Style, debugging, and optimization

Lecture 10
CS 113 – Fall 2007

Advantages of C

- Very good for writing fast code
  - Usually much faster than Java, Matlab, etc.
- Simple syntax
- Portable and ubiquitous
  - C compilers available on almost any platform
- Very powerful; programmer is in control
  - C allows operations that other languages don't (e.g. direct access to memory)

Disadvantages of C

- Difficult to learn
  - Memory management is particularly frustrating
- Difficult to write bug-free, maintainable C code
  - Memory errors are difficult to detect and locate
  - Easy to inadvertently write insecure code
  - C does not enforce good software engineering principles
- Missing some modern language features
  - exceptions, packages, etc.

When to use C?

- For writing fast programs
- When efficiency really matters
  - computation-intensive applications
  - memory-intensive applications
- For systems programming
  - Operating systems, device drivers, embedded systems
- For writing portable code

When *not* to use C?

- For programs where efficiency isn't a concern
  - Many user applications
  - Prototyping new programs or new algorithms
  - Use Matlab, C#, Java, Visual Basic instead
- For large non-systems software projects
  - C++ is often a better choice
- For writing *really* portable code
  - e.g. web applications
- When security is the foremost goal

Bugs in C code

- Bugs in C programs usually cause a program crash


```
Segmentation fault (core dumped)
```

```
Bus error (core dumped)
```

- There are thousands of possible causes!
  - Dereference NULL pointer
  - Array bounds error
  - Forgot to open file
  - free() multiple times
  - Out of memory
  - Function didn't return a value
  - Stack overflow
  - Wrong version of library
  - Forgot to allocate memory
  - Invalid pointer cast
  - Forgot to 0-terminate a string
  - free() a stack variable

Example

```
#include <stdio.h>
#define SIZE 4

int add10(int n) {
  n + 10;
}

int main(int argc, char *argv[])
{
  int A[SIZE], j;
  FILE *fp;

  fp = fopen("C:\data\datafile.txt", "w");

  for(j=0; j<=SIZE; j++) {
    fscanf(fp, "%d", A[j]);
    A[j] = add10(A[j]);
  }

  return 0;
}
```


Tools for debugging

- The compiler itself
  - Turn on all compiler warnings (e.g. `-Wall` for gcc)
- Traditional debugger
  - Lets you step line-by-line through code, inspect variables, etc.
  - e.g. **`gdb`**
- Memory debugger
  - Attempts to find memory management bugs (memory leaks, array overruns, accessing memory that is not allocated, etc.)
  - e.g. Rational Purify (Windows), valgrind (Linux)

Compiler warnings

- Investigate and fix all compiler warnings

```
[crandall@lion ~/cs113 69]% g++ test1.c -Wall
test1.c: In function `int add10(int)':
test1.c:5: warning: statement has no effect
test1.c:6: warning: control reaches end of non-void function
test1.c:13:14: warning: unknown escape sequence '\d'
test1.c:13:14: warning: unknown escape sequence '\d'
```


Memory debuggers

- Work by *instrumenting* compiled C code
  - Adds extra instructions that check each memory access
  - Adds *memory guards* around memory regions
  - Maintains info about allocated/freed memory, etc.
- Instrumented code is *very* slow
  - runs 10-20x slower, uses more memory
- Can easily save hours of work

Memory debugger

```
[crandall@lion ~/cs113 80]% valgrind ./test1
==29770== Memcheck, a memory error detector.
==29770==
==29770== Use of uninitialised value of size 4
==29770==    at 0x8F73F7: _IO_vfscanf (in /lib/tls/libc-2.3.4.so)
==29770==    by 0x8F92E0: vfscanf (in /lib/tls/libc-2.3.4.so)
==29770==    by 0x8FD97E: fscanf (in /lib/tls/libc-2.3.4.so)
==29770==    by 0x80484F1: main (test1.c:16)
==29770==
==29770== Process terminating with default action of signal 11 (SIGSEGV)
==29770==  Bad permissions for mapped region at address 0xD70A08
==29770==    at 0x8F73F7: _IO_vfscanf (in /lib/tls/libc-2.3.4.so)
==29770==    by 0x8F92E0: vfscanf (in /lib/tls/libc-2.3.4.so)
==29770==    by 0x8FD97E: fscanf (in /lib/tls/libc-2.3.4.so)
==29770==    by 0x80484F1: main (test1.c:16)
==29770==
==29770== LEAK SUMMARY:
==29770==    definitely lost: 0 bytes in 0 blocks.
==29770==      possibly lost: 0 bytes in 0 blocks.
==29770==    still reachable: 352 bytes in 1 blocks.
==29770==         suppressed: 0 bytes in 0 blocks.
Segmentation fault
```


C programming style

- Good programming style will help prevent bugs
- C doesn't impose a programming style on you
  - This is both an advantage and a disadvantage
  - It's up to you to develop a style that promotes efficient, safe, maintainable software

Comment!

- Use comments
  - Explain how non-trivial code works
  - Very important: document functions
  - e.g. What does this function do?

```
char *copy_memory(char *m1, char *m2, int n);
```

- Avoid useless comments
  - `x = x + 1; //add one to x`

Be consistent

- Choose style conventions and stick with them
  - How are variables and functions named?, e.g.
    - local variables are one word in lower case, e.g. `counter`
    - function names are lower case with underscores, e.g. `get_name()`
    - constants are in upper case, e.g. `MAX_SIZE`
  - How are errors reported?, e.g.
    - Functions return 1 on success, 0 on failure
  - In what order are function parameters?, e.g.
    - Destination pointer first, then source pointer(s)
  - How is memory management handled?, e.g.
    - Any pointers returned by a function must be free()'d by the caller

Always check for errors

- There is not an exception mechanism in C
  - So you need explicit error checking
- Investing in error checking saves time later

```
FILE *fp;
char *buf

fp = fopen("C:\\data\\datafile.txt", "r");

// if file wasn't opened properly, print the error message
if(!fp) {
  perror("error");
}

buf = (char *) malloc( 25000000 );
assert(fp); // make sure memory was allocated successfully
```


Write clear code

- When choosing among multiple coding strategies, choose the one that is *simpler* and *clearer*
  - Even at the expense of speed or compactness (usually)
- Avoid C's more terse and confusing features
  - Use braces, parentheses and spaces for clarity
  - Avoid using `++` or `--` in complex expressions
  - If statements are usually clearer than conditionals

```
a=(b++>7?b:a>1?1:a)
```

```
if(b > 7)
  a = b;
else if(a > 1)
  a = 1;
else
  a = a + 3;
b++;
```


How not to write C code…

- C code can become incredibly complicated
  - Inspired the International Obfuscated C Contest (IOCC)

```
main(O){int I,Q,l=O;if(I=l*4){l=6;if(l>5)l+=Q-8?l-
  (Q=getchar()-2)%2:l;if(Q*=2)O+="has dirtiest IF"[(I/-
  Q&12)-l/Q%4];}return 8+O%4;}
```


How not to write C code…

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
#include <signal.h>
#define m(b)a=b;z=*a;while(*++a){y=*a;*a=z;z=y;}
#define h(u)G=u<<3;printf("\e[%uq",l[u])
#define c(n,s)case n:s;continue
char x[]="((((((((((((((((((((((",w[]=
"\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b";char r[]={92,124,47},l[]={2,3,1
,0};char*T[]={"  |","  |","%\\|/%"," %%%",""};char d=1,p=40,o=40,k=0,*a,y,z,g=
-1,G,X,**P=&T[4],f=0;unsigned int s=0;void u(int i){int n;printf(
"\233;%uH\233L%c\233;%uH%c\233;%uH%s\23322;%uH@\23323;%uH \n",*x-*w,r[d],*x+*w
,r[d],X,*P,p+=k,o);if(abs(p-x[21])>=w[21])exit(0);if(g!=G){struct itimerval t=
{0,0,0,0};g+=((g<G)<<1)-1;t.it_interval.tv_usec=t.it_value.tv_usec=72000/((g>>
3)+1);setitimer(0,&t,0);f&&printf("\e[10;%u]",g+24);}f&&putchar(7);s+=(9-w[21]
)*((g>>3)+1);o=p;m(x);m(w);(n=rand())&255||--*w||++*w;if(!(**P&&P++||n&7936)){
while(abs((X=rand()%76)-*x+2)-*w<6);++X;P=T;}(n=rand()&31)<3&&(d=n);!d&&--*x<=
*w&&(++*x,++d)||d==2&&++*x+*w>79&&(--*x,--d);signal(i,u);}void e(){signal(14,
SIG_IGN);printf("\e[0q\ecScore: %u\n",s);system("stty echo -cbreak");}int main
(int C,char**V){atexit(e);(C<2||*V[1]!=113)&&(f=(C=*(int*)getenv("TERM"))==(
int)0x756E696C||C==(int)0x6C696E75);srand(getpid());system("stty -echo cbreak"
);h(0);u(14);for(;;)switch(getchar()){case 113:return 0;case 91:case 98:c(44,k
=-1);case 32:case 110:c(46,k=0);case 93:case 109:c(47,k=1);c(49,h(0));c(50,h(1
));c(51,h(2));c(52,h(3));}}
```

Optimizing

- When speed really matters, C is the language to use
  - Can be 10-100x faster than Java or Matlab
- But writing something in C doesn't guarantee speed
  - It's up to you to write efficient code
- There are two general strategies for *optimization*
  - Use a better algorithm or different data structures
    - Always start here! CS 211, CS 482 discuss how to do this
  - Write code that implements the algorithm more efficiently

Optimizing code

- Turn on automatic compiler optimizations
  - Modern compilers include sophisticated algorithms that analyze your code and optimize it automatically
  - Often 5-10x speed up
  - e.g. `-O3` flag to gcc
- Reduce memory use
  - Remove unnecessary variables
  - Reuse memory buffers
  - Use the narrowest types possible (e.g. shorts instead of ints)

Optimizing code

- Reduce number of function calls
  - Change the function into a macro (with #define)
  - Change recursive algorithms into iterative algorithms
  - Put redundant computations outside loops

```
void compute(int A*, int N)
{
  int j;
  for(j=0; j<N; j++)
    A[j] *= sqrt(2);
}
```

```
void compute(int A*, int N)
{
  int j;
  double sqrt_2 = sqrt(2);
  for(j=0; j<N; j++)
    A[j] *= sqrt_2;
}
```


Optimizing code

- Loop unrolling
  - In each loop iteration, the stopping condition and increment statements must be executed
  - Optimize by reducing the number of times a loop executes
    - I.e. do more work per iteration

```
void compute(int A*, int N)
{
  int j;
  double sqrt_2 = sqrt(2);
  for(j=0; j<N; j++)
    A[j] *= sqrt_2;
}
```

```
void compute(int A*, int N)
{
  int j, N2 = N/2;
  double sqrt_2 = sqrt(2);
  for(j=0; j<N2; j++) {
    A[j] *= sqrt_2;
    A[j] *= sqrt_2;
  }
  for( j*=2 ; j < N; j++)
    A[j] *= sqrt_2;
}
```

Optimizing code

- Clever tricks
  - Use bitwise operations to avoid expensive instructions, e.g.

```
/* return the minimum of two integers */
int min(int x, int y) {
  return y + ((x - y) & -(x < y));
}
```

```
/* macro to swap the values in two variables */
#define SWAP(a, b) (((a) ^= (b)), ((b) ^= (a)), ((a) ^= (b)))
```

- Write assembly language code
  - You can include assembly code right in your C program!