This lecture continues our discussion about heap memory.

Sometimes, it is necessary allocating memory using functions. This lecture talks about two ways to allocate memory by calling functions.

Earlier lectures said that it would be a good habit calling malloc and free in the same function so that we always see them in pairs.

Sometimes, we have to call malloc and free in different functions. This may occur when a program is complex and needs to do many things. As a result, the places for malloc and for free have to be far apart. Putting malloc and free in the same function will make this function too longer and difficult to write and debug.

Let’s consider two simplified examples and understand how we may use functions to allocate memory.

This is the entire program.

We will look into the details in a moment.

Let’s start with the main function.

The program has finished assigned five to the local variable size.

The local variable A R R has not been initialized. Thus, its value is marked as U, meaning unknown.

Next, the program calls the function my A L L O C 1. A new frame is pushed to the top of the stack memory.

The return location is the line after calling my A L L O C 1. The value address is the address of A R R because A R R is at the left hand side of the assignment when calling the function.

The argument size is 5 because its value is copied from the size in the main function.

The local variable pee is a pointer and its value is U because we have not initialized the value yet.

The next step calls malloc. Suppose malloc assigns 2000 to pee. This is what the stack and heap memory looks like. Please remember that the memory allocated by malloc is not initialized. Thus, the values are marked U, meaning unknown.

This function returns p’s value 2000 and this value is written to the value at address 104.

Therefore, A R R’s value is address 2000.

The my A L L O C 1 function ends and the top frame is popped. This is what the stack and heap memory is.

Please remember to call free even though malloc is not called directly in the main function.

The next example is a little more complex. Instead of using a function’s returned value to return the allocated memory, this example uses a function’s argument.

Because we use an argument, this argument will reside on the stack memory. When this function ends, the top frame is popped. If we want to make the memory reachable after this function call, we have to use the pointer of a variable in a lower frame.

Let’s see how this may work.

Please notice that the argument p has two \*. The reason will become more obvious in a moment. Also, when calling my A L L O C 2, we need to add ampersand in front of A R R. .

This slide shows the stack memory after the main function has assigned 5 to the local variable size.

The pointer A R R has not been initialized and its value is marked U, meaning unknown.

The program calls the function My A L L O C 2. The argument pee is the address of A R R and thus pee’s value is address 104. The argument size is copied from the size in main and both are 5.

My A L L O C 2 has a local variable called tee. Its value is unknown right now.

The next line calls malloc and assigns the address to tee’s value.

The next line assigns tee’s value to \* pee. This \* pee appears on the left hand side of the equal sign. Thus, we need to use the left hand side rule. The first step takes pee’s value. Treat the value as an address. Go to that address and modify the value.

The value of A R R is changed to 2000.

Please remember the type rule: If A R R’s type is I N T \*, then ampersand A R R’s type is INT \* \*, notice there are two \*.

When function my A L L O C 2 finishes, the top frame is popped. This slide shows the stack and heap memory.

Next, we talk about a common mistake using a function to allocate memory.

If we use G C C to compile the program, a warning message will be given.

Let’s ignore the warning message for now and understand why this program does not work.

This slide shows the state of the stack memory after assigning 5 to the local variable size.

This shows the state of the stack when entering the function wrong A L L O C before the first statement.

The value of pee is copied from the value of A R R and both of them are unknown.

Calling malloc changes the value of pee to 2000.

Please notice that the value of A R R is unchanged.

After the function wrong A L L O C finishes and the top frame popped, the value of A R R is still unknown.

The reason A R R is unchanged is that calling wrong A L L O C does not use the address of A R R.