Chapter 35: X-macros

X-macros are a preprocessor-based technique for minimizing repetitious code and maintaining data / code correspondences. Multiple distinct macro expansions based on a common set of data are supported by representing the whole group of expansions via a single master macro, with that macro's replacement text consisting of a sequence of expansions of an inner macro, one for each datum. The inner macro is traditionally named `X()`, hence the name of the technique.

Section 35.1: Trivial use of X-macros for printfs

```
/* define a list of preprocessor tokens on which to call X */
#define X_123 X(1) X(2) X(3)

/* define X to use */
#define X(val) printf("X(%d) made this print\n", val);
X_123
#undef X
/* good practice to undef X to facilitate reuse later on */
```

This example will result in the preprocessor generating the following code:

```
printf("X(%d) made this print\n", 1);
printf("X(%d) made this print\n", 2);
printf("X(%d) made this print\n", 3);
```

Section 35.2: Extension: Give the X macro as an argument

The X-macro approach can be generalized a bit by making the name of the "X" macro an argument of the master macro. This has the advantages of helping to avoid macro name collisions and of allowing use of a general-purpose macro as the "X" macro.

As always with X macros, the master macro represents a list of items whose significance is specific to that macro. In this variation, such a macro might be defined like so:

```
/* declare list of items */
#define ITEM_LIST(X) \
      X(item1) \
      X(item2) \
      X(item3) \
/* end of list */
```

One might then generate code to print the item names like so:

```
/* define macro to apply */
#define PRINTSTRING(value) printf( #value "\n");

/* apply macro to the list of items */
ITEM_LIST(PRINTSTRING)
```

That expands to this code:

```
printf( "item1" "\n"); printf( "item2" "\n"); printf( "item3" "\n");
```

In contrast to standard X macros, where the "X" name is a built-in characteristic of the master macro, with this style

it may be unnecessary or even undesirable to afterward undefine the macro used as the argument (`PRINTSTRING` in this example).

Section 35.3: Enum Value and Identifier

```
/* declare items of the enum */
#define FOREACH \
      X(item1) \
      X(item2) \
      X(item3) \
/* end of list */

/* define the enum values */
#define X(id) MyEnum_ ## id,
enum MyEnum { FOREACH };
#undef X

/* convert an enum value to its identifier */
const char * enum2string(int enumValue)
{
    const char* stringValue = NULL;
#define X(id) if (enumValue == MyEnum_ ## id) stringValue = #id;
    FOREACH
#undef X
    return stringValue;
}
```

Next you can use the enumerated value in your code and easily print its identifier using :

```
printf("%s\n", enum2string(MyEnum_item2));
```

Section 35.4: Code generation

X-Macros can be used for code generation, by writing repetitive code: iterate over a list to do some tasks, or to declare a set of constants, objects or functions.

Here we use X-macros to declare an enum containing 4 commands and a map of their names as strings

Then we can print the string values of the enum.

```
/* All our commands */
#define COMMANDS(OP) OP(Open) OP(Close) OP(Save) OP(Quit)

/* generate the enum Commands: {cmdOpen, cmdClose, cmdSave, cmdQuit, }; */
#define ENUM_NAME(name) cmd##name,
enum Commands {
  COMMANDS(ENUM_NAME)
};
#undef ENUM_NAME

/* generate the string table */
#define COMMAND_OP(name) #name,
const char* const commandNames[] = {
  COMMANDS(COMMAND_OP)
};
#undef COMMAND_OP

/* the following prints "Quit\n": */
```

```
printf("%s\n", commandNames[cmdQuit]());
```

Similarly, we can generate a jump table to call functions by the enum value.

This requires all functions to have the same signature. If they take no arguments and return an int, we would put this in a header with the enum definition:

```
/* declare all functions as extern */
#define EXTERN_FUNC(name) extern int doCmd##name(void);
COMMANDS(EXTERN_FUNC)
#undef EXTERN_FUNC

/* declare the function pointer type and the jump table  */
typedef int (*CommandFunc)(void);
extern CommandFunc commandJumpTable[];
```

All of the following can be in different compilation units assuming the part above is included as a header:

```
/* generate the jump table */
#define FUNC_NAME(name) doCmd##name,
CommandFunc commandJumpTable[] = {
  COMMANDS(FUNC_NAME)
};
#undef FUNC_NAME

/* call the save command like this: */
int result = commandJumpTable[cmdSave]();

/* somewhere else, we need the implementations of the commands */
int doCmdOpen(void) {/* code performing open command */}
int doCmdClose(void) {/* code performing close command */}
int doCmdSave(void) {/* code performing save command */}
int doCmdQuit(void) {/* code performing quit command */}
```

An example of this technique being used in real code is for GPU command dispatching in Chromium.