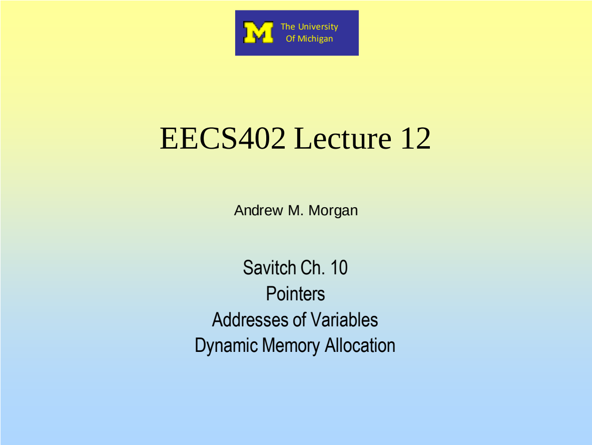
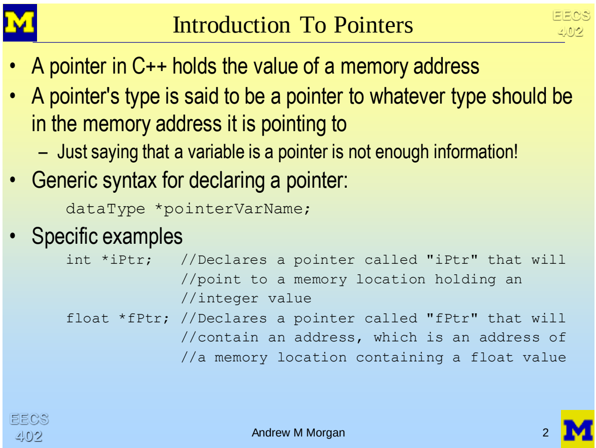


EECS402 Lecture 12

Andrew M. Morgan

Savitch Ch. 10
Pointers
Addresses of Variables
Dynamic Memory Allocation

1



Introduction To Pointers

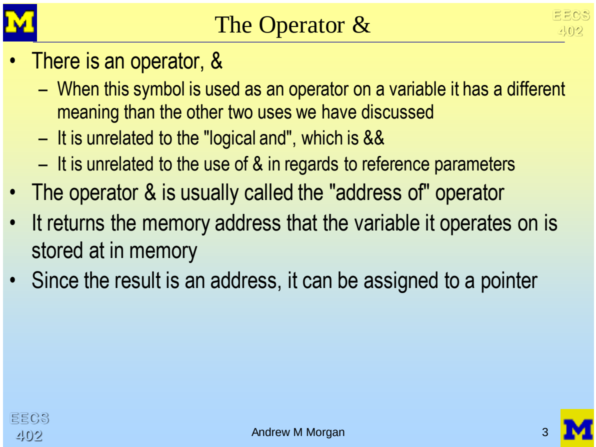
- A pointer in C++ holds the value of a memory address
- A pointer's type is said to be a pointer to whatever type should be in the memory address it is pointing to
  - Just saying that a variable is a pointer is not enough information!
- Generic syntax for declaring a pointer:

```
dataType *pointerVarName;
```

- Specific examples

```
int *iPtr;    //Declares a pointer called "iPtr" that will
              //point to a memory location holding an
              //integer value
float *fPtr;  //Declares a pointer called "fPtr" that will
              //contain an address, which is an address of
              //a memory location containing a float value
```

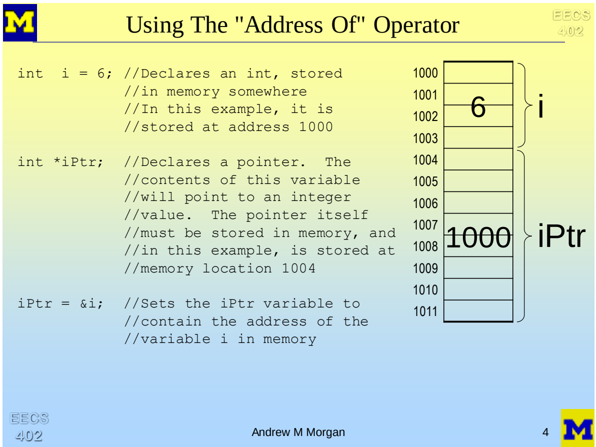
2



The Operator &

- There is an operator, &
  - When this symbol is used as an operator on a variable it has a different meaning than the other two uses we have discussed
  - It is unrelated to the "logical and", which is &&
  - It is unrelated to the use of & in regards to reference parameters
- The operator & is usually called the "address of" operator
- It returns the memory address that the variable it operates on is stored at in memory
- Since the result is an address, it can be assigned to a pointer

3

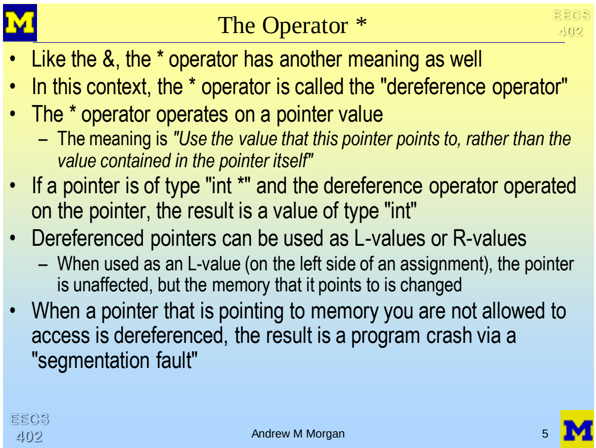


Using The "Address Of" Operator

```
int  i = 6; //Declares an int, stored
            //in memory somewhere
            //In this example, it is
            //stored at address 1000

int *iPtr;  //Declares a pointer.  The
            //contents of this variable
            //will point to an integer
            //value.  The pointer itself
            //must be stored in memory, and
            //in this example, is stored at
            //memory location 1004

iPtr = &i;  //Sets the iPtr variable to
            //contain the address of the
            //variable i in memory
```

1000		i
1001	6	
1002		
1003		
1004		iPtr
1005		
1006		
1007	1000	
1008		
1009		
1010		
1011		

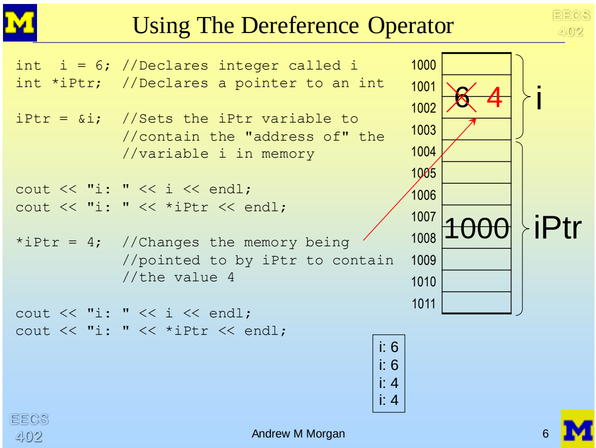
4



The Operator *

- Like the &, the * operator has another meaning as well
- In this context, the * operator is called the "dereference operator"
- The * operator operates on a pointer value
  - The meaning is *"Use the value that this pointer points to, rather than the value contained in the pointer itself"*
- If a pointer is of type "int *" and the dereference operator operated on the pointer, the result is a value of type "int"
- Dereferenced pointers can be used as L-values or R-values
  - When used as an L-value (on the left side of an assignment), the pointer is unaffected, but the memory that it points to is changed
- When a pointer that is pointing to memory you are not allowed to access is dereferenced, the result is a program crash via a "segmentation fault"

5



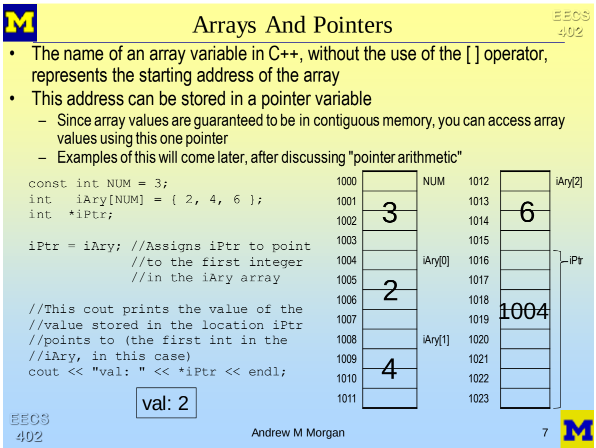
Using The Dereference Operator

```
int  i = 6; //Declares integer called i
int *iPtr;  //Declares a pointer to an int

iPtr = &i;  //Sets the iPtr variable to
            //contain the "address of" the
            //variable i in memory

cout << "i: " << i << endl;
cout << "i: " << *iPtr << endl;

*iPtr = 4;  //Changes the memory being
            //pointed to by iPtr to contain
            //the value 4

cout << "i: " << i << endl;
cout << "i: " << *iPtr << endl;
```

1000		i
1001	~~6~~ 4	
1002		
1003		
1004		iPtr
1005		
1006		
1007	1000	
1008		
1009		
1010		
1011		

```
i: 6
i: 6
i: 4
i: 4
```

6



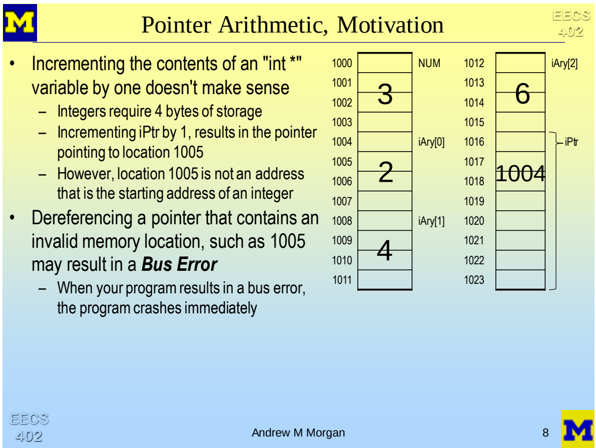
Arrays And Pointers

- The name of an array variable in C++, without the use of the [ ] operator, represents the starting address of the array
- This address can be stored in a pointer variable
  - Since array values are guaranteed to be in contiguous memory, you can access array values using this one pointer
  - Examples of this will come later, after discussing "pointer arithmetic"

```
const int NUM = 3;
int   iAry[NUM] = { 2, 4, 6 };
int  *iPtr;

iPtr = iAry; //Assigns iPtr to point
             //to the first integer
             //in the iAry array

//This cout prints the value of the
//value stored in the location iPtr
//points to (the first int in the
//iAry, in this case)
cout << "val: " << *iPtr << endl;
```

```
val: 2
```

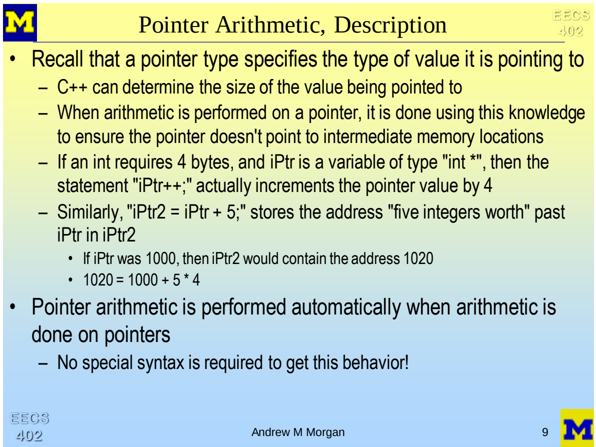
1000		NUM	1012		iAry[2]
1001	3		1013	6	
1002			1014		
1003			1015		
1004		iAry[0]	1016		iPtr
1005	2		1017		
1006			1018	1004	
1007			1019		
1008		iAry[1]	1020		
1009	4		1021		
1010			1022		
1011			1023		



Pointer Arithmetic, Motivation

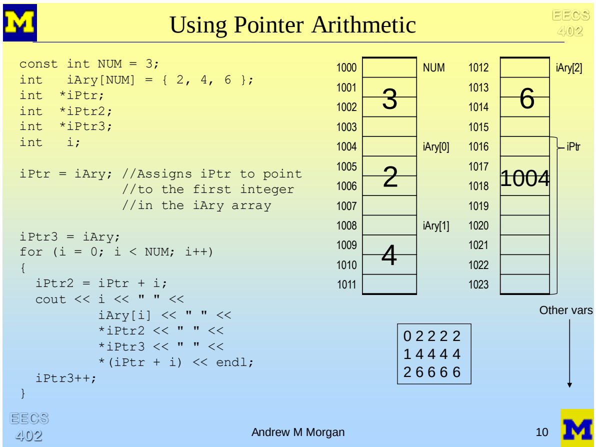
- Incrementing the contents of an "int *" variable by one doesn't make sense
  - Integers require 4 bytes of storage
  - Incrementing iPtr by 1, results in the pointer pointing to location 1005
  - However, location 1005 is not an address that is the starting address of an integer
- Dereferencing a pointer that contains an invalid memory location, such as 1005 may result in a ***Bus Error***
  - When your program results in a bus error, the program crashes immediately

1000		NUM	1012		iAry[2]
1001	3		1013	6	
1002			1014		
1003			1015		
1004		iAry[0]	1016		iPtr
1005	2		1017		
1006			1018	1004	
1007			1019		
1008		iAry[1]	1020		
1009	4		1021		
1010			1022		
1011			1023		



Pointer Arithmetic, Description

- Recall that a pointer type specifies the type of value it is pointing to
  - C++ can determine the size of the value being pointed to
  - When arithmetic is performed on a pointer, it is done using this knowledge to ensure the pointer doesn't point to intermediate memory locations
  - If an int requires 4 bytes, and iPtr is a variable of type "int *", then the statement "iPtr++;" actually increments the pointer value by 4
  - Similarly, "iPtr2 = iPtr + 5;" stores the address "five integers worth" past iPtr in iPtr2
    - If iPtr was 1000, then iPtr2 would contain the address 1020
    - 1020 = 1000 + 5 * 4
- Pointer arithmetic is performed automatically when arithmetic is done on pointers
  - No special syntax is required to get this behavior!



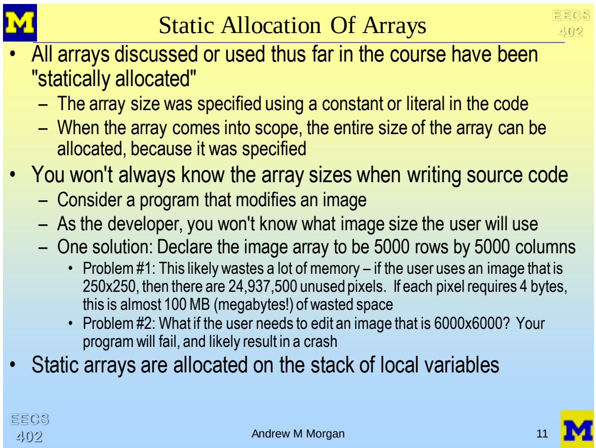
Using Pointer Arithmetic

```
const int NUM = 3;
int   iAry[NUM] = { 2, 4, 6 };
int  *iPtr;
int  *iPtr2;
int  *iPtr3;
int   i;

iPtr = iAry; //Assigns iPtr to point
             //to the first integer
             //in the iAry array

iPtr3 = iAry;
for (i = 0; i < NUM; i++)
{
  iPtr2 = iPtr + i;
  cout << i << " " <<
          iAry[i] << " " <<
          *iPtr2 << " " <<
          *iPtr3 << " " <<
          *(iPtr + i) << endl;
  iPtr3++;
}
```

1000		NUM	1012		iAry[2]
1001	3		1013	6	
1002			1014		
1003			1015		
1004		iAry[0]	1016		iPtr
1005	2		1017		
1006			1018	1004	
1007			1019		
1008		iAry[1]	1020		
1009	4		1021		
1010			1022		
1011			1023		

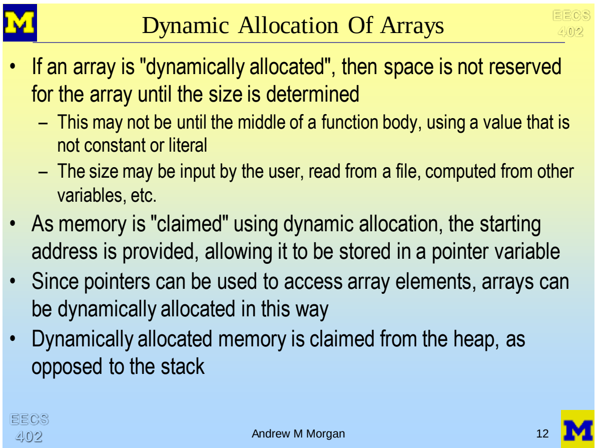
Other vars

```
0 2 2 2 2
1 4 4 4 4
2 6 6 6 6
```



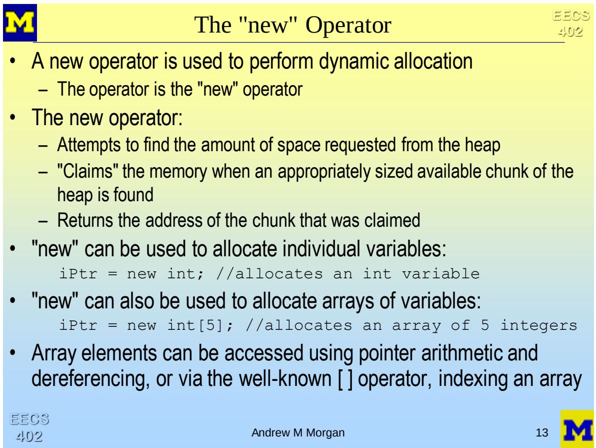
Static Allocation Of Arrays

- All arrays discussed or used thus far in the course have been "statically allocated"
  - The array size was specified using a constant or literal in the code
  - When the array comes into scope, the entire size of the array can be allocated, because it was specified
- You won't always know the array sizes when writing source code
  - Consider a program that modifies an image
  - As the developer, you won't know what image size the user will use
  - One solution: Declare the image array to be 5000 rows by 5000 columns
    - Problem #1: This likely wastes a lot of memory – if the user uses an image that is 250x250, then there are 24,937,500 unused pixels. If each pixel requires 4 bytes, this is almost 100 MB (megabytes!) of wasted space
    - Problem #2: What if the user needs to edit an image that is 6000x6000? Your program will fail, and likely result in a crash
- Static arrays are allocated on the stack of local variables



Dynamic Allocation Of Arrays

- If an array is "dynamically allocated", then space is not reserved for the array until the size is determined
  - This may not be until the middle of a function body, using a value that is not constant or literal
  - The size may be input by the user, read from a file, computed from other variables, etc.
- As memory is "claimed" using dynamic allocation, the starting address is provided, allowing it to be stored in a pointer variable
- Since pointers can be used to access array elements, arrays can be dynamically allocated in this way
- Dynamically allocated memory is claimed from the heap, as opposed to the stack



The "new" Operator

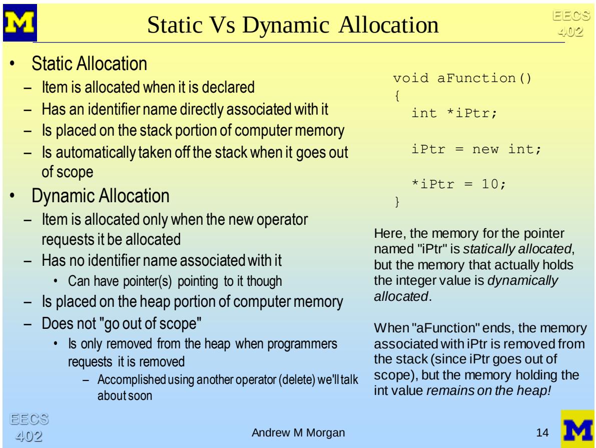
- A new operator is used to perform dynamic allocation
  - The operator is the "new" operator
- The new operator:
  - Attempts to find the amount of space requested from the heap
  - "Claims" the memory when an appropriately sized available chunk of the heap is found
  - Returns the address of the chunk that was claimed
- "new" can be used to allocate individual variables:

```
iPtr = new int; //allocates an int variable
```

- "new" can also be used to allocate arrays of variables:

```
iPtr = new int[5]; //allocates an array of 5 integers
```

- Array elements can be accessed using pointer arithmetic and dereferencing, or via the well-known [ ] operator, indexing an array



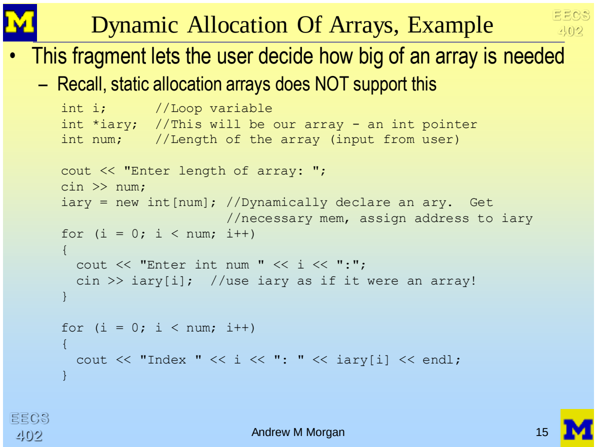
Static Vs Dynamic Allocation

- Static Allocation
  - Item is allocated when it is declared
  - Has an identifier name directly associated with it
  - Is placed on the stack portion of computer memory
  - Is automatically taken off the stack when it goes out of scope
- Dynamic Allocation
  - Item is allocated only when the new operator requests it be allocated
  - Has no identifier name associated with it
    - Can have pointer(s) pointing to it though
  - Is placed on the heap portion of computer memory
  - Does not "go out of scope"
    - Is only removed from the heap when programmers requests it is removed
      - Accomplished using another operator (delete) we'll talk about soon

```
void aFunction()
{
  int *iPtr;

  iPtr = new int;

  *iPtr = 10;
}
```

Here, the memory for the pointer named "iPtr" is *statically allocated*, but the memory that actually holds the integer value is *dynamically allocated*.

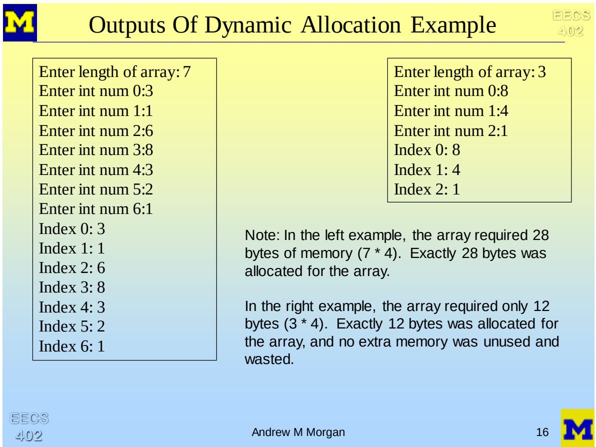
When "aFunction" ends, the memory associated with iPtr is removed from the stack (since iPtr goes out of scope), but the memory holding the int value *remains on the heap!*



Dynamic Allocation Of Arrays, Example

- This fragment lets the user decide how big of an array is needed
  - Recall, static allocation arrays does NOT support this

```
int i;      //Loop variable
int *iary;  //This will be our array - an int pointer
int num;    //Length of the array (input from user)

cout << "Enter length of array: ";
cin >> num;
iary = new int[num]; //Dynamically declare an ary.  Get
                     //necessary mem, assign address to iary
for (i = 0; i < num; i++)
{
  cout << "Enter int num " << i << ":";
  cin >> iary[i];  //use iary as if it were an array!
}

for (i = 0; i < num; i++)
{
  cout << "Index " << i << ": " << iary[i] << endl;
}
```



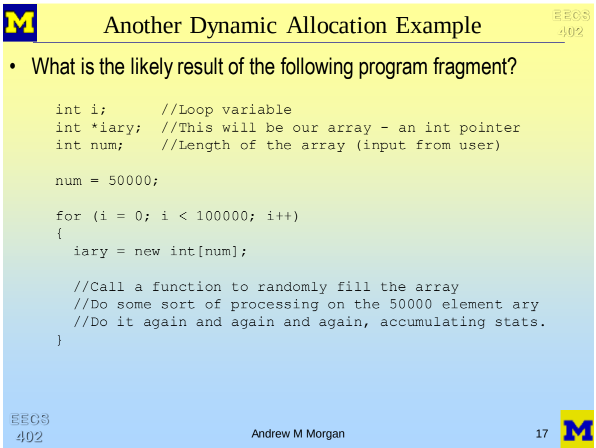
Outputs Of Dynamic Allocation Example

```
Enter length of array: 7
Enter int num 0:3
Enter int num 1:1
Enter int num 2:6
Enter int num 3:8
Enter int num 4:3
Enter int num 5:2
Enter int num 6:1
Index 0: 3
Index 1: 1
Index 2: 6
Index 3: 8
Index 4: 3
Index 5: 2
Index 6: 1
```

```
Enter length of array: 3
Enter int num 0:8
Enter int num 1:4
Enter int num 2:1
Index 0: 8
Index 1: 4
Index 2: 1
```

Note: In the left example, the array required 28 bytes of memory (7 * 4). Exactly 28 bytes was allocated for the array.

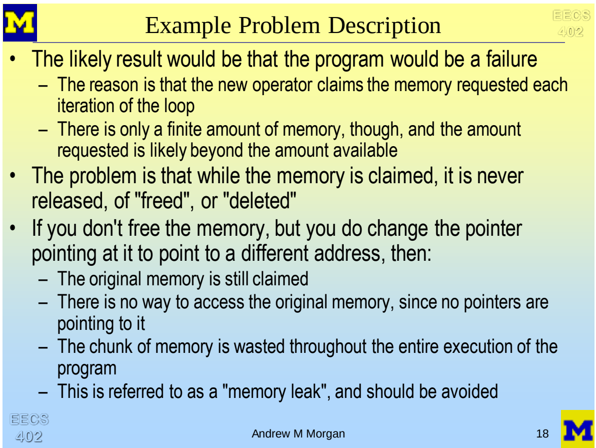
In the right example, the array required only 12 bytes (3 * 4). Exactly 12 bytes was allocated for the array, and no extra memory was unused and wasted.



Another Dynamic Allocation Example

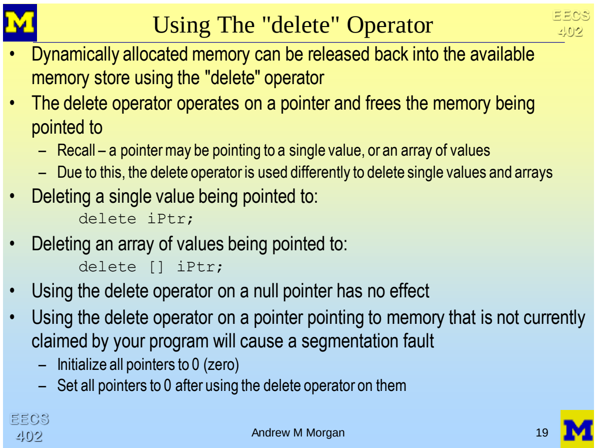
- What is the likely result of the following program fragment?

```
int i;      //Loop variable
int *iary;  //This will be our array - an int pointer
int num;    //Length of the array (input from user)

num = 50000;

for (i = 0; i < 100000; i++)
{
  iary = new int[num];

  //Call a function to randomly fill the array
  //Do some sort of processing on the 50000 element ary
  //Do it again and again and again, accumulating stats.
}
```



Example Problem Description

- The likely result would be that the program would be a failure
  - The reason is that the new operator claims the memory requested each iteration of the loop
  - There is only a finite amount of memory, though, and the amount requested is likely beyond the amount available
- The problem is that while the memory is claimed, it is never released, of "freed", or "deleted"
- If you don't free the memory, but you do change the pointer pointing at it to point to a different address, then:
  - The original memory is still claimed
  - There is no way to access the original memory, since no pointers are pointing to it
  - The chunk of memory is wasted throughout the entire execution of the program
  - This is referred to as a "memory leak", and should be avoided



Using The "delete" Operator

- Dynamically allocated memory can be released back into the available memory store using the "delete" operator
- The delete operator operates on a pointer and frees the memory being pointed to
  - Recall – a pointer may be pointing to a single value, or an array of values
  - Due to this, the delete operator is used differently to delete single values and arrays
- Deleting a single value being pointed to:

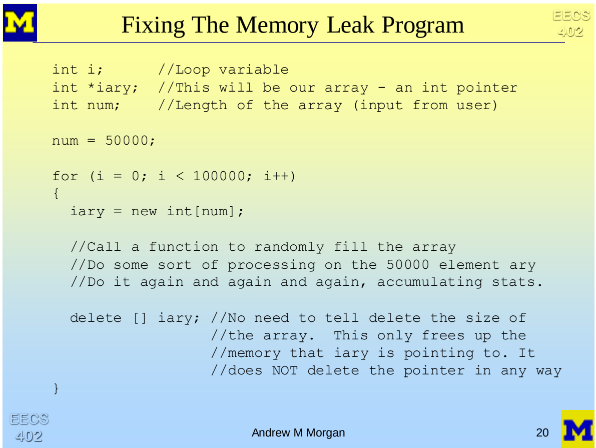
```
    delete iPtr;
```

- Deleting an array of values being pointed to:

```
    delete [] iPtr;
```

- Using the delete operator on a null pointer has no effect
- Using the delete operator on a pointer pointing to memory that is not currently claimed by your program will cause a segmentation fault
  - Initialize all pointers to 0 (zero)
  - Set all pointers to 0 after using the delete operator on them

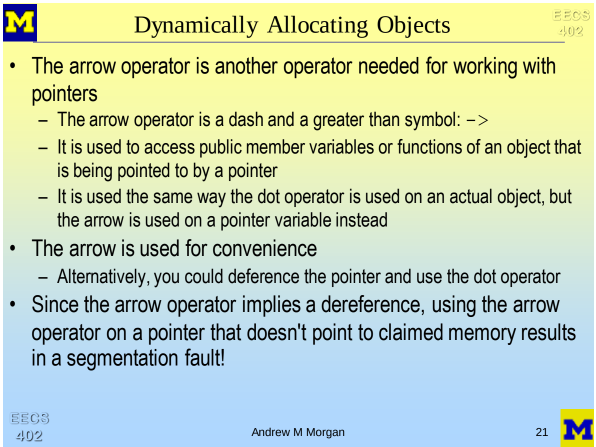
19



Fixing The Memory Leak Program

```
int i;      //Loop variable
int *iary;  //This will be our array - an int pointer
int num;    //Length of the array (input from user)

num = 50000;

for (i = 0; i < 100000; i++)
{
  iary = new int[num];

  //Call a function to randomly fill the array
  //Do some sort of processing on the 50000 element ary
  //Do it again and again and again, accumulating stats.

  delete [] iary; //No need to tell delete the size of
                  //the array.  This only frees up the
                  //memory that iary is pointing to. It
                  //does NOT delete the pointer in any way
}
```

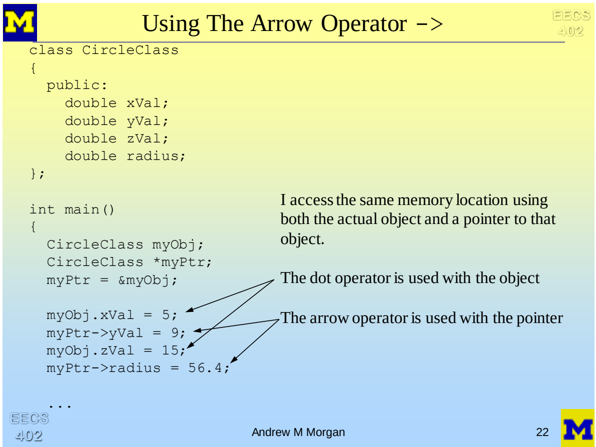
20



Dynamically Allocating Objects

- The arrow operator is another operator needed for working with pointers
  - The arrow operator is a dash and a greater than symbol: ->
  - It is used to access public member variables or functions of an object that is being pointed to by a pointer
  - It is used the same way the dot operator is used on an actual object, but the arrow is used on a pointer variable instead
- The arrow is used for convenience
  - Alternatively, you could deference the pointer and use the dot operator
- Since the arrow operator implies a dereference, using the arrow operator on a pointer that doesn't point to claimed memory results in a segmentation fault!

21



Using The Arrow Operator ->

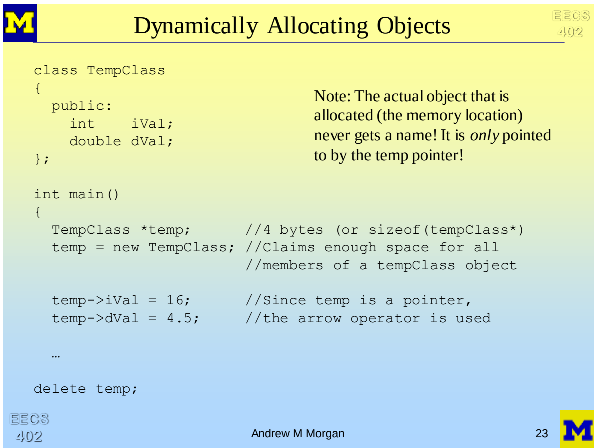
```
class CircleClass
{
  public:
    double xVal;
    double yVal;
    double zVal;
    double radius;
};

int main()
{
  CircleClass myObj;
  CircleClass *myPtr;
  myPtr = &myObj;

  myObj.xVal = 5;
  myPtr->yVal = 9;
  myObj.zVal = 15;
  myPtr->radius = 56.4;

  ...
```

I access the same memory location using both the actual object and a pointer to that object.

The dot operator is used with the object

The arrow operator is used with the pointer

22

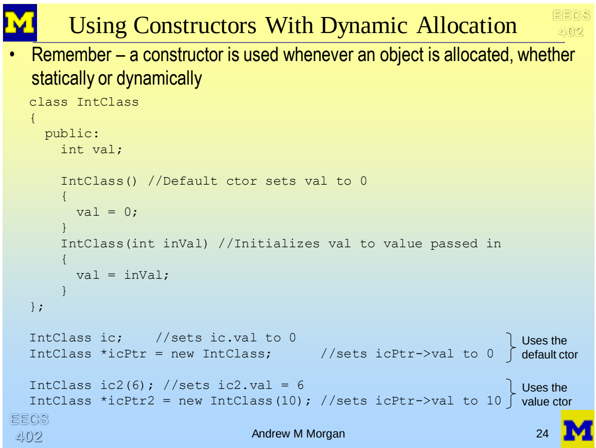


Dynamically Allocating Objects

```
class TempClass
{
  public:
    int    iVal;
    double dVal;
};

int main()
{
  TempClass *temp;      //4 bytes (or sizeof(tempClass*)
  temp = new TempClass; //Claims enough space for all
                        //members of a tempClass object

  temp->iVal = 16;      //Since temp is a pointer,
  temp->dVal = 4.5;     //the arrow operator is used

  …

delete temp;
```

Note: The actual object that is allocated (the memory location) never gets a name! It is *only* pointed to by the temp pointer!

23



Using Constructors With Dynamic Allocation

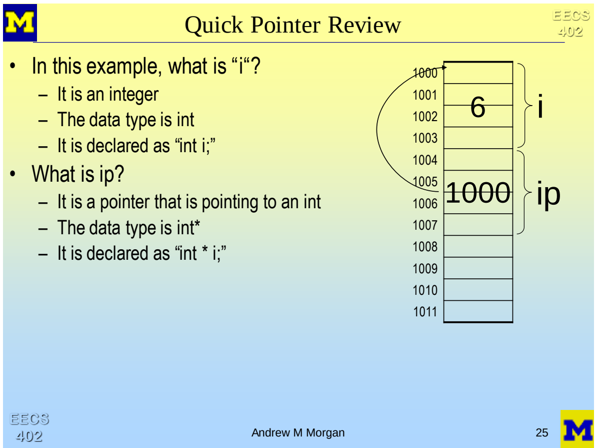
- Remember – a constructor is used whenever an object is allocated, whether statically or dynamically

```
class IntClass
{
  public:
    int val;

    IntClass() //Default ctor sets val to 0
    {
      val = 0;
    }
    IntClass(int inVal) //Initializes val to value passed in
    {
      val = inVal;
    }
};

IntClass ic;    //sets ic.val to 0
IntClass *icPtr = new IntClass;      //sets icPtr->val to 0

IntClass ic2(6); //sets ic2.val = 6
IntClass *icPtr2 = new IntClass(10); //sets icPtr->val to 10
```

Uses the default ctor

Uses the value ctor

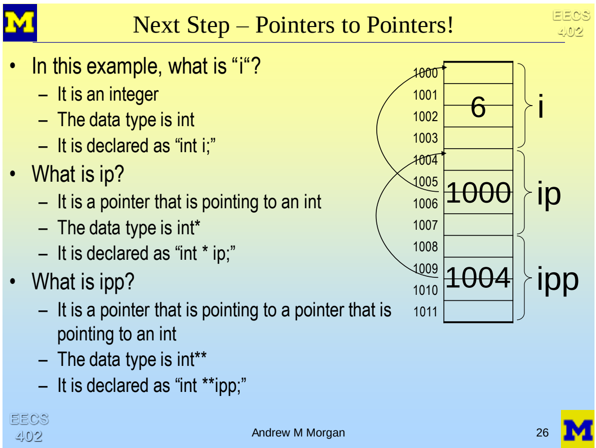
24



Quick Pointer Review

- In this example, what is “i“?
  - It is an integer
  - The data type is int
  - It is declared as “int i;”
- What is ip?
  - It is a pointer that is pointing to an int
  - The data type is int*
  - It is declared as “int * i;”

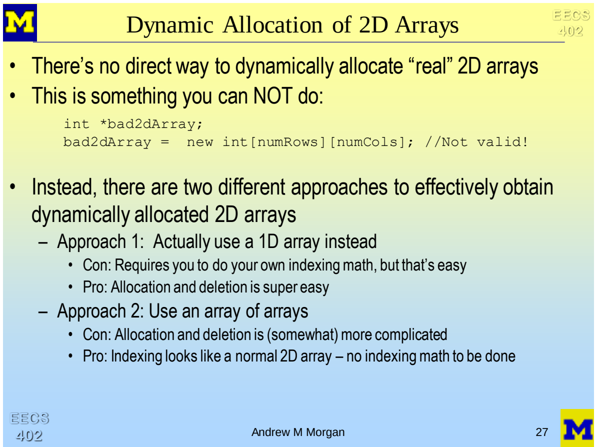
25



Next Step – Pointers to Pointers!

- In this example, what is “i“?
  - It is an integer
  - The data type is int
  - It is declared as “int i;”
- What is ip?
  - It is a pointer that is pointing to an int
  - The data type is int*
  - It is declared as “int * ip;”
- What is ipp?
  - It is a pointer that is pointing to a pointer that is pointing to an int
  - The data type is int**
  - It is declared as “int **ipp;”

26



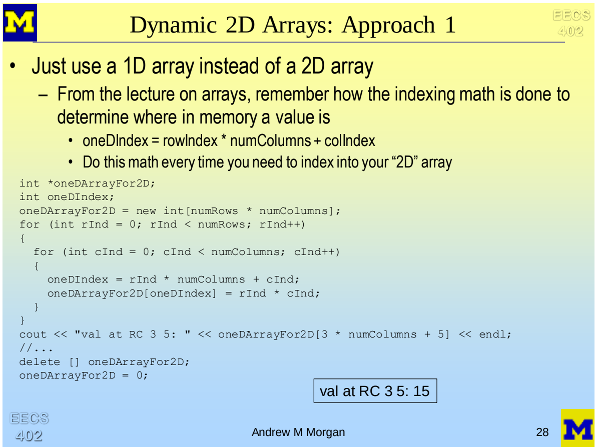
Dynamic Allocation of 2D Arrays

- There’s no direct way to dynamically allocate “real” 2D arrays
- This is something you can NOT do:

```
int *bad2dArray;
bad2dArray =  new int[numRows][numCols]; //Not valid!
```

- Instead, there are two different approaches to effectively obtain dynamically allocated 2D arrays
  - Approach 1: Actually use a 1D array instead
    - Con: Requires you to do your own indexing math, but that’s easy
    - Pro: Allocation and deletion is super easy
  - Approach 2: Use an array of arrays
    - Con: Allocation and deletion is (somewhat) more complicated
    - Pro: Indexing looks like a normal 2D array – no indexing math to be done

27



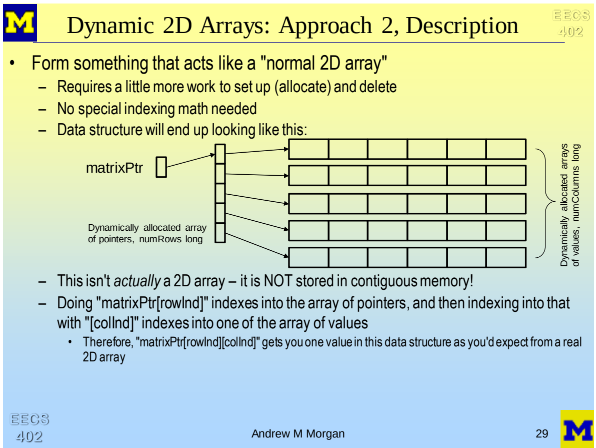
Dynamic 2D Arrays: Approach 1

- Just use a 1D array instead of a 2D array
  - From the lecture on arrays, remember how the indexing math is done to determine where in memory a value is
    - oneDIndex = rowIndex * numColumns + colIndex
    - Do this math every time you need to index into your “2D” array

```
int *oneDArrayFor2D;
int oneDIndex;
oneDArrayFor2D = new int[numRows * numColumns];
for (int rInd = 0; rInd < numRows; rInd++)
{
  for (int cInd = 0; cInd < numColumns; cInd++)
  {
    oneDIndex = rInd * numColumns + cInd;
    oneDArrayFor2D[oneDIndex] = rInd * cInd;
  }
}
cout << "val at RC 3 5: " << oneDArrayFor2D[3 * numColumns + 5] << endl;
//...
delete [] oneDArrayFor2D;
oneDArrayFor2D = 0;
```

```
val at RC 3 5: 15
```

28



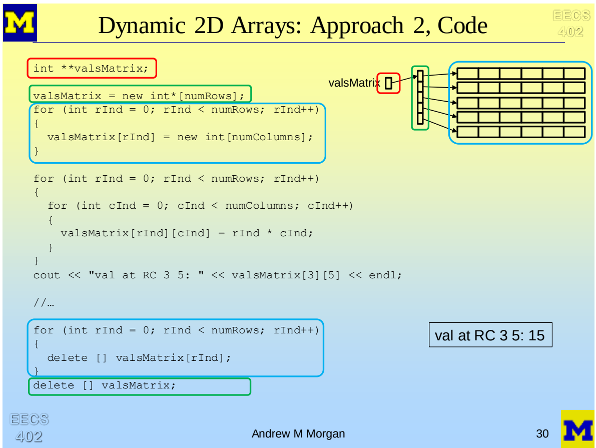
Dynamic 2D Arrays: Approach 2, Description

- Form something that acts like a "normal 2D array"
  - Requires a little more work to set up (allocate) and delete
  - No special indexing math needed
  - Data structure will end up looking like this:

matrixPtr
Dynamically allocated array of pointers, numRows long
Dynamically allocated arrays of values, numColumns long

  - This isn't *actually* a 2D array – it is NOT stored in contiguous memory!
  - Doing "matrixPtr[rowInd]" indexes into the array of pointers, and then indexing into that with "[colInd]" indexes into one of the array of values
    - Therefore, "matrixPtr[rowInd][colInd]" gets you one value in this data structure as you'd expect from a real 2D array

29

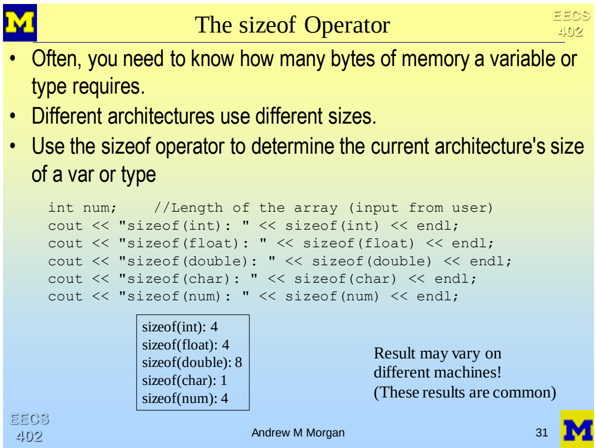


Dynamic 2D Arrays: Approach 2, Code

```
int **valsMatrix;

valsMatrix = new int*[numRows];
for (int rInd = 0; rInd < numRows; rInd++)
{
  valsMatrix[rInd] = new int[numColumns];
}

for (int rInd = 0; rInd < numRows; rInd++)
{
  for (int cInd = 0; cInd < numColumns; cInd++)
  {
    valsMatrix[rInd][cInd] = rInd * cInd;
  }
}
cout << "val at RC 3 5: " << valsMatrix[3][5] << endl;

//…

for (int rInd = 0; rInd < numRows; rInd++)
{
  delete [] valsMatrix[rInd];
}
delete [] valsMatrix;
```

```
val at RC 3 5: 15
```

30



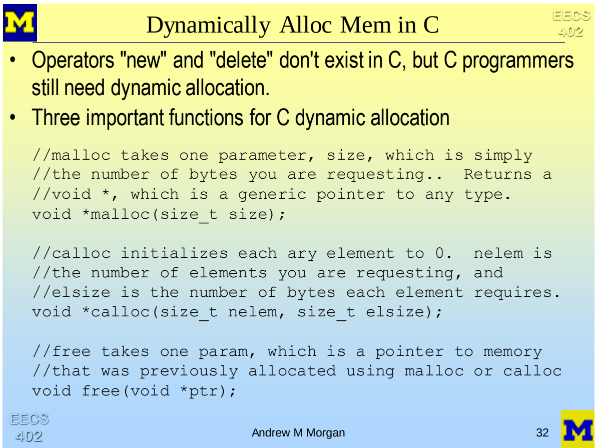
The sizeof Operator

- Often, you need to know how many bytes of memory a variable or type requires.
- Different architectures use different sizes.
- Use the sizeof operator to determine the current architecture's size of a var or type

```
int num;    //Length of the array (input from user)
cout << "sizeof(int): " << sizeof(int) << endl;
cout << "sizeof(float): " << sizeof(float) << endl;
cout << "sizeof(double): " << sizeof(double) << endl;
cout << "sizeof(char): " << sizeof(char) << endl;
cout << "sizeof(num): " << sizeof(num) << endl;
```

```
sizeof(int): 4
sizeof(float): 4
sizeof(double): 8
sizeof(char): 1
sizeof(num): 4
```

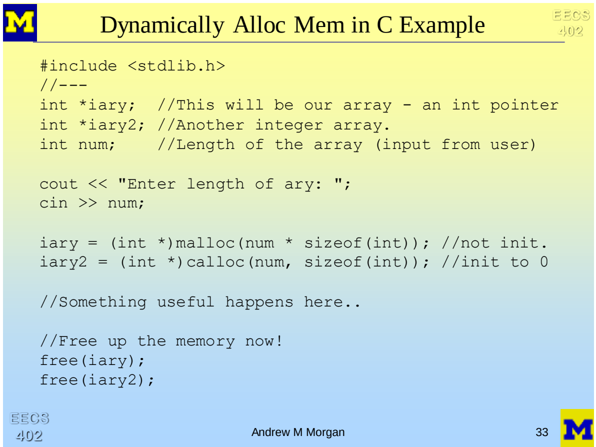
Result may vary on
different machines!
(These results are common)



Dynamically Alloc Mem in C

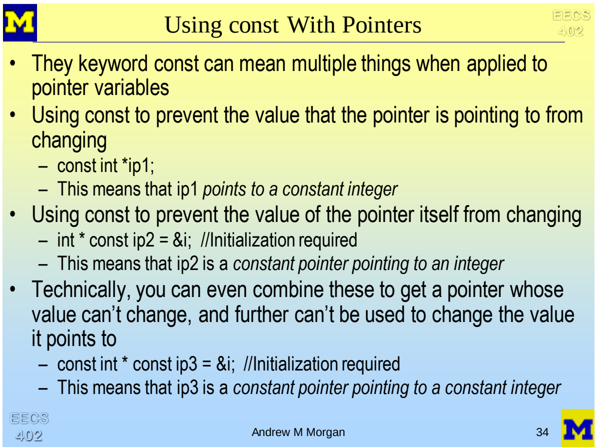
- Operators "new" and "delete" don't exist in C, but C programmers still need dynamic allocation.
- Three important functions for C dynamic allocation

```
//malloc takes one parameter, size, which is simply
//the number of bytes you are requesting.. Returns a
//void *, which is a generic pointer to any type.
void *malloc(size_t size);

//calloc initializes each ary element to 0.  nelem is
//the number of elements you are requesting, and
//elsize is the number of bytes each element requires.
void *calloc(size_t nelem, size_t elsize);

//free takes one param, which is a pointer to memory
//that was previously allocated using malloc or calloc
void free(void *ptr);
```



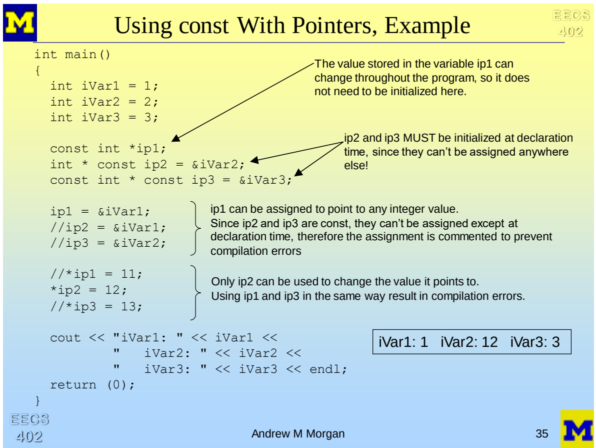
Dynamically Alloc Mem in C Example

```
#include <stdlib.h>
//---
int *iary;  //This will be our array - an int pointer
int *iary2; //Another integer array.
int num;    //Length of the array (input from user)

cout << "Enter length of ary: ";
cin >> num;

iary = (int *)malloc(num * sizeof(int)); //not init.
iary2 = (int *)calloc(num, sizeof(int)); //init to 0

//Something useful happens here..

//Free up the memory now!
free(iary);
free(iary2);
```



Using const With Pointers

- They keyword const can mean multiple things when applied to pointer variables
- Using const to prevent the value that the pointer is pointing to from changing
  - const int *ip1;
  - This means that ip1 *points to a constant integer*
- Using const to prevent the value of the pointer itself from changing
  - int * const ip2 = &i; //Initialization required
  - This means that ip2 is a *constant pointer pointing to an integer*
- Technically, you can even combine these to get a pointer whose value can't change, and further can't be used to change the value it points to
  - const int * const ip3 = &i; //Initialization required
  - This means that ip3 is a *constant pointer pointing to a constant integer*



Using const With Pointers, Example

```
int main()
{
  int iVar1 = 1;
  int iVar2 = 2;
  int iVar3 = 3;

  const int *ip1;
  int * const ip2 = &iVar2;
  const int * const ip3 = &iVar3;

  ip1 = &iVar1;
  //ip2 = &iVar1;
  //ip3 = &iVar2;

  //*ip1 = 11;
  *ip2 = 12;
  //*ip3 = 13;

  cout << "iVar1: " << iVar1 <<
          "   iVar2: " << iVar2 <<
          "   iVar3: " << iVar3 << endl;
  return (0);
}
```

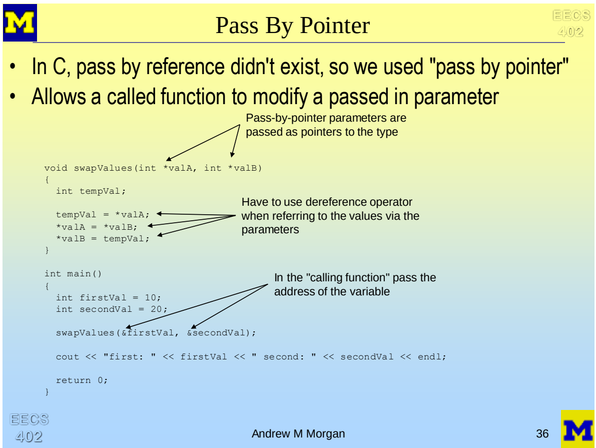
The value stored in the variable ip1 can change throughout the program, so it does not need to be initialized here.

ip2 and ip3 MUST be initialized at declaration time, since they can't be assigned anywhere else!

ip1 can be assigned to point to any integer value.
Since ip2 and ip3 are const, they can't be assigned except at declaration time, therefore the assignment is commented to prevent compilation errors

Only ip2 can be used to change the value it points to.
Using ip1 and ip3 in the same way result in compilation errors.

```
iVar1: 1   iVar2: 12   iVar3: 3
```



Pass By Pointer

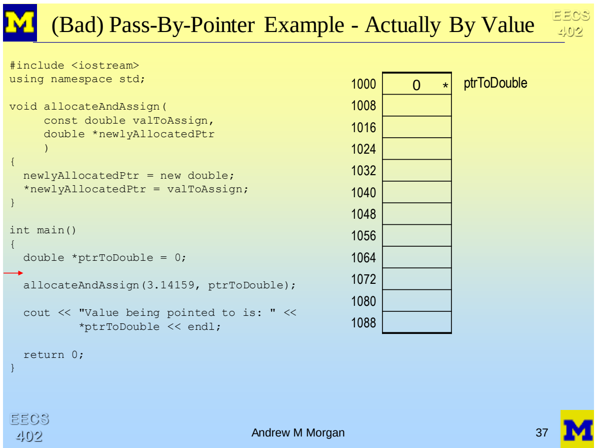
- In C, pass by reference didn't exist, so we used "pass by pointer"
- Allows a called function to modify a passed in parameter

Pass-by-pointer parameters are passed as pointers to the type

```
void swapValues(int *valA, int *valB)
{
  int tempVal;

  tempVal = *valA;
  *valA = *valB;
  *valB = tempVal;
}

int main()
{
  int firstVal = 10;
  int secondVal = 20;

  swapValues(&firstVal, &secondVal);

  cout << "first: " << firstVal << " second: " << secondVal << endl;

  return 0;
}
```

Have to use dereference operator when referring to the values via the parameters

In the "calling function" pass the address of the variable

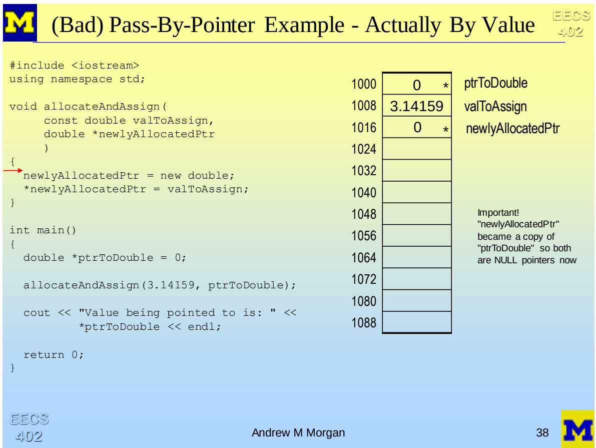


(Bad) Pass-By-Pointer Example - Actually By Value

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

void allocateAndAssign(
     const double valToAssign,
     double *newlyAllocatedPtr
     )
{
  newlyAllocatedPtr = new double;
  *newlyAllocatedPtr = valToAssign;
}

int main()
{
  double *ptrToDouble = 0;

  allocateAndAssign(3.14159, ptrToDouble);

  cout << "Value being pointed to is: " <<
          *ptrToDouble << endl;

  return 0;
}
```

1000	0 *	ptrToDouble
1008		
1016		
1024		
1032		
1040		
1048		
1056		
1064		
1072		
1080		
1088		

37



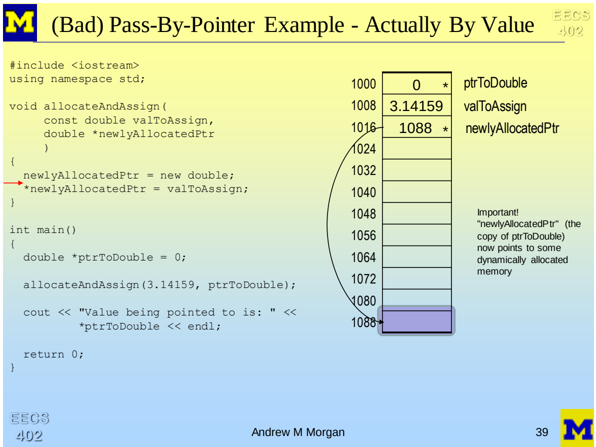
(Bad) Pass-By-Pointer Example - Actually By Value

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

void allocateAndAssign(
     const double valToAssign,
     double *newlyAllocatedPtr
     )
{
  newlyAllocatedPtr = new double;
  *newlyAllocatedPtr = valToAssign;
}

int main()
{
  double *ptrToDouble = 0;

  allocateAndAssign(3.14159, ptrToDouble);

  cout << "Value being pointed to is: " <<
          *ptrToDouble << endl;

  return 0;
}
```

1000	0 *	ptrToDouble
1008	3.14159	valToAssign
1016	0 *	newlyAllocatedPtr
1024		
1032		
1040		
1048		
1056		
1064		
1072		
1080		
1088		

Important! "newlyAllocatedPtr" became a copy of "ptrToDouble" so both are NULL pointers now

38



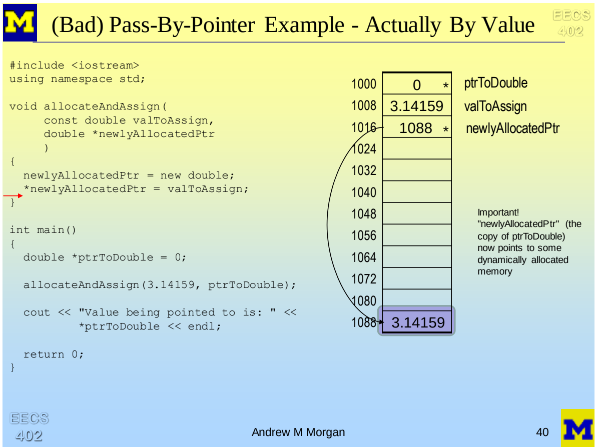
(Bad) Pass-By-Pointer Example - Actually By Value

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

void allocateAndAssign(
     const double valToAssign,
     double *newlyAllocatedPtr
     )
{
  newlyAllocatedPtr = new double;
  *newlyAllocatedPtr = valToAssign;
}

int main()
{
  double *ptrToDouble = 0;

  allocateAndAssign(3.14159, ptrToDouble);

  cout << "Value being pointed to is: " <<
          *ptrToDouble << endl;

  return 0;
}
```

1000	0 *	ptrToDouble
1008	3.14159	valToAssign
1016	1088 *	newlyAllocatedPtr
1024		
1032		
1040		
1048		
1056		
1064		
1072		
1080		
1088		

Important! "newlyAllocatedPtr" (the copy of ptrToDouble) now points to some dynamically allocated memory

39



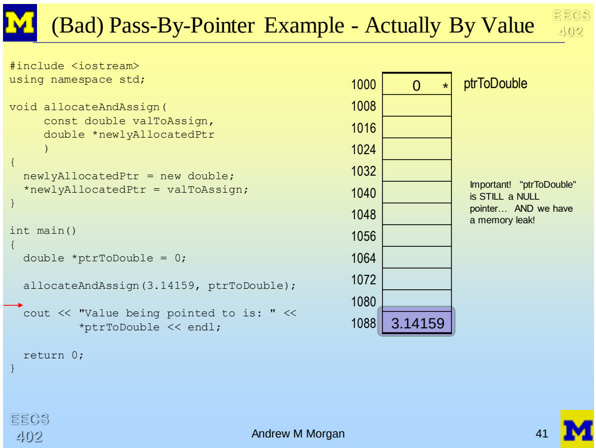
(Bad) Pass-By-Pointer Example - Actually By Value

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

void allocateAndAssign(
     const double valToAssign,
     double *newlyAllocatedPtr
     )
{
  newlyAllocatedPtr = new double;
  *newlyAllocatedPtr = valToAssign;
}

int main()
{
  double *ptrToDouble = 0;

  allocateAndAssign(3.14159, ptrToDouble);

  cout << "Value being pointed to is: " <<
          *ptrToDouble << endl;

  return 0;
}
```

1000	0 *	ptrToDouble
1008	3.14159	valToAssign
1016	1088 *	newlyAllocatedPtr
1024		
1032		
1040		
1048		
1056		
1064		
1072		
1080		
1088	3.14159	

Important! "newlyAllocatedPtr" (the copy of ptrToDouble) now points to some dynamically allocated memory

40



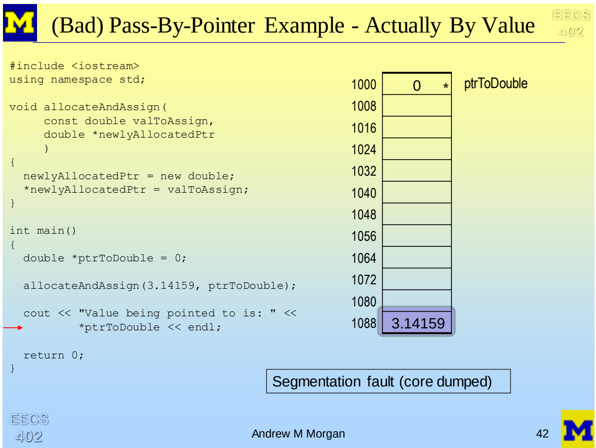
(Bad) Pass-By-Pointer Example - Actually By Value

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

void allocateAndAssign(
     const double valToAssign,
     double *newlyAllocatedPtr
     )
{
  newlyAllocatedPtr = new double;
  *newlyAllocatedPtr = valToAssign;
}

int main()
{
  double *ptrToDouble = 0;

  allocateAndAssign(3.14159, ptrToDouble);

  cout << "Value being pointed to is: " <<
          *ptrToDouble << endl;

  return 0;
}
```

1000	0 *	ptrToDouble
1008		
1016		
1024		
1032		
1040		
1048		
1056		
1064		
1072		
1080		
1088	3.14159	

Important! "ptrToDouble" is STILL a NULL pointer… AND we have a memory leak!

41



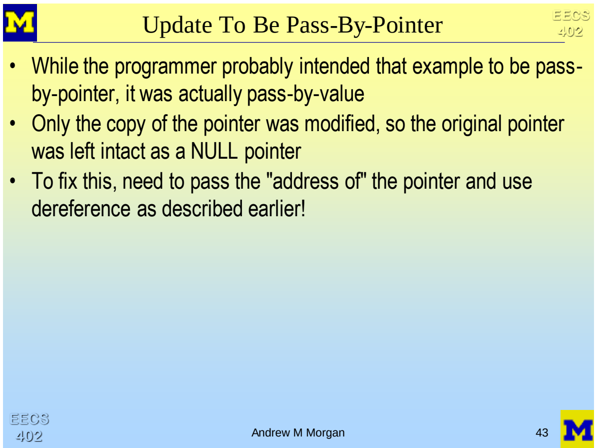
(Bad) Pass-By-Pointer Example - Actually By Value

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

void allocateAndAssign(
     const double valToAssign,
     double *newlyAllocatedPtr
     )
{
  newlyAllocatedPtr = new double;
  *newlyAllocatedPtr = valToAssign;
}

int main()
{
  double *ptrToDouble = 0;

  allocateAndAssign(3.14159, ptrToDouble);

  cout << "Value being pointed to is: " <<
          *ptrToDouble << endl;

  return 0;
}
```

1000	0 *	ptrToDouble
1008		
1016		
1024		
1032		
1040		
1048		
1056		
1064		
1072		
1080		
1088	3.14159	

```
Segmentation fault (core dumped)
```

42

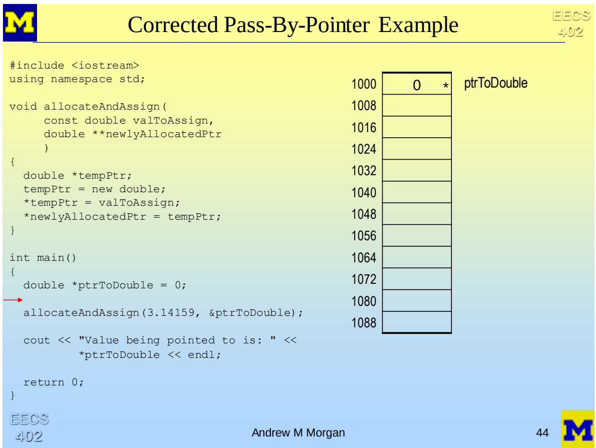


Update To Be Pass-By-Pointer

- While the programmer probably intended that example to be pass-by-pointer, it was actually pass-by-value
- Only the copy of the pointer was modified, so the original pointer was left intact as a NULL pointer
- To fix this, need to pass the "address of" the pointer and use dereference as described earlier!

EECS 402 Andrew M Morgan 43

43



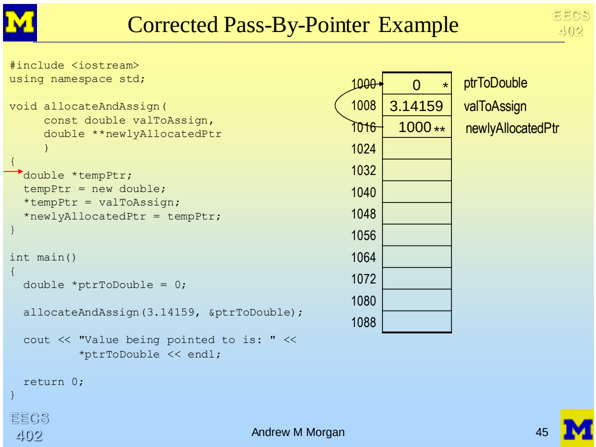
Corrected Pass-By-Pointer Example

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

void allocateAndAssign(
     const double valToAssign,
     double **newlyAllocatedPtr
     )
{
  double *tempPtr;
  tempPtr = new double;
  *tempPtr = valToAssign;
  *newlyAllocatedPtr = tempPtr;
}

int main()
{
  double *ptrToDouble = 0;

  allocateAndAssign(3.14159, &ptrToDouble);

  cout << "Value being pointed to is: " <<
          *ptrToDouble << endl;

  return 0;
}
```

1000	0 *	ptrToDouble
1008		
1016		
1024		
1032		
1040		
1048		
1056		
1064		
1072		
1080		
1088		

EECS 402 Andrew M Morgan 44

44



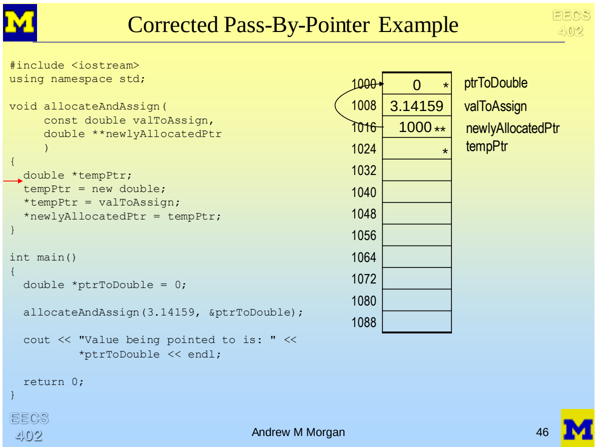
Corrected Pass-By-Pointer Example

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

void allocateAndAssign(
     const double valToAssign,
     double **newlyAllocatedPtr
     )
{
  double *tempPtr;
  tempPtr = new double;
  *tempPtr = valToAssign;
  *newlyAllocatedPtr = tempPtr;
}

int main()
{
  double *ptrToDouble = 0;

  allocateAndAssign(3.14159, &ptrToDouble);

  cout << "Value being pointed to is: " <<
          *ptrToDouble << endl;

  return 0;
}
```

1000	0 *	ptrToDouble
1008	3.14159	valToAssign
1016	1000 **	newlyAllocatedPtr
1024		
1032		
1040		
1048		
1056		
1064		
1072		
1080		
1088		

EECS 402 Andrew M Morgan 45

45



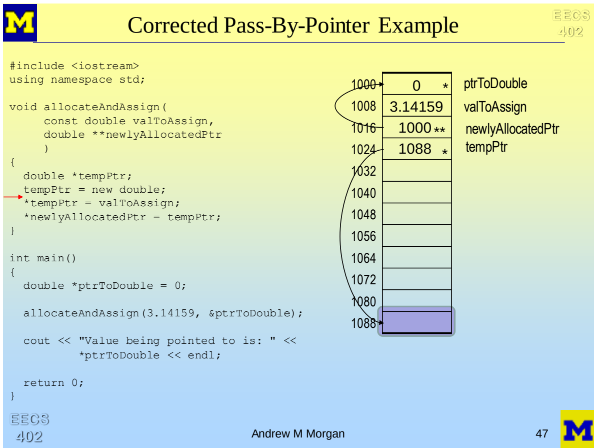
Corrected Pass-By-Pointer Example

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

void allocateAndAssign(
     const double valToAssign,
     double **newlyAllocatedPtr
     )
{
  double *tempPtr;
  tempPtr = new double;
  *tempPtr = valToAssign;
  *newlyAllocatedPtr = tempPtr;
}

int main()
{
  double *ptrToDouble = 0;

  allocateAndAssign(3.14159, &ptrToDouble);

  cout << "Value being pointed to is: " <<
          *ptrToDouble << endl;

  return 0;
}
```

1000	0 *	ptrToDouble
1008	3.14159	valToAssign
1016	1000 **	newlyAllocatedPtr
1024	*	tempPtr
1032		
1040		
1048		
1056		
1064		
1072		
1080		
1088		

EECS 402 Andrew M Morgan 46

46



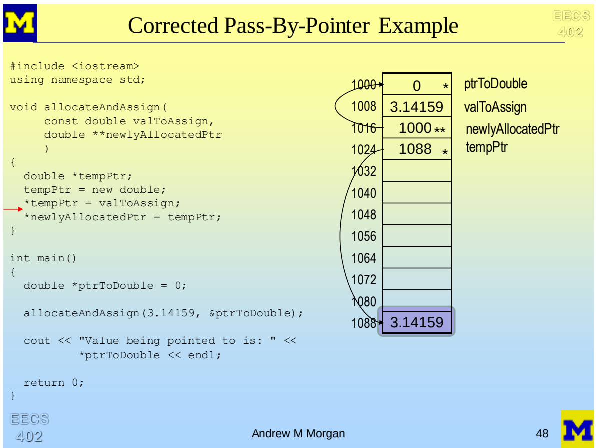
Corrected Pass-By-Pointer Example

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

void allocateAndAssign(
     const double valToAssign,
     double **newlyAllocatedPtr
     )
{
  double *tempPtr;
  tempPtr = new double;
  *tempPtr = valToAssign;
  *newlyAllocatedPtr = tempPtr;
}

int main()
{
  double *ptrToDouble = 0;

  allocateAndAssign(3.14159, &ptrToDouble);

  cout << "Value being pointed to is: " <<
          *ptrToDouble << endl;

  return 0;
}
```

1000	0 *	ptrToDouble
1008	3.14159	valToAssign
1016	1000 **	newlyAllocatedPtr
1024	1088 *	tempPtr
1032		
1040		
1048		
1056		
1064		
1072		
1080		
1088		

EECS 402 Andrew M Morgan 47

47



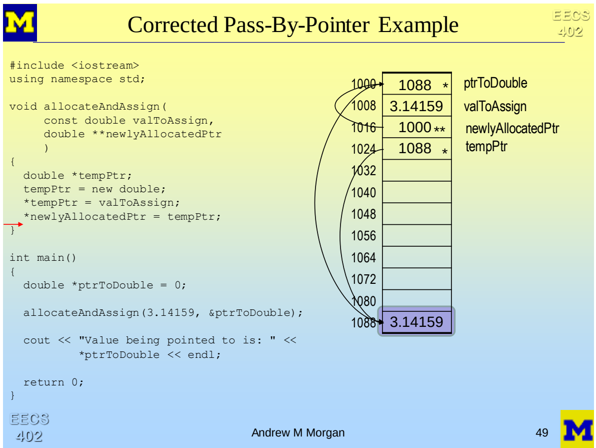
Corrected Pass-By-Pointer Example

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

void allocateAndAssign(
     const double valToAssign,
     double **newlyAllocatedPtr
     )
{
  double *tempPtr;
  tempPtr = new double;
  *tempPtr = valToAssign;
  *newlyAllocatedPtr = tempPtr;
}

int main()
{
  double *ptrToDouble = 0;

  allocateAndAssign(3.14159, &ptrToDouble);

  cout << "Value being pointed to is: " <<
          *ptrToDouble << endl;

  return 0;
}
```

1000	0 *	ptrToDouble
1008	3.14159	valToAssign
1016	1000 **	newlyAllocatedPtr
1024	1088 *	tempPtr
1032		
1040		
1048		
1056		
1064		
1072		
1080		
1088	3.14159	

EECS 402 Andrew M Morgan 48

48



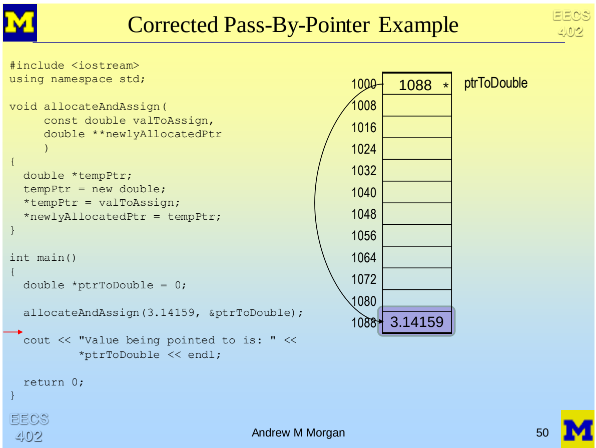
Corrected Pass-By-Pointer Example

EECS 402

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

void allocateAndAssign(
     const double valToAssign,
     double **newlyAllocatedPtr
     )
{
  double *tempPtr;
  tempPtr = new double;
  *tempPtr = valToAssign;
  *newlyAllocatedPtr = tempPtr;
}

int main()
{
  double *ptrToDouble = 0;

  allocateAndAssign(3.14159, &ptrToDouble);

  cout << "Value being pointed to is: " <<
          *ptrToDouble << endl;

  return 0;
}
```

EECS 402 Andrew M Morgan 49

49



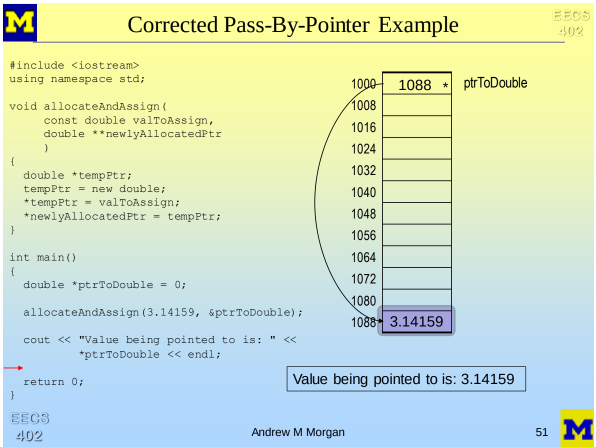
Corrected Pass-By-Pointer Example

EECS 402

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

void allocateAndAssign(
     const double valToAssign,
     double **newlyAllocatedPtr
     )
{
  double *tempPtr;
  tempPtr = new double;
  *tempPtr = valToAssign;
  *newlyAllocatedPtr = tempPtr;
}

int main()
{
  double *ptrToDouble = 0;

  allocateAndAssign(3.14159, &ptrToDouble);

  cout << "Value being pointed to is: " <<
          *ptrToDouble << endl;

  return 0;
}
```

EECS 402 Andrew M Morgan 50

50



Corrected Pass-By-Pointer Example

EECS 402

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

void allocateAndAssign(
     const double valToAssign,
     double **newlyAllocatedPtr
     )
{
  double *tempPtr;
  tempPtr = new double;
  *tempPtr = valToAssign;
  *newlyAllocatedPtr = tempPtr;
}

int main()
{
  double *ptrToDouble = 0;

  allocateAndAssign(3.14159, &ptrToDouble);

  cout << "Value being pointed to is: " <<
          *ptrToDouble << endl;

  return 0;
}
```

```
Value being pointed to is: 3.14159
```

EECS 402 Andrew M Morgan 51

51