IS 0020
Program Design and Software Tools
Introduction to C++ Programming

Review slides
Oct 5, 2004
Review topics

1. Lvalue/rvalue
2. Reference variable + Calling functions by reference
3. Passing arrays to functions
4. Function pointers
5. Arrays of pointers to functions (e.g. menus)
6. Using member selection operators ('.' '->')
7. Returning a reference to a private data member
8. *this pointer
   1. implicit & explicit use of *this pointer
   2. *this pointer & cascaded function calls
9. When are destructors used, why not yet in the classes we're creating?
10. When is it appropriate to use 'new' and 'delete' from the <new> library?
Confusing Equality (==) and Assignment (=) Operators

• **Lvalues**
  – Expressions that can appear on left side of equation
  – Can be changed
    \[ x = 4; \]

• **Rvalues**
  – Only appear on right side of equation
  – Constants, such as numbers (i.e. cannot write \( 4 = x; \))

• **Lvalues** can be used as **rvalues**, but not vice versa
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

• Reference parameter
  – Alias for argument in function call
    • Passes parameter by reference
  – Use & after data type in prototype
    • `void myFunction( int &data )`
    • Read “`data` is a reference to an `int`”
  – Function call format the same
    • However, original can now be changed
References and Reference Parameters

• Pointers
  – Another way to pass-by-reference

• References as aliases to other variables
  – Refer to same variable
  – Can be used within a function

        int count = 1;       // declare integer variable count
        int &cRef = count;   // create cRef as an alias for count
        ++cRef;             // increment count (using its alias)

• References must be initialized when declared
  – Otherwise, compiler error
  – Dangling reference
    • Reference to undefined variable
Passing Arrays to Functions

• Specify name without brackets
  – To pass array `myArray` to `myFunction`
    ```
    int myArray[ 24 ];
    myFunction( myArray, 24 );
    ```
  – Array size usually passed, but not required
    • Useful to iterate over all elements

• Arrays passed-by-reference
  – Functions can modify original array data
  – Value of name of array is address of first element
    • Function knows where the array is stored
    • Can change original memory locations
Passing Arrays to Functions

- Functions taking arrays
  - Function prototype
    - `void modifyArray( int b[], int arraySize );`
    - `void modifyArray( int [], int );`
      - Names optional in prototype
    - Both take an integer array and a single integer
  - No need for array size between brackets
    - Ignored by compiler
  - If declare array parameter as `const`
    - Cannot be modified (compiler error)
    - `void doNotModify( const int [] );`
Using `const` with Pointers

- **`const` qualifier**
  - Value of variable should not be modified
  - `const` used when function does not need to change a variable
  - Principle of least privilege

- **`const` pointers**
  - Always point to same memory location
  - Default for array name
  - Must be initialized when declared

- **Four ways to pass pointer to function**
  - Nonconstant pointer to nonconstant data
    - Highest amount of access
  - Nonconstant pointer to constant data
  - Constant pointer to nonconstant data
  - Constant pointer to constant data
    - Least amount of access
// Fig. 5.13: fig05_13.cpp
// Attempting to modify a constant pointer to
// non-constant data.

int main()
{
    int x, y;

    // ptr is a constant pointer to an int
    // be modified through ptr, but ptr always points to
    // same memory location.
    int * const ptr = &x;

    *ptr = 7;    // allowed: *ptr is not const
    ptr = &y;    // error: ptr is const; cannot assign new address

    return 0;    // indicates successful termination
} // end main

d:\cpphtp4_examples\ch05\Fig05_13.cpp(15): error C2166:
    l-value specifies const object

ptr is constant pointer to
Can modify x (pointed to by ptr) to new address since ptr is constant.

Cannot modify ptr to point to new address since ptr is constant.

Line 15 generates compiler error by attempting to assign new address to constant pointer.
// Fig. 5.14: fig05_14.cpp
// Attempting to modify a constant pointer to constant data.
#include <iostream>

using std::cout;
using std::endl;

int main()
{
    int x = 5, y;

    // ptr is a constant pointer to a constant integer.
    // ptr always points to the same location; the integer
    // at that location cannot be modified.
    const int *const ptr = &x;

    cout << *ptr << endl;

    // error: *ptr is const; cannot assign new value
    *ptr = 7;

    // error: ptr is const; cannot assign new address
    ptr = &y;

    return 0;  // indicates successful termination
} // end main
Relationship Between Pointers and Arrays

• Arrays and pointers closely related
  – Array name like constant pointer
  – Pointers can do array subscripting operations

• Accessing array elements with pointers
  – Element $b[n]$ can be accessed by $*(bPtr + n)$
    • Called pointer/offset notation
  – Addresses
    • $&b[3]$ same as $bPtr + 3$
  – Array name can be treated as pointer
    • $b[3]$ same as $*(b + 3)$
  – Pointers can be subscripted (pointer/subscript notation)
    • $bPtr[3]$ same as $b[3]$
Function Pointers

- Calling functions using pointers
  - Assume parameter:
    - `bool ( *compare ) ( int, int )`
  - Execute function with either
    - `( *compare ) ( int1, int2 )`
      - Dereference pointer to function to execute
    OR
    - `compare( int1, int2 )`
      - Could be confusing
        - User may think `compare` name of actual function in program
// Fig. 5.25: fig05_25.cpp
// Multipurpose sorting program using function pointers.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include <iomanip>

using std::setw;

// prototypes
void bubble( int [], const int, bool (*)( int, int ) );
void swap( int * const, int * const );
bool ascending( int, int );
bool descending( int, int );

int main()
{
    const int arraySize = 10;
    int order;
    int counter;
    int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

cout << "Enter 1 to sort in ascending order, \n"
    << "Enter 2 to sort in descending order: ";
cin >> order;
cout << "Data items in original order\n";

// output original array
for ( counter = 0; counter < arraySize; counter++ )
    cout << setw( 4 ) << a[ counter ];

// sort array in ascending order; pass function ascending
// as an argument to specify ascending sorting order
if ( order == 1 ) {
    bubble( a, arraySize, ascending );
    cout << "Data items in ascending order\n";
}

// sort array in descending order; pass function descending
// as an argument to specify descending sorting order
else {
    bubble( a, arraySize, descending );
    cout << "Data items in descending order\n";
}
// output sorted array
for ( counter = 0; counter < arraySize; counter++ )
    cout << setw( 4 ) << a[ counter ];

cout << endl;

return 0; // indicates successful termination

} // end main

// multipurpose bubble sort; parameter compare is a pointer to
// the comparison function that determines sorting order
void bubble( int work[], const int size,
             bool (*compare)( int, int ) )
{
    // loop to control passes
    for ( int pass = 1; pass < size; pass++ )
    {
        // loop to control number of comparisons per pass
        for ( int count = 0; count < size - 1; count++ )
            // if adjacent elements are out of order, swap them
            if ( (*compare)( work[ count ], work[ count + 1 ] ) )
                swap( &work[ count ], &work[ count + 1 ] );

// compare is pointer to function that receives two integer parameters and returns bool result.

// Parentheses necessary to indicate pointer to function

// Call passed function compare; dereference pointer to execute function.
} // end function bubble

// swap values at memory locations to which // element1Ptr and element2Ptr point
void swap( int * const element1Ptr, int * const element2Ptr )
{
    int hold = *element1Ptr;
    *element1Ptr = *element2Ptr;
    *element2Ptr = hold;
}

} // end function swap

// determine whether elements are out of order // for an ascending order sort
bool ascending( int a, int b )
{
    return b < a;   // swap if b is less than a
}

} // end function ascending
94    // determine whether elements are out of order
95    // for a descending order sort
96    bool descending( int a, int b )
97    {
98        return b > a;  // swap if b is greater than a
99    }  // end function descending

Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in original order
  2   6   4   8  10  12  89  68  45  37
Data items in ascending order
  2   4   6   8  10  12  37  45  68  89

Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

Data items in original order
  2   6   4   8  10  12  89  68  45  37
Data items in descending order
  89  68  45  37  12  10   8   6   4   2
Function Pointers

- Arrays of pointers to functions
  - Menu-driven systems
  - Pointers to each function stored in array of pointers to functions
    - All functions must have same return type and same parameter types
  - Menu choice $\rightarrow$ subscript into array of function pointers
// Fig. 5.26: fig05_26.cpp
// Demonstrating an array of pointers to functions.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

// function prototypes
void function1( int );
void function2( int );
void function3( int );

int main()
{
    // initialize array of 3 pointers to functions that each take an int argument and return void
    void (*f[ 3 ])( int ) = { function1, function2, function3 };

    int choice;

    cout << "Enter a number between 0 and 2, 3 to end: ";
    cin >> choice;
}
// process user's choice
while ( choice >= 0 && choice < 3 ) {

    // invoke function at location choice in array f
    // and pass choice as an argument
    (*f[ choice ])( choice );

    cout << "Enter a number between 0 and 2, 3 to end: ";
    cin >> choice;
}

cout << "Program execution completed." << endl;
return 0;  // indicates successful termination
}

void function1( int a )
{
    cout << "You entered " << a << " so function1 was called\n\n";
}

// Call chosen function by dereferencing corresponding element in array.
void function2( int b )
{
    cout << "You entered " << b
        << " so function2 was called\n\n";
}

void function3( int c )
{
    cout << "You entered " << c
        << " so function3 was called\n\n";
}

Enter a number between 0 and 2, 3 to end: 0
You entered 0 so function1 was called

Enter a number between 0 and 2, 3 to end: 1
You entered 1 so function2 was called

Enter a number between 0 and 2, 3 to end: 2
You entered 2 so function3 was called

Enter a number between 0 and 2, 3 to end: 3
Program execution completed.
Accessing Structure Members

- **Member access operators**
  - Dot operator ( . ) for structure and class members
  - Arrow operator ( \( \rightarrow \) ) for structure and class members via pointer to object
  - Print member `hour` of `timeObject`:
    
    ```cpp
    cout << timeObject.hour;
    ```
    
    OR
    
    ```cpp
    timePtr = &timeObject;
    cout << timePtr->hour;
    ```
    
    - `timePtr->hour` same as \( ( \ast timePtr ).hour \)
  
- Parentheses required
  - * lower precedence than .
Subtle Trap: Returning a Reference to a private Data Member

• Reference to object
  – &pRef = p;
  – Alias for name of object
  – Lvalue
    • Can receive value in assignment statement
      – Changes original object

• Returning references
  – public member functions can return non-const references to private data members
    • Client able to modify private data members
// Fig. 6.21: time4.h
// Declaration of class Time.
// Member functions defined in time4.cpp

// prevent multiple inclusions of header file
#ifndef TIME4_H
#define TIME4_H

class Time {

public:
    Time( int = 0, int = 0, int = 0 );
    void setTime( int, int, int );
    int getHour();

    int &badSetHour( int );  // DANGEROUS reference return

private:
    int hour;
    int minute;
    int second;

}; // end class Time

#endif
// return hour value
int Time::getHour()
{
    return hour;
}

// POOR PROGRAMMING PRACTICE:
// Returning a reference to a private data member.
int &Time::badSetHour( int hh )
{
    hour = ( hh >= 0 && hh < 24 ) ? hh : 0;
    return hour;  // DANGEROUS reference return

} // end function badSetHour

Return reference to private data member hour.
// Fig. 6.23: fig06_23.cpp
// Demonstrating a public member function that
// returns a reference to a private data member.
#include <iostream>

using std::cout;
using std::endl;

// include definition of class Time from time4.h
#include "time4.h"

int main()
{
    Time t;

    // store in hourRef the reference returned by badSetHour
    int &hourRef = t.badSetHour( 20 );

    cout << "Hour before modification: " << hourRef;

    // use hourRef to set invalid value in Time object t
    hourRef = 30;

    cout << "\nHour after modification: " << t.getHour();
}

badSetHour returns reference to private data member hour.
Reference allows setting of private data member hour.
// Dangerous: Function call that returns // a reference can be used as an lvalue!

t.badSetHour(12) = 74;

cout << "\n\n***************************\n" << "POOR PROGRAMMING PRACTICE!!!!!!!\n" << "badSetHour as an lvalue, Hour: \n" << t.getHour() << "\n***************************" << endl;

return 0;

} // end main

Hour before modification: 20
Hour after modification: 30

***************************
POOR PROGRAMMING PRACTICE!!!!!!!
badSetHour as an lvalue, Hour: 74
***************************

Can use function call as lvalue to set invalid value.

Returning reference allowed invalid setting of *private* data member *hour*. 
Default Memberwise Assignment

• Assigning objects
  – Assignment operator (=)
    • Can assign one object to another of same type
    • Default: memberwise assignment
      – Each right member assigned individually to left member

• Passing, returning objects
  – Objects passed as function arguments
  – Objects returned from functions
  – Default: pass-by-value
    • Copy of object passed, returned
      – Copy constructor
        • Copy original values into new object
Dynamic Memory Management with Operators

`new` and `delete`

- Dynamic memory management
  - Control allocation and deallocation of memory
  - Operators `new` and `delete`
    - Include standard header `<new>`

- `new`
  ```
  Time *timePtr;
  timePtr = new Time;
  ```
  - Creates object of proper size for type `Time`
    - Error if no space in memory for object
  - Calls default constructor for object
  - Returns pointer of specified type
  - Providing initializers
    ```
    double *ptr = new double(3.14159);
    Time *timePtr = new Time(12, 0, 0);
    ```
  - Allocating arrays
    ```
    int *gradesArray = new int[10];
    ```
Dynamic Memory Management with Operators

new and delete

• **delete**
  - Destroy dynamically allocated object and free space
  - Consider
    ```
    delete timePtr;
    ```
  - Operator **delete**
    • Calls destructor for object
    • Deallocates memory associated with object
      - Memory can be reused to allocate other objects
  - Deallocating arrays
    ```
    delete [] gradesArray;
    ```
    - Deallocates array to which `gradesArray` points
    • If pointer to array of objects
      • First calls destructor for each object in array
      • Then deallocates memory
**static Class Members**

- **static class variable**
  - “Class-wide” data
    - Property of class, not specific object of class
  - Efficient when single copy of data is enough
    - Only the **static** variable has to be updated
  - May seem like global variables, but have class scope
    - Only accessible to objects of same class
  - Initialized exactly once at file scope
  - Exist even if no objects of class exist
  - Can be **public**, **private** or **protected**
static Class Members

• Accessing static class variables
  – Accessible through any object of class
  – public static variables
    • Can also be accessed using binary scope resolution operator(::)
      Employee::count
  – private static variables
    • When no class member objects exist: Can only be accessed via
      public static member function
      Employee::getCount()

• static member functions
  – Cannot access non-static data or functions
  – No this pointer for static functions
    • static data members and static member functions exist
      independent of objects
// Fig. 7.17: employee2.h
// Employee class definition.
#ifndef EMPLOYEE2_H
#define EMPLOYEE2_H

class Employee {

public:

    Employee( const char *, const char * ); // constructor
    ~Employee(); // destructor
    const char *getFirstName() const; // return first name
    const char *getLastName() const; // return last name

    // static member function
    static int getCount(); // return # objects instantiated

private:

    char *firstName;
    char *lastName;

    // static data member
    static int count; // number of objects instantiated

}; // end class Employee

static member function can only access static data members and member functions.
static data member is class-wide data.
// Fig. 7.18: employee2.cpp
// Member-function definitions for class Employee.
#include <iostream>

using std::cout;
using std::endl;

#include <new> // C++ standard new operator
#include <cstring> // strcpy and strlen prototypes

#include "employee2.h" // Employee class definition

// define and initialize static data member
int Employee::count = 0;

// define static member function that returns number of
// Employee objects instantiated
int Employee::getCount()
{
    return count;
}

} // end static function getCount
23 // constructor dynamically allocates space for
24 // first and last name and uses strcpy to copy
25 // first and last names into the object
26 Employee::Employee( const char *first, const char *last )
27 {
28     firstName = new char[ strlen( first ) + 1 ];
29     strcpy( firstName, first );
30
31     lastName = new char[ strlen( last ) + 1 ];
32     strcpy( lastName, last );
33
34     ++count;  // increment static count of employees
35
36     cout << "Employee constructor for " << firstName
37         << ' ' << lastName << " called." << endl;
38 }
39 // end Employee constructor
40
41 // destructor deallocates dynamically allocated memory
42 Employee::~Employee()
43 {
44     cout << "~Employee() called for " << firstName
45         << ' ' << lastName << endl;
delete [] firstName; // recapture memory
delete [] lastName;    // recapture memory
--count;  // decrement static count of employees

// return first name of employee
const char *Employee::getFirstName() const
{
    // const before return type prevents client from modifying
    // private data; client should copy returned string before
    // destructor deletes storage to prevent undefined pointer
    return firstName;
}

// return last name of employee
const char *Employee::getLastName() const
{
    // const before return type prevents client from modifying
    // private data; client should copy returned string before
    // destructor deletes storage to prevent undefined pointer
    return lastName;
}

© 2003 Prentice Hall, Inc.  
All rights reserved.
Operator Functions As Class Members Vs. As Friend Functions

- **Operator functions**
  - Member functions
    - Use `this` keyword to implicitly get argument
    - Gets left operand for binary operators (like `+`)
    - Leftmost object must be of same class as operator
  - Non member functions
    - Need parameters for both operands
    - Can have object of different class than operator
    - Must be a `friend` to access private or protected data

- **Example Overloaded `<<` operator**
  - Left operand of type `ostream &`
    - Such as `cout` object in `cout << classObject`
  - Similarly, overloaded `>>` needs `istream &`
  - Thus, both must be non-member functions
Operator Functions As Class Members Vs. As Friend Functions

• Commutative operators
  – May want $+$ to be commutative
    • So both “$a + b$” and “$b + a$” work
  – Suppose we have two different classes
  – Overloaded operator can only be member function when its class is on left
    • $\text{HugeIntClass} + \text{Long int}$
    • Can be member function
  – When other way, need a non-member overload function
    • $\text{Long int} + \text{HugeIntClass}$
Overloading Stream-Insertion and Stream-Extraction Operators

• `<<` and `>>`
  – Already overloaded to process each built-in type
  – Can also process a user-defined class

• Example program
  – Class `PhoneNumber`
    • Holds a telephone number
  – Print out formatted number automatically
    • `(123) 456–7890`
// Fig. 8.3: fig08_03.cpp
// Overloading the stream-insertion and
// stream-extraction operators.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;
using std::ostream;
using std::istream;

#include <iomanip>

using std::setw;

// PhoneNumber class definition
class PhoneNumber {
friend ostream &operator<<( ostream&, const PhoneNumber & );
friend istream &operator>>( istream&, PhoneNumber & );

private:
char areaCode[ 4 ]; // 3-digit area code and null
char exchange[ 4 ]; // 3-digit exchange and null
char line[ 5 ]; // 4-digit line and null

}; // end class PhoneNumber

// Notice function prototypes for overloaded operators >> and <<
// They must be non-member friend functions, since the object of class
// PhoneNumber appears on the right of the operator.

cin << object
cout >> object
// overloaded stream-insertion operator; cannot be
// a member function if we would like to invoke it with
// cout << somePhoneNumber;
ostream &operator<<( ostream &output, const PhoneNumber &num )
{
    output << "(" << num.areaCode << ""
    << num.exchange << "-" << num.line;
    return output;  // enables cout << a << b << c;
}

// overloaded stream-extraction operator; cannot be
// a member function if we would like to invoke it with
// cin >> somePhoneNumber;
istream &operator>>( istream &input, PhoneNumber &num )
{
    input.ignore();                     // skip ( 
    input >> setw( 4 ) >> num.areaCode; // input area code
    input.ignore( 2 );                  // skip ) and space
    input >> setw( 4 ) >> num.exchange; // input exchange
    input.ignore();                    // skip dash (-)
    input >> setw( 5 ) >> num.line;     // input line
    return input;      // enables cin >> a >> b >> c;
}

The expression:
cout << phone;
is interpreted as the function call:
operator<<(cout, phone);

output is an alias for cout.

This allows objects to be cascaded.
cout << phone1 << phone2;
first calls
operator<<(cout, phone1), and
returns cout.

Next, cout << phone2 executes.

Stream manipulator setw restricts number of characters read. setw(4) allows 3
characters to be read, leaving room for the null character.
53
54 } // end function operator>>
55
56 int main()
57 {
58     PhoneNumber phone; // create object phone
59
60     cout << "Enter phone number in the form (123) 456-7890:\n";
61
62     // cin >> phone invokes operator>> by implicitly issuing
63     // the non-member function call operator>>( cin, phone )
64     cin >> phone;
65
66     cout << "The phone number entered was: " ;
67
68     // cout << phone invokes operator<< by implicitly issuing
69     // the non-member function call operator<<( cout, phone )
70     cout << phone << endl;
71
72     return 0;
73
74 } // end main

Enter phone number in the form (123) 456-7890:
(800) 555-1212
The phone number entered was: (800) 555-1212