

3.2 Program Components in C++

- Modules: functions and classes
- Programs use new and “prepackaged” modules
 - New: programmer-defined functions, classes
 - Prepackaged: from the standard library
- Functions invoked by function call
 - Function name and information (arguments) it needs
- Function definitions
 - Only written once
 - Hidden from other functions



3.3 Math Library Functions

- Perform common mathematical calculations
 - Include the header file `<cmath>`
- Example

```
cout << sqrt( 900.0 );
```

 - sqrt (square root) function The preceding statement would print 30
 - All functions in math library return a **double**



3.6 Function Prototypes

- Function prototype contains
 - Function name
 - Parameters (number and data type)
 - Return type (**void** if returns nothing)
 - Only needed if function definition after function call
- Prototype must match function definition
 - Function prototype


```
double maximum( double, double, double );
```
 - Definition


```
double maximum( double x, double y, double z )
{
  ...
}
```



3.7 Header Files

- Header files contain
 - Function prototypes
 - Definitions of data types and constants
- Header files ending with .h
 - Programmer-defined header files


```
#include "myheader.h"
```
- Library header files


```
#include <cmath>
```



3.8 Random Number Generation

- **rand** function (`<cstdlib>`)
 - `i = rand();`
 - Generates unsigned integer between 0 and `RAND_MAX` (usually 32767)
- Scaling and shifting
 - Modulus (remainder) operator: `%`
 - `10 % 3` is 1
 - `x % y` is between 0 and `y - 1`
 - Example
 - `i = rand() % 6 + 1;`
 - `"Rand() % 6"` generates a number between 0 and 5 (scaling)
 - `"+ 1"` makes the range 1 to 6 (shift)



3.8 Random Number Generation

- Calling `rand()` repeatedly
 - Gives the same sequence of numbers
- Pseudorandom numbers
 - Preset sequence of "random" numbers
 - Same sequence generated whenever program run
- To get different random sequences
 - Provide a seed value
 - Like a random starting point in the sequence
 - The same seed will give the same sequence
 - `srand(seed);`
 - `<cstdlib>`
 - Used before `rand()` to set the seed



3.8 Random Number Generation

- Can use the current time to set the seed
 - No need to explicitly set seed every time
 - `srand(time(0));`
 - `time(0);`
 - `<ctime>`
 - Returns current time in seconds
- General shifting and scaling
 - $Number = shiftingValue + rand() \% scalingFactor$
 - `shiftingValue` = first number in desired range
 - `scalingFactor` = width of desired range



3.9 Example: Game of Chance and Introducing enum

- Enumeration
 - Set of integers with identifiers
 - `enum typeName {constant1, constant2...};`
 - Constants start at 0 (default), incremented by 1
 - Constants need unique names
 - Cannot assign integer to enumeration variable
 - Must use a previously defined enumeration type
- Example


```
enum Status {CONTINUE, WON, LOST};
Status enumVar;
enumVar = WON; // cannot do enumVar = 1
```



3.9 Example: Game of Chance and Introducing enum

- Enumeration constants can have preset values


```
enum Months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};
```

 - Starts at 1, increments by 1
- Next: craps simulator
 - Roll two dice
 - 7 or 11 on first throw: player wins
 - 2, 3, or 12 on first throw: player loses
 - 4, 5, 6, 8, 9, 10
 - Value becomes player's "point"
 - Player must roll his point before rolling 7 to win



```

1 // Fig. 3.10: fig03_10.cpp
2 // Craps.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 // contains function prototypes for functions srand and rand
9 #include <cstdlib>
10
11 #include <ctime> // contains pr
12
13 int rollDice( void ); // function prototype
14
15 int main()
16 {
17     // enumeration constants represent game status
18     enum Status { CONTINUE, WON, LOST };
19
20     int sum;
21     int myPoint;
22
23     Status gameStatus; // can contain CONTINUE, WON or LOST
24

```

Function to roll 2 dice and return the result as an `int`.

Enumeration to keep track of the current game.



Outline

fig03_10.cpp
(1 of 5)

```

25 // randomize random number generator using current time
26 srand( time( 0 ) );
27
28 sum = rollDice(); // first
29
30 // determine game status and
31 switch ( sum ) {
32
33     // win on first roll
34     case 7:
35     case 11:
36         gameStatus = WON;
37         break;
38
39     // lose on first roll
40     case 2:
41     case 3:
42     case 12:
43         gameStatus = LOST;
44         break;
45

```

switch statement
determines outcome based on
die roll.



Outline

fig03_10.cpp
(2 of 5)

11

© 2003 Prentice Hall, Inc.
All rights reserved.

```

46 // remember point
47 default:
48     gameStatus = CONTINUE;
49     myPoint = sum;
50     cout << "Point is " << myPoint << endl;
51     break; // optional
52
53 } // end switch
54
55 // while game not complete ...
56 while ( gameStatus == CONTINUE ) {
57     sum = rollDice(); // roll dice again
58
59     // determine game status
60     if ( sum == myPoint ) // win by making point
61         gameStatus = WON;
62     else
63         if ( sum == 7 ) // lose by rolling 7
64             gameStatus = LOST;
65
66 } // end while
67

```



Outline

fig03_10.cpp
(3 of 5)

12

© 2003 Prentice Hall, Inc.
All rights reserved.

```

68 // display won or lost message
69 if ( gameStatus == WON )
70     cout << "Player wins" << endl;
71 else
72     cout << "Player loses" << endl;
73
74 return 0; // indicates successful termination
75
76 } // end main
77
78 // roll dice, calculate sum and d
79 int rollDice( void )
80 {
81     int die1;
82     int die2;
83     int workSum;
84
85     die1 = 1 + rand() % 6; // pick random die1 value
86     die2 = 1 + rand() % 6; // pick random die2 value
87     workSum = die1 + die2; // sum die1 and die2
88

```

Function rollDice takes no arguments, so has void in the parameter list.



Outline

13

fig03_10.cpp
(4 of 5)

© 2003 Prentice Hall, Inc.
All rights reserved.

```

89 // display results of this roll
90 cout << "Player rolled " << die1 << " + " << die2
91     << " = " << workSum << endl;
92
93 return workSum; // return sum of dice
94
95 } // end function rollDice

```

```

Player rolled 2 + 5 = 7
Player wins

Player rolled 6 + 6 = 12
Player loses

Player rolled 3 + 3 = 6
Point is 6
Player rolled 5 + 3 = 8
Player rolled 4 + 5 = 9
Player rolled 2 + 1 = 3
Player rolled 1 + 5 = 6
Player wins

```



Outline

14

fig03_10.cpp
(5 of 5)

fig03_10.cpp
output (1 of 2)

© 2003 Prentice Hall, Inc.
All rights reserved.

```
Player rolled 1 + 3 = 4
Point is 4
Player rolled 4 + 6 = 10
Player rolled 2 + 4 = 6
Player rolled 6 + 4 = 10
Player rolled 2 + 3 = 5
Player rolled 2 + 4 = 6
Player rolled 1 + 1 = 2
Player rolled 4 + 4 = 8
Player rolled 4 + 3 = 7
Player loses
```



Outline

15

fig03_10.cpp
output (2 of 2)

© 2003 Prentice Hall, Inc.
All rights reserved.

16

3.10 Storage Classes

- Variables have attributes
 - Have seen name, type, size, value
 - Storage class
 - How long variable exists in memory
 - Scope
 - Where variable can be referenced in program
 - Linkage
 - For multiple-file program (see Ch. 6), which files can use it



3.10 Storage Classes

- Automatic storage class
 - Variable created when program enters its block
 - Variable destroyed when program leaves block
 - Only local variables of functions can be automatic
 - Automatic by default
 - keyword **auto** explicitly declares automatic
 - **register** keyword
 - Hint to place variable in high-speed register
 - Good for often-used items (loop counters)
 - Often unnecessary, compiler optimizes
 - Specify either **register** or **auto**, not both
 - **register int counter = 1;**



3.10 Storage Classes

- Static storage class
 - Variables exist for entire program
 - For functions, name exists for entire program
 - May not be accessible, scope rules still apply (more later)
- **static** keyword
 - Local variables in function
 - Keeps value between function calls
 - Only known in own function
- **extern** keyword
 - Default for global variables/functions
 - Globals: defined outside of a function block
 - Known in any function that comes after it



3.11 Scope Rules

- Scope
 - Portion of program where identifier can be used
- File scope
 - Defined outside a function, known in all functions
 - Global variables, function definitions and prototypes
- Function scope
 - Can only be referenced inside defining function
 - Only labels, e.g., identifiers with a colon (**case :**)



3.11 Scope Rules

- Block scope
 - Begins at declaration, ends at right brace }
 - Can only be referenced in this range
 - Local variables, function parameters
 - **static** variables still have block scope
 - Storage class separate from scope
- Function-prototype scope
 - Parameter list of prototype
 - Names in prototype optional
 - Compiler ignores
 - In a single prototype, name can be used once



3.15 Functions with Empty Parameter Lists

- Empty parameter lists
 - **void** or leave parameter list empty
 - Indicates function takes no arguments
 - Function **print** takes no arguments and returns no value
 - `void print();`
 - `void print(void);`



3.16 Inline Functions

- Inline functions
 - Keyword **inline** before function
 - Asks the compiler to copy code into program instead of making function call
 - Reduce function-call overhead
 - Compiler can ignore **inline**
 - Good for small, often-used functions
- Example


```
inline double cube( const double s )
    { return s * s * s; }
```

 - **const** tells compiler that function does not modify **s**
 - Discussed in chapters 6-7



3.17 References and Reference Parameters

- Call by value
 - Copy of data passed to function
 - Changes to copy do not change original
 - Prevent unwanted side effects
- Call by reference
 - Function can directly access data
 - Changes affect original

