Introduction to debugging a C/C++ program

by SAM
sam@cs.vu.nl

version 1.1

August 13, 1996

1 Introduction

> "Languages that prevent programming errors, a goal advanced by some inveterate language designers, are as feasible as knives that can cut meat but not hands. We need sharp tools to do good work."[1]

C and C++ are very sharp tools; when used correctly, they can be used to do good work. But as with all sharp tools, it's easy to cut yourself.

This document will describe how to debug an application written in C or C++. Most of this document is not C/C++ specific, so the techniques described can be used with other languages as well.

First, the most commonly made mistakes in programming in C and C++ will be mentioned, together with some simple solutions. Next, different methods of finding an error are discussed.

Since it's not feasible to describe every compiler and debugger available, some choice had to be made. Because the GNU versions of these tools are available on almost every platform, the gcc/g++ compiler and the GDB debugger will be used. However, differences with other Unix compiler and debuggers, eg. CC and dbx, are only minor.

2 Most commonly made mistakes

Both C and C++ allow for many little mistakes, not many directly obvious. These mistakes can be of two types: syntactic and semantic errors. This section will describe these both types of errors. First, errors that are common to both C and C++ and next, errors that are C++-only errors.

2.1 Syntactic mistakes

Syntactic mistakes are errors in the syntax of C and C++, which means there's something wrong in the way *how* the statements are written. Syntactic mistakes are mostly typo's that will result in errors when a sourcefile is compiled. It's best to always use the compile flag `-Wall` to instruct the compiler to find as much probable mistakes as possible.

2.1.1 C/C++

- Forgetting a semicolon at the end of the line. Can cause various strange compile errors.
- Putting a semicolon after the function header of the implementation of a function:

  ```
  void foo();
  {
      cout << "foo" << endl;
  }
  ```

  This will result in an error like `test.cc:2: parse error before ‘{’`.
- Forgetting to terminate a string constant, for example

  ```
  void foo()
  {
      cout << "hello world! << endl;
  }
  ```

  This will result in an error like

  ```
  string.cc:5: unterminated string or character constant
  string.cc:5: possible real start of unterminated constant
  ```
- No `;` at the end of an enum/union/struct definition. Can result in lots of compile errors.
- Entering too many or too few `(`, `)`, `{` or `}`. This will also produce lots of compiler errors. A possible solution is using an editor with the ability to match `()` and `{}`, like vi and emacs.
- Recursive including of `.h` files. Will generate an error like
  `header.h:1: macro or ‘#include’ recursion too deep`.
  Solution: put the contents of each header file between a `#ifndef` command, like

  ```
  #ifndef _filename_H_
  #define _filename_H_
  :
  contents of .h file
  :
  #endif
  ```
- Including a header file multiple times. This can sometimes lead to warnings or errors. Solution: same as with recursive including.
- Defining functions without an explicit return type. Functions and variables that are declared without a type default to `int`, not to `void`.
- Defining functions and implementing them with some other return type. For example defining a function in a header file like this:

```
int the_answer();
```

and implementing it like this:

```
unsigned int the_answer()
{
    return 42;
}
```

This will result in an error like:

```
return.c:10: conflicting types for ‘the_answer’
return.c:5: previous declaration of ‘the_answer’
```

The solution is to make the two return types the same.

2.1.2 C++

- Declaring a static member variable, but not reserving space for it in the implementation of the class. This will result in an error when linking: `/tmp/cca001861.o(.text+0x4): undefined reference to ‘A::Aint’`.
- No `;` at the end of a class definition. Can result in lots of compile errors.
- Forgetting to write the classname in front of the function name in the implementation of that member function, for example `int insert_element() { .. }` instead of `int set::insert_element() { .. }`.
- Defining a member function like `int foo() const;` and implementing it as `int bar::foo() { ... }`. The `const` should also be added to the function-header of the implementation.

2.2 Semantic errors

Semantic errors are errors in the semantics of a program, which means there's something wrong in *what* is written in the program. This will mostly be typo's, small errors in an algorithm, design-flaws and incorrect assumptions. Of course, these kind of errors can not all be described here. I will only describe the most common errors that can be put in the category 'slip of the pen' (or sometimes 'slip of the mind').

2.2.1 C/C++

- Putting an extra semicolon after the heading of an `if` or `while` statement, for example

  ```
  if (x > 0);
  {
      x++;
      foo(x);
  }
  ```

  will result in *always* executing the block following the `if`. Something similar can be said for `while` statements.

- Writing `if (x = 0)` instead of `if (x == 0)`. When compiled with `g++ -Wall`, this construction will result in a warning: 'suggest parentheses around assignment used as truth value'. The same goes for `while`.

- Assuming wrong precedence of operators. For example, the construct `if (mask & 0x80 == 0)` looks ok, but it is not, because `==` has a higher precedence than `&`. This construct is in fact equivalent to `if (mask & (0x80 == 0))` and will always be false (left as an exercise for the reader). The obvious solution to this problem is to use parentheses around (complex) expressions before and after the `==` as in `if ((mask & 0x80) == 0)`.

- No `break` at the end of a case-statement; a `switch` such as

  ```
  switch(value)
  {
      case 0:
          printf("value = 0\n");
      case 1:
          printf("value = 1\n");
      default:
          printf("value is not 0 or 1\n");
  }
  ```

  will, when `value` equals 0, produce

  ```
  value = 0
  value = 1
  value is not 0 or 1
  ```

  instead of the 'expected' output

  ```
  value = 0
  ```

  The compiler will not generate a warning for this, since it's perfectly legal.

- Mixing up the logical operators (`&&`, `||`) and the bitwise operators (`&`, `|`).

- Calling a format-function with too few parameters, e.g. `printf("debug: x = %d\n");`. This will result in some random value printed out (depending on the contents of the stack). Fortunately, when compiling with `-Wall`, gcc will warn you:
  `test.cc:5: warning: too few arguments for format`.

- Writing `if (function_name)` instead of `if (function_name())`. The first is also legal in C/C++ and will not generate any errors or warnings.

- Spreading C-comments across multiple lines, thereby putting some statements in comment:

  ```
  int foo(int x)
  {
      int x2, y;

      x2 = x * 3.1415;           /* we're doing some
      y = x2 * x + x * x + 1.1;     pretty extensive
      x = abs(x2) > 1 ? 0 : 1;      calculations here */
      return y * x;
  }
  ```

This looks very nicely formatted, but it won't return the correct result. It's much safer to always begin and end a comment on the same line, or use the `//` when programming in C++.

- Using arrays as 1-based instead of 0-based arrays: `int intarray[10];` is an array that ranges from 0 to 9, not from 1 to 10. Accessing the 10th element when there is none can result in unpredictable behaviour, depending on the variables stored in memory that are immediately following that array. Possible behaviour: memory faults/segmentation violations[1] (memory faults will happen when a program tries to read from or write to memory that does not belong to that program), runtime errors in system-functions like *malloc*, other variables changing values when they should not.

- Returning local arrays, for example

```
char* int2string(int i)
{
    char buffer[64];

    sprintf(buffer, "%d", i);
    return buffer;
}
```

  Local arrays (in C these are called *automatic variables*) are allocated on the stack and are removed as soon as the function returns. The pointer (in this case `buffer`) will then be invalid. This will work:

```
char* int2string(int i)
{
    char* buffer;

    buffer = new char[64];
    sprintf(buffer, "%d", i);
    return buffer;
}
```

- Passing a variabele to a function where an address is required, e.g. `scanf("%d", value)` instead of `scanf("%d", &value)`. The first will most of the time result in a memory fault. When compiling with `-Wall`, gcc will give a warning:
  `test.cc:8: warning: format argument is not a pointer (arg 2).`

2.2.2 C++

- Not making a member function virtual when it should be. This will result in always calling the member function of the superclass.

- Not making a destructor virtual when using polymorphism, for example:

```
#include <stream.h>

class A
{
public:
```

[1]since memory faults and segmentation violations are pretty much the same (although I'm sure the experts would argue that :), I will from now on only use the term *memory fault*.

```
    A() { cout << "A::A()" << endl; }
    ~A() { cout << "A::~A()" << endl; }
};

class B: public A
{
public:
    B() { cout << "B::B()" << endl; }
    ~B() { cout << "B::~B()" << endl; }
};

int main()
{
    A* a = new B;
    delete a;
}
```

will result in:

```
A::A()
B::B()
A::~A()
```

When class B needs to do something in its destructor (for example cleaning up some variables), this is definitely not what you want. Declaring `~A` as virtual will result in:

```
A::A()
B::B()
B::~B()
A::~A()
```

- Hiding a member variable in a member function. For example

```
class A
{
public:
    A() { int i = 3; }
    int geti() { return i; }

private:
    int i;
};
```

When an instance of class `A` is initialized, calling `geti` will most certainly *not* return 3, since it's not the member variable `i` that is set in the constructor, but a *local* variable `i`. Declaring a variable `int i` will hide the member variable `i`. A solution to this is to append or prepend an underscore '_' to each member variable and only to member variables; `i_` will not clash with `i`.

- Forgetting to initialize an object-pointer. This may have strange effects: as long as no member variables are accessed, everything will be ok:

```
#include <stream.h>
```

```
class A
{
public:
    void func1() { cout << "A::func1, no problem" << endl; }
    void func2(int newa) { a = newa; }
private:
    int a;
};

int main()
{
    A* aptr = 0;

    aptr -> func1();
    aptr -> func2(27);

    return 0;
}
```

When the program is run, the output will be:

```
A::func1, no problem
Memory fault - core dumped
```

The example shown here is bit more extreme than just not initializing; the object-pointer is a NULL-pointer. When the initialization of **aptr** to 0 is removed, the resulting behaviour is unpredictable; the program may or may not crash, depending on the value of the uninitialized pointer.

The solution to this problem is to *always* initialize object-pointers. If an object-pointer can not have a 'decent' value (an allocated object) assigned to it, assign 0 to it; this will at least detect unauthorized access to the object-pointer.

3 Debugging a compiled program

3.1 Think before you do

This means that you should always test the assumptions you're making when programming. This can range from using certain not-so-well-documented features of system-calls, to pre- and post-conditions in your own code. It's more easy to find a bug in a small program to test the exact behaviour of `sprintf`, than to have to find out that `sprintf` is not behaving as you expected. It's also helpful to make a good design of your program so you can see what obscure features of the system (or maybe of someone else's code) you will be needing. More about design is explained in the next section.

3.2 Careful design

What helps most in debugging a program is to have a program that let itself be debugged easily. This means that your program should be written in such a way that errors are easy to locate. This document is not about designing a program, so I will not describe ways to integrate debugging possibilities into your design, but during design, you should now and then ask yourself the question if the resulting program is easy to debug.

3.3 Careful reading

When done carefully, reading source can be extremely useful when debugging. This should not be done on a terminal, but on plain paper with no terminal in sight. Debugging on paper gives a better overview of the program and forces you to think about what really happens in a piece of code instead of 'let's try if this will work'.

3.4 Producing debug-output

Producing debug-output is very useful for monitoring the runtime state of a program. At tactical points in the source printf- or cout-statements are placed that will show when some specific code is executed. It's also very practical to print out the value of key-variables to see what value they have and how they change.

3.5 Using assertions

Another possibility is to insert *assertions* in your code that check if the assumptions of your code are valid. An example:

```
#include <assert.h>

int mydivision(int numerator, int divisor)
{
    assert(divisor != 0);

    return numerator / divisor;
}
```

If *divisor* is not equal to 0, the assert will do nothing, but when *divisor* is 0 for some reason, the program will halt with a message like:

```
assertion.cc:8: failed assertion ‘divisor != 0’
IOT trap - core dumped
```

This can be very useful while debugging, but asserting every assumption in your code will require a huge amount of asserts and furthermore, it can be argued that it will *not* result in robust programs; a program that halts as soon as something is wrong can not be said to be very robust.

3.6 Isolating the bug

A bug can be more easily fixed when it's known in what part of the program it resides. Using debug-output and assertions can be used to locate the part of the program where the bug manifests itself. When the part has been narrowed to a single procedure or a couple of instructions, careful reading will result in finding the bug most of the time.

3.7 Using a debugger

A debugger can be very useful for finding certain bugs (memory faults, segmentation violations and bus errors) and for tracing certain parts of code.

3.7.1 Memory faults

A memory fault occurs when a program is trying to read from or write to memory that does not belong to that program. A standard example is reading/writing address 0 (ie. a NULL-pointer), this will always result in a memory fault. A debugger can catch a memory fault and is therefore a useful tool to quickly determine where in your program the memory fault occurs.

Suppose we have a very small program (`bug.c`) with a (obvious) bug:

```
/* simple buggy program to demonstrate GDB */

#include <stdio.h>


void printptr(int* iptr)
{
    printf("%d\n", *iptr);
}


int main()
{
    int a = 5;
    int* ptr = 0;

    printptr(&a);
    printptr(ptr);
}
```

After compiling and running, it will generate a memory fault:

```
$ gcc -o bug bug.c
$ bug
5
Memory fault - core dumped
$
```

How to find the bug with a debugger? First, make sure the executable contains debug information. This is done by providing some special flag (`-g`) to the compiler:

```
$ gcc -g -o bug bug.c
```

Next, run the debugger with the executable as argument:

```
$ gdb bug
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.13 (sparc-sun-solaris2.3),
Copyright 1994 Free Software Foundation, Inc...
(gdb)
```

(gdb) is the prompt of the debugger; you have to type the commands you want to give at the prompt.

Now tell the debugger to run the loaded program:

```
(gdb) run
Starting program: /home/se/doc/debug/source/bug
5

Program received signal SIGSEGV, Segmentation fault.
0x106b4 in printptr (iptr=0x0) at bug.c:8
8           printf("%d\n", *iptr);
(gdb)
```

The debugger stops at the offending instruction, here `printf("%d\n", *iptr);`. It states in which function the statement resides and with what arguments that function was called. It's immediately clear what the problem is: iptr is a null-pointer. This can be verified by telling the debugger to print the value of iptr:

```
(gdb) print iptr
$1 = (int *) 0x0
(gdb)
```

When the offending statement is in some function that's deeply nested, it can be useful to know the calling sequence from main to the that function. This is done by giving the command `backtrace` (abbreviated to `bt`):

```
(gdb) bt
#0  0x106b4 in printptr (iptr=0x0) at bug.c:8
#1  0x106f4 in main () at bug.c:18
(gdb)
```

3.7.2 Debugging, a core business

When a program crashes, it will create a *core dump*. A core dump is a memory image of the code and data of the program when it crashes. This core dump can be used by a debugger to locate the offending instruction without having to run the program within the debugger. Make sure that core dumps really get created; it is possible to limit the coredumpsize to 0, thereby disabling core dumps.

When the faulty program of the previous section is run, it will crash and create a core dump:

```
$ bug
5
Memory fault - core dumped
$ ls -al core
-rw-r--r--   1 se       prac       63276 Aug 13 10:42 core
$
```

When the core file is passed as second parameter to GDB, GDB immediately locates the offending statement:

```
$ gdb bug core
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.15.1 (sparc-sun-solaris2.4),
Copyright 1995 Free Software Foundation, Inc...
Core was generated by ‘bug’.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /usr/lib/libc.so.1...done.
Reading symbols from /usr/lib/libdl.so.1...done.
#0  0x106b4 in printptr (iptr=0x0) at bug.c:8
8           printf("%d\n", *iptr);
(gdb)
```

3.7.3 Tracing code

An example in C When you want to know what exactly is happening in your program, you have to trace it with the debugger. To do this, use the following commands in GDB (the abbreviation of a command is given between parentheses):

- list (l): List a part of the program.
- step (s): Step the program until it reaches a different source line.
- next (n): Like the `step` command as long as function calls do not happen; when they do, the call is treated as one instruction, ie. the function is not entered but it is executed.
- break (b): Set a breakpoint at a specified line or function. A breakpoint is a point at which execution will be stopped when the program is running.
- print (p): Print the value of a variable.

With the program `bug`, a debugging session could look like this:

```
$ gdb bug
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.13 (sparc-sun-solaris2.3),
Copyright 1994 Free Software Foundation, Inc...
(gdb) break main
Breakpoint 1 at 0x106cc: file bug.c, line 14.
(gdb) run
Starting program: /home/se/doc/debug/source/bug

Breakpoint 1, main () at bug.c:14
14          int a = 5;
(gdb) print &a
$1 = (int *) 0xeffff944
(gdb) next
15          int* ptr = 0;
(gdb) next
17          printptr(&a);
```

```
(gdb) next
5
18          printptr(ptr);
(gdb) step
printptr (iptr=0x0) at bug.c:8
8           printf("%d\n", *iptr);
(gdb) print iptr
$2 = (int *) 0x0
(gdb) step

Program received signal SIGSEGV, Segmentation fault.
0x106b4 in printptr (iptr=0x0) at bug.c:8
8           printf("%d\n", *iptr);
(gdb) list
3       #include <stdio.h>
4
5
6       void printptr(int* iptr)
7       {
8           printf("%d\n", *iptr);
9       }
10
11
12      int main()
(gdb) quit
The program is running.  Quit anyway (and kill it)? (y or n) y
$
```

An example in C++ Suppose we've implemented a very simple linked class:

```
#include <iostream.h>


class node
{
public:
    node(int i, node* n) { item_ = i; next_ = n; }

    int item() { return item_; }
    node* next() { return next_; }

protected:
    int item_;
    node* next_;
};


int main()
{
    int i;
    node* current = 0;

// build list
```

```
    node* head = new node(1, NULL);
    head = new node(2, head);       // prepend new node
    head = new node(3, head);       // prepend new node

// print list
    current = head;
    for (i = 0; i <= 3; i++)        // since list-size is now 3
    {
        cout << "item " << i << " is " << current -> item() << endl;
        current = current -> next();
    }

// destroy list
    current = head;
    for (i = 0; i <= 3; i++)
    {
        current = head -> next();
        delete head;
        head = current;
    }

    return 0;
}
```

It looks quite simple, but it still contains a bug:

```
$ g++ -g -o list list.cc
$ list
item 0 is 3
item 1 is 2
item 2 is 1
Memory fault
$
```

First, we let GDB pinpoint the exact location of the memory fault:

```
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.13 (sparc-sun-solaris2.3),
Copyright 1994 Free Software Foundation, Inc...
$ gdb list
(gdb) run
Starting program: /home/se/doc/debug/source/list
item 0 is 3
item 1 is 2
item 2 is 1

Program received signal SIGSEGV, Segmentation fault.
node::item (this=0x0) at list.cc:9
9           int item() { return item_; }
(gdb)
```

If the information provided by GDB is examined closely, it appears that the `this`-pointer is 0! This means a member function is called for a non-existing object and this will indeed cause a memory fault. But how can this happen? Use the command `bt` to show the calling sequence that resulted in the memory fault:

```
(gdb) bt
#0  node::item (this=0x0) at list.cc:9
#1  0x800896e in main () at list.cc:32
(gdb) list list.cc:32
27
28      // print list
29          current = head;
30          for (i = 0; i <= 3; i++)        // since list-size is now 3
31          {
32              cout << "item " << i << " is " << current -> item() << endl;
33              current = current -> next();
34          }
35
36      // destroy list
(gdb)
```

It appears something goes wrong while printing all values. Since `current` is always pointing to the current object, it follows that `current` is at some point set to 0. But `current` can only be 0 at the end of the list (when `next_` equals 0). Let's see what happens when the last item has been printed:

```
(gdb) br list.cc:32
Breakpoint 1 at 0x8008940: file list.cc, line 32.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/se/doc/debug/source/list

Breakpoint 1, main () at list.cc:32
32              cout << "item " << i << " is " << current -> item() << endl;
(gdb) cont
Continuing.
item 0 is 3

Breakpoint 1, main () at list.cc:32
32              cout << "item " << i << " is " << current -> item() << endl;
(gdb) cont
Continuing.
item 1 is 2

Breakpoint 1, main () at list.cc:32
32              cout << "item " << i << " is " << current -> item() << endl;
(gdb)
```

So far, so good. This is the last item to be printed (since there are only 3 list-nodes). We now use tracing to see what happens next:

```
(gdb) n
item 2 is 1
33              current = current -> next();
(gdb) n
34          }
(gdb) print current
$1 = (node *) 0x0
(gdb)
```

As expected, `current` is now 0. But what happens now?

```
(gdb) n
32              cout << "item " << i << " is " << current -> item() << endl;
(gdb)
```

It appears the loop is not finished yet! How can that be?

```
(gdb) print i
$2 = 3
(gdb)
```

After some thinking, the problem is clear: the loop is executed four times instead of three times. The for-loop should be `for (i = 0; i < 3; i++) { ... }` and then it will work (the second loop should of course be changed also).

4 More information

- `man gdb`
- `man gcc`
- `man xxgdb`, xxgdb is the X-version of gdb, with a nice interface, mouse input etc.
- `/usr/local/doc/dvi/gdb.dvi` and `/usr/local/doc/ps/gdb-refcard.ps`
- `man assert`
- `man limit`, `man unlimit`

References

[1] D.L. Parnas. *Why Software Jewels Are Rare.* IEEE Computer, February 1996, pp. 57-60