Lecture 1++

C features, you know or don't

Are you kidding me?

You have to get use to that

~~NULLL~~

- Sins of commission
- sins of omission
- sins of mission
- Declarations in C

Sins of Commission

Sins of Commission

Switch statement

```
switch (expression){
 case constant-expression: zero-or-more-statements
                 default: zero-or-more-statements
 case constant-expression: zero-or-more-statements
}
```

257 case
labels for a switch
statement (ANSI C
Standard,

Sins of Commission

Switch statement

```
switch (i) {
 case 5+3: do_again;
 case 2:   printf("I loop unremittingly \n"); goto do_again;
 default:  i++;
 case 3: ;
}
```

how to fix it?

```
const int two=2;
switch (i) {
 case 1: printf("case 1 \n");
 case two: printf("case 2 \n");
**error** ^^^ integral constant expression expected
case 3: printf("case 3 \n"); default: ;
}
```

Sins of Commission

Switch statement

```
switch (2) {
 case 1: printf("case 1 \n");
 case 2: printf("case 2 \n");
 case 3: printf("case 3 \n");
 case 4: printf("case 4 \n");
 default: printf("default \n");
}
```

```
case 2
case 3
case 4
default
```

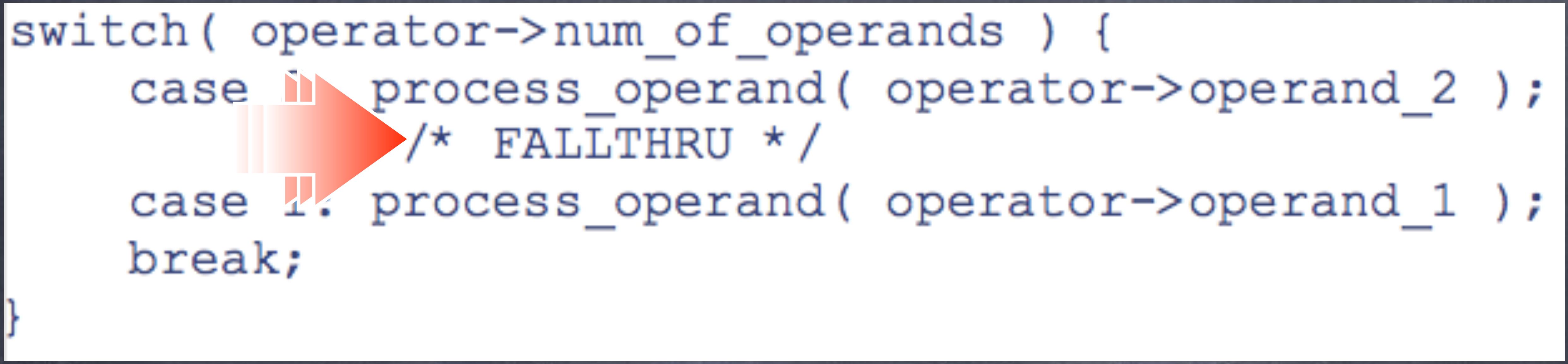
Fall Through

Sins of Commission

Fall Through

- We analyzed the Sun C compiler sources to see how often the default fall through was used. The Sun ANSI C compiler front end has 244 switch statements, each of which has an average of seven cases. Fall through occurs in just 3% of all these cases.
- In other words, the normal switch behavior is wrong 97% of the time.

Sins of Commission



```
switch( operator->num_of_operands ) {
    case 2: process_operand( operator->operand_2 );
        /* FALLTHRU */
    case 1: process_operand( operator->operand_1 );
    break;
}
```

Fall Through

Sins of Commission

Switch statement

```
network code()
{
 switch (line) {
  case THING1: doit1(); break;
  case THING2: if (x == STUFF) {
          do_first_stuff();
          if (y == OTHER_STUFF)
              break;
          do_later_stuff();
          } /* coder meant to break to here...*/
          initialize_modes_pointer();
          break;
  default: processing();
} /* ...but actually broke to here! */
use_modes_pointer();
/* leaving the modes_pointer uninitialized */
}
```

This code eventually caused the first major network problem in AT&T's 114-year history. The saga is described in greater detail on page 11 of the January 22, 1990 issue of *Telephony* magazine.

Sins of Commission

static keyword

```
generate_initializer(char * string)
    {
    static char separator='';
    printf( "%c %s \n", separator, string);
    separator = ',';
   }

function apple (){ /* visible everywhere */ }

extern function pear () { /* visible everywhere */ }

static function turnip(){ /* not visible outside this file */ }
```

The problem of too wide scope is often seen in libraries: one library needs to make an object visible to another library. The only possibility is to make it globally known; but then it is visible to anyone that links with the library. This is an "all-or-nothing" visibility—symbols are either globally known or not known at all. There's no way to be more selective in revealing information in C.

Sins of Commission

static keyword

```
function apple (){ /* visible everywhere */ }

extern function pear () { /* visible everywhere */ }

static function turnip(){ /* not visible outside this file */ }
```

too wide scope

Sins of Omission

Sins of Omission

similar C symbols have multiple different meanings

`Static`	❋ Inside a function, *<u>retains its value between calls</u>* ❋ At the function level, *<u>visible only in this file</u>*
`extern`	❋ Applied to a function definition, *<u>has global scope</u>* (and is redundant) ❋ Applied to a variable, *<u>defined elsewhere</u>*
`void`	❋ As the return type of a function, doesn't return a value ❋ In a pointer declaration, the type of a generic pointer ❋ In a parameter list, takes no parameters
`*`	❋ The multiplication operator ❋ Applied to a pointer, indirection ❋ In a declaration, a pointer
`&`	❋ Bitwise AND operator ❋ Address-of operator

Sins of Omission

similar C symbols have multiple different meanings

`=`
`==`
	✱ Assignment operator
	✱ Comparison operator

`<=`
`<<=`
	✱ Less-than-or-equal-to operator
	✱ Compound shift-left assignment operator

`<` `<`	✱ Less-than operator ✱ Left delimiter in #include directive
`(  )`	✱ Enclose formal parameters in a function definition ✱ Make a function call ✱ Provide expression precedence ✱ Convert (cast) a value to a different type ✱ Define a macro with arguments ✱ Make a macro call with arguments ✱ Enclose the operand of the sizeof operator when it is

Sins of Omission

Some of the Operators Have the Wrong Precedence

Precedence problem	Expression	What people expect	What they actually get
. is higher than * the p->f op was made to smooth over this	*p.f		
[ ] is higher than *	int *ap[]		
function () higher than *	int *fp()		
"==" and "!=" higher precedence than assignment	c=getchar() != EOF		
"==" and "!=" higher precedence than bitwise operators	(val & mask != 0)		
arithmetic higher precedence than shift	msb<<4 + lsb		
"," has lowest precedence of all operators	i = 1, 2;		

Sins of Omission

Some of the Operators Have the Wrong Precedence

Precedence problem	Expression	What people expect	What they actually get
. is higher than * the p->f op was made to smooth over this	*p.f	the f field of what p points to (*p).f	take the f offset from p, use it as a pointer *(p.f)
[ ] is higher than *	int *ap[]	ap is a ptr to array of ints int (*ap) []	ap is an array of pts-to-int int *(ap[])
function () higher than *	int *fp()	fp is a ptr to function returning int int (*fp) ()	fp is a function returning ptr-to-int int *(fp())
"==" and "!=" higher precedence than assignment	c=getchar() != EOF	(c=getchar()) != EOF	c=(getchar() != EOF)
"==" and "!=" higher precedence than bitwise operators	(val & mask != 0)	(val & mask) != 0	val & (mask != 0)
arithmetic higher precedence than shift	msb<<4 + lsb	(msb<<4) + lsb	msb<<(4 + lsb)
"," has lowest precedence of all operators	i = 1, 2;	i = (1, 2);	(i=1), 2;

Sins of Omission

Some of the Operators Have the Wrong Precedence

```
x = f() + g() * h();
```

remember that while precedence and associativity tell you what is grouped with what, the order in which these groupings will be evaluated is *always* undefined

```
int a, b=1, c=2;
a = b = c;
```

All assignment-operators have right associativity. The associativity protocol says that this means the rightmost operation in the expression is evaluated first, and evaluation proceeds from right to left. Thus, the value of c is assigned to b. Then the value of b is stored in a. a gets the value 2. Similarly, for operators with left associativity (such as the bitwise and's and or 's), the operands are grouped from left to right.

Sins of Omission

Some routines in the standard library have unsafe semantics

```
main(argc, argv)
   char *argv[];
{
   char line[512];
         ...
gets(line);
```

line is a 512-byte array allocated automatically on the stack. When a user provides more input than that to the finger daemon, the gets() routine will keep putting it on the stack

```
if (fgets(line, sizeof(line), stdin) == NULL)
    exit(1);
```

Sins of Mission

Sins of Mission

Return a local pointer

```
/* Convert the source file timestamp into a localized date
string */
char * localized_time(char * filename)
{
  struct tm *tm_ptr;
  struct stat stat_block;
  char buffer[120];
  /* get the sourcefile's timestamp in time_t format */
  stat(filename, &stat_block);
  /* convert UNIX time_t into a struct tm holding local time */
  tm_ptr = localtime(&stat_block.st_mtime);}
  /* convert the tm struct into a string in local format */
  strftime(buffer, sizeof(buffer), "%a %b %e %T %Y", tm_ptr);
   eturn buffer;
```

Output

The problem is in the final line of the function, where the buffer is returned. Even if you return a pointer to such a variable, as here, there's no telling what it points to once the function is exited.

Luis$*7 Y dsfg^8 *@!

lundi 6 Avril 2012

Sins of Mission

Right use of argv[][]

```
if ( argv[argc-1][0] == '-' || (argv[argc-2][1]
== 'f' ) )
   readmail(argc, argv);
else
   sendmail(argc, argv);
```

```
mail -h -d -f /usr/zhu/mymailbox

mail effie liu
```

Sins of Mission

Comments

```
int hashval=0;
/* PJW hash function from "Compilers: Principles, Techniques,
and Tools"
* by Aho, Sethi, and Ullman, Second Edition.
while (cp < bound)
{
   unsigned long overflow;
   hashval = ( hashval <<4)+*cp++;
   if ((overflow = hashval & (((unsigned long) 0xF) << 28)) != 0)
   hashval ^= overflow | (overflow >> 24);
}
hashval %= ST_HASHSIZE; /* choose start bucket */
/* Look through each table, in turn, for the name. If we fail,
* save the string, enter the string's pointer, and return it.
*/
for (hp = &st_ihash; ; hp = hp->st_hnext) {
   int probeval = hashval; /* next probe value */
```

Sins of Mission

Skills

Comments

```
int main (int argc, const char * argv[])
{
  int d= -1, x;
    unsigned u_d=(unsigned)d;
    printf("%u\n",u_d);

    // if (d <= TOTAL_ELEMENTS-2)
    #if 0
    if (d <= (int)TOTAL_ELEMENTS-2)
    {
        x = array[d+1];
        printf("d is less\n");
    }
    else
        printf("d is more\n");
    #endif
    return 0;
}
```

Sins of Mission

Skills

lint

```
unsignedProblem.c:15:15:
    Parameter argc not used
    A function parameter is not used in the body of the function. If the
argument is needed for type compatibility or future plans, use /*@unused@*/
in the argument declaration. (Use -paramuse to inhibit warning)
unsignedProblem.c:15:34:
    Parameter argv not used
unsignedProblem.c:10:5:
    Variable exported but not used outside unsignedProblem: array
  A declaration is exported, but not used outside this module. Declaration can
  use static qualifier. (Use -exportlocal to inhibit warning)
```

splint — GCC

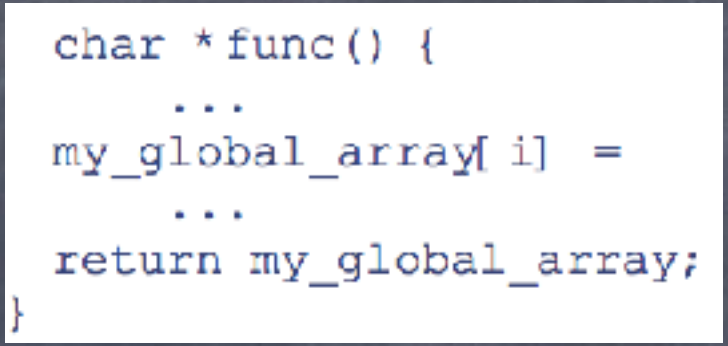
Sins of Mission

Solutions

Return a pointer to a string literal.

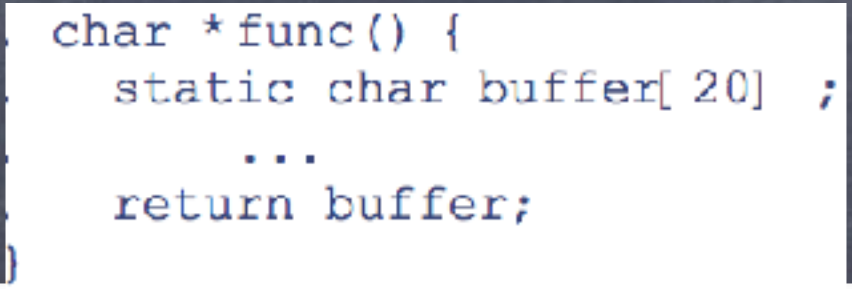
```
char *func() { return "simple strings"; }
```

Use a globally declared array.



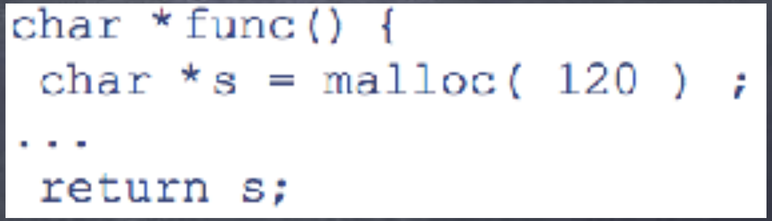
```
 char *func() {
      ...
 my_global_array[ i] =
      ...
 return my_global_array;
}
```

Use a static array.



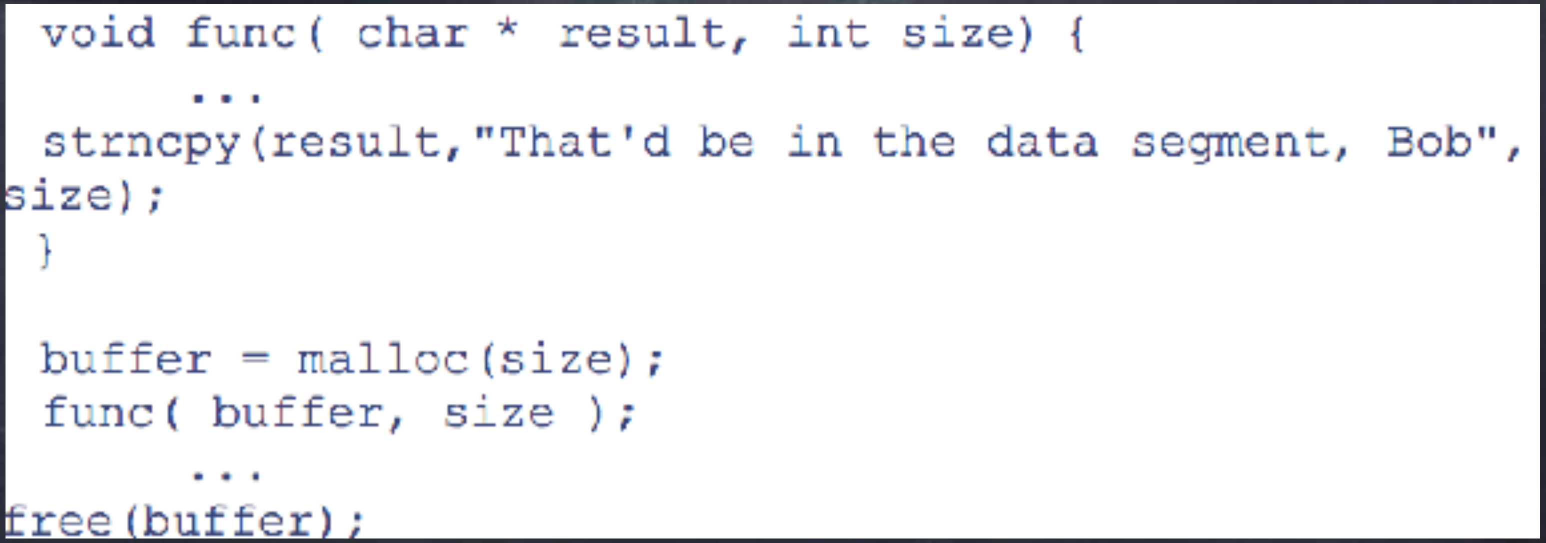
```
  char *func() {
    static char buffer[ 20] ;
        ...
    return buffer;
}
```

Explicitly allocate some memory to hold the return value.



```
char *func() {
 char *s = malloc( 120 ) ;
...
 return s;
```

Require the caller to allocate the memory to hold the return value.



```
 void func( char * result, int size) {
      ...
 strncpy(result,"That'd be in the data segment, Bob",
size);
 }

 buffer = malloc(size);
 func( buffer, size );
      ...
free(buffer);
```

Declarations in C

Declarations in C

Examples

Illegal

- a function can't return a function, so you'll never see `foo()()`
- a function can't return an array, so you'll never see `foo()[]`
- an array can't hold a function, so you'll never see `foo[]()`

Legal

- a function returning a *pointer to* a function is allowed: `int (* fun())();`
- a function returning a *pointer to* an array is allowed: `int (* foo())[]`
- an array holding *pointers to* functions is allowed: `int (*foo[])()`
- an array can hold other arrays, so you'll frequently see `int foo[][]`

Declarations in C

Key Words

QUALIFIER

volatile / const

TYPE

void/char/signed/unsigned/short/int/long/float/double/struct/union/enum

DECLARATOR

Declarations in C

struct

Definition

```
struct {stuff... }
```

```
struct {stuff... } plum, pomegranate, pear;
```

```
struct fruit_tag {stuff... } plum, pomegranate, pear;
```

```
struct optional_tag {
      type_1 identifier_1;
      type_2 identifier_2;
      ...
      type_N identifier_N;
} optional_variable_definitions;
```

Declarations in C

struct

Instances

```
struct date_tag { short dd,mm,yy; } my_birthday, xmas;
struct date_tag easter, groundhog_day;
```

```
/* process ID info */
   struct pid_tag {
   unsigned int inactive :1;
   unsigned int :1;
   unsigned int refcount :6;
   unsigned int :0;
   short pid_id;
   struct pid_tag *link;
};
```

Declarations in C

struct

Skills

```
struct s_tag { int a[100]; };
struct s_tag orange, lime, lemon;
struct s_tag twofold (struct s_tag s) {
     int j;
     for (j=0;j<100;j++) s.a[j] *= 2;
     return s;
}
main() {
     int i;
     for (i=0;i<100;i++) lime.a[i] = 1;
     lemon = twofold(lime);
     orange = lemon; /* assigns entire struct */
}
```

```
/* struct that points to the next struct */
struct node_tag {
     int datum;
     struct node_tag *next;
};
struct node_tag a,b;
a.next = &b;  /* example link-up */
a.next->next=NULL;
```

Declarations in C

struct members

Expansion

Since the language now permits structure parameters, structure assignment and functions returning structures, the concept of a structure expression is now part of the C language. A structure value can be produced by an assignment, by a function call, by a comma operator expression or by a conditional operator expression:

```
s1 = (s2 = s3)
sf(x)
(x, s1)
x ? s1 : s2
```

In these cases, the result is not an lvalue; hence it cannot be assigned to nor can its address be taken.

Declarations in C

struct

Expansion

Similarly, x.y is an lvalue only if x is an lvalue. Thus none of the following valid expressions are lvalues:

```
sf(3).a
(s1=s2).a
((i==6)?s1:s2).a
(x,s1).a
```

Even when x.y is an lvalue, it may not be modifiable:

```
const struct S s1;
s1.a = 3;          /* invalid */
```

Declarations in C

union

Unions are known as the variant part of variant records in many other languages. They have a similar appearance to structs, but the memory layout has one crucial difference. Instead of each member being stored after the end of the previous one, all the members have an offset of zero. The storage for the individual members is thus overlaid: only one member at a time can be stored there.

```
union optional_tag {
    type_1 identifier_1;
    type_2 identifier_2;
    ...
    type_N identifier_N;
} optional_variable_definitions;
```

20 million animals,
save up to 20 Mb

```
struct creature {
    char has_backbone;
    char has_fur;
    short
    num_of_legs_in_excess_of_4;
};
```

```
union secondary_characteristics {
 char has_fur;
 short num_of_legs_in_excess_of_4;
};
struct creature {
 char has_backbone;
 union secondary_characteristics form;
};
```

Declarations in C

union

Skills

```
union bits32_tag {
int whole;
/* one 32-bit value */
struct {char c0,c1,c2,c3;} byte;
/* four 8-bit bytes
} value;
```

This union allows a programmer to extract the full 32-bit value, or the individual byte fields value.byte.c0, and so on.

In reality, structs are about one hundred times more common than unions.

Declarations in C

enum

Enums (enumerated types) are simply a way of associating a series of names with a series of integer values. In a weakly typed language like C, they provide very little that can't be done with a #define, so they were omitted from most early implementations of K&R C.

```
enum optional_tag {stuff... } optional_variable_definitions;
```

```
enum sizes { small=7, medium, large=10, humungous };
```

- The integer values start at zero by default.
- If you assign a value in the list, the next value is one greater, and so on.
- There is one advantage to enums: unlike #defined names which are typically discarded during compilation, enum names usually persist through to the debugger, and can be used while debugging your code.

Declarations in C

The Precedence Rule

```
char* const *(*next)();
```

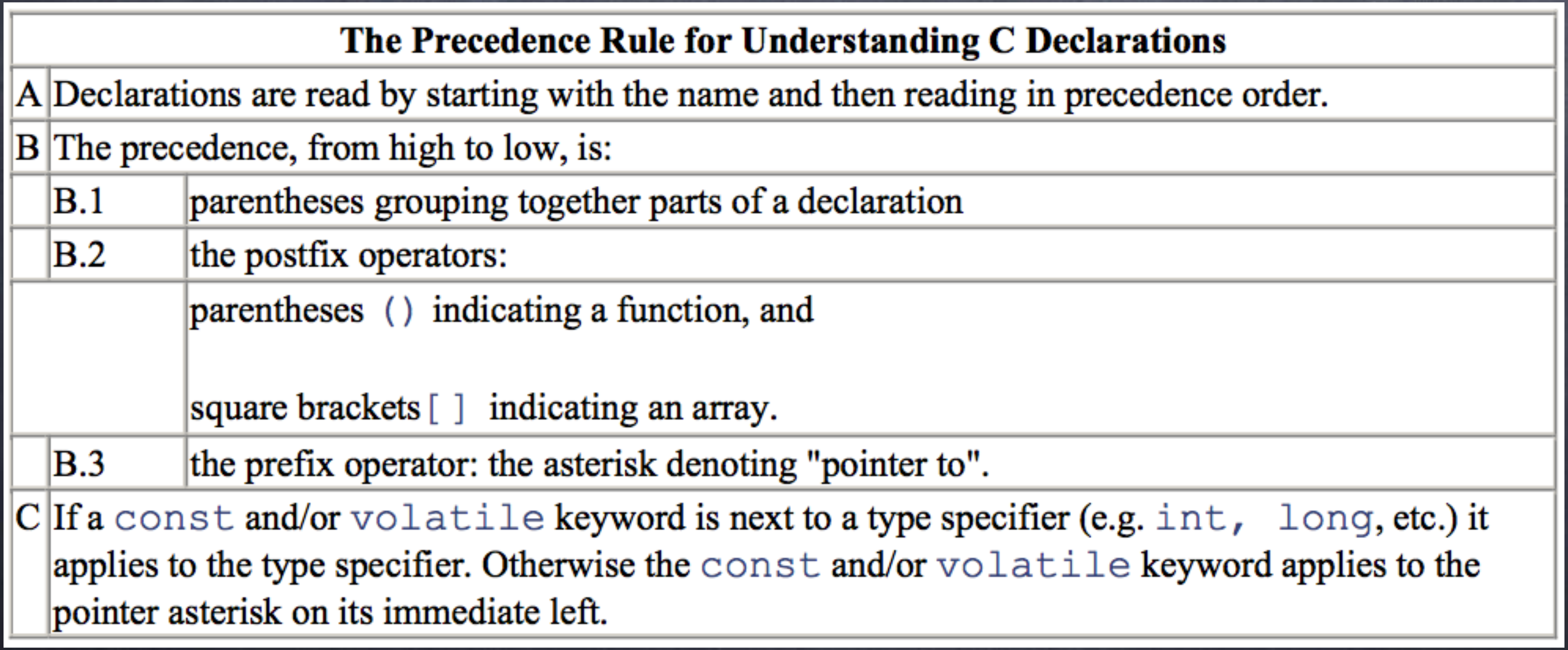
"next is a pointer to a function returning a pointer to a const pointer-to-char"

```
char *(*c[10])(int **p);
```

"c is an array[0..9] of pointer to a function returning a pointer-to-char"

Declarations in C

The Precedence Rule



The Precedence Rule for Understanding C Declarations		
A	Declarations are read by starting with the name and then reading in precedence order.	
B	The precedence, from high to low, is:	
	B.1	parentheses grouping together parts of a declaration
	B.2	the postfix operators:
		parentheses `()` indicating a function, and square brackets `[ ]` indicating an array.
	B.3	the prefix operator: the asterisk denoting "pointer to".
C	If a `const` and/or `volatile` keyword is next to a type specifier (e.g. `int, long`, etc.) it applies to the type specifier. Otherwise the `const` and/or `volatile` keyword applies to the pointer asterisk on its immediate left.	

Declarations in C

typedef

Typedef introduces a new name for a type rather than reserving space for a variable. In some ways, a typedef is similar to macro text replacement—it doesn't introduce a new type, just a new name for a type, but there is a key difference.

Declarations in C

Difference between typedef & #define

```
#define peach int
unsigned peach i;
```

☑

```
typedef int banana;
unsigned banana i;
```

☒

```
#define int_ptr int *
int_ptr chalk,
cheese;
```

```
int * chalk, cheese;
```

```
typedef char * char_ptr;
char_ptr Bentley, Rolls_Royce;
```

Both Bentley and Rolls_Royce to be the same, a pointer to a char.

Declarations in C

Namespaces in C

There are multiple namespaces in C:
* label names
* tags (one namespace for all structs, enums and unions)
* member names (each struct or union has its own namespace)
* everyting else

```
struct foo {int foo;} foo;
```

$\downarrow$

```
sizeof (foo) ?
```

Declarations in C

Namespaces in C

Skills

```
typedef struct baz {int baz;} baz;
          struct baz variable_1;
                 baz variable_2;
```

typedef struct fruit {int weight, price_per_lb } fruit;

struct fruit mandarin; /* uses structure tag "fruit" */

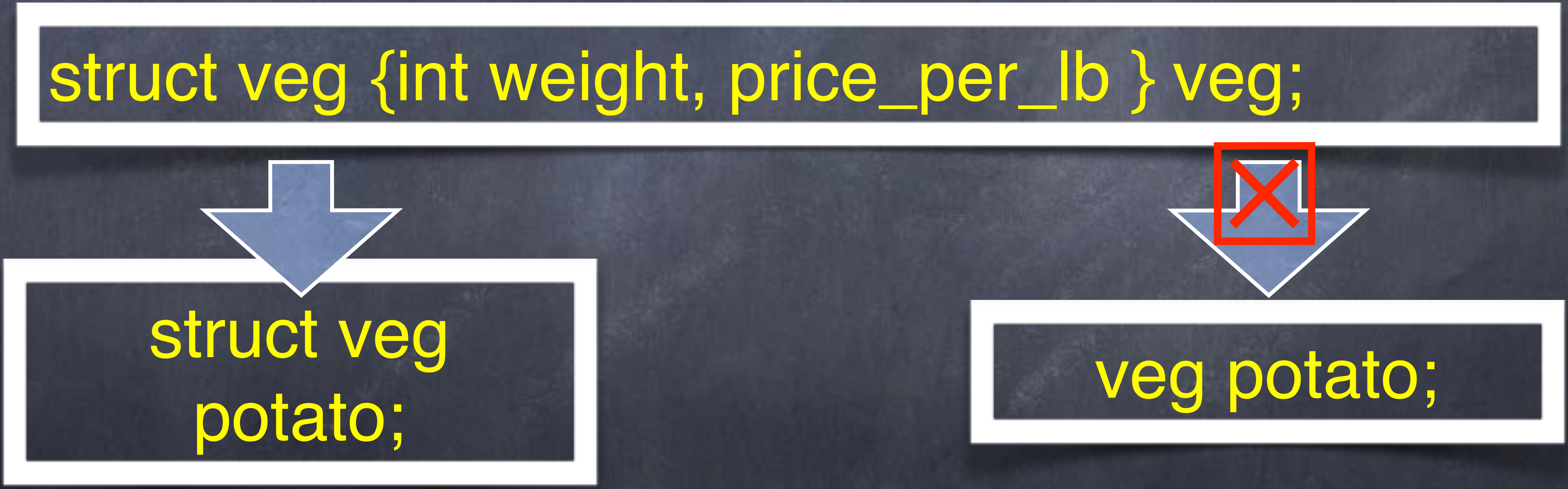
fruit tangerine; /* uses structure type "fruit" */

Declarations in C

Namespaces in C

Skills

But



```
struct veg {int weight, price_per_lb } veg;
```

```
struct veg
potato;
```

```
veg potato;
```

Declarations in C

Namespaces in C

Handy Heuristic

- Don't bother with typedefs for structs.
- All they do is to save you writing the word "struct", which is a clue that you probably shouldn't be hiding anyway.
- Use typedefs for:
  - types that combine arrays, structs, pointers, or functions.
  - portable types. When you need a type that's at least (say) 20-bits, make it a typedef. Then when you port the code to different platforms, select the right type, short, int, long, making the change in just the typedef, rather than in every declaration.

Homework 1

- 想象一个应用场景，应用返回函数 (*foo[])()形式
- 要求：
  - 给出应用场景的描述
  - 给程序加上足够的注解
  - 描述程序的应用局限性、优点

You have to get use to that

- Sins of commission
- sins of omission
- sins of mission
- Declarations in C