DEBUGGING YOUR PROGRAM

- The debugging phase is the hardest of them all if the program is complex enough and/or if it manipulates pointers.
- Serves two purposes:
  - Testing before release (aka submission).
  - Maintenance.
- Debugging techniques include:
  - Verbose output
  - Using interactive debuggers
  - Code inspection
  - The confessional

SERIAL DEBUGGING

- Before you start debugging, save the “working” program in a safe place.
  - During the debugging process, you may need to add debug code, or change your code in order to try and eliminate the problems.
  - If you find yourself barking up the wrong tree, you could painlessly start all over again.
  - It is also easy to remove the debugging code once debugging is complete.
- Even so, do identify clearly any changes you make in the program.

VERBOSE OUTPUT

- Insert `cout` statements to see what's going on.
  1. Isolate the problem
     - Put `cout` statements to see where data turns bad and/or the program ceases to work.
       * They will print critical data or simply messages that show you where you are in the program.
     - Keep it up until you locate the bug as tightly as you can.
     - Something like a binary search.
  2. Solve the problem
     - Keep around `cout`'s that print critical data when you modify the buggy code.

MAINTENANCE VERBOSE OUTPUT

- When a program is maintained, it is wise to keep the debugging code (i.e., those `cout`'s) for the entire life of the program.
- Since this code is not needed all the time, surround it by `#ifdef`/`#endif` statements, e.g.,

```
#ifdef DEBUG
  cout << "+++ rmth: end of list, nothing to delete\n";
#endif
```

- Normally, `DEBUG` will not be defined, so the debugging code is not used.
- When you need to do debugging, define `DEBUG`, either directly in the files containing your code, or by using the `-D` switch of `g++`, e.g.,

```
g++ -g -Wall -DDEBUG -o foo foo.cc
```

- You can also create appropriate targets into the makefile, so that `make` will construct the normal program while, say, `make debug` will construct the verbose variant.

COMMAND-LINE SWITCHES FOR VERBOSE OUTPUT

- The use of `#ifdef` has the disadvantage of requiring recompilation each time debugging is desired.
- A smart alternative is to replace it with command-line switches.
- Normal way to obtain the command line arguments:

```
#include <iostream>
#include <unistd.h>
using namespace std;



int main (int argc, char** argv) {









  cout << "------ remaining args: ------\n";
  for (int i = 1; i < argc; i++) {
    cout << "argv[" << i << "] = " << argv[i] << "\n";  }    }
```

COMMAND-LINE SWITCHES FOR VERBOSE OUTPUT

- The use of `#ifdef` has the disadvantage of requiring recompilation each time debugging is desired.
- A smart alternative is to replace it with command-line switches.
- Obtain command line arguments by identifying switches:

```
#include <iostream>
#include <unistd.h>
using namespace std;

extern char *optarg;
extern int optind;

int main (int argc, char** argv) {
  int c;
  cout << "--------- options: ---------\n";
  while ((c = getopt (argc,argv,"abcd:")) != -1) {
    cout << "opt: " << (char)c << "\n";
    if (optarg) cout << "-> arg: " << optarg << "\n";
  }
  argc -= optind - 1; argv += optind - 1;
  cout << "------ remaining args: ------\n";
  for (int i = 1; i < argc; i++) {
    cout << "argv[" << i << "] = " << argv[i] << "\n";  }    }
```

COMMAND-LINE SWITCHES FOR VERBOSE OUTPUT (CONT'D)

```
int verbose[4] = {0,0,0,0};
const int vcar = 0;   const int vcdr = 1;
const int vcons = 2;  const int vrmth = 3;

int main (int argc, char** argv) {
  int c;
  while ((c = getopt (argc,argv,"v::")) != -1) {
    if (optarg == NULL)
      verbose[vrmth] = verbose[vcons] =
      verbose[vcdr] = verbose[vcar] = 1;
    else {
      if (strcmp(optarg,"car") == 0)
        verbose[vcar] = 1;
      if (strcmp(optarg,"cdr") == 0)
        verbose[vcdr] = 1;
      if (strcmp(optarg,"cons") == 0)
        verbose[vcons] = 1;
      if (strcmp(optarg,"rmth") == 0)
        verbose[vrmth] = 1;
    }
  }
... // stuff with car, cdr, cons, etc.
}
```

```
int car (list cons) {
  if (verbose[vcar])
    cout << "### car: " << cons
         << " -> car = "
         << cons -> car  << "\n";
  return cons -> car;
}
```

INTERACTIVE DEBUGGERS

- Using a debugger is incredibly time consuming, and I do not recommend it; you can get off track very easily.
- But if you have to have it, all Linux systems come with a powerful debugger called `gdb`.
- Call gdb as `gdb <program_name>`. Then useful gdb commands include

  run start execution of a program
  break n places a breakpoint at line n
  break f places a breakpoint at the beginning of function f.
  delete i removes breakpoint number i
  cont continues execution till the next breakpoint
  print e computes e and prints the result
  step executes a single line; steps into function calls
  next executes a single line; skips function calls
  list lists program
  where prints the function call chain
  info breakpoints prints breakpoint information

OTHER DEBUGGING METHODS

- Believe it or not, the following do work, most often than not.

  Code inspection. Get a coffee, a listing of your program, and a red pen; start to read your program and do not hesitate to mark it heavily.

  – Works especially after you isolated the bug to a relatively small piece of code.

  The confessional

  > "Hey Pete, here is my program, it drives me crazy because it keeps segment faulting on me somewhere here. And I have nothing else in the damn piece of code than a lousy `printf` which takes this string and this integer and. . . sheesh, here it is, I forgot to erase this %s here, yay."