Programming Language Specific Help
----------------- ----------------- -------

  c++
    How to write C++ solutions.

gdb
    How to debug infinite loops and crashes in
    C and C++ programs.
C++ Language Program Structure

--- -------- ------- ---------

Typical program structure including end of file detection, finding symbols such as ‘*’ that end a string of numbers, and debugging is:

```
#include <iostream>
#include <cstdio>    // printf
using std;

// String and character computation.
#include <cstring>   // strcmp, strcpy, ...
#include <cctype>    // isalpha, islower, ...

// Math and Number Limits.
#include <cmath>     // M_PI, atan2, ...
#include <climits>   // INT_MIN, INT_MAX, ...
#include <cfloat>    // DBL_MIN, DBL_MAX

// Debugging.
#include <cassert>   // assert

bool debug = false;
#define dout if ( debug ) cout
#define dprintf if ( debug ) printf

char line[81];
// You can assume lines longer than 80 characters will not need to be read, UNLESS the problem statement says otherwise. You need 1 extra character for string ending ‘\0’.
```

```
int main ( int argc, char * argv[] )
{
    // argc == number of program arguments + 1.
    debug = ( argc > 1 );
    . . . . .
    while ( cin.getline ( line, sizeof ( line ) ),
            cin.good() )
    {
        // Print test case name. */
        cout << line << endl;
        /* Read input. */
        cin >> ... . . . . .
        // Read a string of numbers ending in ‘*’. //
        double d;
        while ( cin >> d, cin.good() )
        {
            . . . . .
        }
        cin.clear();
        // After returning a false value for // cin.good(), we MUST cin.clear() // before reading more.
        . . . . .
        // Read ‘*’.
        . . . . .
        char c;
        cin >> c;
        assert ( c == ‘*’ );
        . . . . .
    }
}
```
// After reading test case data with cin,
// input will be just before the line feed
// at the end of the test case data. This
// must be skipped or else it will be
// erroneously read as an empty next test
// case name.

cin.getline ( line, sizeof ( line ) );
    // The line read here consists of just a
    // line feed and is discarded.
assert ( line[0] == 0 );

// Compute.

dout << ... << endl;  // Print if debug.
dprintf ( ... );      // Print if debug.
. . . .

// Output.

cout << ... << endl;
printf ( ... );
}

// Return 0 from `main' to tell system there
// is no error.
//
return 0;

Occasionally one has a conflict with some name defined
in the `#include's. E.g., you may have trouble naming a
global variable `time'. To fix this, change the name
of your variable. A simple way of doing this is to put
the following right after the `#include's:

#define time Time

Input
-----

Input is read from the standard input `cin'; you MUST
NOT open any file for input.

You may assume that input is correctly formatted, except
for the rare problem where you are told to produce
special output if the input contains a formatting error.
Do not waste time checking for input errors when you do
not have to.

Test cases begin with a test case name line that is
read by the first `getline'. When any input method call
fails, `cin.good()' becomes false, so the `while' loop
test consists of `cin.getline ( ... ), cin.good()' and
is true if `cin.getline ( ... )' succeeded and false if
`cin.getline ( ... )' failed because cin is at an end of
file. So the outer loop above processes one test case
at a time. As soon as the test case name is successfully
read into 'line', it is printed by 'cout << line <<
endl'. All test case name lines are 80 characters or
less, so 'line' has enough room for 80 characters plus
the string ending '\0' supplied by 'cin.getline'.

After reading and printing the test case name line, the
remaining test case data is read with 'cin >>'. This
data may consist of numbers, perhaps with something
like ‘*’ thrown in to indicate the end of a sequence of
numbers. Lines containing numbers may be very long, so
they should be read one number at a time with 'cin >>',
and NOT read in first with 'cin.getline'. A ‘*’ may be
detected by looking at ‘cin.good()’, which will be true
if 'cin >>' successfully read a number, and false if it
did not because the next non-whitespace character in the
input was, say, a ‘*’. In this last case, you must
first clear the failed state bits in cin using
‘cin.clear()’, and then skip the ‘*’ using ‘cin >> c’.
See the ‘summer’ demonstration problem solution which reads numbers.

Sometimes a problem will specifically ask you to read lines to be processed as character strings, in which case use ‘cin.getline’ and be sure your line buffer is long enough.

For some problems there are no test case name lines but instead each test case has one input line containing text and one output line containing text. For these the test case input line is read the same way as the test case name line is read above.

See the ‘reverser’ demonstration problem solution which reads text lines.

Sometimes it is necessary to read whitespace characters explicitly. In these cases ‘int c = cin.get()’ may be used to get the next character, where c will be set to the special value EOF if cin is at the end of file. ‘cin.unget()’ may be used to back up over a character in the input, but after reading a character from a line, you should not attempt to back up into a preceding line. See the ‘vcalc’ demonstration problem solution for a use of these functions.

Output
------

Output is written to the standard output ‘cout’; you MUST NOT open any file for output; you MUST NOT write to the standard error output, ‘cerr’.

In general, to be correct your program must produce EXACTLY the one and only correct sequence of output characters. The main exception is that when floating point numbers are output with a given number of decimal places, you are permitted to output numbers which differ from other correct output by one unit in the last decimal place. You must use the correct upper or lower case and use only a single space character as a separator unless instructed to line things up in columns.

Output can be written with ‘cout << ... << endl’. Output that has special formatting requirements may be written with ‘printf’. Some ‘printf’ formats that may be useful are "%.3f" to print a double with exactly 3 decimal places in as few columns as possible, "%10.3f" to print the double right adjusted in 10 columns, "%10s" to print a string right adjusted in 10 columns, and "%-10s" to print a string LEFT adjusted in 10 columns. Here the numbers 3 and 10 are merely representative, and can be replaced by any other non-zero positive integers.

Debugging
---------

When your program is executed by the judge, it will NOT be passed ANY arguments. A standard debugging technique is to output extra debugging information if and only if your program is passed an argument.

The above program defines ‘debug’ to be true if and only if the program is called with one or more arguments, and defines dout (or dprintf) to do what cout (or printf) does if and only if ‘debug’ is true. Thus dout and dprintf can be used to print debugging information.
Debugging is best done with information printed by `dout' and `dprintf', and not with a debugger like `gdb'. The exception is debugging programs that crash or go into an infinite loop, which is best done with a debugger like `gdb'. See `help gdb' for how to use `gdb' on programs that crash or go into an infinite loop.

It is also a good idea to use `assert' statements to check that assumptions you have made are valid during actual program execution. For example, if you are certain that `cin >> c' should read a `*', then follow this with

```cpp
assert ( c == '*' );
```

See the ‘reverser’ and ‘summer’ demonstration problem solutions for uses of dout, dprintf, and assert.

---

You can get full documentation of the functions mentioned above plus other useful information by using the UNIX commands:

```bash
man 3 printf
man assert
man math.h
man limits.h
man float.h
```

and looking for web documentation for

```cpp
istream::operator>>
istream::getline
istream::good
istream::clear
ostream::operator<<
ostream::endl
```

In ACM contests in which you are not permitted access to the web, C++ documentation is usually available on-line during the contest via the command:

```bash
c++help
```

The following is an extract from ‘man’ and web pages of details needed to use the functions mentioned above, and also a few numeric constants. Here we assume you already know something about how to use ‘cin’, ‘cout’, and ‘printf’, so we do not have to explain them from scratch.

Note that an ‘int’ is guaranteed to be at least 32 bits and a ‘long long int’ is guaranteed to be at least 64 bits. Floating point computations should always be done using ‘double’s, and NOT ‘float’s, to avoid having too little precision.
int i; long long li; double d; char name[1000], char c;
cin >> i >> li >> d >> name >> c

```cpp
int c = cin.get()
cin.unget()
```

Cin.get() returns the next input character, even if that is whitespace or '\n'. If there is no character because cin is at the end of file, the value EOF is returned (this is typically defined in the iostream header file as the integer -1). To backup one character in the input, so the next character read will be the character backed up over, use cin.unget(). The amount of backup allowed is unspecified, but to be cautious, after reading a character of a line, one should not try to backup into the preceding line.

```cpp
int i; long long li; double d; char name[1000]; char c;
cout << i << li << d << name << "..." << c
```

cout << v outputs v. Strings are '\0' terminated and the '\0' is not output. The output of numbers does not output any superfluous characters, and may output scientific notation for floating point numbers. If you want to format output more precisely, use printf.

```cpp
cout << endl
```

Output an end of line.

```cpp
int printf ( const char * format, ... )
```

Format can contain the following directives, in which W and P stand for sequences of decimal digits, where W is used for 'widths' and P for 'precisions':

```cpp
%d       Outputs an 'int' as a decimal integer with no spaces before or after.
%Wd      Outputs an 'int' as a decimal integer right adjusted in W columns.
```
%lld Outputs a ‘long long int’ as a decimal integer with no spaces before or after.

%Wlld Outputs a ‘long long int’ as a decimal integer right adjusted in W columns.

%f Outputs a ‘double’ as a floating point number with 6 decimal places and no spaces before or after.

%.Pf Ditto but with P decimal places instead of 6 decimal places.

%Wf Outputs a ‘double’ as a floating point number with 6 decimal places right adjusted in W columns.

%W.Pf Ditto but with P decimal places instead of 6 decimal places.

%s Outputs a ‘const char *’ or ‘char *’ string with no spaces before or after.

%Ws Ditto but outputs the string RIGHT adjusted in W columns.

%-Ws Ditto but outputs the string LEFT adjusted in W columns.

%c Outputs a ‘char’ as an ASCII character with no spaces before or after.

M_PI, M_E

The mathematical constants PI and e from ‘cmath’.

Writing Your Own << And >> Operators

It is often convenient to define your own << operator to output something. Some examples are:

```cpp
ostream & operator <<
    ( ostream & out, const mytype & value )
{
    . . .
    out << . . .
    . . .
    return out;
}
```

Suppose you want to output integers in a distinctive format. A simple way is

```cpp
struct myformat {
    int value;
    myformat ( int value ) : value ( value ) {};
};
ostream & operator <<
    ( ostream & out, const myformat & f )
{
    out << ... << f.value << ...;
    . . .
    return out;
}
```

... cout << myformat ( 99 );
Here we have invented a typed structure to encapsulate the value when it is to be printed. Note the argument to << may NOT be 'myformat & s' without the 'const', because in use the 'myformat' value is a temporary and temporaries are read-only.

Input operators can be define by:

```cpp
    istream & operator >>
        ( istream & in, mytype & value )
    {
        . . .
        in >> . . .
        . . .
        return in;
    }
```

The authors have placed this file in the public domain; they make no warranty and accept no liability for this file.
Debugging with GDB

Introduction

It is recommended that a debugger like GDB not be used unless your program crashes or goes into an infinite loop. In all other cases it is better to put print statements (such as dprintf or dout - see ‘help c’ or ‘help c++’) into your code to print just exactly what you need to see.

So in this help file we assume you are debugging a C or C++ program that crashes or loops infinitely.

Running the Program

Assume the program you are debugging is named PPPP and its input file is PPPP.in. Then

```
% gdb PPPP
(gdb) run <PPPP.in
```

If the program is going to crash, it will do so. Otherwise if the program loops infinitely, wait until it is so looping, and type control-C to stop the program.

Examining a Crashed or Stopped Program

Once stopped execute

```
(gdb) back
```

to see the backtrace of functions that have been called. Each function call is referred to as a ‘frame’. The frames are numbered, 0, 1, 2, ..., with 0 being the frame that was executing at the time of the crash or control-C, and the highest number frame being the ‘main()’ function.

You can see what the current instruction and variables of frame N are by executing

```
(gdb) frame N
(gdb) p VARIABLE
```

Here ‘p’ stands for ‘print’ and variables that can be printed in a frame are the local variables of the frame plus any global variables visible to the code that executes the frame.

When you execute ‘frame N’ the line at which frame N stopped executing will be printed, along with its line number. You can use an editor to look at the source code, or you can use the command

```
(gdb) list M,N
```

to list lines, where M,N are the first, last line numbers to be listed.
If your program is in an infinite loop, it may be helpful to continue the program. The following may be useful:

Continue: (gdb) c

Go to next statement in current frame: (gdb) n

Go to next statement in current frame or frame called by current frame: (gdb) s

Set a breakpoint at line N: (gdb) b N

Continue to next breakpoint: (gdb) c

List breakpoints: (gdb) i b

Delete breakpoint NUMBER K: (gdb) d K

Output documentation for command C: (gdb) h C

Find commands concerning word W: (gdb) apropos W