哈爾濱工業大學(深圳)
HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

高级语言程序设计
High-level Language Programming

Lecture 10 Pointers and Memories

Yitian Shao (shaoyitian@hit.edu.cn)
School of Computer Science and Technology

Pointers and Memories

Course Overview

- Addresses and pointers
- Pointers and arrays
- Pointers to class/struct
- Pointers as function arguments
- Dynamic memory allocation

10.1 Addresses and Pointers

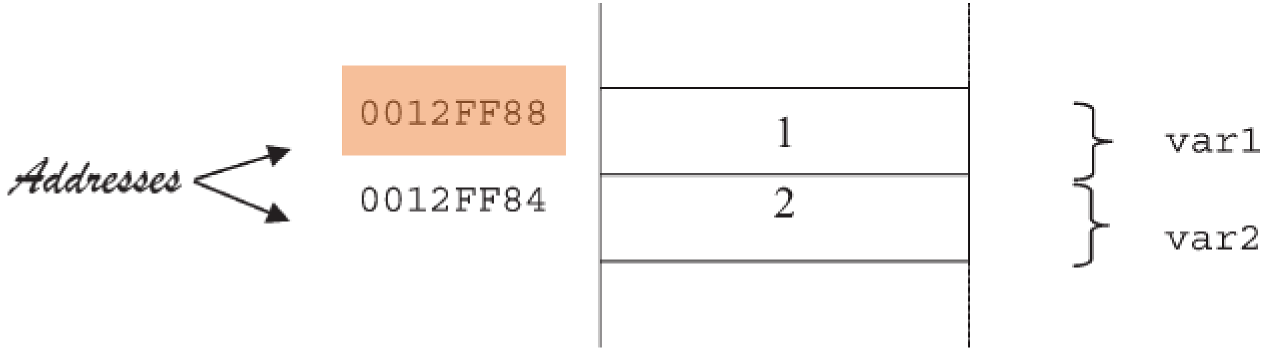
- Every variable and object used in a C++ program is stored in a specific place in memory.
  - Each location in memory has a **unique address**
  - uses **&** to **get the address** of a variable

```cpp
// Program to display the address of variables.
#include <iostream>
#include <iomanip>
using namespace std ;

main()
{
   int var1 = 1 ;
   float var2 = 2 ;

   cout << "var1 has a value of " << var1
        << " and is stored at " << &var1 << endl ;
   cout << "var2 has a value of " << var2
        << " and is stored at " << &var2 << endl ;
}
```

```
var1 has a value of 1 and is stored at 0012FF88
var2 has a value of 2 and is stored at 0012FF84
```

10.1 Addresses and Pointers

- How the variables **var1** and **var2** are stored in memory?



- Different computers may give different addresses from the ones above.
  - Various computers and operating systems will store variables at different memory locations.
  - The addresses are in **hexadecimal** (base 16).

10.1 Addresses and Pointers

- A **pointer variable** is a **variable** that **holds the address** of another variable.

```
data_type* variable_name ;
```

data_type can be any data type (such as char, int, float, a struct, a class and so on)

variable_name can be any valid variable name.

```
int* int_ptr ;      // int_ptr is a pointer to an int variable.
float* float_ptr ; // float_ptr is a pointer to a float variable.
bank_account* b ;  // b is a pointer to a bank_account object.
```

- Whitespace in a pointer definition is not relevant.

```
int * int_ptr ;        int *int_ptr ;        int*int_ptr ;
```

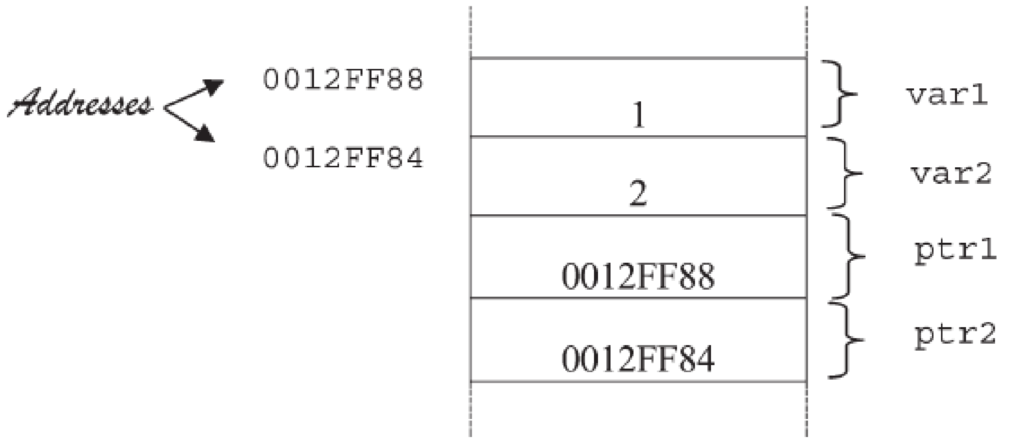
10.1 Addresses and Pointers

- Pointer definitions are read backwards from the variable name, replacing * with the words “is a pointer”.

  - **int*** int_ptr means that int_ptr is a pointer to an int
  - **float*** float_ptr means that float_ptr is a pointer to a float

10.1 Addresses and Pointers

```
// Demonstration of pointer variables.
#include <iostream>
using namespace std ;

main()
{
  int var1 = 1 ;
  float var2 = 2 ;
  int* ptr1 ;
  float* ptr2 ;

  ptr1 = &var1 ;   // ptr1 contains the address of var1.
  ptr2 = &var2 ;   // ptr2 contains the address of var2.
  cout << "ptr1 contains " << ptr1 << endl ;
  cout << "ptr2 contains " << ptr2 << endl ;
}
```



The two variables ptr1 and ptr2 are used to store the addresses of the other two variables, var1 and var2.

```
ptr1 contains 0012FF88
ptr2 contains 0012FF84
```

10.1 Addresses and Pointers

- The **dereference operator *** is used to access the value of a variable, whose address is stored in a pointer
  - ****ptr*** means the value of the variable at the address stored in the pointer variable *ptr*

```
2  // Demonstration of dereference operator *
3  #include <iostream>
4  using namespace std ;
5
6  main()
7  {
8    int var =1 ;
9    int* ptr ;
10
11   ptr = &var ; // ptr contains the address of var
12   cout << "ptr contains " << ptr << endl ;
13   cout << "*ptr contains " << *ptr << endl ;
14 }
```

```
ptr contains 0012FF88
*ptr contains 1
```

•Line 13 displays the value at the address held in *ptr* by using the dereference operator *. This is called *dereferencing the* pointer *ptr.*
•The value of *ptr is the same as the value of *var*.

•In line 9, the * is used to **define ptr as a pointer** to an int.
•In line 13, the * is used to **access the value** of the memory location, the address of which is in ptr.

10.1 Addresses and Pointers

- When defining a pointer, the pointer itself, the value it points to or both can be made constant. The position of const in the definition determines which of these apply.

```
int i, j ;

const int* p = &i ; // *p is a constant but p is not.

*p = 5 ; // Illegal

i = 5 ; // Legal

p = &j ; // Legal
```

The definition reads as "p is a pointer to an **integer constant**"

- **The integer is constant and cannot be changed using pointer p**
- The value of i can be changed
- The pointer may be changed

10.1 Addresses and Pointers

- This definition of p reads as "p is a pointer to a **constant integer**".

```
const int* p = &i ;        int const* p = &i ;   // *p is a constant; p is not.
```

This means that the integer is constant and cannot be changed using pointer p

```
*p = 5 ; // Illegal     p = &j ; // Legal
```

- This definition of p reads, "p is a **constant pointer** to an integer".

```
int* const p = &i ;   // p is a constant; *p is not.
```

This means that the pointer is a constant but not what it points to

```
*p = 5 ; // Legal: *p can be changed.
p = &j ; // Illegal: p is a constant.
```

10.1 Addresses and Pointers

- This definition of p reads, "p is a **constant pointer** to a **constant integer**".

```
const int* const p = &i ; // Both p and *p are constants.
```

  - This means that **both the pointer and the integer it points to are constants**.

```
*p = 5 ; // Illegal: *p is a constant.
p = &j ; // Illegal: p is a constant.
```

10.2 Pointers and Arrays

- The **name of an array** is a **pointer** to the **first element** of the array.

```
int a[5] ;
```

  - The elements of this array are: a[0], a[1], a[2], a[3], a[4]
  - The **name of the array** *a* is equivalent to the address of the first element; **a** is the same as **&a[0]**

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

main()
{
  int a[5] ;

  cout << "a is " << a
       << " and &a[0] is " << &a[0] << endl  ;
}
```

```
a is 0012FF78 and &a[0] is 0012FF78
```

- a + 1 is the address of the second element
- a + 2 is the address of the third element
- ….

10.2 Pointers and Arrays

- As the name of an array is a pointer to the first element of the array, the **dereference operator** * can be used to access the elements of the array

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

main()
{
  int a[5] = { 10, 13, 15, 11, 6 } ;

  for ( int i = 0 ; i < 5 ; i++ )
    cout << "Element " << i << " is " << *( a + i ) << endl ;
}
```

```
Element 0 is 10
Element 1 is 13
Element 2 is 15
Element 3 is 11
Element 4 is 6
```

*(a+0) or *a is equivalent to a[0]
*(a+1) is equivalent to a[1]
*(a+2) is equivalent to a[2],

the expression *(a+i) is not equal to the expression *a+i

10.2 Pointers and Arrays

- Use pointers to access the elements of any array

```
float numbers[100] ;
```

  - numbers[i] is equivalent to *(numbers + i).
  - Although the name of an array is a pointer to the first element of the array, you **cannot** change its value; this is because it is a **constant pointer**.
    - a++ or numbers+=2 are invalid

```
int a[5] ;
int* p ;
p = a ;
 // Valid: assignment of a constant to a variable.
a++ ;
 // Invalid: the value of a constant cannot change.
p++ ;
 // Valid: p is a variable. p now points to
 // element 1 of the array a.
p-- ;
 // Valid: p points to element 0 of the array a.
p += 10 ;
 // Valid, but p is outside the range of the array a,
 // so *p is undefined. A common error.
p = a - 1 ;
 // Valid, but p is outside the range of the array.
```

10.2 Pointers and Arrays

- A constant may be added to or subtracted from the value of a pointer, allowing access to different memory locations.

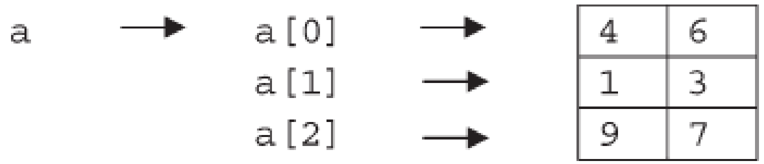
- **Not** all arithmetic operations are permissible on pointers.
  - The multiplication of two pointers is illegal, because the result would not be a valid memory address.

10.2 Pointers and Arrays (Further)

- Access the elements of a multi-dimensional array using pointers

```
int a[3][2] = { { 4, 6 },
                { 1, 3 },
                { 9, 7 } } ;
```

  - A two-dimensional array is stored as an ‘array of arrays’.
  - This means that a is a one-dimensional array whose elements are themselves a one-dimensional arrays of integers



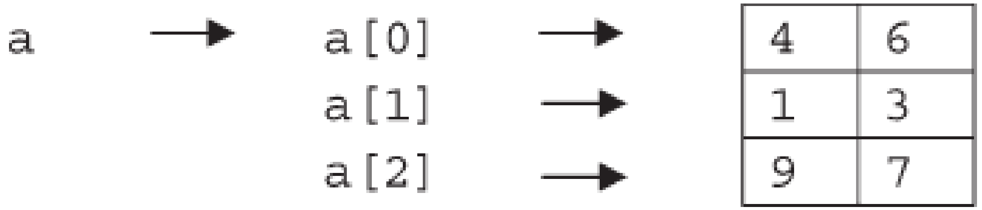
•The name of a two-dimensional array is a pointer to the first element of the array.
-------a is equivalent to &a[0]

•a[0] is itself an array of two integers
-------a[0] is equivalent to &a[0][0]

10.2 Pointers and Arrays (Further)

- a[0], a[1] and a[2] are **pointers** (data type is int*) and a is a **pointer to a pointer** (data type is int **).
  - a[0] is the address of the first element in the first row of the array. *a[0] is a[0][0], which is 4.
  - a[1] is the address of the first element in the second row. *a[1] is a[1][0], which is 1.
  - a[2] is the address of the first element in the third row. *a[2] is a[2][0], which is 9.

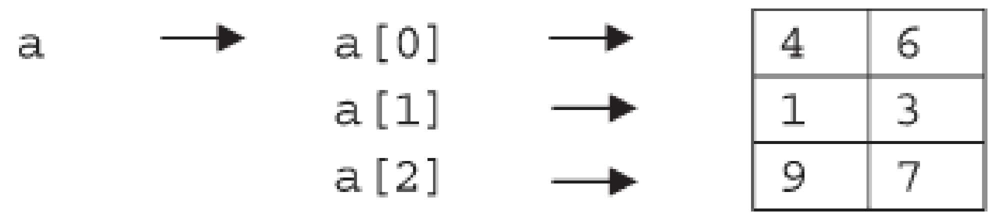
```
int a[3][2] = { { 4, 6 },
                { 1, 3 },
                { 9, 7 } } ;
```



10.2 Pointers and Arrays (Further)

- a[0], a[1] and a[2] are pointers (data type is int*) and a is a pointer to a pointer (data type is int **).
  - a[0]+1 is the address of the second element in the first row. *(a[0]+1) is a[0][1], which is 6.
  - *(a[1]+1) is a[1][1], which is 3.
  - a[2]+1 is the address of the second element in the third row. *(a[2]+1) is a[2][1], which is 7.
  - a[1]+1 is the address of the second element in the second row.

```
int a[3][2] = { { 4, 6 },
                { 1, 3 },
                { 9, 7 } } ;
```



10.2 Pointers and Arrays (Further)

- *a is the same as a[0]
- *(a+1) is the same as a[1]
- *(a+2) is the same as a[2]

```
1. a[0][0] is *a[0] is *(*a) or **a
2. a[1][0] is *a[1] is *(*(a+1))
3. a[2][0] is *a[2] is *(*(a+2))
4. a[0][1] is *(a[0]+1) is *(*a+1)
5. a[1][1] is *(a[1]+1) is *(*(a+1)+1)
6. a[2][1] is *(a[2]+1) is *(*(a+2)+1)
```

10.3 Pointers to class/struct

- Define a pointer to a variable of a type defined by *struct* or *class*
  - The general format for defining **a pointer to a structure**

```
struct tag_name* variable_name ;
```

    - *tag_name* is the structure tag
    - *variable_name* is the name of the pointer variable
    - Example

```
struct student_rec  // Structure template.
{
  int number ;
  float scores[5] ;
} ;

struct student_rec student ; // Define a structure variable.
```

10.3 Pointers to class/struct

- Define a pointer ptr to the student_rec structure

```
struct student_rec *ptr ;
```

Note that it is the address of the structure variable *student* and not the address of the structure tag *student_rec* that is assigned to *ptr*

- A value can be assigned to ptr by using the (get) **address operator** &

```
ptr = &student ;
```

- The members of a structure variable can be referenced by using the **dereference operator** *.

```
(*ptr).number
```

The parentheses are necessary, because the selection operator. has a higher priority than the dereference operator *

10.3 Pointers to class/struct

- For **accessing the members of a structure**, the arrow notation **->** (‘-’ and ‘>’ together) can be used in place of the dot notation

```
ptr -> number    ==    (*ptr).number
```

  - The expression ***ptr->number*** reads as “the member number of the structure **pointed by** *ptr*”.

10.3 Pointers to class/struct

- Defining a pointer to a class object is similar to defining a pointer to a structure variable.

```
class_name* variable_name ;
```

  - *class_name* is the name of the class
  - *variable_name* is the name of the pointer variable.

```
struct tag_name* variable_name ;
```

10.3 Pointers to class/struct

```
3   #include <iostream>
4   #include <iomanip>
5   #include "bank_ac.h"
6   #include "bank_ac.cpp"
7   using namespace std ;
8
9   main()
10  {
11    bank_account ac ;           // ac is a bank_account object.
12    bank_account* ac_ptr ;      // ac_ptr is a pointer to a bank_account.
13
14    ac_ptr = &ac ;      // ac_ptr contains the address of the object ac.
15    ac_ptr -> deposit( 100 ) ;
16    ac_ptr -> display_balance() ;
17  }
```

- Line 12 defines *ac_ptr* as a pointer to a *bank_account* object
- Line 14 assigns the address of the *bank_account* object ac to *ac_ptr.*

10.3 Pointers to class/struct

- The public members of a class object may be accessed by using the dereference operator *.

```
(*ac_ptr).deposit( 100 )
```

  - The first pair of parentheses are necessary, because the selection operator. has a higher priority than the dereference operator *.
  - The arrow notation -> can be used in place of the dot notation

    -> is more convenient and common.

```
ac_ptr -> deposit( 100 ) ;    =    (*ac_ptr).deposit( 100 ) ;
```

10.4 Pointers as function arguments

- Program Example: uses pointers in place of references

```
6  void swap_vals( float* val1, float* val2 ) ;
10 main()
11 {
12   float num1, num2 ;
13
14   cout << "Please enter two numbers: " ;
15   cin >> num1 ;
16   cin >> num2 ;
17   // Swap values around so that the smallest is in num1
18   if ( num1 > num2 )
19     swap_vals( &num1, &num2 ) ;
20   cout << "The numbers in order are "
21         << num1 << " and " << num2 << endl ;
22 }
23
24 void swap_vals( float* ptr1, float* ptr2 )
25 {
26   float temp = *ptr1 ;
28   *ptr1 = *ptr2 ;
29   *ptr2 = temp ;
30 }
```

Pointer arguments

- Line 19 passes the addresses of the two floating-point variables num1 and num2 to the function swap_vals().
- These addresses are received by the parameters ptr1 and ptr2, declared as pointers to floats in the function header on line 24.
- Line 26 stores the value of *num1* (= *ptr1) in the variable *temp*
- Line 28 is equivalent to *num1 = num2* ;
- Line 29 assigns the value of temp (12.1) to num2, because *ptr2 is the same as num2.

```
Please enter two numbers: 12.1 6.4
The numbers in order are 6.4 and 12.1
```

10.4 Pointers as function arguments

- It is easier to **use references rather than pointers**
- **&** must be used to **pass the address of the variables** to the function
- Within the function the **dereference operator** * must be used to access the value of each of the numbers.
- Some library functions use pointers as parameters
  - *ctime*(), converting the time in seconds to a character string containing the date and time.

```
char* ctime( const std::time_t* time );
```

10.4 Pointers as function arguments

- Program Example

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

main()
{
  time_t current_time ; // Define a variable of type time_t.

  current_time = time( 0 ) ; // Get the current time in seconds.
  // Display the current date and time as a text string.
  cout << "Current date and time: " << ctime( &current_time ) << endl ;
}
```

10.4 Pointers as function arguments

(Lecture 10)

```
    swap_vals( &num1, &num2 ) ;
  cout << "The numbers in order are "
       << num1 << " and " << num2 << endl ;
}

void swap_vals( float* ptr1, float* ptr2 )
```

Pointer arguments

(Lecture 8 Functions – 8.3)

```
    if( num1 > num2 )
        swap_vals( num1 , num2 );
    cout << "The numbers in order are"
         << num1 << "and" << num2 << endl;
}

void swap_vals ( float& val1, float& val2 );
{
    float temp = val1;

    val1 = val2 ;
    val2 = temp ;
}
```

Arguments passed by reference

local variable

- The variable temp is a *local variable to the function* swap_vals().
- Local variables are known only within the function where they are defined.

	Pointer	**Reference**
Declare	**int*** myPtr1 **Float*** myPtr2 **char*** myPtr3	void myFunction(**int&** myVariable){} void swap_vals(**float&** val1, **float&** val2){}

Access	*myPtr1
	*myPtr2
	*myPtr3
		int x; float y;
		myPtr1 = &x;
		myPtr2 = &y;

pass by reference

dereferencing

getting address

10.5 Dynamic memory allocation

- Problem:
  - When defining an array, the **number of elements** in the array must be **specified in advance** of the program execution. Sometimes, either all the elements specified are not used or more elements than were originally anticipated are required.

- C++ has the ability to allocate memory while a program is executing
  - Using the **memory allocation operator** **new**.

10.5 Dynamic memory allocation

- **Allocating memory dynamically for an array**
  - The *new* **memory allocation operator** can be used to allocate a contiguous block of memory for an array of any data type, whether the data type is built-in or is a user-defined structure or class.

```
pointer = new data_type[ size ] ;
```

- *pointer* is a pointer to the allocated memory
- *data_type* is the data type of the array
- *size* is the number of elements in the array

```
    // Allocate memory for 10 integers.
int* int_ptr = new int[10] ;
```

```
delete[] int_ptr ; // Free the allocated memory.
```

10.5 Dynamic memory allocation

- When **allocating memory for an array of class objects**, there must be a **default constructor for the class** so that the elements of the array get initialized.

```
bank_account*    ac_ptr ;

ac_ptr = new back_account[5] ;
```

Default constructor of the “bank_account” class being called five times

10.5 Dynamic memory allocation (Further)

- **Allocating memory for multi-dimensional arrays**
  - In C++, multi-dimensional arrays are implemented as ‘**arrays of arrays**’.

```
main()
{
  int no_of_rows, no_of_cols ;
  int i, j ;
  float **data ;

  cout<< "Number of rows: " ;
  cin >> no_of_rows ;
  cout<< "Number of columns: " ;
  cin >> no_of_cols ;
```

10.5 Dynamic memory allocation (Further)

```
18    // Allocate requested storage:
19
20    // (a) allocate storage for the rows.
21    data = new float* [no_of_rows] ;
22
23    // (b) allocate storage for each column.
24    for ( j = 0 ; j < no_of_rows; j++ )
25      data[j] = new float[no_of_cols] ;
26
27    // Place some values in the array.
28    for ( i = 0 ; i < no_of_rows ; i++ )
29      for ( j = 0 ; j < no_of_cols ; j++ )
30        data[i][j] = i * 10 + j ;
```

```
32    // Display elements of the array.
33    for ( i = 0 ; i < no_of_rows ; i++ )
34    {
35      for ( j = 0 ; j < no_of_cols ; j++ )
36        cout << data[i][j] << ' ' ;
37      cout << endl ;
38    }
39
40    // Free the allocated storage:
42    // (a) delete the columns.
43    for ( i = 0 ; i < no_of_rows ; i++ )
44      delete[] data[i] ;
45
46    // (b) delete the rows.
47    delete[] data ;
48 }
```

- Lines 44 and 47 free the memory allocated in lines 21 and 25 separately.
- For each pointer returned from new in lines 21 and 25 there is a corresponding call to delete with that pointer in lines 47 and 44.

10.5 Dynamic memory allocation (Further)

- **Out of memory error**
  - It was assumed that the memory requested with new was allocated, regardless of whether memory was available or not.
  - C++ handles **insufficient memory errors** produced by new by calling a function specified in ***set_new_handler().***

10.5 Dynamic memory allocation (Further)

```
6  void out_of_memory() ;
7
8  main()
9  {
10   const int ONE_MB = 1024 * 1024 ;
11   int memory_allocated = 0 ;
12   int* ptr ;
14   set_new_handler( out_of_memory ) ;
16   for ( ; ; )  // Infinite loop.
17   {
18     ptr = new int[ONE_MB] ; // Allocate memory in 1MB blocks.
19     memory_allocated++ ;
20     cout << memory_allocated << " MB allocated..." << endl ;
21   }
22 }
24 void out_of_memory()
25 {
26   cerr << "Error: Out of memory" << endl ;
27   exit( 1 ) ;
28 }
```

- Line 27 terminates the program and exits to the operating system with a status code of 1.
- A non-zero status code is usually used to indicate an abnormal exit from a program.

10.5 Dynamic memory allocation (Further)

- The function *out_of_memory()* inserts an error message into the stream *cerr* rather than into *cout*.
  - The stream *cerr* is typically used for error messages while *cout* is used for displaying the results of a program.
  - Like *cout, cerr* is, by default, connected to the screen.
  - The output for *cout* is often redirected to a device other than the screen (e.g. a disk file). In this case *cout* is unsuitable for error messages that may require immediate attention, so *cerr* is used instead.
- Redirecting output streams to different devices is done by the operating system commands, not by C++.

HOMEWORK

Homework 10

- 1. Given the following:

```
int* i_ptr ;
float* f_ptr ;
int i = 1, k = 2 ;
float f = 10.0 ;
```

which of these statements are valid?

(a) `i_ptr = &i ;`	(b) `f_ptr = &f ;`	(c) `f_ptr = f ;`
(d) `f_ptr = &i ;`	(e) `k = *i ;`	(f) `k = *i_ptr ;`
(g) `i_ptr = &k ;`	(h) `*i_ptr = 5 ;`	(i) `i_ptr = &5 ;`

Homework 10

- 2. What does this program segment display?

```
int a, b ;
int* p1, *p2 ;
a = 1 ;
b = 2 ;
p1 = &a ;
p2 = &b ;
b = *p1 ;
cout << a << ' ' << b << endl ;
cout << *p1 << ' ' << *p2 << endl ;
*p1 = 15 ;
cout << a << ' ' << b << endl ;
*p1 -= 3 ;
cout << a << ' ' << b << endl ;
*p2 = *p1 ;
cout << a << ' ' << b << endl ;
(*p1)++ ;
cout << a << ' ' << *p2 << endl ;
p1 = p2 ;
*p1 = 50 ;
cout << a << ' ' << b << endl ;
```

Homework 10

- 3. What is the output from the following?

```
string* sp1 = new string ( "asdfghjk" ) ;
string* sp2 ;
string s = *sp1 ;
string& r = s ; // r is a reference to s.
sp2 = &s ;      // sp2 contains the address of s.
s.at( 0 ) = 'A' ;
sp1 -> erase( 2, 3 ) ;
cout << s << endl ;
cout << r << endl ;
cout << *sp1 << endl ;
cout << *sp2 << endl ;
```

Homework 10

- 4. What does this program segment do?

```
int a[5] ;
int *p ;
for ( int i = 0 ; i < 5 ; i++ )
  cin >> *(a+i) ;
for ( p = a ; p < a+5 ; p++ )
  cout << ' ' << *p ;
```

Homework 10

- 5. What is the value of ＊p, ＊p+4 and ＊(p+4) in each of the following?

(a)
```
int one_d[] = {1,3,4,5,-1} ;
int *p ;
p = one_d ;
```
(b)
```
float f[] = { 1.25, 11.0, 9.5, 3.5, 6.5, 1.0 } ;
float *p ;
p = f ;
```
(c)
```
int two_d[3][6] = { {1, 5, 0,  9,11, -4},
                    {3, 9, 4, 6, 10, 123},

                    {11, 7, 4, -10, 19, 15} } ;
int *p ;
p = two_d[1] ;
```

Homework 10

- 6. Given the following definitions:

```
int numbers[10] = { 1,7,8,2 } ;
int *ptr = numbers ;
```

what is in the array numbers after each of the following?

(a) `*(ptr+4) = 10 ;`

(b) `*ptr-- ;`

(c) `*(ptr+3) = *(ptr+9) ;`

(d) `ptr++ ;`

(e) `*ptr = 0 ;`

(f) `*(numbers+1) = 1 ;`