Tip 29: Preprocessor tricks!

Ben Klemens

28 November 2011

level: getting advanced
purpose: turn code into more code

The token reserved for the preprocessor is the octothorpe, #, and the preprocessor makes three (3) entirely different uses of it.

You know that a preprocessor directive like `#define` begins with a # at the head of the line. Whitespace is ignored, so here's your first tip: you can put throwaway macros in the middle of a function, just before they get used, and indent them to flow with the function. According to the Old School, putting the macro right where it gets used is against the Correct organization of a program (which puts all macros at the head of the file), but having it right there makes it easy to refer to and makes the throwaway nature of the macro evident.

Next use of the #: in a macro, it turns input code into a string. Here's the code from Tip #23 (Entry #073), slightly rewritten:

```
#include <stdio.h>

int main(){
    #define peval(cmd) printf(#cmd ": %g\n", cmd);
    double *liszt = (double[]){1, 2, 3};
    double list[] = {1, 2, 3};
    peval(sizeof(liszt)/(sizeof(double)+0.0));
    peval(sizeof(list)/(sizeof(double)+0.0));
}
```

When you try it, you'll see that the input to the macro is printed as plain text, because `#cmd` is equivalent to `cmd` as a string.

So `peval(list[0])` would expand to

```
printf("list[0]" ": %g\n", list[0]);
```

Does that look malformed to you, with the two strings `"list[0]" ": %g\n"` next to each other? Next preprocessor trick: if two literal strings are adjacent, the preprocessor merges them into one: `"list[0]: %g\n"`. And this isn't just in macros:

```
printf("You can use the preprocessor's string "
```

```
        "concatenation to break long lines of text "
        "in your program. I think this is easier than "
        "using backslashes, but be careful with spacing.");
```

Conversely, you may want to join together two things that are not strings. Here, use two octothorpes, which I will herein dub the hexadecathorpe : ##. If the input is `LL`, then when you see `name ## _list`, read it as `LL_list`, which is a valid and useful variable name.

Gee, you comment, *I sure wish every array had an auxiliary variable that gave its length.* OK, let's write a macro that declares a local variable ending in `_len` for each list you tell it to care about. We'll even make sure every list has a terminating marker, so you don't even need the length.

That is, this macro is total overkill, but does demonstrate how you can generate lots of little temp variables that follow a naming pattern that you choose.

```
#include <stdio.h>
#include <math.h> //NAN

#define Setup_list(name, ...) \
    double *name ## _list = (double []){__VA_ARGS__, NAN};   \
    int name ## _len = 0; \
    for (name ## _len =0; !isnan(name ## _list[name ## _len]); \
                              name ## _len ++) /*do nothing.*/;


int main(){
    Setup_list(items, 1, 2, 4, 8);
    // Now we can use items_len and items_list:
    double sum=0;
    for (double *ptr= items_list; !isnan(*ptr); ptr++)
        sum += *ptr;
    printf("total for items list: %g\n", sum);


    // Some systems let you query an array for its
    // own length using a form like this:
    #define Length(in) in ## _len

    sum=0;
    Setup_list(next_set, -1, 2.2, 4.8, 0.1);
    for (int i=0; i < Length(next_set); i++)
        sum += next_set_list[i];
    printf("total for next set list: %g\n", sum);
}
```

Discussion:
The macro above really is pretty bad form, and tries to hard for the sake of demonstrat-

ing multiple new variables. But there are some in the Old School who eye all macros warily. C is built upon *expressions*, which evaluate to produce other expressions, and can be plugged in wherever. That's why you can write things like `if (x = (y==f(z))...` and it all makes sense (or at least, compiles correctly). The macros above don't produce expressions, but are blocks of code that generate new variables and do all sorts of math along the way.

Macros are a pain to debug and can do tricky things, so here's my subjective style tip: lean toward macros that aren't expressions, and which can't be used in the middle of a stream of math even if you wanted to. When something breaks, you'll need to hunt through the macro to find the error; it's a pain, but at least you have the block of code the macro produces isolated.