CS 213

Data Structures

Pointers and Dynamic Arrays

These slides are from

C++ Classes and Data Structures

Jeffrey S. Childs

Clarion University of PA

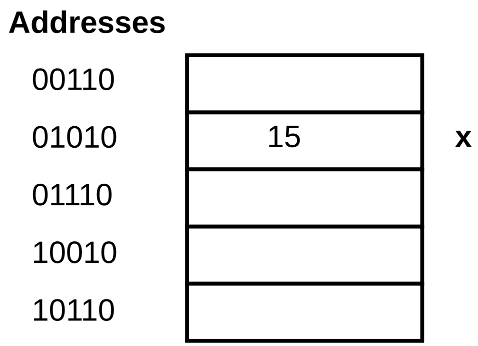
What Size Should We Make for an Array?

- If it is too small, it might get filled up.
- If it is too large, we only use a small fraction of the array space, and we will be wasting memory.
- The best approach might be to start an array off as small, then have it grow larger as more data come in.
- This cannot be done with ordinary arrays, but it can be done with ***pointers*** and ***dynamically-allocated memory.***

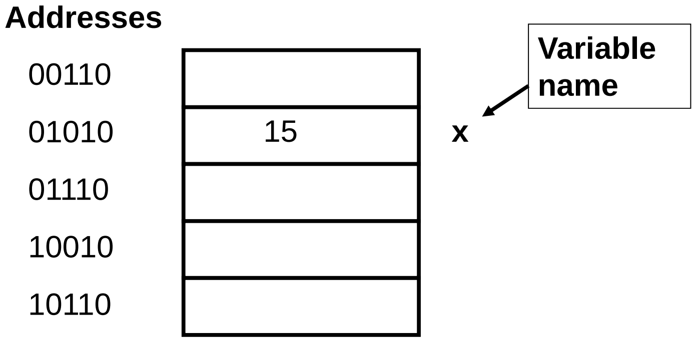
Memory Terminology

- variable name
- variable
- value
- address – a binary number used by the operating system to identify a memory cell of RAM
- It is important to know the precise meanings of these terms

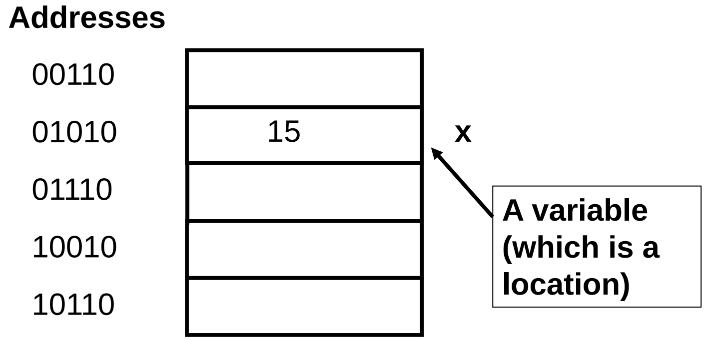
Memory Terminology (cont.)



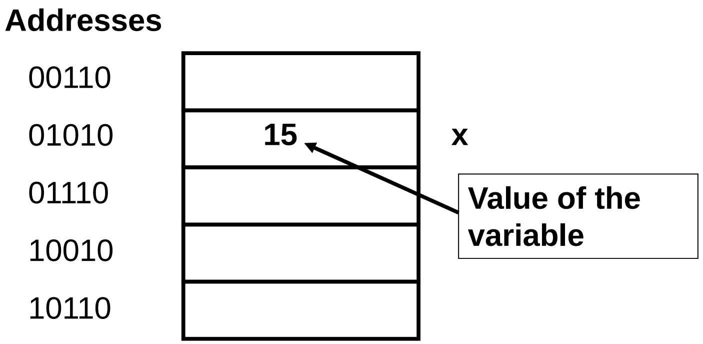
Memory Terminology (cont.)



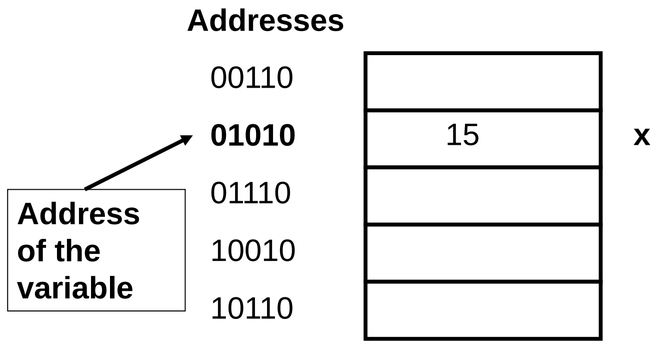
Memory Terminology (cont.)



Memory Terminology (cont.)



Memory Terminology (cont.)



Pointers

- A ***pointer*** is a variable used to store an address
- The declaration of a pointer must have a data type for the address the pointer will hold (looks like this):

```
int *ptr;
char *chptr;
```


Location Behavior

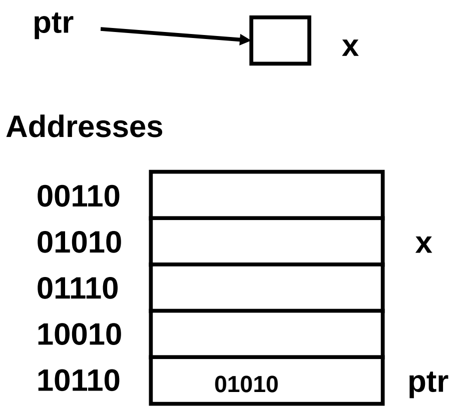
- A variable is a location
- When locations are used in code, they behave differently depending on whether or not they are used on the left side of an assignment
- If not used on the left side of an assignment, the value at the location is used
- When used on the left side of assignment, the location itself is used

Address-of operator

- An address can be assigned to a pointer using the ***address-of*** operator &:

```
int *ptr;
int x;
ptr = &x;
```


Result



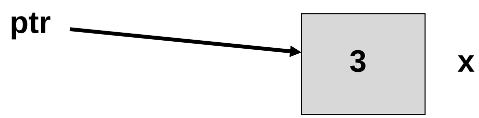
Dereference Operator

- The ***dereference*** operator, *, is used on a pointer (or any expression that yields an address)
- The result of the dereference operation is a ***location*** (the location at the address that the pointer stores)
- Therefore, the way the dereference operator behaves depends on where it appears in code (like variables)

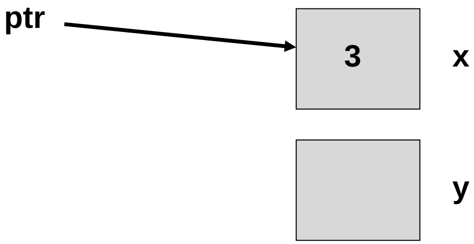
int x = 3, *ptr;

ptr = &x;
(what happens?)

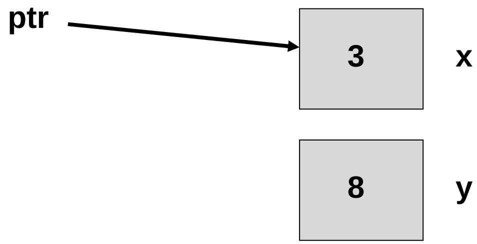
ptr = &x;



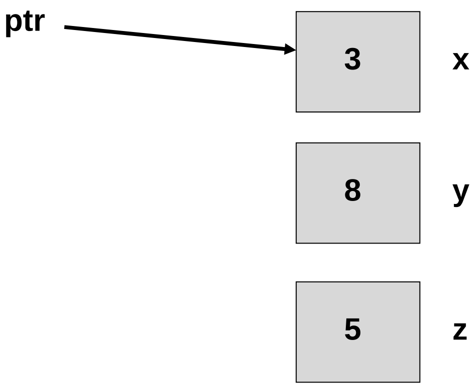
y = *ptr + 5;
(what happens?)



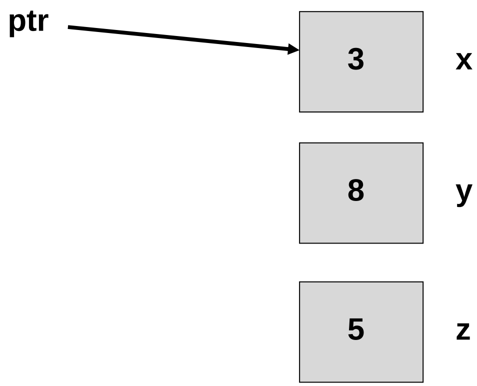
y = *ptr + 5;



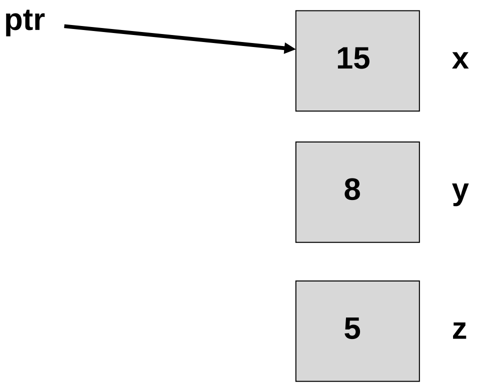
int z = 5;



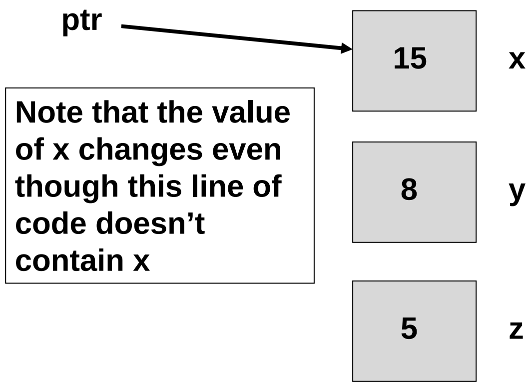
*ptr = 10 + z;
(what happens?)



*ptr = 10 + z;



*ptr = 10 + z;



Arrays

- The address of an array is the same as the address of the first element of the array.
- An array's name (without using an index) contains the address of the array.
- The address of the array can be assigned to a pointer by assigning the array's name to the pointer.
- An array name is not a pointer because the address in it cannot be changed (the address in a pointer can be changed).

The [ ] Operator

- When an array is indexed, [ ] is an operator.
- For example, nums[3] produces the result *(nums + 3).
- C++ produces an address from the expression nums + 3, by:
  - Noting what data type is stored at the address that nums contains
  - Multiplying 3 by the number of bytes in that data type
  - Adding the product onto the address stored in nums
- After the address is produced from nums + 3, it is dereferenced to get a location.

The Heap

- The ***heap*** is a special part of RAM memory reserved for program usage.
- When a program uses memory from the heap, the used heap memory is called ***dynamically-allocated memory***.
- The only way to dynamically allocate memory (use memory in the heap) is by using the ***new*** operator.

The new Operator

- The new operator has only one operand, which is a data type.
- The new operation produces the address of an unused chunk of heap memory large enough to hold the data type (dynamically allocated)
- In order to be useful, the address must be assigned to a pointer, so that the dynamically allocated memory can be accessed through the pointer:

```
int *ptr;
ptr = new int;
*ptr = 5;
```


int *ptr;

ptr

ptr = new int;

ptr

ptr = new int;

ptr

ptr = new int;

ptr

Found a block in heap

110110

ptr = <u>new int;</u>

Replaces new operation

ptr

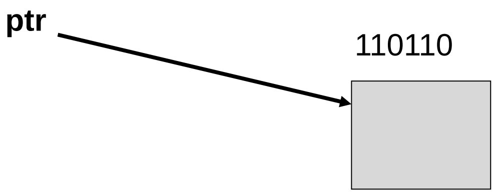
110110

ptr = 110110;

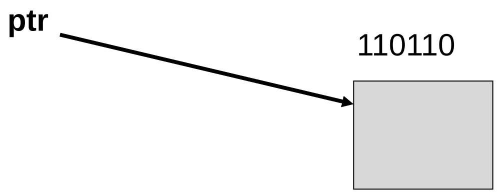
ptr

110110

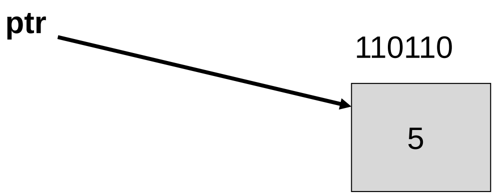
ptr = 110110;



*ptr = 5;
(what happens?)



*ptr = 5;



Dynamically-allocated memory

- Has no name associated with it (other locations have variable names associated with them)
- Can only be accessed by using a pointer
- The compiler does not need to know the size (in bytes) of the allocation at compile time
- The compiler *does* need to know the size (in bytes) of any *declared* variables or arrays at compile time

Dynamic Arrays

- The real advantage of using heap memory comes from using arrays in heap memory
- Arrays can be allocated by the new operator; then they are called ***dynamic arrays*** (but they have no name either)

```
int *ptr;
ptr = new int [5];    // array of 5 integer elements
ptr[3] = 10;          // using the [ ] operator
```


Dynamic Arrays (cont.)

- The size of a dynamic array does not need to be known at compile time:

```
int numElements;
cout << "How many elements would you like?";
cin >> numElements;
float *ptrArr = new float [ numElements ];
```


What Happens at the End of This Function?

```
void foo( )
{
    int numElements;
    cout << "How many elements would you like the array
              to have? ";
    cin >> numElements;
    float *ptrArr = new float [ numElements ];

    // the array is processed here
    // output to the user is provided here
}
```


Memory Leak

- All local variables and the values they contain are destroyed (numElements and ptrArr)
- The address of the dynamic array is lost
- BUT…the dynamic array is not destroyed
- The dynamic array can no longer be used, but the new operator will consider it as used heap memory (and cannot reuse it for something else).
- This is called ***memory leak.***
- Memory leak is not permanent – it will end when the program stops.

Memory Leak (cont.)

- Memory leak can easily be prevented during execution of a program.
- A program that continually leaks memory may run out of heap memory to use.
- Therefore, it is poor programming practice to allow memory leak to occur.

The delete Operator

```
void foo( )
{
      int numElements;
      cout <<  "How many elements would you like the array
                to have? ";
      cin >> numElements;
      float *ptrArr = new float [ numElements ];

      // the array is processed here
      // output to the user is provided here

      delete [ ] ptrArr;
}
```

Prevents memory leak – it frees the dynamic array so this memory can be reused by the new operator later on

The delete Operator (cont.)

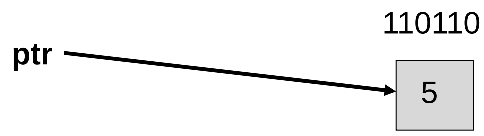
- When the delete operator is being used to free a variable pointed to by ptr:

  delete ptr;
- When the delete operator is being used to free an array pointed to by ptr:

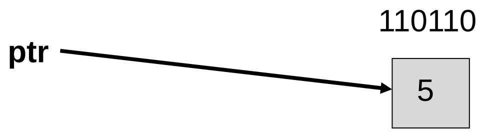
  delete [ ] ptr;
- If you omit [ ] (common mistake), no compiler error will be given; however, only the first element of the array will be freed

Another Common Cause of Memory Leak

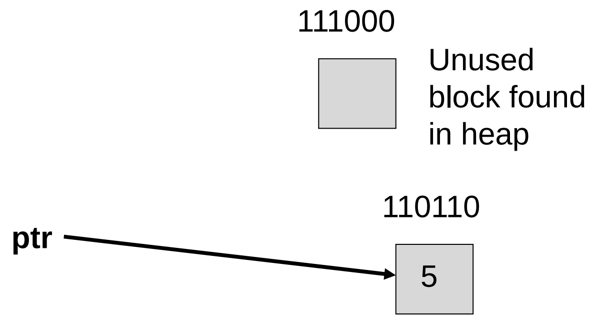
First, pointer ptr is assigned the address of dynamically allocated memory.



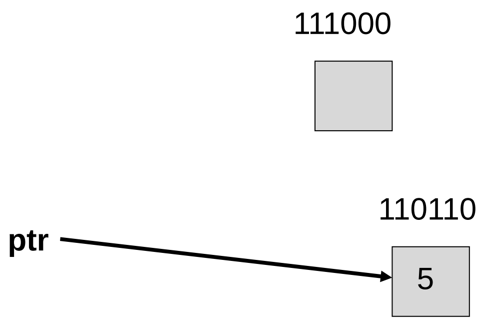
ptr = new int;
(what happens?)



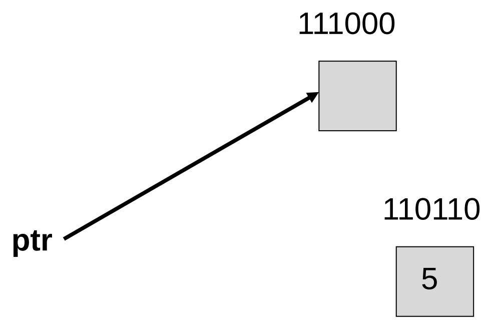
ptr = new int;



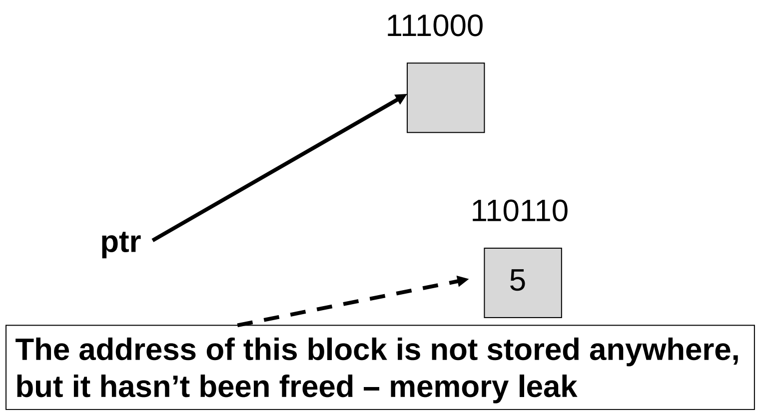
ptr = 111000;



ptr = 111000;



ptr = 111000;



Avoiding Memory Leak

- When you want to change the address stored in a pointer, always think about whether or not the current address is used for dynamically-allocated memory
- If it is (and that memory is no longer useful), use the delete operator before changing the address in the pointer; for example,

```
delete ptr;
```


Pointers to Objects

- Pointers can be used to point to objects
- When ***new*** is called, the constructor is also invoked automatically.

  Rational *p = **new** Rational;

  declare a pointer variable ***p*** , allocates a new object of type *Rational,* initializes it to 0 and set ***p*** point to

  Rational *p = **new** Rational(2,5);

  initializes it to 2/5

- The delete operator is used the same way:
  delete p;

Running out of Heap Memory

- We can run out of heap memory if:
  - We write poor code that continually allows memory leak while constantly using the new operator
  - OR …
  - If we use excessive amounts of heap memory
- If the new operator cannot find a chunk of unused heap memory large enough for the data type, the new operation ***throws an exception***

Code for Exceptions

```
int main( )
{
   char *ptr;

   try {
          ptr = new char[ 1000000000 ];
   }

   catch( … ) {
      cout << "Too many elements" << endl;
   }

   return 0;
}
```

try clause

catch clause

Code for Exceptions (cont.)

```
int main( )
{
   char *ptr;

   try {
          ptr = new char[ 1000000000 ];
   }

   catch( … ) {
      cout << "Too many elements" << endl;
   }

   return 0;
}
```

The program will crash if try/catch clauses are not written for code that ultimately causes an exception

Another Example

```
int main( )
{
        Foo foo;
        try {
                foo.bar1( 35 );
                foo.bar2( 10 );
                foo.bar3( );
        }
        catch ( … )  {
                cout << "Out of memory" << endl;
        }
return 0;
}
```

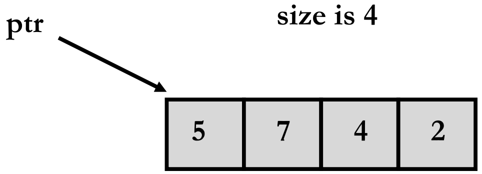
These functions use the new operator.

Another Example (cont.)

```
int main( )
{
        Foo foo;
        try {
                foo.bar1( 35 );
                foo.bar2( 10 );
                foo.bar3( );
        }
        catch ( … )  {
                cout << "Out of memory" << endl;
        }
return 0;
}
```

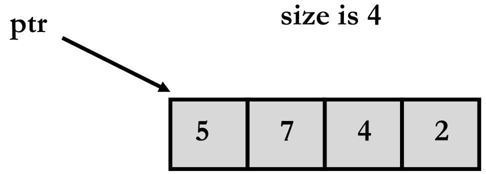
The client usually writes the exception-handling code to do whatever they wish to do.

Array Expansion

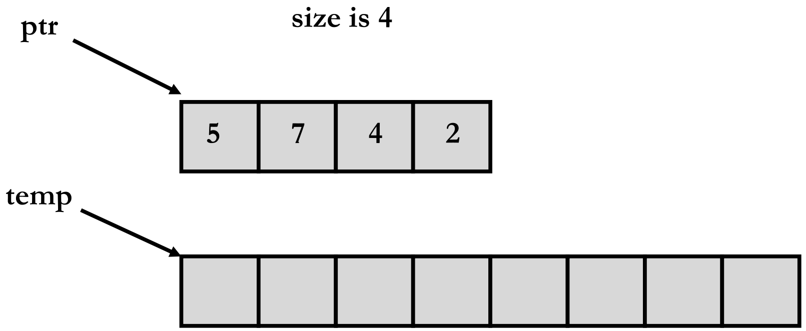


A dynamic array is filled, and more data needs to be put in. Therefore, the array needs to be expanded.

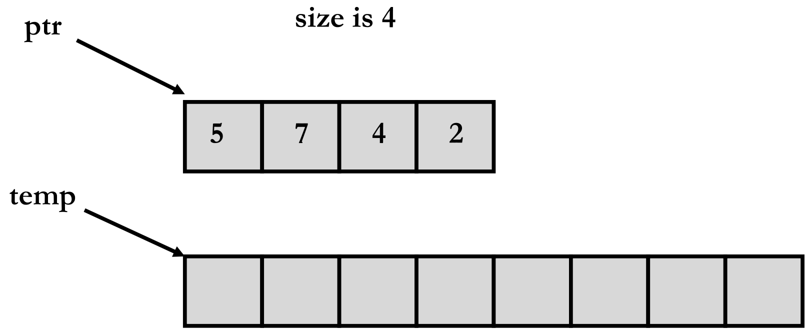

```
int *temp = new int [size * 2];
```



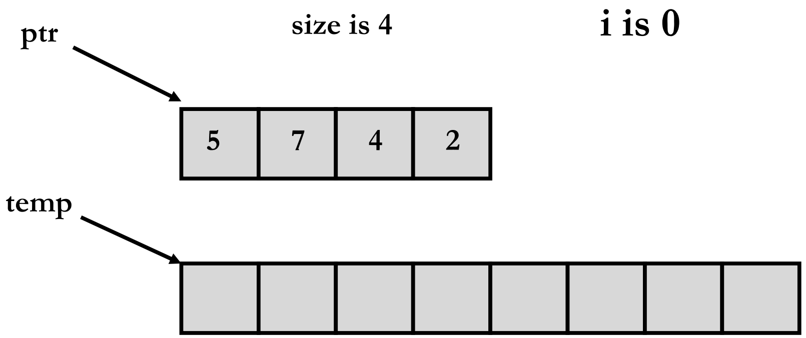

```
int *temp = new int [size * 2];
```



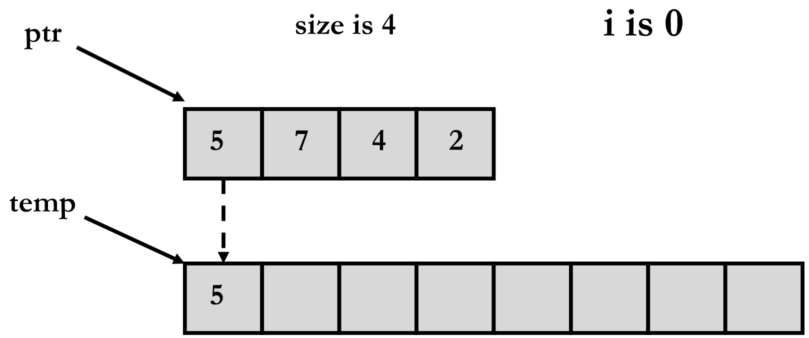

```
for ( int i = 0; i < size; i++ )
    temp[ i ] = ptr[ i ];
```



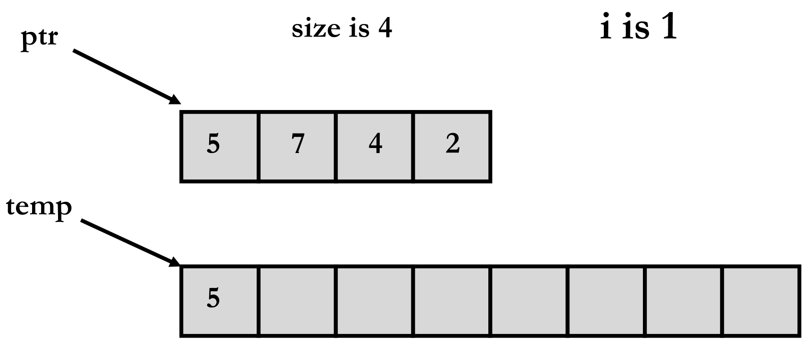
```
for ( int i = 0; i < size; i++ )
    temp[ i ] = ptr[ i ];
```



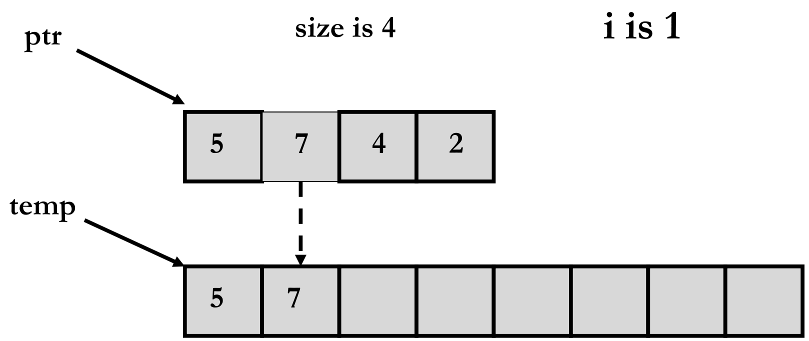

```
for ( int i = 0; i < size; i++ )
    temp[ i ] = ptr[ i ];
```



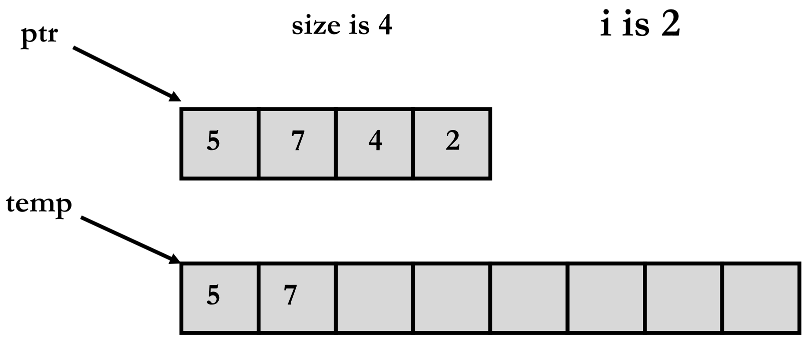

```
for ( int i = 0; i < size; i++ )
    temp[ i ] = ptr[ i ];
```



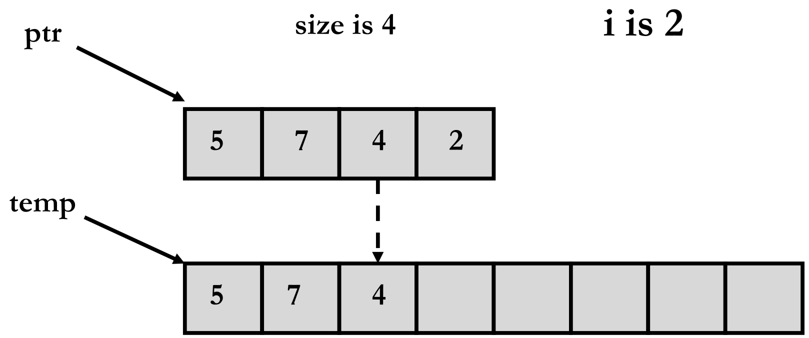

```
for ( int i = 0; i < size; i++ )
    temp[ i ] = ptr[ i ];
```



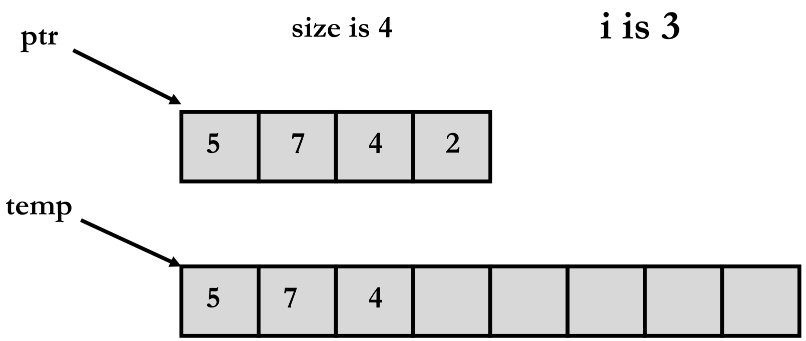
```
for ( int i = 0; i < size; i++ )
    temp[ i ] = ptr[ i ];
```



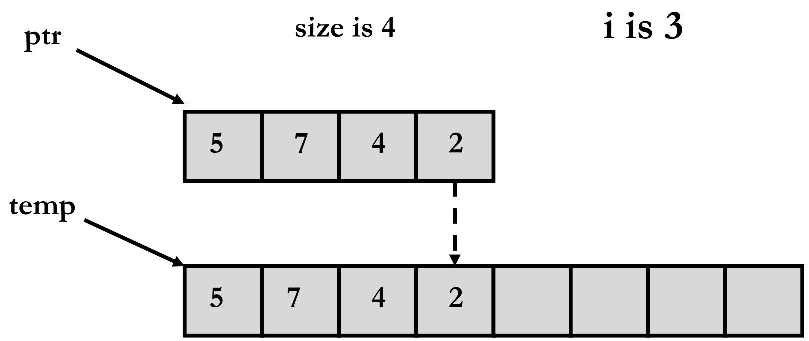

```
for ( int i = 0; i < size; i++ )
    temp[ i ] = ptr[ i ];
```



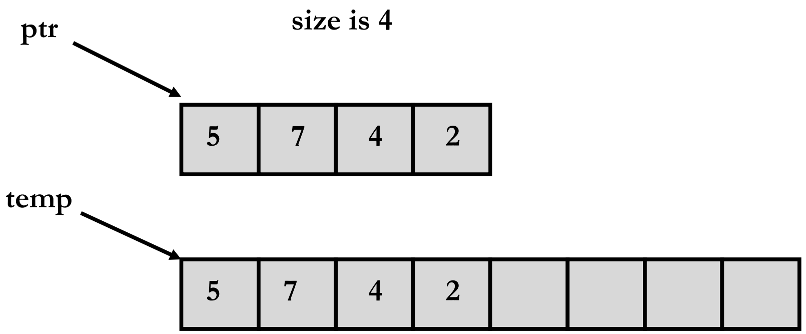

```
for ( int i = 0; i < size; i++ )
    temp[ i ] = ptr[ i ];
```



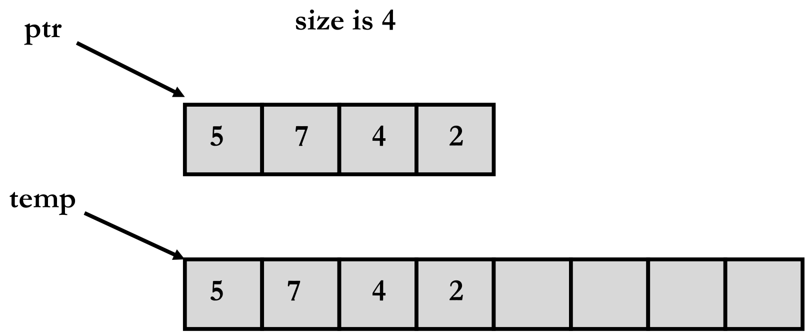

```
for ( int i = 0; i < size; i++ )
    temp[ i ] = ptr[ i ];
```



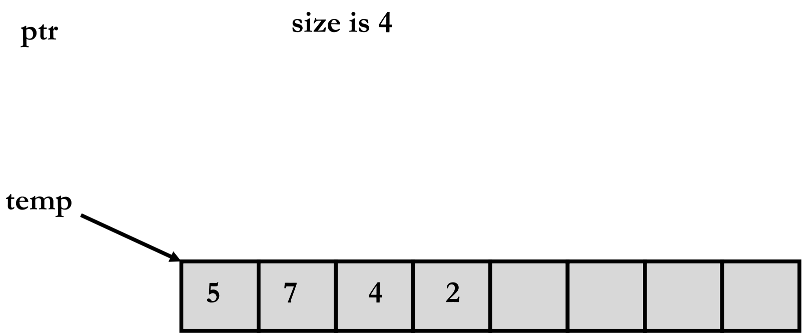
```
for ( int i = 0; i < size; i++ )
    temp[ i ] = ptr[ i ];
```



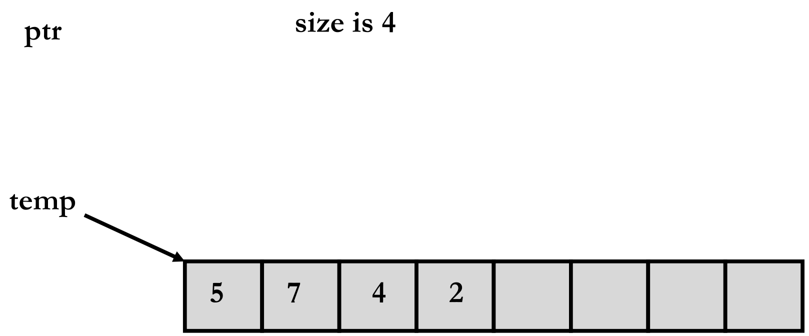

```
delete [ ] ptr;
```



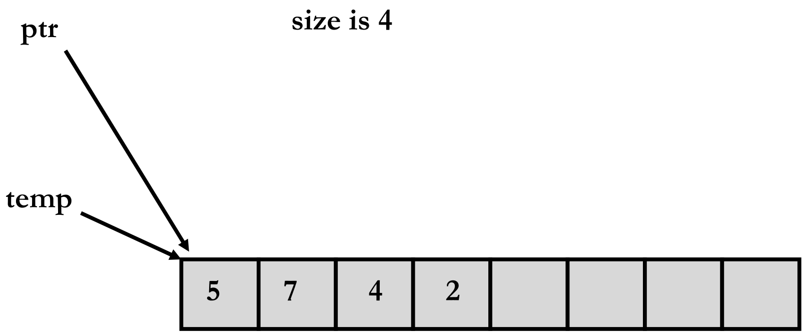

```
delete [ ] ptr;
```



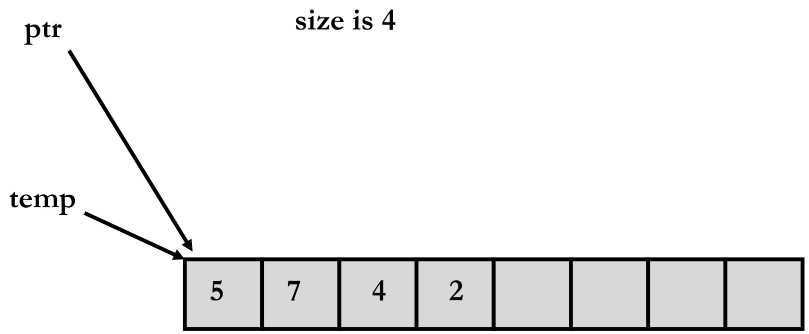

```
ptr = temp;
```



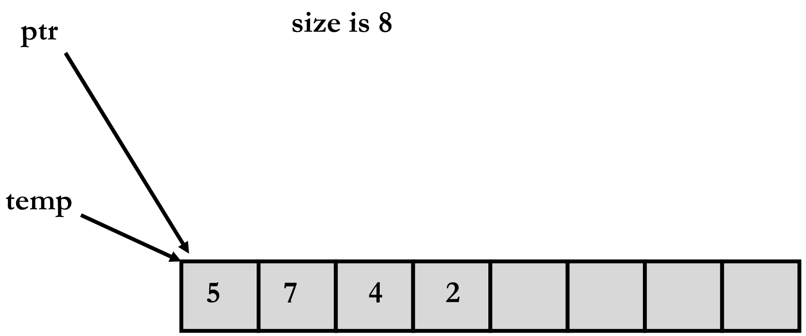
ptr = temp;



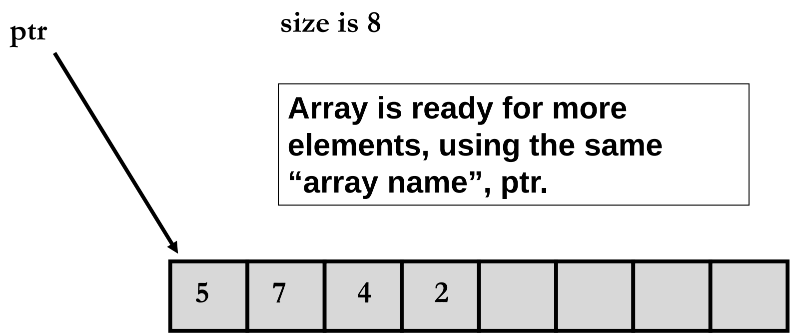
size = size * 2;



size = size * 2;



Expansion Completed



Pointers to Objects

- Pointers can be used to point to objects
- When ***new*** is called, the constructor is also invoked automatically.

```
Rational  *p = new Rational;
```

declare a pointer variable ***p*** , allocates a new object of type *Rational,* initializes it to 0 and set ***p*** point to

```
Rational  *p = new Rational(2,5);
```

initializes it to 2/5

- The delete operator is used the same way:

```
delete p;
```


The *this* Pointer

- Each object maintains a pointer to itself which is called the "***this***" pointer.
- Each object can determine it's own address by using the "***this***" keyword.
- *Implicit* parameter passed to a member function (by the compiler)

Using the *this* Pointer

- Can be used to access members that may be hidden by parameters with same name:

```
class SomeClass
{
  private:
          int num;
  public:
          void setNum(int num)
         { this->num = num; }
};
```

Using the *this* Pointer

- Can use it to reference the address of an object

```
void X::printObjectDetails( )
{
      cout << "The object at address " << this
      cout << " has value " << (*this).y << endl;
}
```

- Suppose ***a*** is an object of class X, when we execute

```
a.printObjectDetails
```

the *address* of ***a*** and the *value* of one of its members ***y*** are printed

Using the *this* Pointer

- It may seem redundant but the “this” pointer does have some uses:
  - Prevents an object from being assigned to itself.
  - Enables cascading member function calls.

The const Specifier

- When added to the end of a function heading, it tells the compiler that no changes should be made to any private members during the execution of that function

  int getX() const; (in the class definition)

  int X::getX() const (defined outside the class)

The const Specifier (cont.)

- ***const*** can also be used for parameters

  int setNum (const int num)
- Objects are often passed by reference for speed
  - in pass by value, it can take a long time to copy
  - in pass by reference, only the address is copied
- The use of ***const*** here specifies that the parameter should not change – called passing by ***const reference***

Rules for Passing Objects

- ***Pass objects by value*** when the function will change them and you don’t want the change to be reflected to the caller
- ***Pass objects by reference*** when you want changes to be reflected to the caller
- ***Pass objects by const reference*** for speed when objects won’t be changed – the compiler will catch mistaken changes