**getline** is a library function that you use like this:  
***getline****(cin, personsName)*  
How is getline different from cin? Cin starts reading when it gets to the first non-space character and *stops* when it reaches a space again. Getline starts reading at the first non-space character and ***keeps reading until it reaches a newline character (almost always the end of the input, when the user presses enter).***

If you call a function and initialize a new variable, but put an & sign after the declaration, you are essentially declaring a new name for the same value. Ex:

*Void exchange(int& m, int& n)*

would copy the values of m and n from what was declared in this function the time it was called previously. This is called “**pass-by-reference**” because it looks at previous values used in the function and assigns new variables to the same values. If you make a change to the new variables declared, it will also change the old variables – this is important! It lets you go back and change stuff in your program later if the program calls for it.

Arrays can be used to hold a lot of values. You initialize them as *type name [number of items] = {item1, item2…itemN}*

Then you can call a particular element of the array as so: ***name [# you want]*.** **You start counting array elements at 0, not 1. So if you call *array[4]* you will get the *fifth* element.**

Arrays can have ints, doubles, strings, or any datatype you want.

If you need to declare a variable and use it only once (ex: you want to divide 2 ints without dropping the fraction), C++ has an easier method to do so – **temporary** **variables**. Declare as such

***Static\_cast<variableType>(source expression)***

You can declare a multidimensional array by using 2 brackets – ex: ***attendance[5][7]* makes a 35 element array, going from 0 🡪 6 on the xaxis and 0 🡪 4 on the yaxis.**

0 1 2 3 4 5 6   
1  
2  
3  
4  
C strings

S: 0 1 2 3 4 5 6 7 8 9 10 … 99  
 J e l l o \0 . . . . . .

T: 0 1 2 3 4 5 6 7 8  
 J e l l o \0 . . .

#include <cstring>  
using namespace std;  
  
char s[100] = “”;  
char t[9] = “Hello”;  
t[0] = ‘J’;  
cout << strlen(t); //writes 5  
  
cout << t;  
cin.getline(s, 100);  
  
s = t; //Error! Cannot assign arrays!  
**strcpy(s,t);** // ***strcpy(destination, source)*;**

s =+ “!!!”; //Error! Can’t do this with c-strings.

**Strcat(s, “!!!”); // *strcat****(string to add on to, string being added)* adds the second string onto the first  
  
if (t < s) // compiles, but doesn’t compare the text of the strings.

**Comparison: strcmp(s, t)  
 negative: if s comes before t  
 0: if s == t  
 positive: if t comes before s**

Pointers

A pointer is basically just an indication of where something is; it’s another way to implement passing by reference (mostly a C thing). You can also use them to traverse arrays, manipulate dynamic storage, and represent relationships in data structures.

To declare a pointer-to-double you use the syntax *double\* p.* Using this, instead of passing actual values to functions you can just pass an “arrow” to a value from another function. Put an “&” in front of the value when calling the function to use a pointer to the object rather than simply copying the value of it, and put a “\*” in front of the value whenever you call in in the new function. This means any operations done by the function to that variable will change the value of the original.

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.

**friends[0].name == (\*friends).name == friends ->name** //how to reference the first value of name in an array of friends.  
*This* always points to the function you are inside of. If you are implementing an element of a class, *this* always refers to the class at large.

When you run the destructor argument you don’t have to worry about array structure or anything except removing all the dynamically allocated memory. Remember, this function is only called when the class is about to go away – **none of the interior aspects matter** except getting rid of the memory allocations.

You can only **overload** a function if the passed variables **differ** in some way in terms of type or number.

What if there are 2 declared functions, one taking a double and the other taking an int, and you pass it a 3? The compiler will take the *best match* and use that function. In this case the best match would be the function taking an int, so it would call that one.

What about this:  
void f(double d, int j);  
void f(int d, double j);  
  
f(2,3);  
  
This will not compile because both are equally “good.” You will get an error message about “ambiguity.”

#include <cmath>  
//defines double sqrt(double d);  
//defines float sqrt(float f);  
  
… sqrt(2.4) … // 2.4 is a double and the function requiring a double will be called.  
… sqrt(2) … //2 is an int; the two functions that can be called are double and float. This won’t compile! It will give an ambiguity error.  
… sqrt(2.0) … //Fixed it. Calls the double function successfully.

There is no structure built into the language for complex numbers; however, we can create our own.  
  
struct Complex  
{  
 Couplex(double real; double imag)  
 double re;  
 double im;  
};  
  
bool operator==(Complex c1, Complex c2)  
{  
 return c1.re == c2.re && c1.im == c2.im;  
}  
  
Complex z1(3,4);  
Complex z2(-4,5);  
double d;  
..  
Complex z3(3,2\*d);

If ( z1 == z3 //means operator==(z1,z3))  
 …  
  
The operator word **overloads** the == comparison operator. Basically by declaring it as operator, you can call == to call that specific function (it’s much more natural .)