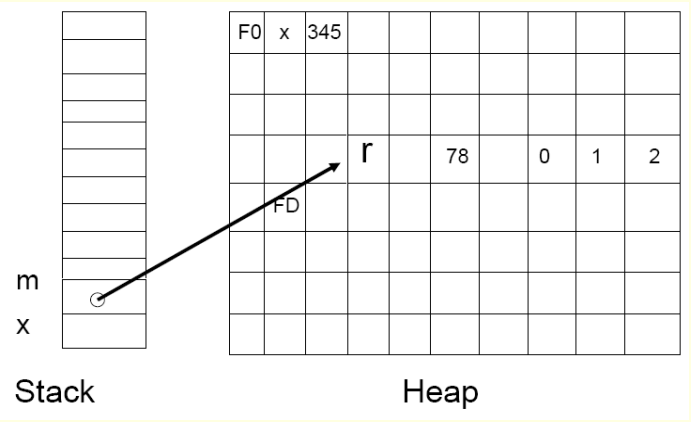
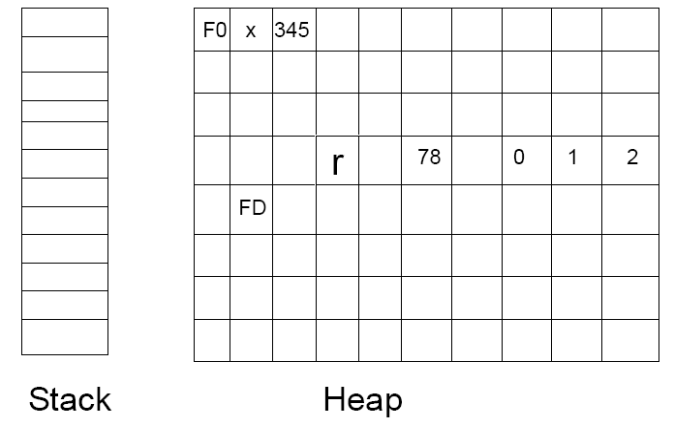
Lecture 6.3

Dynamic Memory Allocation

Stack and Heap with dynamic memory allocation



Stack and Heap when function goes out of scope

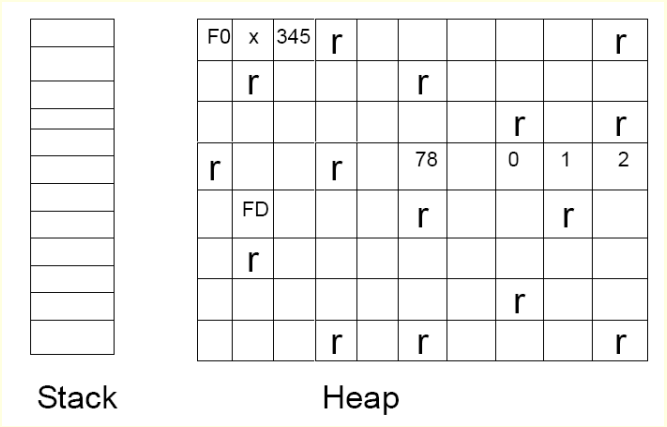


What happens if you call the function 15 times?

```
void myFunction (int x){
   char *m = new char('r');
   cout << m ;
}

int main (void){
   for (int i = 0; i < 15; i++)
   myFunction(3);
}
```

Stack and Heap when function goes out of scope **without** delete



What's the problem with this?

- As the program continued to operate, more and more memory will be lost from the heap (free store).
- If the program runs long enough, eventually no memory will be available, and the program will no longer operate.

What's the problem with this? (cont)

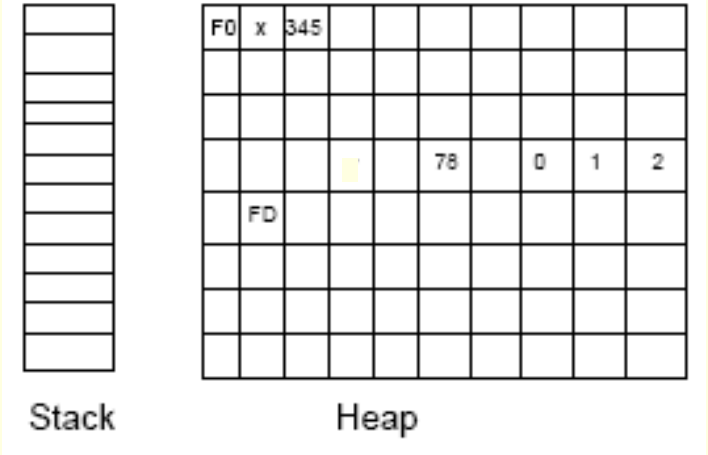
- Even if we don't run out of memory, the reduced pool of available memory affects system performance.
- The moral of all this: **Be sure to delete.**
- Every new should be paired with a delete in your code to avoid memory leaks.

If you "free" the memory with delete

```
void myFunction (int x){
    char *m = new char('r');
    cout << m ;
    delete m;
}

int main (void){
    for (int i = 0; i < 15; i++)
        myFunction(3);
}
```

Stack and Heap when function goes out of scope **with** delete



Dynamically Allocating Arrays

- Arrays of built- in and user- defined data types may be dynamically allocated.
- User- defined data types include classes.

```
int *pt = new int [ 1024 ]; //allocates an array
                            //of 1024 ints
double *myBills = new double [ 10000 ];
//allocates array of 10000 doubles
```


Important! Do not get confused!

- Note the difference between:

```
int *pt = new int [ 1024 ]; //allocates an array
                            //of 1024 ints
int *pt = new int ( 1024 ); //allocates a single
                            //int with value 1024
```


Initializing a dynamically allocated array

```
int *buff = new int [ 1024 ];
for ( i = 0; i < 1024; i++ )
{
   *buff = 52; //Assigns 52 to each element;
   buff++;
}
```

Initializing a dynamically allocated array (cont)

- Or

```
int *buff = new int [ 1024 ];
for ( i = 0; i < 1024; i++ )
{
    buff [ i ] = 52; //Assigns 52 to each
                     //element;
}
```


How to use "delete" with a dynamically allocated array

- The syntax of the "delete" operator for dynamically allocated arrays is slightly different from what we saw for single objects.

```
delete [ ] pt;
delete [ ] myBills;
```

- The square brackets after the delete tell the compiler to delete a dynamic array rather than a single object.

So what is different from what we are used to?

- Use new to allocate memory and always assign it to a pointer of the same type of our allocated memory

```
int *buff = new int [ 1024 ];
```

- The pointer can be used just as we are used to (with an index like an array or with the * to indicate "value")

```
buff [ 1 ] = 43;
```

or

```
buff++;
*buff = 43;
```

- Use delete when we are done.

```
delete [ ] buff;
```


Dangling Pointers

- Take a look at this snippet of code.

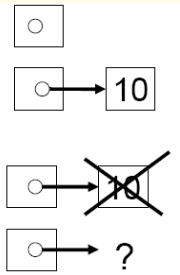
```
1 int *myPointer;
2 myPointer = new int(10);
3 cout << *myPointer << endl;
4 delete myPointer;
5 *myPointer = 5;
6 cout << *myPointer << endl;
```

What's wrong with this?

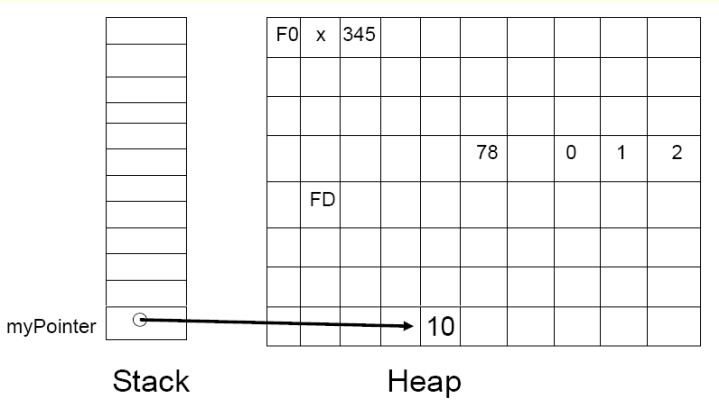
- myPointer is dangling!
- We've released the memory of the object whose address myPointer holds and then continued to use it.

What's happening?

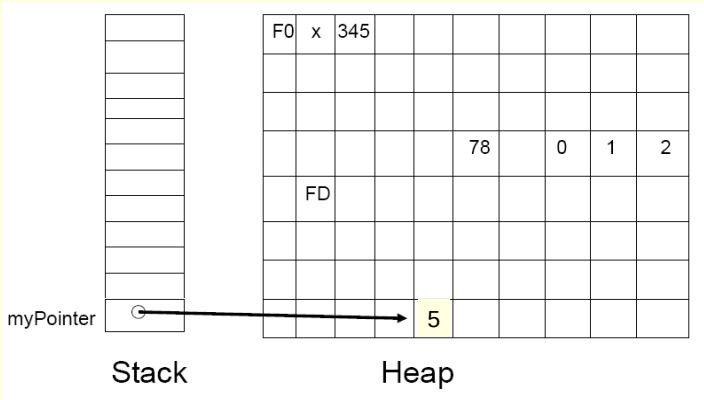
```
1 int *myPointer;
2 myPointer = new int(10)
3 cout <<*myPointer;
4 delete myPointer;
5 *myPointer = 5;
6 cout<<*myPointer;
```



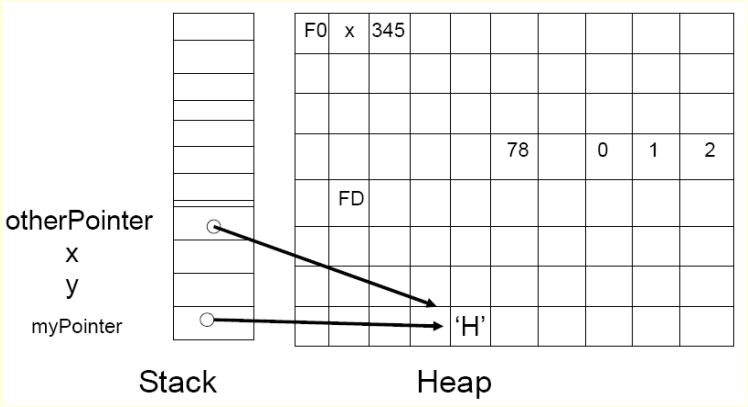
Stack and Heap before dangling pointer



Stack and Heap with dangling pointer



Stack and Heap with mem location used by other pointer



What is the problem with this?

- Although the program may run, this section of memory may be used by another dynamic object allocated after the delete.
- The values in that object will be corrupted by the continued use of myPointer.
- This is a very subtle programming bug and is very difficult to isolate.

Can you prevent it?

- To avoid this bug, always set a pointer to NULL, after the delete is called.
- Subsequent attempts to use the pointer will result in a run-time exception.
- This will immediately allow the bug to be identified and fixed.

Corrected code

```
int *myPointer;
myPointer = new int(10);
cout << *myPointer << endl;
delete myPointer;
myPointer = NULL;
*myPointer = 5; //This statement will cause a
                //run-time exception, now.
cout << *myPointer << endl;
```