Suppose we have the following declarations:

int i = 1, j = 2;

int\* p = &i;

int\* q = &j;

Then the following are equivalent:

i = j;

\*p = j;

\*p = \*q;

All assign the 2 in j to i. So to copy an int from j to i via pointers, then those pointers must be pointers to int. Duh.

Now let’s mix things up a little. Add the following declarations:

int\*\* p2 = &p;

int\*\* q2 = &q;

What is the output of the following code?

i = 1;

\*p2 = \*q2;

cout << \*\*p2 << endl; // 2

Here we have copied the address in q to p indirectly through p2 and q2, which are pointers to pointers to int (a int\*\*). So in general, whenever we want to copy a value of type T from one location to another indirectly via pointers, those pointers have to be of type T\*. Duh again.

Now pretend that we’re using arrays of chars instead of ints:

char a[10]{"hello"};

char b[10]{"goodbye"};

char\* s = a; // Same as = &a[0] because of pointer decay

char\* t = b; // Same as = &b[0] because of pointer decay

cout << s << endl; // hello

cout << t << endl; // goodbye

s = t;

cout << s << endl; // goodbye

The last assignment, s = t, copies the address in t (b’s address) to s, so s and t both point to b.

Now let’s increase the level of indirection one more, like we did before.

s = a; // Restore s to point to a

char\*\* s2 = &s;

char\*\* t2 = &t;

\*t2 = \*s2; // Same as t = s;

cout << t << endl; // hello

Since I am copying the address s holds into t indirectly through pointers to s and t respectively, those pointers must be pointers to pointers to char (i.e., char\*\*). Duh yet again.

Now consider what you must do in Program 1. You want to insert a particular byte address (a char\*) in an arbitrary place in memory (p, say, another char\*). The key difference here is that we have char\* addresses, but want to copy a char\* value into a particular bytes address. Suppose p is a char\* that points to where we want to insert the char\* address value q. We can’t just do the following:

\*p = \*q;

because this will only copy a single byte. Why? Because these are pointers to a character (byte), and when we use indirection through pointers, only a char (the type being pointed to) is copied (because they are pointers to char!). We need to reinterpret p as a pointer to pointer to char (a char\*\*), so we can place a char\* at its location, instead of a single char.

This should be another “duh”.

So we can do the following:

char\*\* p2 = reinterpret\_cast<char\*\*>(p);

\*p2 = q;

We have to use reinterpret\_cast since a char\* and a char\*\* are really not related (they are different “dimensions” of pointers). Now, when we use the expression “\*p2” on the left side of an assignment, it means “let the value at the address p2 (same address as p) be the char\* stored in the variable q.

Similarly, to read the value at an arbitrary address p as a char\* instead of a char, we can do the following:

char\*\* p2 = reinterpret\_cast<char\*\*>(p);

char\* x = \*p2;

If you understand simple pointer indirection 101, this is straightforward, and shows you an easy way (better than memcpy) to copy pointers around.