

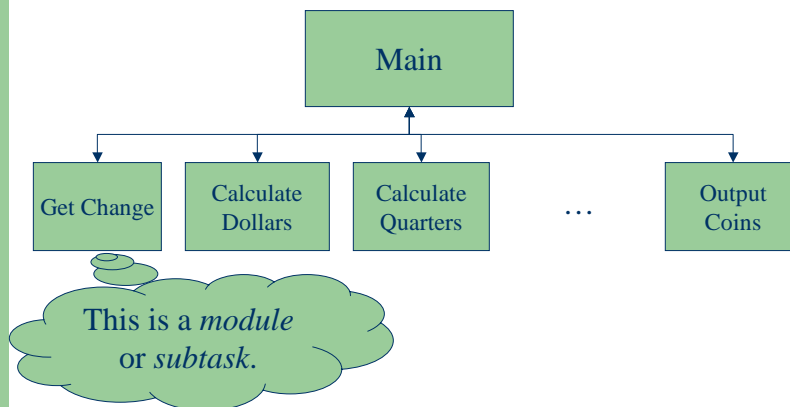
The C++ Language

Functions

Program Design

- Small Programs
 - Easily understood in a single sequence of steps
 - Little refinement
 - A single algorithm is sufficient
- Larger Programs
 - Difficult to understand and remember a long sequence of steps
 - Usually consist of several small problems requiring lots of refinement
 - Use several small algorithms

Modular Design (Structure Chart)



3

Struble - Functions

Design Principles

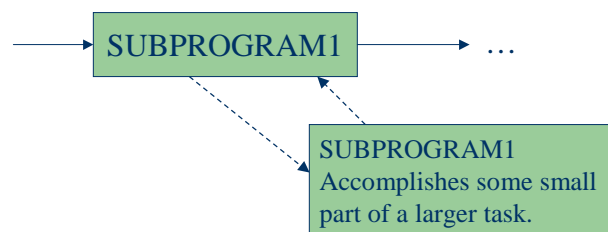
- Interface
 - Definition of communication between two modules
- Encapsulation
 - Provide functionality through well defined interfaces
- Information Hiding
 - Reduce the scope of information (variables) as much as possible.
 - Achieved through encapsulation.

4

Struble - Functions

Subprograms in C++

- C++ supports subprograms through *functions*
 - Look like small programs within a program
- Flow



5

Struble - Functions

Functions

- Three parts to a function
 - *Function call or invocation*, tells the program when to use a function
 - *Function prototype*, describes the *interface* to the function
 - *Function definition*, describes what statements the function executes

6

Struble - Functions

Function Call

- We've actually seen this several times
- Syntax

```
FunctionIdentifier ( Argument1 , Argument2 ... )
```

- Arguments are expressions
 - May not have any arguments
 - Also called *actual arguments* or *actual parameters*

7

Struble - Functions

Function Call

- Examples

```
In.ignore(INT_MAX, '\n');  
In.get(c);  
getline(In, aLine);  
double root2 = sqrt(2); // need #include <cmath>  
double twoCubed = pow(2, 3); // need #include <cmath>
```

- Exercise: Identify the arguments for each invocation.

8

Struble - Functions

Invoker and Invokee

- An *invoker* is the function that contains a function invocation
- An *invokee* is the function that named by the function invocation

```
void main() {  
    cin.ignore(INT_MAX, '\n'); // ignore a line  
    ...  
}
```

main is the invoker

cin.ignore is the invokee.

9

Struble - Functions

Function Prototype

- Describes the interface for communication with the function
 - Also known as *function declaration*
- Syntax

```
DataType FunctionName ( DataType & VariableName , DataType & VariableName ... ) ;
```

- Variable names describe *parameters*
 - Also called *formal parameters* or *formal arguments*

10

Struble - Functions

Parameters

- Similar to variable declarations inside the parentheses
- Store/refer to the values of the arguments during a function invocation
- Communication mechanism between *invoker* and *function*

11

Struble - Functions

Function Prototypes

- Examples

```
void IgnoreLines(istream& In, int numLines);  
int GetChange(istream& In);  
double pow(double x, double y);
```

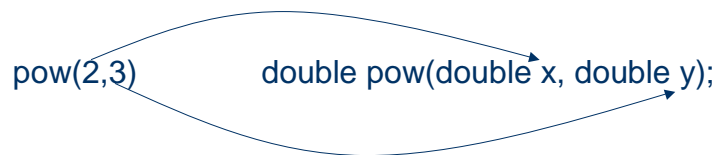
- Exercise: Identify the parameters for each function.

12

Struble - Functions

Relationship Between Invocation and Prototype

- The values (or variables as we'll see) of the actual parameters in an invocation are matched up with the formal parameters
 - Matched up by order
 - Types must be compatible
- Example



13

Struble - Functions

Exercise

- The first step to using functions is to identify several modules.
 1. Identify several modules for program 5.
 2. Draw a structure chart describing the module relationships.

14

Struble - Functions

Function Definitions

- A *function definition* describes the steps to take when the function is invoked.
- Syntax

```
DataType FunctionName(Parameters) → Function Header
{
    Statement1 ;
    Statement2 ; → Function Body
    ...
}
```

15

Struble - Functions

Function Definitions

- The *function header* must match the function prototype with the same name
- The *function body* contains statements to execute when the function is invoked
 - Looks just like a main program

16

Struble - Functions

You must include function documentation for each function in your program.

Example (Function Documentation)

```
// IgnoreLines() ignores the specified number of lines
// in an input stream.
//
// Parameters: In      - input stream
//             numLines - the number of lines to ignore
// Returns:   nothing
// Calls:    none
// Called by: main()
// Pre:     In is an open input stream without errors
//          with the read marker at the start of a line
//          and there are at least numLines lines in the
//          stream
// Post:    numLines lines are skipped and the read marker
//          is placed at the start of a line
```

17

Struble - Functions

Example (Function Definition)

```
void IgnoreLines(istream& In, int numLines)
{
    int skipped = 0;
    for (skipped = 0; skipped < numLines; skipped++)
    {
        In.ignore(INT_MAX, '\n');
    }
}
```

18

Struble - Functions

Function Usage (Local Prototype Scope)

```
#include <iostream>
using namespace std;
void main()
{
    void IgnoreLines(istream& In, int numLines);

    IgnoreLines(cin, 3); // ignore the first 3 lines
    ...
}

// IgnoreLines() ...
void IgnoreLines(istream& In, int numLines)
{
    ...
}
```

Must have prototype before invocation.

Definition follows main program.

19

Struble - Functions

Function Communication

- C++ provides several techniques for communicating information between invoker and function
 - Input only
 - Pass by value
 - Pass by constant reference
 - Output only
 - Return values
 - Input and output
 - Pass by reference

20

Struble - Functions

Pass by value

- Used when you are passing simple types as input only
- Simplest communication mechanism
- Copies values of arguments into the parameters

21

Struble - Functions

Pass by Value Example

```
void main()
{
    void PrintStars(int numStars);

    PrintStars(10);
}

void PrintStars(int numStars)
{
    int printed = 0;
    for (printed = 0; printed < numStars; printed++)
    {
        cout << '*';
    }
}
```

10 is copied into numStars during invocation.

22

Struble - Functions

Pass by Value Example

```
void main()
{
    void PrintStars(int numStars);
    int stars;
    cin >> stars;
    PrintStars(stars);
}

void PrintStars(int numStars)
{
    for (; numStars > 0; numStars--)
    {
        cout << '*';
    }
}
```

Value stored in stars
is **copied**.

numStars contains
copy of the value in
stars. The variable
stars is **not**
modified!

23

Struble - Functions

Value Returning Functions

- Allows functions to *return* values as a result of invocation
 - The value returned replaces the function invocation in an expression.
- This is an output only communication method
 - Use when you are creating a new value
- The `return` statement determines the value for the function invocation.

24

Struble - Functions

Example

```
void main() {
    int Minimum(int x, int y);
    int a, b, minVal;
    cin >> a >> b;
    minVal = Minimum(a,b);
    cout << "The minimum value is " << minVal << endl;
}

int Minimum(int x, int y) {
    if (x < y) {
        return x;
    } else {
        return y;
    }
}
```

25

Struble - Functions

Return Statement

- The `return` statement determines the value of the function
 - Can appear in several places
 - Stops execution of function and returns to invoker
- Syntax

```
return Expression ;
```

- Expression type must match function type.
 - Expression is not used for `void` functions

26

Struble - Functions

Pass By Reference

- Input and output mechanism
- Use when you need to modify the parameters
 - Modification must be reflected outside of the function
- The ampersand (&) following the datatype denotes pass by reference

27

Struble - Functions

Pass By Reference Example

```
void main() {  
    int a, b;  
    cin >> a >> b;  
    cout << "a is " << a << ", b is " << b << endl;  
    Swap(a, b);  
    cout << "a is " << a << ", b is " << b << endl;  
}
```

```
void Swap(int& x, int& y) {  
    int temp;  
    temp = x;  
    x = y;  
    y = temp;  
}
```

Notice the ampersand.

Changes to formal parameters also change actual parameters.

28

Struble - Functions

Passing Streams

- Input and output streams **MUST** be passed by reference
- When you read from or write to a stream, you are changing its contents
 - Moving the read marker
 - Making the stream contain more output
- Recall that pass by value makes a copy
 - Copies don't make sense!

29

Struble - Functions

Passing Streams (Examples)

```
void ReadDataSet(istream& In, string& name, int& age) {
    const char DELIMITER = '|';
    getline(In, name, DELIMITER);
    In >> age;
}

void OutputDataSet(ostream& Out, string name, int age) {
    Out >> name >> " is " >> age >> " years old."
    >> endl;
}
```

30

Struble - Functions

Pass By Constant Reference

- Input only parameter passing
 - Use to pass complex data types
 - More efficient than copying all data
 - Compiler guarantees no changes are made
- Use `const` keyword before the parameter type and `&` after

31

Struble - Functions

Pass By Constant Reference (Example)

- Old

```
void OutputDataSet(ostream& Out, string name, int age) {
    Out >> name >> " is " >> age >> " years old."
    >> endl;
}
```
- New

```
void OutputDataSet(ostream& Out, const string& name,
    int age)
{
    Out >> name >> " is " >> age >> " years old."
    >> endl;
}
```

32

Struble - Functions

Global Variables

- Variables in global scope
 - Accessible in functions following declaration
 - Violates principles of information hiding and encapsulation
 - **DO NOT USE!** (although you will be tested on it)

33

Struble - Functions

Global Variables Example

```
int numNames = 0;

void main() {
    bool ReadDataSet(istream& In, string& name);
    ifstream In("Input.txt");
    string name;
    while (ReadDataSet(In, name)) {
        cout << "Name read is " << name << endl;
    }
    cout << numNames << " were read." << endl;
}
(continued on next slide...)
```

34

Struble - Functions

Global Variables Example

```
// Note, returns whether or not reading was
// successful.
bool ReadDataSet(istream& In, string& name) {
    In >> name;
    if (In) {
        numNames++;
        return true;
    }
    return false;
}
```

35

Struble - Functions

Scope and Functions

- Constants and function prototypes are OK to put in global scope
 - Constants are useful to share and cannot change
 - Prototypes are often easier to manage in global scope even though technically a violation of information hiding.
- See Notepack Chapter 8, slides 23-28

36

Struble - Functions

Expressions as Actual Parameters

- Recall that an expression can be used for actual parameters
- Examples

```
minVal = Minimum(a + 2, b - 3);  
minVal = Minimum( Minimum(a, b), c );
```

- This works for *pass by value*, but does **NOT** work for *pass by reference* or *pass by constant reference*
 - Also cannot pass literal constants by reference!

37

Struble - Functions

Expressions As Actual Parameters

- So, these invocations are errors, but why?

```
void Swap(int& x, int& y);  
  
void main() {  
    int a, b;  
    cin >> a >> b;  
  
    Swap(a + 2, b);  
    Swap(b, 2);  
}  
  
void Swap(int& x, int& y) {  
    int temp;  
    temp = x;  
    x = y;  
    y = temp;  
}
```

38

Struble - Functions

Exercise

- What's output by the following program?

```
void DoIt(int a, int& b);
void main() {
    int Tmp = 15;
    int Ben = -5, Jer = 42;
    DoIt(Ben, Jer);
    cout << "Ben = " << Ben << endl;
    cout << "Jer = " << Jer << endl;
    cout << "DoIt Tmp = " << Tmp
        << endl;
}

void DoIt(int a, int& b) {
    int Tmp;
    a = a + 100;
    tmp = b;
    b = 999;
}
```

39

Struble - Functions

Exercise

- Which are valid function invocations?

```
void Fix(double &realVar, int intVar);
int someInt = 42;
double someFloat = 3.14;

Fix(6.85, 24);
Fix(someFloat, 24);
Fix(someFloat, someInt);
Fix(someFloat, someInt + 5);
Fix(someFloat, 25.3);
Fix(someInt, someFloat);
```

Try this at home.

40

Struble - Functions

Exercise

- Rewrite the program on slides 34 and 35 so that you do not use a global variable.
 - What additional information do you need to pass?
 - What parameter passing technique do you need to use and why?