CMSC 202

Pointers
Dynamic Memory Allocation
A simple variable

- A variable is drawn as a labeled box

```c
int x;
```

x:  


Complex Variables

• A complex variable is drawn as complex boxes (horizontal for array, vertical for struct)

```c
struct Astruct {
    int y;
    char c;
};

int intArray[4];
Astruct myStruct;
```
Pointer *value*

- A pointer *value* is drawn as an arrow. The arrow points to a picture of the object to which the pointer points. The pointer is said to *reference* the object.
Pointer Variable

- A pointer variable is drawn as a labeled box, like any other simple variable

```
int *iPtr;    iPtr:
```

- NULL is a pointer value that doesn’t point to anything and is drawn as a box with a slash
When a pointer variable points to an object ...

... the base of the arrow representing the pointer value is placed in the pointer variable’s box

\[ \text{iPtr : } \begin{array}{c}
\end{array} \rightarrow 42 \]
Assignment to pointer variables

• A pointer variable receives a value through assignment.
  – Newly allocated space (using `new`)
  – Previously allocated space
Allocating space with “new”

• Graphically depict the result of allocating memory with new by
  1. Figure out what kind of space is being allocated
  2. Draw a box to represent the allocated space
  3. Draw an arrow pointing to the new box and place its base in the box representing the pointer variable being assigned to
class Car
{
    public:
        . . . .
    private:
        string m_color;
        int m_mileage;
};

Car *CarPtr;
CarPtr = new Car;
What’s in the “new” data?

• Nothing
  – Conceptually, after you use `new`, you get some blank space to fill with data

• Something
  – Your new space is not initialized to some zero value. Try allocating some new memory with `new` and immediately printing its contents.
  – You will see all sorts of random values. This is related to a security concern we will discuss later.
Pointer Assignment

Suppose ‘thing1’ and ‘thing2’ are “pointers to int” and ‘thing1’ points to an int whose value is 17.

thing1: [ ] \rightarrow 17

thing2: [ ]
And now execute the statement

\[
\text{thing2} = \text{thing1};
\]

1. Draw a new arrow that points to the box being referenced by the pointer value on the right hand side

2. Place the the base of the new arrow in the box representing the pointer variable on the left hand side

\[
\begin{align*}
\text{thing1:} & \quad 17 \\
\text{thing2:} &
\end{align*}
\]
Dereferencing pointers

Given the following picture

thing1: 17

The * (star) operator “dereferences” a pointer variable. In terms of our picture it means “follow the arrow”.

We can use dereferencing to change what a pointer value points to:

*thing1 = 42; results in this picture

thing1: 42
An exercise for the student

Given the following declarations and code

typedef int* intPtrArray [ 3 ];
typedef intPtrArray* arrayPtr;

arrayPtr cat;
int *dog;
dog = new int;
*dog = 42;
cat = new intPtrArray;
cat[0] = dog;
cat[1] = NULL;
cat[2] = new int;
*(cat[2]) = 17;
The Questions

1. What type of variable is dog?
2. What type of variable is cat? (NO, the answer is NOT arrayPtr).
3. Draw the picture that results from the code
The & operator

- The & (address of) operator is used to create an arrow that points to an existing box. Consider the following code

```c
int value;
int *pValue;

value = 17;
pValue = &value;
```

The expression &value gives us a pointer to value
Pointers and Arrays

• A strong relationship between pointers and arrays.
  – The name of an array is defined to be a pointer to the first (0th) element of the array.

Consider the declaration  

```c
int bongo [ 4 ];
```

The name `bongo` can be used as a pointer to reference `bongo[0]`.

```c
*bongo = 42;
```

is equivalent to

```c
bongo[0] = 42;
```
Pointers to array elements

If a pointer value references a particular array element, then adding an integer, \( n \), to that pointer value will result in a new pointer value that references the array element \( n \) elements way from the original element.
Putting these together...

... we observe that the nth element of an array called myArray can be referenced as

\[ \text{myArray + n} \]

Thus, the following expressions are equivalent

\[ \text{myArray [n]} \quad \text{and} \quad *(\text{myArray + n}) \]

\&\text{myArray [n]} \quad \text{and} \quad \text{myArray + n} \]
Using ‘delete’

- delete is the operator which is used to recycle memory allocated with new
- The forms of new and delete must match
  
  ```c++
  // for a single object
  int * x = new int;
  delete x;

  // for an array of object
  int *intp = new int[7];
  delete [ ] intp;
  ```
A delete example

Suppose ‘soup’ is a pointer variable that references an int containing 42

```
soup: 42
```

Then the statement
```
delete soup
```
recycles the space referenced by soup.

```
soup: 42
```
A potential problem

```c
int *soup, *salad;
soup = new int ( 42 );
salad = soup;
delete soup;
```

What does the picture of memory look like?
What’s the potential problem?
A potential security problem

• Recall that newly allocated space is filled with random data.
  – Where does that data come from? It’s not truly random, but it is whatever old data was left behind when some program used delete. Your deleted data is not gone; it is floating around somewhere in the computer’s memory, to be reallocated when another program needs the space.
A potential security problem

- So, what happens to the data in a credit card number when you delete it? What if some clever hacker could figure out a way to get that deleted memory reallocated to another program?
  - To avoid this problem, any sensitive data should have its value changed (set to zero) before calling delete.
Multi-dimensional Arrays

We declare a static 2-dimensional array as

```c
char chArray1 [ 3 ] [ 4 ];
```

What we envision is

```
0       1      2      3
```

```
chArray1 :
```

0
1
2
Static 2-D array in Memory

What we get in memory is

chArray1:

```
  row  0      0      0      0      1      1      1       1     2 2       2      2
column 0      1      2      3      0      1      2       3    0     1       2      3
```

We use double subscripts (chArray [ 2 ][ 1 ]) to reference the elements by row/column. The compiler uses the row number, column number, number of columns and array type to calculate a memory address.
A dynamic 2-D array

To dynamically allocate a 2-D array, we must create an array of pointers (the rows) in which each row element points to an array of elements (the columns).

We need only declare the name of the dynamic array correctly

```c
char **chArray2;
```
char **chArray2

This declaration is read as
“chArray2 is a pointer to a pointer to a char”
In reality, the compiler doesn’t “know” if you plan to use chArray2 to point to one pointer to char or as the name of an array of many pointers to char.
When we use it for dynamic 2-D arrays, we use it to point to an array of pointers to char
The code

// the array of pointers for the rows
chArray2 = new char* [ 3 ];

// now allocate the elements of each row
for (int r = 0; r < 3; r++)
    chArray2 [ r ] = new char [ 4 ];
Dynamic 2-D Array in Memory

We can still use double subscript (chArray[2][1]) to access the elements. The compiler uses pointer arithmetic to find the memory address.
delete’ing a dynamic array

To deallocate a dynamically allocated multi-dimensional array, we must deallocate memory in the reverse order that it allocated

```cpp
for (int r = 0; r < 3; r++)
    delete [ ] chArray2[ r ];
delete [ ] chArray2;
```
Exercises for the student

1. Draw the results of the following code

```c
char *fischer;
fischer = new char[3];
fischer[0] = 'p';
fischer[1] = 'k';
fischer[2] = '4';
```
2. Draw the results of the following code:

```c
struct soprano {
    int x;
    char ch[3];
};
typedef soprano* alto;

soprano *jane;
alto *joe;
jane = new soprano;
jane->x = 1;
jane->ch[jane->x] = 'G';
joe = new alto[4];
joe[0] = new soprano;
joe[0]->ch[2] = 'Q';
joe[3] = jane;
```
Even more exercises

3. Write the typedefs, declarations and code that will create the picture below

foo:

17

42
Still more

4. Write the typedefs, declarations and code that will produce the picture below
And one last exercise

5. Write the typedefs, declarations and code that will produce the picture below.

tom: