Lecture 18: Pointers & Classes

Ioan Raicu
Department of Electrical Engineering & Computer Science
Northwestern University

EECS 211
Fundamentals of Computer Programming II
April 27th, 2010
There are four ways to pass a pointer to a function:

- A nonconstant pointer to nonconstant data
- A nonconstant pointer to constant data (Fig. 8.10)
- A constant pointer to nonconstant data (Fig. 8.11)
- A constant pointer to constant data (Fig. 8.12)

Each combination provides a different level of access privilege.
• A nonconstant pointer to constant data
  – A pointer that can be modified to point to any data item of the appropriate type, but the data to which it points cannot be modified through that pointer.
• Might be used to receive an array argument to a function that will process each array element, but should not be allowed to modify the data.
• Any attempt to modify the data in the function results in a compilation error.
• Sample declaration:
  – \texttt{const int *countPtr;}
  – Read from right to left as “\texttt{countPtr} is a pointer to an integer constant.”
• Figure 8.10 demonstrates the compilation error messages produced when attempting to compile a function that receives a nonconstant pointer to constant data, then tries to use that pointer to modify the data.
8.5 Using const with Pointers (cont.)

// Fig. 8.10: fig08_10.cpp
// Attempting to modify data through a nonconstant pointer to constant data.

void f( const int * ); // prototype

int main()
{
    int y;

    f( &y ); // f attempts illegal modification
} // end main

// xPtr cannot modify the value of constant variable to which it points
void f(  const int *xPtr )
{
    *xPtr = 100; // error: cannot modify a const object
} // end function f

Microsoft Visual C++ compiler error message:

c:\cpphtp7_examples\ch08\Fig08_10\fig08_10.cpp(17) :
    error C3892: 'xPtr' : you cannot assign to a variable that is const

Fig. 8.10 | Attempting to modify data through a nonconstant pointer to constant data. (Part I of 2.)
8.5 Using const with Pointers (cont.)

GNU C++ compiler error message:

fig08_10.cpp: In function `void f(const int*)':
fig08_10.cpp:17: error: assignment of read-only location

Fig. 8.10  |  Attempting to modify data through a nonconstant pointer to constant data. (Part 2 of 2.)
• A constant pointer to nonconstant data is a pointer that always points to the same memory location; the data at that location can be modified through the pointer.
• An example of such a pointer is an array name, which is a constant pointer to the beginning of the array.
• All data in the array can be accessed and changed by using the array name and array subscripting.
• A constant pointer to nonconstant data can be used to receive an array as an argument to a function that accesses array elements using array subscript notation.
• Pointers that are declared `const` must be initialized when they’re declared.
• If the pointer is a function parameter, it’s initialized with a pointer that’s passed to the function.
8.5 Using const with Pointers (cont.)

**Common Programming Error 8.6**

*Not initializing a pointer that is declared `const` is a compilation error.*
8.5 Using const with Pointers (cont.)

```c
// Fig. 8.11: fig08_11.cpp
// Attempting to modify a constant pointer to nonconstant data.

int main()
{
    int x, y;

    int * const ptr = &x; // const pointer must be initialized

    *ptr = 7; // allowed: *ptr is not const
    ptr = &y; // error: ptr is const; cannot assign to it a new address
}
```

Microsoft Visual C++ compiler error message:

c:\cpphtp7_examples\ch08\Fig08_11\fig08_11.cpp(14) : error C3892: 'ptr' : you cannot assign to a variable that is const

GNU C++ compiler error message:

Fig. 8.11 | Attempting to modify a constant pointer to nonconstant data. (Part 1 of 2.)
Fig. 8.11 | Attempting to modify a constant pointer to nonconstant data. (Part 2 of 2.)
The minimum access privilege is granted by a constant pointer to constant data.

- Such a pointer always points to the same memory location, and the data at that location cannot be modified via the pointer.
- This is how an array should be passed to a function that only reads the array, using array subscript notation, and does not modify the array.

The program of Fig. 8.12 declares pointer variable `ptr` to be of type `const int * const` (line 13).

This declaration is read from right to left as “`ptr` is a constant pointer to an integer constant.”

The figure shows the error messages generated when an attempt is made to modify the data to which `ptr` points and when an attempt is made to modify the address stored in the pointer variable.
8.5 Using const with Pointers (cont.)

```cpp
// Fig. 8.12: fig08_12.cpp
// Attempting to modify a constant pointer to constant data.
#include <iostream>
using namespace std;

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;

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

Microsoft Visual C++ compiler error message:

Fig. 8.12  |  Attempting to modify a constant pointer to constant data. (Part 1 of 2.)
8.5 Using const with Pointers (cont.)

```
c:\cpphtp7_examples\ch08\Fig08_12\fig08_12.cpp(17) : error C3892: 'ptr' : you cannot assign to a variable that is const
```

**GNU C++ compiler error message:**

```
fig08_12.cpp: In function `int main()':
fig08_12.cpp:17: error: assignment of read-only location
fig08_12.cpp:18: error: assignment of read-only variable `ptr'
```

**Fig. 8.12** Attempting to modify a constant pointer to constant data. (Part 2 of 2.)
• A `void *` pointer cannot be dereferenced.
  – The compiler must know the data type to determine the number of bytes to be dereferenced for a particular pointer—for a pointer to `void`, this number of bytes cannot be determined from the type.
Pointers can be compared using equality and relational operators.

- Comparisons using relational operators are meaningless unless the pointers point to members of the same array.
- Pointer comparisons compare the addresses stored in the pointers.

A common use of pointer comparison is determining whether a pointer is 0 (i.e., the pointer is a null pointer—it does not point to anything).
• An array name can be thought of as a constant pointer.
• Pointers can be used to do any operation involving array subscripting.
• Assume the following declarations:
  • `int b[5]; // create 5-element int array b`
  • `int *bPtr; // create int pointer bPtr`
• Because the array name (without a subscript) is a (constant) pointer to the first element of the array, we can set `bPtr` to the address of the first element in array `b` with the statement
  • `bPtr = b; // assign address of array b to bPtr`
• equivalent to
  • `bPtr = &b[0]; // also assigns address of array b to bPtr`
• Array element \( b[3] \) can alternatively be referenced with the pointer expression
  \*\((bPtr + 3)\)

• The 3 in the preceding expression is the offset to the pointer.

• This notation is re-ferred to as pointer/offset notation.
  – The parentheses are necessary, because the precedence of \* is higher than that of +.
• Just as the array element can be referenced with a pointer expression, the address
  • \&b[ 3 ]
• can be written with the pointer expression
  • bPtr + 3
• The array name (which is implicitly \texttt{const}) can be treated as a pointer and used in pointer arithmetic.
• For example, the expression
  • \*( b + 3 )
• also refers to the array element \texttt{b[ 3 ]}.
• In general, all subscripted array expressions can be written with a pointer and an offset.
• Pointers can be subscripted exactly as arrays can.

• For example, the expression
  • \( bPtr[1] \)

• refers to the array element \( b[1] \); this expression uses pointer/subscript notation.
8.9 Relationship Between Pointers and Arrays (cont.)

```cpp
// Fig. 8.18: fig08_18.cpp
// Using subscripting and pointer notations with arrays.
#include <iostream>
using namespace std;

int main()
{
    int b[] = { 10, 20, 30, 40 }; // create 4-element array b
    int *bPtr = b; // set bPtr to point to array b

    // output array b using array subscript notation
    cout << "Array b printed with:\n\nArray subscript notation\n";

    for ( int i = 0; i < 4; i++ )
        cout << "b[" << i << "] = " << b[i] << '\n';

    // output array b using the array name and pointer/offset notation
    cout << "\nPointer/offset notation where "
        << "the pointer is the array name\n";

    for ( int offset1 = 0; offset1 < 4; offset1++ )
        cout << "*(b + " << offset1 << ") = " << *(b + offset1) << '\n';
```

Fig. 8.18 | Referencing array elements with the array name and with pointers. (Part 1 of 3.)
8.9 Relationship Between Pointers and Arrays (cont.)

```c++
23  // output array b using bPtr and array subscript notation
24  cout << "\nPointer subscript notation\n";
25
26  for ( int j = 0; j < 4; j++ )
27      cout << "bPtr[" << j << "] = " << bPtr[ j ] << '\n';
28
29  cout << "\nPointer/offset notation\n";
30
31  // output array b using bPtr and pointer/offset notation
32  for ( int offset2 = 0; offset2 < 4; offset2++ )
33      cout << "*(bPtr + " << offset2 << ") = "
34          << *( bPtr + offset2 ) << '\n';
35  }
36  // end main
```

Array b printed with:

Array subscript notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

Fig. 8.18 | Referencing array elements with the array name and with pointers. (Part 2 of 3.)
8.9 Relationship Between Pointers and Arrays (cont.)

Pointer/offset notation where the pointer is the array name
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40

Pointer subscript notation
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

Pointer/offset notation
*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40

Fig. 8.18 | Referencing array elements with the array name and with pointers. (Part 3 of 3.)
• Characters are the fundamental building blocks of C++ source programs.

• **Character constant**
  – An integer value represented as a character in single quotes.
  – The value of a character constant is the integer value of the character in the machine’s character set.

• A string is a series of characters treated as a single unit.
  – May include letters, digits and various special characters such as +, -, *, / and $.

• **String literals**, or **string constants**, in C++ are written in double quotation marks

• A pointer-based string is an array of characters ending with a **null character** (`\0`).

• A string is accessed via a pointer to its first character.
8.10 Pointer-Based String Processing (cont.)

Common Programming Error 8.15

Not allocating sufficient space in a character array to store the null character that terminates a string is an error.
Common Programming Error 8.16

Creating or using a C-style string that does not contain a terminating null character can lead to logic errors.
Error-Prevention Tip 8.4

When storing a string of characters in a character array, be sure that the array is large enough to hold the largest string that will be stored. C++ allows strings of any length to be stored. If a string is longer than the character array in which it’s to be stored, characters beyond the end of the array will overwrite data in memory following the array, leading to logic errors.
Because a string is an array of characters, we can access individual characters in a string directly with array subscript notation.

A string can be read into a character array using stream extraction with `cin`.

The `setw` stream manipulator can be used to ensure that the string read into `word` does not exceed the size of the array.

- Applies only to the next value being input.
• In some cases, it’s desirable to input an entire line of text into a character array.

• For this purpose, the cin object provides the member function getline.

• Three arguments—a character array in which the line of text will be stored, a length and a delimiter character.

• The function stops reading characters when the delimiter character '\n' is encountered, when the end-of-file indicator is entered or when the number of characters read so far is one less than the length specified in the second argument.

• The third argument to cin.getline has '\n' as a default value.
• A character array representing a null-terminated string can be output with `cout` and `<<`.

• The characters of the string are output until a terminating null character is encountered; the null character is not printed.

• `cin` and `cout` assume that character arrays should be processed as strings terminated by null characters; `cin` and `cout` do not provide similar input and output processing capabilities for other array types.
Arrays may contain pointers.

A common use of such a data structure is to form an array of pointer-based strings, referred to simply as a string array.

Each entry in the array is a string, but in C++ a string is essentially a pointer to its first character, so each entry in an array of strings is simply a pointer to the first character of a string.

```c
const char * const suit[4] = {
    "Hearts",
    "Diamonds",
    "Clubs",
    "Spades"
};
```

- An array of four elements.
- Each element is of type “pointer to char constant data.”
8.11 Arrays of Pointers

![Graphical representation of the suit array.](image)

Fig. 8.19 | Graphical representation of the suit array.
• String arrays are commonly used with command-line arguments that are passed to function `main` when a program begins execution.

• Such arguments follow the program name when a program is executed from the command line.

• A typical use of command-line arguments is to pass options to a program.
Our first example (Figs. 9.1–9.3) creates class `Time` and a driver program that tests the class.
9.2 Time Class Case Study

```
1 // Fig. 9.1: Time.h
2 // Declaration of class Time.
3 // Member functions are defined in Time.cpp
4
5 // prevent multiple inclusions of header file
6 #ifndef TIME_H
7 #define TIME_H
8
9 // Time class definition
10 class Time
11 {
12   public:
13     Time(); // constructor
14     void setTime( int, int, int ); // set hour, minute and second
15     void printUniversal(); // print time in universal-time format
16     void printStandard(); // print time in standard-time format
17   private:
18     int hour; // 0 - 23 (24-hour clock format)
19     int minute; // 0 - 59
20     int second; // 0 - 59
21   }; // end class Time
22
23 #endif
```

**Fig. 9.1** Time class definition.
In Fig. 9.1, the class definition is enclosed in the following preprocessor wrapper:

```c
// prevent multiple inclusions of header file
#ifndef TIME_H
#define TIME_H
...
#endif
```

- Prevents the code between `#ifndef` and `#endif` from being included if the name `TIME_H` has been defined.
- If the header has not been included previously in a file, the name `TIME_H` is defined by the `#define` directive and the header file statements are included.
- If the header has been included previously, `TIME_H` is defined already and the header file is not included again.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.
Fig. 9.2: Time.cpp

// Member-function definitions for class Time.
#include <iostream>
#include <iomanip>
#include "Time.h" // include definition of class Time from Time.h
using namespace std;

// Time constructor initializes each data member to zero.
// Ensures all Time objects start in a consistent state.
Time::Time()
{
    hour = minute = second = 0;
} // end Time constructor

// set new Time value using universal time; ensure that
// the data remains consistent by setting invalid values to zero
void Time::setTime( int h, int m, int s )
{
    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
} // end function setTime
9.2 Time Class Case Study (cont.)

```cpp
24  // print Time in universal-time format (HH:MM:SS)
25  void Time::printUniversal()
26  {
27      cout << setfill( '0' ) << setw( 2 ) << hour << ":
28      << setw( 2 ) << minute << ":" << setw( 2 ) << second;
29  } // end function printUniversal
30
31  // print Time in standard-time format (HH:MM:SS AM or PM)
32  void Time::printStandard()
33  {
34      cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ) << ":
35      << setfill( '0' ) << setw( 2 ) << minute << ":" << setw( 2 )
36      << second << ( hour < 12 ? " AM" : " PM" );
37  } // end function printStandard
```

**Fig. 9.2** Time class member-function definitions. (Part 2 of 2.)
• Even though a member function declared in a class definition may be defined outside that class definition, that member function is still within that class’s scope.

• If a member function is defined in the body of a class definition, the compiler attempts to inline calls to the member function.
• Once class **Time** has been defined, it can be used as a type in object, array, pointer and reference declarations as follows:

```cpp
Time sunset;  // object of type Time
Time arrayOfTimes[5];  // array of 5 Time objects
Time &dinnerTime = sunset;  // reference to a Time object
Time *timePtr = &dinnerTime;  // pointer to a Time object
```
9.2 Time Class Case Study (cont.)

```cpp
// Fig. 9.3: fig09_03.cpp
// Program to test class Time.
// NOTE: This file must be compiled with Time.cpp.
#include <iostream>
#include "Time.h" // include definition of class Time from Time.h
using namespace std;

int main()
{
    Time t; // instantiate object t of class Time

    // output Time object t's initial values
    cout << "The initial universal time is ";
    t.printUniversal(); // 00:00:00
    cout << "\nThe initial standard time is ";
    t.printStandard(); // 12:00:00 AM

    t.setTime(13, 27, 6); // change time

    // output Time object t's new values
    cout << "\n\nUniversal time after setTime is ";
    t.printUniversal(); // 13:27:06
```

**Fig. 9.3** | Program to test class Time. (Part I of 2.)
9.2 Time Class Case Study (cont.)

```cpp
23    cout << "\nStandard time after setTime is ";
24    t/printStandard(); // 1:27:06 PM
25
26    t/setTime( 99, 99, 99 ); // attempt invalid settings
27
28    // output t's values after specifying invalid values
29    cout << "\n\nAfter attempting invalid settings:";
30       << "\nUniversal time: ";
31    t/printUniversal(); // 00:00:00
32    cout << "\nStandard time: ";
33    t/printStandard(); // 12:00:00 AM
34    cout << endl;
35 } // end main
```

The initial universal time is 00:00:00
The initial standard time is 12:00:00 AM

Universal time after setTime is 13:27:06
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM

**Fig. 9.3** | Program to test class Time. (Part 2 of 2.)
• People new to object-oriented programming often suppose that objects must be quite large because they contain data members and member functions.

• Logically, this is true—you may think of objects as containing data and functions (and our discussion has certainly encouraged this view); physically, however, this is not true.
Performance Tip 9.2

Objects contain only data, so objects are much smaller than if they also contained member functions. Applying operator sizeof to a class name or to an object of that class will report only the size of the class’s data members. The compiler creates one copy (only) of the member functions separate from all objects of the class. All objects of the class share this one copy. Each object, of course, needs its own copy of the class’s data, because the data can vary among the objects. The function code is nonmodifiable and, hence, can be shared among all objects of one class.
A class’s data members and member functions belong to that class’s scope.

Nonmember functions are defined at global namespace scope.

Within a class’s scope, class members are immediately accessible by all of that class’s member functions and can be referenced by name.

Outside a class’s scope, public class members are referenced through one of the handles on an object—an object name, a reference to an object or a pointer to an object.
• Member functions of a class can be overloaded only by other member functions of that class.

• If a member function defines a variable with the same name as a variable with class scope, the class-scope variable is hidden by the block-scope variable in the local scope.
  – Such a hidden variable can be accessed by preceding the variable name with the class name followed by the scope resolution operator (::).

• The dot member selection operator (.) is preceded by an object’s name or with a reference to an object to access the object’s members.

• The arrow member selection operator (->) is preceded by a pointer to an object to access the object’s members.
9.3 Class Scope and Accessing Class Members (cont.)

```cpp
// Fig. 9.4: fig09_04.cpp
// Demonstrating the class member access operators . and ->
#include <iostream>
using namespace std;

// class Count definition
class Count {
  // public: // public data is dangerous
  // sets the value of private data member x
  void setX( int value ) {
    x = value;
  } // end function setX

  // prints the value of private data member x
  void print() {
    cout << x << endl;
  } // end function print
};
```

**Fig. 9.4** | Accessing an object’s member functions through each type of object handle—the object’s name, a reference to the object and a pointer to the object. (Part 1 of 3.)
private:
    int x;
}; // end class Count

int main()
{
    Count counter; // create counter object
    Count *counterPtr = &counter; // create pointer to counter
    Count &counterRef = counter; // create reference to counter

    cout << "Set x to 1 and print using the object's name: ";
    counter.setX( 1 ); // set data member x to 1
    counter.print(); // call member function print

    cout << "Set x to 2 and print using a reference to an object: ";
    counterRef.setX( 2 ); // set data member x to 2
    counterRef.print(); // call member function print

    cout << "Set x to 3 and print using a pointer to an object: ";
    counterPtr->setX( 3 ); // set data member x to 3
    counterPtr->print(); // call member function print

} // end main

---

**Fig. 9.4** Accessing an object’s member functions through each type of object handle—the object’s name, a reference to the object and a pointer to the object. (Part 2 of 3.)
Set x to 1 and print using the object's name: 1
Set x to 2 and print using a reference to an object: 2
Set x to 3 and print using a pointer to an object: 3

Fig. 9.4 | Accessing an object’s member functions through each type of object handle—the object’s name, a reference to the object and a pointer to the object. (Part 3 of 3.)
• Separating classes into two files—a header file for the class definition (i.e., the class’s interface) and a source code file for the class’s member-function definitions (i.e., the class’s implementation) makes it easier to modify programs.
• **Access functions** can read or display data.

• A common use for access functions is to test the truth or falsity of conditions—such functions are often called *predicate functions*.

• A **utility function** is a *private* member function that supports the operation of the class’s *public* member functions.
9.5 Access Functions and Utility Functions

```cpp
// Fig. 9.5: SalesPerson.h
// SalesPerson class definition.
// Member functions defined in SalesPerson.cpp.
#ifndef SALESP_H
#define SALESP_H

class SalesPerson {
public:
    static const int monthsPerYear = 12; // months in one year
    SalesPerson(); // constructor
    void getSalesFromUser(); // input sales from keyboard
    void setSales(int, double); // set sales for a specific month
    void printAnnualSales(); // summarize and print sales
private:
    double totalAnnualSales(); // prototype for utility function
    double sales[monthsPerYear]; // 12 monthly sales figures
}; // end class SalesPerson
#endif
```

**Fig. 9.5** SalesPerson class definition.
9.5 Access Functions and Utility Functions

```cpp
// Fig. 9.6: SalesPerson.cpp
// SalesPerson class member-function definitions.
#include <iostream>
#include <iomanip>
#include "SalesPerson.h" // include SalesPerson class definition
using namespace std;

// initialize elements of array sales to 0.0
SalesPerson::SalesPerson()
{
    for ( int i = 0; i < monthsPerYear; i++ )
        sales[ i ] = 0.0;
} // end SalesPerson constructor

// get 12 sales figures from the user at the keyboard
void SalesPerson::getSalesFromUser()
{
    double salesFigure;

    for ( int i = 1; i <= monthsPerYear; i++ )
    {
        cout << "Enter sales amount for month " << i << ": ";
        cin >> salesFigure;
    }
}
```

Fig. 9.6 | SalesPerson class member-function definitions. (Part I of 3.)
9.5 Access Functions and Utility Functions

```c++
setSales( i, salesFigure );

} // end for
}
} // end function getSalesFromUser

// set one of the 12 monthly sales figures; function subtracts
// one from month value for proper subscript in sales array
void SalesPerson::setSales( int month, double amount )
{
    // test for valid month and amount values
    if ( month >= 1 && month <= monthsPerYear && amount > 0 )
        sales[ month - 1 ] = amount; // adjust for subscripts 0-11
    else // invalid month or amount value
        cout << "Invalid month or sales figure" << endl;

} // end function setSales

// print total annual sales (with the help of utility function)
void SalesPerson::printAnnualSales()
{
    cout << setprecision( 2 ) << fixed
        << "The total annual sales are: \$" << totalAnnualSales() << endl; // call utility function
}
} // end function printAnnualSales
```

Fig. 9.6 | SalesPerson class member-function definitions. (Part 2 of 3.)
// private utility function to total annual sales
double SalesPerson::totalAnnualSales()
{
   double total = 0.0; // initialize total

   for ( int i = 0; i < monthsPerYear; i++ ) // summarize sales results
      total += sales[ i ]; // add month i sales to total

   return total;
} // end function totalAnnualSales

Fig. 9.6 | SalesPerson class member-function definitions. (Part 3 of 3.)
```cpp
// Fig. 9.7: fig09_07.cpp
// Utility function demonstration.
// Compile this program with SalesPerson.cpp

// include SalesPerson class definition from SalesPerson.h
#include "SalesPerson.h"

int main()
{
    SalesPerson s; // create SalesPerson object s

    s.getSalesFromUser(); // note simple sequential code; there are
    s.printAnnualSales(); // no control statements in main
}
```

**Fig. 9.7** Utility function demonstration. (Part 1 of 2.)
Enter sales amount for month 1: 5314.76
Enter sales amount for month 2: 4292.38
Enter sales amount for month 3: 4589.83
Enter sales amount for month 4: 5534.03
Enter sales amount for month 5: 4376.34
Enter sales amount for month 6: 5698.45
Enter sales amount for month 7: 4439.22
Enter sales amount for month 8: 5893.57
Enter sales amount for month 9: 4909.67
Enter sales amount for month 10: 5123.45
Enter sales amount for month 11: 4024.97
Enter sales amount for month 12: 5923.92

The total annual sales are: $60120.59

Fig. 9.7  |  Utility function demonstration. (Part 2 of 2.)