```
---------------------------------------------------------------
varscope.cpp: variable scope and name hiding (shadowing)
---------------------------------------------------------------

#include <...>
using namespace std;

static int number = 1;

struct X {
  X() { }
  static int number;
};

int X::number = 4;

void print(const char *txt, int number) {
  static int k = 0;
  cout << setw(10) << left << setfill('.') << txt
       << setw(3) << right << setfill(' ') << number
       << " " << k++ << "\n";
}

int main() {
  int number = 2;

  print("Global", ::number);
  print("Local", number);

  if (1) {
    int number = 3;
    print("Nested", number);
  }

  print("Struct1", X::number);

  X obj[3];
  for (int i=0; i<3; i++)
    print("Struct2", obj[i].number);

  obj[0].number = -4;
  for (int i=0; i<3; i++)
    print("Struct3", obj[i].number);
}
```

```
---------------------------------------------------------------
Hint: Global variables are defined outside all functions. They
can be accessed by all code.

Hint: Static global variables are only visible to code declared
in the same file.

Hint: Local variables are associated with a { }-block. The latter
can be implicit (for-loops above) or explicit (if-conditional).
Any such variables shadow same named variables outside the block.

Hint: Static local variables are remembered between calls (k).

Hint: Class member variables are unique to the objects.

Hint: Static class member variables are accesible by all objects
instantiated from that class (X::number).

Hint: Memory is divided into four segments called text (code),
data/bss (initialized/uninitialized global and static variables),
heap (dynamically allocated data) and stack (local variables).
See http://en.wikipedia.org/wiki/Data_segment for more details.

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

unix> varscope

Global....  1 0
Local.....  2 1
Nested....  3 2
Struct1...  4 3
Struct2...  4 4
Struct2...  4 5
Struct2...  4 6
Struct3... -4 7
Struct3... -4 8
Struct3... -4 9
```

```
----------------------------------------------------------------
trycatch.cpp: try-catch exception handling
----------------------------------------------------------------

#include <...>
using namespace std;

#define ENEGNUM 10
#define EBIGNUM 11

#define STR(MACRONAME) #MACRONAME

struct error {
  error() { }
  char errmsg[80];
};

struct error_negnum : error {
  error_negnum() {
    sprintf(errmsg, "%d %s", ENEGNUM, STR(ENEGNUM));
  }
};

struct error_bignum : error {
  error_bignum() {
    sprintf(errmsg, "%d %s", EBIGNUM, STR(EBIGNUM));
  }
};

void process_number(int number) {
  if (number < 0) {
    throw error_negnum();
  } else
  if (100000 < number) {
    throw error_bignum();
  }

  cout << "good: do something\n";
}

----------------------------------------------------------------
Hint: The #-preprocessor command expands a macro name into a
string literal that be printed like any other character string.

Hint: The syntax class A : B { ... } means class A is derived
from base class B. Member functions and variables are inherited
(not so for the constructor and destructor). Inhertitance allows
sharing of code and thus functionaility between classes. More
about this important concept later in the semester.

----------------------------------------------------------------
Hint: Derived classes look their base class. Either exception
thrown matches the base class reference below. Different catch
handling could be implemented by catching on the specific throw
object.

Hint: The catch-all handler is defined by a ... argument. This
matches any exception thrown that hasn't been caught.

Hint: When a catch handler cannot be found, std::terminate() is
called followed by std::abort(). The former can be redefined.
----------------------------------------------------------------

int main() {
  try {
    int number;
    while (cin >> number) {
      process_number(number);
    }
    throw 0; // for illustrational purposes only
  }

  catch (error & e) {
    cerr << "error: " << e.errmsg << "\n";
  }

  catch (...) {
    throw 0; // for illustrational purposes only
  }
}

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

unix> ./trycatch
302
good: do something

123456
error: 11 EBIGNUM

unix> ./trycatch
302
good: do something
CTRL-D

libc++abi.dylib: terminating with uncaught exception of type int
Abort

----------------------------------------------------------------
```

stack.cpp: namespace, application try-catch exception handling

```
#include <...>
using namespace std;

namespace Really_Long_Name_Stack_v1R3 {
  struct stack_overflow { };
  struct stack_underflow { };
}

namespace Really_Long_Name_Stack_v1R3 {
  template <typename T>
  class stack {
    public:
      stack();
      ~stack();

      bool empty() const { return N == 0; }
      int  size()  const { return N; }
      int  max_size()  const { return Nmax; }

      void resize();

      void push(const T &);
      void pop();
      const T & top();

    private:
      int N;
      int Nmax;
      T *v;
  };

  template <typename T>
  stack<T>::stack() {
    N = 0;
    Nmax = 32;

    v = new T [Nmax];
    for (int i=0; i<Nmax; i++)
      v[i] = T();
  }

  template <typename T>
  stack<T>::~stack() {
    delete [] v;
  }

  template <typename T>
  void stack<T>::resize() {
    T *tmp = new T [2*Nmax];

    for (int i=0; i<Nmax; i++)
      tmp[i] = v[i];

    for (int i=Nmax; i<2*Nmax; i++)
      tmp[i] = T();

    Nmax *= 2;

    delete [] v;
    v = tmp;
  }

  template <typename T>
  void stack<T>::push(const T &din) {
    if (N == Nmax)
      throw stack_overflow();

    v[N++] = din;
  }

  template <typename T>
  void stack<T>::pop() {
    if (N == 0)
      throw stack_underflow();

    v[--N] = T();
  }

  template <typename T>
  const T & stack<T>::top() {
    if (N == 0)
      throw stack_underflow();

    return v[N-1];
  }
}

namespace Stack = Really_Long_Name_Stack_v1R3;
```

Hint: The above namespace defines a simple stack and associated exception handling objects. The above namespace redefinition is an alias that simplies making references to the members.

```
int main(int argc, char *argv[]) {
  bool verbose = false;
  if (argc == 2 && strcmp(argv[1], "-verbose") == 0)
    verbose = true;

  Stack::stack<char> v;

  while (char c = cin.get()) {
    if (cin.eof())
      break;

    try {
      v.push(c);
    }

    catch (Stack::stack_overflow) {
      if (verbose)
        cerr << "stack overflow (" << v.size() << ")\n";

      cin.putback(c);
      v.resize();
    }
  }

  while (!v.empty()) {
    try {
      cout << v.top();
      v.pop();
    }

    catch (Stack::stack_underflow) {
      if (verbose)
        cerr << "stack underflow\n";
    }
  }
}
```

Hint: The above application (stack driver) code handles overflow exceptions by resizing the stack. Error messages are only printed to stderr if requested by a command line argument.

Hint: By executing the top-pop loop using a stack-empty check, underflow cannot occur. The catch handler will never be exeuted.

```
unix> echo "CS" | od -c
0000000    C   S  \n
0000003

unix> echo "CS" | ./stack | od -c
0000000   \n   S   C
0000003

unix> cat message.txt | wc -c
     257

unix> cat message.txt | ./stack -verbose

stack overflow (32)
stack overflow (64)
stack overflow (128)
stack overflow (256)

yelraM boB--

thgirla eb annog si
 gniht elttil yreve esuaC'
gniht a tuoba yrrow t'noD

'uoy ot egassem ym si sihT' gniyaS
eurt dna erup seidolem fO
sgnos teews gnigniS
petsrood ym yb hctiP
sdrib elttil eerhT
nus gnisir eht htiw delimS
gninrom siht pu esiR
```

Hint: The message is so long that it requires the stack buffer to be resized four times. You would not know that it happened without the request for verbose output.

Hint: Implement the program and run it on the above output to see the original message.
