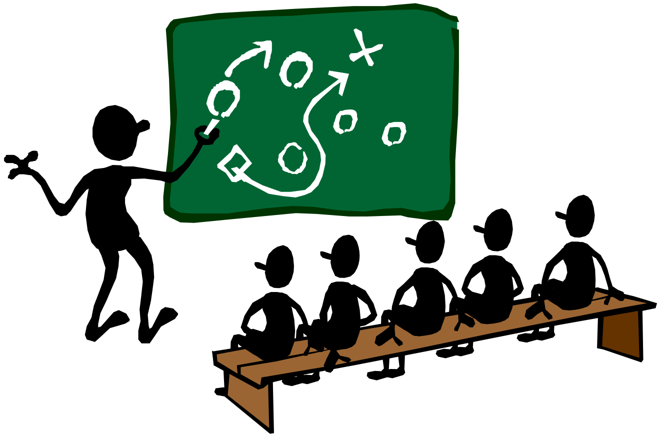
C++ Programming Language
Chapter 3 Function Basics



Juinn-Dar Huang

Associate Professor

jdhuang@mail.nctu.edu.tw

February 2011

Learning Objectives

- Predefined functions
  - those that return a value and those that don’t
- Programmer-defined functions
  - declaration, definition, call
  - recursive functions
- Scope rules
  - local names (constants, variables, …)
  - global names
  - name scope and name hiding

Introduction to Functions

- Building blocks of programs
- Other terminology in other languages:
  - procedures, subprograms, subroutines, methods, …
  - in C++: functions
- I-P-O
  - Input – Process – Output
  - basic subparts to any program
  - use functions for these pieces

Predefined Functions

- Libraries full of functions for our use!
- Two types:
  - those that return a value
  - those that do not (i.e., return void)
- Must #include appropriate library header file
  - e.g.,
    - <cmath>, <cstdlib> (Original C libraries)
    - <iostream> (for cout, cin, …)

Using Predefined Functions

- Math functions are very plentiful
  - found in library <cmath>
  - most return a value (answer)
- Example: double root = sqrt(9.0);
  - components:

sqrt ➔	name of library function
root ➔	variable used to get the returned value
9.0 ➔	argument (or parameter) for function

  - in I-P-O:
    - I = 9.0
    - P = “compute the square root”
    - O = 3, which is returned & assigned to root

Function Call

- Back to this assignment:

  double root = sqrt(9.0);

  – the expression sqrt(9.0) is known as a function *call*, or function *invocation*

  – the argument in a function call 9.0 can be a literal, a variable, or an expression

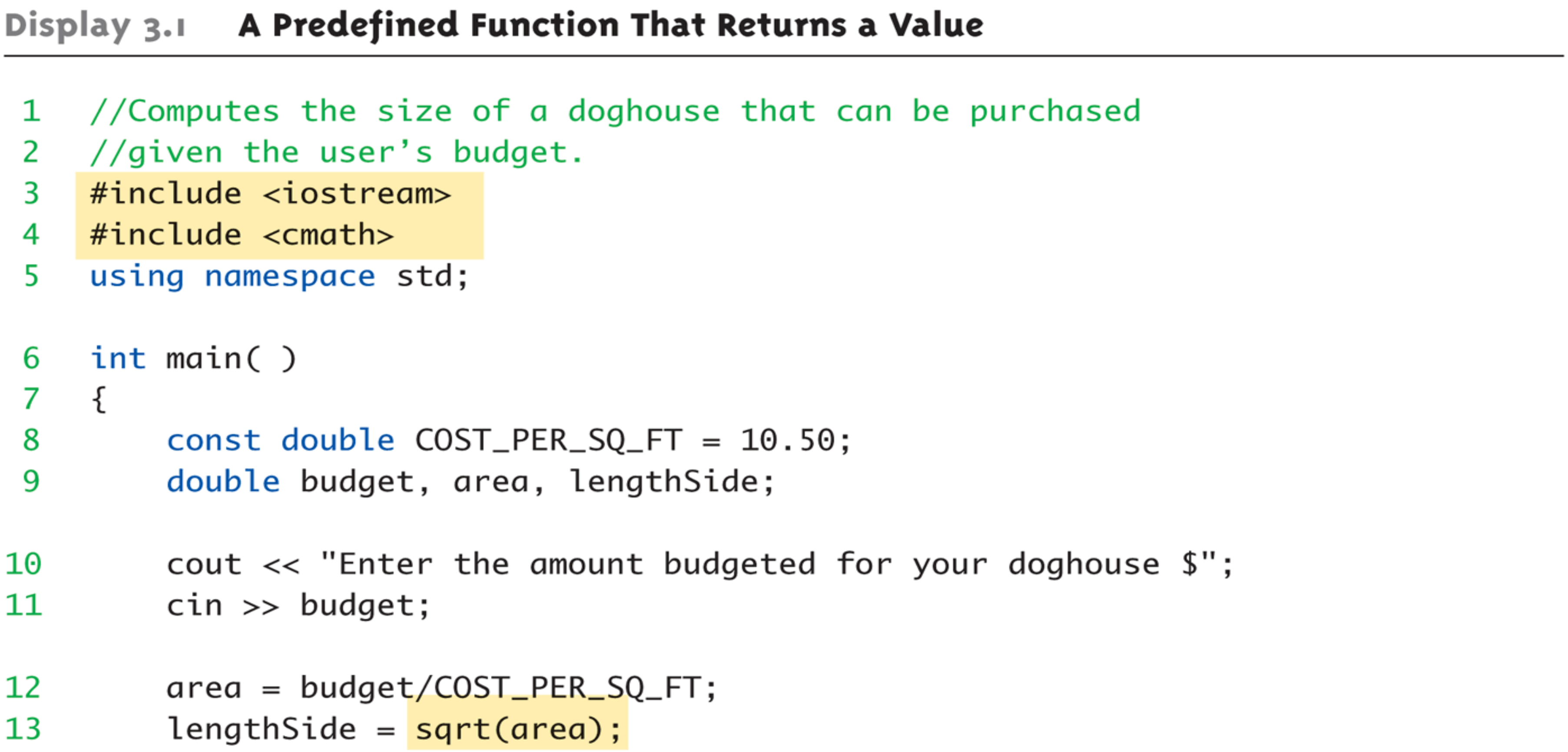
  sqrt(9.0)   sqrt(root)   sqrt(root / 2.0)

  – the call itself can be part of an expression:

  bonus = sqrt(sales)/10.0;

    - a function call is allowed wherever it is legal to use an expression of the function’s return type

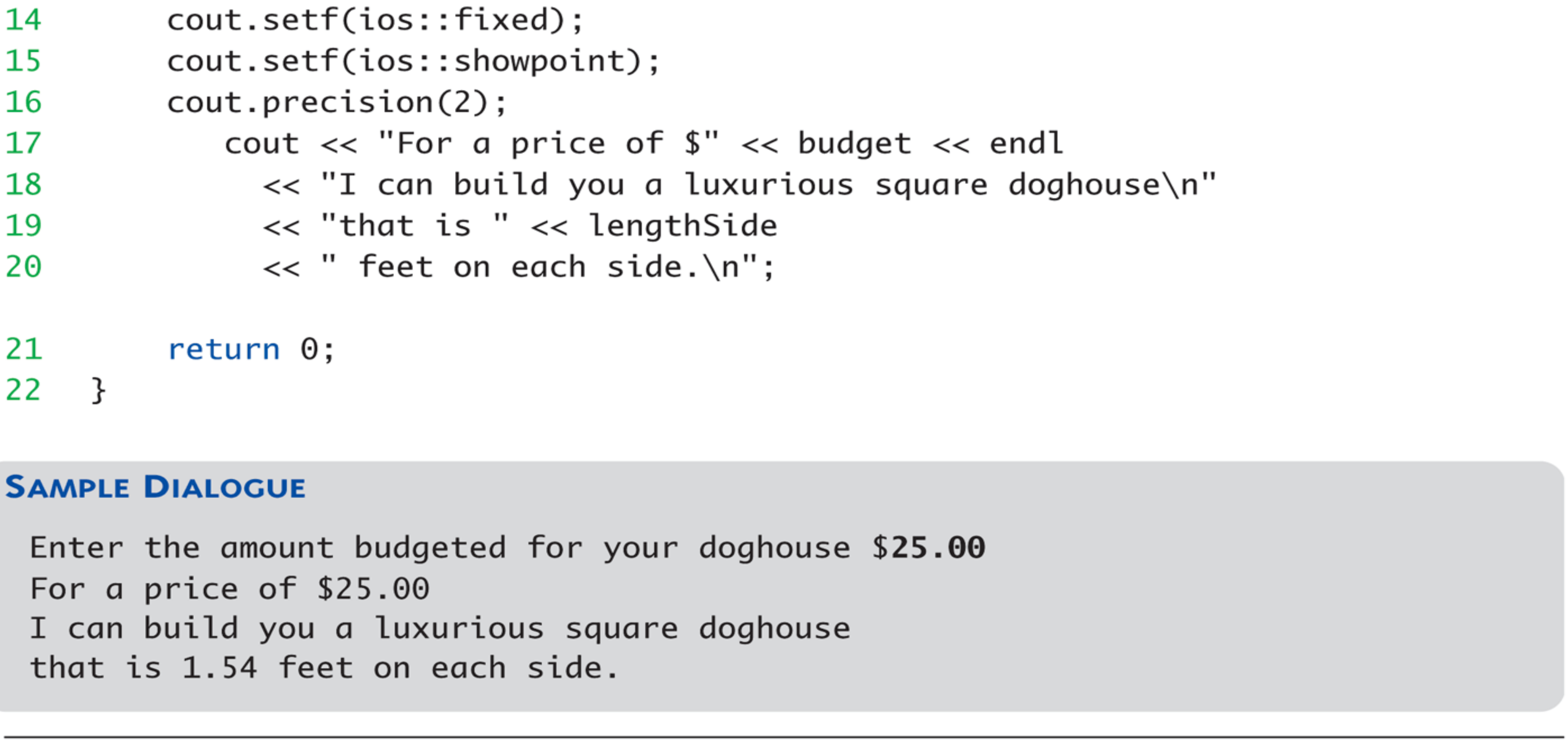
A Predefined Function That Returns a Value (1/2)



Display 3.1 A Predefined Function That Returns a Value

```
//Computes the size of a doghouse that can be purchased
//given the user’s budget.
#include <iostream>
#include <cmath>
using namespace std;

int main( )
{
    const double COST_PER_SQ_FT = 10.50;
    double budget, area, lengthSide;

    cout << "Enter the amount budgeted for your doghouse $";
    cin >> budget;

    area = budget/COST_PER_SQ_FT;
    lengthSide = sqrt(area);
```

A Predefined Function That Returns a Value (2/2)



```
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
       cout << "For a price of $" << budget << endl
         << "I can build you a luxurious square doghouse\n"
         << "that is " << lengthSide
         << " feet on each side.\n";

    return 0;
}
```

SAMPLE DIALOGUE

```
Enter the amount budgeted for your doghouse $25.00
For a price of $25.00
I can build you a luxurious square doghouse
that is 1.54 feet on each side.
```

More Predefined Functions

- #include <cstdlib>
  - library contains functions like:
    - int abs(int)   // returns absolute value of an int
    - long labs(long)   // returns absolute value of a long int
    - double fabs(double)   // returns absolute value of a double
  - fabs() is actually in library <cmath>
    - can be confusing
    - for historical reasons
    - check Appendix 4 for more (still partial) library functions or C++ related manuals for details

A Predefined Math Function pow

- double pow(double x, double y); // declaration
  - returns x to the power y
  - e.g.,

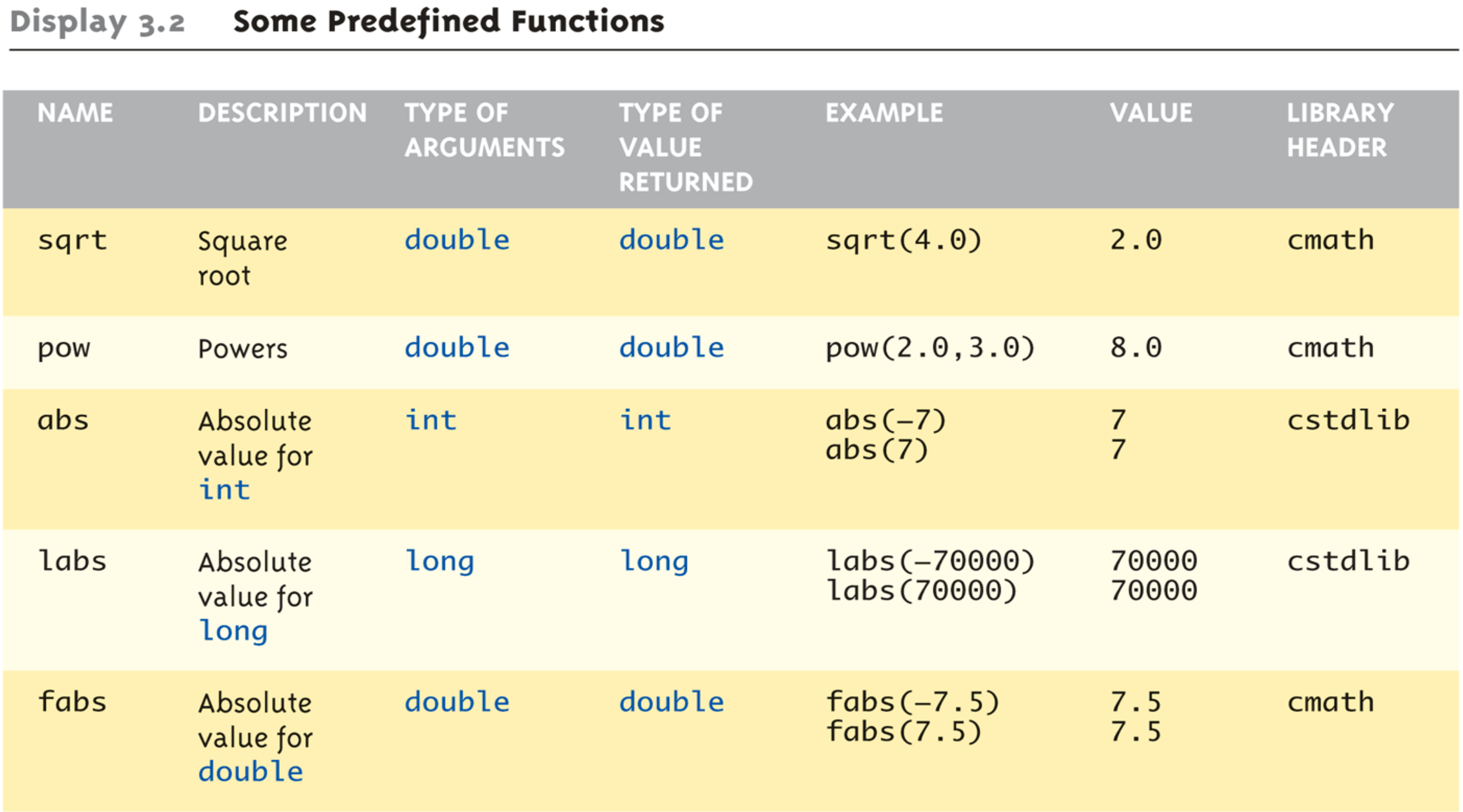
```
double result, x = 3.0, y = 2.0;
result = pow(x, y);     // function call
cout << result;
```

    - Here 9.0 is displayed since $3.0^{2.0}$ = 9.0

★ **double pow(double d, int i) also supported in <cmath>**

- Notice this function receives two arguments
  - a function can have any number of arguments, of varying data types
  - a function can have no argument as well

More Predefined Math Functions (1/2)



Display 3.2 **Some Predefined Functions**

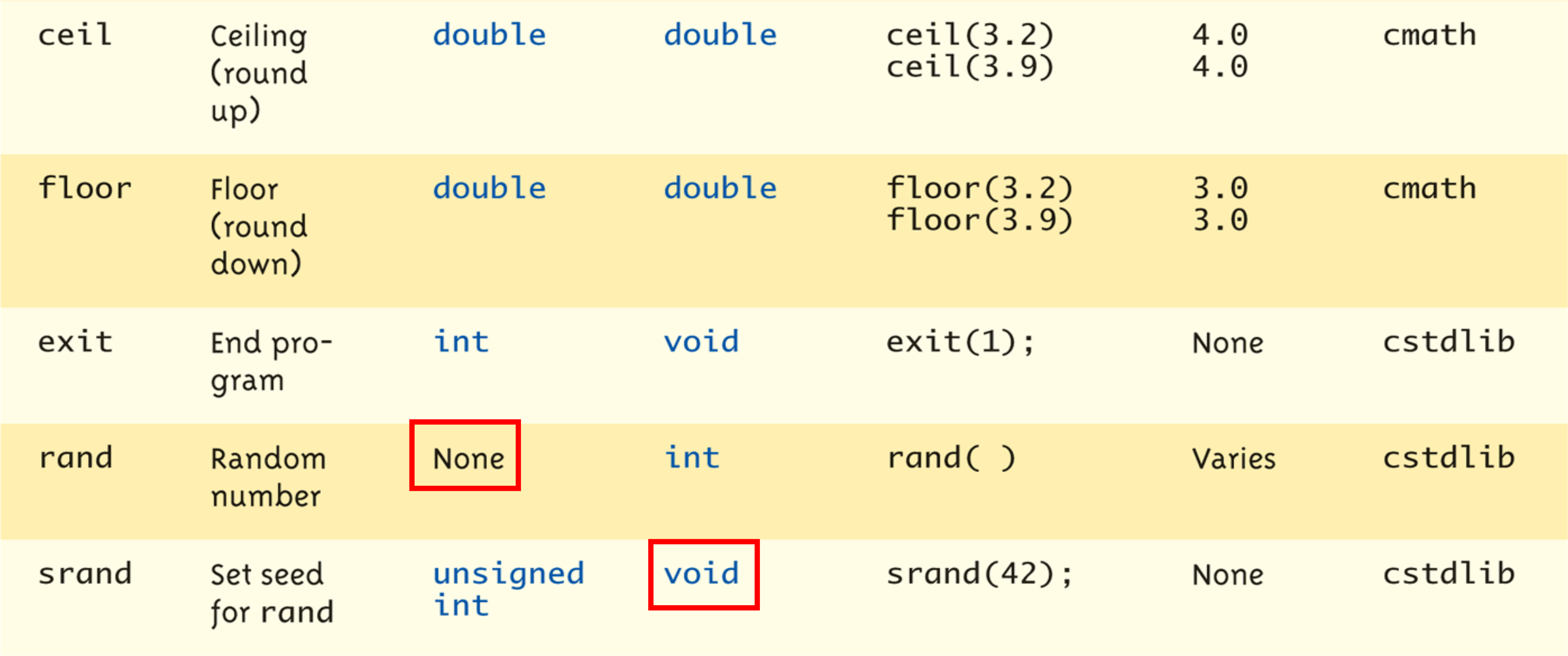
NAME	DESCRIPTION	TYPE OF ARGUMENTS	TYPE OF VALUE RETURNED	EXAMPLE	VALUE	LIBRARY HEADER
`sqrt`	Square root	`double`	`double`	`sqrt(4.0)`	`2.0`	`cmath`
`pow`	Powers	`double`	`double`	`pow(2.0,3.0)`	`8.0`	`cmath`

`abs`	Absolute value for `int`	`int`	`int`	`abs(-7)`
				`abs(7)`
					`7`
					`7`
						`cstdlib`

`labs`	Absolute value for `long`	`long`	`long`	`labs(-70000)`
				`labs(70000)`
					`70000`
					`70000`
						`cstdlib`

`fabs`	Absolute value for `double`	`double`	`double`	`fabs(-7.5)`
				`fabs(7.5)`
					`7.5`
					`7.5`
						`cmath`

More Predefined Math Functions (2/2)



ceil	Ceiling (round up)	double	double	ceil(3.2)
				ceil(3.9)
					4.0
					4.0
						cmath

floor	Floor (round down)	double	double	floor(3.2)
				floor(3.9)
					3.0
					3.0
						cmath

exit	End program	int	void	exit(1);	None	cstdlib
rand	Random number	None	int	rand( )	Varies	cstdlib
srand	Set seed for rand	unsigned int	void	srand(42);	None	cstdlib

Check www.cplusplus.com/reference/ for more details

Predefined Void Functions

- **No** return value
- Performs an action, but sends no answer out
- When called, it is a statement itself
  - exit(1); // no return value, so not assigned
    - this call terminates program
- All aspects same as functions that return a value
  - they just don't return a value!

For example → void func(int a, double b);

Pseudo-Random Number Generator

- Return a pseudo-randomly chosen number
- Used for simulations, games, …
  - rand() // in <cstdlib>
    - takes no arguments
    - returns value between 0 and RAND_MAX (defined in <cstdlib>) **uniformly**
  - scaling
    - squeezes random number into smaller range
      e.g., rand() % 6
    - returns random value between 0 and 5
  - shifting
    e.g., rand() % 6 + 1
    - shifts range between 1 and 6 (e.g., die roll)

Random Number Seed

- Pseudo-random numbers
  - calls to rand() produce a given sequence of random numbers
  - a built-in algorithm produces that sequence based on a given seed
  - different/same seed ➔ different/same sequence
- Use different seed to alter that sequence
  void srand(unsigned int seed);
  - void function (nothing returned)
  - need one unsigned integer argument, i.e., the seed
  - can use any seed value, including system time:
    srand(time(0));
    - time(0) returns system time (an unsigned integral value) as the seed
    - library <ctime> contains time() functions

Random Examples

- Random integer between 1 and 6:
  rand() % 6 + 1
  - "%" is modulus operator (remainder)

- Random double between 0.0 and 1.0:
  rand() / static_cast<double>(RAND_MAX)
  - static type cast used to force double-precision division

Programmer-Defined Functions

- Write your own functions!
- Building blocks of programs
  - divide and conquer
  - readability and maintainability
  - reuse
- Function definition can be in either:
  - same file as main()
  - separate file so others can use it, too

Components of Function Use

- 3 pieces for using functions:
  - function declaration (or function prototype)
    - information required by compiler
    - to properly interpret calls
  - function definition
    - actual implementation/code for what function does
  - function call (or function invocation)
    - use the specified function
    - transfer control to function

Function Declaration

- Also called function prototype
- An informational declaration for compiler
- Tell compiler how to interpret calls
  - syntax:
    <return_type> FuncName(<formal-parameter-list>);
  - example:
    double totalCost(int numberParameter, double priceParameter);
    or,
    double totalCost(int, double);

optional

- Placed before any calls
  - again, declaration-before-use scenario

Function Definition

- Implementation of function
- Just like implementing function main()
- Example:

formal parameter, mandatory

```
double totalCost( int numberParameter,
                  double priceParameter)
{
    const double TAXRATE = 0.05;
    double subTotal;
    subtotal = priceParameter * numberParameter;
    return (subtotal + subtotal * TAXRATE);
}
```

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Function Definition Placement

- Placed outside function main()
- Actually, no function is ever part of another
  - i.e., you can NOT define another function inside a function
- Formal parameters in definition
  - placeholders for data sent in
    - variable name used to refer to data in function definition
- return statement
  - sends answer back to caller

Word Bank

Caller
Callee

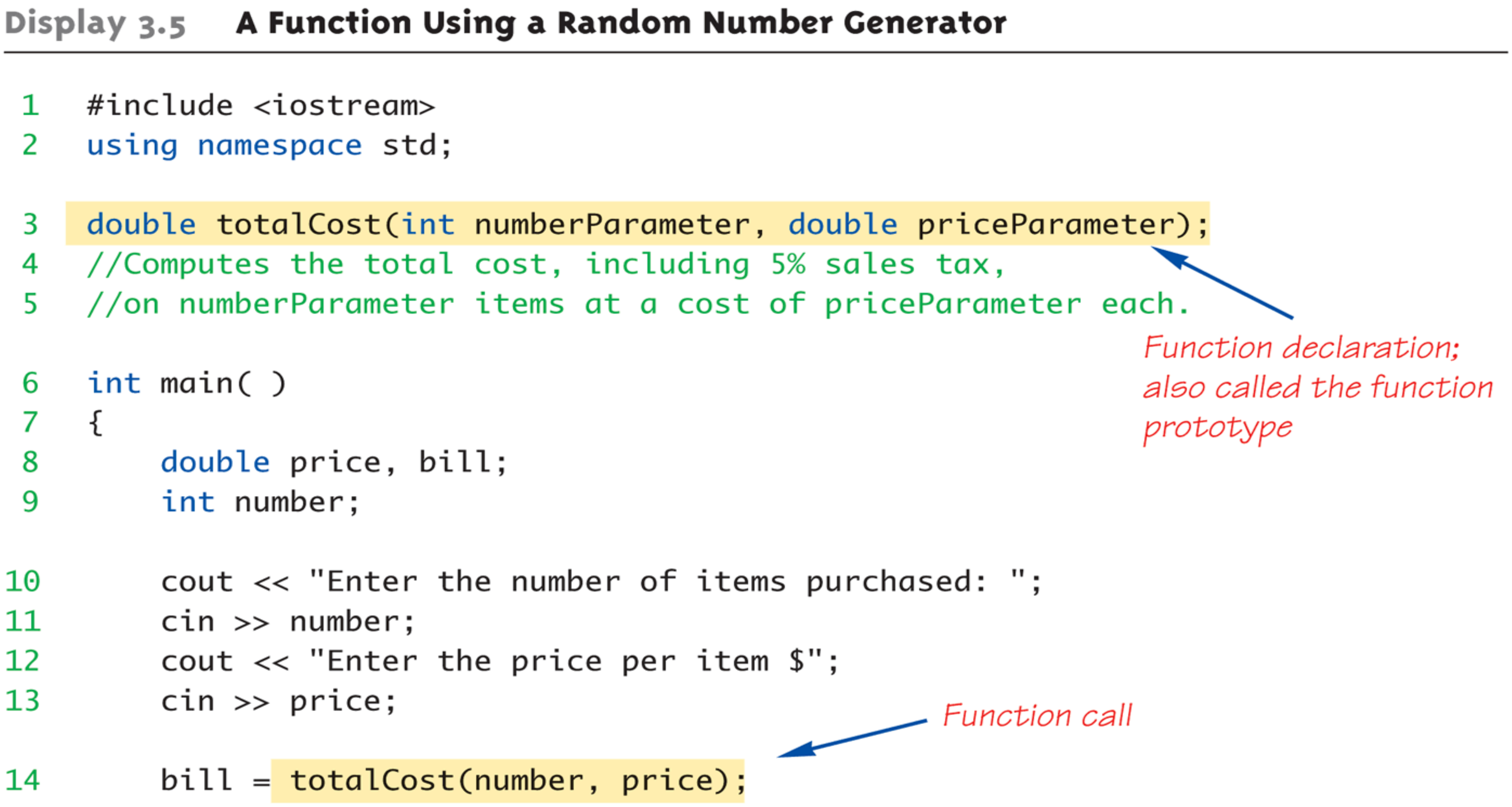
Function Call

- Just like calling predefined function
  bill = totalCost(number, price);

actual argument, mandatory

- Recall: totalCost returns double value
  - assigned to a variable named bill
- Arguments here: number, price
  - recall arguments can be literals, variables, expressions, or combination of above
  - in function call, arguments often called **actual** arguments
    - because they contain the actual data being sent

Function Example (1/2)



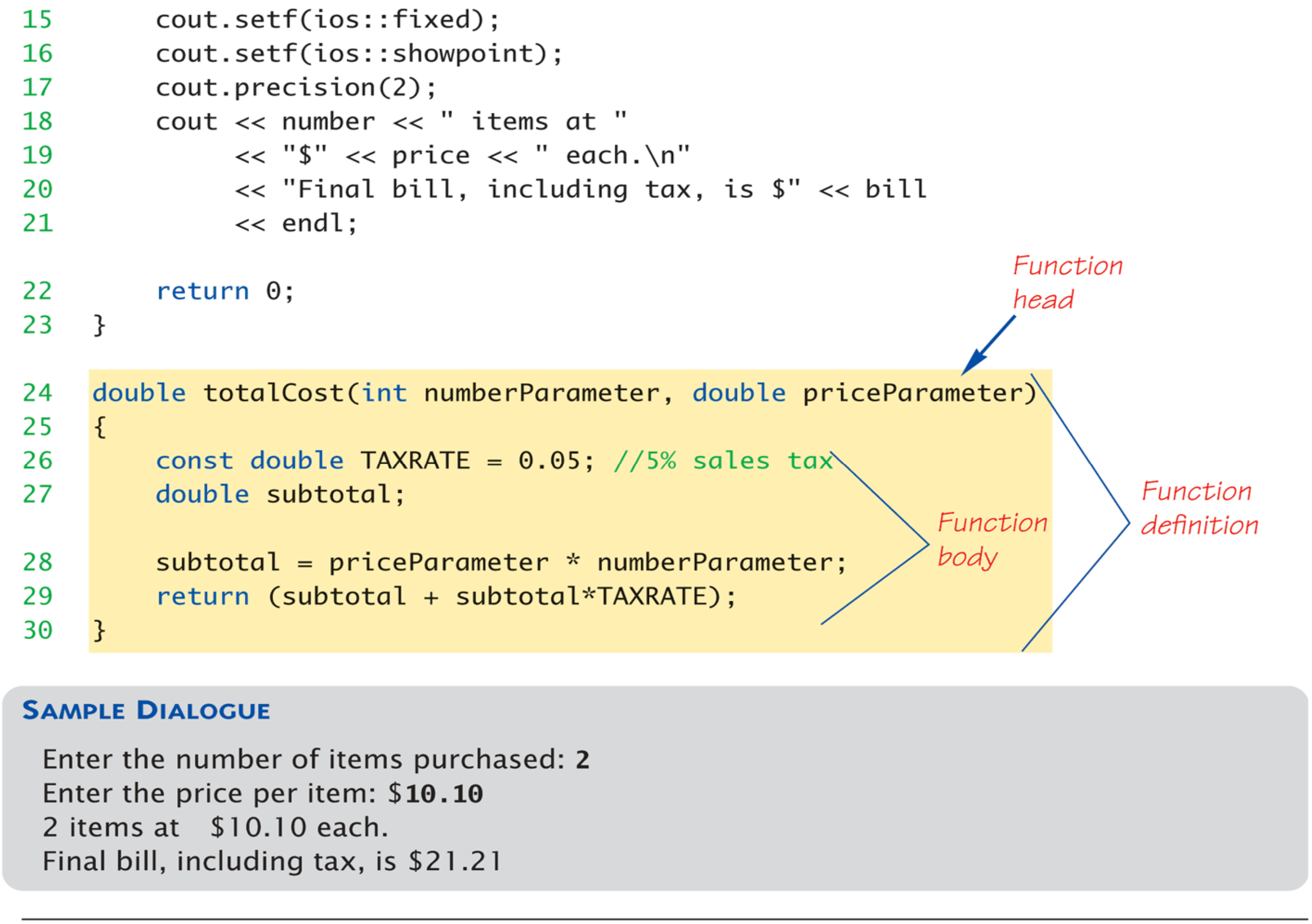
Display 3.5 **A Function Using a Random Number Generator**

```
#include <iostream>
using namespace std;

double totalCost(int numberParameter, double priceParameter);
//Computes the total cost, including 5% sales tax,
//on numberParameter items at a cost of priceParameter each.

int main( )
{
    double price, bill;
    int number;

    cout << "Enter the number of items purchased: ";
    cin >> number;
    cout << "Enter the price per item $";
    cin >> price;

    bill = totalCost(number, price);
```

Function declaration; also called the function prototype

Function call

Function Example (2/2)



```
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << number << " items at "
         << "$" << price << " each.\n"
         << "Final bill, including tax, is $" << bill
         << endl;

    return 0;
}

double totalCost(int numberParameter, double priceParameter)
{
    const double TAXRATE = 0.05; //5% sales tax
    double subtotal;

    subtotal = priceParameter * numberParameter;
    return (subtotal + subtotal*TAXRATE);
}
```

SAMPLE DIALOGUE

```
Enter the number of items purchased: 2
Enter the price per item: $10.10
2 items at  $10.10 each.
Final bill, including tax, is $21.21
```

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Factorial

```
#include <iostream>
using namespace std;

int factorial(int n); // function declaration, n is the optional formal parameter

int main() {
    int i = 8;
    cout << “8! = ” << factorial(i) << endl; // function call, i is the actual argument
    cout << “6! = ” << factorial(6) << endl; // function call, 6 is the actual argument
     return 0;
}

int factorial(int fac) { // function definition, fac is the mandatory formal parameter
    int result = 1;
    for( ; fac > 1; --fac)
            result *= fac;
    return result;
}
```

Alternative Function Declaration

- Function declaration just provides information required by compiler
- Compiler only needs to know:
  - return type
  - function name
  - list of parameter types
- Formal parameter names are not required actually

  double totalCost(int, double); // work perfectly

- You can still put in formal parameter names
  - improves readability
  - compiler simply ignores them

Be Careful: Argument Order

- pow() provided by <cmath>

```
double  pow(double base, double exponent);

int main() {
double result;
// …

// want to calculate 5^3
result = pow(5.0, 3.0);           // get what you want
result = pow(3.0, 5.0);           // Oops, no compilation error! Be careful!
result = pow(“abc”, “def”)      // compilation error

//…
}
```

C and C++ use positional argument mapping

Parameter vs. Argument

- Terms often used interchangeably
- Formal parameters/arguments
  - in function declaration
  - in the header of function definition
- Actual parameters/arguments
  - in function call
- Technically, parameter is formal piece while argument is actual piece
  - however, terms not always used this way

Calling Functions in a Function

- We are already doing this!
  - main() IS a function!
- Only requirement:
  - declaration of the called function (callee) must appear first
- Function’s definition typically elsewhere
  - e.g., after main(), in a separate file, in a library
- Function
  - declaration: can be multiple as long as they are consistent
  - definition: one and only one
  - call: can be multiple, of course
- Function can even call itself ➔ Recursion (Chap 13)

Boolean Return-Type Functions

- Return-type can be any valid type
  - given function declaration:
    ```
    bool appropriate(int rate);
    ```
  - function definition:
    ```
    bool appropriate (int rate) {
                return ( ( (rate>=10) && (rate<20) ) || (rate==0) );
    }
    ```
  - return value is either true or false
  - function call from some other function:
    ```
    if ( appropriate(entered_rate) )
        cout << “Rate is valid\n”;
    ```

Declaring void Functions

- Similar to functions returning a value
- Return type specified as void
- Example:
  - function declaration:
    void showResults(double fDegrees, double cDegrees);
    - return-type is void
    - nothing is returned

Defining void Functions

- Function definition:

```
void showResults(double fDegrees, double cDegrees) {
        cout.setf(ios::fixed);
        cout.setf(ios::showpoint);
        cout.precision(1);
        cout      << fDegrees << " degrees fahrenheit equals \n"
                  << cDegrees << " degrees celsius.\n";
        // return;           // this line is optional
}
```

- Notice: **NO** return statement is OK
  - – optional for void functions

Calling void Functions

- Calling from some other function, like main():
  - showResults(degreesF, degreesC);
  - showResults(32.5, 0.3);

- A call to a void function cannot be a right-hand-side (RHS) operand of assignment operators
  - since no value returned
  - the following statement causes a compilation error

```
int result = showResults(32.5, 0.3); // compilation error
```

More on return Statements

- Transfers control back to its calling function (caller)
- For return type other than void, a function MUST have return statement
  - typically the LAST statement in function definition
    - but not necessarily true
- return statement is optional for void functions
  - closing } would implicitly return control from a void function to its caller
- A function CAN have multiple return statements

Put Them All Together (1/2)

```
int func1(int);
// func1 needs one int argument, and returns an int

int func2(int, char); // arguments separated by comma
// func2 needs two arguments(1st: int, 2nd: char), and returns an int

int func3( );
// func3 needs NO argument, and returns an int

void func4(int);
// func4 needs one int argument, and return NOTHING
// Q: Is it OK to omit void in declaration?  A: Not OK
```

Put Them All Together (2/2)

```
int func1( );
void func2( );

int main( ) {
    int i = func1();        // ok, the return value is assigned to i
    func1( );               // ok, just discard the return value
    func2( );               // ok
    i = func2( ) ;          // error, func2 returns nothing
    // …
}
```

inline Function (Advanced)

```
inline double f2c(double f) { return (f – 32.0) * 5 / 9; }

int main() {
double ctemp1 = f2c(1.0);
double ctemp2 = f2c(2.0);
//…
double ctemp100 = f2c(100.0);
// …
}
```

- In general
  - inline function is faster
  - inline function makes executable larger
  - inline is just a hint to compiler; compiler will or will not do it

Preconditions and Postconditions

- Similar to I-P-O discussion previously
- Comment function declaration:

  void showInterest(double balance, double rate);
  //Precondition: balance is nonnegative account balance
  //   rate is interest rate as percentage
  //Postcondition: amount of interest on given balance,
  //   at given rate …
- Often called inputs and outputs

main()

- Recall: main() IS a function
- One and **ONLY** one main() will exist in a C/C++ program
- Who calls main()?
  - operating system
- Tradition holds it should have return statement
  - value returned to its caller ➔ operating system
  - should return int or void ; int in tradition

Local Names

- Local names (e.g., variables, constants)
  - declared inside a function
  - scope: available (visible) from its declaration to the end of the **block** in which its declaration occurs
- Hence, different functions can define their own variables/constants even with a same name

```
int func1() {
    double abc;
    // …
}
```

```
void func2() {
    const int abc = 10;
     // …
}
```

Global Names

- Declared inside a function
  - local name
- Declared outside all functions
  - global name
  - scope: available (visible) from its declaration to the end of the **file**
  - typically, it is declared at the beginning of the file (before function definitions)
- Global names are typical for constants:
  - e.g., const double TAXRATE = 0.05;
  - all functions in that file can use it
- Global variables
  - you can use them, but you'd better avoid using them
  - hard to understand and maintain
  - a disaster for debugging!

Blocks

- A block is a section of code delimited by **{ }**
- You can declare a name within a block
  - **name hiding** issue!

```
int func1() {
    int x = 10;
    while( x != 10) {
        int x = 1;
        // green x in scope
    }  // green x dies here
    // red x in scope
} // red x dies here
```

- A function definition itself is also a block!

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Function Basics

for Loop

- Variables CAN be declared in the initializer part of a for loop
  - scope: from their declarations to the end of the for loop

```
sum = 0;

for (int ctr = 0; ctr < 10; ++ctr) {
      sum+=ctr;
} // ctr dies here
```

Name Hiding (Advanced)

```
int x;                          // global x
void f() {
    int x;                      // local x hides global x
    x = 1;                      // assign 1 to local x
    {
        int x;                  // hides first local x
        x = 2;                  // assign 2 to second local x
        ::x = 3;                // assign 3 to global x
                                // NO WAY to access first local x in this block
    }
    cout << x;                  // the value of first local x is outputted; i.e., 1
    x = 4;                      // assign 4 to first local x
}
```

- A hidden global name can be referred to using the scope resolution operator **::**
- **No** way to access hidden local names

Local vs. Global Variables

- Suggestion: local variables are preferred
  - better maintainability
  - less errors and better for debugging
- Suggestion: minimize the use of global variables
  - hard to understand and maintain
  - a disaster for debugging!
- Suggestion: minimize name hiding
  - not good for debugging

Static Local Variables (Advanced)

```
void func(int a) {
        while(a--) {
                static int n = 0;     // gets executed ONLY once
                int x = 0;            // gets executed EVERY time
                cout << “n: ” << n++ << “, x: ” << x++ << ‘\n’;
        }
}

int main() {
        func(3); return 0;
}
```

```
Output:
n: 0, x: 0
n: 1, x: 0
n: 2, x: 0
```

- Static local variables are initialized only at the **first** time
- Non-static local variables are initialized **every** time

Procedural Abstraction

- You just need to know **what** function does, not **how** it does it!
  - do you know how rand() works?
- Function is considered a **black box**
  - you know how to use, but not it's method of operation
  - think about your 50" LCD TV at home
- Implement functions like black box
  - users of a function only need its declaration
  - does NOT need its definition
    - information hiding, abstraction, encapsulation
    - hide details of how a function gets its job done!

Summary (1/2)

- Pre-defined functions in libraries and user-defined functions
- Functions should be black boxes
  - function declarations should self-document
    - provide pre- & post-conditions in comments
    - provide all caller needs for use
  - hide “how” details
- Parameters (Arguments)
  - formal: in function declaration and definition
    - placeholder for incoming data
  - actual: in function call
    - actual data passed to function

Summary (2/2)

- Local names
  - declared within functions
- Global names
  - declared outside all functions
  - OK for constants
  - extreme cares for variables
- Name scope
  - name hiding issue
- Static local variables