How to interpret complex C/C++ declarations

Vikram A Punathambekar

3 Jul 2004 CPOL

Rate me: 4.73/5 (162 votes)

Ever came across a declaration like int * (* (*fp1) (int)) [10]; or something similar that you couldn't fathom? This article will teach you to interpret such complex C/C++ declarations, including the use of typedef, const, and function pointers.

Contents

Introduction

Ever came across a declaration like `int * (* (*fp1) (int) ) [10];` or something similar that you couldn't fathom? This article will teach you to interpret C/C++ declarations, starting from mundane ones (*please* bear with me here) and moving on to very complex ones. We shall see examples of declarations that we come across in everyday life, then move on to the troublesome `const` modifier and `typedef`, conquer function pointers, and finally see the right-left rule, which will allow you to interpret any C/C++ declaration accurately. I would like to emphasize that it is not considered good practice to write messy code like this; I'm merely teaching you how to understand such declarations. **Note**: This article is best viewed with a minimum resolution of 1024x768, in order to ensure the comments don't run off into the next line.

[Back to contents]

The basics

Let me start with a very simple example. Consider the declaration:

Hide Copy Code

```
int n;
```

This should be interpreted as "declare `n` as an `int`".

Coming to the declaration of a pointer variable, it would be declared as something like:

Hide Copy Code

```
int *p;
```

This is to be interpreted as "declare p as an int * i.e., as a pointer to an int". I'll need to make a small note here - it is always better to write a pointer (or reference) declaration with the * (or &) preceding the variable rather than following the base type. This is to ensure there are no slip-ups when making declarations like:

Hide Copy Code

```
int* p,q;
```

At first sight, it looks like p and q have been declared to be of type int *, but actually, it is only p that is a pointer, q is a simple int.

We can have a pointer to a pointer, which can be declared as:

Hide Copy Code

```
char **argv;
```

In principle, there is no limit to this, which means you can have a pointer to a pointer to a pointer to a pointer to a float, and so on.

Consider the declarations:

Hide Copy Code

```
int RollNum[30][4];
int (*p)[4]=RollNum;
int *q[5];
```

Here, p is declared as a pointer to an array of 4 ints, while q is declared as an array of 5 pointers to integers.

We can have a mixed bag of *s and &s in a single declaration, as explained below:

Hide Copy Code

```
int **p1;  //  p1 is a pointer   to a pointer   to an int.
int *&p2;  //  p2 is a reference to a pointer   to an int.
int &*p3;  //  ERROR: Pointer    to a reference is illegal.
int &&p4;  //  ERROR: Reference  to a reference is illegal.
```

[Back to contents]

The const modifier

The const keyword is used when you want to prevent a variable (oops, that's an oxymoron) from being modified. When you declare a const variable, you need to initialize it, because you can't give it a value at any other time.

Hide Copy Code

```
const int n=5;
int const m=10;
```

The two variables n and m above are both of the same type - constant integers. This is because the C++ standard states that the const keyword can be placed before the type or the variable name. Personally, I prefer using the former style, since it makes the const modifier stand out more clearly.

const is a bit more confusing when it comes to dealing with pointers. For instance, consider the two variables p and q in the declaration below:

Hide Copy Code

```
const int *p;
```

```
int const *q;
```

Which of them is a pointer to a `const int`, and which is a `const` pointer to an `int`? Actually, they're both pointers to `const int`s. A `const` pointer to an `int` would be declared as:

Hide Copy Code

```
int * const r= &n; // n has been declared as an int
```

Here, `p` and `q` are pointers to a `const int`, which means that you can't change the value of `*p`. `r` is a `const` pointer, which means that once declared as above, an assignment like `r=&m;` would be illegal (where `m` is another `int`) but the value of `*r` can be changed.

To combine these two declarations to declare a `const` pointer to a `const int`, you would have to declare it as:

Hide Copy Code

```
const int * const p=&n // n has been declared as const int
```

The following declarations should clear up any doubts over how `const` is to be interpreted. Please note that some of the declarations will NOT compile as such unless they are assigned values during declaration itself. I have omitted them for clarity, and besides, adding that will require another two lines of code for each example.

Hide Copy Code

```
char ** p1;                   //        pointer to       pointer to       char
const char **p2;              //        pointer to       pointer to const char
char * const * p3;            //        pointer to const pointer to       char
const char * const * p4;      //        pointer to const pointer to const char
char ** const p5;             //  const pointer to       pointer to       char
const char ** const p6;       //  const pointer to       pointer to const char
char * const * const p7;      //  const pointer to const pointer to       char
const char * const * const p8; //  const pointer to const pointer to const char
```

[Back to contents]

The subtleties of typedef

`typedef` allows you a way to overcome the *-applies-to-variable-not-type rule. If you use a `typedef` like:

Hide Copy Code

```
typedef char * PCHAR;
PCHAR p,q;
```

both `p` and `q` become pointers. If the `typedef` had not been used, `q` would be a `char`, which is counter-intuitive.

Here are a few declarations made using `typedef`, along with the explanation:

Hide Copy Code

```
typedef char * a;  // a is a pointer to a char

typedef a b();     // b is a function that returns
                   // a pointer to a char

typedef b *c;      // c is a pointer to a function
                   // that returns a pointer to a char

typedef c d();     // d is a function returning
```

```
                    // a pointer to a function
                    // that returns a pointer to a char

typedef d *e;       // e is a pointer to a function
                    // returning  a pointer to a
                    // function that returns a
                    // pointer to a char

e var[10];          // var is an array of 10 pointers to
                    // functions returning pointers to
                    // functions returning pointers to chars.
```

`typedef`s are usually used with structure declarations as shown below. The following structure declaration allows you to omit the `struct` keyword when you create structure variables even in C, as is normally done in C++.

Hide Copy Code

```
typedef struct tagPOINT
{
    int x;
    int y;
}POINT;

POINT p; /* Valid C code */
```

[Back to contents]

Function pointers

Function pointers are probably the greatest source of confusion when it comes to interpreting declarations. Function pointers were used in the old DOS days for writing TSRs; in the Win32 world and X-Windows, they are used in callback functions. There are lots of other places where function pointers are used: virtual function tables, some templates in STL, and Win NT/2K/XP system services. Let's see a simple example of a function pointer:

Hide Copy Code

```
int (*p)(char);
```

This declares `p` as a pointer to a function that takes a `char` argument and returns an `int`.

A pointer to a function that takes two `float`s and returns a pointer to a pointer to a `char` would be declared as:

Hide Copy Code

```
char ** (*p)(float, float);
```

How about an array of 5 pointers to functions that receive two `const` pointers to `char`s and return a `void` pointer?

Hide Copy Code

```
void * (*a[5])(char * const, char * const);
```

[Back to contents]

The right-left rule [Important]

This is a simple rule that allows you to interpret any declaration. It runs as follows:

Start reading the declaration from the innermost parentheses, go right, and then go left. When you encounter parentheses, the direction should be reversed. Once everything in the parentheses has been parsed, jump out of it. Continue till the whole declaration has been parsed.

One small change to the right-left rule: When you start reading the declaration for the first time, you have to start from the identifier, and not the innermost parentheses.

Take the example given in the introduction:

Hide Copy Code

```
int * (* (*fp1) (int) ) [10];
```

This can be interpreted as follows:

1. Start from the variable name ---------- `fp1`
2. Nothing to right but `)` so go left to find `*` ---------- is a pointer
3. Jump out of parentheses and encounter (`int`) --------- to a function that takes an `int` as argument
4. Go left, find `*` ---------- and returns a pointer
5. Jump put of parentheses, go right and hit `[10]` -------- to an array of 10
6. Go left find `*` ---------- pointers to
7. Go left again, find `int` ---------- `int`s.

Here's another example:

Hide Copy Code

```
int *( *( *arr[5])())();
```

1. Start from the variable name ---------- `arr`
2. Go right, find array subscript ---------- is an array of 5
3. Go left, find `*` ---------- pointers
4. Jump out of parentheses, go right to find `()` ------ to functions
5. Go left, encounter `*` ---------- that return pointers
6. Jump out, go right, find `()` ---------- to functions
7. Go left, find `*` ---------- that return pointers
8. Continue left, find `int` ---------- to `int`s.

[Back to contents]

Further examples

The following examples should make it clear:

Hide Shrink Copy Code

```
float ( * ( *b()) [] )();             // b is a function that returns a
                                      // pointer to an array of pointers
                                      // to functions returning floats.

void * ( *c) ( char, int (*)());      // c is a pointer to a function that takes
                                      // two parameters:
                                      //     a char and a pointer to a
                                      //     function that takes no
                                      //     parameters and returns
                                      //     an int
                                      // and returns a pointer to void.

void ** (*d) (int &,
```

```
  char **(*)(char *, char **));        // d is a pointer to a function that takes
                                       // two parameters:
                                       //     a reference to an int and a pointer
                                       //     to a function that takes two parameters:
                                       //        a pointer to a char and a pointer
                                       //        to a pointer to a char
                                       //     and returns a pointer to a pointer
                                       //     to a char
                                       // and returns a pointer to a pointer to void

float ( * ( * e[10])
    (int &) ) [5];                    // e is an array of 10 pointers to
                                       // functions that take a single
                                       // reference to an int as an argument
                                       // and return pointers to
                                       // an array of 5 floats.
```

[Back to contents]

Suggested reading

- A Prelude to pointers by Nitron.
- `cdecl` is an excellent utility that explains variable declarations and does much more. You can download the Windows port of cdecl from here.

[Back to contents]

Credits

I got the idea for this article after reading a thread posted by Jörgen Sigvardsson about a pointer declaration that he got in a mail, which has been reproduced in the introduction. Some of the examples were taken from the book "Test your C skills" by Yashvant Kanetkar. Some examples of function pointers were given by my cousin Madhukar M Rao. The idea of adding examples with mixed `*`s and `&`s and `typedef` with `struct`s was given by my cousin Rajesh Ramachandran. Chris Hills came up with modifications to the right-left rule and the way in which some examples were interpreted.