Pseudorandom Numbers

• int rand(void)
  • Generate pseudorandom number between 0 and RAND_MAX inclusive
  • **This function is dangerous — avoid it!!** In older versions, it is *not* pseudorandom in the low order bits. (On newer Linux systems, it’s OK)

• long random(void)
  • Generate pseudorandom number between 0 and $2^{31}-1$ inclusive

• All require a starting point – called a *seed*
Pseudorandom Number Seeds

- void srand(unsigned int seed)
  - Initialize the rand() pseudorandom number generator with seed
- void srandom(unsigned int seed)
  - Initialize the random() pseudorandom number generator with seed
- Pick seed as randomly as possible
- There are defaults, useful for regenerating the same sequence for debugging
  - rand/srand default seed is 1
  - random/srandom default seed is 1
Random Numbers

• Linux has a pool of bits generated from sources such as hardware timings and other natural sources that are considered random
  • They are not generated by an algorithm as pseudorandom numbers are

\[
\text{getrandom}(\text{void } *\text{buf}, \text{size_t sz}, \text{unsigned int flags})
\]

• Generates sz random bytes and store them in the given \textit{buf}

• Returns number of bytes stored in \textit{buf}

• Flags:
  • \texttt{GRND\_NONBLOCK} prevents \texttt{getrandom()} from blocking; if it would block it returns –1 and sets \texttt{errno} to \texttt{EAGAIN}
  • \texttt{GRND\_RANDOM} draws from a random source more limited than the one used when this flag is given (avoid using this one)
Example Use

unsigned int rnd;
int count;
count = getrandom(&rnd, sizeof(unsigned int), GRND_NONBLOCK);
if (count == -1)
    perror("getrandom");
else
    for(i = 0; i < count; i++)
        printf("0x%02x\n", count, (rnd>>i)&0xff);
String Functions

• strcpy, strcat, strcmp, strncpy, strncmp, strlen
  • You’ve seen these

• char *strdup(char *s): make a duplicate of string s
  • Space is malloc’ed

• char *strchr(char *s, int c): return pointer to first occurrence of character c in s; NULL if not there

• char *strrchr(char *s, int c): like strchr, but points to last occurrence

• char *strstr(char *s, char *t): like strchr, but looks for first occurrence of string t
String Functions

- char *strtok(char *s, char *delim): breaks a string into a sequence of 0 or more nonempty tokens (substrings)
  - On first call, s points to string to be parsed
  - On subsequent calls for the same string, set s to NULL
  - delim is a string of characters that delimit tokens
  - strtok returns NULL when there are no more tokens to return
  - strtok \textit{always} returns a nonempty token
  - Warning: strtok overwrites delimiters with \textquoteleft\textbackslash 0\textquoteright, so don’t give it a read-only string
- int strcasecmp(char *a, char *b): useful for homework; look it up
Memory Functions

• void *memcpy(void *dest, void *src, unsigned int n): copy n bytes from src to dest
  • Behavior undefined if src, dest overlap
• int memcmp(void *s1, void *s2, unsigned int n): compare first n bytes of s1 and s2; returns negative, zero, positive depending on whether s1 is less than, equal to, greater than s2
Math Functions

• double floor(double d), double ceil(double d): round d down, up to the nearest integer
• double log(double d), double log10(double d): return the natural log, base 10 log of d
• double exp(double d), double pow(double m, double e): return $e^d, m^e$
• double sin(double d): compute sine of $d$ in radians
  • same with cos, tan
• double atan(double x): return principal value of arctan of $d$
  • In range $[-\pi/2, +\pi/2)$
• double atan2(double x, double y): return arctan of $y/x$
  • Handles cases where x is 0; returns value in range $[-\pi, \pi]$
Bug: Stack Smashing

• Problem: failure to check input length
• Going back to the stack, here is what it looks like when a function is called:
The Program bad.c

```c
#include <stdio.h>
char *gets(char *);
int main(void)
{
    int above = 100;
    char input[24];
    int below = 200;
    printf("BEFORE INPUT: above = %#010x; below = %#010x\n", above, below);
    if (gets(input) == NULL){
        fprintf(stderr, "Unexpected EOF\n");
        return(1);
    }
    printf(" AFTER INPUT: above = %#010x; below = %#010x\n", above, below);
    return(0);
}
```
A Program Run

- **BEFORE INPUT:** above = 0x00000064; below = 0x000000c8
- aaaaaaaaaaaaaaaaaaaaaaaaaaaa
- **AFTER INPUT:** above = 0x00000064; below = 0x00006161

26 a's (overflowing input by 2 chars)  'a' is represented by the number 0x61 in the computer
May Change Variable Values Unexpectedly

• Here is the stack frame after `gets` is called

```
above
input
below
return address
other control stuff
parameter
```

- Variables in `main()`
- Control data and parameters for `gets()`
Writing a Program with Random Numbers

• Monty Hall problem:
  • In a game, Monty asks a contestant to pick one of three doors. Behind one is a valuable prize; behind the other two are joke prizes (like a goat or a wheelbarrow full of mud).
  • The contestant picks a door.
  • Monty says, "Before I show you what is behind that door let me show you what is behind one of the doors you did not select". They pick such a door, it is opened, and behind it is a joke prize.
  • Monty asks if the contestant wants to switch to the other, unopened door.
  • The problem asks, should the contestant do so?

May 28, 2024

ECS 36A, Spring Quarter 2024
Programming Step 1

• First, we decide how to represent the doors
  • 3 doors, so call them 1, 2, and 3
• How do we determine which one has the good prize?
  • Let's pick one of the doors at random
• Which door does the contestant pick?
  • We can do this
    • We can have the computer select randomly among the 3 doors
• What happens if the contestant:
  • Switches?
  • Doesn't switch?
Programming Step 2

• First draft of program: human does everything:
  1. Human picks where the prize goes
  2. Human picks which door the contestant picks
  3. Human picks door to open (it cannot be the one with the prize)
  4. Human decides whether to switch
• This lets us create the framework for the program.
First Version – monty1.c

• Written as outlined above
• Oops ... There's a bug:
  Select door where prize is > 1
  Select door for contestant > 2
  I will show you door 3
  Does contestant switch doors? > y
  y or n please! > y
  You picked door 1, but the prize is behind door 1 -- you win!
• Let's use gdb to debug it
First Version – monty1.c

• Aha! We forgot to eat the rest of the line after scanf reads the entered number!
• We'll fix this in the next version
• Returning to the design . . .
First Version – monty1.c

• Looks pretty complicated – can we simplify it?
  • We go through an awful lot to figure out what door the contestant switches to, if they switch
  • Do we really need to do this?
First Version – monty1.c

• Looks pretty complicated – can we simplify it?
  • We go through an awful lot to figure out what door the contestant switches to, if they switch
  • Do we really need to do this?

• It doesn't matter what door they pick – what matters is whether they wind up picking the prize door
Second Version – monty2.c

• Try another approach using that observation

1. Human picks where the prize goes
2. Human picks which door the contestant picks
3. Human picks door to open (it cannot be the one with the prize)
4. If user decides to switch:
   a. If user picked prize door, they lose
   b. If user did not pick prize door, they win
Third Version – monty3.c

• Now we add randomness
• Wherever user asked for a number, generate a random one
  • So we change `ask_user()` to return a random number of 1, 2, or 3
• Consider whether it is necessary to show which door Monty shows
Third Version – monty3.c

• Now we add randomness

• Wherever user asked for a number, generate a random one
  • So we change ask_user() to return a random number of 1, 2, or 3

• Consider whether it is necessary to show which door Monty shows
  • It isn't; all we care about is whether the contestant switches their selection of doors
Fourth Version – monty4.c

• We delete the `monty_shows_door()` routine
• Next, do we need to ask user whether to switch?
Fourth Version – monty4.c

• We delete the `monty_shows_door()` routine
• Next, do we need to ask user whether to switch?
Fourth Version – monty4.c

• We delete the `monty_shows_door()` routine

• Next, do we need to ask user whether to switch?

• No; we can get the relevant result utilizing symmetry

• If the user does not switch:
  • If the first selected door is the same as the prize door, a win
  • If the first selected door is the not same as the prize door, a loss

• If the user switches:
  • If the first selected door is the same as the prize door, a loss
  • If the first selected door is the not same as the prize door, a win
Fifth Version – monty5.c

• We rewrite the main() routine to implement the above
• Now we can delete switch_or_not()
• Program is messy, though, so need to clean it up
Sixth Version – monty6.c

• We clean up some things and add a clearer statement of the output
• Now our program works – for 1 game. So we still cannot answer our question.
• To do so, we need to play a lot of games, counting how many win with switching and how many win without switching, and compare the numbers against the total number of games played
• To do this, we make a loop of what is in main()
• For now, we'll assume 10,000 games
Seventh Version – monty7.c

• Some more clean up
• First, delete print message for the random number routine and rename it appropriately
• Next, the two routines picking the prize and contestant door are 1 line once you remove the printf statements, so put the line into the main function
Seventh Version – monty7.c

• Some more clean up

• First, delete print message for the random number routine and rename it appropriately

• Next, the two routines picking the prize and contestant door are 1 line once you remove the printf statements, so put the line into the main function

• Do we always want 10,000 runs – maybe when you get to 100,000 or 1,000,000 games, the ratio between the switching and not switching becomes closer to 0.5 or something else?
Eighth Version – monty8.c

• Make the 10,000 games a macro and define it at the head of the file
• This way, we can change the number without searching the program for the number
Eighth Version – monty8.c

• Make the 10,000 games a macro and define it at the head of the file
• This way, we can change the number without searching the program for the number
• But to change the number we have to edit the source code and recompile it. We should allow the user to change it without doing this.
Ninth (and Final) Version – monty9.c

• We can either read the number as input or as a command-line argument
  • monty9.c implements the latter
  • The former is left as an exercise to the student 😊

• Whichever you choose, do not forget to check for errors!
  • If the argument is not present, use a default value
  • If there is more than 1 number given, report an error

May 28, 2024

ECS 36A, Spring Quarter 2024