CS16, 10W, UCSB—Handout for 02/18—The type expression game (PDF)
http://www.cs.ucsb.edu/~pconrad/cs16/10W/lectures/02.18 PLEASE CONTINUE TO READ: In Etter: 6.4, 7.1, 7.2

Why type is important

The concept of type is very important in C. This is also true in C++, which you'll study if you go on to CS24, CS32, and CS48. And, the way types work in C++ is very similar—almost identical, in fact— to how they work in C.

When you get error messages from either the Ch interpreter or the cc compiler, the message may say things like: `found (int *) value where (int) value was expected`. So understanding the difference between int and int * is very important to getting your programs to compile correctly, and understanding the messages you get when they don't.

The basic exercise

A basic exercise I've used in C/C++ courses for many years is the one illustrated below. We start with a segment of code, such as this:

```
int main(int argc, char *argv[])
{
  int a;
  int *b;
  // rest of the program would go here
  return 0;
}
```

This is obviously not a "useful" C program—to be useful, there would have to be some more code at the comment line that says "rest of code goes here". However, it does give us a context to answer some questions about type.

The question is in the form a table where the left column contains an expression, and the right column asks what the type of that expression would be. For example:

What you'll be given as the problem

Expression	Type
a	
b	
*a	
*b	
&a	
&b	

What the correct answers are:

Expression	Type
a	int
b	int *
*a	error
*b	int
&a	int *
&b	int **

Here's how each of those is solved:

`a` is of type `int`, so the correct answer is `int` (cover up the `a` in the declaration `int a;` and what is left is `int`).

`b` is of type `int *`, so the correct answer is `int` * (cover up the `b` in the declaration `int *b;` and what is left is `int` *).

`*a` is an error—since `a` is not a pointer, it cannot be dereferenced. So the correct answer is **error**.

*b however, is not an error: since b is of type `int *`, it points to something of type `int`. So the answer is `int`

- The unary * operator means "dereference", i.e. follow the pointer.
- So if we follow an `int *` pointer, to what it points to, what we are left with is an `int`. So the correct answer is `int`
- Here's another way to think about it:
    - The unary * in an expression takes away a star from the declaration.
    - So if a * appears in front of something of type `int *`, the stars cancel each other out, and we are left with `int`.
    - Using this rule, if there isn't a star to remove, then you have an error.

For &a, we start by noting that a is of type int, and taking the address of an int gives us an int *. So the answer is int *

- You can also think of it this way: an & operator *adds a star to the type* (provided the expression it is applied to is a valid expression)

Similarly for &b, since b is of type int *, taking the address of b gives us an int **

- The adding a star rule still applies. An int ** is a pointer to an int *, i.e. an pointer to a pointer to an int.
- Or, we could say that an int ** is the address of a variable, which itself contains the address of some other int variable.

A note about the ** variables:

- ** type variables do occur in practice when handling certain pointer situations that arise in CS24 and CS32.
- *** and **** and even higher levels of star are legal, but are much more rare in practice.
- If your code is getting to the point of needing four or more stars, it may be getting too complex,
  and you may want to look for a simpler way to solve your problem.

Finally, a note that if we put in `double` (or `char`, etc.) instead of `int`, the rules are the same:—e.g. for `double *c;` we have:

- `c` of type `double *`, `*c` of type `double`, and `&c` of type `double **`.

Please turn over for more.

Continued from other side

Adding arrays into the type expression game

As we recall the name of an array is a pointer to its first element.
So in the type expression game, if we are given the name of an array of int for example, we should treat it as an int *.
Also, each element of the array is of the type of the array, and array subscripting, is just another form of pointer dereference, i.e.

- a[0] is equivalent to *(a)
- a[1] is equivalent to *(a + 1)

See if you can use those facts to understand the answers in the example below.

Code:

```
int main()

{
  int a[] = {12, 23, 45};
  double b[] = {0.4, 0.5, 0.6};
  // ...
  return 0;
}
```

What the correct answers are:

Expression	Type
a	int *
*a	int
a[1]	int
a[3]	int (*see explanation)
&a	int **
b	double *
b[2]	double
*b[2]	error
&b[2]	double *

*Note that although it is likely a logic error to subscript a[3] when a contains only elements a[0], a[1] and a[2], it is *not* a *type* error. So the correct answer here is still `int`, not error.

Adding structs into the type expression game

As we recall from the handout from 02.16, and sections 7.1 and 7.2 in the textbook, a struct is a way to *create a new type*—in addition to int, double, char, char *, int *, etc. A struct has members inside it: for example:

```
struct Point {
  double x;
  double y;
};




int main()
{
   struct Point p;
   struct Point *q;
   int a;
   // ...
   return 0;
}
```

In the type expression game, if we reference a variable that is an entire struct, the answer is the type of that struct.

- so p is of type struct Point, and so is *q
- q is of type struct Point *, and so is &p
- *p is an error, since p is not a pointer to anything.
- &q is of type struct Point **

If we reference an individual member of a struct, the answer is the type of the member of the struct.

- p.x is of type double
- &(p.y) is of type double *
- (*q).x is of type double
- (*p).x is an error, since we cannot dereference p (it isn't a pointer)

If we reference a member of a struct that doesn't exist, or use the . operator on something that isn't a struct, that's an error.

- p.z is an error, since p is of type struct Point, and there is not member named z in a struct point
- q.x is an error, because q isn't a struct Point, it is a `struct Point *` so we can't use the . on it.
- a.x is an error, because a isn't a struct at all—it is an int.

Finally, we need to keep in mind that p->x is an abbreviation for (*p).x

So whenever we see p->x, we can just convert to (*p).x and then apply the rules above.
Eventually, you'll get used to the p->x syntax, and you won't need to convert to understand what to do with the p->x notation.

End of Handout for 02.18