Scoping and Testing

Prof. David August

COS 217

Overview of Today's Lecture

- Scoping of variables
  - Local or automatic variables
  - Global or external variables
  - Where variables are visible
- Testing of programs
  - Identifying boundary conditions
  - Debugging the code and retesting

Global Variables

- Functions can use global variables defined outside and above them

```
int stack[100];

int main(void) {
    . . .          <-- stack is in scope
}

int sp;

void push(int x) {
    . . .          <-- stack, sp are in scope
}
```


Definition vs. Declaration

- Definition
  - Where a variable is created and assigned storage
- Declaration
  - Where the nature of a variable is stated, but no storage allocated
- Global variables
  - Defined once (e.g., "**`int stack[100]`**")
  - Declared where needed (e.g., "**`extern int stack[]`**")
    - Only needed if the function does not appear after the definition
    - Convention is to define global variables at the start of the file

Local Variables and Parameters

- Functions can define local variables
  - Created upon entry to the function
  - Destroyed upon departure and value not retained across calls
    - Exception: “static” storage class (see chapter 4 of K&R)
- Function parameters behave like initialized local variables
  - Values copied into “local variables”
  - C is pass by value (so must use pointers to do “pass by reference”)

Local Variables & Parameters

- Function parameters and local definitions “hide” outer-level definitions (gcc -Wshadow)

```
int x, y;
. . .
void f(int x, int a) {
    int b;
    . . .
    y = x + a * b;
    if (. . .) {
        int a;
        . . .
        y = x + a * b;
    }
}
```

different x

same y

different a

Local Variables & Parameters

- Cannot declare the same variable twice in one scope

```
void f(int x) {
    int x;        error!
    . . .
}
```


Scope Example

```
int a, b;

int main (void) {
    a = 1; b = 2;
    f(a);
    print(a, b);
    return 0;
}

void f(int a) {
    a = 3;
    {
        int b = 4;
        print(a, b);
    }
    print(a, b);
    b = 5;
}
```

Scope: Another Example

interface.h
```
extern int A;

void f(int C);
```

module1.c
```
#include "interface.h"

int A;
int B;

void f(int C) {
   int D;
   if (...) {
      int E;
      ...
   }
}

void g(...) {
    int H;
   ...
}
```

module2.c
```
#include "interface.h"


int J;

void m(...) {
   int K;
   ...
}

void g(...) {
    int H;
   ...
}
```


Scope: A

interface.h
```
extern int A;

void f(int C);
```

module1.c
```
#include "interface.h"

int A;
int B;

void f(int C) {
   int D;
   if (...) {
      int E;
      ...
   }
}

void g(...) {
    int H;
   ...
}
```

module2.c
```
#include "interface.h"


int J;

void m(...) {
   int K;
   ...
}

void g(...) {
    int H;
   ...
}
```


Scope: B

interface.h
```
extern int A;

void f(int C);
```

module1.c
```
#include "interface.h"

int A;
int B;

void f(int C) {
   int D;
   if (...) {
      int E;
      ...
   }
}

void g(...) {
    int H;
   ...
}
```

module2.c
```
#include "interface.h"


int J;

void m(...) {
   int K;
   ...
}

void g(...) {
    int H;
   ...
}
```


Scope: C

interface.h
```
extern int A;

void f(int C);
```

module1.c
```
#include "interface.h"

int A;
int B;

void f(int C) {
   int D;
   if (...) {
      int E;
      ...
   }
}

void g(...) {
    int H;
   ...
}
```

module2.c
```
#include "interface.h"


int J;

void m(...) {
   int K;
   ...
}

void g(...) {
    int H;
   ...
}
```

Scope: D

interface.h
```
extern int A;

void f(int C);
```

module1.c
```
#include "interface.h"

int A;
int B;

void f(int C) {
   int D;
   if (...) {
      int E;
      ...
   }
}

void g(...) {
   int H;
   ...
}
```

module2.c
```
#include "interface.h"


int J;

void m(...) {
   int K;
   ...
}

void g(...) {
   int H;
   ...
}
```


Scope: E

interface.h
```
extern int A;

void f(int C);
```

module1.c
```
#include "interface.h"

int A;
int B;

void f(int C) {
   int D;
   if (...) {
      int E;
      ...
   }
}

void g(...) {
   int H;
   ...
}
```

module2.c
```
#include "interface.h"


int J;

void m(...) {
   int K;
   ...
}

void g(...) {
   int H;
   ...
}
```


Scope: Keeping it Simple

- Avoid duplicate variable names
  - Don't give a global and a local variable the same name
  - But, duplicating local variables across different functions is okay
    - E.g., array index of **i** in many functions
- Avoid narrow scopes
  - Avoid defining scope within just a portion of a function
    - Even though this reduces the storage demands somewhat
- Use narrow scopes judiciously
  - Avoid re-defining same/close names in narrow scopes
- Define global variables at the start of the file
  - Makes them visible to all functions in the file
  - Though, avoiding global variables whenever possible is useful

Scope and Programming Style

- Avoid using same names for different purposes
  - Use different naming conventions for globals and locals
  - Avoid changing function arguments
  - But, duplicating local variables across different functions is okay
    - E.g., array index of **i** in many functions
- Define global variables at the start of the file
  - Makes them visible to all functions in the file
- Use function parameters rather than global variables
  - Avoids misunderstood dependencies
  - Enables well-documented module interfaces
- Declare variables in smallest scope possible
  - Allows other programmers to find declarations more easily
  - Minimizes dependencies between different sections of code

Testing

Chapter 6 of “The Practice of Programming”

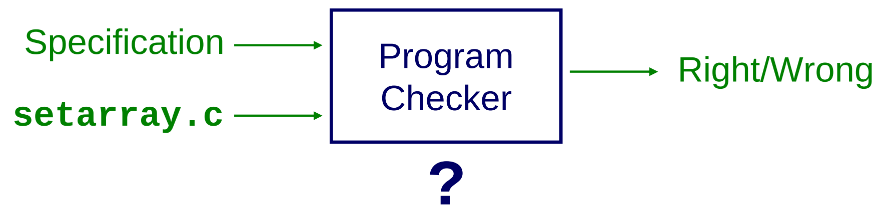
"On two occasions I have been asked [by members of Parliament!], `Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?' I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question."
-- Charles Babbage

Testing, Profiling, & Instrumentation

- How do you know if your program is correct?
  - Will it ever crash?
  - Does it ever produce the wrong answer?
  - How: testing, testing, testing, testing, …
- How do you know what your program is doing?
  - How fast is your program?
  - Why is it slow for one input but not for another?
  - How much memory is it using?
  - How: timing, profiling, and instrumentation (later in the course)

Program Verification

- How do you know if your program is correct?
  - Can you **prove** that it is correct?
  - Can you **prove** properties of the code?
    - e.g., It terminates



"Beware of bugs in the above code;
I have only proved it correct, not tried it." -- Donald Knuth

Program Testing

- Convince yourself that your program probably works

How do you write a test program?

Test Programs

- Properties of a good test program
  - Tests boundary conditions
  - Exercise as much code as possible
  - Produce output that is known to be right/wrong

How do you achieve all three properties?

Program Testing

- Testing boundary conditions
  - Almost all bugs occur at boundary conditions
  - If program works for boundary cases, it probably works for others
- Exercising as much code as possible
  - For simple programs, can enumerate all paths through code
  - Otherwise, sample paths through code with random input
  - Measure test coverage
- Checking whether output is right/wrong?
  - Match output expected by test programmer (for simple cases)
  - Match output of another implementation
  - Verify conservation properties
  - Note: real programs often have fuzzy specifications

Test Boundary Conditions

- Code to get line from stdin and put in character array

```
int i;
char s[MAXLINE];

for (i=0; (s[i]=getchar()) != '\n' && i < MAXLINE-1; i++)
    ;
s[--i] = '\0';
```

- Boundary conditions
  - Input starts with \n (empty line) ← what happens?
  - End of file before \n
  - End of file immediately (empty file)
  - Line exactly MAXLINE-1 characters long
  - Line exactly MAXLINE characters long
  - Line more than MAXLINE characters long

Test Boundary Condition

- Rewrite the code

```
int i;
char s[MAXLINE];
for (i=0; i<MAXLINE-1; i++)
    if ((s[i] = getchar()) == '\n')
        break;
s[i] = '\0';
```

- Another boundary condition: EOF

```
for (i=0; i<MAXLINE-1; i++)
    if ((s[i] = getchar()) == '\n' || s[i] == EOF)
        break;
s[i] = '\0';
```

This is wrong; why?

- What are other boundary conditions?
  - Nearly full
  - Exactly full
  - Over full

A Bit Better...

- Rewrite yet again

```
for (i=0; ; i++) {
    int c = getchar();
    if (c==EOF || c=='\n' || i==MAXLINE-1) {
      s[i]='\0';
      break;
    }
    else s[i] = c;
}
```

- There's still a problem...

Input:

```
Four
score and seven
years
```

Output:

```
FourØ
score anØ
 sevenØ
yearsØ
```

Where's the 'd'?

Ambiguity in Specification

- If line is too long, what should happen?
  - Keep first MAXLINE characters, discard the rest?
  - Keep first MAXLINE-1 characters + '\0' char, discard the rest?
  - Keep first MAXLINE-1 characters + '\0' char, save the rest for the next call to the input function?
- Probably, the specification didn't even say what to do if MAXLINE is exceeded
  - Probably the person specifying it would prefer that unlimited-length lines be handled without any special cases at all
  - Moral: testing has uncovered a design problem, maybe even a specification problem!
- Define what to do
  - Truncate long lines?
  - Save the rest of the text to be read as the next line?

Moral of This Little Story:

- Complicated, messy boundary cases are often symptomatic of bad design or bad specification
- Clean up the specification if you can
- If you can't fix the specification, then fix the code

Test As You Write Code

- Use “assert” generously (the time you save will be your own)
- Check pre- and post-conditions for each function
  - Boundary conditions
- Check invariants
- Check error returns

Test Automation

- Automation can provide better test coverage
- Test program
  - Client code to test modules
  - Scripts to test inputs and compare outputs
- Testing is an iterative process
  - Initial automated test program or scripts
  - Test simple parts first
  - Unit tests (i.e., individual modules) before system tests
  - Add tests as new cases created
- Regression test
  - Test all cases to compare the new version with the previous one
  - A bug fix often create new bugs in a large software system

Stress Tests

- Motivations
  - Use computer to generate inputs to test
  - High-volume tests often find bugs
- What to generate
  - Very long inputs
  - Random inputs (binary vs. ASCII)
  - Fault injection
- How much test
  - Exercise all data paths
  - Test all error conditions

Who Tests What

- Implementers
  - White-box testing
  - Pros: An implementer knows all data paths
  - Cons: influenced by how code is designed/written
- Quality Assurance (QA) engineers
  - Black-box testing
  - Pros: No knowledge about the implementation
  - Cons: Unlikely to test all data paths
- Customers
  - Field test
  - Pros: Unexpected ways of using the software, “debug” specs
  - Cons: Not enough cases; customers don’t like “participating” in this process; malicious users exploit the bugs

Conclusions

- Scoping
  - Knowing which variables are accessible where
  - C rules for determining scope vs. good programming practices
- Testing
  - Identifying boundary cases
  - Stress testing the code
  - Debugging the code, and the specification!