POINTERS

OVERVIEW

OVERVIEW

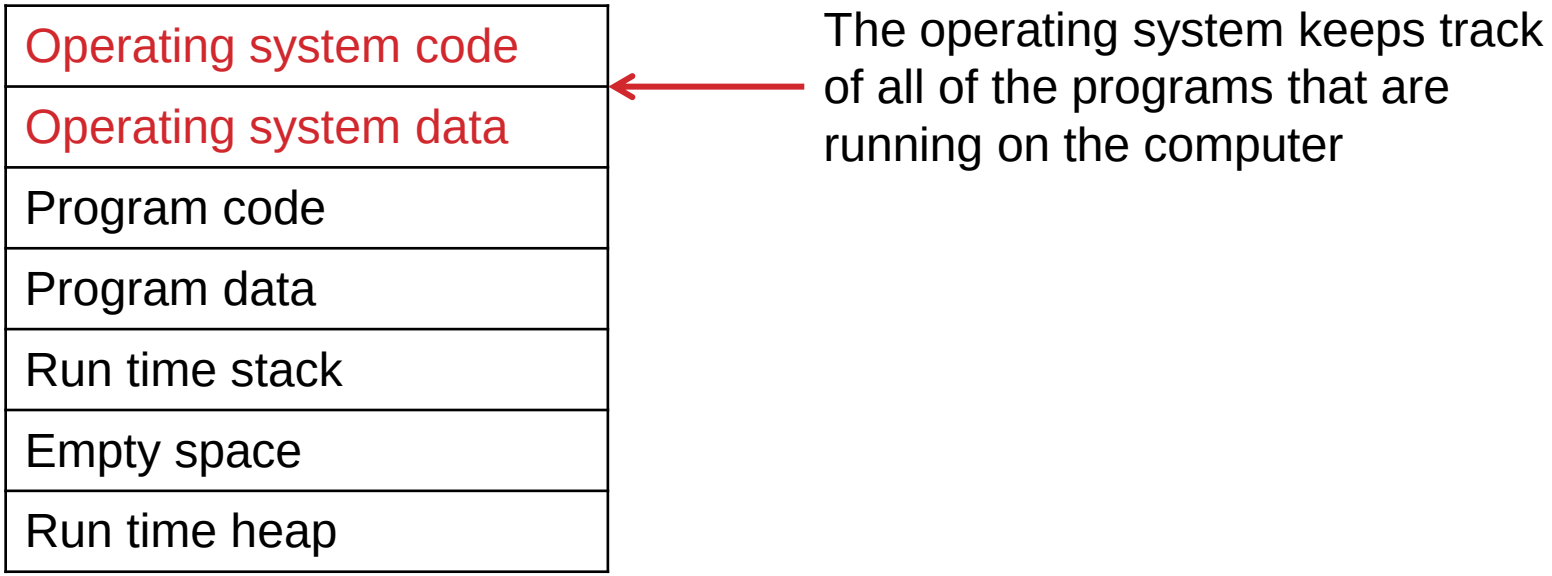
- **Pointers are special variables that can contain the memory address of another variable**

- **We can use pointers to access and modify variables**
  - Special C++ syntax to get address and follow pointer

- **We can also create dynamic data structures that grow and shrink with the needs of an application**
  - Special C++ commands to allocate and release memory

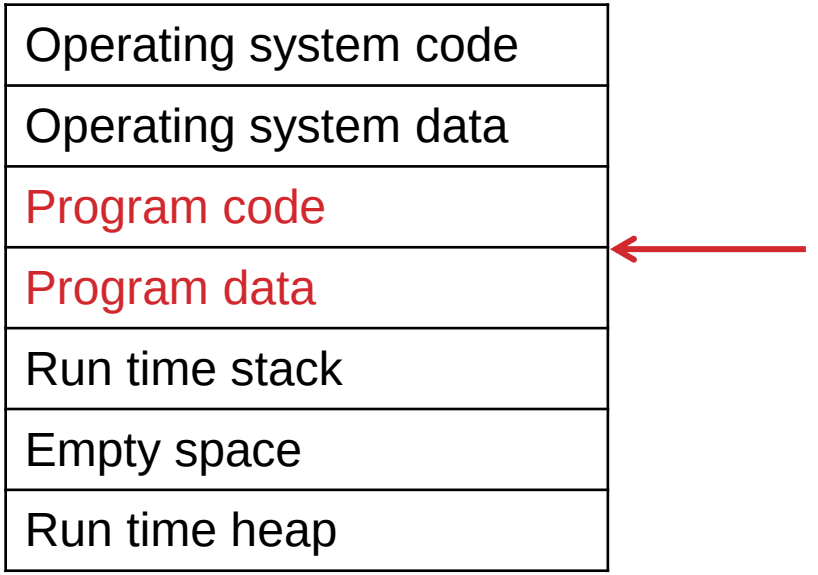
OVERVIEW

- **Memory on a computer can be viewed as a very long array divided into seven parts**



OVERVIEW

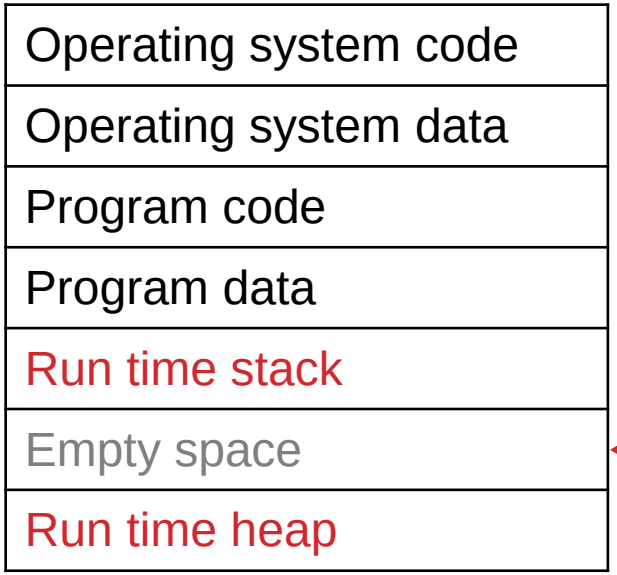
- **Memory on a computer can be viewed as a very long array divided into seven parts**



Each program is given a piece of memory to store the code and variables

OVERVIEW

- **Memory on a computer can be viewed as a very long array divided into seven parts**



The stack and heap grow and shrink in size as program runs providing memory for variables

OVERVIEW

- **Lesson objectives:**
  - Learn the basic syntax for pointer variables
  - Learn how to use pointers to access/modify variables
  - Learn about the run time stack and run time heap
  - Learn how to allocate and release dynamic memory
  - Study example program with dynamic memory
  - Study example class with dynamic memory

POINTERS

PART 1

POINTER BASICS

POINTER VARIABLES

- **A pointer variable allows us to store the memory address of another variable in the program**
  - We can them use this pointer variable to access and modify the original variable
  - We can also use pointer variables to create dynamic data structures that grow/shrink with needs of program

- **Pointers can only "point to" variables of one data type**
  - An integer pointer can store address of an integer variable
  - A float pointer can store address of a float variable
  - A char pointer can store address of a char variable

POINTER VARIABLES

- **We declare pointer variables using a * operator between he data_type and the variable_name**
  - int * ptr1 – will declare an integer pointer
  - float * ptr2 – will declare a float pointer
  - char * ptr3 – will declare a char pointer

- **It does not matter where the * is placed**
  - int* ptr1 – it can go next to the data_type
  - float *ptr2 – it can go next to the variable_name
  - char * ptr3 – it can go half way in between

ADDRESS OPERATOR

- **We need some way to obtain the address of variables**
  - The address operator & gives address of following variable
  - We can store this address in a pointer variable

```
int a = 42;
int b = 17;

int *aptr = &a;
int *bptr = &b;
```

aptr now contains address of a

bptr now contains address of b

ADDRESS OPERATOR

- **Every time we run a program, the operating system will give our program a different portion of memory to run in**
  - This means that the memory addresses we get with the address operator could be different every time!

```
int a = 42;
int b = 17;

int *aptr = &a;
int *bptr = &b;
```

First time we run program aptr = 1000
Next time we run program aptr = 1234
We can not predict this value

ADDRESS OPERATOR

- **Variables are assigned to consecutive memory locations**
  - The differences between variable memory addresses will be the same every time we run the program

```
int a = 42;
int b = 17;

int *aptr = &a;
int *bptr = &b;
```

Assume aptr = 1000, then bptr = 1004
The difference will always be 4

ADDRESS OPERATOR

- **We can print addresses just like any other variable**

```
cout << aptr << endl;
cout << &b << endl;
```

- **Memory addresses are normally printed in hexadecimal**
  - Starts with “0x” to indicate value is in hexadecimal
  - Followed by N hexadecimal digits (0123456789abcdef)
  - Linux uses 12 hexadecimal digits (48-bits) for addresses

Eg: 0x7fff541aec3c
0x7fff541aec38

INDIRECTION OPERATOR

- **We need some way to "follow pointers" to variables**
  - The indirection operator * is used to do this
  - When we put * in front of a pointer variable we can access or modify the variable at that memory location

```
float n = 3.14;
float *ptr = &n;

*ptr = 1.23;
```

*ptr is another way to access the variable n, so now n equals 1.23

```
cout << n << endl;          // will print 1.23
```

INDIRECTION OPERATOR

- **It is possible to have multiple pointers to one variable**
  - We must assign one pointer value (address) to another pointer variable of the same type

```
float n = 3.14;
float *ptr = &n;
float *ptr2 = ptr;
```

```
*ptr2 = 8.76;
*ptr = 5.67;
n = 1.41;
```

We now have three ways to change the value of variable n

ARRAYS AND POINTERS

- **Arrays and pointers are actually very similar in C++**
  - When we declare “int data[10]” the variable “data” stores the address of the first array element
  - This means that the indirection operator * can be used with the array name to access variables in an array

```
int data[10];
data[0] = 24;   <-- This will store 24 in data[0]
*data = 42;     <-- This will store 42 in data[0]
data[1] = 17;   <-- This will store 17 in data[1]
*data = 72      <-- This will store 72 in data[0]
```

ARRAYS AND POINTERS

- **We can also use array subscripts [ ] and pointer variables to access and modify variables**
  - We have to be very careful with array bounds

```
int data[10];
int *ptr = data;
ptr[3] = 77;
```

ptr will now point at data array

This will store 77 in data[3]

```
ptr[12] = 0;
ptr[-3] = 17;
```

Both will go out of array bounds and cause bugs

ARRAYS AND POINTERS

- **It is possible to do some very sneaky programming using arrays and pointers**
  - When we add one to a pointer variable, we point to adjacent variable of the same data type
  - This pointer arithmetic gives us another way to access variables in an array (very ugly and not recommended)

```
int data[10];
int *ptr = data;
*(ptr+3) = 77;
```

This is the same as data[3] = 77;

SUMMARY

- **In this section, we went over the syntax for declaring and using pointers in C++**
  - We declare pointer variables by adding * between the data_type and variable_name
  - The address operator & is used to get the current memory address of a variable
  - The indirection operator * is used to “follow” a pointer and access the variable at that address
  - Arrays and pointers are similar in several ways and can be accessed using similar notation

POINTERS

PART 2

DYNAMIC DATA STRUCTURES

RUN TIME STACK

- **The run time stack is used to give a program space for function parameters and local variables**
  - The stack “grows” in size for every function call
  - Growth size depends on the number and types of the parameters and local variables
  - The stack “shrinks” in size when the function finishes

- **If there is infinite recursion in a program, the stack will use up all available empty space and the program will die with a “stack overflow” error**

RUN TIME STACK

- **Consider the following program**

```
int process(int number)
{
   int result = (number + 5) / 2;
   return result;
}
```

The process function has one integer parameter and one integer variable and needs 8 bytes

```
int main()
{
   int value = process(17);
   cout << value;
   return 0;
}
```

RUN TIME STACK

- **Consider the following program**

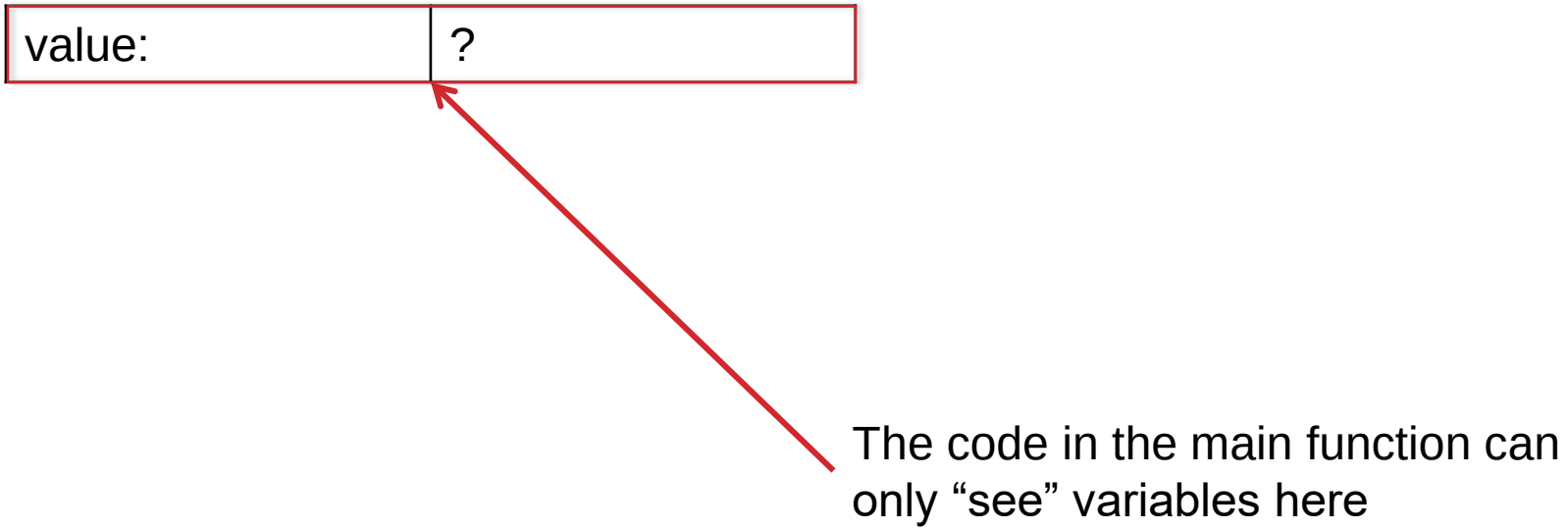
```
int process(int number)
{
   int result = (number + 5) / 2;
   return result;
}
```

```
int main()
{
   int value = process(17);
   cout << value;
   return 0;
}
```

The main function has one integer variable and needs 4 bytes

RUN TIME STACK

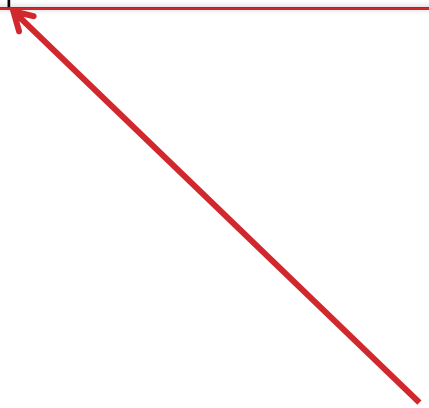
- **When the program starts, the stack grows to contain space for variables in the “main” function**



RUN TIME STACK

- **When the function “process” is called, the stack grows to contain space for variables in this function**

value:	?
number:	17
result:	11



The code in the process function can only “see” variables here

RUN TIME STACK

- **When the function “process” returns its result, we no longer need the space for these local variables, so the stack shrinks in size**

value:	11

RUN TIME HEAP

- **The run time heap is used to give a program space for dynamic variables**
  - The user can “allocate” space on the heap for a variable
  - The size of the variable can be decided at run time to exactly meet the needs of the application
  - The user must “release” space to the heap when finished using the dynamic variable

- **If space is not returned properly, a program can have a “memory leak” and may run out of space and die**

DYNAMIC MEMORY ALLOCATION

- **The “new” command allocates memory**
  - We allocate space for one variable using “new data_type”
  - This will return the address of the allocated memory
  - We use the indirection operator * to access this variable

```
float *ptr;
ptr = new float;
*ptr = 42;
```

DYNAMIC MEMORY ALLOCATION

- **The "new" command can also allocate space for arrays**
  - We have to specify the data_type
  - We also have to specify how much space to allocate
  - The syntax is similar to how we declare arrays
  - We can access this dynamic memory using [ ] notation

```
int * ptr;
ptr = new int[10];       // work same "int ptr[10]"
for (int i=0; i<10; i++)
   ptr[i] = 42 + i;
```

RELEASING DYNAMIC MEMORY

- **The "delete" command releases memory**
  - When we are finished using a dynamic variable we must release its memory using "delete pointer_name"
  - The operating system will then add this memory to the "empty space" between the stack and the heap

```
float *ptr;
ptr = new float;
*ptr = 42;
…
delete ptr;
ptr = NULL;
```


You should never attempt to use *ptr after you have called delete because you no longer "own" this memory

RELEASING DYNAMIC MEMORY

- **The "delete [ ]" command also releases memory**
  - When we are finished using a dynamic array we must release its memory using "delete [ ] pointer_name"
  - The operating system will then add this memory to the "empty space" between the stack and the heap

```
int * ptr;
ptr = new int[10];
…
delete [ ] ptr;
ptr = NULL;
```

You should never attempt to use ptr[i] after you have called delete because you no longer "own" this memory

SAMPLE PROGRAM

- **Assume we are given an ascii file containing an unknown number of integer values and we want to sort this data**
  - We do not want to “guess” the size of data array
  - Guess too high → waste memory space
  - Guess too low → program fails to work properly

- **Our algorithm:**
  - Read input file to count how many values are in file
  - Allocate a dynamic array large enough for this data
  - Read data from input file into the dynamic array
  - Perform sorting algorithm on data in array
  - Print sorted data and release memory

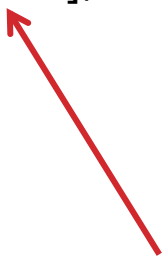
SAMPLE PROGRAM

```
// Read input file to count values
ifstream din;
din.open("numbers.txt");
int count = 0;
int number = 0;
while (din >> number)
   count++;
din.close();
```

This read operation returns “true” if data is read correctly and “false” when we reach the end of file

SAMPLE PROGRAM

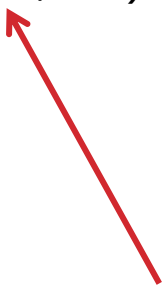
```
// Allocate dynamic array
int * data;
data = new int[count];
```



The count variable was initialized above so we can allocate exactly the right size array to process this data

SAMPLE PROGRAM

```
// Read data into dynamic array
din.open("numbers.txt");
for (int i=0; i<count; i++)
   din >> data[i];
din.close();
```



We can read data in a for loop without checking for end of file because we know exactly how many values there are in the input file

SAMPLE PROGRAM

```
// Sort data using bubble sort
for (int i=0; i<count; i++)
   for (int j=1; j<count; j++)
      if (data[j-1] > data[j])
      {
         int temp = data[j-1];
         data[j-1] = data[j];
         data[j] = temp;
      }
```

SAMPLE PROGRAM

```
// Print sorted data and release memory
for (int i=0; i<count; i++)
   cout << data[i] << " ";
cout << endl;
delete [ ] data;
```

We are finished with the data array so we can release this memory to the “free space”

SUMMARY

- **The run time stack is used for variables in functions**
  - Grows and shrinks as we call functions and return
- **The run time heap is used for dynamic variables**
  - Grows and shrinks as we allocate and release memory

- **The "new" command allocates space on heap**
  - int *ptr = new int[10];
- **The "delete" command releases memory**
  - delete [ ] ptr;

POINTERS

PART 3

POINTERS IN CLASSES

POINTERS IN CLASSES

- **Pointers and dynamic memory allocation are often used in classes to create dynamic abstract data types (ADTs)**
  - Memory is allocated in constructor methods
  - Memory is released in destructor methods
  - This way the ADT can grow/shrink as needed
  - This approach will help us avoid memory leaks

POINTERS IN CLASSES

- **Dynamic ADTs fall into two categories**
  - Array based
    - Where a dynamic array grows/shrinks to store data
    - The C++ “vector” class uses this approach
  - Node based
    - Where data is stored in a collection of “nodes”
    - Pointers are used to link these nodes together
    - “linked lists” and “binary trees” use this approach
    - These ADTs will be studied in detail in PF2

MYARRAY CLASS

- **The "MyArray" class demonstrates use of dynamic array**
  - The "Data" variable contains a pointer to dynamic memory
  - The "Size" variable contains size of dynamic memory

```
class MyArray
{
private:
   int * Data;
   int Size;
…
```

MYARRAY CLASS

- **Access to this memory will be provided by class methods**

```
…
public:
   MyArray(const int size = 10);
   MyArray(const MyArray & array);
   ~MyArray();

   int Get(const int index);
   void Set(const int index, const int value);
   void Resize(const int size);
};
```

CONSTRUCTOR

- **If we want to create an ADT using a dynamic array we must allocate memory in the constructor method**

```
MyArray::MyArray(const int size)
{
   Size = size;
   Data = new int[Size];
   for (int i=0; i<Size; i++)
      Data[i] = 0;
}
```

Data = new int[Size]; ← Allocate memory

Data[i] = 0; ← Initialize memory

COPY CONSTRUCTOR

- **The copy constructor method for a dynamic array ADT must allocate memory and copy array data**

```
MyArray::MyArray(const MyArray & array)
{
   Size = array.Size;
   Data = new int[Size];          <-- Allocate memory
   for (int i=0; i<Size; i++)
      Data[i] = array.Data[i];    <-- Copy private Data
}
```

DESTRUCTOR

- **The destructor method is automatically called when an object is no longer "in scope" and no longer needed**

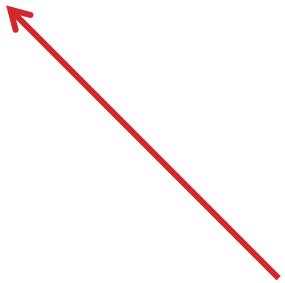
```
MyArray::~MyArray()
{
   delete [ ] Data;
   Size = 0;
   Data = NULL;
}
```

Setting the pointer to NULL makes it clear to users of this class that the array can no longer be accessed

GET METHOD

- **To access data in the the array, we use Get method**

```
int MyArray::Get(const int index)
{
  if ((index >= 0) && (index < Size))
      return Data[index];
  else
      return -1;
}
```

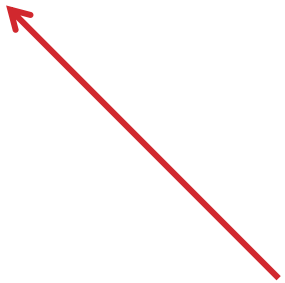


We perform error checking on index before we access the Data

GET METHOD

- **To access data in the the array, we use Get method**

```
bool MyArray::Get(const int index, int & value)
{
  if ((index >= 0) && (index < Size))
  {   value = Data[index];
      return true; }
  else
      return false;
}
```

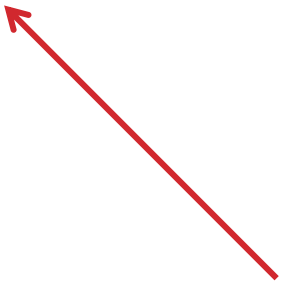


We perform error checking on index before we access the Data

GET METHOD

- **To access data in the the array, we use Get method**

```
bool MyArray::Get(const int index, int & value)
{
  if ((index < 0) || (index >= Size))
     return false;
  value = Data[index];
  return true;
}
```

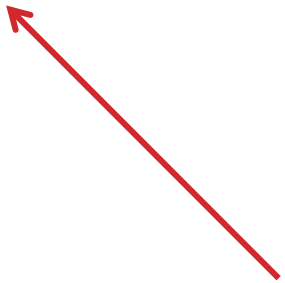


We perform error checking on index before we access the Data

SET METHOD

- **To store data in the the array, we use Set method**

```
void MyArray::Set(const int index, const int value)
{
  if ((index >= 0) && (index < Size))
     Data[index] = value;
}
```



We perform error checking on index before we access the Data

RESIZE METHOD

- **To resize the array, we must allocate a new array of desired size and copy all of the old data into new array**

```
void MyArray::Resize(const int size)
{
   // allocate and initialize new array
   int * OldData = Data;
   Data = new int[size];
   for (int i=0; i<size; i++)
      Data[i] = 0;
```

We save a copy of the Data pointer so we can still access the old data

Allocate new space and store address in Data pointer

RESIZE METHOD

```
 …
// copy old data into new array
for (int i=0; i<size && i<Size; i++)
      Data[i] = OldData[i];


// release old memory
Size = size;
delete [ ] OldData;
}
```

This will result in data loss if the new size is less than old Size

We return memory for the original Data array back to the heap

USING MYARRAY

```
#include "MyArray.h"
int main()
{
   // Create array
   int size = 0;
   cout << "Enter array size:";
   cin >> size;
   MyArray data(size);

   // Set array values
   for (int index = 0; index < size; index++)
      data.Set(index, random() % 17);
```

USING MYARRAY

```
…
// Print array values
for (int index = 0; index < size; index++)
   cout << data.Get(index) << " ";
cout << endl;

// Resize the array
cout << "Enter new array size: ";
cin >> size;
data.Resize(size);
```

USING MYARRAY

```
    …
    // Update array values
    for (int index = 0; index < size; index++)
       data.Set(index, data.Get(index) + 42);

    // Print array values
    for (int index = 0; index < size; index++)
       cout << data.Get(index) << " ";
    cout << endl;
    return 0;
}
```

The destructor function is called automatically here

SUMMARY

- **The MyArray class could be extended to include a wide variety of operations**
  - Min / Max / Mean / Standard Deviation methods
  - Searching / Sorting / Median methods
  - Input / Output methods

- **Similar dynamic arrays have been created for signal processing and image processing applications**
  - Grow to fit the size of input audio file or image
  - Contain hundreds of specialized operations

SUMMARY

- **Pointers and dynamic memory allocation are often used in classes to create dynamic abstract data types (ADTs)**
  - Memory is allocated in constructor methods
  - Memory is released in destructor methods
  - This way the ADT can grow/shrink as needed
  - This approach will help us avoid memory leaks

- **In this section, we described how an "array based" dynamic ADT can be implemented in C++**
- **In future classes you will learn about "node based" dynamic ADTs (linked lists, binary trees, etc)**


The End
(For
Now)