Chapter 11: Pointers and Dynamic Memory Management

Sections 11.1–11.2, 11.5–11.7

Textbooks: Y. Daniel Liang, Introduction to Programming with C++, 3rd Edition
© Copyright 2016 by Pearson Education, Inc. All Rights Reserved.

These slides were adapted by Prof. Gheith Abandah from the Computer Engineering Department of the University of Jordan for the Course: Computer Skills for Engineers (0907101)
Updated by Dr. Ashraf Suyyagh (Spring 2021)

Outline

- Introduction
- Pointer Basics
- Arrays and Pointers
- Passing Pointer Arguments in a Function Call
- Returning a Pointer from Functions

Introduction

- *Pointer variables*, simply called *pointers*, are designed to hold memory addresses as their values.
- Normally, a variable contains a specific value, e.g., an integer, a floating-point value, and a character.
- However, a pointer contains the memory address of a variable that in turn contains a specific value.

Outline

- Introduction
- Pointer Basics
- Arrays and Pointers
- Passing Pointer Arguments in a Function Call
- Returning a Pointer from Functions

Pointer Basics

```
int count = 5;
short status = 2;
char letter = 'A';
string word = "ABC";
int* pCount = &count;
```

&count means the address of count, in this case 0x0013FF60

(00 13 FF 60)

Address	Content
0013FF72	
0013FF71	00
0013FF70	13
0013FF6F	FF
0013FF6E	60
0013FF6D	
0013FF6C	
0013FF6B	
0013FF6A	
0013FF69	57
0013FF68	56
0013FF67	55
0013FF66	55
0013FF65	0
0013FF64	02
0013FF63	0
0013FF62	0
0013FF61	0
0013FF60	05

pCount (int pointer type, 4)

word (string type, 3 bytes)

letter (char type, 1 byte)

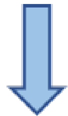
status (short type, 2 bytes)

Count (int type, 4 bytes)

Pointer Basics 2

The * operator means the value at this address

```
cout << *pCount << endl;
```



*(0x0013FF60)

5

Address	Content
0013FF72	
0013FF71	00
0013FF70	13
0013FF6F	FF
0013FF6E	60
0013FF6D	
0013FF6C	
0013FF6B	
0013FF6A	
0013FF69	57
0013FF68	56
0013FF67	55
0013FF66	55
0013FF65	0
0013FF64	02
0013FF63	0
0013FF62	0
0013FF61	0
0013FF60	05

Pointer Basics 3

Declaration and Initialization:

```
int * pCount = &Count;
```

`int * pCount` → Create a pointer that points to an integer value.

`&Count` → Get the address of the integer value

Using (Dereferencing) the pointer

```
cout<< *pCount << endl;
```

`*pCount` → Get the value at the address stored in the pointer

Address	Content
0013FF72	
0013FF71	00
0013FF70	13
0013FF6F	FF
0013FF6E	60
0013FF6D	
0013FF6C	
0013FF6B	
0013FF6A	
0013FF69	57
0013FF68	56
0013FF67	55
0013FF66	55
0013FF65	0
0013FF64	02
0013FF63	0
0013FF62	0
0013FF61	0
0013FF60	05

Declare a Pointer

- Like any other variables, pointers must be declared before they can be used. To declare a pointer, use the following syntax:
```
dataType* pVarName;
```

- Each variable being declared as a pointer must be preceded by an asterisk (*). For example, the following statement declares a pointer variable named **pCount** that can point to an **int** variable.
```
int* pCount;
```

TestPointer Run

TestPointer.cpp

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

int main()
{
    int count = 5;
    int* pCount = &count;

    cout << "The value of count is " << count <<
endl;
    cout << "The address of count is " << &count <<
endl;
    cout << "The address of count is " << pCount <<
endl;
    cout << "The                                   nt <<
endl;

    return 0;
```

```
The value of count is 5
The address of count is
00AFF980
The address of count is
00AFF980
The value of count is 5
```

Dereferencing

- Referencing a value through a pointer is called *indirection*. The syntax for referencing a value from a pointer is:

```
*pointer
```

- For example, you can increase **count** using:

```
count++;       // direct reference
```

or

```
(*pCount)++; // indirect reference
```

- The asterisk (*) is the *indirection operator* or *dereference operator*.

Pointer and Value Assignment

```
int x = 9;
int y = 5;
```

Address	Content
0013FF72	
0013FF71	
0013FF70	
0013FF6F	
0013FF6E	
0013FF6D	
0013FF6C	
0013FF6B	
0013FF6A	
0013FF69	
0013FF68	
0013FF67	0
0013FF66	0
0013FF65	0
0013FF64	05
0013FF63	0
0013FF62	0
0013FF61	0
0013FF60	09

y (0013FF64–0013FF67)

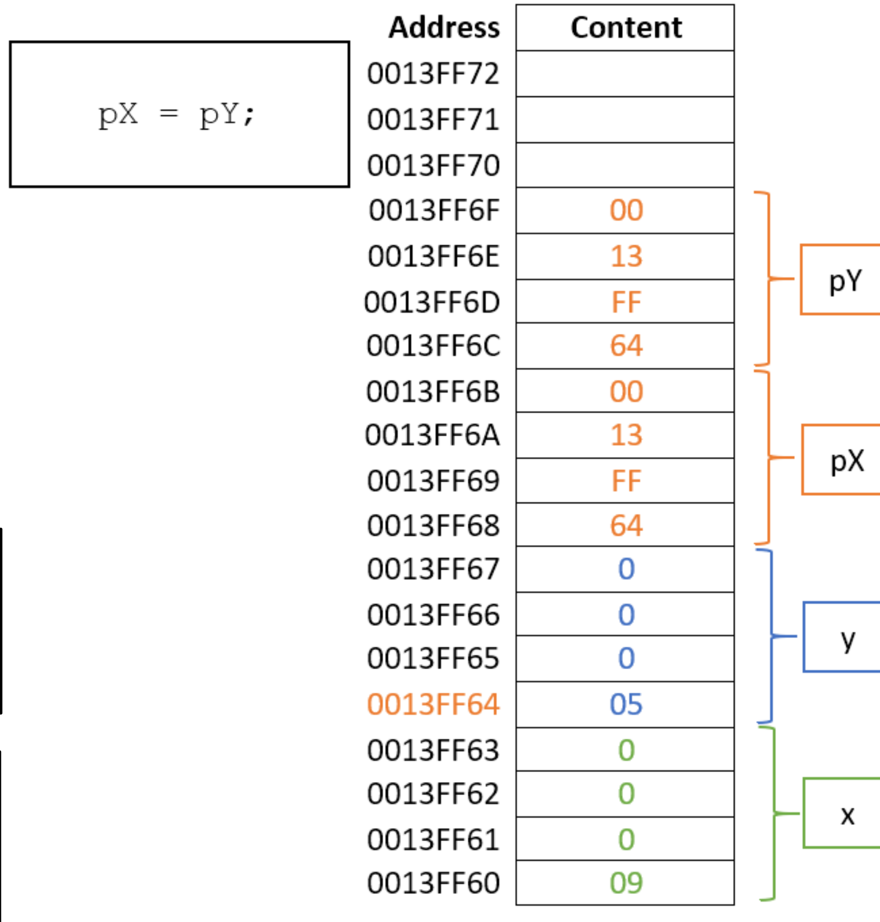
x (0013FF60–0013FF63)

Pointer and Value Assignment

```
int* pX = &x;
int* pY = &y;
```

Address	Content	
0013FF72		
0013FF71		
0013FF70		
0013FF6F	00	pY
0013FF6E	13	
0013FF6D	FF	
0013FF6C	64	
0013FF6B	00	pX
0013FF6A	13	
0013FF69	FF	
0013FF68	60	
0013FF67	0	y
0013FF66	0	
0013FF65	0	
0013FF64	05	
0013FF63	0	x
0013FF62	0	
0013FF61	0	
0013FF60	09	

Pointer and Value Assignment




```
cout<<*pX<<endl;
cout<<*pY<<endl;
cout<<x<<endl;
cout<<y<<endl;
```

```
5
5
9
5
```

Pointer and Value Assignment

```
*pX = *pY;
```

Address	Content
0013FF72	
0013FF71	
0013FF70	
0013FF6F	00
0013FF6E	13
0013FF6D	FF
0013FF6C	64
0013FF6B	00
0013FF6A	13
0013FF69	FF
0013FF68	60
0013FF67	0
0013FF66	0
0013FF65	0
0013FF64	05
0013FF63	0
0013FF62	0
0013FF61	0
0013FF60	05

pY

pX

y

x

```
cout<<*pX<<endl;
cout<<*pY<<endl;
cout<<x<<endl;
cout<<y<<endl;
```

```
5
5
5
5
```

Pointer Type

- A pointer variable is declared with a type such as **`int`**, **`double`**, etc.
- You have to assign the address of the variable of the same type.
- It is a syntax error if the type of the variable does not match the type of the pointer. For example, the following code is wrong.

```
int area = 1;
double* pArea = &area; // Wrong
```

Initializing Pointer

- Like a local variable, a local pointer is assigned an arbitrary value if you don't initialize it.
- A pointer may be initialized to **`0`**, which is a special value for a pointer to indicate that the pointer points to nothing.
- You should always initialize pointers to prevent errors.
- Dereferencing a pointer that is not initialized could cause fatal runtime error or it could accidentally modify important data.

Caution

- You can declare two variables on the same line. For example, the following line declares two **int** variables:
```
int i= 0, j = 1;
```

- Can you declare two pointer variables on the same line as follows?
```
int* pl, pj;
```

- No, the right way is:
```
int *pl, *pj;
```

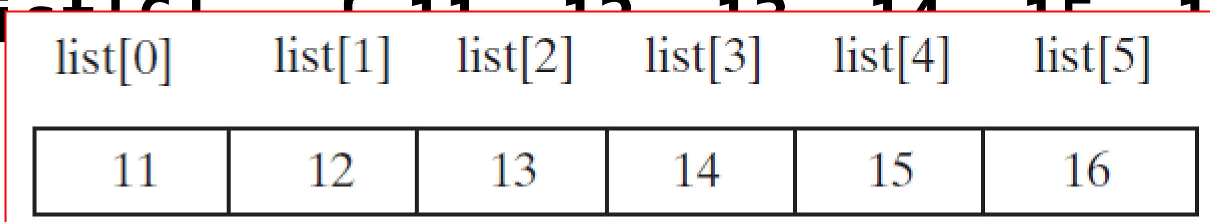
Outline

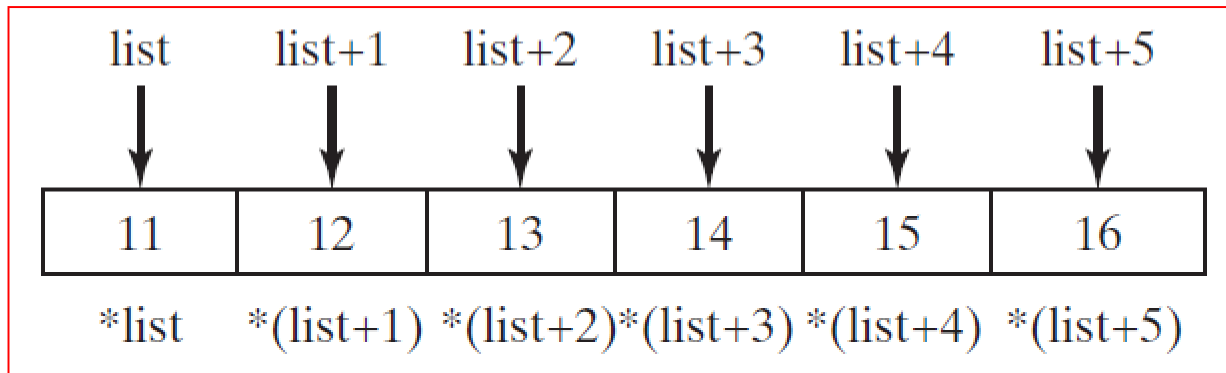
- Introduction
- Pointer Basics
- Arrays and Pointers
- Passing Pointer Arguments in a Function Call
- Returning a Pointer from Functions

Arrays and Pointers

- An array variable without a bracket and a subscript actually represents the starting address of the array.
- The array variable is essentially a pointer. Suppose you declare an array of **int** value as follows:

```
int list[6] = { 11, 12, 13, 14, 15, 16 };
```





Array Pointer

- **`*(list + 1)`** is different from **`*list + 1`**. The dereference operator (**`*`**) has precedence over **`+`**.
- So, **`*list + 1`** adds **`1`** to the value of the first element in the array, while **`*(list + 1)`** dereference the element at address **`(list + 1)`** in the array.

ArrayPointer Run

PointerWithIndex Run

ArrayPointer.cpp

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

int main()
{
    int list[6] = { 11, 12, 13, 14, 15, 16 };

    for (int i = 0; i < 6; i++)
        cout << "address: " << (list + i) <<
        " value: " << *(list + i) << " " <<
        " value: " << list[i] << endl;

    return 0;
}
```

```
address: 0013FF4C value: 11  value: 11
address: 0013FF50 value: 12  value: 12
address: 0013FF54 value: 13  value: 13
address: 0013FF58 value: 14  value: 14
address: 0013FF5C value: 15  value: 15
address: 0013FF60 value: 16  value: 16
```

PointerWithIndex.cpp

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

int main()
{
    int list[6] = { 11, 12, 13, 14, 15, 16 };
    int* p = list;

    for (int i = 0; i < 6; i++)
        cout << "address: " << (list + i) <<
        " value: " << *(list + i) << " " <<
        " value: " << list[i] << " " <<
        " value: " << *(p + i) << " " <<
        " value: " << p[i] << endl;

    return 0;
}
```

```
address: 0013FF4C value: 11  value: 11  value: 11  value: 11
address: 0013FF50 value: 12  value: 12  value: 12  value: 12
address: 0013FF54 value: 13  value: 13  value: 13  value: 13
address: 0013FF58 value: 14  value: 14  value: 14  value: 14
address: 0013FF5C value: 15  value: 15  value: 15  value: 15
address: 0013FF60 value: 16  value: 16  value: 16  value: 16
```

Outline

- Introduction
- Pointer Basics
- Arrays and Pointers
- Passing Pointer Arguments in a Function Call
- Returning a Pointer from Functions

Passing Pointer Arguments

- A pointer argument can be passed by value or by reference. For example, you can define a function as follows:

```
void f(int* p1, int*& p2);
```

- which is equivalently to

```
typedef int* intPointer;
void f(intPointer p1, intPointer & p2);
```

- Here p1 is pass-by-value and p2 is pass-by-reference.

TestPointerArgument

Run

Four Versions of the Swap Function

```
// Swap two variables using
// pass-by-value
void swap1(int n1, int n2)
{
    int temp = n1;
    n1 = n2;
    n2 = temp;
}
```

```
// Pass two pointers by
value
void swap3(int* p1, int* p2)
{
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}
```

```
// Swap two variables using
// pass-by-reference
void swap2(int& n1, int& n2)
{
    int temp = n1;
    n1 = n2;
    n2 = temp;
}
```

```
// Pass two pointers by
// reference
void swap4(int*& p1, int*&
p2)
{
    int* temp = p1;
    p1 = p2;
    p2 = temp;
}
```

TestPointerArgument.cpp 1/6

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

// Function prototypes are here
int main()
{
    // Declare and initialize variables
    int num1 = 1;
    int num2 = 2;

    cout << "Before invoking the swap function, num1 is "
        << num1 << " and num2 is " << num2 << endl;

    // Call one of the first three swap functions here

    cout << "After invoking the swap function, num1 is "
         << num1 << " and num2 is " << num2 << endl;
}
```

TestPointerArgument.cpp 2/6

```
Before invoking the swap function, num1 is 1 and num2 is 2
```

```
swap1(num1, num2)
```

```
// Swap two variables using
// pass-by-value
void swap1(int n1, int n2)
{
    int temp = n1;
    n1 = n2;
    n2 = temp;
}
```

```
After invoking the swap function, num1 is 1 and num2 is 2
```

TestPointerArgument.cpp 3/6

```
Before invoking the swap function, num1 is 1 and num2 is 2
```

```
swap2(num1, num2)
```

```
// Swap two variables using
// pass-by-reference
void swap2(int& n1, int& n2)
{
    int temp = n1;
    n1 = n2;
    n2 = temp;
}
```

After invoking the swap function, num1 is 2 and num2 is 1

TestPointerArgument.cpp 4/6

```
Before invoking the swap function, num1 is 1 and num2 is 2
```

```
swap3(&num1,&num2)
```

```
// Pass two pointers by
value
void swap3(int* p1, int* p2)
{
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}
```

After invoking the swap function, num1 is 2 and num2 is 1

TestPointerArgument.cpp 5/6

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

void swap4(int*& p1, int*& p2);
int main()
{   // Declare and initialize variables
    int num1 = 1;
    int num2 = 2;
    int* pointer1 = &num1;
    int* pointer2 = &num2;

    cout << "Before invoking the swap function, num1 is "
        << pointer1 << " and num2 is " << pointer2 <<
endl;

    swap4(pointer1, pointer2);

    cout << "After invoking the swap function, num1 is "
         << pointer1 << " and num2 is " << pointer2 <<
endl;
```

TestPointerArgument.cpp 6/6

Before invoking the swap function, pointer1 is 0028FB84 and pointer2 is 0028FB78

```
swap4(pointer1, pointer2)
```

```
// Pass two pointers by
// reference
void swap4(int*& p1, int*& p2)
{
    int* temp = p1;
    p1 = p2;
    p2 = temp;
}
```

After invoking the swap function, pointer1 is 0028FB78 and pointer2 is 0028FB84

Array Parameter or Pointer Parameter

- An array parameter in a function can always be replaced using a pointer parameter.

`void m(int list[], int size)`	can be replaced by	`void m(int* list, int size)`
`void m(char c_string[])`	can be replaced by	`void m(char* c_string)`

`const` Parameter

If an object value does not change, you should declare it **`const`** to prevent it from being modified accidentally.

ConstParameter

Run

ConstParameter.cpp

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

void printArray(const int*, const int);

int main()
{
    int list[6] = { 11, 12, 13, 14, 15, 16 };
    printArray(list, 6);

    return 0;
}

void printArray(const int* list, const int size)
{
    for (int i = 0; i < size; i++)
        cout << list[i] << " ";
}
```

```
11 12 13 14 15 16
```

Outline

- Introduction
- Pointer Basics
- Arrays and Pointers
- Passing Pointer Arguments in a Function Call
- Returning a Pointer from Functions

Returning a Pointer from Functions

- You can use pointers as parameters in a function.
- A C++ function may return a pointer as well.

ReverseArrayUsingPointer

Run

ReverseArrayUsingPointer.cpp 1/2

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

int* reverse(int* list, int size)
{
    for (int i = 0, j = size - 1; i < j; i++, j--)
    {
        // Swap list[i] with list[j]
        int temp = list[j];
        list[j] = list[i];
        list[i] = temp;
    }

    return list;
}
```

ReverseArrayUsingPointer.cpp 2/2

```
void printArray(const int* list, int size)
{
    for (int i = 0; i < size; i++)
        cout << list[i] << " ";
}

int main()
{
    int list[] = { 11, 12, 13, 14, 15, 16 };
    int* p = reverse(list, 6);
    printArray(p, 6);

    return 0;
}
```

```
16 15 14 13 12 11
```

Outline

- Introduction
- Pointer Basics
- Arrays and Pointers
- Passing Pointer Arguments in a Function Call
- Returning a Pointer from Functions