A Comprehensive description of CIP::input_handler:

In order to use input_handler to interpret input, one must first construct it by providing the input it is to parse, then set it up using its member functions, then call the member function interpretInput(). Excepting the two get-functions, which return strings, the class will not do anything until interpretInput is called (unless undefined behaviour is invoked, as described later), and will do everything it is supposed to do in that function call.

An input_handler is constructed by passing an int and an array of c-strings. The class was designed so that the arguments provided to main(int argc, char* argv[]) can be passed directly to input_handler's constructor. The specific requirements on the two variables are:

- argc is equal to one greater than the number of arguments which are to be processed by the IH. This means that argc must be at least one.
- argv begins with a string which is not a user-provided argument, where this string is implementation-defined. In some implementations, it may represent the program's full path, in others it may be the empty string, etc.
- argv must contain at least argc elements. Any elements past the argc-th element are ignored.
- argv *does not* need to end with an empty string.

The class contains a number of variables which are treated as constant during interpretInput, but can be set prior using the **Behavior/Constant-Setting Functions.** These variables (and their defaults, in brackets) are:

- `escapeChar ('\\') – the character used to allow ordered variables strings to begin with the character that denotes flags`
- `flagDelim ('/') – the character used to denote flags`
- `flagVarDefaultMarker ("~") – if an input to a flag variable with a default matches this string, said var will be set to its default`
- `flagFillDefaultMarker ("~~") – if an input to a flag variable with a default matches this string, said var` *`and all remaining vars for that flag`* `will be set to their defaults`

`(note: escapeChar currently cannot be used to escape the above two strings)`

- `helpMessage – the message which will be displayed to the user when they ask for help (with "/?" or "[flagDelim]help")`
- `minOrderedInput – the smallest number of ordered inputs which the program can accept`

As previously stated, the point at which interpretInput is called is the point where input_handler does everything that it is supposed to do. Before continuing to read this document, you may find it useful to read through interpretInput()'s documentation, so that you understand exactly what it does.

Containing Variables and Functions in the Class:

There are three things which the class contains: Flags, ordered variables, and tests.
Ordered variables consist of a variable-reference and (optional) a default value
Flags consist of any number of variable-reference and (optional) a default value for each
Tests consist of a CIP::function<> pointer.

For each of the above thing-names, there are two functions: setNum[name] and set[name]. For each of them, the first function must be called once, possibly through the macros CIP_[name]_START and CIP_[name]_END, before the second is called at all. Afterwards, the second may be called any number of times.
Variable-references for ordered variables and flags are handled similarily. In both cases, they are held as void-pointers, since there is no guarentee that all variables contained in this function will be of the same type (in fact, it's very likely they will not be so). orderedVars is a private void* array, where each element is a pointer to a variable. flagVars is a private void** array, where each element is a void* array containing that flag's arguments (or is a nullptr, if said flag has no arguments).
Both have a second array of the same form, which contains a list of integral values that correspond to the types pointed at by each pointer. This library contains a sub-namespace TYPE, which defines constants for that purpose.
In any case where all variables and defaultVals passed to setOrderedVar/setFlag are the same type, there is a template which removes the need to pass an array of type constants. This is always true for setOrderedVar, as only one var/defaultVal can be passed to that function at a time, but is not always true for setFlagVar. As such, it may be necessary to provide setFlagVar with arrays of void pointers. Do so through the function vars.
Lastly, it is possible to define aliases for flags. Flag aliases are a set of names which are different from a flag's 'main' name, which is the name that will be placed in any strings which the.

SetTest works a bit differently, as it is always either passed a CIP::function<bool> or the materials needed to build one. It's a lot simpler as a result. There's only one concept you need to understand, and that is the concept of test indexing/sequences.
When interpretInput runs the set tests, it runs each test sequence in the order in which they are defined. If any of them return false, test_failed is thrown. A test sequence returns true whenever any test in that sequence returns true. If a test in a sequence returns true, none of the remaining tests run.
Tests sequences are stored (and run) in the order in which they are defined, and under any circumstance where information about test results makes it out of interpretInput, the index of the entire sequence a given test was in is provided.
Test sequences can be used as a means of allowing recovery from failed tests. For example, you may test for the presence of a file and return false if it exists
(perhaps with `fptr(CIP::NOT, CIP::FILE_EXISTS_AND_CAN_BE_OPENED(filename)`), and then set the next test to ask the user if they wish to overwrite that file (say by using CIP::`YESNO(CIP::NAME_PLUS(`
`filename, "warning: ", "will be overwritten! Are you sure? (y/n)"), false`).
Alternatively, a failed test may be recoverable in some circumstances, so you could define a test-function with side affects which attempts to fix the problem and returns false if it fails, and place it in a sequence with the first test. Test sequences can contain as few as one member, so simple tests are still acceptable.

Dynamic Input:

The class is designed to take command line input and interpret it, but sometimes one wants a program which accepts command line input to be able to ask their user for such input if it is run with no (user-provided) arguments, so that the user can run the program by simply opening it. In order to allow this, the class contains a mode known as dynamic input, which can be set up through the function setupDynamicInput. For the rest of this segment, dynamic input will be referred to as IPI (in-program-input, its internal name)

The prompts which IPI provides to the user are always stored as function<strings>. This is so that prompts may change based on prior inputs. They can be passed as an array of c-strings, in which case that overload will use STR to wrap them all in function<strings>.

It is possible to a test sequence to run after each ordered input. This can be done to e.g. make sure that files exist as the user provides their names. Currently, only one test sequence can be run per ordered input. Tests will not be run again if they have already run unless specified otherwise, so that e.g. the user isn’t asked if it’s okay to overwrite a file twice.