less dangerous. You should notice that function which causes memory leaks can cause huge lost of memory when it is called hundreds or thousands times during runtime. Of course after program termination garbage collector will take care of memory which hasn't been freed. So why we are worry over memory which has not been released? In case of applications of everyday use small memory leaks are unnoticeable. Situation gets worse when programs works with

no break for weeks or even months (e.g. http servers, dns, e-mail servers, databases etc.). If such application doesn't release unnecessary areas in memory it causes larger memory usage. It can affect on system killing application which may have had very important function in system.

Situation is even worse in case of applications which use a lot of resources. Good example of such program can be 3D rendering software. Process of creating 3D scenes requires a lot of RAM. If this software allow itself not to release memory when it is not requisite, we will lose many hours of hardware's work.

Detecting

To detect memory leaks (and errors) we can use one of dedicated applications like valgrind. This software is available for operating systems with Linux's kernel and others compatibile with UNIX. To test our program with valgrind we need its executable file (binary file).

How to use valgrind?

First of all you have to know that valgrind tests program during its runtime. It's interested in neither source code nor binary code. We just have to use the application. What might be a problem is that leaks or errors will not be detected if function which cause them stay unused during runtime. The easiest way to omit this issue is to use all functions in simple testing application. If your code is more sophisticated and has a lot of modules. You should write a few smaller testing apps - one or two per each module.

If you write unit tests and use for this purpose a good library which does not generate neither error nor leaks, you can use these tests with valgrind. Running it looks like this:

```
1  valgrind [valgrind's options] /path/to/app apps startup params
```

To check valgrind's options type:

```
1  valgrind --help
```

If you want to get detailed information about leaks, use option: `--leak-check=full` .

Examples of memory leaks

Example 1

```
1  #include <iostream>
2
3  int main(int argc, char** argv)
4  {
5    int *x = new int[1000];
6    {
7      int y;
8      int *z = new int[1000];
9    } // z pointer i y varible are removed from memory
10
11   delete [] x; // releasing memory allocated in 5th line
12   return 0;
13 }
```

In this program we create 3 variables. Two of them (x and z) are pointers. We create also one scope with braces. At the beginning we allocate in memory 1000 int objects. Address of first is in x pointer. In scope we declare varibale y and pointer z. To pointer we write address of next 1000 objects of int. As I have mentioned before when program goes out of scope varibales are removed from memory. It also concerns pointers. Pointer z is deleted, but memory on which it pointed not. In 9th line with get first memory leak. We have lost address of 1000 objects of int. We cannot release this memory now.

Example 2

Even if we take care of releasing memory by using delete operator we can meet with exception between allocation and deallocation. When exception is thrown program go back to the place where exception is caught or program is terminated if such does not exist. We can assume that program is well written and supervises exceptions to be caught in catch block. If exception is supported program will probably still work and memory will not be released because program hasn't reached delete operator. Example:

```
 1 #include <iostream>
 2 #include <stdexcept>
 3
 4 int main(int argc, char** argv)
 5 {
 6   try {
 7     int *x = new int[1000];
 8     throw std::logic_error("logic_error");
 9     delete [] x;
10   } catch (std::logic_error &e)
11   {
12     std::cerr << "an exception has been caught: " << e.what() << "\n";
13     return 1;
14   }
15 }
```

Situation has been intended. In 8th line we could have had other function call which probably could throw exception. If program will still work such cases will appear more times which will cause huge memory leak.

Example 3

This code is very similar to the previous. It better depicts situation when programmer is not conscious that exception might be thrown. We declare class which allocates specified amount of memory in constructor and release this area in destructor. Class has also `get` function which returns element of array with `index` passed as argument. If choosen element doesn't exist function throws `std::out_of_range` exception. This is code of this class with example of use:

```
 1  #include <iostream>
 2  #include <stdexcept>
 3
 4  class Database
 5  {
 6  private:
 7    int size;
 8    int* data;
 9
10  public:
11    Database(int n) :
12      size(n),
13      data(new int[n])
14    {}
15
16    int get(int i)
17    {
18      if (i < 0 || i >= size)
19      {
20        throw std::out_of_range("Database");
21      }
22
23      return data[i];
24    }
25
26    ~Database()
27    {
28      delete [] data;
29    }
30  };
31
32  int main(int argc, char** argv)
33  {
34    try {
35      Database *db = new Database(10);
36      db->get(10);
37      delete db;
38    } catch (std::out_of_range &e)
39    {
40      std::cerr << "out_of_range in " << e.what() << "\n";
41      return 1;
42    }
43    return 0;
44  }
```

We can ignore fact that dynamic allocation of Database object makes no sense here. We could have done it with standard object allocated on stack. Let's imagine that we had serious reason why we

used pointer and dynamic allocation. When we have database with 10 elements, the highest index we can get is 9. Trying to get 10th element will cause exception. Valgrind's report confirms leak.

In above three cases we could avoid leaks.

1. First way to avoid problems is to create pointer out of scope in which we had it currently. It will allow us to release memory in different scope than it was allocated, because we have access to pointer which has its address. Despite that fact we have to remember to delete this memory! We can meet exception on the way to delete place and we have leak.
2. Another solution is to use so-called handler to resources. Handler is nothing more than container from standard library or other class which manages resources. We can allocate such handler on stack. When program goes out of scope in which handler has been initialized its destructor releases memory dynamically allocated by constructor. It gives us 100% confidence of no memory leaks. The exception from this rule is when we keep pointers to dynamically allocated memory in such container, because container will remove only pointers.
3. If we have to use pointers it might be reasonable to look at C++11's 'smart pointers'. These pointers release memory on which they show automatically when they are destructed. I'm going to tell you something about them in further part of article.

The best way is avoiding explicit dynamic allocation od memory. We cannot work without dynamic allocation at all, but we can minimize its occurance number. A lot of inexperienced programmers treat presence of pointers and dynamic allocation in C++ as if they were necessary everywhere. It affects in usage of new operator in many places where standard object allocated on stack would be good enough and safe solution. Third example is a great case of code which corresonds with this rule. We wouldn't have any problem with not released memory. After going out of try...catch block it would be removed from memory.

Leaks aren't the only problem with memory.

Memory corruption

Another mistake we can make is using memory which doesn't belong to us. This mistake is often made by beginning programmers who has just started coding and are not used to fact that maximum index of array whith 100 elements is not 100, but 99. This error might be detected during runtime but it's not a rule. Segmentation fault message occurs only when program tries to use memory assigned

to the other process. It doesn't mean that everything is OK when we use in bad way our own memory. Let's look at the example.

Example 1

This is mistake of 'beggining programmer':

```
1  int main(int argc, char** argv)
2  {
3    int tab[100];
4    tab[100] = 10;
5    return 0;
6  }
```

This program will probably run and exit with 0 code - no errors. However, valgrind will communicate *invalid memory usage*:

```
1  ==7933== Invalid write of size 4
2  ==7933==    at 0x400623: main (example5.cpp:4)
3  ==7933==  Address 0x595a1d0 is 0 bytes after a block of size 400 alloc'd
4  ==7933==    at 0x4C28147: operator new[](unsigned long)
5  (vg_replace_malloc.c:348)
6  ==7933==    by 0x400614: main (example5.cpp:3)
```

In this case nothing serious had happened, but when we write memory out of array's range we will probably overwrite something. Next example illustrates such case:

Example 2

```
#include <iostream>

int main(int argc, char** argv)
{
  int a[100];
  int b[100];
  b[0] = 1;
  std::cout << "b[0] = " << b[0] << "\n";
  a[100] = 2;
  std::cout << "b[0] = " << b[0] << "\n";
  std::cout << "Comparison of a[100] and b[0] addresses:\n";
  int* p1 = &a[100];
  int* p2 = &b[0];
  if (p1 == p2)
  {
    std::cout << "Addresses are the same! (" << reinterpret_cast<void*>(p1) << ")\n";
  } else
  {
    std::cout << "Addresses are different!\n";
  }
  return 0;
}
```

As we can see, `b[0]` has been overwritten because of using bad indexing in `a` array. By using `a[101]` we access `b[1]`. In this example valgrind would not signal error! Due to the fact that memory belongs to current process and has been initialized (we have write to `b[0]` number 1), error couldn't be discovered.

When we use dynamic allocation of memory, going out of range will surely cause valgrind's reaction.

Example 3

```
int main(int argc, char** argv)
{
  int* p = new int[10];
  p[10] = 10;
  int a = p[10];
  return 0;
}
```

In this code we have two errors. First occurs when we try to write number 10 in `p[10]` (just one

'place' after allocated area). Second error occures when we try to read value from this address. Both operations will succeed only if area after memory reserved in third line belongs to current process. Looking at this code allow us to think that trial will succeed. Heap in new operating systems is alway bigger than 40 bytes which are used by 10 integers.

As I mentioned before, such errors causes reaction of valgrind. This is a piece of returned warning:

```
1  ==8840== Invalid write of size 4
2  ==8840==    at 0x400621: main (example6.1.cpp:4)
3  ==8840==  Address 0x595a068 is 0 bytes after a block of size 40 alloc'd
4  ==8840==    at 0x4C28147: operator new[](unsigned long) (vg_replace_malloc.c:348)
5  ==8840==    by 0x400614: main (example6.1.cpp:3)
6  ==8840==
7  ==8840== Invalid read of size 4
8  ==8840==    at 0x40062B: main (example6.1.cpp:5)
9  ==8840==  Address 0x595a068 is 0 bytes after a block of size 40 alloc'd
10 ==8840==    at 0x4C28147: operator new[](unsigned long) (vg_replace_malloc.c:348)
11 ==8840==    by 0x400614: main (example6.1.cpp:3)
```

Similar error occurs when process try to write something in just freed memory.

Example 4

```
1  int main(int argc, char** argv)
2  {
3    int* p = new int[10];
4    delete [] p;
5    p[2] = 10;
6    return 0;
7  }
```

Valgrind's log is slightly different than last:

```
1  ==9086== Invalid write of size 4
2  ==9086==    at 0x400684: main (example9.cpp:5)
3  ==9086==  Address 0x595a048 is 8 bytes inside a block of size 40 free'd
4  ==9086==    at 0x4C275BC: operator delete[](void*) (vg_replace_malloc.c:490)
5  ==9086==    by 0x40067B: main (example9.cpp:4)
```

Not initialized memory

How many times have you heard: 'set variable's value after it's creation!'? Almost every programming textbook mention this rule. It will be less harmful when you fail the exam from algorithms because you wrote `int sum;` instead of `int sum = 0;` . It will be worse if you lose many hours because of not initialized memory. When we use not initialized pointer which shows wherever, it is highly probable that we will use another's process memory and our process will be dumped by operating system. Normal variable which was initialized by us surely belongs to our process so lack of initialization can be more troublesome.

```
1  int main(int argc, char** argv)
2  {
3    int x;
4    if (x == 10)
5    {
6      x = 20;
7    }
8    return 0;
9  }
```

Fortunately valgrind inform us about using uninitialized variable or pointer:

```
1  ==8777== Conditional jump or move depends on uninitialised value(s)
2  ==8777==    at 0x4005AD: main (example7.cpp:4)
```

Preventing memory errors

Preventing memory errors in C++ programs is possible even without specialised tools. Presence of destructors which are often underestimated by programmers is very helpful. If you create object which allocate memory by itself, its destructor should take care of releaseing this memory. These are some good prctices which using should prevent you from leaks and errors:

1. If you use new operator in constructor, use delete in destructor in proper way.
2. If you use local pointer in member function (which is not an attribute of current class), it is highly probable that local variable is good enough.
3. If you are sure that in situation from 2nd point local variable is not good enough, consider using

delete operator before leaving this function.
4. If you need to allocate dynamically memory in function and cannot free it before leaving scope, save address in external pointer which is available out of function.

C++11 facilities

We have standard library's containers which release memory automatically. Using them skilfully will prevent you from problems with memory management. It can also be good to use `std::vector::at` function instead of operator `[]`. First of them checks if index is not out of vector's range. In case of too big or too small index it throws exception.

From C++11 standard we have also smart pointers which automatically release memory which they points to. I can describe their functionality with two simple rules:

- `std::shared_ptr` can be copied. When last copy is removed, memory which it used to point is released,
- `std::unique_ptr` cannot be copied (it's unique). When it is removed, memory which it used to point is also released.

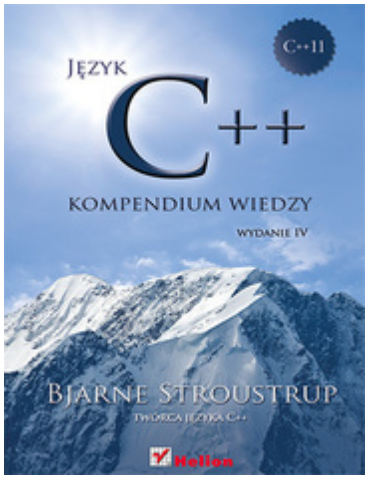
It is very important to be consistent and use only pointers smart pointers for one place in memory. Using standard pointers with shared or unique pointers can cause serious problem (e.g. double freeing memory). You should use `std::make_unique` and `std::make_shared` functions to create intelligent pointers smart pointers. You mustn't use them to point memory allocated on stack and memory which is released manually (by delete operator).

Following above rules should allow you to elliminate problems with memory. Facilities from C++11 and C++14 causes that we have no good explanation for memory leaks. I encourage you to learn something about new C++ standards tools . Using their reasonably will make your code better.

See also

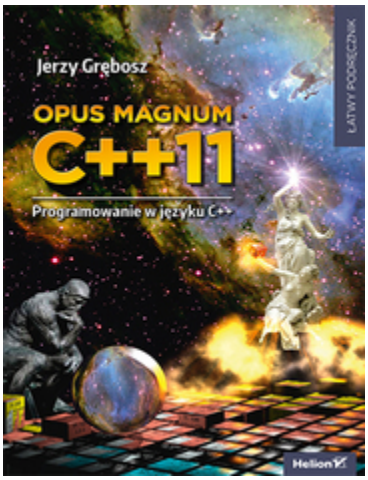
(//kacperkolodziej.pl
(https://github.com
/kolodziej.atom.xml)

Recommended books



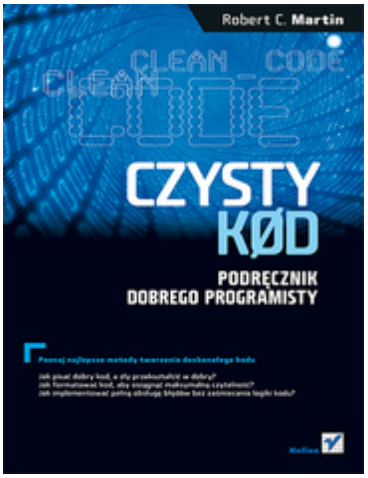
(http://helion.pl/view/5988w/jcppkw.htm)

Buy for 149.00 PLN (http://helion.pl/add/5988w/jcppkw)



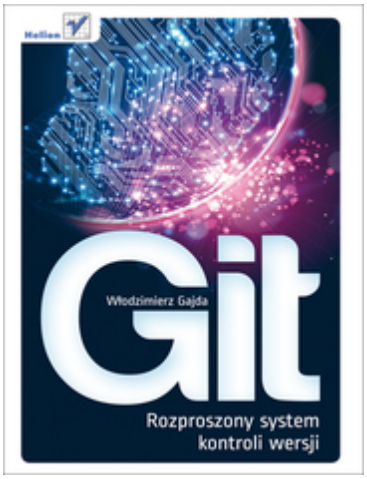
(http://helion.pl/view/5988w/ocpp11.htm)

Buy for 149.00 PLN (http://helion.pl/add/5988w/ocpp11)



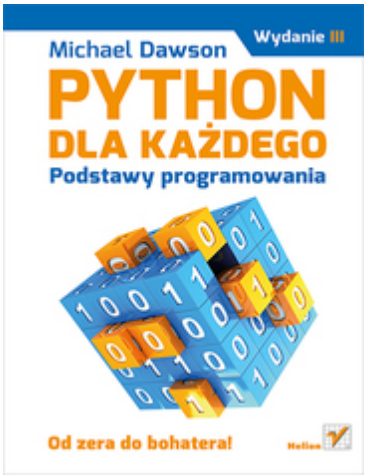
(http://helion.pl/view/5988w/czykov.htm)

Buy for 69.00 PLN (http://helion.pl/add/5988w/czykov)



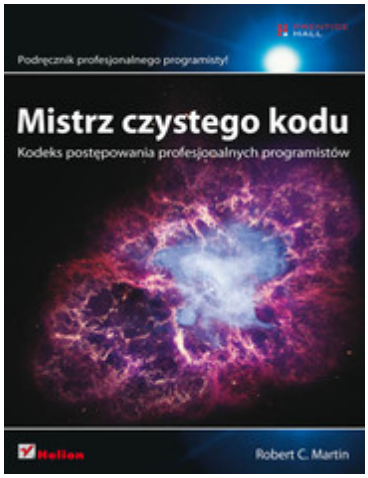
(http://helion.pl/view/5988w/gitroz.htm)

Buy for 54.90 PLN (http://helion.pl/add/5988w/gitroz)



(http://helion.pl/view/5988w/pytdk3.htm)

Buy for 69.00 PLN (http://helion.pl/add/5988w/pytdk3)



(http://helion.pl/view/5988w/mckkod.htm)

generated with pelican (http://getpelican.com/)