Identifiers 2/ident

Purpose

To find the identifier names of functions and structures, and monitor in which sections of the program they are used; and so to police the accuracy of declarations at the head of each section.

Definitions

¶1. In some languages, we can detect function and/or structure definitions, and also the places where they are used: we use this to provide a little bit of syntax-colouring in the weaver, and to provide little footnotes to sections explaining cross-section function calls.

We also check the section's declared `@Interface`, if the web is set up with "Strict Usage Rules" to "On" – though the default is "Off".

¶2. Yet further section arrays are created here:

(S9) `$section_errors[]` will hopefully be the empty text, but otherwise accumulates any errors we find in the section's declared interface to other sections. (In languages or projects not using interfaces, this is always empty and has no effect.)

(S10) `$section_declared_a_service[]` is set to indicate that the section's `@Interface` says it provides functions to the entire web and therefore doesn't want to have to detail all its client sections. (Again, this has no effect in a web not using interface declarations.)

(S11) `$section_declared_structure_interface[]` is a concatenation of the structure ownership part of the `@Interface` for a section, with blank lines removed.

(S12) `$section_I6_template_identifiers[]` is colon-separated list of Inform 6 routines invoked from the template interpreter, and is always blank unless the language is C-for-Inform.

(S13) `$section_declared_used_by_block[]` is a block of used-by declarations from the section's `@Interface`, but rejigged into section list form (see below).

(S14) `$section_declared_uses_block[]` is the same, but for the "uses" part of the `@Interface`.

(S15) `$section_correct_uses_block[]` is what (S13) ought to be, based on the actual function calls observed in the web source.

(S16) `$section_correct_used_by_block[]` is what (S14) ought to be, based on the actual function calls observed in the web source.

(S17) `$section_correct_structures_block[]` is what (S11) ought to be, based on the actual structure usage observed in the web source.

¶3. In order to detect function definitions and structure elements, we need to know all of the types in use, and in particular we make a hash called `$base_types{}` whose keys are the base type names available to the code in the web: for instance, `int` and `char` are valid base types, and so, for convenience's sake, is `void`. These are only base types, so types resulting from qualification or pointer constructions, like `unsigned int` or `char *`, do not count.

¶4. As we go, we also accumulate data on functions, which are stored in hashes indexed by the C identifier of the function name:

(F1) `$functions_line{}` is the line number where the function declaration begins.
(F2) `$functions_declared_scope{}` is the declared scope, if any, of the function in "row of stars" notation, that is:
  (0) `""` means the scope is section-wide (which if we are checking strictly means the function may only be used in the current section);
  (1) `"*"` is illegal and never happens;
  (2) `"**"` means the scope is chapter-wide;
  (3) `"***"` means the scope is web-wide;
  (4) `"****"` is used only in C-for-Inform mode, and means the scope extends to allowing calls from the Inform template interpreter, i.e., under the control of data files outside the web;
  (5) `"*****"` means the function is `main()`, where execution begins.
(F3) `$functions_return_type{}` is the C return type of the function, e.g., `"void"` or `"unsigned char **"`;
(F4) `$functions_arglist{}` is the argument list in the declaration of the function, e.g., `"int x, int y"`;
(F5) `$functions_usage_count{}` is the number of times the function is referred to throughout the web (including in its definition, so this is always at least 1);
(F6) `$functions_usage_section_list{}` records which other sections call the function, if any do; the value is a section list concatenating string entries in the form `:C-L`, where `C` is the chapter number and `L` is the leafname of the section;
(F7) `$functions_usage_concisely_described{}` is essentially the same data, but in the more concise form of `:S` where `S` is the section number – so, for instance, `:4:13` means it is found in sections 4 and 13;
(F8) `$functions_usage_verbosely_described{}` is a verbose form of the same thing, used in comments in the tangled code (and thus unnecessary, really, but it was useful during the debugging of `inweb`).

¶5. Where exactly are functions used? We track this with `$function_usage_hash{}`, whose keys are not function names but are pairs in the form

```
function_name+Section Name.w
```

Thus the existence of a key means that the function with this name is used in the section with this name: note that `+` is not legal in a C identifier. Again, definition counts as usage, so every function is listed in this hash under its own section name.

¶6. We also want to track structures, their members, and the usage of these. For convenience of looping over all structures, a hash `$structures{}` is created with no useful value but such that its list of keys is exactly the set of structures. (The list of keys of, say, (T1) below would do just as well, but that would make the loops look a bit misleading.)

Structures then use the following hashes, whose keys are their typedef names:

(T1) `$structure_declaration_line{}` is the line number on which the `typedef` for this structure is made.
(T2) `$structure_declaration_end{}` is the final line number of its `typedef`, which is always $\geq$ (T1).
(T3) `$structure_CREATE_in_section{}`, which is used in C-for-Inform mode only, records the second number in which the `CREATE(whatever)` macro is used for the structure `whatever`. (In NI, each individual structure is allowed to be created in only one section – a self-imposed rule to improve the encapsulation of the code.)
(T4) `$structure_owner{}` is the section number of the section deemed to be the "owner" of this structure, a concept used to help `inweb` to enforce some encapsulation of code around related data.
(T5) `$structure_usage_section_list{}` is a list of leafnames of the sections which access this structure's members, – not those which merely use pointers to them, which doesn't break any encapsulation rules.
(T6) `$structure_ownership_summary{}` is similar, but in a more compressed form and using sigils; this helps the weaver to print neat footnotes under structure definitions.
(T7) `$structure_incorporates{}` is a list of structures incorporated into this one, that is, which are used as members of the current one. (Pointers do not count.)

(T8) `$structure_declaration_tangled{}` is a flag indicating that the structure's `typedef` block has now been tangled.

¶7. And members use the following hashes, whose keys are the member names:

(M1) The hash `$member_usage_section_list{}` is a section list of each distinct section using the member, which is indexed by its name. A section list is a convenient string representation of a set of sections: for instance,

```
"-Mushrooms and Toadstools.w:-Badgers.w:"
```

Each entry in the list begins with a dash and ends with a colon, and the latter character is not permitted in a section leafname, so this is an unambiguous representation.

(M2) `$member_structures{}` is a list of all structures using members with this name, with a ':' placed after each structure name: for instance, `"parse_node:tree_fern:"`.

¶8. The following global variable keeps track of which `typedef struct` definition we are currently scanning:

```
$current_struct = "";
```

---

§**1. More detailed parsing for C-like languages.** This will be another distinct phase within `inweb`: something which happens once, towards the middle of the run, though this time it happens only for projects written in a C-like language. But first we have to step back in time a little and look at something which happened during the Parser phase of `inweb`: the discovery of the types.

§**2.** The hash `$base_types{}` is used simply as a set of keys: we initialise it to the names of some built-in types, then add more from any `typedef struct` definitions we find in the web.

```
sub create_base_types_hash {
    my $i;
    $base_types{"char"} = 1;
    $base_types{"int"} = 1;
    $base_types{"float"} = 1;
    $base_types{"void"} = 1;
    $base_types{"FILE"} = 1;
    for ($i=0; $i<$no_lines; $i++) {
        if ($line_text[$i] =~ m/^typedef\s+struct\s+(\S+)\s+/) {
            $base_types{$1} = 1;
        }
    }
}
```

§**3. Parsing a section interface.** Most of what we want to do is to go through the web and look to see how functions and structures are being used, as we shall see. In a web with Strict Usage Rules set to "On", the results of that parsing have to exactly match the various declarations made in the sections. In such a web, each section has to contain an `@Interface:` block like the following: it is required to declare

(a) any functions it provides to the Inform 6 template interpreter (this is only used in C-for-Inform mode, of course);
(b) any other sections which call its functions (those it is "used by");
(c) any other sections whose functions it calls (those it "uses");
(d) any structures it creates, and whether they are accessed only in this section ("private") or from other sections (in which case it goes on to say exactly which ones).

```
-- Defines {-callv:index_sounds}
-- Used by Chapter 5/Literal Productions.w
-- Used by Chapter 10/Bibliographic Data.w
-- Used by Chapter 10/Figures.w
-- Uses Chapter 1/Platform-Specific Definitions.w
-- Uses Chapter 2/Sound Durations.w
-- Uses Chapter 10/Figures.w
-- Owns struct blorb_sound (private)
```

§**4.** The Parser section offers us the chance to grab and parse any such line of interface specification by calling this routine on each possible line. If we recognise the line as being an interface syntax, we return 1; if not, we return 0. That whole process will be complete *before* the call to `parse_C_like_sections`, so the hashes we create here will all be in existence in time to be used by `parse_C_like_sections` and its subroutines.

The scanner is used not only to spot parts of the `@Interface`, but also to notice any typedefs of structures which fly by in the code. In those cases we return 0, though, as we don't want to make the Parser categorise those lines as part of an interface.

```
sub scan_line_for_interface {
    my $i = $_[0];
    my $l = $line_text[$i];
    my $secname = $section_leafname[$line_sec[$i]];
    ⟨Try a structure definition part of the interface 5⟩;
    if ($web_language == $C_FOR_INFORM_LANGUAGE)
        ⟨Try an Inform template interpreter part of the interface 10⟩;
    ⟨Try a uses or used by part of the interface 6⟩;
    if ($current_struct ne "")
        ⟨Spot the declaration of any structure member in the current typedef 9⟩;
    ⟨Notice the beginning of a typedef structure definition 7⟩;
    ⟨Notice the end of a typedef structure definition 8⟩;
    return 0;
}
```

§**5.** We do little more than bottle this up for later:

⟨Try a structure definition part of the interface 5⟩ ≡
```
if ($l =~ m/^\-\-\s+Owns struct (\S+)\s+/) {
    $section_declared_structure_interface[$line_sec[$i]] .= $l."\n";
    $structure_declared_owner{$1} = $secname;
    return 1;
}
if ($l =~ m/^\s+\!\- shared with/) {
    $section_declared_structure_interface[$line_sec[$i]] .= $l."\n";
    return 1;
}
```
This code is used in §4.

§**6.**

⟨Try a uses or used by part of the interface 6⟩ ≡
```
if ($l =~ m/^\-\-\s+Used by Chapter (\d+)\/(.*?)\s*$/) {
    $section_declared_used_by_block[$line_sec[$i]] .= ':'.$1.'-'.$2;
    return 1;
}
if ($l =~ m/^\-\-\s+Uses Chapter (\d+)\/(.*?)\s*$/) {
    $section_declared_uses_block[$line_sec[$i]] .= ':'.$1.'-'.$2;
    return 1;
}
if ($l =~ m/^\-\-\s+Service\: used widely\s*$/) {
    $section_declared_a_service[$line_sec[$i]] = 1;
    return 1;
}
```
This code is used in §4.

§**7.** We exclude any structures whose names are made with the C preprocessor concatenation operator `##`, as anything with macros creating multiple structures is going to be waaaaay too hard for us to analyse. (The Memory section of `NI` does this.)

⟨Notice the beginning of a typedef structure definition 7⟩ ≡
```
if ($l =~ m/typedef\s+struct\s+(\S+)/) {
    $candidate = $1;
    if (not ($candidate =~ m/\#\#/)) {
        $structure_declaration_line{$candidate} = $i;
        $current_struct = $candidate;
        $structures{$candidate} = 1;
    }
}
```
This code is used in §4.

§**8.** The definition is required by us to end at its first close brace:

⟨Notice the end of a typedef structure definition 8⟩ ≡
```
if (($current_struct ne "") && ($l =~ m/^\}/)) {
    $structure_declaration_end{$current_struct} = $i;
    $current_struct = "";
}
```
This code is used in §4.

§**9.** When one structure contains another one – as opposed to merely a pointer to another one – we need to take note, as this will affect the order in which we have to tangle their definitions.

⟨Spot the declaration of any structure member in the current typedef 9⟩ ≡
```
if ($l =~ m/^\s*struct\s+([A-Za-z_][A-Za-z0-9_]*)\s+(\**)(.*?)\s*\;/) {
    The member is either another structure or a pointer to one
    if ($2 eq "") {                                    An actual incorporation of another structure
        $structure_incorporates{$current_struct} .= '->'.$1;
    }
    member_detected($current_struct, $1, $2, $3, 1);
} elsif ($l =~ m/^\s*([A-Za-z_][A-Za-z0-9_]*)\s+()\(\*(.*?)\)\(.*\)\;/) {
    The member is a function pointer
    member_detected($current_struct, $1, $2, $3, 0);
} elsif ($l =~ m/^\s*([A-Za-z_][A-Za-z0-9_]*)\s+(\**)(.*?)\s*\;/) {
    The member has a base type which is not a structure
    member_detected($current_struct, $1, $2, $3, 0);
}
```
This code is used in §4.

§**10.** For the format of Inform template commands, see the I6 template files. These declarations basically refer to Inform 6 identifier names used in compilation.

⟨Try an Inform template interpreter part of the interface 10⟩ ≡
```
if ($l =~ m/^\-\-\s+Defines \{-callv*\:(.*?)\}\s*$/) {
    $id = $1; $id =~ s/::/__/g;
    $section_I6_template_identifiers[$line_sec[$i]] .= $id.':';
    return 1;
}
if ($l =~ m/^\-\-\s+Defines \{-array\:(.*?)\}\s*$/) {
    $id = $1; $id =~ s/::/__/g;
    $section_I6_template_identifiers[$line_sec[$i]] .= $id.'_array:';
    return 1;
}
if ($l =~ m/^\-\-\s+Defines \{-routine\:(.*?)\}\s*$/) {
    $id = $1; $id =~ s/::/__/g;
    $section_I6_template_identifiers[$line_sec[$i]] .= $id.'_routine:';
    return 1;
}
```
This code is used in §4.

§**11.** When a new member is detected, we need to register it in a dictionary of all structure members used in the web. These are defined in C with lines like

```
int one, two[10];
```

which would lead to the function call

```
member_detected("what_have_you", "int", "", "one, two[10]", 0);
```

being made. We need to uncouple a list such as `"one, two[10]"` and remove the punctuation and array extents to obtain the actual member names present: in this case `one` and `two`.

```
sub member_detected {
    my $structure = $_[0];
    my $return_type = $_[1];
    my $return_type_pointer_stars = $_[2];
    my $member_name = $_[3];
    my $vouched_for = $_[4];
    ⟨Recursively uncouple the member list 12⟩;
    $member_name =~ s/\[.*$//;
    if (exists($blacklisted_members{$member_name})) { return; }
    if ((exists $base_types{$return_type}) || ($vouched_for == 1)) {
        $member_structures{$member_name} .= $structure.":";
        $member_types{$member_name} .= " ".$return_type.$return_type_pointer_stars;
    } else {
        inweb_error("member '".$member_name."' of structure '".$structure.
            "' has unknown type '".$return_type."'");
    }
}
```

§**12.**

⟨Recursively uncouple the member list 12⟩ ≡

```
    if ($member_name =~ m/^\s*(.*?)\,\s*(.*?)\s*$/) {
        my $left_chunk = $1;
        my $right_chunk = $2;
        member_detected($structure, $return_type, $return_type_pointer_stars,
            $left_chunk, $vouched_for);
        member_detected($structure, $return_type, $return_type_pointer_stars,
            $right_chunk, $vouched_for);
        return;
    }
```

This code is used in §11.

§13. The parsing-C-like-sections stage. We can now get on with the main narrative of this section: what happens when `inweb` parses a section of a C-like-language web to look at its functions, its structures, and (if Strict Usage Rules is "On") also its `@Interface` declarations. The check comes in two parts, with any errors reported in a unified way at the end:

```
sub parse_C_like_sections {
    my $i;
    for ($i=0; $i<$no_sections; $i++) { $section_errors[$no_sections] = ""; }
    ⟨Stage I: police functions and section usage 14⟩;
    ⟨Stage II: police structures and member usage 15⟩;
    report_any_interface_errors_found();
}
```

§14.

⟨Stage I: police functions and section usage 14⟩ ≡

```
    find_function_definitions();
    scan_identifiers_in_source();
    find_actual_section_usage();
    check_interface_declarations_for_uses_and_used_by();
```

This code is used in §13.

§15.

⟨Stage II: police structures and member usage 15⟩ ≡

```
    find_structure_ownership();
    establish_structure_usage();
    check_interface_declarations_for_structures();
    if ($web_language == $C_FOR_INFORM_LANGUAGE) { check_uniqueness_of_structure_members(); }
```

This code is used in §13.

§16. Recognising C function definitions. A function has to be declared inside code, or inside an `@d` definition:

```
sub find_function_definitions {
    my $i;
    for ($i=0; $i<$no_lines; $i++) {
        if (($line_category[$i] == $CODE_BODY_LCAT)
            || ($line_category[$i] == $BEGIN_DEFINITION_LCAT)
            || ($line_category[$i] == $CONT_DEFINITION_LCAT))
            ⟨Look for a function definition on this line 17⟩;
    }
}
```

§**17.** We recognise a C function as being a line which starts with an optional comment in the form `/**/`, `/***/`, `/****/` or `/*****/`, and then (or instead) takes the form `type identifier(args...`, where type is either one of the base types found already or else a pointer to a type.

⟨Look for a function definition on this line 17⟩ ≡

```
my $look_for_identifiers = $line_text[$i];
my $stars_in_comment = "";
my $type_qualifiers = "";
if ($look_for_identifiers =~ m/^\s*\/(\*+)\/\s*(.*)$/) {
    $stars_in_comment = $1; $look_for_identifiers = $2;
}
if ($look_for_identifiers =~ m/^\s*(signed|unsigned|long)\s+(.*)$/) {
    $type_qualifiers = $1; $look_for_identifiers = $2;
}
if ($look_for_identifiers =~
    m/^\s*([A-Za-z_][A-Za-z0-9_]*)\s+(\**)(([A-Za-z_][A-Za-z0-9_]*|\:\:)+)\s*\((.*)$/) {
    my $return_type = $1;
    my $return_type_pointer_stars = $2;
    my $fname = $3;
    my $arguments = $5;
    if ((not(exists $blacklisted_functions{$fname})) &&
        ($bibliographic_data{"Namespaces"} eq "On")) {
        if ($stars_in_comment ne "") {
            inweb_error("with Namespaces on, $fname should not be marked /*...*/");
        }
        if (($fname =~ m/::/) && ($stars_in_comment eq "")) {
            $stars_in_comment = "***";
        }
    }
    $fname =~ s/::/__/g;
    if ((exists $base_types{$return_type}) &&
        (not(exists $blacklisted_functions{$fname}))) {
        ⟨Deal with type qualifiers and constructors 18⟩;
        ⟨Accept this as a new function definition 19⟩;
    }
}
```

This code is used in §16.

§**18.** Combine the three parts of the return type:

⟨Deal with type qualifiers and constructors 18⟩ ≡

```
if ($return_type_pointer_stars ne "") {
    $return_type .= " ".$return_type_pointer_stars;
}
if ($type_qualifiers ne "") {
    $return_type = $type_qualifiers." ".$return_type;
}
```

This code is used in §17.

§**19.** In the following, we merge the text of up to the next 9 subsequent lines of code, in order to have enough of the function to be pretty sure that we've got the whole of its arguments: then we peel this off to obtain the argument list in the definition.

⟨Accept this as a new function definition 19⟩ ≡

```
$functions_line{$fname} = $i;
$functions_return_type{$fname} = $return_type;
$functions_declared_scope{$fname} = $stars_in_comment;

my $idash;
for ($idash = $i+1;
    (($idash<$no_lines) && ($idash-$i<10));
    $idash++) {
    $arguments .= ' '.$line_text[$idash];
}
if ($arguments =~ m/^(.*?)\)\s+\{/) {
    $functions_arglist{$fname} = $1;
} else {
    inweb_error("unable to find spec: $fname (args: '$arguments')");
}
```

This code is used in §17.

§**20. Scanning for function calls and accesses of structure members.** In other words, here we look for where things are actually used, rather than where they are declared or defined. We skip material in the appendices, as these might be configuration files likely to cause false positives.

```
sub scan_identifiers_in_source {
    my $i;
    for ($i=0; $i<$no_lines; $i++) {
        if (($line_category[$i] == $CODE_BODY_LCAT)
            || ($line_category[$i] == $BEGIN_DEFINITION_LCAT)
            || ($line_category[$i] == $CONT_DEFINITION_LCAT)) {
            my $current_section_leafname = $section_leafname[$line_sec[$i]];
            my $current_section_number = $line_sec[$i];
            my $look_for_identifiers = $line_text[$i];
            if ($section_sigil[$line_sec[$i]] =~ m/^[A-O]/) { next; }
            ⟨Remove any C comments from the line 21⟩;
            if ($web_language == $C_FOR_INFORM_LANGUAGE)
                ⟨Detect some of NI's fruitier macros 25⟩;
            ⟨Detect use of C's dot operator for structure member access 22⟩;
            ⟨Detect use of C's arrow operator for structure member access 23⟩;
            while ($look_for_identifiers =~ m/^.*?([A-Za-z_]([A-Za-z0-9_]|::)*)(.*?)$/) {
                $identifier = $1;
                $look_for_identifiers = $3;
                $identifier =~ s/::/__/g;
                ⟨Is this identifier a function whose name we recognise? 24⟩;
            }
        }
    }
}
```

§**21.** This would be fooled by e.g. `a="/*"; call_me(); b="*/";`, but we'll take the risk. None of what we do in this section is mission-critical in that its failure would cause the program to mis-compile: the worst case is that we don't always print the right footnotes in the woven form, which would be sad, but not the end of the world.

⟨Remove any C comments from the line 21⟩ ≡
```
$look_for_identifiers =~ s/\/\*(.*?)\*\///g;
```
This code is used in §20.

§**22.** For instance, spot `parse_node.word_ref1` as an access of member `word_ref1`.

⟨Detect use of C's dot operator for structure member access 22⟩ ≡
```
while ($look_for_identifiers =~ m/^(.*?)\.([A-Za-z_][A-Za-z0-9_]*)(.*?)$/) {
    $look_for_identifiers = $1.$3;
    note_usage_of_structure_member($2, $i);
}
```
This code is used in §20.

§**23.** And almost identically but for `->`, with pointers to structures:

⟨Detect use of C's arrow operator for structure member access 23⟩ ≡
```
while ($look_for_identifiers =~ m/^(.*?)\-\>([A-Za-z_][A-Za-z0-9_]*)(.*?)$/) {
    $look_for_identifiers = $1.$3;
    note_usage_of_structure_member($2, $i);
}
```
This code is used in §20.

§**24.** Since we know all the function names by this point...

define `$debug_identifier_detection 0`

⟨Is this identifier a function whose name we recognise? 24⟩ ≡
```
if (exists $functions_line{$identifier}) {
    if ($debug_identifier_detection == 1) { print STDERR "ID: ", $identifier, "\n"; }
    if ($identifier eq "main") { next; }                    which of course isn't called in the web
    my $section_defining_function =
        $section_leafname[$line_sec[$functions_line{$identifier}]];
    $functions_usage_count{$identifier}++;
    $function_usage_hash{$identifier.'+'.$section_chap[$line_sec[$i]]} ++;
    if ($current_section_leafname ne $section_defining_function) {
        $functions_usage_section_list{$identifier} .=
            ":".$section_chap[$line_sec[$i]]."-".$current_section_leafname;
        $functions_usage_concisely_described{$identifier} .=
            ":".$current_section_number;
        $functions_usage_verbosely_described{$identifier} .=
            language_comment("called by Chapter ".
                $section_chap[$line_sec[$i]]."/".$current_section_leafname.
                " line ".$line_source_file_line[$i]);
    }
    if ($debug_identifier_detection == 1) {
        print STDERR "Added fn to code area $section_chap[$line_sec[$i]]\n";
    }
}
```
This code is used in §20.

§**25.** For more on what these macros do, see the Inform source. `CREATE(T)` is a macro creating a structure of type `T`, which is defined with:

```
@d CREATE(type_name) ...
```

This definition shouldn't count as a creation of anything, but if `T` is anything else then it does. The sole point of this check is to warn us if we've accidentally created instances of the same structure in more than one section of Inform's web.

Macros such as `ALLOW_ARRAY(scene_data)` are used in Inform to set up what is legal when reading I6 template code: for instance, this would enable

```
{-array:scene_data}
```

the effect being that Inform's `compile_scene_data_array` function would be called whenever this token in an I6 template was expanded. (We have to take note of that here since the use of a macro disguises that a function call is being made.)

⟨Detect some of NI's fruitier macros 25⟩ ≡

```
    if ($look_for_identifiers =~ m/CREATE\((.*?)\)/) {
        if ($1 ne "type_name") {
            if ((exists ($structure_CREATE_in_section{$1})) &&
                ($structure_CREATE_in_section{$1} ne $section_leafname[$line_sec[$i]])) {
                add_error_to_section($current_section_number,
                    "Use of CREATE($1) in multiple sections");
            }
            $structure_CREATE_in_section{$1} = $section_leafname[$line_sec[$i]];
        }
    }
    while ($look_for_identifiers =~ m/^(.*?)ALLOW_([A-Z]+)\(([A-Za-z_0-9:_]*)\)(.*)$/) {
        $identifier = $3;
        $metalanguage_type = $2;
        $look_for_identifiers = $1.$4;
        $identifier =~ s/::/__/g;
        if (($metalanguage_type eq "CALL") ||
            ($metalanguage_type eq "CALLV")) {
            $dot_i6_identifiers{$identifier} = 1;
        }
        if ($metalanguage_type eq "ARRAY") {
            $dot_i6_identifiers{$identifier."_array"} = 1;
        }
        if ($metalanguage_type eq "ROUTINE") {
            $dot_i6_identifiers{$identifier."_routine"} = 1;
        }
    }
```

This code is used in §20.

§**26.** Keep track of each distinct section in which a given structure member is used:

```
sub note_usage_of_structure_member {
    my $member_name = $_[0];
    my $line_number = $_[1];
    my $found_in = "-".$section_leafname[$line_sec[$line_number]].":";
    if ($member_usage_section_list{$member_name} =~ m/$found_in/) { return; }
    $member_usage_section_list{$member_name} .= $found_in;
}
```

§**27. Determine which sections call which other sections.** We now know all of the functions, and where all of the function calls occur. This means we can determine (i) the actual scope of a function and (ii) how the "used by" and "uses" parts of a section's `@Interface` ought to read, if it were going to have them.

```
sub find_actual_section_usage {
    my $sn;
    for ($sn=0; $sn<$no_sections; $sn++) {
        my %usages = ();
        ⟨Check that the section's declared I6 template interface is correct 32⟩;
        foreach $f (sort keys %functions_line) {
            if ($section_leafname[$line_sec[$functions_line{$f}]] eq $section_leafname[$sn]) {
                ⟨Check that this function is properly declared, and add sections calling it to the usages hash 28⟩;
            }
        }
        if ($section_declared_a_service[$sn] == 1) { %usages = (); }
        ⟨Translate the usages hash into an unsorted form of the correct used-by block 29⟩;
    }
    for ($sn=0; $sn<$no_sections; $sn++)
        ⟨Deduce the correct uses block from the collection of all correct used-by blocks 30⟩;
    for ($sn=0; $sn<$no_sections; $sn++)
        ⟨Tidy up and finalise the correct uses and used-by blocks 31⟩;
}
```

§**28.** A neat trick here is that we knock the function's own chapter out of its usage list with a search and replace, and then look for chapter markers in what remains: if there are any, then the usage list must have contained sections from other chapters.

⟨Check that this function is properly declared, and add sections calling it to the usages hash 28⟩ ≡

```
    my $owning_chapter = $section_chap[$line_sec[$functions_line{$f}]];
    my $cp = $functions_usage_section_list{$f};
    $cp =~ s/\:$owning_chapter\-//g;
    my $actual_scope = "";
    if ($functions_usage_section_list{$f} ne "") { $actual_scope = "**"; }
    if ($f eq "main") { $actual_scope = "*****"; }
    elsif (exists $dot_i6_identifiers{$f}) {
        $actual_scope = "****";
        check_section_declares_template_I6($sn, $f);
    } elsif ($cp =~ m/\:\d+\-/) { $actual_scope = "***"; }
    check_function_declared_correct_scope($sn, $f, $actual_scope);
    $cp = $functions_usage_section_list{$f};
    while ($cp =~ m/^(\:\d+\-[^:]*)(.*)$/) {
        $usages{$1}++;
        $cp = $2;
    }
```

This code is used in §27.

§**29.** The usages hash is really just used as a set of keys here, and we concatenate it into a single string. (The list is unsorted but at least can't contain duplicates.) We tack a colon onto the end for convenience of the next search, but after that we'll remove it again.

⟨Translate the usages hash into an unsorted form of the correct used-by block 29⟩ ≡

```
$section_correct_used_by_block[$sn] = "";
foreach $u (sort keys %usages) {
    $section_correct_used_by_block[$sn] .= $u;
}
$section_correct_used_by_block[$sn] .= ":";
```

This code is used in §27.

§**30.** The uses blocks form a sort of transpose of the used-by blocks, if we think of these data structures as matrices rather than arrays: and as with all matrix-esque calculations, we end up with quadratic running time in the dimensions of the matrix. That is, if there are s sections, then the following takes $O(s^2)$ time, but since in practice $s < 200$ this is quite bearable.

⟨Deduce the correct uses block from the collection of all correct used-by blocks 30⟩ ≡

```
$section_correct_uses_block[$sn] = "";
my $j;
for ($j=0; $j<$no_sections; $j++) {
    if ($section_correct_used_by_block[$j] =~ m/\-$section_leafname[$sn]\:/) {
        if ($section_declared_a_service[$j] != 1) {
            $section_correct_uses_block[$sn]
                .= ":".$section_chap[$j]."-".$section_leafname[$j];
        }
    }
}
```

This code is used in §27.

§**31.** Note the promised removal of the terminating colons in the used-by blocks:

⟨Tidy up and finalise the correct uses and used-by blocks 31⟩ ≡

```
$section_correct_used_by_block[$sn] =~ s/\:$//;
$section_correct_used_by_block[$sn] =
    section_list_sort($section_correct_used_by_block[$sn]);
$section_correct_uses_block[$sn] =
    section_list_sort($section_correct_uses_block[$sn]);
```

This code is used in §27.

§**32.** If a section in a C-for-Inform web which uses an `@Interface` is going to name any I6 template things, well, they'd better be right: even in Weak mode we're not going to forgive any error in this.

⟨Check that the section's declared I6 template interface is correct 32⟩ ≡

```
my $a = $section_I6_template_identifiers[$sn];
while ($a =~ m/^(.*?)\:(.*)$/) {
    $a = $2;
    if (not(exists $dot_i6_identifiers{$1})) {
        add_error_to_section($sn, "Bad scope: $1 isn't a {-} command as claimed");
    }
    if ($section_leafname[$line_sec[$functions_line{$1}]] ne $section_leafname[$sn]) {
        add_error_to_section($sn, "Bad scope: $1 isn't defined in this section");
    }
}
```

This code is used in §27.

§**33. Structures.** We now come to the second part of the checking code for C-like sections: the part which verifies that structures and their members are used "correctly". We don't come to the party with nothing in hand, though: we have already found all the `typedef` structure declarations, so we know what the names of the structures are and on what line they are created; and we have kept an eye out, if in C-for-Inform mode, for any use of the `CREATE(S)` memory management macro as applied to a structure `S`.

§**34.** First: which section rightfully "owns" a structure? First, anyone with the power to `CREATE` has dominion; if nobody does, then anyone who claims the right in his `@Interface` is allowed to have it; and if nobody even claims the structure, well then, it belongs to whichever section declared its `typedef`.

```
sub find_structure_ownership {
    my $struc;
    foreach $struc (keys %structures) {
        if (exists $structure_CREATE_in_section{$struc}) {
            $structure_owner{$struc} = $structure_CREATE_in_section{$struc};
        } elsif (exists $structure_declared_owner{$struc}) {
            $structure_owner{$struc} =
                $structure_declared_owner{$struc};
        } else {
            $structure_owner{$struc} =
                $section_leafname[$line_sec[$structure_declaration_line{$struc}]];
        }
    }
}
```

§**35.** Second: which sections use which members, and by extension, which structures? Are any members entirely unused and consequently redundant?

```
sub establish_structure_usage {
    my $member;
    ⟨Check that any -analyse-structure command line setting wasn't mistyped 36⟩;
    foreach $member (sort keys %member_structures) {
        if (exists $member_usage_section_list{$member}) {
            my $msul = $member_usage_section_list{$member};
            while ($msul =~ m/^\-(.*?)\:(.*)$/) {
                my $observed_in = $1; $msul = $2;
                my $owners = $member_structures{$member};
                while ($owners =~ m/^(.*?)\:(.*)$/) {
                    my $owner = $1; $owners = $2;
                    my $susl_chunk = "-".$observed_in.":";
                    ⟨In -analyse-structure mode, throw an error for each member used in a foreign section 37⟩;
                    if (not ($structure_usage_section_list{$owner} =~ m/$susl_chunk/)) {
                        $structure_usage_section_list{$owner} .= $susl_chunk;
                    }
                }
            }
        } else {
            member_declared_but_not_used($member);
        }
    }
    ⟨Compose the correct structure declaration to appear in the Interface of each section 38⟩;
}
```

§**36.** As crude as this tool is, it was surprisingly helpful during a consolidation period when the code of NI was being tidied up into a more encapsulated form.

⟨Check that any -analyse-structure command line setting wasn't mistyped 36⟩ ≡

```
if ($analyse_structure_setting ne "") {
    if (not(exists($structures{$analyse_structure_setting}))) {
        inweb_fatal_error("no such structure: ".$analyse_structure_setting);
    }
    print "The structure $analyse_structure_setting is owned by ",
        $structure_owner{$analyse_structure_setting}, "\n";
    print "Members of the structure used from other sections are as follows:\n";
}
```

This code is used in §35.

§**37.** Perhaps these aren't really errors, but it does make for a tidy end to the process after we're done.

⟨In -analyse-structure mode, throw an error for each member used in a foreign section 37⟩ ≡

```
if (($analyse_structure_setting ne "") &&
    ($analyse_structure_setting eq $owner)) {
    if ($observed_in ne $structure_owner{$owner}) {
        inweb_error($member.": Chapter ".
            $section_chap[$section_number_from_leafname{$observed_in}].
            "/".$observed_in);
    }
}
```

This code is used in §35.

§**38.** The following assumes that structures are not used in the Appendices, which of course is true since these do not contain code.

⟨Compose the correct structure declaration to appear in the Interface of each section 38⟩ ≡

```
my $sec_number;
for ($sec_number=0; $sec_number<$no_sections; $sec_number++) {
    my $declaration = "";
    my $struc;
    foreach $struc (sort keys %structures) {
        if ($structure_owner{$struc} eq $section_leafname[$sec_number]) {
            my $structure_billing = "";
            ⟨Work out the billing for this structure 39⟩;
            $declaration .= $structure_billing;
        }
    }
    $section_correct_structures_block[$sec_number] = $declaration;
}
```

This code is used in §35.

§**39.** What we declare for the structure is: whether private or public, that is, open for other sections to access its members; whether it is `typedef`d by a section other than its rightful owner (which is allowed but deprecated, in terms of `NI`); and if it is public, then which other sections have action.

⟨Work out the billing for this structure 39⟩ ≡

```
my %usage_list = ();
my %shorter_usage_list = ();
my $structure_used_externally = 0;
my $deviant_ownership_note = "";
if ($section_leafname[$line_sec[$structure_declaration_line{$struc}]]
    ne $structure_owner{$struc}) {
    $deviant_ownership_note =
        ": typedef in $section_leafname[$line_sec[$structure_declaration_line{$struc}]]";
}
⟨Fill the usage list hash with the names of external sections accessing structure 40⟩;
if ($structure_used_externally == 0) {
    $structure_billing = "-- Owns struct $struc (private$deviant_ownership_note)\n";
    $private_structures++;
} else {
    $structure_billing = "-- Owns struct $struc (public$deviant_ownership_note)\n";
    $shared_structures++;
    foreach $k (sort keys %usage_list) {
        $structure_billing .= $usage_list{$k};
        $structure_ownership_summary{$struc} .= $shorter_usage_list{$k} . " ";
    }
}
```

This code is used in §38.

§**40.**

⟨Fill the usage list hash with the names of external sections accessing structure 40⟩ ≡

```
my $susl = $structure_usage_section_list{$struc};
while ($susl =~ m/^\-(.*?)\:(.*)$/) {
    my $client_section = $1; $susl = $2;
    if ($client_section ne $section_leafname[$sec_number]) {
        $structure_sharings++;
        $structure_used_externally = 1;
        $usage_list{sprintf("%05d", $section_number_from_leafname{$client_section})}
            .= "   !- shared with Chapter ".
            $section_chap[$section_number_from_leafname{$client_section}].
            "/".$client_section."\n";
        $shorter_usage_list{sprintf("%05d", $section_number_from_leafname{$client_section})}
            .= $section_sigil[$section_number_from_leafname{$client_section}];
    }
}
```

This code is used in §39.

§**41. Hacking with section list strings.** We take a section list string to pieces, sort it and reassemble it in sorted order. Don't look so suspicious: it works.

```
sub section_list_sort {
    my $list = $_[0];
    my %lines_unpacked = ();
    my $this_line;
    my $account = "";
    while ($list =~ m/^\:(\d+)\-([^:]*)(.*)$/) {
        $this_line = $prefix. "Chapter ". $1. "/". $2. "\n";
        $lines_unpacked{sprintf("%04d", $section_number_from_leafname{$2})} = ":".$1."-".$2;
        $list = $3;
    }
    foreach $this_line (sort keys %lines_unpacked) {
        $account .= $lines_unpacked{$this_line};
    }
    return $account;
}
```

§**42.** To unpack a section list is to reconstruct it as a sequence of lines in sorted order and with a slightly different format.

```
sub section_list_unpack {
    my $prefix = $_[0];
    my $list = $_[1];
    my %lines_unpacked = ();
    my $this_line;
    my $account = "";
    while ($list =~ m/^\:(\d+)\-([^:]*)(.*)$/) {
        $this_line = $prefix. "Chapter ". $1. "/". $2. "\n";
        $lines_unpacked{sprintf("%04d", $section_number_from_leafname{$2})} = $this_line;
        $list = $3;
    }
    foreach $this_line (sort keys %lines_unpacked) {
        $account .= $lines_unpacked{$this_line};
    }
    return $account;
}
```

§43. Applying the Strict Usage Rules of a vengeful god.

```
sub check_interface_declarations_for_uses_and_used_by {
    my $sn;
    if ($bibliographic_data{"Strict Usage Rules"} eq "Off") { return; }
    if ($bibliographic_data{"Declare Section Usage"} eq "Off") { return; }
    for ($sn=0; $sn<$no_sections; $sn++) {
        if ($section_declared_used_by_block[$sn] ne
            $section_correct_used_by_block[$sn]) {
            $em = "Interface error:\n"
                .section_list_unpack(
                    "%    Says used by ", $section_declared_used_by_block[$sn])
                .section_list_unpack(
                    "-- Used by ", $section_correct_used_by_block[$sn]);
            add_error_to_section($sn, $em);
        }
        if ($section_declared_uses_block[$sn] ne
            $section_correct_uses_block[$sn]) {
            $em = "Interface error:\n"
                .section_list_unpack(
                    "%    Says uses ", $section_declared_uses_block[$sn])
                .section_list_unpack(
                    "-- Uses ", $section_correct_uses_block[$sn]);
            add_error_to_section($sn, $em);
        }
    }
}
```

§**44.** And similarly for the structures part of the `@Interface:...`

```
sub check_interface_declarations_for_structures {
    my $sn;
    for ($sn=0; $sn<$no_sections; $sn++) {
        if ($section_declared_structure_interface[$sn] ne
            $section_correct_structures_block[$sn]) {
            add_error_to_section($sn,
                "Incorrect structure billing: should be...\n".
                $section_correct_structures_block[$sn].
                "...and not...\n".
                $section_declared_structure_interface[$sn]);
        }
    }
}
```

§**45.** If we're being Strict, functions must also be marked with their correct scopes:

```
sub check_function_declared_correct_scope {
    my $sn = $_[0];
    my $f = $_[1];
    my $actual_scope = $_[2];
    if ($bibliographic_data{"Namespaces"} eq "On") {
        if ($f ne "main") {
            if (($actual_scope ne "") && ($functions_declared_scope{$f} eq "")) {
                add_error_to_section($sn,
                    "Begin externally called, function $f should belong to a :: namespace");
                return;
            }
            if (($actual_scope eq "") && ($functions_declared_scope{$f} ne "")) {
                add_error_to_section($sn,
                    "Begin internally called, function $f must not belong to a :: namespace");
                return;
            }
        }
        $functions_declared_scope{$f} = $actual_scope;
        return;
    }
    if ($bibliographic_data{"Strict Usage Rules"} eq "Off") { return; }
    if ($actual_scope ne $functions_declared_scope{$f}) {
        add_error_to_section($sn,
            "Bad scope: $f should be /".$actual_scope."/ not /".
                $functions_declared_scope{$f}."/");
    }
    if ($f =~ m/^([A-Z].*__)(.*?)$/) {
        my $declared = $1;
        my $within = $section_namespace[$sn];
        $within =~ s/::/__/g;
        if ($within eq "") {
            $fcc = restore_quadpoint($f);
            add_error_to_section($sn,
                "Bad scope: $fcc declared outside of any namespace");
        } elsif (not ($declared =~ m/^$within/)) {
            $fcc = restore_quadpoint($f); $wcc = restore_quadpoint($within);
            add_error_to_section($sn,
                "Bad scope: $fcc not allowed inside the section's namespace $wcc");
        }
    }
}
sub restore_quadpoint {
    my $f = $_[0];
    $f =~ s/__/::/g;
    return $f;
}
```

§**46.** The following is applied only in C-for-Inform mode; in ordinary C, no such functions occur.

```
sub check_section_declares_template_I6 {
    my $sn = $_[0];
    my $f = $_[1];
    if ($bibliographic_data{"Strict Usage Rules"} eq "Off") { return; }
    f = s/::/g;
    if (not ($section_I6_template_identifiers[$sn] =~ m/$f/)) {
        add_error_to_section($sn,
            "Bad scope: $f isn't declared:\n-- Defines {-callv:".$f."}\n");
    }
}
```

§**47.** It is, of course, a Sin to declare structures with member names that are never actually used. Unless of course we are in a messy program which uses them in concealed ways because of macro expansion – hence the exemption clause.

```
sub member_declared_but_not_used {
    my $member = $_[0];
    if ($bibliographic_data{"Strict Usage Rules"} eq "Off") { return; }
    if (exists($members_allowed_to_be_unused{$member})) { return; }
    inweb_error("structure(s) '".$member_structures{$member}."' has or have member '".
        $member."' declared but not used");
}
```

§**48.** `inweb` prefers a tidy setup where the same member name is not used in multiple structures, and in strict mode, it insists of this: that is, where `p->drivel` identifies the type of `p` since `drivel` is a member in only one possible structure. The following routine polices that rule.

```
sub check_uniqueness_of_structure_members {
    my $member;
    if ($bibliographic_data{"Strict Usage Rules"} eq "Off") { return; }
    foreach $member (sort keys %member_structures) {
        my $owner_count = 0;
        my $x = $member_structures{$member};
        while ($x =~ m/^(.*?)\:(.*)$/) { $x = $2; $owner_count++; }
        if ($owner_count > 1) {
            inweb_error("element '".$member."' belongs to multiple structures: ".
                $member_structures{$member});
        }
    }
}
```

§**49. Interface errors.** When an interface error comes to light, we call:

```
sub add_error_to_section {
    my $section_number = $_[0];
    my $error_text = $_[1];
    if ($error_text eq "") { print "*** Bad error text ***\n"; exit(1); }
    $section_errors[$section_number] .= $error_text."\n";
}
```

§**50.** So that at the end of the checking process, we can report and take action:

```
sub report_any_interface_errors_found {
    my $infractions = 0;
    my $sn;
    for ($sn=0; $sn<$no_sections; $sn++) {
        if ($section_errors[$sn] ne "") {
            inweb_error("interface errors on $section_sigil[$sn] ($section_leafname[$sn])");
            print STDERR $section_errors[$sn];
            $infractions++;
        }
    }
    if ($infractions > 0) { inweb_fatal_error("halting because of section errors"); }
}
```