```cpp
//////// Lecture 83: Strings I - Raw Strings
#include <cstring> // C-STRING!
// let's say we want to write code to let a user fill in a first name and second name and then combine it into 1 full name.
const char* Combine (const char *pfirst, const char* pSecond){
    char fullname[20];
    strcpy(fullname, pfirst);
    strcpy(fullname, plast);
    return fullname;
}
int main(){
    char first[10];
    char last[10];
    std::cin.getline(first,10);
    std::cin.getline(last, 10);
    char fullname[] = Combine(first, last);

    return 0;
}
// This won't work. We are returning address of a local variable, which will be gone by the end of the scope. So basically this results in undefined behaviour.
// The last time of fullname is constrained to this function, so there is no guarantee that it will be available after the function has returned. But if you disregard this warning and run the executable anyway, you can see this resulted in undefined behaviour (Garbage values)
// it print garbage values because fullname is destroyed after the combine() function returns.
// so its address is reclaimed by other parts of the code.
// so if you try to get the data from main(), there is NO GUARANTEE(it's possible, but no guarantee) that the fullname has not been overwritten.
// In our case, the data is lost.

// This is not the only problem, we are using FIXED SIZE CHARACTER ARRAYS. What if the first and last
```

```
names do not fit?  Will the full name fit into the array?
// So obviously we cannot use the fixed size array here, so we will use the dynamic memory allocation.

////////////
// So we will allocate the amount of memory we want to allocate using new - where we compute the length of the first name and the last name,
const char* Combine(const char* pfirst, const char* pSecond)
{
    char* fullname = new char[strlen(pfirst) + strlen(pSecond)];
    // now this should work.
    strcpy(fullname, pfirst);
    strcpy(fullname, pSecond);
    return fullname; // This should work! why? Because we created memory on the heap, which goes beyond the lifetime of a function.
}
// With the same main(), this would compile. We allocated memory on the heap, but we need to deallocate it somewhere else. So we need to deallocate it in main().
int main(){
    char first[10];
    char last[10];
    std::cin.getline(first,10);
    std::cin.getline(last, 10);
    const char* fullname = Combine(first, last);
    std::cout << fullname << endl;
    // If we look at this from main(), we don't know whether it is pointed to a character or a string.
    // So we'll have to go into the Combine() function and see how memory has allocated.
    delete []fullname; // delete with the subscript.
    return 0;
}

// So you have to remember all these minor details
```

```
while writing your code.
// And it is possible that you'll forget some of these details.

// when you run this in debug mode however, the program crashes. The reason for this is when you allocate memory for the combined string you also need to allocate one extra byte for the null terminating character.
// strcpy and strcat() automatically append a null terminating character at the end of the string, regardless of whether memory is available or not. in our case, since we did not allocate memory for the null terminator, these functions will cause a BUFFER OVERFLOW, and this is the cause of the crash when we try to free the memory because the runtime detects that the memory is corrupted.
// so we need to allocate 1 byte extra, so we start again in debug mode, and it works again without any errors.
const char* Combine(const char* pfirst, const char* pSecond)
{
    char* fullname = new char[strlen(pfirst) + strlen(pSecond)+1]; // for the null terminating character.
    // now this should work.
    strcpy(fullname, pfirst);
    strcpy(fullname, pSecond);
    return fullname; // This should work! why? Because we created memory on the heap, which goes beyond the lifetime of a function.
}

// this is why C++ has a class for strings - they are objects, and we'll use them in the next video, because C strings are just error prone.


/////////// Lecture 84: Strings II - std::string
#include <string>
// The string class has many constructor, it provides
```

```
  a default constructor, copy constructor, constructor thorugh which the string object can be initialised with a raw string and other constructors.

// initialise and assign:
std::string s; // s will be empty - this will invoke the parameterised constructor for the string. You can either initialise it like this and you may assign the string to it later
s = "Hi!"; // the assignment operator is overloaded to accept a raw string.

//Access
// Individual elements
s[0] = 'w';  // overloaded subscript operator
char ch  = s[1];

// If you would like to print the string on the console, you can directly do it.
std::cout << s << std::endl;
std::cin >> s ; // cin will stop reading once it encounters the first space.
// If you would like to read the entire line, then there is a global function called getline() that takes the stream and the string object.
std::getline(std::cin, s); // stream is std::cin, string object is s.
// Note that the count of characters does not need to be specified.

// Size functions

// If you would like to know how many characters are inside the string object, you can use the method length()
// Here's one advantage of the string class compared to a raw string - if you would need the length of a raw string and if you need a length of a string object, the string object will be faster because it CACHES the string length.
// So querying the length will take constant time:
```

```
// So querying the length will take constant time.
s.length();

// Compared to
strlen(s.c_str()); // which would take LINEAR time

// Insert and concatenate
// The + operator is overloaded for concatenation of 2 strings, as well as the += operator.
std::string s1{"Hello"},  s2{"World"};
s = s1 + s2; // this feels more natural, the + operator is going to concatenate s1 and s2 and the result would be stored in s.

// If you want to add string to an existing string object, you can use the += operator.
s += s1;

// There is a method called "insert()" you can decide where the string should be inserted into the target string, so we can use insert() and it has a lot of overloads, so we use the one that inserts a string at a specific position.
// So the first argument maybe the position - like 6 and the next argument is a string. This takes O(n) time.
s.insert(6,"Hello world!"); // index of insertion, raw string

// Comparison
if (s1 != s2) {}

// Removal
s.erase(0,5); // this will erase the first 5 characters of the string.
s.clear(); // clears the entire string.

// Search
auto pos = s.find("World", 0); // finds a substring within a given string.
// Substring, index to start searching from, so if you put index 0 it will search the whole string.
// If it is not able to find it, it will output an
```

```
// If it is not able to find it, it will output an 'npos' constant, which has a value of -1.
if (pos != std::string::npos){
    // Substring found
}



/// Initialising Strings
std::string first1 = "Umar";
std::string last1("Lone");
std::string name{"Umar Lone"}; // PREFERRED (C++11)

// There is no difference between these 3 styles of initialisation, but you should always prefer direct initialisation.
// C and C++ already have a feature where you can prefix or suffix a character to a type, and this character is called a literal.
// it simply changes the meaning of the value.

// for example, if I want to create an unsigned integer type, I would initialise it like this.

unsigned int value = 100u; // this 'u' is a literal and there are other types also where you can prefix and suffix literals. C++11 added the ability of creating custom literals, and in C++14, the standard added literals for some library types.


// std::string name2 = "Umar Lone"s; // this initialisation isn't valid yet, this 's' is a literal, and it is defined inside the standard library, but to use it, you have to open its corresponding namespace.

// So something like this:
using namespace std::string_literals; // this line is needed!!
std::string name2 = "Umar Lone"s; // this literal is actually a FUNCTION, and that FUNCTION accepts this type as an argument and initialises a string object
```

```
    type as an argument and initialises a string object
    and returns that as the return value.

// I can further reduce the syntactic noise here
    using auto.
auto name2 = "Umar Lone"s;

// You can create literals for your own user defined
    types, but this will be covered in a future lecture.
```