Object Oriented Programming

Shaobai Kan

chapter 7 (Continue...)

Outline

Outline

- Passing arguments to functions by reference with pointers

Outline

- Passing arguments to functions by reference with pointers
- Using const with Pointers

Outline

- Passing arguments to functions by reference with pointers
- Using const with Pointers
- sizeof operator

Outline

- Passing arguments to functions by reference with pointers
- Using const with Pointers
- sizeof operator
- Project I

Passing arguments to functions by reference with pointers

Passing arguments to functions

Definition. There are two two ways in C++ to pass arguments to a function

Passing arguments to functions

Definition. There are two two ways in C++ to pass arguments to a function

- pass-by-value
- pass-by-reference

Passing arguments to functions

Definition. There are two two ways in C++ to pass arguments to a function

- pass-by-value
- pass-by-reference
  - pass-by-reference with reference arguments (&)
  - pass-by-reference with pointer arguments (*)

Passing arguments to functions with pointers

Fact.

- In C++, programmers can use pointers and the indirection operator (*) to accomplish pass-by-reference.
- When calling a function with an argument that should be modified,the address of the argument is passed.
- It is normally accomplished by applying the address operator (&) to the name of the variable whose value will be modified.

Pass-by-value

Passing arguments by value

```
# include <iostream>
using namespace std;
void cubeByValue ( int );        //prototype

int main( )
{
        int number = 5 ;

        cout << "number : " << number << endl ;

        cubeByValue ( number) ;

        cout << "cube: " << number << endl;
        return 0 ;
}

 void cubeByValue ( int n)        //header
{
         n = n * n * n;
}
```

Output: pass-by-value

```
number:5
cube:5
```

Pass-by-reference with pointers

Passing arguments by reference with Pointers

```
# include <iostream>
using namespace std;
void cubeByReference ( int * );          //prototype
int main( )
{
        int number = 5 ;
        cout << "number:" << number << endl;
         cubeByReference ( & number );
         cout << "number:" << number << endl;
        return 0 ;
 }
  void cubeByReference ( int * nPtr )          //header
 {
       * nPtr = * nPtr * * nPtr * * nPtr;
  }
```

Output: pass-by-reference with pointers

```
number:5
number:125
```

Pass-by-reference (references *v.s.* pointers)

References *v.s.* Pointers

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

void square ( int & );

int main( )
{
        int a = 2 ;

        square ( a );

        cout << "a: " << a << endl;

        return 0 ;
}

void square ( int & aRef )
{
        aRef = aRef * aRef;
}
```

argument

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

void square ( int * );

int main( )
{
        int a = 2 ;

        square ( &a );

        cout << "a: " << a << endl;

        return 0 ;
}

void square ( int * aPtr)
{
        *aPtr = *aPtr * *aPtr ;
}
```

argument

Using const with Pointers

Constant variable *v.s.* constant pointer

Recall. Constant variable is a variable whose value can not be modified after it is initialized.

—— constant variable must be initialized with a constant expression.

Constant variable *v.s.* constant pointer

Recall. Constant variable is a variable whose value can not be modified after it is initialized.

—— constant variable must be initialized with a constant expression.

Fact. Constant pointer is a pointer that always points to the same memory location.

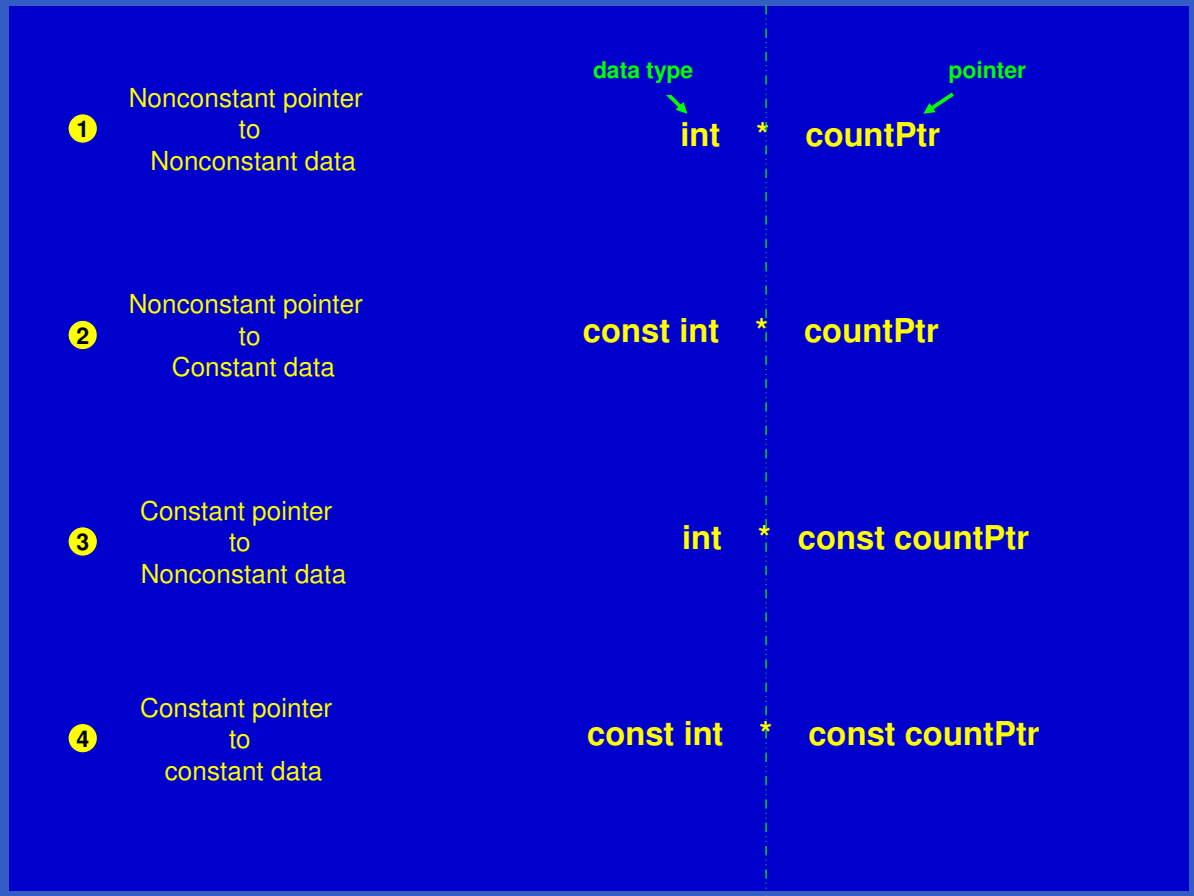
—— A constant pointer must be initialized.

Const and pointer declarations

Fact. There are 4 ways to combine const with pointer declarations:

- Nonconstant Pointer to Nonconstant Data
  — e.g. `int * countPtr;`
- Nonconstant Pointer to constant Data
  — e.g. `const int * countPtr;`
- Constant Pointer to Nonconstant Data
  — e.g. `int * const countPtr;`
- Constant Pointer to Constant Data
  — e.g. `const int * const countPtr;`

Const and pointer declarations



Nonconstant pointer to nonconstant data

Nonconstant pointer to nonconstant data: the data can be modified through the dereferenced pointer, and the pointer can be modified to point to other data.

— example.

```
.....
int a = 5, b = 7;
int * y;
y = & a;
* y = 6;
y = & b;
.....
```

Nonconstant pointer to constant data

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.

Fact. Such a 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.

Nonconstant pointer to constant data

Nonconstant pointer to constant data

```
# include <iostream>
using namespace std;
int max ( const int  * , const int );

int main( )
{
        const int n = 6;
        int a [ n ] = { 2, 6, 10, -4, 1, -10} ;
        cout << "The maximum is: " << max (a, 6) << endl ;
        return 0 ;
 }

int max ( const int  * aPtr, const int size )
{
        int  maximum = *aPtr;
        for ( int i = 0; i < size; i++ )
        {
              if ( maximum < * (aPtr + i) )
                 maximum = *(aPtr + i);
        }
        return maximum;
}
```

array size

Constant pointer to nonconstant data

Constant pointer to nonconstant data: a pointer that always points to the same memory location; the data at that location *can* be modified through the pointer.

Fact. An array name is a typical constant pointer to nonconstant data. It is a constant pointer to the beginning of the array.

Constant pointer to nonconstant data

Constant Pointer to Nonconstant Data

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

int main( )
{
        int x = 2, y = 4 ;
        int  * const ptr = &x;
        * ptr = 9;          // nonconstant data
        // ptr = &y;        Error!!! --- constant pointer
        cout << "x: " << x << "y:" << y << endl;
        return 0 ;
}
```

Constant pointer to constant data

Constant pointer to constant data: a pointer that always points to the same memory location; the data at that location cannot be modified via the pointer.

Fact. 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.

Constant pointer to constant data

Constant Pointer to Constant Data

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

int main( )
{
        int x = 2, y = 4 ;
        const int  * const ptr = &x;
       // * ptr = 9;         Error!!! --- constant data
       // ptr = &y;         Error!!! --- constant pointer
        cout << "x: " << x << "y:" << y << endl;
        return 0 ;
}
```

Sizeof operator

sizeof operator

Fact.
C++ provides the unary operator sizeof to determine the size of an array (or of any other data type, variable or constant) in bytes during program compilation.

sizeof operator

sizeof operator

```
# include <iostream>

using namespace std;

int main( )
{
        double y = 5.0 ;
        int x = 2, size1, size2 ;

         size1 = sizeof (y);
         size2 = sizeof (x);

        cout << "size of variable y: " << size1 << "  bytes\n" ;
        cout << "size of variable x: " << size2 << "  bytes\n" ;
        return 0 ;
 }
```

Output: sizeof operator

```
Size of variable y: 8 bytes
Size of variable x: 4 bytes
```

Project I

Project I

Project. *Simulation: The Tortoise and the Hare.*

#7.12 Textbook Page 330.

—— *Deadline: 2/13/2013*

Homework:

- Read Sec. 7.4 - 7.8
- practice the program in Figure 7.13 (in the textbook section 7.6)
- practice the program in Figure 7.15 (in the textbook section 7.7)
- Exercise 7.8, 7.10