

Declarations

- **Objectives**
 - Learn the syntax of declaration
- **Contents**
 - Reading declarations: the Green-Cross method
 - Writing declarations: the Operator-Precedence method
 - Using *typedef*
 - Casting operations
 - Using *const*
- **Practical**
 - Practice and test your understanding
- **Summary**



In this chapter, we shall examine in some detail the rules for formulating and understanding complex declarations, a topic which even the most experienced C programmers sometimes have nightmares about!

There is in fact a relatively straightforward rule quaintly referred to as the "Green-Cross" method, which will help you in most situations to understand the meaning of a particularly complicated declaration.

However, when it comes to writing complex declarations of your own, a more logical approach is required and it becomes necessary to consult the C precedence table to decide what is needed. A number of examples are presented throughout the chapter to make sure you understand the rules.

The chapter closes with a study of `typedef` and casting operations, and shows how the `const` keyword may be used in a variety of interesting and useful ways in complex declarations.

International note:

The "Green-Cross" method is named after a old road-safety campaign in the United Kingdom. If you drive on the *right* in your country then do not use it for crossing the road! However it is perfectly safe to use for reading C declarations, wherever you are.

The Green Cross method

- Find the variable being declared
- Look to the right for attributes
- Look to the left for attributes
- Keep looking right, then left, until the whole declaration is read
- Note: parentheses can change the right/left order



On finding:	Say:
*	Pointer to ...
[]	Array of ...
()	Function which returns a ...

Examples:

int *a[100];

Thing being declared is "a".

Look to the right to find "[100]", say *array of 100*.

Look to the left and find "*", say *pointer to*.

Look to the right and find ";".

Look to the left and find "int".

a is an array of 100 pointers to integers.

float *f (int);

Thing being declared is "f".

Look to the right and find "(", say *function which returns a*.

Look to the left and find "*", say *pointer to*.

Look to the left and find "float".

f is a function which returns a pointer to a float.

char (*p) (void);

Thing being declared is "p".

Prevented from looking right by parentheses.

Look left and find "*", say *pointer to*.

Look right and find "(", say *function which returns a*.

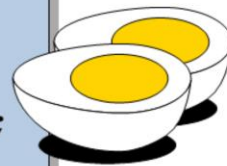
Look left and see "char".

p is a pointer to a function which returns a char.

Note the difference between the parentheses which force us to look left first and the () brackets which declare a "function which returns a".

Examples

```
int    **pp;
float  *apf[10];
int    *frp(int, char);
char   (*pf)(short);
int    *(*pf2)(void);
date   *func(void);
date   *(*pfunc)(void);
int    (*x)[10];
```



Eggs-ample

```
int    **pp;
```

pp is a *pointer to a pointer to an int*.

```
float  *apf [10];
```

apf is an *array of 10 pointers to float*.

```
int    *frp (int, char);
```

frp is a *function which returns a pointer to an int* (the function accepts two parameters, the first is an int, the second is a char).

```
char   (*pf) (short);
```

pf is a *pointer to a function which returns a char* (the function accepts a short int as a parameter).

```
int    *(*pf2) (void);
```

pf2 is a *pointer to a function which returns a pointer to an int* (the function which is pointed to accepts no parameters).

```
date   *func (void);
```

func is a *function which returns a pointer to a date structure* (the function accepts no parameters).

```
date   *(*pfunc) (void);
```

pfunc is a *pointer to a function which returns a pointer to a date structure* (the function pointed to accepts no parameters).

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

x is a *pointer to an array of 10 ints*.

Problems with Green Cross

- Although straightforward, the Green-Cross method can fail in some declarations:

```
int m[5][9];
```

Green Cross says "m" is an
"array of 5 int arrays of 9"

- It would be possible to modify the rule to cover these cases
- A better approach is to look at operator precedences - this will help us to *write* declarations as well as to read them

```
double (*apf[10])(int, float);
```

Green Cross does not predict
that the extra set of parentheses
are necessary

Although the Green-Cross rule is easy to remember and has worked well in the cases covered in the previous slide, it can sometimes fail.

One problem arises when reading declarations of arrays of arrays. Using the Green-Cross method on the declaration above yields "m is an array of 5 integer arrays of 9". This is not only incorrect (since m is an array of 5 arrays of 9 ints), it doesn't even make sense.

Admittedly, it would be possible to modify the rule and say something like "don't look left if [] have just been seen". This might actually work.

Far more serious is the inability of the rule to predict when parentheses are necessary. We have already seen the declaration of a pointer to a function. It would seem natural to declare an array of these. Applying the Green-Cross method to the following declaration:

```
double *apf [10](int, float);
```

would seem to give "apf is an array of 10 pointers to functions returning double". However, what is actually declared is "an array of 10 functions which return pointers to doubles". C does not allow arrays of *functions*, but does allow arrays of *pointers to functions*. An array of functions would be an impossible object, since all the functions would have to be of the same size and be arranged contiguously in memory!

The Green-Cross method is good for *reading* declarations, but not for *writing* them.

The Operator Precedence method

- The symbols `*` `[]` and `()` are actually C operators with their own precedence as shown in the following:

Operator	Description
<code>[]</code>	Array index
<code>()</code>	Function call
<code>*</code>	Contents of

- Thus in:

```
int (*apf[10])(int, float);
```

- Extra parenthesis are required to bind `*` to `apf`. Otherwise the function call operator `()` would bind too tightly

The Operator-Precedence method states that each of the `*` `[]` or `()` symbols used in a declaration are actually operators. Accepting this implies that they have the precedence defined in the precedence table for the C language.

The table puts the array access operator `[]` and function call operator `()` at joint highest precedence in the whole table. The `*` operator has a slightly lower precedence.

The Operator-Precedence method helps us to understand the following declaration:

```
int z[2][3][4];
/* z is an array of 2 arrays of 3 arrays of 4 ints */
```

The method explains why the extra set of parentheses are necessary in the following:

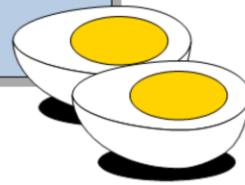
```
int (*apf [10]) (int, float);
/* apf is an array of 10 pointers to functions */
```

It also helps to explain why the following declaration makes no sense in C:

```
int *apf [10] (int, float);
// Invalid declaration - cannot have array of functions!
```

Examples

```
void (*fpv[14]) (void);  
  
void (*cf(void (*) (void))) (char);  
  
int (*cw(void)) [5];  
  
float (*aap[10]) [23];
```



```
void (*fpv[14]) (void);
```

fpv is an *array of 14 pointers to functions which return void*. The functions pointed to take no parameters.

```
void (* cf (void (*) (void)) ) (char);
```

cf is a *function which returns a pointer to a function that returns void* and takes a single *char* parameter. *cf* itself accepts one parameter of type *pointer to a function which returns void* and takes no parameters.

```
int (*cw(void)) [5];
```

cw is a *function which returns a pointer to an array of 5 ints*. The function *cw* does not take any parameters.


```
float (*aap[10]) [23];
```

aap is an *array of 10 pointers to arrays of 23 floats*.

The typedef keyword

- The `typedef` keyword is used to declare new names for existing types
- Instead of declaring a variable, `typedef` declares a type name
- Pretend that `typedef` is absent - the type declared is the type the variable would have had

*Type names in
all uppercase?*



```
typedef    float (*PTAF) [23];
PTAF      app[10];
PTAF      func(void);

typedef    void (*PTVF) (void);
PTVF      fpv[14];
PTVF      frpf(char);

typedef    void (*PTVFC) (char);
PTVFC      cf(PTVF);
```

Many of the horrors of overly complex C declarations may be banished by using `typedef`, which creates new *names* for existing types, rather than new *types*.

The key to understanding `typedef` is to pretend that the keyword is absent and that a variable is being declared. Consider the following example:

```
typedef float (*PTAF) [23];
// PTAF is a typedef representing "a pointer to an
// array of 23 floats"
```

If the word `typedef` were covered up, this would leave:

```
float (*PTAF) [23];
// PTAF - pointer to array of 23 floats
```

Since `typedef` was present, wherever `PTAF` is used, the compiler will understand it as being a *pointer to array of 23 floats*. Which of the following (equivalent) declarations would you prefer to see?

```
PTAF app[10];    // Declaration using our typedef PTAF
float (*app[10]) [23];
// Equivalent without use of typedef!
```

The slide questions the use of an all uppercase typedef name. This is because uppercase names are traditionally reserved for the preprocessor. If `PTAF` has been `#defined` by a preprocessor directive then the code you see will not be the code the compiler sees and you might easily spend hours of frustration trying to track the problem down.

Casting

- **To cast an object to another type, take the declaration, remove the variable and add brackets:**

```
void (*fp1) (int) ;
void (*fp2) (char) ;

fp1 = fp2;                /* no go: different types */
fp1 = (void (*) (int))fp2; /* works with a cast */

int (*p) [20];             /* ptr to array of 20 ints */
int (*x) [30];             /* ptr to array of 30 ints */
p = (int (*) [20])x;

typedef int * PTI;         /* typedef of ptr to int */
char *pc;                 /* declare ptr to char */

PTI a;
a = pc;                   /* no go: different types */
a = (PTI)pc;              /* ok */
```

Casting an object to another type is very straightforward. For instance, to cast NULL to a pointer to a function, taking no parameters and returning no value, do the following:

1. Declare `x` as a pointer to a function which takes no parameters and returns no value:

```
void (*x) (void);
```

2. Cast NULL to this type by removing `x` and the semicolon:

```
void (*) (void)
```

3. Add brackets (so that it becomes a cast), and add the NULL value which is to be cast:

```
(void (*) (void)) NULL
```

Note: the `(*)` looks rather strange, but it is necessary. If the parentheses are removed as shown below, NULL is cast to a function which takes no parameters and returns a void pointer:

```
(void *(void)) NULL
```

The const keyword

- On finding the `const` keyword in a declaration say "constant"
- Examples:

```
int const * p1;           /* ptr to a const int */
const int * p2;           /* as above */

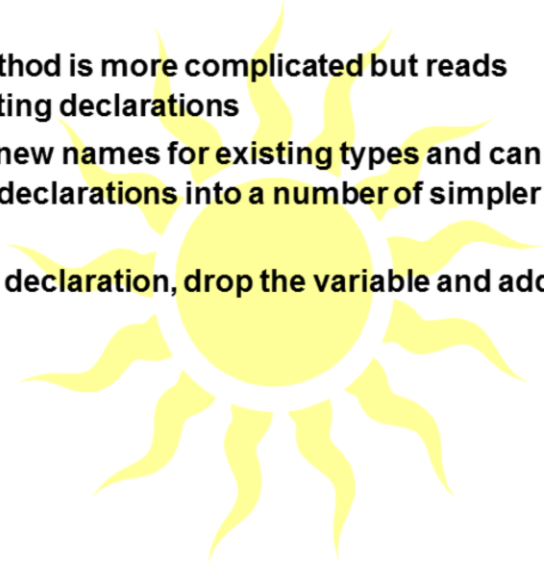
int * const p3 = ...;     /* const ptr to an int */
int (*const cp)[10] = ...; /* const ptr to array of 10 ints */
int const (*pc)[10];      /* ptr to array of 10 const ints */

void (*const fp)(int) = ... ; /* const ptr to function */
```

The above examples show that, in practice, coping with the `const` keyword is really a simple extension of the rules already discussed. Note that a `const` pointer must be initialised at its point of definition otherwise it will be a random value which cannot be altered.

Summary

- The Green-Cross method is simple and works *most* of the time for reading declarations
- The Operator-Precedence method is more complicated but reads any declaration - good for writing declarations
- The `typedef` keyword creates new names for existing types and can be used to break up complex declarations into a number of simpler steps
- For complex casting, take the declaration, drop the variable and add brackets
- `const` - no problem



In this chapter, we have studied two techniques for reading and writing complex declarations.

Firstly, we have seen how the Green-Cross method can be used in 90% of declarations to understand what they mean. When it comes to writing declarations of your own, it becomes necessary to consult the C precedence table to decide where parentheses are required.

The **`typedef`** keyword has been introduced and explored, and we have seen how useful it can be for simplifying otherwise horrific declarations. Typedefs are also a useful tool for simplifying complex casts.

