*Notes 11/10*NULL isn’t defined by the C or C++ language, but it is defined by most libraries, including string, cstring, and iostream. You can also symbolize NULL by expressing an integer constant 0 in the context where a pointer is required.

Double p\* = NULL;  
double p\* = 0;  
  
Both of these point to the same value.

NULL is clearer because an offhand glance lets you see what the function returns – a pointer, not an integer. If you try to follow a NULL or 0 pointer it will usually return a lot of 0s (0x000000, 0x0, 0, 000000, etc.)

If you write cout << p, it won’t write out the value pointed by p but it will write the physical address in memory of the value pointed to.

Cout << &p would write out the value pointed.

Local variables are created “on the stack.” When your program starts up, there is some space reserved for the main routine’s local variables that is used dynamically – given and taken on the fly. Variables on the stack come into existence when they are declared, and when you leave the function they automatically go away. Variables declared outside of any function (“global variables”) are created in the global storage area/static storage area. They don’t go away and have a lifetime of the entire program.

You can tell the program when to make the storage go away if and when you want it to. This is called *dynamically allocated storage.* It is allocated from “the heap.”

Void f()  
{  
 int n;  
 cin >> n;  
 double a[n]; //NOT ALLOWED in standard C++.  
 a[0] = 12.3;  
 a[1] = 6.7;  
}  
  
If we want to make this work, we can do it with dynamically allocated storage.

Void f()  
{  
 int n;  
 cin >> n;  
 double\* a = new double[n]; //this will work. *New* is a key word signifying to use dynamic memory.  
 a[0] = 12.3;  
 a[1] = 6.7;  
}

When you use *new* the program will create the variable when it reaches that point in the program. This allows you to use *a* like an array.  
  
The only way to get rid of a dynamically allocated chunk of storage is to tell it to go away. It doesn’t go away until the end of the program, and it is useless once you leave the function because you have no way to access it! It becomes *garbage*. If f is called repeatedly, more and more doubles will be created and left in memory – this could become a serious problem if not dealt with! This is called a *memory leak*, and it can be a bitch to deal with because there are no symptoms until the program crashes. It can take months or years to have any symptoms.

How do we “give back” storage?

*Delete [] a;*will work in f().  
  
Void f()  
{  
 int n;  
 cin >> n;  
 double\* a = new double[n]; // assigns a dynamic array to pointer a   
 a[0] = 12.3;  
 a[1] = 6.7;

…  
 delete [] a; // returns the dynamic array to the heap. Following pointer a is now undefined behavior – be careful!  
 ...  
 a = new double[2\*n] // a is now defined again to another array  
 …  
 delete [] a; // a is a dangling pointer again.  
}  
  
Dangling pointers are erased when you leave the function, or when you reassign the pointer to another variable.

It is usually more typical to allocate storage in one function and delete it in another one.

Example:  
  
double\* getData(int m)  
{  
 double\* p = new double[m];  
 ...initiallizes values…  
 return p;  
}

Int main()  
{  
 int n;  
 cin >> n;  
 double\* a = getData(n); //pointer a is now pointing to a new double array  
 …  
 delete [] a; //removes the allocated new double array from memory  
}  
  
**Saying delete [] a does not delete a, it deletes the *storage that a points to*.**  
You can’t use delete to remove part of an array that was allocated – you have to delete all or none of it.  
If you try to delete a again after you have deleted it once, the behavior is undefined – it will probably crash the program right away.

You can declare new types of variables using the construct *struct.*

struct Employee //start your own types with a capital letter.  
{   
 string name;  
 double salary;  
 int age;  
}; //DON’T FORGET THE FUCKING SEMICOLON  
  
We say that Employee has 3 *data members*.   
  
int main()  
{  
 Employee e1;  
 Employee e2;  
  
 e1.name = “Fred”; //use the dot operator to assign elements of Employee; this is basically saying “e1’s name”

e1.salary = 60000;  
 e1.age = 30;  
  
 e2.name = “Ethel”;  
 e2.salary = 70000;  
 e2.age = 25;  
  
 e1.age++; //this increases the age element of e1 by 1.  
 cout << e2.same.size() //this writes the size of the name field of e2 (5)  
  
 Employee company[100]; //declares an array of 100 Employee items, each with 3 data members.  
 company[2].name = “Lucy”; //declares the name of the second element in the company array to “Lucy”  
  
}