Pointers II

Class 30

Compile Time

- all of the variables we have seen so far have been declared at compile time
- they are written into the program code
- you can see by looking at the program how many variables will exist in the entire program
- they are in an area of memory called the stack
- this is fine when you know exactly how many variables you will need in the entire program

Compile Time

- all of the variables we have seen so far have been declared at compile time
- they are written into the program code
- you can see by looking at the program how many variables will exist in the entire program
- they are in an area of memory called the stack
- this is fine when you know exactly how many variables you will need in the entire program

- but sometimes you do not know in advance how many variables you will need

Run Time

- it is possible to create (and destroy) variables at runtime
- this is called dynamic allocation
- while a program is running, the logic may dictate that a new variable is needed, not known when the program was written

Run Time

- it is possible to create (and destroy) variables at runtime
- this is called dynamic allocation
- while a program is running, the logic may dictate that a new variable is needed, not known when the program was written

- there is a pool of memory that is available to be drawn from at need
- this pool is called the heap

New

- a program can allocate a piece of this memory at runtime by using the operator new

```
int* value = new int;
```

New

- a program can allocate a piece of this memory at runtime by using the operator new

```
int* value = new int;
```

- this line of code requests 4 bytes of memory from the heap

New

- a program can allocate a piece of this memory at runtime by using the operator new

```
int* value = new int;
```

- this line of code requests 4 bytes of memory from the heap
- the operating system responds with the address of the 4-byte chunk in the heap

New

- a program can allocate a piece of this memory at runtime by using the operator new

```
int* value = new int;
```

- this line of code requests 4 bytes of memory from the heap
- the operating system responds with the address of the 4-byte chunk in the heap
- since new provides an address, it must be assigned to a pointer

New

- a program can allocate a piece of this memory at runtime by using the operator new

```
int* value = new int;
```

- this line of code requests 4 bytes of memory from the heap
- the operating system responds with the address of the 4-byte chunk in the heap
- since new provides an address, it must be assigned to a pointer
- this new variable can be accessed by dereferencing the pointer variable value

Dynamic Allocation

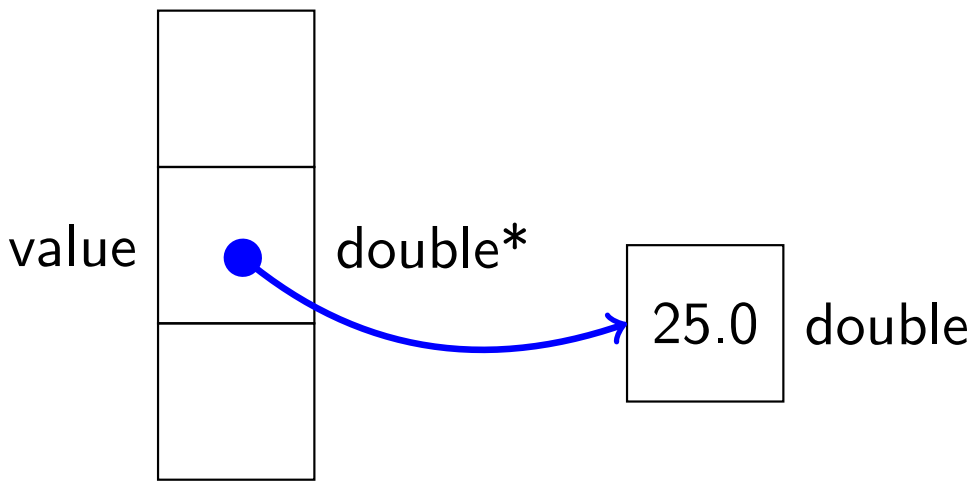
- here is a program fragment that uses a dynamically allocated variable

```
double* value = new double;
*value = 25.0;
*value *= 2.0;
cout << "the value is: " << *value << endl;
```

Diagramming Dynamic Memory

- even though a computer's memory is one huge linear list of locations
- we diagram the stack and heap as separate areas

```
double* value = new double;
*value = 25.0;
```



Dynamic Allocation of Arrays

- in reality, allocating a single variable isn't very useful
- the true power of dynamic allocation is in allocating an entire array at runtime

```
cout << "how many values do you need to store? ";
cin >> value_count;

int* values = new int[value_count];
```

- finally we can create an array based on a variable!

Stack and Heap

- the stack area of memory is where all statically (compile-time) declared variables exist
- the operating system gives it to your program when your program starts running and reclaims it when your program finishes

Stack and Heap

- the stack area of memory is where all statically (compile-time) declared variables exist
- the operating system gives it to your program when your program starts running and reclaims it when your program finishes

- the heap is where all dynamically (run-time) declared variables are allocated
- the programmer is responsible for allocating it, and the programmer is responsible for de-allocating it before your program finishes

Stack and Heap

- the stack area of memory is where all statically (compile-time) declared variables exist
- the operating system gives it to your program when your program starts running and reclaims it when your program finishes

- the heap is where all dynamically (run-time) declared variables are allocated
- the programmer is responsible for allocating it, and the programmer is responsible for de-allocating it before your program finishes

- when you use memory from the heap
- you have to give it back before the program finishes

De-Allocating Memory

- just as the new operator allocates a piece of memory
- the delete operator gives it back

```
double* value = new double;
```

must always be followed eventually by:

```
delete value;
```

De-Allocating Memory

- just as the new operator allocates a piece of memory
- the delete operator gives it back

```
double* value = new double;
```

must always be followed eventually by:

```
delete value;
```

```
int* values = new int[50];
```

must always be followed eventually by:

```
delete[] values; // note the syntax!!
```

- every program must have exactly as many deletes as news

Skipped Sections

- we will not do sections
  - 9.9
  - 9.10
  - 9.11