Reading Types in C Using the Right Left Walk Method

Jim Canning
Department of Computer Science
University of Massachusetts Lowell
Lowell, Massachusetts 01854 USA
canning@cs.uml.edu

Ali Rafieymehr
Dept of Mathematics and Computer Science
Western New England College
Springfield, Massachusetts 01119-2684 USA
arafieym@wnec.edu

William Moloney
Department of Computer Science
University of Massachusetts Lowell
Lowell, Massachusetts 01854 USA
moloney@cs.uml.edu

Demetrio Rey
Department of Electrical Engineering
University of Carabobo
Valencia, Venezuela
drey@cs.uml.edu

Abstract
We describe a technique for beginning students to read and verbalize a declarator in the C programming language. Once learned, students will be able to easily state the type of an object without hesitation. It is suggested that this technique be taught to students during the first week of an introductory C programming course. Getting students to correctly read and verbalize types hastens their learning of the language and supports necessary discussion of a program's translation and execution environments.

Keywords: Object type, verbalizing a declarator, verbalizing C types, C declarators

1. Introduction

One reason why beginning C programmers have difficulty is that they do not understand C declarators very well. C declarators are either <u>declarations</u> that reserve no space but convey type information or they are <u>definitions</u> that reserve space and convey type information. One of the first steps to understanding a declarator is to know how to read it and verbalize it. This paper outlines a technique by which types in C can be read with ease, clarity, consistency, and conviction. The technique, which we call the *Right Left Walk Method*, is useful when teaching the C programming language. This method is easy to learn and once mastered, it will be difficult to stump students – no matter how complicated the declarator may be.

2. Right Left Walk Method (RLWM)

We consider a declarator in the C programming language to be a collection of delimiter and non-delimiter tokens. The Right Left Walk Method (RLWM) is a strategy for visually sweeping over a valid declarator, processing tokens with the intent of verbalizing the type of an object under consideration. The idea is to find the object of interest (usually an identifier) and then sweep back and forth over the declarator, encountering unmarked tokens while speaking and reversing direction as appropriate. The technique requires that the initial sweep move to the right. As tokens are encountered, the reader either remains silent or verbalizes an exact phrase that corresponds to the token. Tokens that are delimiters force the reader to remain silent and reverse direction. Tokens that are not delimiters cause the reader to maintain direction and usually verbalize a phrase. Any token that has been swept over is marked as processed and it will not be re-processed. The technique is completed when all tokens have been processed.

Table 1 provides a list of ten tokens that one may encounter in a declarator. The first three tokens listed: the semi-colon, the grouping right parenthesis and the grouping left parenthesis, are the only delimiter tokens. When any of these three delimiter tokens are encountered during a visual sweep of the declarator, nothing is spoken and the direction of the sweep is reversed.

The remaining tokens: identifier, 'int', '*', function parentheses, '[' and ']' are not delimiters. The table provides the exact phrase to speak when we encounter these tokens during the sweep. For example, the declarator

```
    int   *y ;
```

has four tokens: the delimiter token ";" and the three non-delimiter tokens 'int', '*' and 'y'. Using the information found in the Table 1, the RLWM can now be applied to answer the question: "What is y's type?"

Answer: <u>y is</u> <u>a pointer to</u> <u>an integer</u>

Table 1: A list of tokens found in typical C declarators

Token	**Delimiter?**	**Exact phrase to speak**	Reverse Direction?
;	Yes		Yes
grouping)	Yes		Yes
grouping (	Yes		Yes
Identifier	No	identifier is	No
Int	No	an integer	No
*	No	a pointer to	No
Function (	No	a function	No
Function)	No	that returns	No
[*n*	No	an array of *n*	No
]	No		No

This is a four-step sweep. In step 1, the reader would find the object of interest and speak, "y is". In doing so, the token y would be marked as processed. In step 2, the reader would visually sweep right and then encounter the next un-marked token, the semi-colon. Since the semi-colon is a delimiter, the speaker would say nothing and reverse direction to the left. Moving left the speaker again encounters the identifier 'y', but does nothing with it since it has already been processed. Continuing on to step 3, the non-delimiter token '*' is encountered forcing the reader to speak the exact phrase "a pointer to". Finally, in step 4, movement continues to the left where the token 'int' is swept over causing the phrase "an integer" to be stated. Since all four tokens have been processed, the application of the method is complete.

Table 2 provides five additional examples. The second and fifth examples illustrate the use of grouping right and left parentheses. In the second example, although y's type is identical to our original example, a proper visual sweep would cause the reader to move right then left then right then left. That is, after 'y' is identified, the sweep would first encounter the grouping right parenthesis, and then reverse direction subsequently sweeping over the marked token 'y' again and then sweeping over the token '*' causing the phrase "a pointer to" to be verbalized. Then sweeping into the grouping left parenthesis would cause yet another change in direction and processing would continue past the marked tokens '*', 'y' and ')'. Upon reaching the far right side, the delimiter token ';' would be processed forcing the sweep to move left towards the token 'int' causing the phrase "an integer" to be verbalized. To simplify discussion, in examples 3, 4, and 5, the types and number of the function arguments have been ignored.

However, the RLWM can be applied to each argument individually. Example five contains both grouping parentheses and function parentheses. When asked "What is test's type?", a student using the RLWM would first utter "test is" and begin to sweep right, immediately encountering the grouping right parenthesis. This delimiter would cause a reverse in direction and then force the speaker to say "a pointer to" and continue past the '*' onto the grouping left parenthesis. A reverse of direction would ensue and then the function left parenthesis would be encountered yielding the phrase "a function". The function right parenthesis would not cause a change in direction and it would add the phrase "that returns" to the verbalization. Ultimately, the semi-colon would cause a reverse in direction and the final words "an integer" would complete the process. In total, the verbalization would be:

"test is a pointer to a function that returns an integer"

As an illustration of the utility of the method, beginning programmers would have difficulty reading the declarator of the UNIX's signal system call, shown here:

```
void ( *signal ( int sig,
        void (*function) (int) ) ) (int);
```

However, using the visual sweeping technique of the RLWM the answer to the question "What is signal's type?" is readily verbalized as:

signal is a function that returns a pointer to
a function that returns a void

Table 2: Five examples of C declarators with their corresponding RLWM phrasing

Example	Simple declarators	How They Should be Read Using the RLWM
1	int ** y ;	y is a pointer to a pointer to an integer
2	int (*y) ;	y is a pointer to an integer
3	int y (int x) ;	y is a function that returns an integer
4	int *test (int x) ;	test is a function that returns a pointer to an integer
5	int (*test)(int x);	test is a pointer to a function that returns an integer

3. Reading Arrays

The tokens '[' and ']' are special non-delimiter tokens. The '[' is special because it is a pluralizer for the reading of subsequent tokens. The ']' is special because it is a silent non-delimiter token. Table 3 contains the pluralized version that replaces the singular phrasing after a '[' has been processed. For example, the reading of the declarator "int x[3];"

x is   an array of 3   integers

Table 3: Pluralization of token phrasing after a "[" is encountered.

Token	Singular Phrase	Pluralized Phrase
int	an integer	integers
*	a pointer to	pointers to
function (	a function	functions
function)	That returns	that return
'['	an array of	arrays

Table 4 provides four examples of how the RLWM would force the reading of a declarator containing an array. Most beginning students, even after reading the appropriate chapter on arrays in their textbook would be unable to verbalize the type of the object 'y' with clarity, accuracy, and confidence. The good news is that after a thirty-minute lesson covering the RLWM, most beginning students will be forever able to read types.

Table 4: Examples of how the RLWM would read types that incorporate arrays.

Example	Declarator	Verbalization
1	int *y[6] ;	y is an array of 6 pointers to integers
2	int y[3][2] ;	y is array of 3 arrays of 2 integers
3	int (*y)[4] ;	y is a pointer to an array of 4 integers
4	int *(*y[2])(int x);	y is an array of two pointers to functions that return pointers to integers

4. Diagnosing Mismatched Types

Sometimes the object of interest is not just an identifier, but a larger construct within the declarator. For example, consider the following declarator:

```
int  ***p ;
```

The RLWM method also enables the reader to answer the question "What is *p's type?" This is done by applying the technique to *p. That is,

*p is   a pointer to   a pointer to   an integer

Since the rightmost * is part of the object's name it is immediately marked as being processed. Similarly,

**p is   a pointer to   an integer

***p is   an integer

Knowing how to verbalize a declarator and clearly state an object's type is valuable to students who are trying to debug code that has unintended mismatched types. To illustrate this, consider the following three declarators and the associated assignment statements found in Table 5.

```
int    x ;
int   *y ;
int  **p ;
```

Table 5: Knowing how to read types can be useful when diagnosing mismatched type errors

Statement	Left Hand Side Type	Right Hand Side Type	Mismatched Types?
y = p	y is a pointer to an integer	p is a pointer to a pointer to an integer	Yes
*p = *y	*p is a pointer to an integer	*y is an integer	Yes
**p = x	**p is an integer	x is an integer	No

5. Final Thoughts

Learning to read, understand, and communicate C's type declarators confidently is a necessary pre-condition to learning the language. However, the vast majority of introductory textbooks used do not emphasize the importance of reading types, and even if they do, it is not done early enough in the semester. We believe that a lecture describing the Right Left Walk Method should be given in the first week of classes, replete with supporting pen and pencil homework assignments. Only when students learn what declarators actually do will they master the mechanics of the language. Reading declarators correctly is the first step towards this end. Subsequent steps involve knowing how much space a definition declarator reserves and where this space is located.