Chapter Ten

Modular Development

1

Stepwise Refinement

- At first, simple programs consist of only *one source file* and *one function* (the function main)
- Then, larger programs consist of only *one source file*, while it may contain *several functions*
- At last, complex programs may consist of *several source files* and each file contains *several functions*

2

Modules

- Each source file in a complex program is called a *module*
- The module containing the function main is called the *main module*
- Each of the other modules acts as a *library* for the main module

3

Modules

- When dividing a program into modules, it is important to *limit the extent to which the modules depend on each other*
- Each module should be carefully designed so that it can *be reused in many different programs*

4

A Case Study – Pig Latin

- Pig Latin is an invented language formed by transforming each English word based on:
- If the word begins with a *vowel*, add the suffix *way*
- If the word begins with a *consonant*, move the *initial consonant string* to the *end* and add the suffix *ay*

5

An Example

apple → appleway

scram → amscray

```
Enter a line: this is pig latin.
isthay isway igpay atinlay.
```

6

To-Down Design

- With top-down design, you successively solve a problem by dividing it into a series of simpler problems
- It is usually easier to write these simpler problems in English first
- Programs that consist of a mixture of English and C code are said to be written in *psudocode*

7

An Example

```
main()
{
   Read in a line of text from the user.
   Translate the line of text into Pig Latin.
}
```

8

An Example

```
main()
{
   string  line;

   printf("Enter a line: ");
   line = getLine();
   Translate the line of text into Pig Latin.
}
```

9

An Example

```
main()
{
   string  line;

   printf("Enter a line: ");
   line = getLine();
   translateLine(line);
}
```

10

An Example

```
void translateLine(string line)
{
   Divide the line into words.
   Translate each word into Pig Latin.
   Display each translated word.
   Display a newline to complete the line.
}
```

11

An Example

```
void translateLine(string line)
{
   while (there are any words left on the line) {
      Get the next word.
      Translate that word into Pig Latin.
      Display the translated word.
   }
   Display a newline to complete the line.
}
```

12

Spaces and Punctuation

```
Enter a line: this is pig latin.
isthayiswayigpayatinlay
```

13

Refining the Definition of a Word

this is pig latin. → 4 words

this␣is␣pig␣latin␣ → 8 words

- A sequence of characters that acts as a coherent unit is called a *token*

14

An Example

```
void translateLine(string line)
{
  while (there are any tokens left on the line) {
    Get the next token.
    if (the token is a regular English word) {
      Replace the token by is Pig Latin translation.
    }
    Display the token.
  }
  Display a newline to complete the line.
}
```

15

Token Scanning

- The division of the input text into a sequence of tokens is called *token scanning*
- Token scanning occurs in all language processing applications, e.g., compilers and WWW browser. Hence, it is suitable to be designed as a separate module

16

The Token Scanner

- The token scanner needs two functions
  - atEndOfLine( )
  - getNextToken( )
- How could getNextToken return different tokens each time it is called?
  - maintain the *internal state* within the module
  - the drawLine function in graphics library
  - the rand function in stdlib library

17

The Token Scanner

```
void initScanner(string line);

string getNextToken(void);

bool atEndOfLine(void);
```

18

An Example

```
void translateLine(string line)
{
  string  token;
  initScanner(line);
  while (!atEndOfLine( )) {
    token = getNextToken( );
    if (isLegalWord(token)) token = translateWord(token);
    printf("%s", token);
  }
  printf("\n");
}
```

19

An Example

```
bool isLegalWord(string token)
{
  int  i;

  for (i = 0; i < stringLength(token); i++) {
    if (!isalpha((ithChar(token, i))) return FALSE;
  }
  return TRUE;
}
```

20

An Example

```
string translateWord(string word)
{
  Find the position of the first vowel.
  if (the vowel appears at the beginning of the word) {
    return the word concatenated with "way".
  } else {
    head = extract the initial consonant substring.
    tail = extract the rest substring.
    Return the tail, concatenated with head,
      concatenated with "ay".
  }
}
```

21

An Example

```
string translateWord(string word) {
  int  vp;    string  head, tail;
  vp = findFirstVowel(word);
  if (vp == -1) {
    return word;
  } else if (vp == 0) {
    return concat(word, "way");
  } else {
    head = subString(word, 0, vp – 1);
    tail = subString(word, vp, stringLength(word) – 1);
    return concat(tail, concat(head, "ay"));
  }
}
```

22

An Example

```
int findFirstVowel(string word)
{
  int  i;

  for (i = 0; i < stringLength(word); i++) {
    if (isVowel((ithChar(word, i))) return i;
  }
  return -1;
}
```

23

An Example

```
bool isVowel(char ch)
{
  switch (tolower(ch)) {
    case 'a': case 'e': case 'i': case 'o': case 'u':
      return TRUE;
    default:
      return FALSE;
  }
}
```

24

Local Variables

- Variables declared inside a function are called *local* variables
- The memory locations for local variables *exist* when the function is *called*, and *vanish* when the function is *returned*
- These memory locations are created in a region called the *stack area*

25

An Example

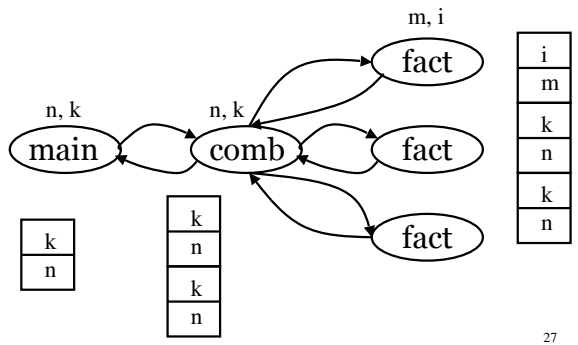
```
int fact(int m) {
   int  product, i;
   product = 1;
   for (i = 1; i <= m; i++)
      product *= i;
   return product;
}
```

```
main( ) {
   int n, k;
   scanf("%d %d", &n, &k);
   printf("C(%d, %d) = %d\n",
            n, k, comb(n, k));
}
```

```
int comb(int n, int k) {
   return fact(n) / (fact(k) * fact(n – k) ;
```

26

An Example



27

Global Variables

- Variables declared outside functions are called *global* variables
- The memory locations for global variables *exist* when the program *starts*, and *vanish* when the program *terminates*
- These memory locations are created in a region called the *static area*

28

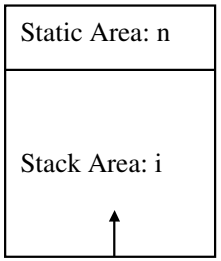
An Example

```
int n = 1;

int genNumber(void) {
   return n++;
}

main() {
   int i;
   for (i = 0; i < 10; i++) {
      printf("%d\n", getNumber());
}
```

29

An Example

Data section for a program:



30

Scope of Variables

- The portion of the program in which a variable can be used is called the *scope* of the variable
- A *local* variable can only be used *within a single function*
- A *global* variable can be used *within all functions* in the program

31

The Dangers of Global Variables

- *Parameter passing* is one way to communicate information between functions
  - Access of information is under better control
- *Using global variables* is another way to communicate information between functions
  - Access of information may be out of control
- Global variables should be used *carefully*

32

Keep Global Variables Private

- It is best to ensure that *each global variable is never used by more than one module*
- A global variable can be declared as *private* to a single module using the keyword *static*

```
static int n;
```

33

Initialization of Global Variables

- A global variable can be assigned an initial value before program starts. This is called *static initialization*

```
static int n = 1;
```

- A global variable can also be assigned an initial value after program starts. This is called *dynamic initialization*

```
initScanner(line);
```

34

Cases for Static Initialization

- If the value of a variable is *constant* throughout the lifetime of the program
- If a variable usually has a *particular initial value* that only a few clients might want to change
- Values that are used within a program unless a client specifically changes them are called *default values*

35

Private Functions

- The keyword *static* can also be used to indicate that a function is *private* to a particular module
- All functions other than main and those *explicitly exported* by an interface should be declared as *static*

36

pigLatin.c

```
static void translateLine(string line);
static string translateWord(string word);
static bool isLegalWord(string token);
static  int  findFirstVowel(string word);
static  bool  isVowel(char ch);
```

37

The Token Scanner

```
static  string  buffer;
static  int  buflen;
static  int  cpos;
```

38

initScanner

```
void initScanner(string line)
{
   buffer = line;
   buflen = stringLength(buffer);
   cpos = 0;
}
```

39

atEndOfLine

```
bool atEndOfLine(void)
{
   return cpos >= buflen;
}
```

40

getNextToken

```
string getNextToken(void)
{
   char  ch;  int  start;

   if (cpos >= buflen) {
      printf("Error: No more tokens");
      exit(1);
   }
   ch = ithChar(buffer, cpos);
   /* next page */
}
```

41

getNextToken

```
if (isalnum(ch)) {
  start = cpos;
  while (cpos < buflen && isalnum(ithChar(buffer,cpos))) {
        cpos++;
  }
  return subString(buffer, start, cpos – 1);
} else {
  cpos++;
  return charToString(ch);
}
```

42

Summary

```
/* xxx.h */
f1();
f2();
f3();
```

```
/* xxx.c */
static int v1;
static int v2;
f1() { …}
f2() { …}
f3() { …}
static h1() { …}
static h2() { …}
```

43