CSC 2105 Data Structures

Lecture 3: Pointer

Pointers

- Powerful feature of the C++ language
- Essential for construction of interesting data structures

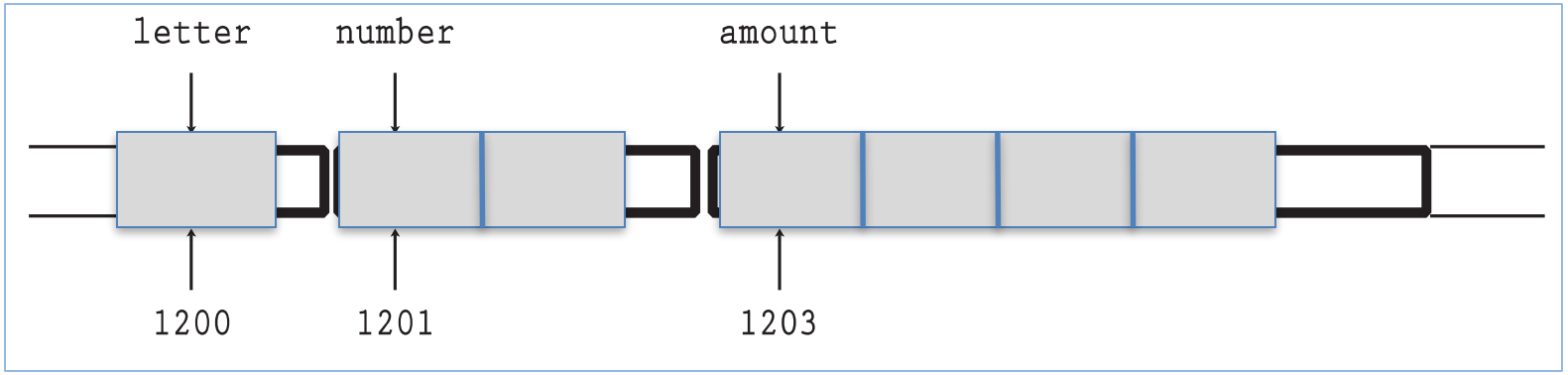
Pointers and the Address Operator &

- The address of a memory location is called a pointer.
- Every variable is assigned a memory location whose address of this memory location can be retrieved using address operator & of C++.

Each byte of memory has a unique address. A variable's address is the address of the first byte allocated to that variable. Suppose that the following variables are defined in a program:

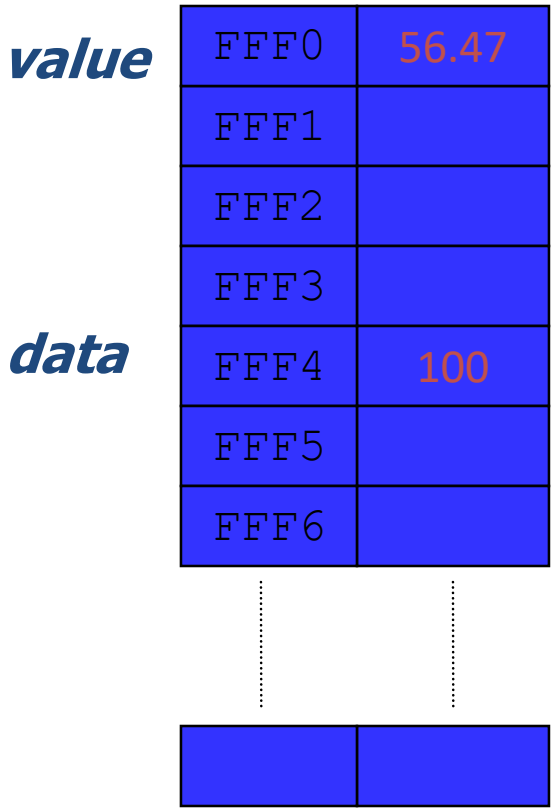
```
char letter;
short number;
float amount;
```

Following figure shows their memory arrangement and addresses:
&letter is 1200 , &number is 1201, &amount is 1203



Example

```
#include <iostream.h>

void main( )
{
    int data = 100;
    float value = 56.47;
    cout << data << &data << endl;
    cout << value << &value << endl;
}
```



Output:

```
100 FFF4
56.47 FFF0
```

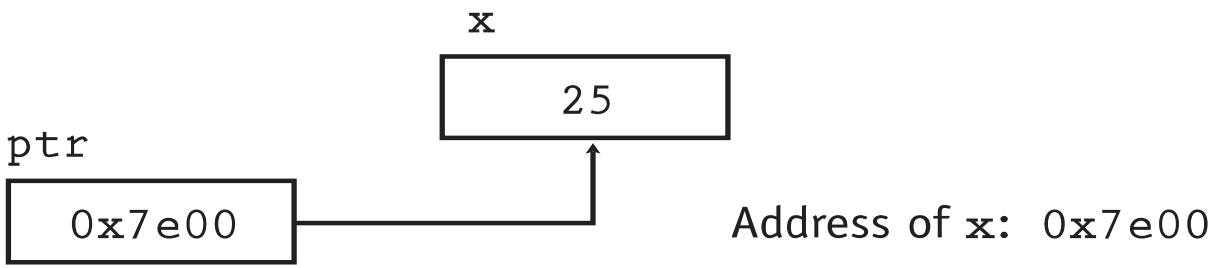
Pointer Variables

A pointer variable is a variable that holds addresses of memory locations.

OR

A variable that stores an address is called a *pointer variable*.

```
int x = 25;   // int variable
int *ptr;     // Pointer variable, can point to an int

ptr = &x;     // Store the address of x in ptr
```



Like any variable or constant, you must declare a pointer before you can use it to store any variable address.

Declaration of Pointer Variables

```
Type *pointerVarName;
```

- The * before the *pointerVarName* indicates that this is a pointer variable, not a regular variable
- The * is not a part of the pointer variable name

Some valid pointer declaration:

```
int     *inp;   // pointer to an integer
char    *chp    // pointer to a character
double  *dp;    // pointer to a double
float   *flp;   // pointer to a float
```

The pointer data type

- A data type for containing an address (hexadecimal number) rather than a data value
- Provides indirect access to values
- The actual data type of the value of all pointers, (integer, float, character) is the same, a long hexadecimal number that represents a memory address.
- The only difference between pointers of different data types is the data type of the variable that the pointer points to.

Declaration of Pointer Variables (Cont ..)

- Whitespace doesn‘t matter and
- each of the following will declare
  - **`ptr`** as a pointer (to a **`float`**) variable and
  - **`data`** as a **`float`** variable

```
float *ptr, data;
float* ptr, data;
float (*ptr), data;
float data, *ptr;
```

Assignment of Pointer Variables

- A pointer variable has to be assigned a valid memory address before it can be used in the program
- Example:

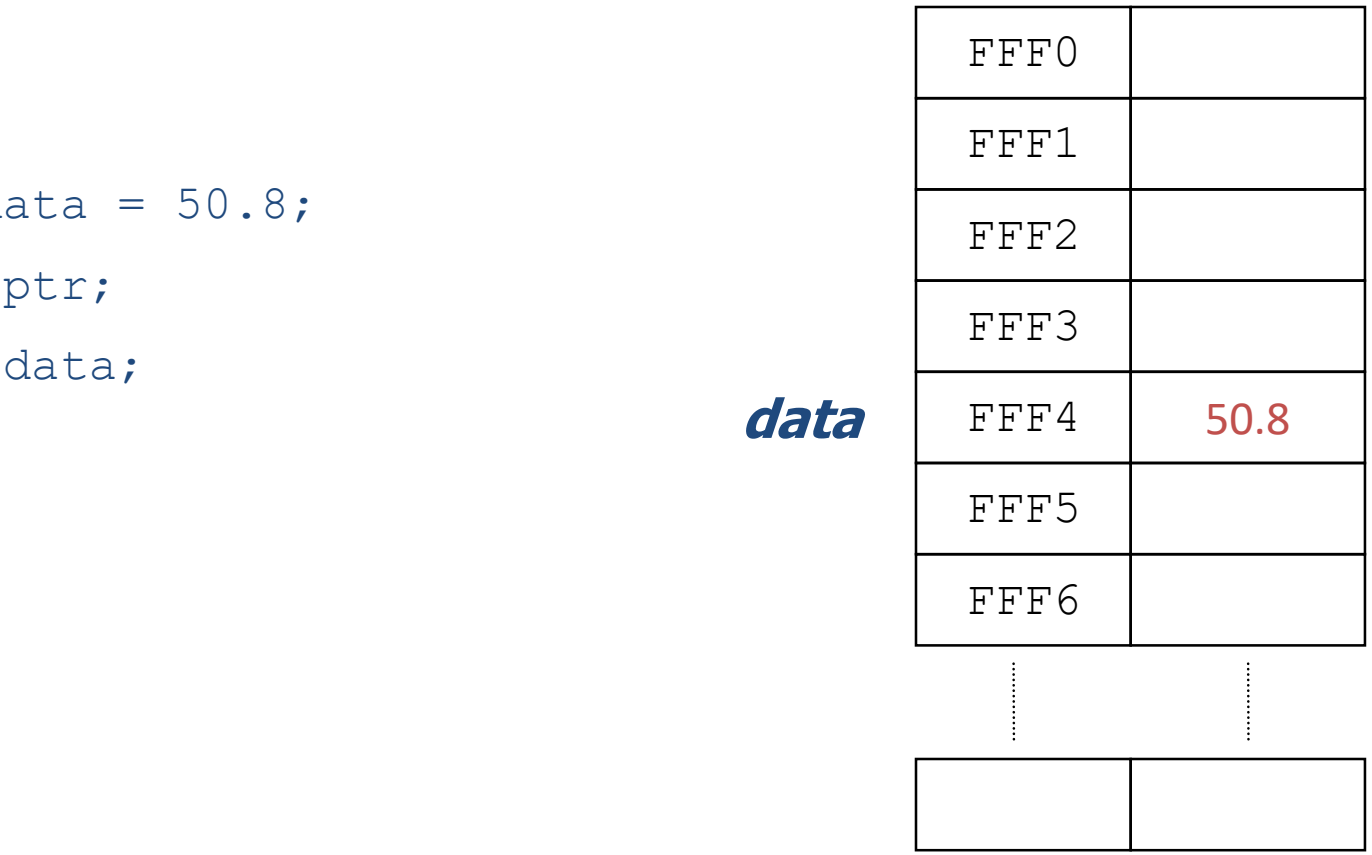
```
float data = 50.8;
float *ptr;
ptr = &data;
```

  - This will assign the address of the memory location allocated for the floating point variable **`data`** to the pointer variable **`ptr`**.

  - This is valid, since the variable **`data`** has already been allocated some memory space having a valid address

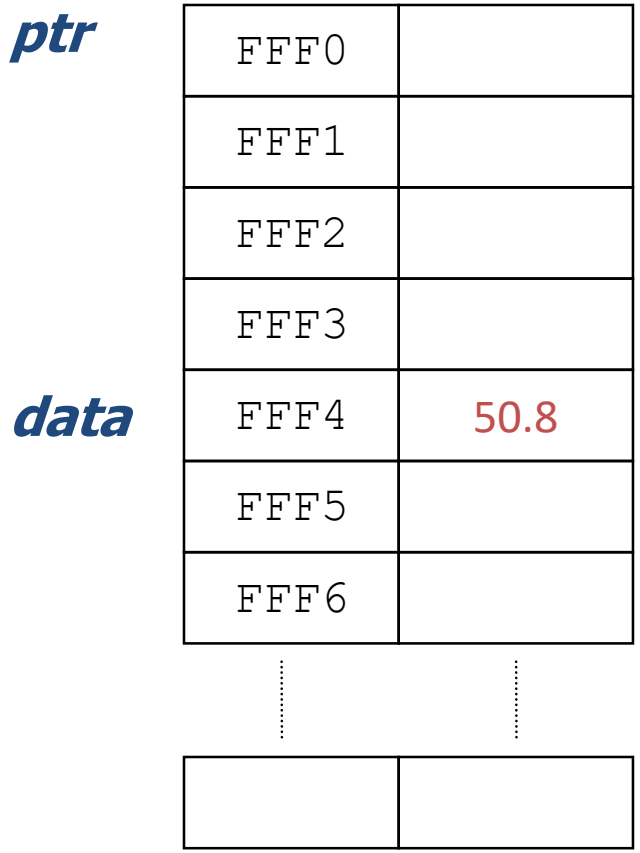
Assignment of Pointer Variables (Cont ..)

```
float data = 50.8;
float *ptr;
ptr = &data;
```



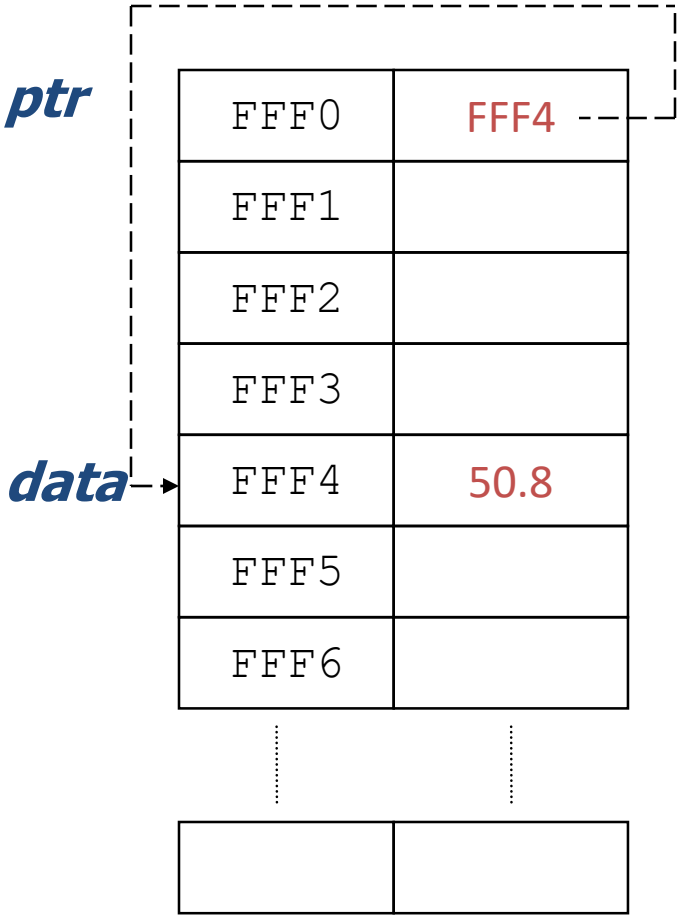
Assignment of Pointer Variables (Cont ..)

```
float data = 50.8;
float *ptr;
ptr = &data;
```



Assignment of Pointer Variables (Cont ..)

```
float data = 50.8;
float *ptr;
ptr = &data;
```



Assignment of Pointer Variables (Cont ..)

- Don't try to assign a specific integer value to a pointer variable since it can be disastrous

```
float *ptr;
ptr = 120; //This is wrong.
```

```
float data = 50.0;
float *ptr;
ptr = &data; //This is valid
```

Initializing pointers

- A pointer can be initialized during declaration by assigning it the address of an existing variable

```
float data = 50.8;
float ptr = &data;
```

- If a pointer is not initialized during declaration, it is wise to give it a **NULL** (0) value

```
float *fp = NULL;
```

The **NULL** pointer

- The **NULL** pointer is a valid address for any data type.
  - But **NULL** is not memory address 0.
- It is an error to dereference a pointer whose value is **NULL**.
  - Such an error may cause your program to crash, or behave erratically.
  - It is the programmer's job to check for this.

Dereferencing

- *Dereferencing* – Using a pointer variable to access the value stored at the location pointed by the variable
  - Provide indirect access to values and also called *indirection*
- Done by using the *dereferencing operator* * in front of a pointer variable
  - Unary operator
  - Highest precedence

Dereferencing (Cont ..)

- Example:

```
float data = 50.8;
float *ptr;
ptr = &data;
cout << *ptr;
```

- Once the pointer variable **ptr** has been declared, ***ptr** represents the value pointed to by **ptr** .

OUTPUT:

```
50.8
```

Dereferencing (Cont ..)

- The dereferencing operator * can also be used in assignments.

```
*ptr = 200;
```

  - Make sure that ptr has been properly initialized

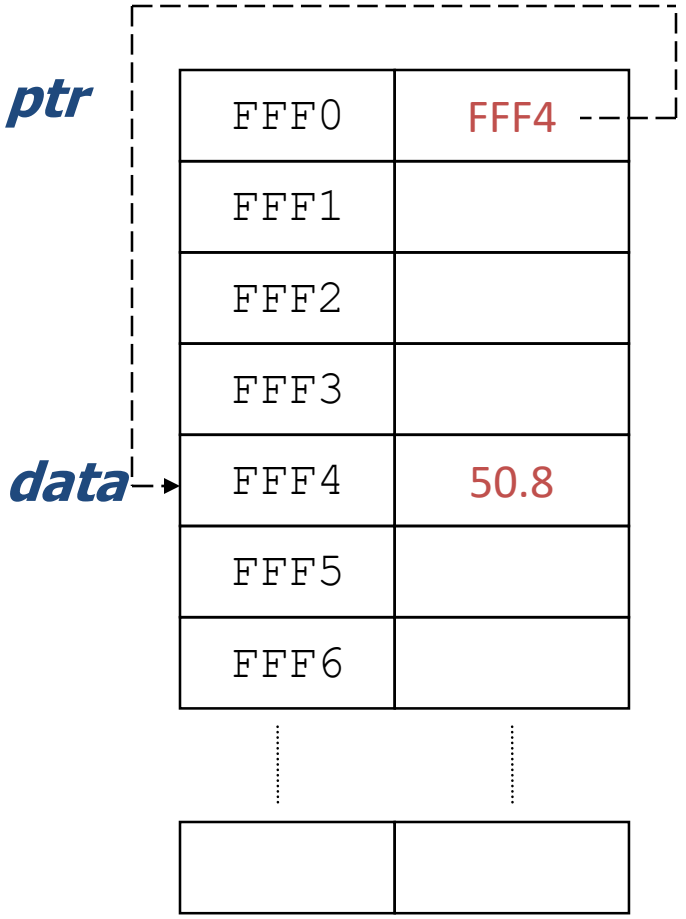
```
float data;
float *ptr;
ptr = &data;
*ptr = 200;
cout << *ptr;
```

OUTPUT:

```
200
```

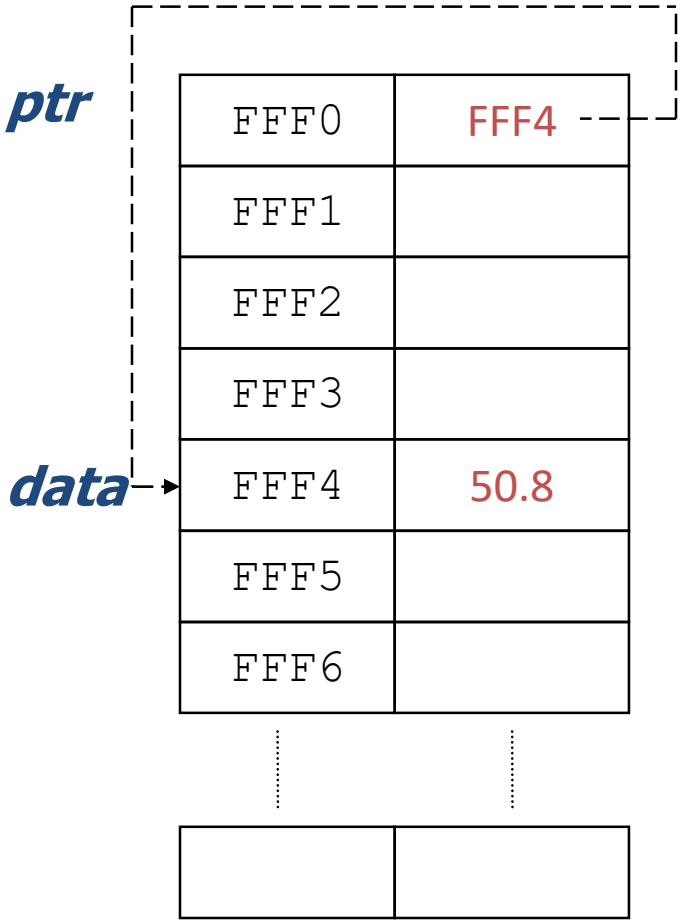
Dereferencing Example

```
#include <iostream.h>

void main()
{
    float data = 50.8;
    float *ptr;
    ptr = &data;
    cout << ptr << *ptr << endl;
    *ptr = 27.4;
    cout << *ptr << endl;
    cout << data << endl;
}
```



Dereferencing Example (Cont ..)

```
#include <iostream.h>

void main()
{
    float data = 50.8;
    float *ptr;
    ptr = &data;
    cout << ptr <<" "<< *ptr << endl;
    *ptr = 27.4;
    cout << *ptr << endl;
    cout << data << endl;
}
```

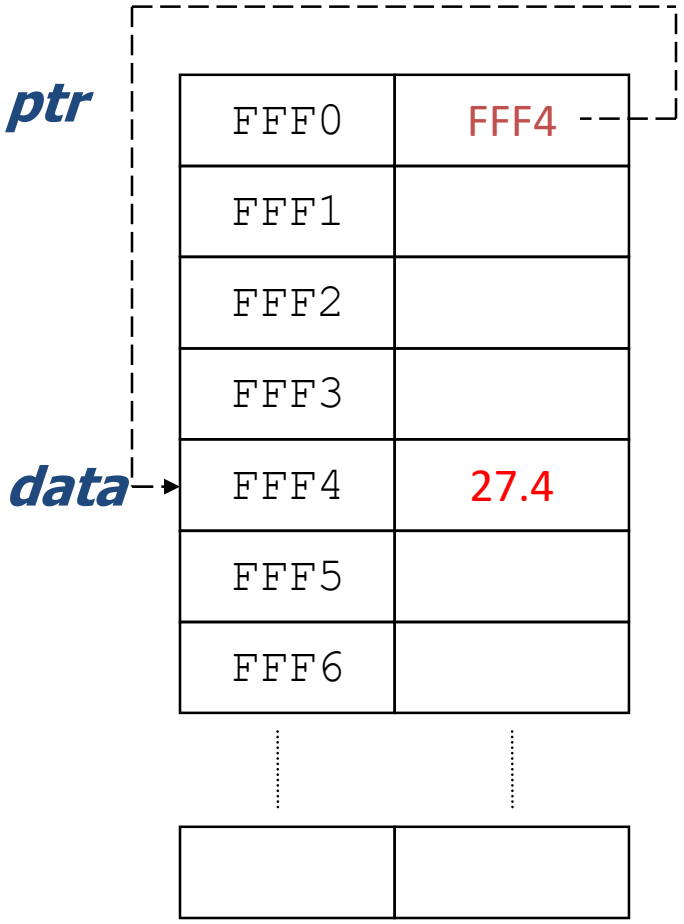


OUTPUT:

FFF4 50.8

Dereferencing Example (Cont ..)

```
#include <iostream.h>

void main()
{
    float data = 50.8;
    float *ptr;
    ptr = &data;
    cout << ptr <<" "<< *ptr << endl;
    *ptr = 27.4;
    cout << *ptr << endl;
    cout << data << endl;
}
```



OUTPUT:

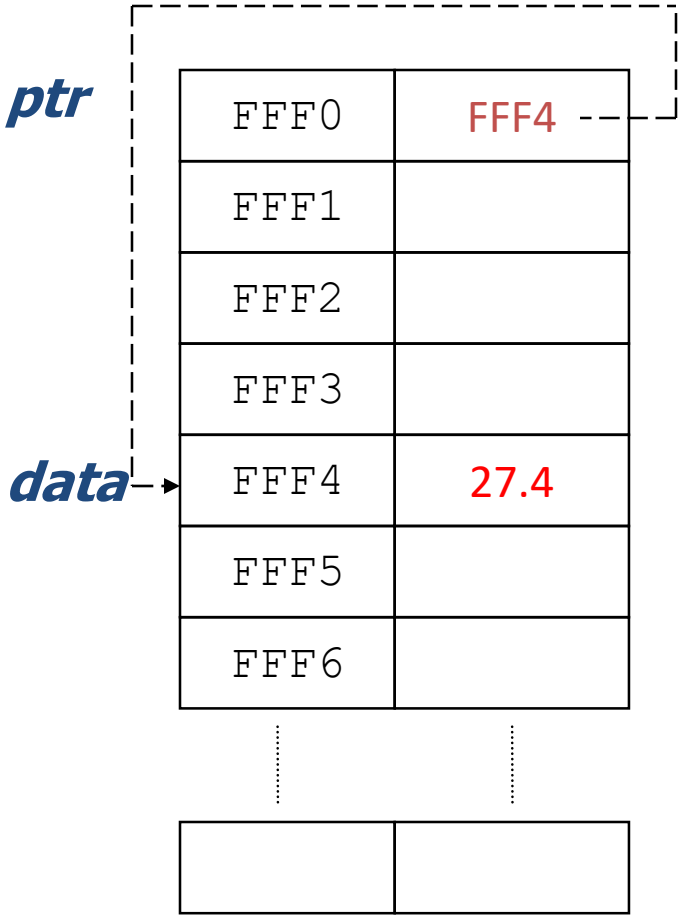
FFF4 50.8

Dereferencing Example (Cont ..)

```
#include <iostream.h>

void main()
{
    float data = 50.8;
    float *ptr;
    ptr = &data;
    cout << ptr << *ptr << endl;
    *ptr = 27.4;
    cout << *ptr << endl;
    cout << data << endl;
}
```

OUTPUT:

27.4

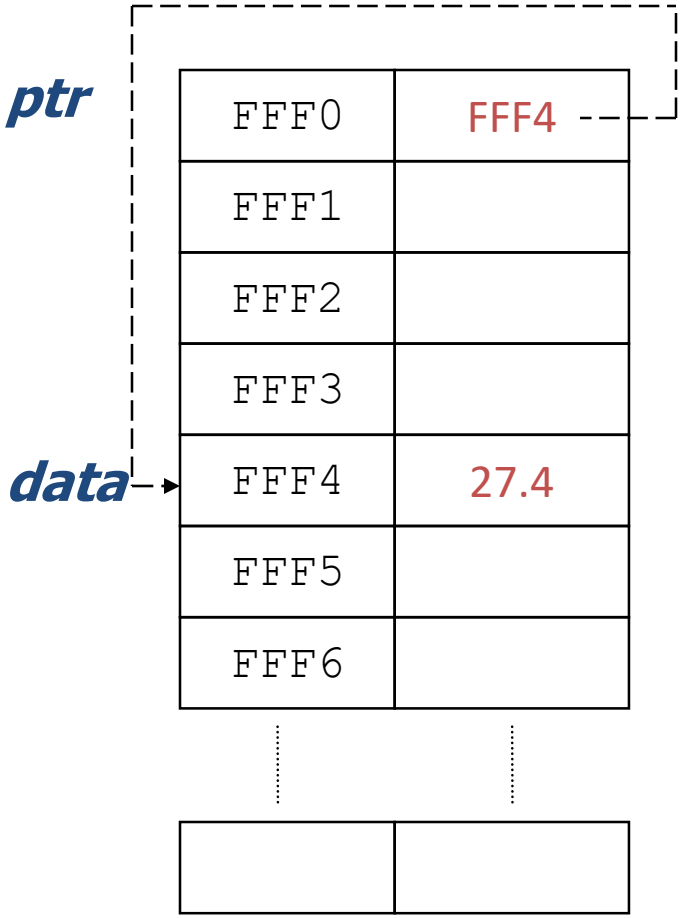


Dereferencing Example (Cont ..)

```
#include <iostream.h>

void main()
{
    float data = 50.8;
    float *ptr;
    ptr = &data;
    cout << ptr << *ptr << endl;
    *ptr = 27.4;
    cout << *ptr << endl;
    cout << data << endl;
}
```

OUTPUT

???



```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue = 5, secondvalue = 15;
  int * p1, * p2;

  p1 = &firstvalue;  // p1 = address of firstvalue
  p2 = &secondvalue; // p2 = address of secondvalue
  *p1 = 10;          // value pointed to by p1 = 10
  *p2 = *p1;         // value pointed to by p2 = value pointed by p1
  p1 = p2;           // p1 = p2 (value of pointer is copied)
  *p1 = 20;          // value pointed by p1 = 20

  cout << "firstvalue is " << firstvalue << '\n';
  cout << "secondvalue is " << secondvalue << '\n';
  return 0;
}
```

Why Pointers?

- To modify a variable in a function that is not a global, or a local to that function
- To save space
- To save time
- To use dynamic memory (Lecture A6)
- Used extensively in linked structures

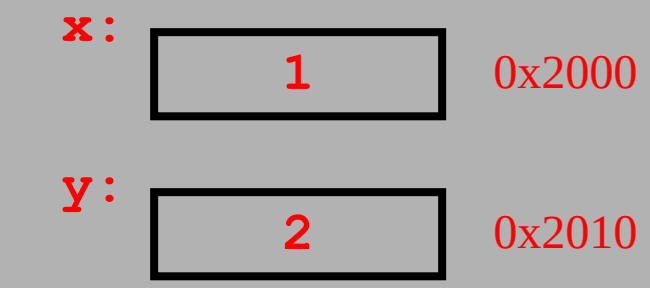
Passing pointers to a function

Solution 1

```
#include <iostream>

void fakeSwap(int a, int b)
{
   int tmp;

   tmp = a;
   a = b;
   b = tmp;
}

int main()
{
   int x = 1, y = 2;

   fakeSwap(x, y);
   cout << x << "  " << y << endl;
}
```

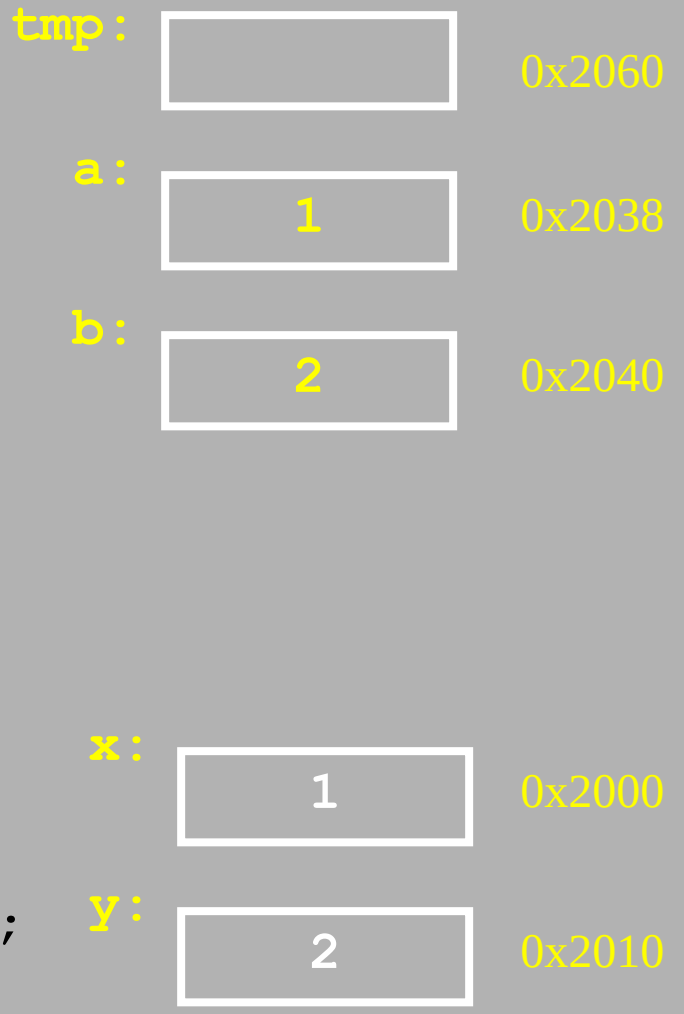
Solution 1

```
#include <iostream >

void fakeSwap(int a, int b)
{
   int tmp;

   tmp = a;
   a = b;
   b = tmp;
}

int main()
{
   int x = 1, y = 2;

   fakeSwap(x, y);
   cout << x << "  " << y << endl;
}
```



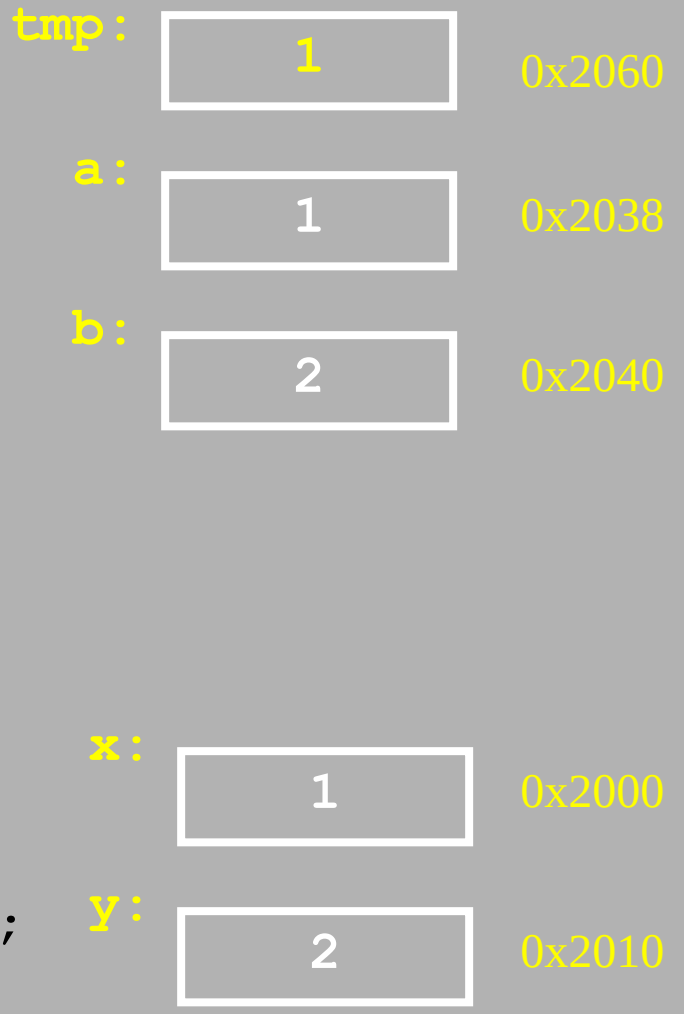
Solution 1

```
#include < iostream >

void fakeSwap(int a, int b)
{
    int tmp;

    tmp = a;
    a = b;
    b = tmp;
}

int main()
{
    int x = 1, y = 2;

    fakeSwap(x, y);
    cout << x << "  " << y << endl;
}
```



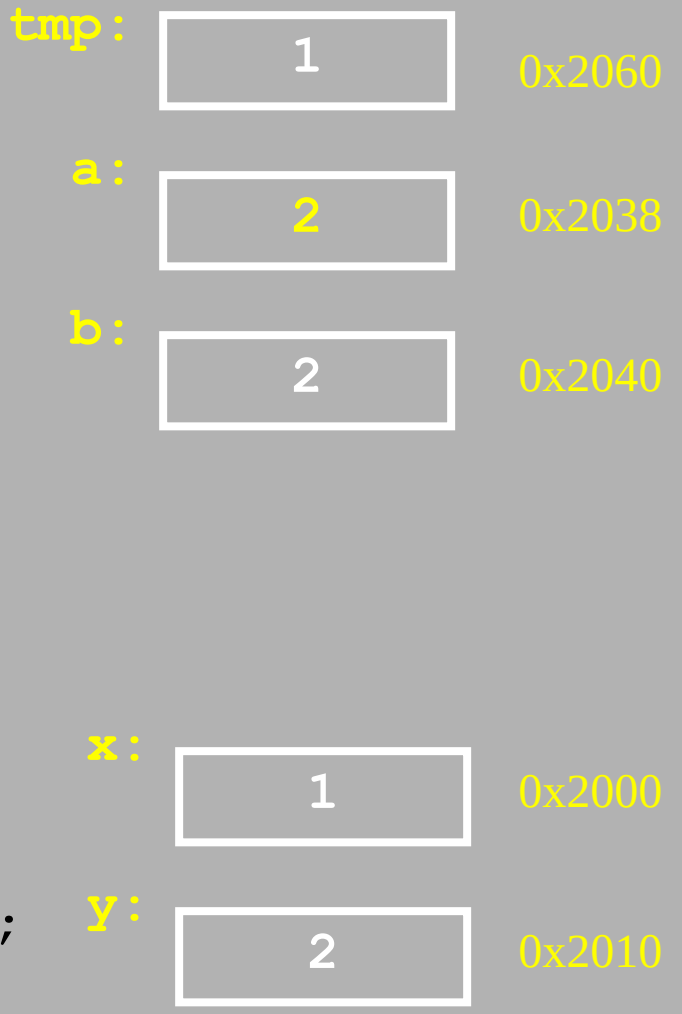
Solution 1

```
#include < iostream >

void fakeSwap(int a, int b)
{
    int tmp;

    tmp = a;
    a = b;
    b = tmp;
}

int main()
{
    int x = 1, y = 2;

    fakeSwap(x, y);
    cout << x << "  " << y << endl;
}
```



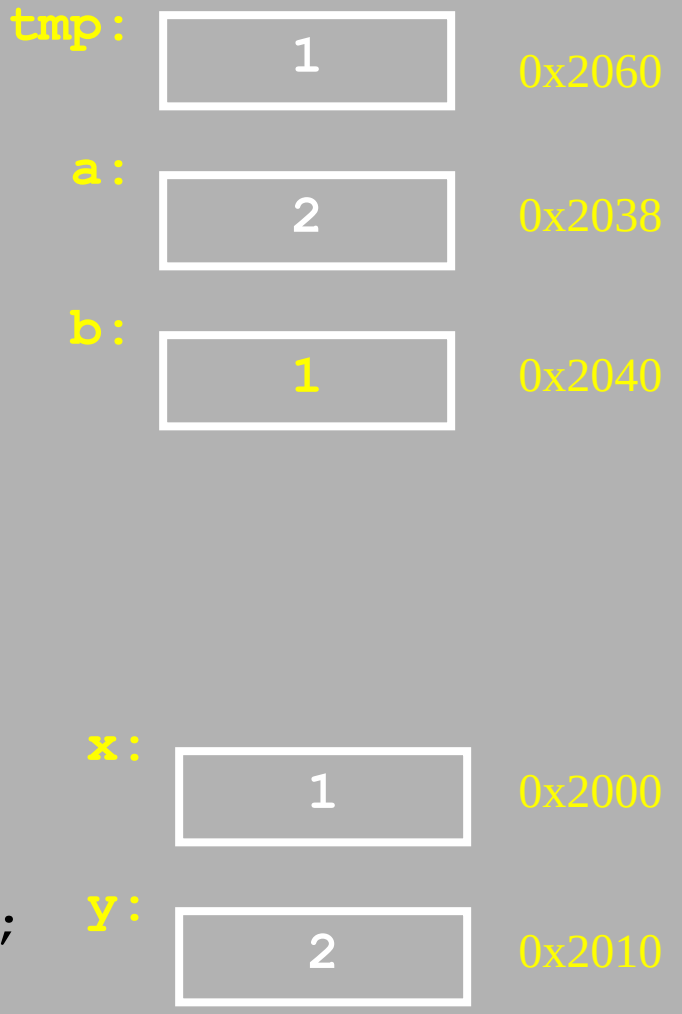
Solution 1

```
#include < iostream >

void fakeSwap(int a, int b)
{
    int tmp;

    tmp = a;
    a = b;
    b = tmp;
}

int main()
{
    int x = 1, y = 2;

    fakeSwap(x, y);
    cout << x << "  " << y << endl;
}
```



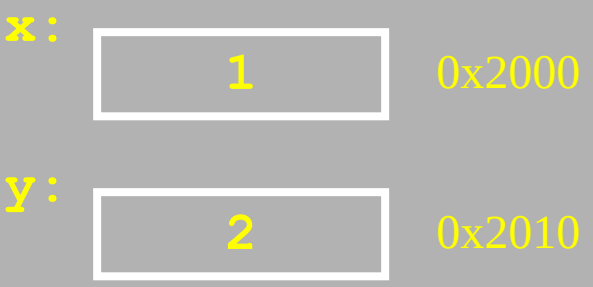
Solution 1

```
#include < iostream >

void fakeSwap(int a, int b)
{
    int tmp;

    tmp = a;
    a = b;
    b = tmp;
}

int main()
{
    int x = 1, y = 2;

    fakeSwap(x, y);
    cout << x << "  " << y << endl;
}
```



Solution 1

```
#include < iostream >

void fakeSwap(int a, int b)
{
    int tmp;

    tmp = a;
    a = b;
    b = tmp;
}

int main()
{
    int x = 1, y = 2;

    fakeSwap(x, y);
    cout << x << "  " << y << endl;
}
```

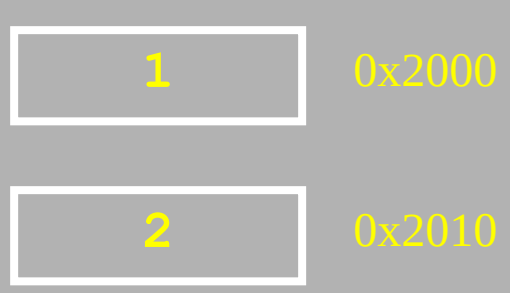


Solution 2

```
#include < iostream >

void trueSwap(int* a, int* b)
{
   int tmp;

   tmp = *a;
   *a = *b;
   *b = tmp;
}

int main()
{
   int x = 1, y = 2;

   trueSwap(&x, &y);
   cout << x << "  " << y << endl;
}
```

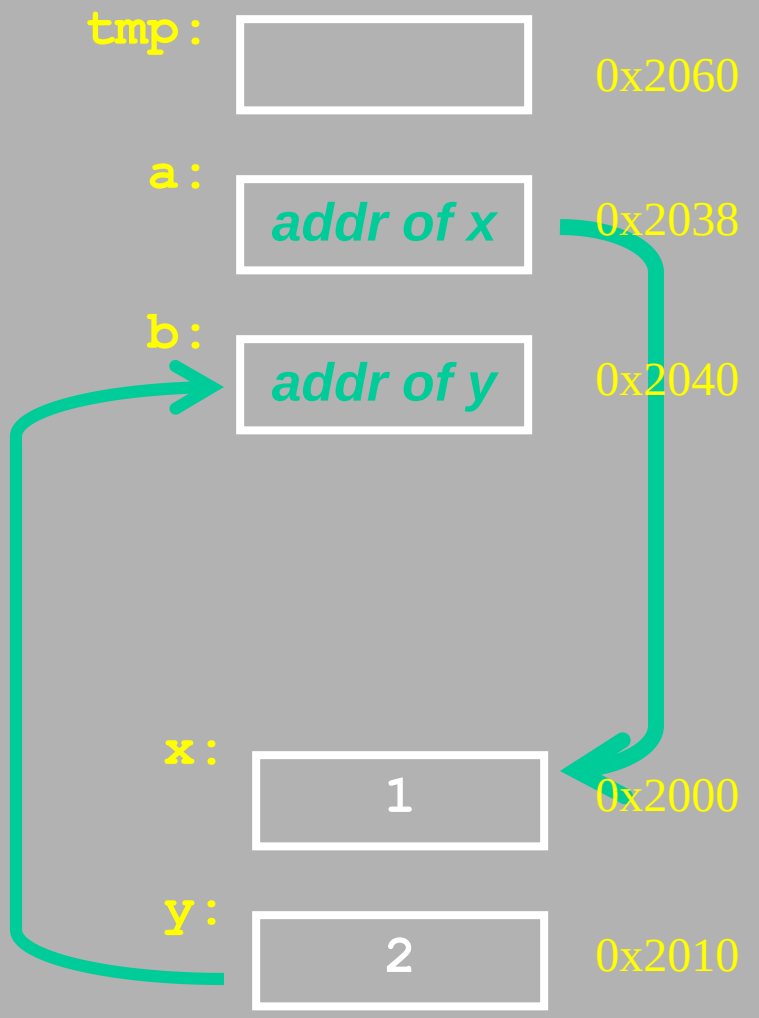
Solution 2

```
#include < iostream >

void trueSwap(int* a, int* b)
{
    int tmp;

    tmp = *a;
    *a = *b;
    *b = tmp;
}

int main()
{
    int x = 1, y = 2;

    trueSwap(&x, &y);
    cout << x << "  " << y << endl;
}
```



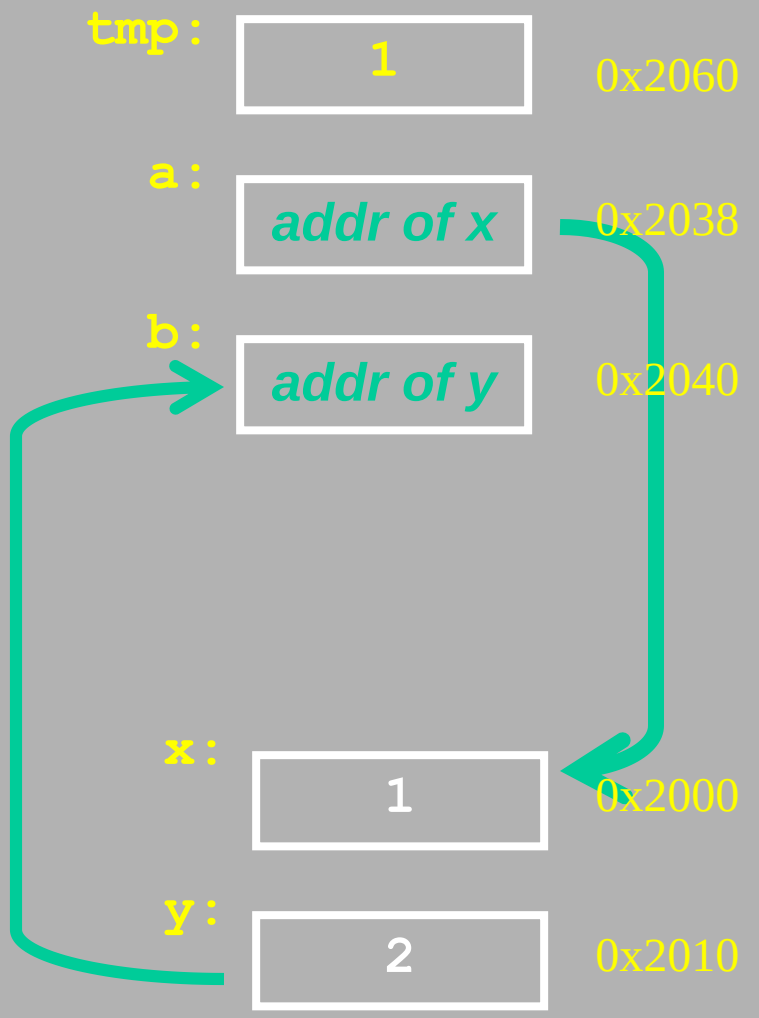
Solution 2

```
#include < iostream >

void trueSwap(int* a, int* b)
{
    int tmp;

    tmp = *a;
    *a = *b;
    *b = tmp;
}

int main()
{
    int x = 1, y = 2;

    trueSwap(&x, &y);
    cout << x << "  " << y <<
endl;
}
```



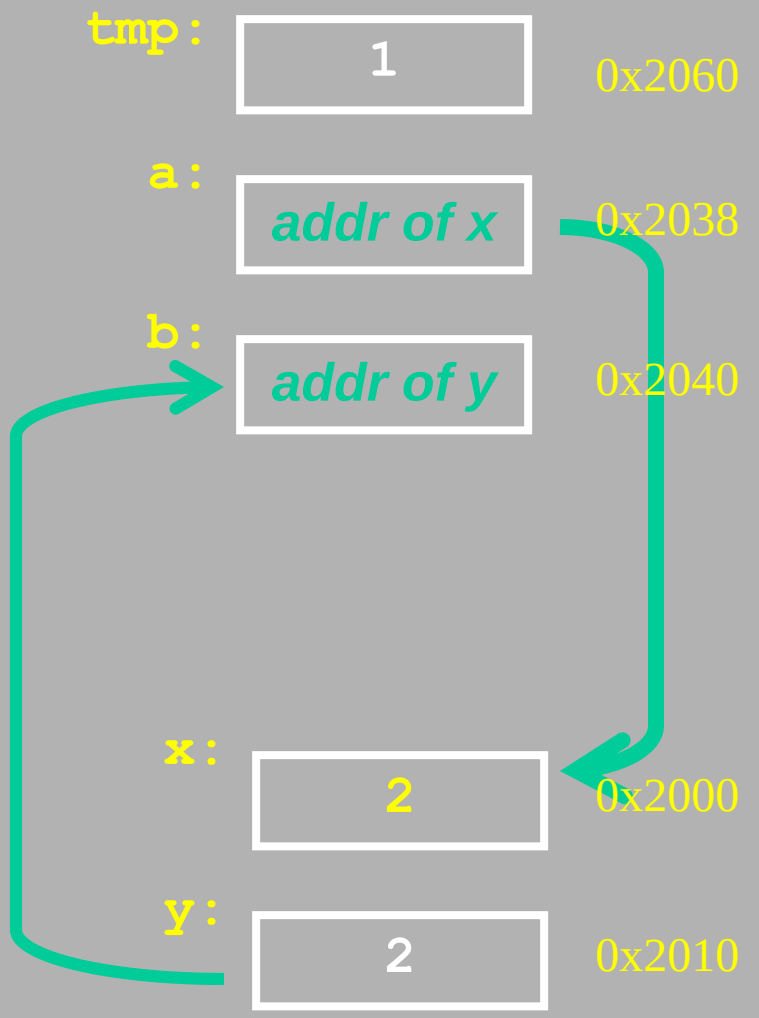
Solution 2

```
#include < iostream >

void trueSwap(int* a, int* b)
{
    int tmp;

    tmp = *a;
    *a = *b;
    *b = tmp;
}

int main()
{
    int x = 1, y = 2;

    trueSwap(&x, &y);
    cout << x << "  " << y <<
endl;
}
```



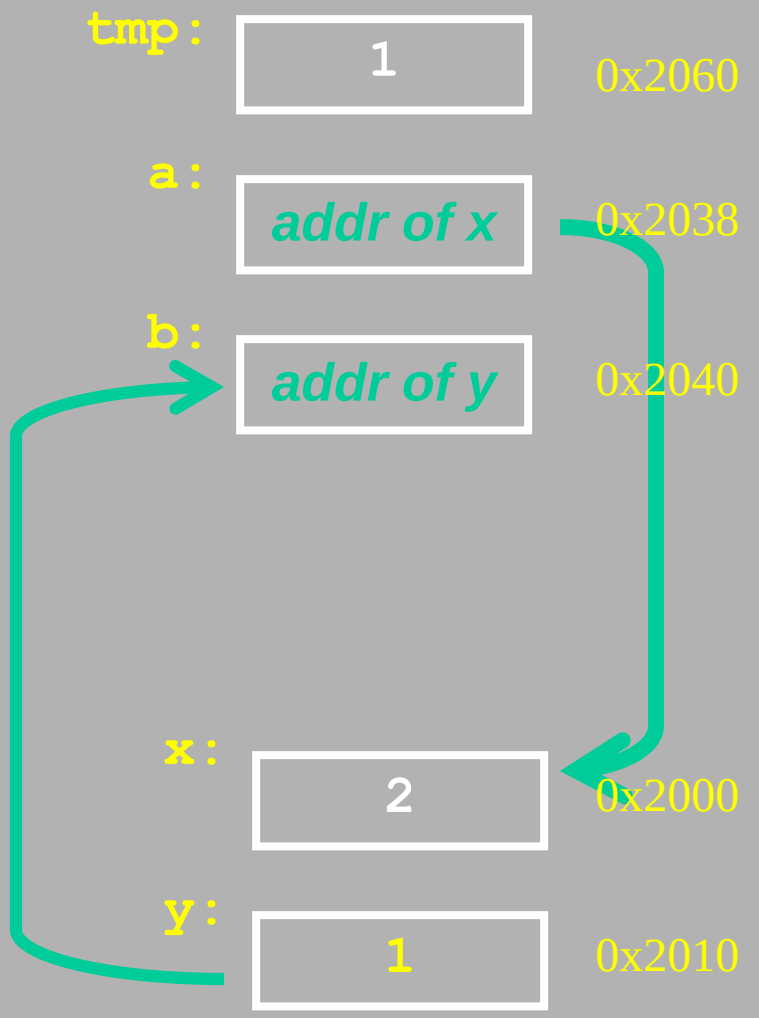
Solution 2

```
#include < iostream >

void trueSwap(int* a, int* b)
{
    int tmp;

    tmp = *a;
    *a = *b;
    *b = tmp;
}

int main()
{
    int x = 1, y = 2;

    trueSwap(&x, &y);
    cout << x << "  " << y <<
endl;
}
```



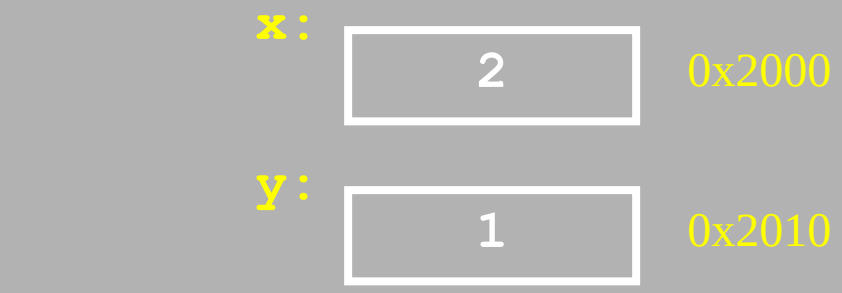
Solution 2

```
#include < iostream >

void trueSwap(int* a, int* b)
{
    int tmp;

    tmp = *a;
    *a = *b;
    *b = tmp;
}

int main()
{
    int x = 1, y = 2;

    trueSwap(&x, &y);
    cout << x << "  " << y <<
endl;
}
```



Solution 2

```
#include < iostream >

void trueSwap(int* a, int* b)
{
    int tmp;

    tmp = *a;
    *a = *b;
    *b = tmp;
}

int main()
{
    int x = 1, y = 2;

    trueSwap(&x, &y);
    cout << x << "  " << y <<
endl;
}
```



More example

```
void main ()
{
   int r, s = 5, t = 6;
   int *tp = &t;
   r = MyFunction(tp,s);
     cout << r <<“ ”<< t<< endl;

   r = MyFunction(&t,s);
     cout << r <<“ ”<< t<< endl;

   r = MyFunction(&s,*tp);
     cout << r <<“ ”<< t<< endl;

}
```

```
int MyFunction(int *p, int i)
{
   *p = 3;
   i = 4;
   return i;
}
```

Out Put:

```
4  3
4  3
4  6
```

The Relationship Between Arrays and Pointers

- Array names can be used as a pointer constants, and pointer can be used as array names.

- Array name is really a pointer as it holds the starting address of the array.

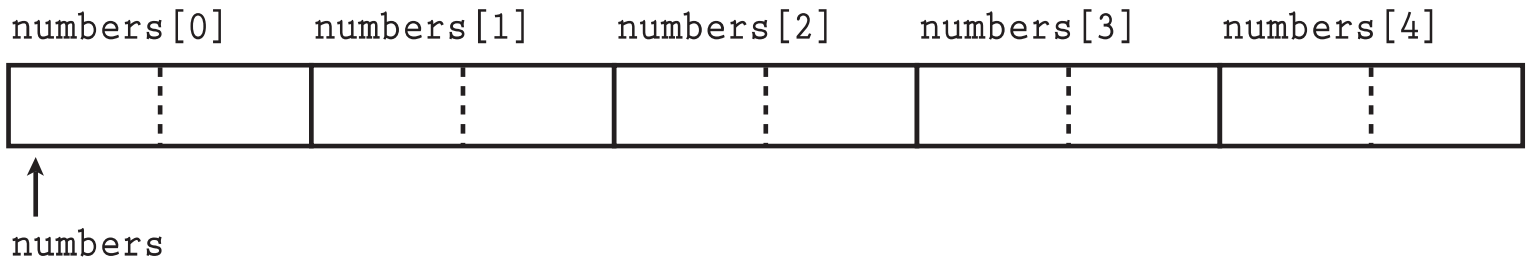
```
1  // This program shows an array name being dereferenced with the *
2  // operator.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     short numbers[] = {10, 20, 30, 40, 50};
9
10    cout << "The first element of the array is ";
11    cout << *numbers << endl;
12    return 0;
13 }
```

Program Output

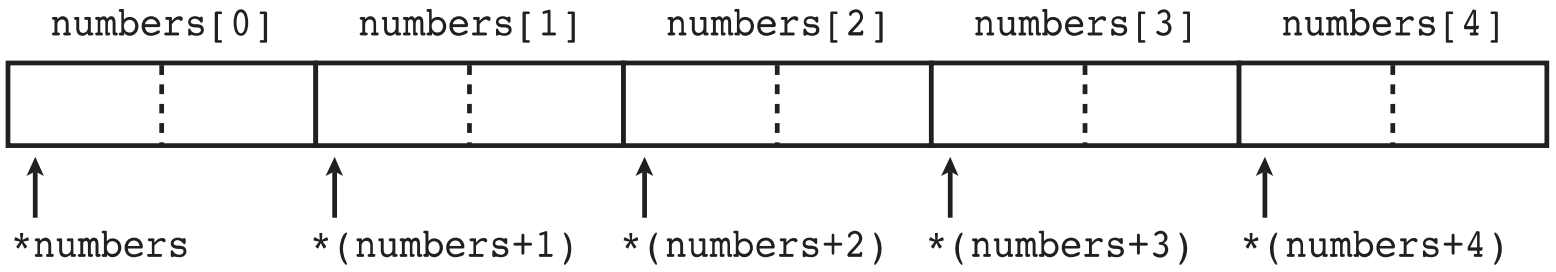
```
The first element of the array is 10
```

What will be the out put of
cout << numbers << endl; ???

Array elements always store together into memory as in figure



Following figure shows the equivalence of subscript notation and pointer notation of an Array.



Pointers and Arrays

```
Type array[size];
Type* pPtr = array + i;
Type* qPtr = array + j;
```

- The name `array` is equivalent to `&array[0]`
- `pPtr++` increments `pPtr` to point to the next element of `array`.
- `pPtr += n` increments `pPtr` to point to n elements beyond where it currently points.
- `pPtr-qPtr` equals `i-j`.

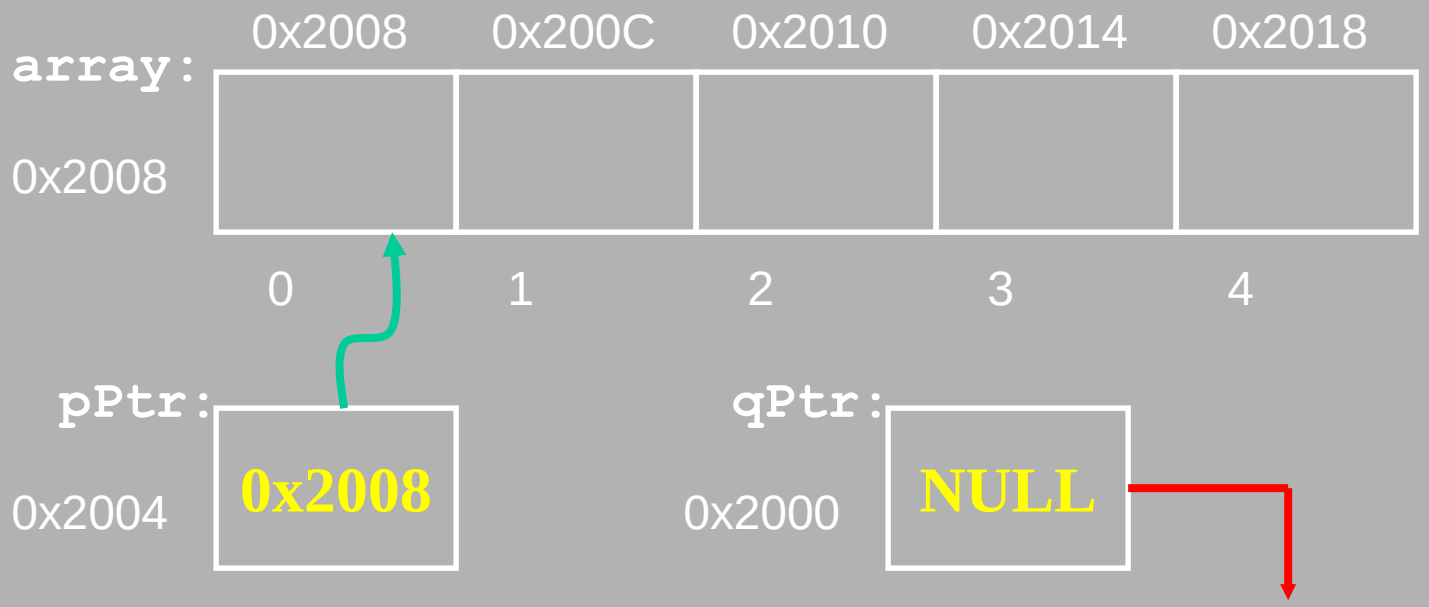
Pointers and Arrays (cont)

A normal 1 dimensional array:

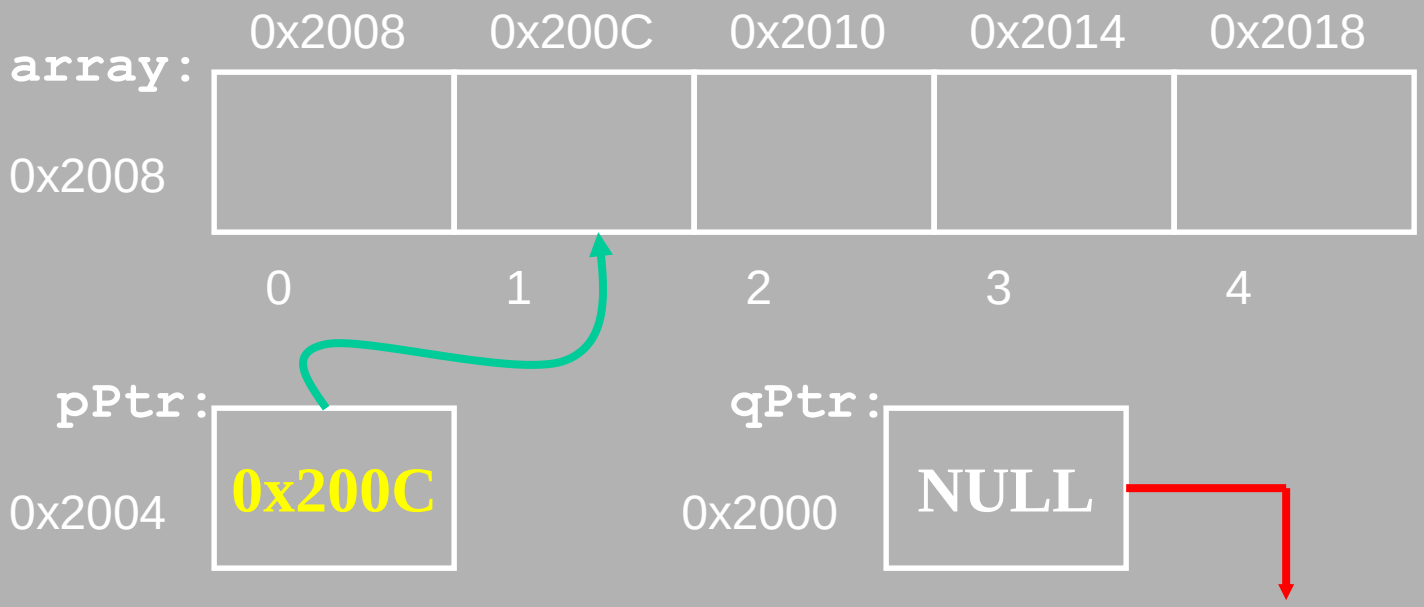
***Type* `array[`*`size`*`];`**

- `array[0]` **is equivalent to** `*array`
- `array[n]` **is equivalent to** `*(array + n)`

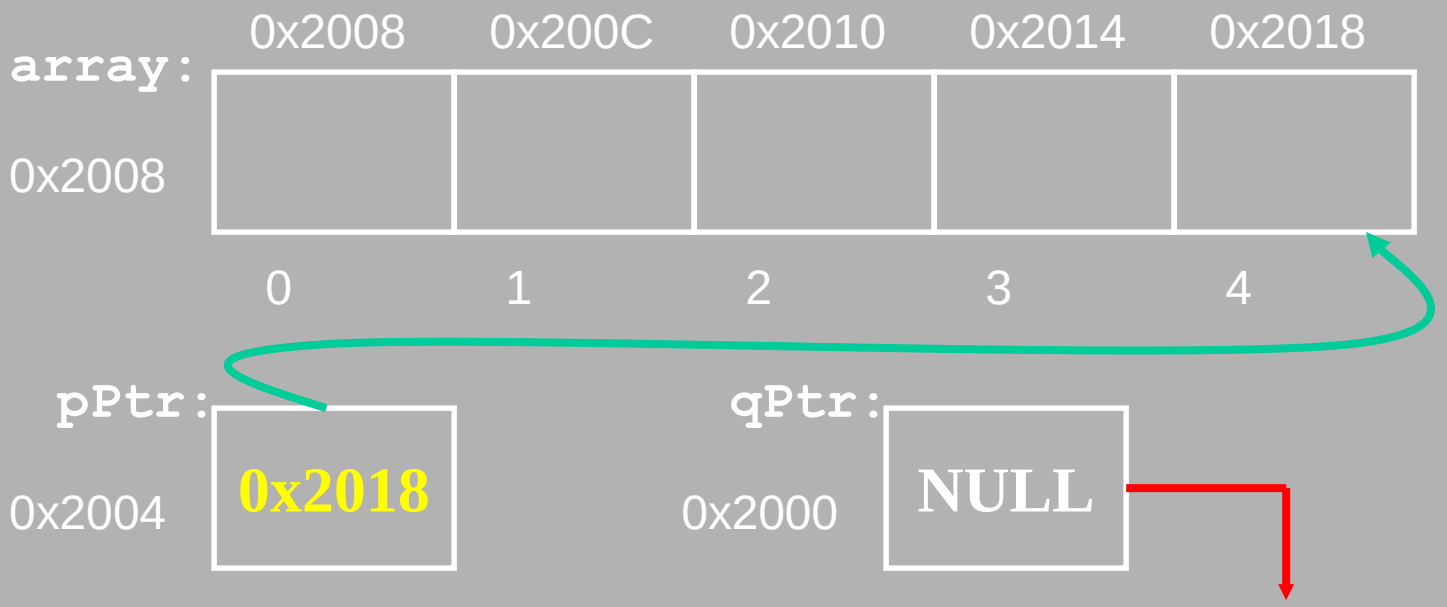
Basic Pointer Arithmetic



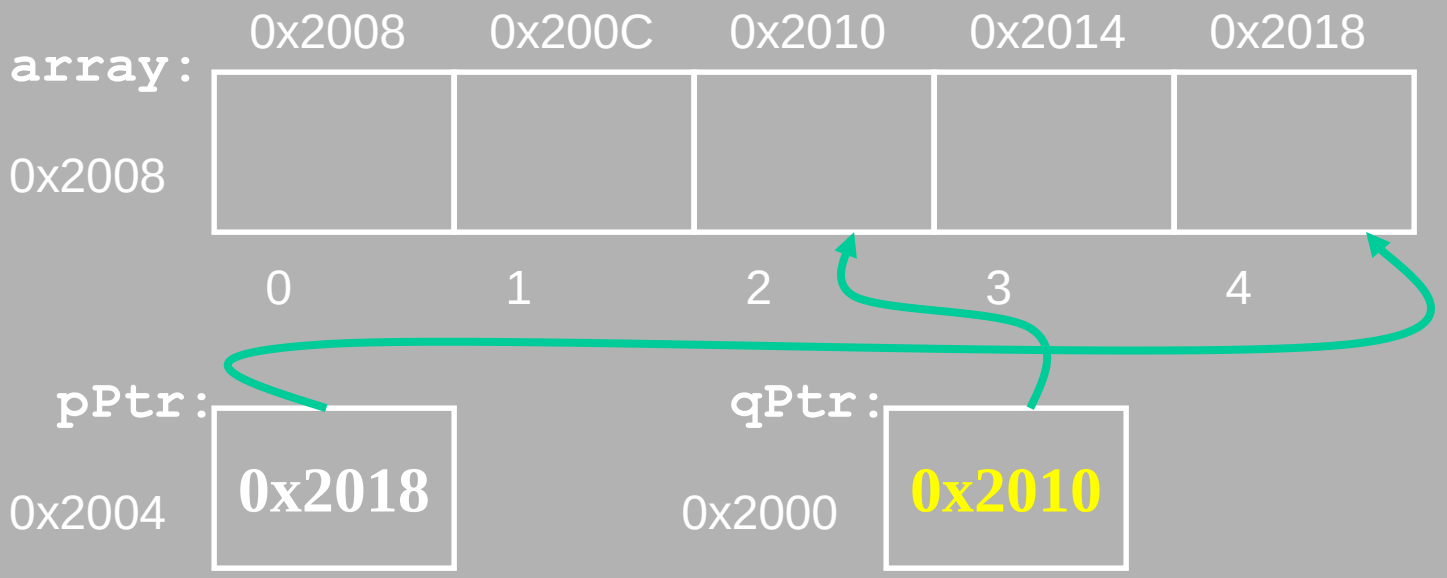

```
float  array[5];
float* pPtr = array;
float* qPtr = NULL;
```



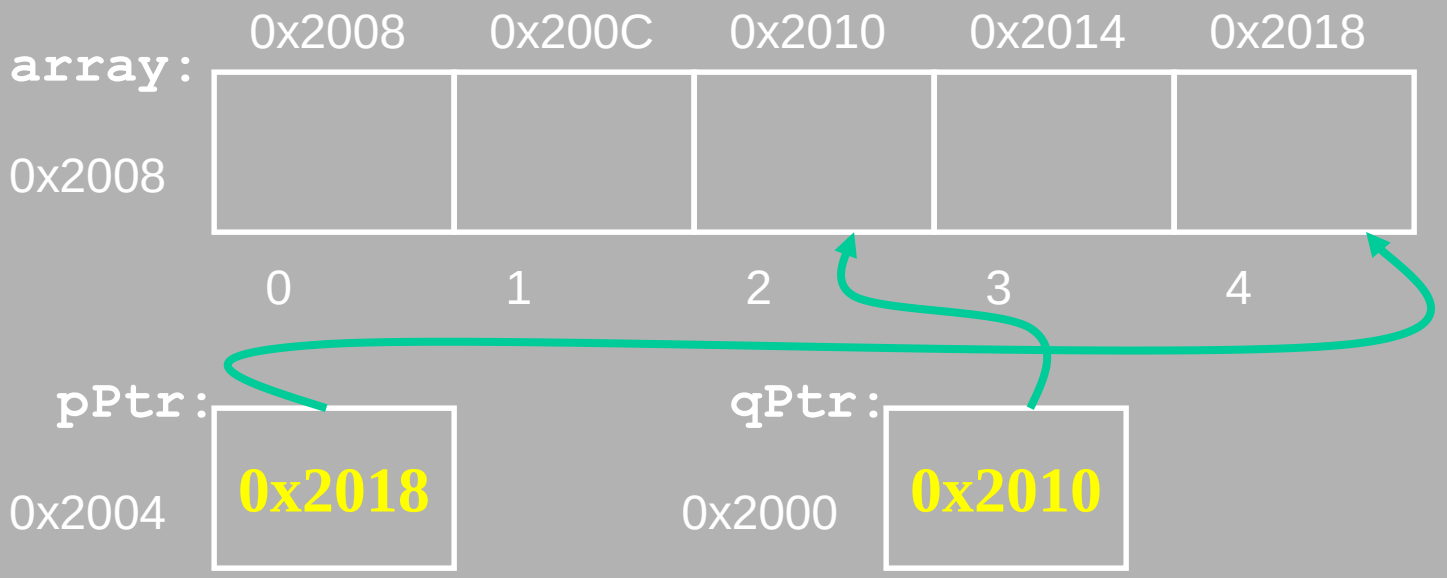

```
float  array[5];
float* pPtr = array;
float* qPtr = NULL;

pPtr++; /* pPtr now holds the address: &array[1] */
```




```
float  array[5];
float* pPtr = array;
float* qPtr = NULL;


pPtr++; /* pPtr = &array[1] */
pPtr += 3; /* pPtr now hold the address: &array[4] */
```




```
float  array[5];
float* pPtr = array;
float* qPtr = NULL;


pPtr++; /* pPtr = &array[1] */
pPtr += 3; /* pPtr = &array[4] */
qPtr = array + 2; /*qPtr now holds the address &array[2]*/
```




```
float  array[5];
float* pPtr = array;
float* qPtr = NULL;


pPtr++; /* pPtr = &array[1] */
pPtr += 3; /* pPtr = &array[4] */
qPtr = array + 2; /* qPtr = &array[2] */
Cout<< pPtr-qPtr;
```

Pointer to an Array

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

int main ()
{
      double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
      double *p;
      p = balance;

      cout << "Array values using pointer " << endl;

      for ( int i = 0; i < 5; i++
            cout << *(p + i) << endl;
      }
   return 0;
}
```

Array values using pointer

1000.0
2 .0
3.4
17 .0
50.0

```
1 void main( void )
2 {
3  float r[5] = {22.5,34.8,46.8,59.1,68.3};
4  cout <<"1st element: "<< r[0] <<"\n";
5  cout <<"1st element: "<< *r <<"\n";
6  cout <<"3rd element: "<< r[2] <<"\n";
7  cout <<"3rd element: "<< *(r+2)<<"\n";
8  float *p;
9  p = r; //&r[0]
10 cout <<"1st element: "<< p[0] <<"\n";
11 cout <<"1st element: "<< *p <<"\n";
12 cout <<"3rd element: "<< p[2]<<"\n";
13 cout <<"3rd element: "<< *(p+2)<<"\n";
14 for(int i=0; i<5; i++)
15   cout <<"Element "<<(i+1)<<" is: "<<*p++<<"\n";
16 }
```

```
1st element: 22.5
1st element: 22.5
3rd element: 46.8
3rd element: 46.8
1st element: 22.5
1st element: 22.5
3rd element: 46.8
3rd element: 46.8
Element 1 is: 22.5
Element 2 is: 34.8
Element 3 is: 46.8
Element 4 is: 59.1
Element 5 is: 68.3
```

An array is simply a block of memory. An array can be accessed with pointers as well as with [ ] square brackets. *The name of an array variable is a pointer to the first element in the array.* So, any operation that can be achieved by array subscripting can also be done with pointers or vice-versa.

```
1st element: 22.5
1st element: 22.5
3rd element: 46.8
3rd element: 46.8
1st element: 22.5
1st element: 22.5
3rd element: 46.8
3rd element: 46.8
Element 1 is: 22.5
Element 2 is: 34.8
Element 3 is: 46.8
Element 4 is: 59.1
Element 5 is: 68.3
```