ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Short Introduction to C++ & STL

Florian Jug
Christoph Krautz
Thomas Rast

September 20, 2010

Department of Computer Science, ETH Zürich

Contents

Chapter 1

Introduction

1.1 What you will find here

- how to compile, run, debug
- how to estimate the runtime of your algorithm
- pointer, memory management
- I/O (streams), strings
- STL concepts
  1. templated functions/container
  2. container
  3. iterators
  4. STL functions/working with iterators

1.2 What you will not find here

- Control Structures (if, switch, for, while, ...)
- Data Types (int, float, double, char, ...)
- Object Oriented Programming in C++

1.3 Literature

Will be available in the computer room.

- C++ Tutorial: http://www.cplusplus.com/doc/tutorial/
- C++ Reference: http://www.cppreference.com/wiki/start

- STL Reference: http://www.sgi.com/tech/stl/table_of_contents.html
- Bjarne Stroustrup, The C++ Programming Language
- Meyers, More Effective C++
- Meyers, Effective STL
- Sedgewick, Algorithmen in C++
- Sedgewick, Algorithms in C++ (Part 5)
- Cormen, Introductions to Algorithms

Chapter 2

The Basics

2.1 The Beginning: C++'s Adam & Eve

How will your problem be judged?

You'll get a problem description and the description of the input format. The input will be given on `stdin`, which means you can just read it using `cin` or `getline(..)`. (See below.) Further you'll get a description of the output format you will have to write on `stdout`, using for example `cout`.

The judging system will take your source code, will compile it for you and run it giving input to test your solution. The output of your program will be collected and compared to the correct output. The outcome of your trials can be seen (almost) immediately.

Example: Hello world!

```
// needed, e.g., for cin & cout
#include <iostream>

using namespace std;

int main (void) {
  cout << "Hi!\n";  // \n is faster than endl

  // you must return 0, anything else is an error
  return 0;
}
```

Imagine you have to solve the following exercise: you get n testcases. The first line of the input contains n. For each testcase you'll get first a line containing m, the number of lines following. Each of the m lines following contains a double-value. Your program should add all these numbers and write the result in a single line.

Example: reading input, writing output

```
#include <iostream>

using namespace std;

int main (void) {
  int cases;
  cin >> cases;             // read n

  for (int c=0; c<cases; c++) {
    int m;
    cin >> m;               // read m

    double sum=0, summand;
    for (int i=0; i<m; i++) {
      cin >> summand;
      sum += summand;
    }

    cout << sum << endl;    // write result on stdout
  }
}
```

After you've tested you program locally you'll have to submit it to the Judge (www.elabs.inf.ethz.ch). The judge will give you one of the following feedbacks:

correct You win!

presentation-error There are (only) whitespace differences between the reference output and your output. Check if you are correctly formatting the output according to the problem specification.

wrong-answer Your program gave a wrong answer. If you are lost at this point, design your own testcases. It may also help to let other students give you their solutions for your own testcases.

assertion-failure Your program terminated with a SIGABRT. This indicates an assertion failure, which among other things can be caused by wrong

use of library routines or memory (malloc arena) corruption. Use GDB for "real" assertions and valgrind to track down memory corruption.

segmentation-fault Your program terminated with a SIGSEGV. This indicates it attempted to access memory that it is not allowed to, and is the result of a programming error, such as an out-of-bounds array access or dereferencing a NULL pointer. Use valgrind.

run-error Your program exited with nonzero exit status. Most probably your `main()` function returns a nonzero value. (This status also includes all signal exits not covered by the two preceding items.)

timelimit Your program has exceeded the allowed running time. Check for inefficient use of data structures, such as inserting at the front of a vector or taking the length of a list. Failing that, see if you can use a faster algorithm.

forbidden Your program tried to use a system call not on the whitelist; or it tried to break out of the valgrind environment.

Note that the Judge will (almost always) test your program with more than one input file. As a consequence your program might pass the easy testset correctly, but fail a harder (bigger) one with e.g. "wrong-answer" or "time-limit". The immediate feedback you'll get from the judge will just show you the worst of all feedbacks. If you click on it you'll get further details about the results of the individual runs.

2.2 Data structures: STL, Arrays and Pointers

See http://www.sgi.com/tech/stl/table_of_contents.html for lots of information about the STL.

The examples below assume that you import all symbols from the `std` namespace, like so:

```
#include <iostream>
#include <vector>
// etc.

// after all includes
using namespace std;
```

This lets you omit the `std::` prefix, which gets rather noisy if you use the STL a lot.

2.2.1 Simple Template: Pair

- Idea: reusable code for arbitrary data types.
- Example 1: use the STL container `pair` for any kind of content.

```
pair<int,string> my_pair1;
// set the parts...
my_pair1.first = 1;
my_pair1.second = "foo";
// or set all at once using 'make_pair'...
my_pair1 = make_pair (1, "foo");
```

- Example 2: you can use more than just standard data types

```
pair<double,pair<int,string> > my_pair2;
// set the parts...
my_pair2.first = 1.23;
my_pair2.second = my_pair1;
// or set all at once using 'make_pair'...
my_pair2 = make_pair (1.23, my_pair1);
```

  Hints:

  - Typical mistake: you *must* spell it `pair<int, pair<int,int> >`. '...>>' parses as the entirely different shift operator. Don't miss the space!
  - You'll have to include `utility` in order to use `pair`.

2.2.2 Arrays and Vectors

As for arrays: Don't use them. If you use them, don't be surprised.

STL vectors are array-like data structures with automatic management. First include the corresponding header:

```
#include <vector>
```

There are several useful constructors:

```
vector<int> a;          // a starts out empty
vector<int> b(10);      // b starts out with 10 elements
vector<int> c(10, 42); // c starts out with 10 elements
                        // each set to 42
```

Element access works as you would expect, numbered from zero and *without any bounds checking*. There is also the highly useful `at()` method which does check bounds.

```
b[5] = 1;       // element number 5 set to 1
a[0] = 1;       // this will most likely segfault
b[10] = 1;      // this will most likely NOT segfault
                // and cause trouble later!
b.at(10) = 1;   // this will terminate with SIGABRT
```

Vectors also trivially support non-constant size, resizing, etc.:

```
b.clear();              // b is now empty
a.push_back(5);         // appends 5 to a
```

See the STL documentation for more information.

2.2.3 Other useful containers

- Maps are an associative array (like a hash table, but the STL maps are not implemented as hashes).

  ```
  map<string, int> m;
  m["foo"] = 10;
  cout << m["foo"];
  ```

  HINT: You'll have to include `map`.

- Sets are just that: a collection of items, each of which only occurs once. There are plenty of methods to compute differences, intersections etc.

  ```
  set<int> set_a, set_b, result;
  set_a.insert(4711); ...
  set_difference(set_a.begin(), set_a.end(),
                         set_b.begin(), set_b.end(),
                         inserter(result, result.begin()));
  set_a = result;
  result.clear();
  set_union(set_a.begin(), set_a.end(),
            set_b.begin(), set_b.end(),
            inserter(result, result.begin()));
  ```

  HINT: You'll have to include `set`.

Other useful containers too look at:

- `list`: doubly-linked list. Rarely useful, but you can insert/remove any item in constant time provided you already have an iterator pointing at it.
- `multiset`, `multimap`: versions of set and map that allow a key to occur more than once.

- `stack`: efficiently implements a LIFO. Most people just use a vector and `push_back()/pop_back()`.
- `queue`: efficiently implements a FIFO, useful e.g. for breadth-first search. (Lists could be used for the same asymptotic performance, but have a noticeable memory management overhead.)

2.2.4 Iterators

- They are like pointers to iterate over the elements of a container.
- Many times you don't see them, but you use them (like in Example 3).
- `begin`: iterator to first element
- `end`: iterator pointing past last element
- Example 1: some random iterator-code

```
vector<int> v, w;
...
for (vector<int>::iterator it(v.begin());
     it != v.end(); it++) {
  cout << *it << endl;
}
...
sort (v.begin(), v.end());
copy (v.begin(), v.end(), w.begin());
```

- Example 2: reversing a string

```
string str("abcdefg");
reverse(str.begin(), str.end());
// str is now: "gfedcba";
```

- Example 3: set all elements in a vector to 5

```
vector<int> a(200);
fill(a.begin(), a.end(), 5);
```

2.2.5 Algorithms

STL provides many reusable algorithms in the `algorithm` header. The most useful example is sorting. To sort a vector:

```
vector<string> vec;
// fill vector with strings...
sort(vec.begin(), vec.end());
```

This works with any data type that is ordered, i.e., for which the < operator works.

There are several ways to sort custom data structures. The simplest one is to define the < operator appropriately. E.g., if the 'bar' field of your class holds the sort key:

```
class Foo {
public:
    operator< (const Foo& other) const {
        return bar < other.bar;
    }
private:
    double bar;
}

vector<Foo> vec;
// ... put stuff in there ...
sort(vec.begin(), vec.end());
```

An order is also required if you want to use your class as the key in a `map`.

2.2.6 Pointers and references

Don't use pointers unless you have good reason to, such as building custom data structures.

To pass variables by reference instead of value, use – surprise – references, declared with an ampersand `&`:

```
void f(int x, int &y) {
    y += x;
}

// ...
int s = 0;
f(4, s);   // s is now 4
f(2, s);   // s is now 6
```

2.2.7 Passing by const reference

Vectors and other large data structures that are constant across the invocation of a function should usually be passed as a const reference:

```
// this function doesn't modify the out-edge lists
void depth_first_search(int start,
        const vector<vector<int> > &outedges) {
    // ...
}
```

This avoids the large overhead associated with copying, especially when the data structures are nested.

2.3 Memory Management

Don't. Use STL containers instead.

If you do, use the C++ style. This guarantees that your objects are properly constructed and destructed.

- You can allocate memory with `new`:

  ```
  vector<int> *vecptr = new vector<int>();
  int *intarray = new int[15];
  ```

- You can free memory with `delete`:

  ```
  delete vecptr;
  delete[] intarray;
  ```

  Note that it is very important that you use the `[]` form to delete arrays.

2.4 I/O Streams

- Read integers till end of the input and output the sum:

  ```
  int sum = 0;
  int n;
  while(cin >> n) {
    sum += n;
  }
  cout  << "the␣sum:␣" << sum << endl;
  ```

- Read one line:

  ```
  string line;
  getline (cin, line);
  if (line.empty()) { ... } // empty line check
  ```

- Read input line by line:

```
string line;
while (getline(cin, line)) {
  ...
}
```

- Write something without/with fixed precision:

- Write something without/with fixed precision:

```
double a,b,c;
a = 3.1415926534;
b = 2006.0;
c = 1.0e-10;
cout.precision(5);
cout       <<         a << '\t' << b << '\t' << c << endl;
cout <<   fixed    << a << '\t' << b << '\t' << c << endl;
cout << scientific << a << '\t' << b << '\t' << c << endl;
```

The snippet above creates the following output:

```
3.1416          2006            1e-010
3.14159         2006.00000      0.00000
3.14159e+000    2.00600e+003    1.00000e-010
```

HINT: You can gain a lot of I/O performance if you promise to never use the C stdio.h facilities:

```
cin.sync_with_stdio(false);
cout.sync_with_stdio(false);
```

If you break this promise (e.g., by using `printf`) the result of your I/O operations is undefined.

2.5 Strings

HINT: You'll have to include `string`.

Examples

- Example 1: read, write, concatenate

```
string name;
cin >> name;
cout << "Hello␣" + name + "!";
```

Note that it reads a single word, not a line.

- Example 2: create an initialized string

```
string s(10, '␣'); // a string of ten spaces
```

- Example 3: reverse a string

```
reverse(s.begin(), s.end());
```

Hint:

Look at http://www.sgi.com/tech/stl/basic_string.html to find some information about strings in C++.

Chapter 3

Compiling, Running, Debugging

All tools necessary to solve the exercises for the Algorithms Lab are installed on the student lab computers. We suggest to use `Kate` or `emacs` to edit your sourcecode. Kate offers a shell window and makes it easy to compile and run your program from within Kate. It also comes with syntax highlighting. If you are not yet decided give it a try!

If you want to use your own machines you'll have to install the GNU compiler collection. (Linux and Mac users should already have it on their machines, Windows users could install Cygwin or comparable tools. Also working remote on the student-lab machines is a possibility.)

Note For the second and third part of the Algorithms Lab you'll have to use the Boost Graph Library and CGAL. In order to use your own computer you'll have to install also these libraries later on.

IMPORTANT Also if you use your own computer you should get used to the setup on the student lab machines. You will have to write the exam on them!

3.1 How to Compile

We're using `g++` to compile your C/C++ code. In the folder you've put your source type the following command in order to compile it:

```
g++ -Wall -O3 -o <BINARY> <SOURCE>
```

g++ This is the traditional nickname of `GNU C++`, a freely redistributable C++ compiler. It is part of `GCC`, the GNU compiler collection.

-Wall Turns on all optional warnings which are desirable for normal code.

-O3 Turns on many compiler optimizations. Your program will run faster! (Note: 'O' is not a zero but the capital character 'O' like in "Optimization"!)

-o <**BINARY**> Place output in file `<BINARY>`.
(Something like "`funnyexercise`" or "`funnyexercise.exe`" on Windows systems.)

<**SOURCE**> Indicates the file containing the source.
(Something like "`funnyexercise.cpp`".)

3.2 How to Run

Once you've compiled your code you should be able to run it. Since your programs - as explained in section 2.1 - reads the inputs from `stdin` and writes all outputs on `stdout` the program will wait for inputs after starting it.

You can use your keyboard to type some test input, but usually it is less time consuming to prepare the input in a textfile and pipe it directly into your program when starting it. In the command-line (terminal/ console) it looks like this:

```
./funnyexercise <inputfile.in
```

The content of `inputfile.in` will be sent to `stdin` after starting the binary `funnyexercise`. The output will be printed on your screen though. In order to print the output to another file use the following command:

```
./funnyexercise <inputfile.in >funnyoutput.out
```

3.3 Common Mistakes, And How to Debug Them

In the following, we will discuss some common mistakes, how they become apparent, and how you can use debugging tools to track them down.

We do not discuss the debugging tools beyond what is required for these examples. Refer to tutorials on the Internet, e.g.,

`http://www.unknownroad.com/rtfm/gdbtut/gdbtoc.html`
`http://cs.ecs.baylor.edu/~donahoo/tools/valgrind/`

As a first introductory remark, you should *always* compile your program *without optimization* (`-O0`) and with debugging (`-g`). So modify the compilation command above as follows:

```
g++ -Wall -O0 -g -o <BINARY> <SOURCE>
```

You can find full, compilable, broken and fixed sources for each of the sections below in the course materials for week 1.

3.3.1 Use Assertions

With `#include <cassert>` you can state implicit assumptions in your code as *assertions*. (You are not supposed to use this for *input checking*, but since judge inputs are supposed to be error-free, you can usually disregard the latter.)

You simply state a condition that should always hold in the `assert()` macro, which is used like a function. For example, to read the number of sub-cases in a test, you might run:

```
int k;
cin >> k;
assert(k > 0);
```

If your program ever gets confused about its position in the input, and reads a negative number, you will notice immediately instead of later in the algorithms:

```
$ ./01_assert < 01_assert_sample.in
01_assert: 01_assert.cpp:16: int main(): Assertion
   k > 0'
   failed.Aborted
```

Since this is a termination by signal, `gdb` will stop straight at the failed assertion:

```
$ gdb ./01_assert
[...]
(gdb) r < 01_assert_sample.in
[...]
Program received signal SIGABRT, Aborted.
0x00000030018329e5 in raise () from /lib64/libc.so
   .6
```

You can now issue debugger commands to inspect the stopped state. A good first idea is to look at the `backtrace` (`bt`) to see the series of function calls that led here:

```
(gdb) bt
#0  0x00000030018329e5 in raise () from /lib64/libc
   .so.6
```

```
#1  0x0000003001833ee6 in abort () from /lib64/libc
   .so.6
#2  0x0000003001 82b235 in __assert_fail () from /
   lib64/libc.so.6
#3  0x0000000000400bc3 in main () at 01_assert.cpp
   :16
```

In this case, `raise()`, `abort()`, and `__assert_fail()` are just the way that `assert()` is implemented internally, so we go up three stack frames to our `main()`:

```
(gdb) up 3
#3  0x0000000000400bc3 in main () at 01_assert.cpp
   :16
16              assert(k > 0);
```

You can now use various commands to see what happens in this stack frame. To see the source around the current line, run `list` (`l` for short):

```
(gdb) l
11
12          assert(cases > 0);
13          while (--cases) {
14              int k;
15              cin >> k;
16              assert(k > 0);
17              vector<int> a;
18              while (--k)
19                  a.push_back(k);
20          }
```

To print the contents of a variable, run `print` (`p`):

```
(gdb) p k
$1 = -1
(gdb) p cases
$2 = 1
```

Note that the `print` command is quite complex and powerful. Consult `help print` for more information, especially if the debugger cannot guess the data type of your variables correctly.

3.3.2 Counterintuitive Comparisons

Two very common mistakes in C/C++ are exhibited in the following snippet:

```
if (-2 <= x <= 2)
    cout << "close␣to␣0" << endl;
else if (x = 42)
    cout << "the␣answer!" << endl;
```

They are actually quite different:

- The first comparison is bogus because you *cannot chain* these operators, i.e., the condition is equivalent to `((-2 <= x)<= 2)`. For hysterical raisins, `(-2 <= x)` evaluates to either 0 (`false`) or 1 (`true`) in a numerical context, so the outer comparison `<= 2` is always true.
- The second "comparison" is always true because *it is actually an assignment*. The value of an assignment is the assigned value, i.e., 42 in this case. The correct way to spell "is equal to" is ==.

Both of these are caught by the compiler if you remember to use `-Wall` (as you should!):

```
$ g++ -O0 -g -Wall 02_chain_comparison.cpp -o 02
   _chain_comparison
02_chain_comparison.cpp: In function 'int main()':
02_chain_comparison.cpp:10:17: warning: comparisons
    like 'X<=Y<=Z' do not have their mathematical
   meaning
02_chain_comparison.cpp:12:17: warning: suggest
   parentheses around assignment used as truth
   value
```

No need to use a debugger, yay!

3.3.3 switch needs break

To wit:

```
switch (x) {
case 1:
    cout << "one" << endl;
case 2:
    cout << "two" << endl;
case 3:
    cout << "three" << endl;
default:
    cout << "I␣can␣only␣count␣to␣three!" << endl;
}
```

Running this code with `x = 1` will actually print *all four* of the strings.

Unlike the `case` statements of some other languages, a C/C++ `switch()` needs to explicitly `break` out when you do not want to continue with the next case. It is quite common to add a comment like

```
/* fall through */
```

when the fall-through behavior is intended.

This is a logic error, so it is hard to give generic debugging advice, and the compiler cannot issue warnings. Train your eyes.

3.3.4 Uninitialized Variables

Very common. Suppose you are writing a program that reads a series of `int`s from `stdin`, and prints the largest one.

```
int max;
int x;
while (cin >> x) {
    if (x > max)
        max = x;
}
cout << max;
return 0;
```

Most of the time this will work fine. Sometimes you may see strange behavior, like a seemingly random number being printed.

This is one of the cases where `valgrind` really shines. You use it as a wrapper around your program, and it will run your program (at a huge speed penalty, but still) while checking it for various memory issues.

```
$ echo 1 2 3 | valgrind ./04_uninitialized
[...]
==29901== Conditional jump or move depends on
   uninitialised value(s)
==29901==    at 0x400834: main (04_uninitialized.
   cpp:11)
==29901==
3==29901==
[...]
==29901== ERROR SUMMARY: 1 errors from 1 contexts (
   suppressed: 5 from 5)
```

Note how the error message is annotated with a backtrace (in this case of only a single frame).

Use `valgrind` *a lot*. In fact always run your program through `valgrind` at least once before submitting.

3.3.5 Array Overruns

This most frequently bites programmers who try to be clever and static arrays:

```
int a[5];
int i = 0;
while (cin >> a[i])
    i++;
```

This code invariably starts screwing up in random ways as soon as you give it more than 10 numbers. Try, for example:

```
$ seq 1 10 > seq_10
$ gdb ./05_array_overrun
[...]
(gdb) r < seq_10
[...]
Program received signal SIGSEGV, Segmentation fault
   .
0x000000300000000a in ?? ()
(gdb) bt
#0  0x000000300000000a in ?? ()
#1  0x0000000000000000 in ?? ()
```

Notice how the overrun smashed the stack, making the backtrace useless. (Which is still better than making it *misleading*!)

Heap-allocated arrays are more fortunate because at least valgrind can detect them. Turning the above into a straightforward heap-allocated version, we get:

```
$ valgrind ./05_array_overrun_dynamic  < seq_10
[...]
==30470== Invalid write of size 4
==30470==    at 0x300607DE54: std::istream::
   operator>>(int&) (istream.tcc:186)
==30470==    by 0x400980: main (05
   _array_overrun_dynamic.cpp:9)
```

```
==30470==  Address 0x4c41054 is 0 bytes after a
   block of size 20 alloc'd
==30470==    at 0x4A069B9: operator new[](unsigned
   long) (vg_replace_malloc.c:305)
==30470==    by 0x400955: main (05
   _array_overrun_dynamic.cpp:7)
[...]
```

3.3.6 Invalid pointers

These come in various flavors. A fairly simple one would be:

```
vector<int>& read_input()
{
    vector<int> a;
    // ...
    return a;
}
```

In fact the compiler can even warn about such a trivial case:

```
$ g++ -O0 -g -Wall 07_lost_pointer_2.cpp -o 07
   _lost_pointer_2
07_lost_pointer_2.cpp: In function 'std::vector<int
   >& read_input()':
07_lost_pointer_2.cpp:8:17: warning: reference to
   local variable 'a' returned
```

This is a one-character bug: remove the `&`, and you avoid the problem at the cost of a copy. (Rewrite with an argument `vector<int>& a` to avoid the copy.)

A more involved but unfortunately common case with the STL is exhibited in this snippet:

```
vector<int> a;
int x;
int max = numeric_limits<int>::min();
int *max_p;
int i = 0;

while (cin >> x) {
    a.push_back(x);
    // now a[i] == x
    if (x > max) {
```

```
            max_p = &(a[i]);
            max = x;
        }
        i++;
    }
```

The problem here is that pointers and iterators into vectors may be invalidated *any time you add or remove objects* to/from the vector. `valgrind` helps:

```
$ echo 5 1 | valgrind ./06_lost_pointer
[...]
==31162== Invalid read of size 4
==31162==    at 0x400DF0: main (06_lost_pointer.cpp
   :25)
==31162==  Address 0x4c41040 is 0 bytes inside a
   block of size 4 free'd
==31162==    at 0x4A05E4F: operator delete(void*) (
   vg_replace_malloc.c:387)
==31162==    by 0x40172F: __gnu_cxx::new_allocator<
   int>::deallocate(int*, unsigned long) (
   new_allocator.h:95)
==31162==    by 0x401489: std::_Vector_base<int,
   std::allocator<int> >::_M_deallocate(int*,
   unsigned long) (stl_vector.h:146)
==31162==    by 0x401319: std::vector<int, std::
   allocator<int> >::_M_insert_aux(__gnu_cxx::
   __normal_iterator<int*, std::vector<int, std::
   allocator<int> > >, int const&) (vector.tcc:361)
==31162==    by 0x400FBF: std::vector<int, std::
   allocator<int> >::push_back(int const&) (
   stl_vector.h:749)
==31162==    by 0x400D8C: main (06_lost_pointer.cpp
   :16)
==31162==
```

Note the stacktrace of the reallocation inside `std::vector`.

3.3.7 Stack Overflow

Due to the way the OS manages your program's stack, a stack overflow shows up as a segmentation fault. Consider:

```
int factorial(int n)
{
```

```
    return n * factorial(n-1);
}

// ...

    factorial(n)
```

This code fairly obviously goes into an infinite recursion. GDB can debug this, but takes a few seconds to analyze the stack and generate a backtrace:

```
$ gdb ./08_stack_overflow
[...]
(gdb) r
1

[...]
Program received signal SIGSEGV, Segmentation fault
   .
0x000000000040090c in factorial (n=Cannot access
   memory at address 0x7fffff7feffc
) at 08_stack_overflow.cpp:6
6       {
(gdb) bt
#0  0x000000000040090c in factorial (n=Cannot
   access memory at address 0x7fffff7feffc
) at 08_stack_overflow.cpp:6
#1  0x000000000040091c in factorial (n=-261944) at
   08_stack_overflow.cpp:7
#2  0x000000000040091c in factorial (n=-261943) at
   08_stack_overflow.cpp:7
#3  0x000000000040091c in factorial (n=-261942) at
   08_stack_overflow.cpp:7
[...]
#60 0x000000000040091c in factorial (n=-261885) at
   08_stack_overflow.cpp:7
---Type <return> to continue, or q <return> to quit
   ---q
Quit
```

In general it is useless to continue through all 261947 frames 60 at a time, so you can use the optional argument to `bt` to look at only the last few:

```
(gdb) bt -10
#261938 0x000000000040091c in factorial (n=-7) at
   08_stack_overflow.cpp:7
```

```
#261939 0x000000000040091c in factorial (n=-6) at
   08_stack_overflow.cpp:7
#261940 0x000000000040091c in factorial (n=-5) at
   08_stack_overflow.cpp:7
#261941 0x000000000040091c in factorial (n=-4) at
   08_stack_overflow.cpp:7
#261942 0x000000000040091c in factorial (n=-3) at
   08_stack_overflow.cpp:7
#261943 0x000000000040091c in factorial (n=-2) at
   08_stack_overflow.cpp:7
#261944 0x000000000040091c in factorial (n=-1) at
   08_stack_overflow.cpp:7
#261945 0x000000000040091c in factorial (n=0) at 08
   _stack_overflow.cpp:7
#261946 0x000000000040091c in factorial (n=1) at 08
   _stack_overflow.cpp:7
#261947 0x0000000000400936 in main () at 08
   _stack_overflow.cpp:15
```

And then use the `frame` command to directly jump to one of them:

```
(gdb) frame 261945
#261945 0x000000000040091c in factorial (n=0) at 08
   _stack_overflow.cpp:7
7           return n * factorial(n-1);
```

In fact if the case were not so clear cut, you might also want to use a conditional breakpoint to stop exactly at the function invocation where `n=0`:

```
(gdb) break factorial if n==0
Breakpoint 1 at 0x40090f: file 08_stack_overflow.
   cpp, line 7.
(gdb) r
[...]
1

Breakpoint 1, factorial (n=0) at 08_stack_overflow.
   cpp:7
7           return n * factorial(n-1);
```

(Note the use of ==!)

You can then inspect the state in this recursive invocation as usual, and step through the execution using `next` (`n`) or `step` (`s`).

Chapter 4

How to Estimate the Runtime of Your Algorithm

4.1 The Clue

Of course we are usually searching for the fastest solution. But how fast should that be?

Every exercise can be solved in many different ways and all of them will usually have a different runtime. So how should you know what we are looking for?

If you click on the "Submit solution" link for the specific exercise you'll find the timelimit for this exercise. This is the maximal CPU-time your program is allowed to run on our Judge in order to come up with the correct solution for a single testset.

4.2 How to Use the Clue

There is a very simple rule of thumb: between 10^6 and 10^7 loops can be performed in 1 second. This is of course not always true and grossly oversimplified. It is surprisingly often a relatively good estimate though.

If you know that the solution you would like to implement has a complexity of $\mathcal{O}(n^2)$, the exercise sheet tells you that you'll get n's up to 50000 and the timelimit for the solution is set to 3 seconds you might reconsider your choice and implement the more complicated $\mathcal{O}(n \log n)$ algorithm.