Chapter Eight

Designing Libraries

1

Programming Complexity

- As long as we continue to solve problems of *ever-increasing sophistication*, the process of programming will become more *sophisticated* as well
- To make programming *manageable*, we must *reduce the complexity* of the programming process *as much as possible*
- *Libraries* offer a similar reduction in programming complexity as *functions*, but at a *higher level of detail*

2

Criteria for Designing Libraries

- Unified
- Simple
- Sufficient
- General
- Stable

3

A Unifying Theme

- A central feature of a well-designed library is that it presents a *unified and consistent abstraction and theme*
- The *math* library consists of *mathematical functions*, and the *graphics* library provides functions for *drawing pictures on the screen*
- All the functions in graphics use *coordinates* in *inches* and *angles* in *degrees*, and all drawings *begin* at the *current* position of the pen

4

Simplicity & Information Hiding

- A library should be *as easy to use as possible*
- A library interface acts as a *reference guide* and contains *precisely* the information a *client* needs to know *and no more*
- Often, the real value of an interface lies *not* in the information it *reveals* but rather in the information it *hides*

5

Simplicity & Information Hiding

- It is best to think of an interface not primarily as a *communication channel* between the client and implementation, but instead as a *wall* that divides them
- Ideally, all the complexity involved in the *realization* of a library lies on the *implementation side* of the wall

6

Meeting the needs of Clients

- Everything should be *as simple as possible*, *but no simpler*
- Simplicity is only part of the story. Your interface must provide *sufficient functionality* to serve the clients' needs
- Learning to strike the right balance between *simplicity* and *completeness* in interface design is one of the fundamental challenges in programming

7

Advantages of General Tools

- A good library abstraction serves the needs of *many* different clients
- When you design the interface for a library, *forget about the application* that cause you to design it in the first place and instead design it for the *most general possible audience*
- The function drawGrid in graphics library is a good example

8

The Value of Stability

- People *change* and *forget to tell each other*. Too bed—causes so many mistakes
- Interfaces tend to be *stable* over *long periods of time*
- As long as the interface *does not change*, both implementers and clients are *relatively free to make changes* on their own side of the abstraction boundary

9

The Value of Stability

- Changing an interface in such a way that *existing programs will continue to run without modification* is called *extending* the interface
- If you need to *make evolutionary changes* over the lifetime of an interface, it is usually best to *make those changes by extension*

10

Generating Random Numbers

- Being able to *simulate random behavior* is necessary in situations that involves flipping a coin or rolling a die
- A program is *deterministic* if its actions are *completely predictable* given any set of input values; otherwise, it is *nondeterministic*

11

Pseudo Random Numbers

- From a theoretical perspective, a number is *random* if there is no way to determine in advance what value it will have among a set of equally probable possibilities
- Computers in fact use a *deterministic* procedure to generate what we call *random* numbers

12

Pseudo Random Numbers

- In most applications, it does not matter if the numbers are *truly random*; all that matters is that the numbers *appear to be random*
- They should behave like random numbers from a *statistical* point of view
- They should be *sufficiently difficult* to predict in advance that no user would bother

13

rand

- The function rand in the library stdlib.h produces *pseudo* random numbers
  int rand(void)
- The values returned are integers between 0 and RAND_MAX, inclusive

14

An Example

```
#include <stdio.h>
#include <stdlib.h>

main()
{
   int  i;
   printf("On this computer, RAND_MAX = %d.\n",
      RAND_MAX);
   for (i = 1; i < 10; i++) {
      printf("The %dth number = %10d\n", i, rand());
   }
}
```

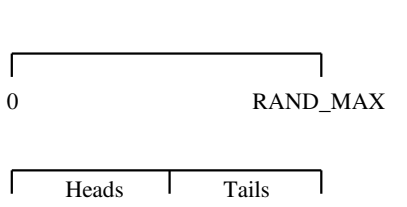
15

An Example

```
On this computer, RAND_MAX = 32767.
The 1th number =         41
The 2th number =      18467
The 3th number =       6334
The 4th number =      26500
The 5th number =      19169
The 6th number =      15724
The 7th number =      11478
The 8th number =      29358
The 9th number =      26962
Press any key to continue
```

16

Changing the Range



Heads
Tails
Heads
Tails
Tails
Heads
Heads
Tails
Tails

17

Changing the Range

```
#include <stdio.h>
#include <stdlib.h>
main() {
   int  i;
   for (i = 1; i < 10; i++) {
      if (rand( ) <= RAND_MAX / 2) {
         printf("Heads\n");
      } else {
         printf("Tails\n");
      }
   }
}
```

18

Pitfalls

- The *only random property* that you are allowed to count on when using rand is *the position of the result along the number line*
- When converting the result of rand to a more restricted range of integers, do *not* try to use the *remainder* operator
- On many computers, the results of rand alternate between even and odd numbers

19

An Example

```
int rollDie(void)
{
   int  r;
   r = rand();
   if (r < RAND_MAX / 6) { return 1; }
   else if (r < RAND_MAX / 6 * 2) { return 2; }
   else if (r < RAND_MAX / 6 * 3) { return 3; }
   else if (r < RAND_MAX / 6 * 4) { return 4; }
   else if (r < RAND_MAX / 6 * 5) { return 5; }
   else { return 6; }
}
```

20

A General Function

- In general, what you need is not a function that chooses a number between 0 and RAND_MAX but one that chooses *a random integer between two limits that you supply*

```
int randomInt(int low, int high);
```

21

A General Function

- *Normalize* the integer result into a floating point *d* in the range $0 \le d < 1$
- *Scale* the value *d* by multiplying it by the size of the desired range *high – low* + 1
- *Truncate* the number back to an integer *k* by throwing away any fraction
- *Translate* the integer, *k* + *low*, so that the range begins at the desired lower bound

22

A General Function

```
int randomInt(int low, int high)
{
   int k;
   double d;

   d = (double) rand() / ((double) RAND_MAX + 1);
   k = (int) (d * (high – low + 1));
   return (k + low);
}
```

23

The Contents of an Interface

- Each definition exported by an interface is called an *interface entry*
- An interface entry may be
  - a *function prototype*
  - a *constant definition*
  - a *type definition*

24

The Contents of an Interface

```
#ifndef  _name_h
#define  _name_h
any required #include lines
interface entries
#endif
```

25

The Interface random.h

```
/*
 * File: random.h …
 */
#ifndef  _random_h
#define  _random_h
/*
 * Function: randomInt …
 */
int randomInt(int low, int high);
#endif
```

Comments for clients

Comments for clients

26

The Implementation random.c

```
/* File: random.c … */
#include  <stdlib.h>
#include  "random.h"

/* Function: randomInt … */
int randomInt(int low, int high) {
   int k;
   double d;
   d = (double) rand() / ((double) RAND_MAX + 1);
   k = (int) (d * (high – low + 1));
   return (k + low);
}
```

Comments for implementers

Comments for implementers

27

Testing the Library

```
#include  <stdio.h>
#include  "random.h"
int rollDie(void)
{
   return randomInt(1, 6);
}

main() {
   int  i;
   for (i = 1; i < 10; i++)
      printf("%d\n", rollDie());
}
```

```
1
4
2
5
4
3
3
6
5
```

28

Deterministic Sequence

- The sequence of results of rand is the *same* for every run of the program
- This is because rand is a *deterministic* procedure
- This deterministic property facilitates the *debugging* of programs

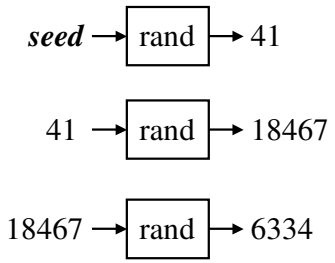
29

Initializing the Generator

- Each time rand is called, it uses the *last* random number to perform a series of calculations to produce the *next* random number
- The *value* that is used to get the entire process started is called a *seed* for the random number generator

30

Initializing the Generator



31

Randomizing the Seed

- The *seed* of the random number generator can be initialized using the function srand
  void srand(int seed);
- The function time in library time.h can be used to *randomize* the seed

```
void randomize(void) {
    srand( (int) time(NULL));
}
```

32

Evaluating the Library

- Is it *unified*? Yes!
- Is it *simple*? Yes!
- Is it *suffient*? Questionable!
- Is it *general*? Yes!
- Is it *stable*? Yes!

33

Generating Random Real Numbers

- In some applications, random *real* numbers are required
  double randomReal(double low, double high);

```
double randomReal(double low, double high)
{
   double d;
   d = (double) rand() / ((double) RAND_MAX + 1);
   return (d * (high – low + 1)) + low;
}
```

34

Simulating a Probabilistic Event

- In some applications, you need to simulate events that occur with *random probability*
  bool randomeChance(double p);

```
bool randomChance(double p)
{
   return (randomReal(0, 1) < p);
}
```

35

Simulating a Probabilistic Event

```
#include <stdio.h>
#include <stdlib.h>
main() {
   int  i;
   for (i = 1; i < 10; i++) {
      if (randomChance(0.5)) {
         printf("Heads\n");
      } else {
         printf("Tails\n");
      }
   }
}
```

36

Complete random.h

```
/* File: random.h … */
#ifndef _random_h
#define _random_h
/* Function: randomInt … */
int randomInt(int low, int high);
/* Function: randomize … */
void randomize(void);
/* Function: randomReal … */
double randomReal(double low, double high);
/* Function: randomChance … */
bool randomChance(double p);
#endif
```

37

Complete random.c

```
/* File: random.c … */
#include <stdio.h>
#include <stdlib.h>
#include "random.h"
/* Function: randomInt … */
int randomInt(int low, int high) { … }
/* Function: randomize … */
void randomize(void) { … }
/* Function: randomReal … */
double randomReal(double low, double high) { … }
/* Function: randomChance … */
bool randomChance(double p) { … }
```

38

Play the Craps Game

- Roll a pair of dice
- If the total is 2, 3, or 12, you *lose*
- If the total is 7 or 11, you *win*
- If the total is 4, 5, 6, 8, 9, or 10, this number is your *point*. You will continue to roll the dice until the total is equal to your point or 7. If the total is 7, you *lose*. If the total is your *point*, you *win*

39

Play the Craps Game

```
#include <stdio.h>
#include "random.h>

main()
{
   randomize();
   playCrapsGame();
}
```

40

Play the Craps Game

```
void playCrapsGame(void) {
   int  total, point;
   total = rollTwoDice();
   if (total == 7 || total == 11) {
      printf("You win.\n");
   } else if (total == 2 || total == 3 || total == 12) {
      printf("You lose.\n")
   } else {
      point = total;
      playPoint(point);
   }
}
```

41

Play the Craps Game

```
void playPoint(int point) {
   int  total;
   printf("Your point is %d.\n", point);
   while (TRUE) {
      total = rollTwoDice();
      if (total == point) {
         printf("You win.\n"); break;
      } else if (total == 7) {
         printf("You lose.\n"); break;
      }
   }
}
```

42

Play the Craps Game

```
int rollTwoDice(void)
{
   int  d1, d2, total;

   d1 = randomInt(1, 6);
   d2 = randomInt(1, 6);
   total = d1 + d2;
   printf("You rolled %d + %d = %d.\n", d1, d2, total);
   return total;
}
```

43