AMA2222 Principles of Programming

Leung Man Kin, Adam
Instructor

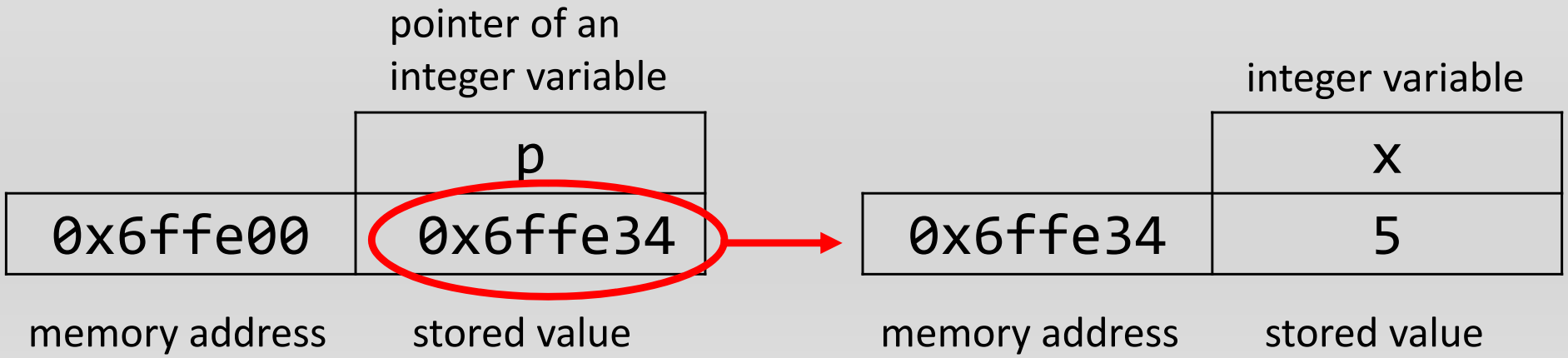
adam.leung@polyu.edu.hk
TU720

Chapter 10: Pointer
Pointer declaration, pointer and array,
passing and returning pointer,
dynamic memory allocation

Introduction to pointer

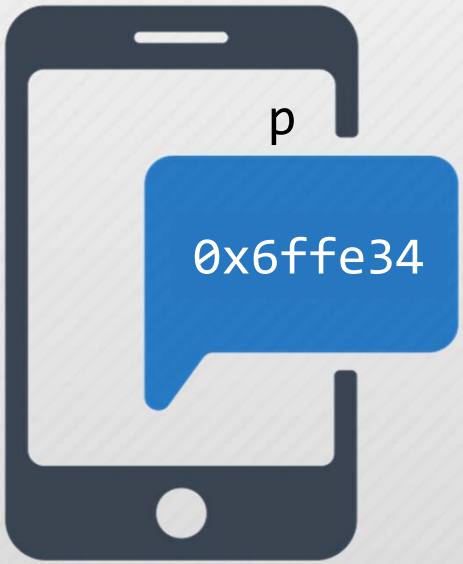
Pointer is a very special feature in C++. A pointer is also a variable, which stores an **memory address**. Such a memory address can refers to a variable, an array or an object. A pointer with the value 0 or NULL points to nothing

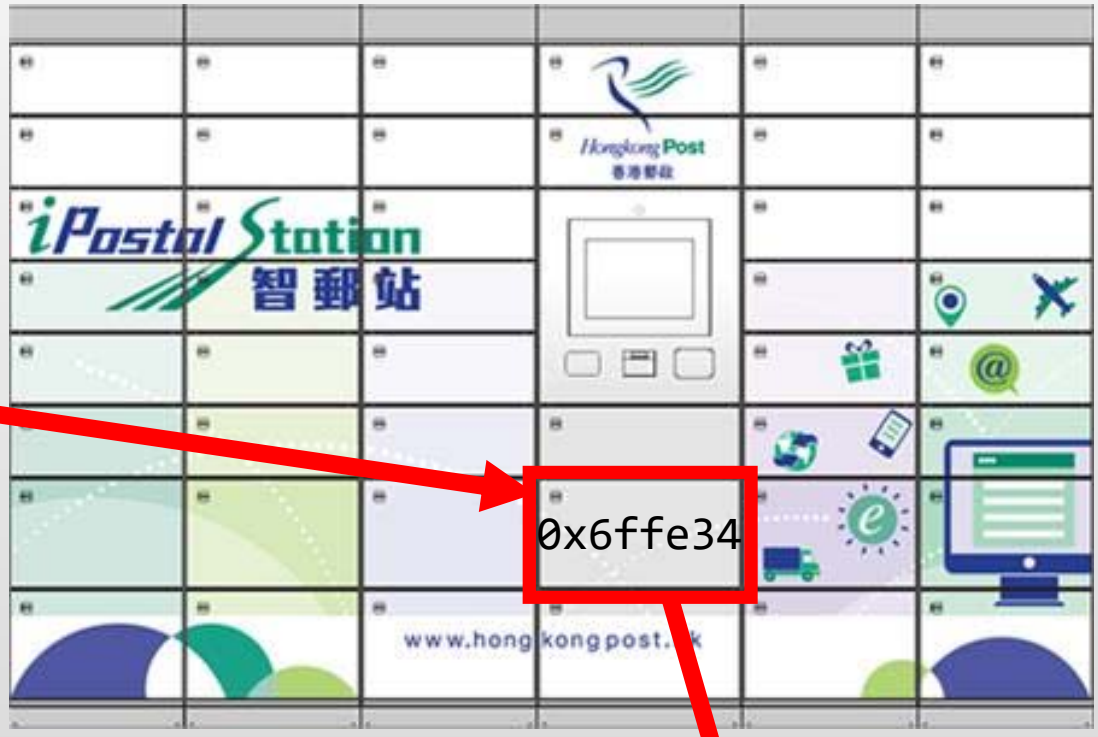
For example, **x** is an integer variable with some stored value, let's say 5. **p** is the pointer of **x**, then p would contain the memory address of **x**, i.e. where it is store in the memory space.



An analogy of how pointer works:

SMS message
contains an address

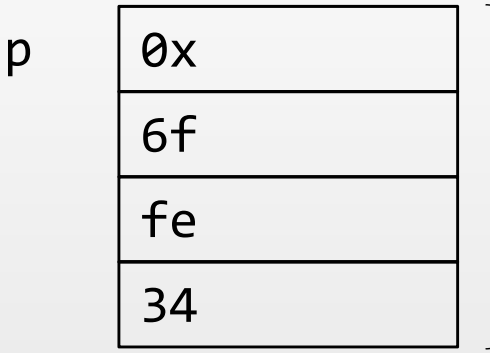




the address refers
to a position at the
postal station

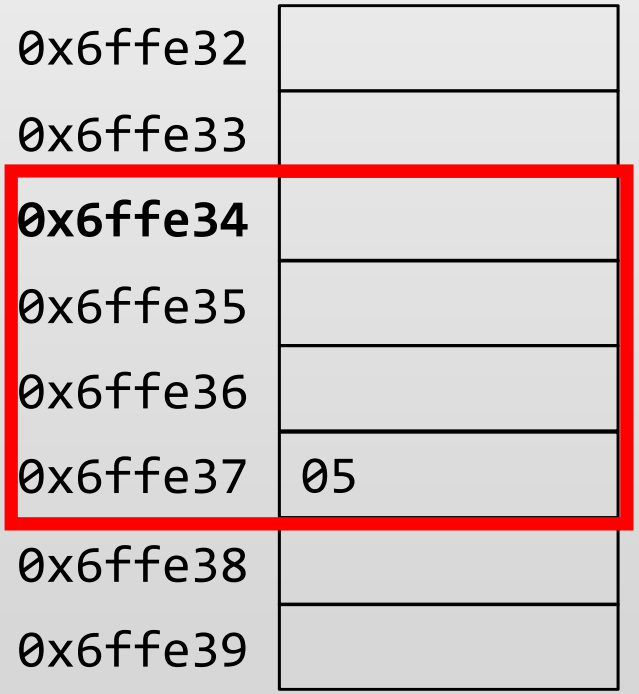
x

with the right address,
you can retrieve the
parcel stored in the
postal station



"0x" indicates hexadecimal form (base-16).

p stores the address, 0x6ffe34, of **x**



address from 0x6ffe34 to 0x6ffe37 stores the value of integer variable x as the data type `int` is 4 bytes (32 bit). Other types got different byte sizes!

Notice that each byte can store two hexadecimal digits, that means:

$$16 \times 16 = 2^4 \times 2^4 = 2^8$$

To assign a pointer to a variable, or to retrieve the value of the variable pointed by a pointer, we can use the reference operator & and the dereference operator *.

Initialize an int variable called x with value 5:

```
int x = 5;
```

Defines a pointer p that points to an int variable:

```
int* p;
```

Assign p to the memory address of x:

```
p = &x;
```

To retrieve the value stored in the memory address indicated by pointed p, then assign it to another int variable num:

```
int num = *p;
```

Example program 10.1

In this simple program we will first understand the relationship between the pointer p and the integer variable x, and how to relate them using the * and & symbols.

```
 1 // 10.1 Defining pointer
 2 #include <iostream>
 3 using namespace std;
 4 int main()
 5 {
 6           int x = 5;
 7           int* p;
 8           p = &x;
 9           cout << "The address of x is: " << p << endl;
10           cout << "The value pointed by p is: " << *p;
11           return 0;
12 }
```

Output: (notice that the address is machine dependent)

```
The address of x is: 0x6ffe34
The data stored in p is: 5
```

Classwork 10.1: Suppose

	pY			y
0x22fef8	???		0x22fafc	5

Determine outputs and the value stored in pY after the following:

```
int y=5;
int *pY;
pY = &y;
cout << pY << endl;
cout << *pY << endl;
cout << y << endl;
cout << &y << endl;
cout << &pY << endl;
```

	output	meaning
pY	0x22fafc	address of y
*pY	5	value stored in the address of y
y	5	value of y
&y	0x22fafc	address of y
&pY	0x22fef8	address of address of y

Classwork 10.2 Suppose p is 0x6ffe34, evaluate the output.

```
int count = 5;
int* p = &count;
p++;
cout << "The next address is: " << p << endl;
p++;
cout << "The next address is: " << p << endl;
```

Very tricky! Here the ++ doesn't mean plus 1, instead it jumps to the next memory address. Since p is pointer of an integer variable with 4 bytes, the next address increases by 4.

```
The next address is: 0x6ffe38
The next address is: 0x6ffe3c
```

Pointer and array

An array without brackets and subscript is actually a pointer which points to the beginning element of the array. An array name is a const variable which always points to a fixed memory location.

A pointer may be incremented (++) or decremented (--), an integer may be added (+ or +=) or subtracted (- or -=) to a pointer.

The pointer is not simply incremented or decremented by that integer, but instead, by that integer times the size of the object to which the pointer refers. The number of bytes depends on the object's data type.

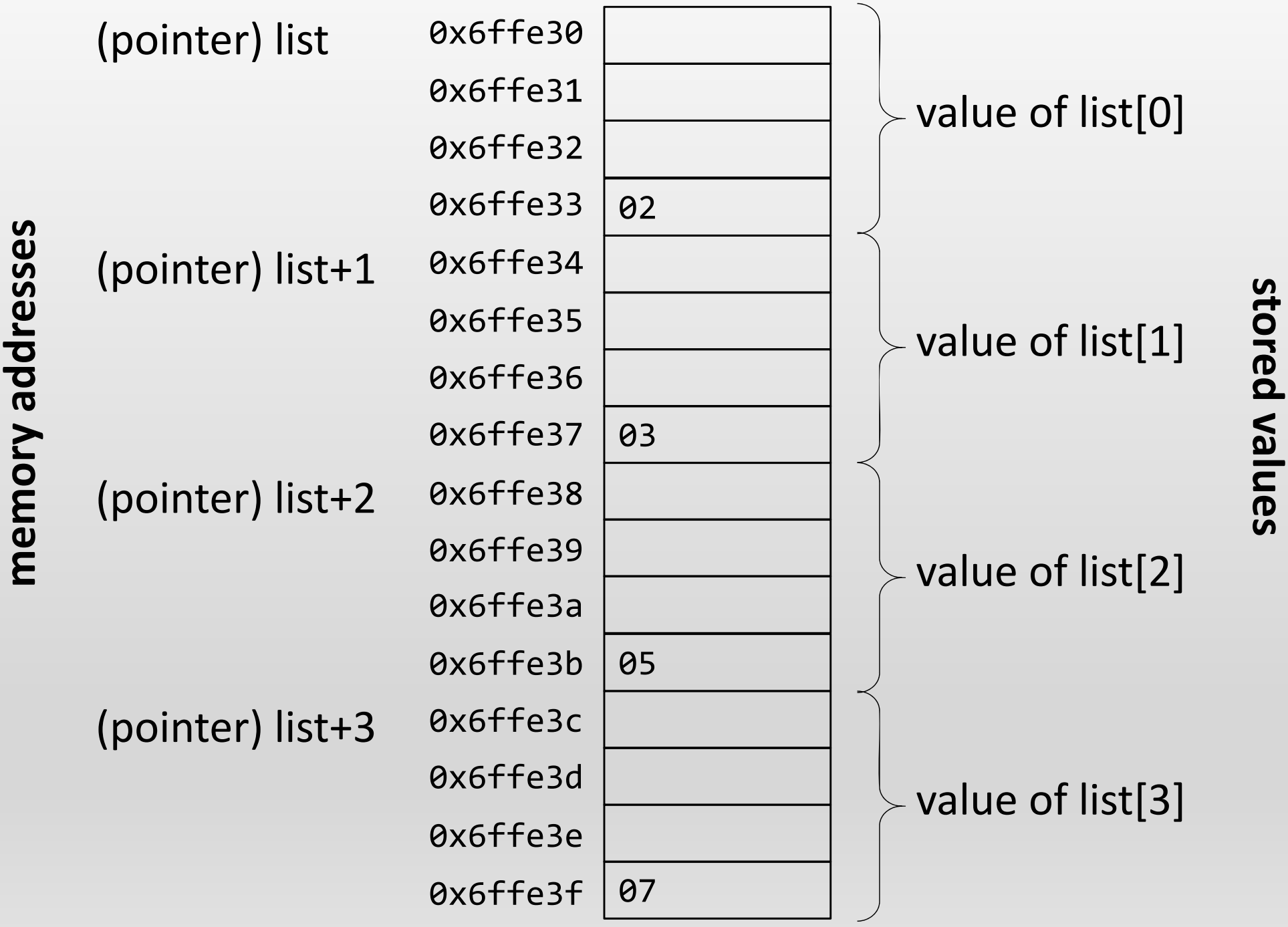
Example 10.2 We will compare two different methods to print out each elements of the same array.

```
// 10.2a Pointer and Array
#include <iostream>
using namespace std;
int main()
{
        int list[4] = {2, 3, 5, 7};
        for (int i=0; i<4; i++)
        {
        cout << list + i << endl;
        cout << *(list + i) << endl;
        }
        return 0;
}
```

```
0x6ffe30    address of list[0]
2           value of list[0]
0x6ffe34    address of list[1]
3           value of list[1]
0x6ffe38    address of list[2]
5           value of list[2]
0x6ffe3c    address of list[3]
7           value of list[3]
```

```
// 10.2b Pointer and Array
#include <iostream>
using namespace std;
int main()
{
        int list[4] = {2, 3, 5, 7};
        for (int i=0; i<4; i++)
        {
        int* p = &list[i];
        cout << p << endl;
        cout << *p << endl;
        }
        return 0;
}
```

```
0x6ffe20
2
0x6ffe24
3
0x6ffe28
5
0x6ffe2c
7
```


memory addresses
(pointer) list
(pointer) list+1
(pointer) list+2
(pointer) list+3
0x6ffe30
0x6ffe31
0x6ffe32
0x6ffe33
02
0x6ffe34
0x6ffe35
0x6ffe36
0x6ffe37
03
0x6ffe38
0x6ffe39
0x6ffe3a
0x6ffe3b
05
0x6ffe3c
0x6ffe3d
0x6ffe3e
0x6ffe3f
07
value of list[0]
value of list[1]
value of list[2]
value of list[3]
stored values

Example program 10.3 Printing an array.

Both programs below contains a function that prints an array.

```
 1 // 10.3a Printing an array
 2 #include <iostream>
 3 using namespace std;
 4 void print(int list[], int size)
 5 {
 6          for (int i=0; i<size; i++)
 7          cout << list[i] << " ";
 8 }
 9 int main()
10 {
11          int list[6]
12          = {11, 12, 13, 14, 15, 16};
13          print(list, 6);
14          return 0;
15 }
```

```
 1 // 10.3b Printing an array
 2 #include <iostream>
 3 using namespace std;
 4 void print(int* list, int size)
 5 {
 6          for (int i=0; i<size; i++)
 7          cout << *(list+i) << " ";
 8 }
 9 int main()
10 {
11          int list[6]
12          = {11, 12, 13, 14, 15, 16};
13          print(list, 6);
14          return 0;
15 }
```

In 8.3a, we input the integer array `list[]` into the function.

In 8.3b, we input the integer pointer `list` into the function.

Apart from using the array name directly as a pointer, we might also use the following functions to an array `arr[]` :

`begin(arr)`	give the memory address of the beginning element
`end(arr)`	give the memory address after the last element
`sizeof(arr)`	give the total memory size (in bytes) of the whole array
`sizeof(arr[0])`	give the memory size (in bytes) of the beginning element

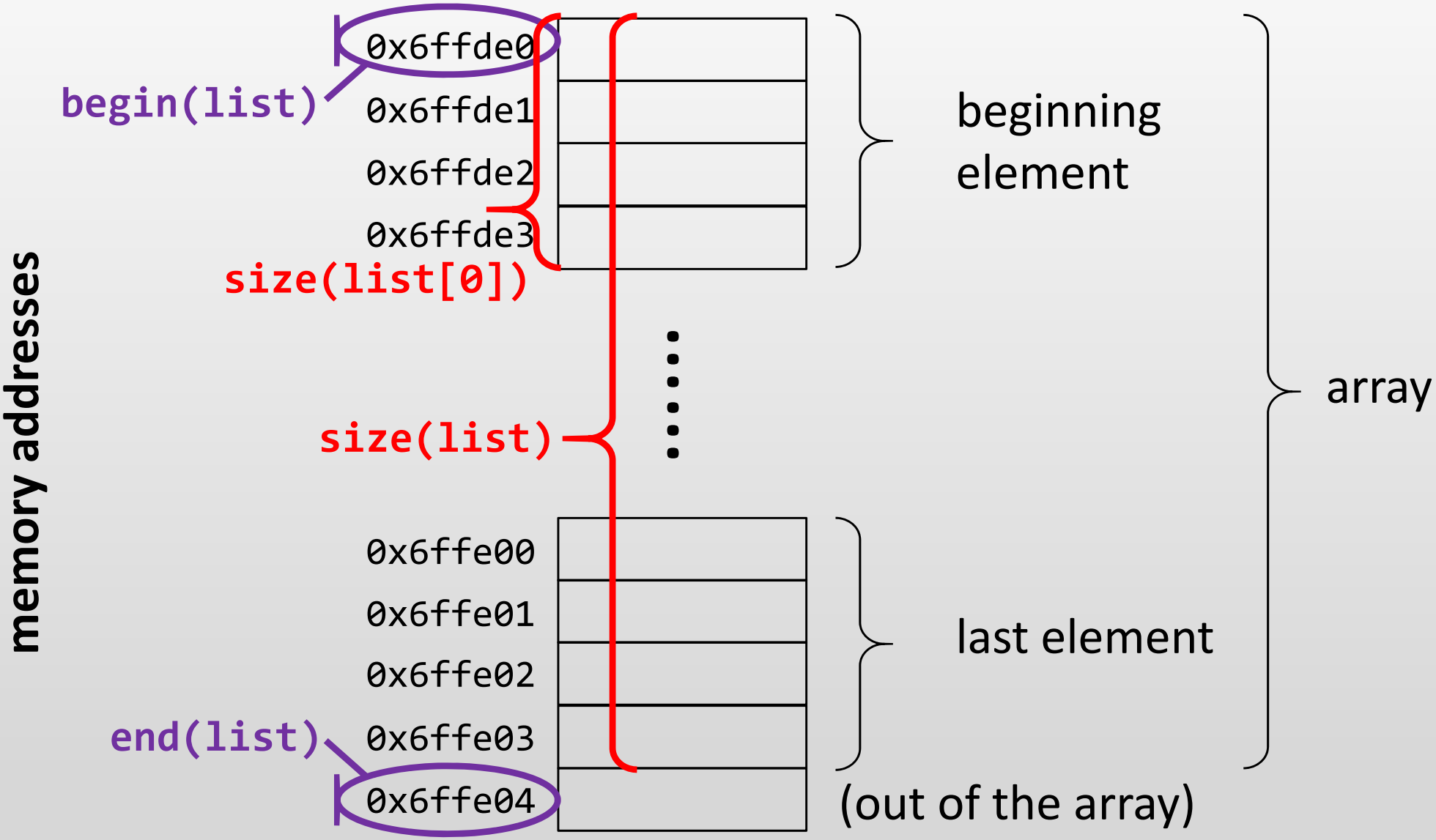
Hence the number of elements of an array can be calculated by the total memory size divided by the memory size of a single element:

$$arraySize = \frac{sizeof(arr)}{sizeof(arr[0])}$$

Or equivalently, the number of elements of an array can be calculated by taking difference of the address after the end and at the beginning:

$$arraySize = end(arr) - begin(arr)$$

Notice that the result obtained by the subtraction between two memory addresses is automatically represented in terms of the byte size of a single element. Therefore We do not need to take further division.


memory addresses
0x6ffde0
0x6ffde1
0x6ffde2
0x6ffde3
begin(list)
size(list[0])
size(list)
beginning element
array
0x6ffe00
0x6ffe01
0x6ffe02
0x6ffe03
0x6ffe04
end(list)
last element
(out of the array)

Example program 10.4 We will examine the relationship between memory address, total memory size, element memory size and number of elements of an array.

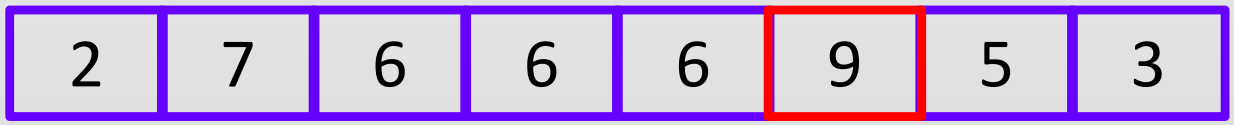
```
address at the beginning: 0x6ffde0
address after the end: 0x6ffe04
total memory size of the array: 36
memory size of an element: 4
number of elements of the array: 9
number of elements of the array: 9
```

```
1  // 10.4 memory size of an array
2  #include <iostream>
3  using namespace std;
4  int main()
5  {
6           int list[] = {11,22,33,44,55,66,77,88,99};
7           cout << "address at the beginning: ";
8           cout << begin(list) << endl;
9           cout << "address after the end: ";
10          cout << end(list) << endl;
11          cout << "total memory size of the array: ";
12          cout << sizeof(list) << endl;
13          cout << "memory size of an element: ";
14          cout << sizeof(list[0]) << endl;
15          cout << "number of elements of the array: ";
16          cout << sizeof(list)/sizeof(list[0]) << endl;
17          cout << "number of elements of the array: ";
18          cout << (end(list)-begin(list)) << endl;
19          return 0;
20 }
```

Since an array can be regarded as a pointer with its values stored consecutively, we can simplify our coding in array processing. Consider an revision of the divide-and-conquer problem:

Example 10.4 Write a function `maxv( )` which gives the maximum value of a given array of integers. For example:

```
int arr[] = {2, 7, 6, 6, 6, 9, 5, 3};
```

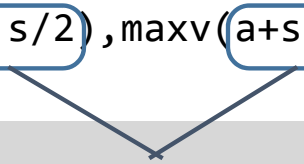


Then `maxv(arr,8)` should give return value 9.

In the previous example, we need to split the array `a[]` into two new arrays `b[]` and `c[]` which is tedious in coding, use up a lot of memory and computation time. Instead, we will keep the same

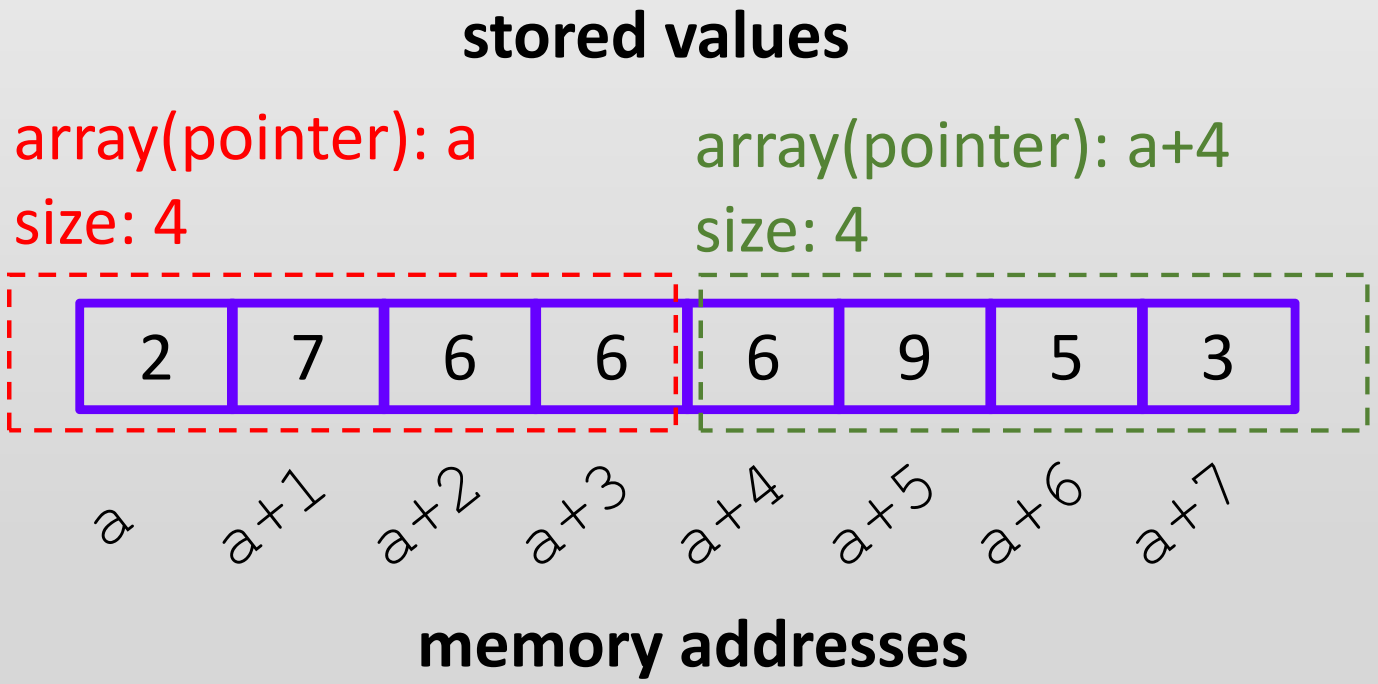
```
//example 8.4 maximum value of array
int maxv(int a[], int s)
{
        if (s==1) return a[0];
        int b[s/2];
        int c[s-s/2];
        for (int i=0; i<s/2; i++)
                b[i]=a[i];
        for (int i=s/2; i<s; i++)
                c[i-s/2]=a[i];
        return max(maxv(b,s/2),maxv(c,s-s/2));
}
```

```
//example 10.5 maximum value of array
int maxv(int a[], int s)
{
        if (s==1) return a[0];
        return max(maxv(a,s/2),maxv(a+s/2,s-s/2));
}
```



we regard the inputs a and a+s/2 as two points

Instead of setting up two new arrays, we simply shift the pointer of the array, which indicates the memory address of the beginning element. Notice that elements of an array are stored in consecutive memory addresses.



Passing and returning pointers

You can pass pointer arguments in a function call. A pointer argument can be passed by value or passed by reference.

A function receiving an address as an argument must define a pointer parameter to receive the address.

In the following example, we will try four functions which are supposed to swap two numbers. Which of them will work?

eg10.6a integer variable, pass by value
eg10.6b integer variable, pass by reference
eg10.6c integer pointer, pass by value
eg10.6d integer pointer, pass by reference

eg10.6a pass two variables by values and swap these values

```
 1 // 10.5a Passing a pointer
 2 #include <iostream>
 3 using namespace std;
 4 void swap1(int n1, int n2)
 5 {
 6           int temp = n1;
 7           n1 = n2;
 8           n2 = temp;
 9 }
10 int main()
11 {
12           int num1 = 5;
13           int num2 = 6;
14           swap1(num1, num2);
15           cout << num1 << endl;
16           cout << num2 << endl;
17           return 0;
18 }
```

Outputs:

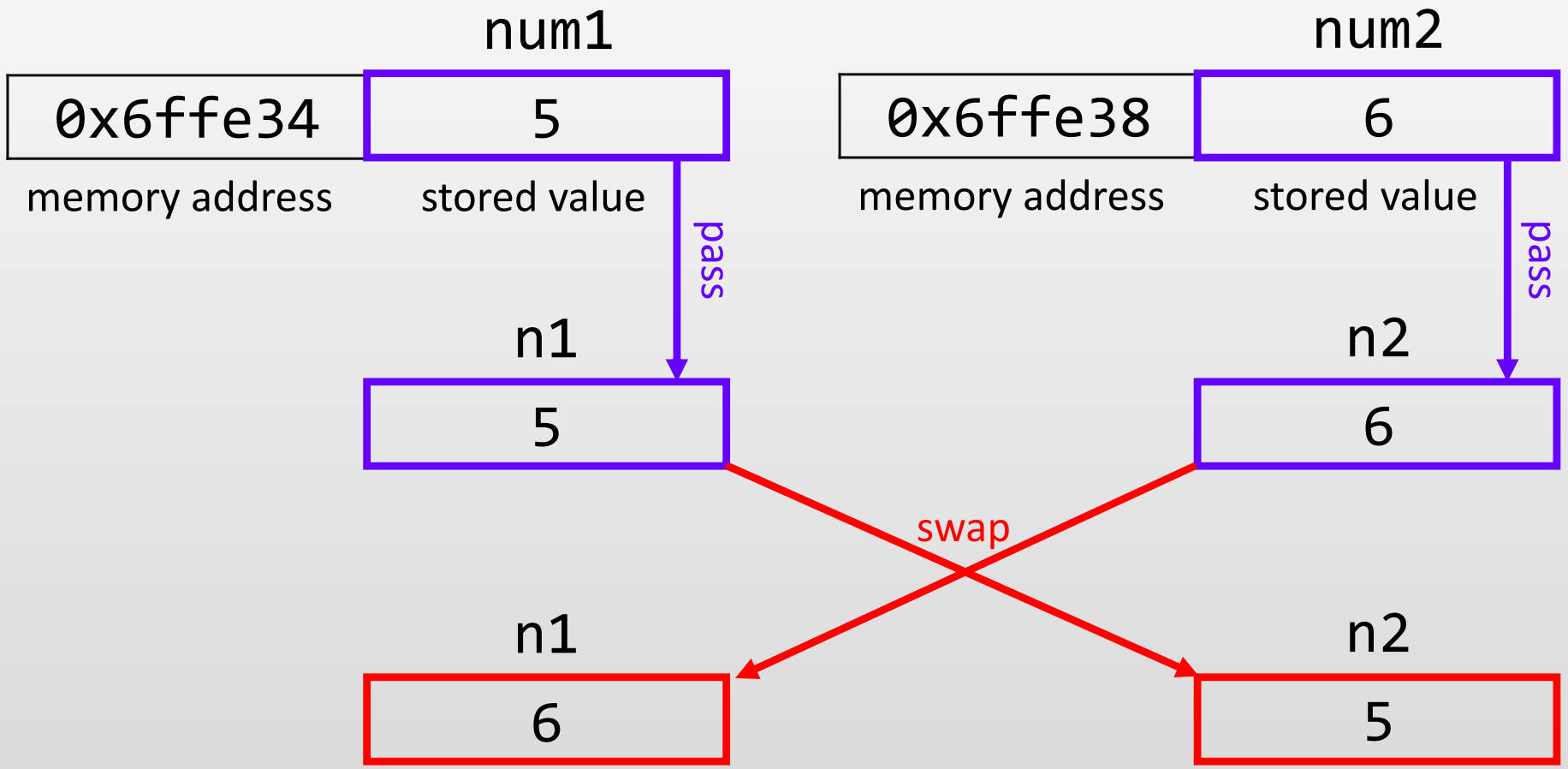
```
5
6
```

eg10.6b pass two variables by reference and swap their values

```
 1 // 10.5b Passing a pointer
 2 #include <iostream>
 3 using namespace std;
 4 void swap2(int &n1, int &n2)
 5 {
 6           int temp = n1;
 7           n1 = n2;
 8           n2 = temp;
 9 }
10 int main()
11 {
12           int num1 = 5;
13           int num2 = 6;
14           swap2(num1, num2);
15           cout << num1 << endl;
16           cout << num2 << endl;
17           return 0;
18 }
```

```
6
5
```

eg10.6a

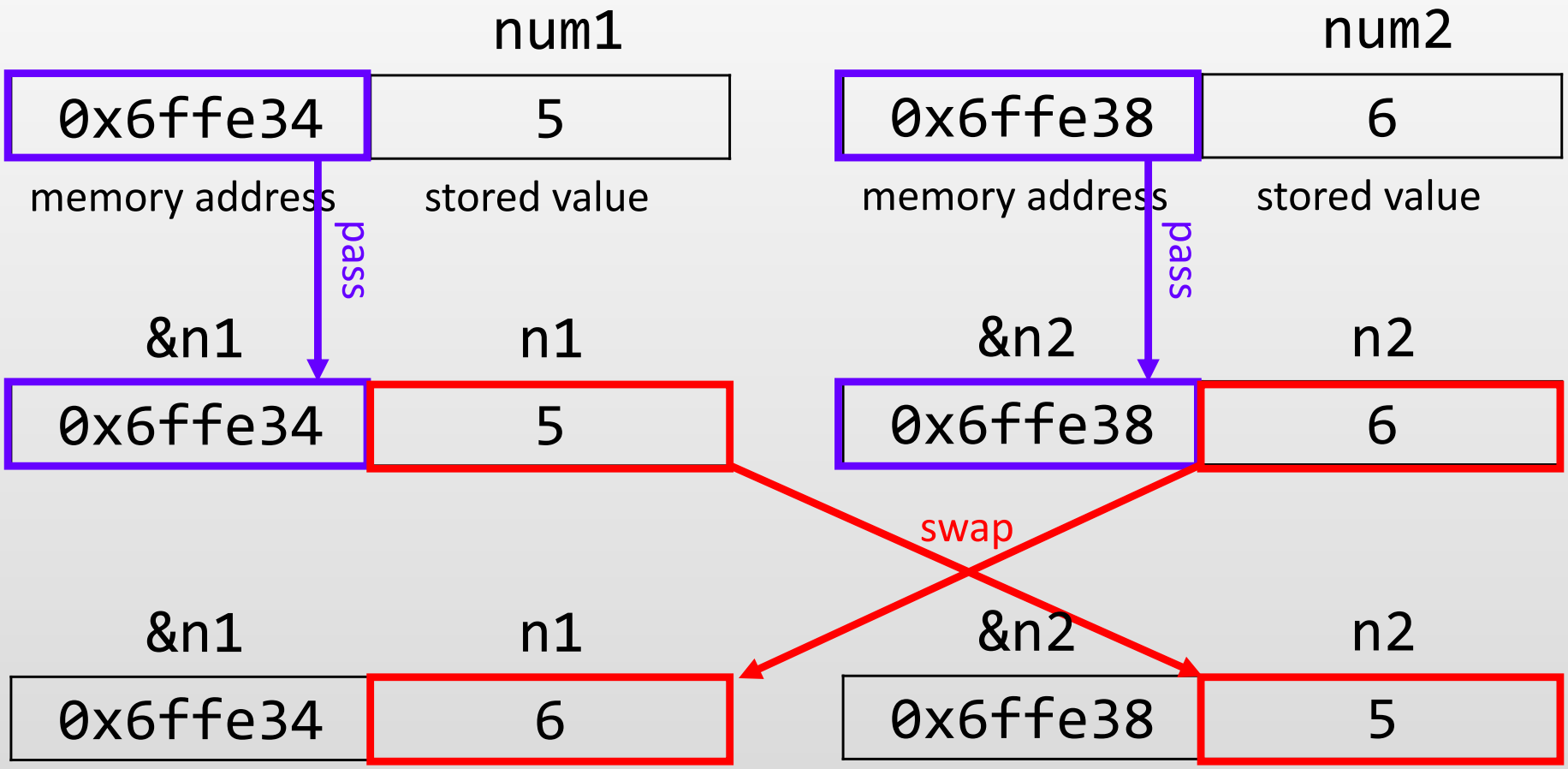


result:

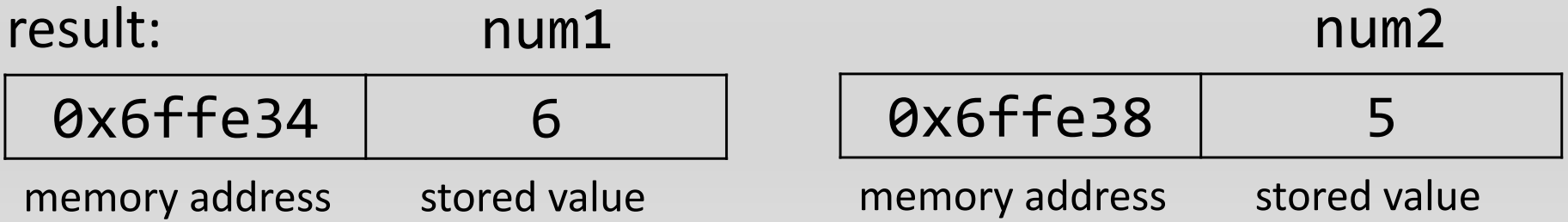
memory address	num1 stored value
0x6ffe34	5

memory address	num2 stored value
0x6ffe38	6

eg10.6b



result:



eg10.6c pass two pointers by value and swap their pointing values

```
 1 // 10.6c Passing a pointer
 2 #include <iostream>
 3 using namespace std;
 4 void swap3(int* p1, int* p2)
 5 {
 6           int temp = *p1;
 7           *p1 = *p2;
 8           *p2 = temp;
 9 }
10 int main()
11 {
12           int num1 = 5;
13           int num2 = 6;
14           int* ptr1 = &num1;
15           int* ptr2 = &num2;
16           swap3(ptr1, ptr2);
17           cout << num1 << endl;
18           cout << num2 << endl;
19           return 0;
20 }
```

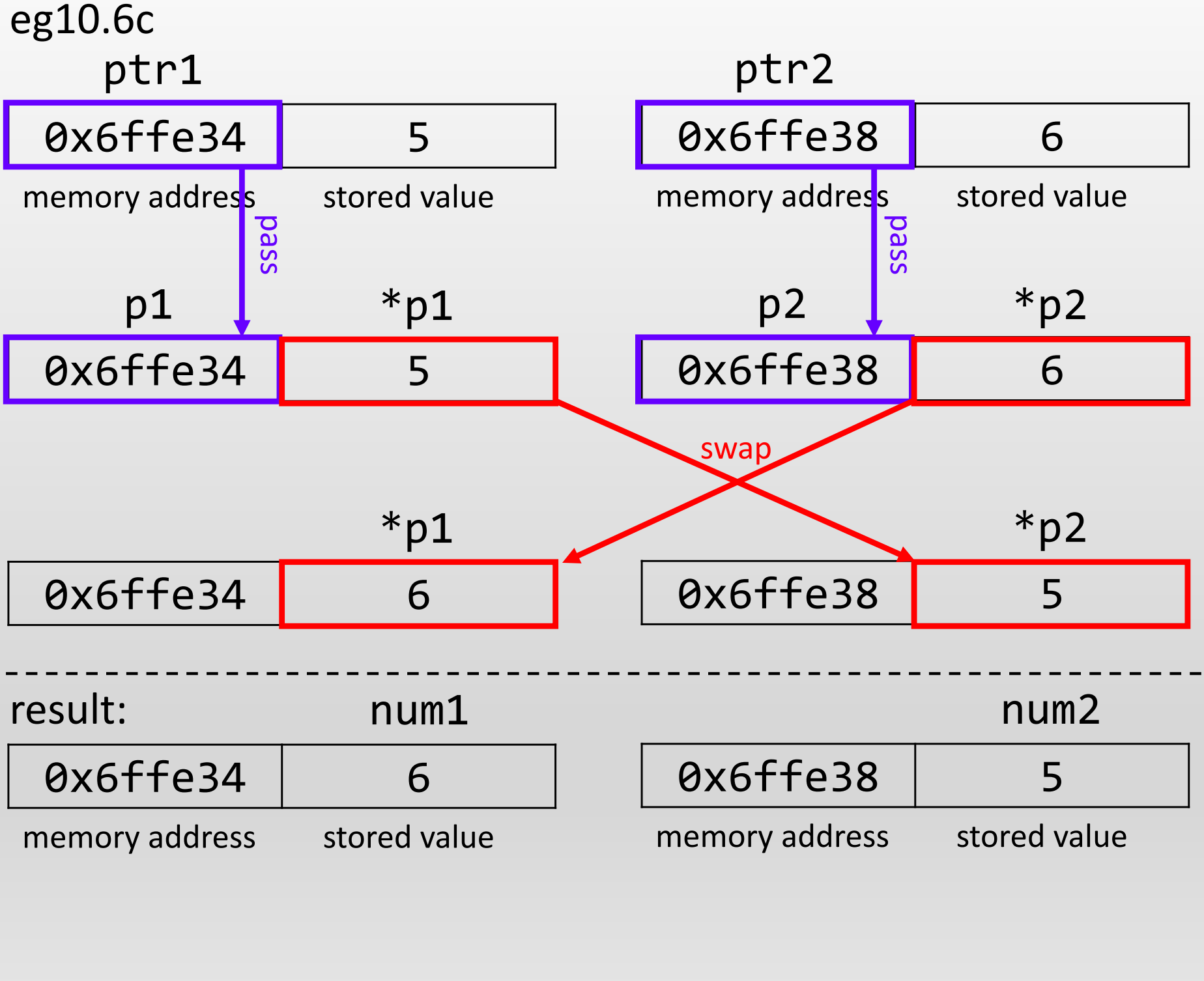
Outputs:

```
6
5
```

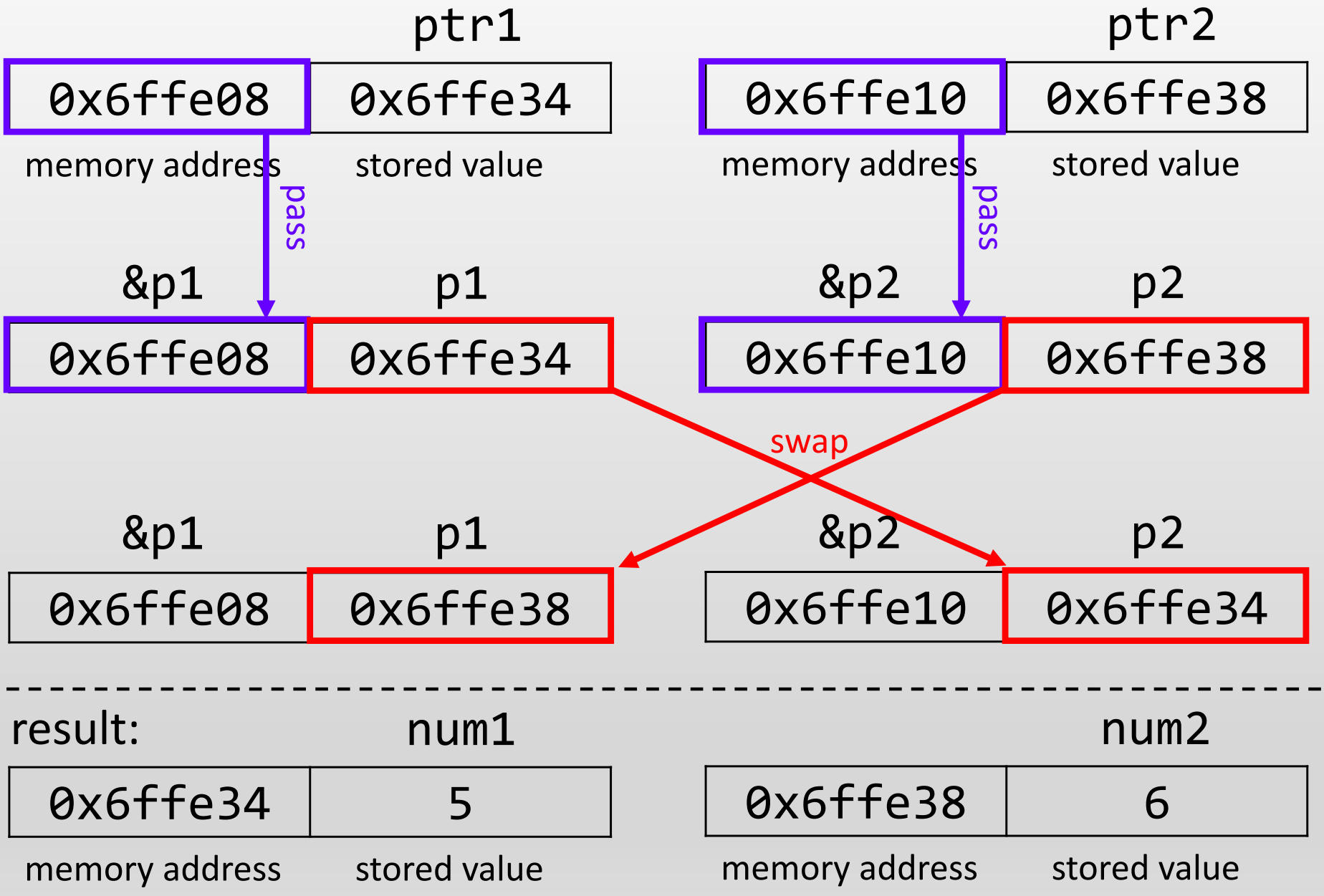
eg10.6d pass two pointers by reference and swap these two pointers

```
 1 // 10.6d Passing a pointer
 2 #include <iostream>
 3 using namespace std;
 4 void swap4(int* &p1, int* &p2)
 5 {
 6           int* temp = p1;
 7           p1 = p2;
 8           p2 = temp;
 9 }
10 int main()
11 {
12           int num1 = 5;
13           int num2 = 6;
14           int* ptr1 = &num1;
15           int* ptr2 = &num2;
16           swap4(ptr1, ptr2);
17           cout << num1 << endl;
18           cout << num2 << endl;
19           return 0;
20 }
```

```
5
6
```


eg10.6c
ptr1
0x6ffe34
5
memory address
stored value
passed
ptr2
0x6ffe38
6
memory address
stored value
passed
p1
*p1
0x6ffe34
5
p2
*p2
0x6ffe38
6
swap
*p1
0x6ffe34
6
*p2
0x6ffe38
5
result:
num1
0x6ffe34
6
memory address
stored value
num2
0x6ffe38
5
memory address
stored value

eg10.6d


ptr1
ptr2
0x6ffe08
0x6ffe34
0x6ffe10
0x6ffe38
memory address
stored value
memory address
stored value
passed
passed
&p1
p1
&p2
p2
0x6ffe08
0x6ffe34
0x6ffe10
0x6ffe38
swap
&p1
p1
&p2
p2
0x6ffe08
0x6ffe38
0x6ffe10
0x6ffe34

result:
num1
num2
0x6ffe34
5
0x6ffe38
6
memory address
stored value
memory address
stored value

Dynamic memory allocation

To allocate memory during runtime, use `new` operator. The memory remains available until you explicitly free it or the program terminates. For example, to allocate memory for an integer, use

```
int* ptr = new int;
```

To free the memory, use delete operator. For example, to free the memory for an integer, use

```
delete ptr;
```

To allocate memory for an integer array, use

```
int* ptr = new int[size];
```

To free the memory, use

```
delete [] ptr;
```

Example 10.7 Suppose we are writing a function which inputs an array and return the reversed array (by its memory address).

```
 1 // 10.7a Reversing an array (wrong)
 2 #include <iostream>
 3 using namespace std;
 4 int* reverse(int* arr, int size)
 5 {
 6         int result[size];
 7         for (int i = 0; i < size; i++)
 8         {
 9                 result[size - i -1] = arr[i];
10         }
11         return result;
12 }
13 void print(int* arr, int size)
14 {
15         for (int i = 0; i < size; i++)
16                 cout << arr[i] << " ";
17 }
18 int main()
19 {
20         int list[] = {1, 2, 3, 4, 5, 6};
21         int* p = reverse(list, 6);
22         print(p, 6);
23         return 0;
24 }
```

The array result is stored in activation record in the call stack, which the memory doesn't persist. When the function returns, this record is thrown away.

We can understand problem and remedy using the concept of dynamic memory.

```
1  // 10.7b Reversing an array (correct)
2  #include <iostream>
3  using namespace std;
4  int* reverse(int* arr, int size)
5  {
6          int* result = new int[size];
7          for (int i = 0; i < size; i++)
8          {
9                  result[size - i -1] = arr[i];
10         }
11         return result;
12 }
13 void print(int* arr, int size)
14 {
15         for (int i = 0; i < size; i++)
16                 cout << arr[i] << " ";
17 }
18 int main()
19 {
20         int list[] = {1, 2, 3, 4, 5, 6};
21         int* p = reverse(list, 6);
22         print(p, 6);
23         return 0;
24 }
```

Here, the modified line `new int[size]` tells the computer to allocate memory space for an `int` array with the specific number of elements, and the address of the array is assigned to list.

The array created using the `new` operator is known as a **dynamic array.**

After completing the task, we can free the memory by:

```
delete [] ptr;
```

```
18 int main()
19 {
20          int list[] = {1, 2, 3, 4, 5, 6};
21          int* p = reverse(list, 6);
22          print(p, 6);
23          delete [] p;
24          return 0;
25 }
```

When a pointer hold address of memory that is freed, the pointer is called a **dangling pointer**. Applying the dereference operator * on a dangling pointer may cause serious errors.

When an allocated memory is not pointed to by any pointer, there is **memory leak**.

After understanding the concept of dynamic memory allocation, we can write a function that returns an array as a pointer. We will illustrate this feature with the following example.

Example 10.8 Write a function which with an array of integers ranging 0 to 10 and its size, then returns a new array which represents the frequency distribution of the input array.

For example, if the array is the following:

```
int list[] = {9,6,8,6,7,5,9,7,6,5,3,7,8,8,1};
```

We can convert this data into a frequency distribution table:

value	0	1	2	3	4	5	6	7	8	9	10
frequency	0	1	0	1	0	2	3	3	3	2	0

Then the result should be like:

```
{0,1,0,1,0,2,3,3,3,2,0}
```

```
1  // 10.8 returning an array
2  #include <iostream>
3  using namespace std;
4  int* frequency(int* arr, int size)
5  {
6          int* result = new int[11];
7          for (int i = 0; i < 11; i++)
8                  result[i] = 0;
9          for (int i = 0; i < size; i++)
10                 result[arr[i]] += 1;
11         return result;
12 }
13 void printfreq(int* arr)
14 {
15         for (int i = 10; i >= 0; i--)
16                 if (arr[i]>0)
17                         cout << i << " : " << arr[i] << endl;
18 }
19 int main()
20 {
21         int list[] = {9,6,8,6,7,5,9,7,6,5,3,7,8,8,1};
22         int* p = frequency(list, 15);
23         printfreq(p);
24         return 0;
25 }
```

Since the integers range from 0 to 10, the array to represent their frequency distribution should have a size of 11. Here we must allocate a new memory address for storing the result as an array.

initialize the frequency to be zeros

make increment to frequency of the corresponding integers in arr

the return value is a pointer

we print out only those integers with positive frequency, in descending order of their values