Chapter 63: Common pitfalls

This section discusses some of the common mistakes that a C programmer should be aware of and should avoid making. For more on some unexpected problems and their causes, please see Undefined behavior

Section 63.1: Mixing signed and unsigned integers in arithmetic operations

It is usually not a good idea to mix `signed` and `unsigned` integers in arithmetic operations. For example, what will be output of following example?

```
#include <stdio.h>

int main(void)
{
    unsigned int a = 1000;
    signed int b = -1;

    if (a > b) puts("a is more than b");
    else puts("a is less or equal than b");

    return 0;
}
```

Since 1000 is more than -1 you would expect the output to be `a is more than b`, however that will not be the case.

Arithmetic operations between different integral types are performed within a common type defined by the so called usual arithmetic conversions (see the language specification, 6.3.1.8).

In this case the "common type" is `unsigned int`, Because, as stated in Usual arithmetic conversions,

> 714 Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type.

This means that `int` operand `b` will get converted to `unsigned int` before the comparison.

When -1 is converted to an `unsigned int` the result is the maximal possible `unsigned int` value, which is greater than 1000, meaning that `a > b` is false.

Section 63.2: Macros are simple string replacements

Macros are simple string replacements. (Strictly speaking, they work with preprocessing tokens, not arbitrary strings.)

```
#include <stdio.h>

#define SQUARE(x) x*x

int main(void) {
    printf("%d\n", SQUARE(1+2));
    return 0;
}
```

You may expect this code to print `9` (`3*3`), but actually `5` will be printed because the macro will be expanded to `1+2*1+2`.

You should wrap the arguments and the whole macro expression in parentheses to avoid this problem.

```
#include <stdio.h>

#define SQUARE(x) ((x)*(x))

int main(void) {
    printf("%d\n", SQUARE(1+2));
    return 0;
}
```

Another problem is that the arguments of a macro are not guaranteed to be evaluated once; they may not be evaluated at all, or may be evaluated multiple times.

```
#include <stdio.h>

#define MIN(x, y) ((x) <= (y) ? (x) : (y))

int main(void) {
    int a = 0;
    printf("%d\n", MIN(a++, 10));
    printf("a = %d\n", a);
    return 0;
}
```

In this code, the macro will be expanded to `((a++) <= (10) ? (a++) : (10))`. Since `a++` (`0`) is smaller than `10`, `a++` will be evaluated twice and it will make the value of `a` and what is returned from `MIN` differ from you may expect.

This can be avoided by using functions, but note that the types will be fixed by the function definition, whereas macros can be (too) flexible with types.

```
#include <stdio.h>

int min(int x, int y) {
    return x <= y ? x : y;
}

int main(void) {
    int a = 0;
    printf("%d\n", min(a++, 10));
    printf("a = %d\n", a);
    return 0;
}
```

Now the problem of double-evaluation is fixed, but this `min` function cannot deal with `double` data without truncating, for example.

Macro directives can be of two types:

```
#define OBJECT_LIKE_MACRO     followed by a "replacement list" of preprocessor tokens
#define FUNCTION_LIKE_MACRO(with, arguments) followed by a replacement list
```

What distinguishes these two types of macros is the character that follows the identifier after `#define`: if it's an *lparen*, it is a function-like macro; otherwise, it's an object-like macro. If the intention is to write a function-like

macro, there must not be any white space between the end of the name of the macro and (. Check this for a detailed explanation.

Version ≥ C99

In C99 or later, you could use `static inline int min(int x, int y) { ... }`.

Version ≥ C11

In C11, you could write a 'type-generic' expression for `min`.

```
#include <stdio.h>

#define min(x, y) _Generic((x), \
                        long double: min_ld, \
                        unsigned long long: min_ull, \
                        default: min_i \
                        )(x, y)

#define gen_min(suffix, type) \
    static inline type min_##suffix(type x, type y) { return (x < y) ? x : y; }

gen_min(ld, long double)
gen_min(ull, unsigned long long)
gen_min(i, int)

int main(void)
{
    unsigned long long ull1 = 50ULL;
    unsigned long long ull2 = 37ULL;
    printf("min(%llu, %llu) = %llu\n", ull1, ull2, min(ull1, ull2));
    long double ld1 = 3.141592653L;
    long double ld2 = 3.141592652L;
    printf("min(%.10Lf, %.10Lf) = %.10Lf\n", ld1, ld2, min(ld1, ld2));
    int i1 = 3141653;
    int i2 = 3141652;
    printf("min(%d, %d) = %d\n", i1, i2, min(i1, i2));
    return 0;
}
```

The generic expression could be extended with more types such as `double`, `float`, `long long`, `unsigned long`, `long`, `unsigned` — and appropriate `gen_min` macro invocations written.

Section 63.3: Forgetting to copy the return value of realloc into a temporary

If `realloc` fails, it returns `NULL`. If you assign the value of the original buffer to `realloc`'s return value, and if it returns `NULL`, then the original buffer (the old pointer) is lost, resulting in a *memory leak*. The solution is to copy into a temporary pointer, and if that temporary is not NULL, **then** copy into the real buffer.

```
char *buf, *tmp;

buf = malloc(...);
...

/* WRONG */
if ((buf = realloc(buf, 16)) == NULL)
    perror("realloc");
```

```c
/* RIGHT */
if ((tmp = realloc(buf, 16)) != NULL)
    buf = tmp;
else
    perror("realloc");
```

Section 63.4: Forgetting to allocate one extra byte for \0

When you are copying a string into a `malloc`ed buffer, always remember to add 1 to `strlen`.

```c
char *dest = malloc(strlen(src)); /* WRONG */
char *dest = malloc(strlen(src) + 1); /* RIGHT */

strcpy(dest, src);
```

This is because `strlen` does not include the trailing `\0` in the length. If you take the `WRONG` (as shown above) approach, upon calling `strcpy`, your program would invoke undefined behaviour.

It also applies to situations when you are reading a string of known maximum length from `stdin` or some other source. For example

```c
#define MAX_INPUT_LEN 42

char buffer[MAX_INPUT_LEN]; /* WRONG */
char buffer[MAX_INPUT_LEN + 1]; /* RIGHT */

scanf("%42s", buffer);  /* Ensure that the buffer is not overflowed */
```

Section 63.5: Misunderstanding array decay

A common problem in code that uses multidimensional arrays, arrays of pointers, etc. is the fact that `Type**` and `Type[M][N]` are fundamentally different types:

```c
#include <stdio.h>

void print_strings(char **strings, size_t n)
{
    size_t i;
    for (i = 0; i < n; i++)
        puts(strings[i]);
}

int main(void)
{
    char s[4][20] = {"Example 1", "Example 2", "Example 3", "Example 4"};
    print_strings(s, 4);
    return 0;
}
```

Sample compiler output:

```
file1.c: In function 'main':
file1.c:13:23: error: passing argument 1 of 'print_strings' from incompatible pointer type [-
Wincompatible-pointer-types]
         print_strings(strings, 4);
                       ^
file1.c:3:10: note: expected 'char **' but argument is of type 'char (*)[20]'
```

```
    void print_strings(char **strings, size_t n)
```

The error states that the `s` array in the `main` function is passed to the function `print_strings`, which expects a different pointer type than it received. It also includes a note expressing the type that is expected by `print_strings` and the type that was passed to it from `main`.

The problem is due to something called *array decay*. What happens when `s` with its type `char[4][20]` (array of 4 arrays of 20 chars) is passed to the function is it turns into a pointer to its first element as if you had written `&s[0]`, which has the type `char (*)[20]` (pointer to 1 array of 20 chars). This occurs for any array, including an array of pointers, an array of arrays of arrays (3-D arrays), and an array of pointers to an array. Below is a table illustrating what happens when an array decays. Changes in the type description are highlighted to illustrate what happens:

Before Decay		After Decay	
`char [20]`	**array of (20 chars)**	`char *`	**pointer to (1 char)**
`char [4][20]`	**array of (4 arrays** of 20 chars)	`char (*)[20]`	**pointer to (1 array** of 20 chars)
`char *[4]`	**array of (4 pointers** to 1 char)	`char **`	**pointer to (1 pointer** to 1 char)
`char [3][4][20]`	**array of (3 arrays** of 4 arrays of 20 chars)	`char (*)[4][20]`	**pointer to (1 array** of 4 arrays of 20 chars)
`char (*[4])[20]`	**array of (4 pointers** to 1 array of 20 chars)	`char (**)[20]`	**pointer to (1 pointer** to 1 array of 20 chars)

If an array can decay to a pointer, then it can be said that a pointer may be considered an array of at least 1 element. An exception to this is a null pointer, which points to nothing and is consequently not an array.

Array decay only happens once. If an array has decayed to a pointer, it is now a pointer, not an array. Even if you have a pointer to an array, remember that the pointer might be considered an array of at least one element, so array decay has already occurred.

In other words, a pointer to an array (`char (*)[20]`) will never become a pointer to a pointer (`char **`). To fix the `print_strings` function, simply make it receive the correct type:

```
void print_strings(char (*strings)[20], size_t n)
/* OR */
void print_strings(char strings[][20], size_t n)
```

A problem arises when you want the `print_strings` function to be generic for any array of chars: what if there are 30 chars instead of 20? Or 50? The answer is to add another parameter before the array parameter:

```
#include <stdio.h>

/*
 * Note the rearranged parameters and the change in the parameter name
 * from the previous definitions:
 *      n (number of strings)
 *   => scount (string count)
 *
 * Of course, you could also use one of the following highly recommended forms
 * for the `strings` parameter instead:
 *
 *    char strings[scount][ccount]
 *    char strings[][ccount]
 */
void print_strings(size_t scount, size_t ccount, char (*strings)[ccount])
{
    size_t i;
    for (i = 0; i < scount; i++)
```

```
        puts(strings[i]);
}

int main(void)
{
    char s[4][20] = {"Example 1", "Example 2", "Example 3", "Example 4"};
    print_strings(4, 20, s);
    return 0;
}
```

Compiling it produces no errors and results in the expected output:

```
Example 1
Example 2
Example 3
Example 4
```

Section 63.6: Forgetting to free memory (memory leaks)

A programming best practice is to free any memory that has been allocated directly by your own code, or implicitly by calling an internal or external function, such as a library API like `strdup()`. Failing to free memory can introduce a memory leak, which could accumulate into a substantial amount of wasted memory that is unavailable to your program (or the system), possibly leading to crashes or undefined behavior. Problems are more likely to occur if the leak is incurred repeatedly in a loop or recursive function. The risk of program failure increases the longer a leaking program runs. Sometimes problems appear instantly; other times problems won't be seen for hours or even years of constant operation. Memory exhaustion failures can be catastrophic, depending on the circumstances.

The following infinite loop is an example of a leak that will eventually exhaust available memory leak by calling `getline()`, a function that implicitly allocates new memory, without freeing that memory.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char *line = NULL;
    size_t size = 0;

    /* The loop below leaks memory as fast as it can */

    for(;;) {
        getline(&line, &size, stdin); /* New memory implicitly allocated */

        /* <do whatever> */

        line = NULL;
    }

    return 0;
}
```

In contrast, the code below also uses the `getline()` function, but this time, the allocated memory is correctly freed, avoiding a leak.

```
#include <stdlib.h>
#include <stdio.h>
```

```c
int main(void)
{
    char *line = NULL;
    size_t size = 0;

    for(;;) {
        if (getline(&line, &size, stdin) < 0) {
            free(line);
            line = NULL;

            /* Handle failure such as setting flag, breaking out of loop and/or exiting */
        }

        /* <do whatever> */

        free(line);
        line = NULL;

    }

    return 0;
}
```

Leaking memory doesn't always have tangible consequences and isn't necessarily a functional problem. While "best practice" dictates rigorously freeing memory at strategic points and conditions, to reduce memory footprint and lower risk of memory exhaustion, there can be exceptions. For example, if a program is bounded in duration and scope, the risk of allocation failure might be considered too small to worry about. In that case, bypassing explicit deallocation might be considered acceptable. For example, most modern operating systems automatically free all memory consumed by a program when it terminates, whether it is due to program failure, a system call to `exit()`, process termination, or reaching end of `main()`. Explicitly freeing memory at the point of imminent program termination could actually be redundant or introduce a performance penalty.

Allocation can fail if insufficient memory is available, and handling failures should be accounted for at appropriate levels of the call stack. `getline()`, shown above is an interesting use-case because it is a library function that not only allocates memory it leaves to the caller to free, but can fail for a number of reasons, all of which must be taken into account. Therefore, it is essential when using a C API, to read the *documentation (man page)* and pay particular attention to error conditions and memory usage, and be aware which software layer bears the burden of freeing returned memory.

Another common memory handling practice is to consistently set memory pointers to NULL immediately after the memory referenced by those pointers is freed, so those pointers can be tested for validity at any time (e.g. checked for NULL / non-NULL), because accessing freed memory can lead to severe problems such as getting garbage data (read operation), or data corruption (write operation) and/or a program crash. In most modern operating systems, freeing memory location 0 (`NULL`) is a NOP (e.g. it is harmless), as required by the C standard — so by setting a pointer to NULL, there is no risk of double-freeing memory if the pointer is passed to `free()`. Keep in mind that double-freeing memory can lead to very time consuming, confusing, and *difficult to diagnose* failures.

Section 63.7: Copying too much

```c
char buf[8]; /* tiny buffer, easy to overflow */

printf("What is your name?\n");
scanf("%s", buf); /* WRONG */
scanf("%7s", buf); /* RIGHT */
```

If the user enters a string longer than 7 characters (- 1 for the null terminator), memory behind the buffer `buf` will

be overwritten. This results in undefined behavior. Malicious hackers often exploit this in order to overwrite the return address, and change it to the address of the hacker's malicious code.

Section 63.8: Mistakenly writing = instead of == when comparing

The = operator is used for assignment.

The == operator is used for comparison.

One should be careful not to mix the two. Sometimes one mistakenly writes

```
/* assign y to x */
if (x = y) {
     /* logic */
}
```

when what was really wanted is:

```
/* compare if x is equal to y */
if (x == y) {
    /* logic */
}
```

The former assigns value of y to x and checks if that value is non zero, instead of doing comparison, which is equivalent to:

```
if ((x = y) != 0) {
    /* logic */
}
```

There are times when testing the result of an assignment is intended and is commonly used, because it avoids having to duplicate code and having to treat the first time specially. Compare

```
while ((c = getopt_long(argc, argv, short_options, long_options, &option_index)) != -1) {
        switch (c) {
        ...
        }
}
```

versus

```
c = getopt_long(argc, argv, short_options, long_options, &option_index);
while (c != -1) {
        switch (c) {
        ...
        }
        c = getopt_long(argc, argv, short_options, long_options, &option_index);
}
```

Modern compilers will recognise this pattern and do not warn when the assignment is inside parenthesis like above, but may warn for other usages. For example:

```
if (x = y)          /* warning */

if ((x = y))        /* no warning */
```

```
if ((x = y) != 0)  /* no warning; explicit */
```

Some programmers use the strategy of putting the constant to the left of the operator (commonly called Yoda conditions). Because constants are rvalues, this style of condition will cause the compiler to throw an error if the wrong operator was used.

```
if (5 = y) /* Error */

if (5 == y) /* No error */
```

However, this severely reduces the readability of the code and is not considered necessary if the programmer follows good C coding practices, and doesn't help when comparing two variables so it isn't a universal solution. Furthermore, many modern compilers may give warnings when code is written with Yoda conditions.

Section 63.9: Newline character is not consumed in typical scanf() call

When this program

```
#include <stdio.h>
#include <string.h>

int main(void) {
    int num = 0;
    char str[128], *lf;

    scanf("%d", &num);
    fgets(str, sizeof(str), stdin);

    if ((lf = strchr(str, '\n')) != NULL) *lf = '\0';
    printf("%d \"%s\"\n", num, str);
    return 0;
}
```

is executed with this input

```
42
life
```

the output will be `42 ""` instead of expected `42 "life"`.

This is because a newline character after `42` is not consumed in the call of `scanf()` and it is consumed by `fgets()` before it reads `life`. Then, `fgets()` stop reading before reading `life`.

To avoid this problem, one way that is useful when the maximum length of a line is known -- when solving problems in online judge syste, for example -- is avoiding using `scanf()` directly and reading all lines via `fgets()`. You can use `sscanf()` to parse the lines read.

```
#include <stdio.h>
#include <string.h>

int main(void) {
    int num = 0;
    char line_buffer[128] = "", str[128], *lf;

    fgets(line_buffer, sizeof(line_buffer), stdin);
    sscanf(line_buffer, "%d", &num);
```

```
    fgets(str, sizeof(str), stdin);

    if ((lf = strchr(str, '\n')) != NULL) *lf = '\0';
    printf("%d \"%s\"\n", num, str);
    return 0;
}
```

Another way is to read until you hit a newline character after using `scanf()` and before using `fgets()`.

```
#include <stdio.h>
#include <string.h>

int main(void) {
    int num = 0;
    char str[128], *lf;
    int c;

    scanf("%d", &num);
    while ((c = getchar()) != '\n' && c != EOF);
    fgets(str, sizeof(str), stdin);

    if ((lf = strchr(str, '\n')) != NULL) *lf = '\0';
    printf("%d \"%s\"\n", num, str);
    return 0;
}
```

Section 63.10: Adding a semicolon to a #define

It is easy to get confused in the C preprocessor, and treat it as part of C itself, but that is a mistake because the preprocessor is just a text substitution mechanism. For example, if you write

```
/* WRONG */
#define MAX 100;
int arr[MAX];
```

the code expands to

```
int arr[100;];
```

which is a syntax error. The remedy is to remove the semicolon from the `#define` line. It is almost invariably a mistake to end a `#define` with a semicolon.

Section 63.11: Incautious use of semicolons

Be careful with semicolons. Following example

```
if (x > a);
   a = x;
```

actually means:

```
if (x > a) {}
a = x;
```

which means x will be assigned to a in any case, which might not be what you wanted originally.

Sometimes, missing a semicolon will also cause an unnoticeable problem:

```
if (i < 0)
    return
day = date[0];
hour = date[1];
minute = date[2];
```

The semicolon behind return is missed, so day=date[0] will be returned.

One technique to avoid this and similar problems is to always use braces on multi-line conditionals and loops. For example:

```
if (x > a) {
    a = x;
}
```

Section 63.12: Undefined reference errors when linking

One of the most common errors in compilation happens during the linking stage. The error looks similar to this:

```
$ gcc undefined_reference.c
/tmp/ccoXhwF0.o: In function `main':
undefined_reference.c:(.text+0x15): undefined reference to `foo'
collect2: error: ld returned 1 exit status
$
```

So let's look at the code that generated this error:

```
int foo(void);

int main(int argc, char **argv)
{
    int foo_val;
    foo_val = foo();
    return foo_val;
}
```

We see here a *declaration* of foo (`int foo();`) but no *definition* of it (actual function). So we provided the compiler with the function header, but there was no such function defined anywhere, so the compilation stage passes but the linker exits with an `Undefined reference` error.
To fix this error in our small program we would only have to add a *definition* for foo:

```
/* Declaration of foo */
int foo(void);

/* Definition of foo */
int foo(void)
{
    return 5;
}

int main(int argc, char **argv)
{
    int foo_val;
    foo_val = foo();
    return foo_val;
}
```

Now this code will compile. An alternative situation arises where the source for `foo()` is in a separate source file `foo.c` (and there's a header `foo.h` to declare `foo()` that is included in both `foo.c` and `undefined_reference.c`). Then the fix is to link both the object file from `foo.c` and `undefined_reference.c`, or to compile both the source files:

```
$ gcc -c undefined_reference.c
$ gcc -c foo.c
$ gcc -o working_program undefined_reference.o foo.o
$
```

Or:

```
$ gcc -o working_program undefined_reference.c foo.c
$
```

A more complex case is where libraries are involved, like in the code:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc, char **argv)
{
    double first;
    double second;
    double power;

    if (argc != 3)
    {
        fprintf(stderr, "Usage: %s <denom> <nom>\n", argv[0]);
        return EXIT_FAILURE;
    }

    /* Translate user input to numbers, extra error checking
     * should be done here. */
    first = strtod(argv[1], NULL);
    second = strtod(argv[2], NULL);

    /* Use function pow() from libm - this will cause a linkage
     * error unless this code is compiled against libm! */
    power = pow(first, second);

    printf("%f to the power of %f = %f\n", first, second, power);

    return EXIT_SUCCESS;
}
```

The code is syntactically correct, declaration for `pow()` exists from `#include <math.h>`, so we try to compile and link but get an error like this:

```
$ gcc no_library_in_link.c -o no_library_in_link
/tmp/ccduQQqA.o: In function `main':
no_library_in_link.c:(.text+0x8b): undefined reference to `pow'
collect2: error: ld returned 1 exit status
$
```

This happens because the *definition* for `pow()` wasn't found during the linking stage. To fix this we have to specify we want to link against the math library called `libm` by specifying the `-lm` flag. (Note that there are platforms such

as macOS where `-lm` is not needed, but when you get the undefined reference, the library is needed.)

So we run the compilation stage again, this time specifying the library (after the source or object files):

```
$ gcc no_library_in_link.c -lm -o library_in_link_cmd
$ ./library_in_link_cmd 2 4
2.000000 to the power of 4.000000 = 16.000000
$
```

And it works!

Section 63.13: Checking logical expression against 'true'

The original C standard had no intrinsic Boolean type, so `bool`, **`true`** and **`false`** had no inherent meaning and were often defined by programmers. Typically **`true`** would be defined as 1 and **`false`** would be defined as 0.

Version ≥ C99

C99 adds the built-in type `_Bool` and the header **`<stdbool.h>`** which defines `bool` (expanding to `_Bool`), **`false`** and **`true`**. It also allows you to redefine `bool`, **`true`** and **`false`**, but notes that this is an obsolescent feature.

More importantly, logical expressions treat anything that evaluates to zero as false and any non-zero evaluation as true. For example:

```
/* Return 'true' if the most significant bit is set */
bool isUpperBitSet(uint8_t bitField)
{
    if ((bitField & 0x80) == true)  /* Comparison only succeeds if true is 0x80 and bitField has
that bit set */
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

In the above example, the function is trying to check if the upper bit is set and return **`true`** if it is. However, by explicitly checking against **`true`**, the `if` statement will only succeed if `(bitfield & 0x80)` evaluates to whatever **`true`** is defined as, which is typically `1` and very seldom `0x80`. Either explicitly check against the case you expect:

```
/* Return 'true' if the most significant bit is set */
bool isUpperBitSet(uint8_t bitField)
{
    if ((bitField & 0x80) == 0x80) /* Explicitly test for the case we expect */
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Or evaluate any non-zero value as true.

```c
/* Return 'true' if the most significant bit is set */
bool isUpperBitSet(uint8_t bitField)
{
    /* If upper bit is set, result is 0x80 which the if will evaluate as true */
    if (bitField & 0x80)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Section 63.14: Doing extra scaling in pointer arithmetic

In pointer arithmetic, the integer to be added or subtracted to pointer is interpreted not as change of *address* but as number of *elements* to move.

```c
#include <stdio.h>

int main(void) {
    int array[] = {1, 2, 3, 4, 5};
    int *ptr = &array[0];
    int *ptr2 = ptr + sizeof(int) * 2; /* wrong */
    printf("%d %d\n", *ptr, *ptr2);
    return 0;
}
```

This code does extra scaling in calculating pointer assigned to `ptr2`. If `sizeof(int)` is 4, which is typical in modern 32-bit environments, the expression stands for "8 elements after `array[0]`", which is out-of-range, and it invokes *undefined behavior*.

To have `ptr2` point at what is 2 elements after `array[0]`, you should simply add 2.

```c
#include <stdio.h>

int main(void) {
    int array[] = {1, 2, 3, 4, 5};
    int *ptr = &array[0];
    int *ptr2 = ptr + 2;
    printf("%d %d\n", *ptr, *ptr2); /* "1 3" will be printed */
    return 0;
}
```

Explicit pointer arithmetic using additive operators may be confusing, so using array subscripting may be better.

```c
#include <stdio.h>

int main(void) {
    int array[] = {1, 2, 3, 4, 5};
    int *ptr = &array[0];
    int *ptr2 = &ptr[2];
    printf("%d %d\n", *ptr, *ptr2); /* "1 3" will be printed */
    return 0;
}
```

`E1[E2]` is identical to `(*((E1)+(E2)))` (N1570 6.5.2.1, paragraph 2), and `&(E1[E2])` is equivalent to `((E1)+(E2))`

(N1570 6.5.3.2, footnote 102).

Alternatively, if pointer arithmetic is preferred, casting the pointer to address a different data type can allow byte addressing. Be careful though: endianness can become an issue, and casting to types other than 'pointer to character' leads to strict aliasing problems.

```
#include <stdio.h>

int main(void) {
    int array[3] = {1,2,3};  // 4 bytes * 3 allocated
    unsigned char *ptr = (unsigned char *) array;  // unsigned chars only take 1 byte
    /*
     * Now any pointer arithmetic on ptr will match
     * bytes in memory.  ptr can be treated like it
     * was declared as: unsigned char ptr[12];
     */

    return 0;
}
```

Section 63.15: Multi-line comments cannot be nested

In C, multi-line comments, /* and */, do not nest.

If you annotate a block of code or function using this style of comment:

```
/*
 * max(): Finds the largest integer in an array and returns it.
 * If the array length is less than 1, the result is undefined.
 * arr: The array of integers to search.
 * num: The number of integers in arr.
 */
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}
```

You will not be able to comment it out easily:

```
//Trying to comment out the block...
/*

/*
 * max(): Finds the largest integer in an array and returns it.
 * If the array length is less than 1, the result is undefined.
 * arr: The array of integers to search.
 * num: The number of integers in arr.
 */
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
```

```
}

//Causes an error on the line below...
*/
```

One solution is to use C99 style comments:

```
// max(): Finds the largest integer in an array and returns it.
// If the array length is less than 1, the result is undefined.
// arr: The array of integers to search.
// num: The number of integers in arr.
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}
```

Now the entire block can be commented out easily:

```
/*

// max(): Finds the largest integer in an array and returns it.
// If the array length is less than 1, the result is undefined.
// arr: The array of integers to search.
// num: The number of integers in arr.
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}

*/
```

Another solution is to avoid disabling code using comment syntax, using `#ifdef` or `#ifndef` preprocessor directives instead. These directives *do* nest, leaving you free to comment your code in the style you prefer.

```
#define DISABLE_MAX /* Remove or comment this line to enable max() code block */

#ifdef DISABLE_MAX
/*
 * max(): Finds the largest integer in an array and returns it.
 * If the array length is less than 1, the result is undefined.
 * arr: The array of integers to search.
 * num: The number of integers in arr.
 */
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}
```

```
#endif
```

Some guides go so far as to recommend that code sections must *never* be commented and that if code is to be temporarily disabled one could resort to using an `#if 0` directive.

See #if 0 to block out code sections.

Section 63.16: Ignoring return values of library functions

Almost every function in C standard library returns something on success, and something else on error. For example, `malloc` will return a pointer to the memory block allocated by the function on success, and, if the function failed to allocate the requested block of memory, a null pointer. So you should always check the return value for easier debugging.

This is bad:

```
char* x = malloc(100000000000UL * sizeof *x);
/* more code */
scanf("%s", x); /* This might invoke undefined behaviour and if lucky causes a segmentation
violation, unless your system has a lot of memory */
```

This is good:

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char* x = malloc(100000000000UL * sizeof *x);
    if (x == NULL) {
        perror("malloc() failed");
        exit(EXIT_FAILURE);
    }

    if (scanf("%s", x) != 1) {
        fprintf(stderr, "could not read string\n");
        free(x);
        exit(EXIT_FAILURE);
    }

    /* Do stuff with x. */

    /* Clean up. */
    free(x);

    return EXIT_SUCCESS;
}
```

This way you know right away the cause of error, otherwise you might spend hours looking for a bug in a completely wrong place.

Section 63.17: Comparing floating point numbers

Floating point types (`float`, `double` and `long double`) cannot precisely represent some numbers because they have finite precision and represent the values in a binary format. Just like we have repeating decimals in base 10 for fractions such as 1/3, there are fractions that cannot be represented finitely in binary too (such as 1/3, but also, more importantly, 1/10). Do not directly compare floating point values; use a delta instead.

```c
#include <float.h> // for DBL_EPSILON and FLT_EPSILON
#include <math.h>  // for fabs()

int main(void)
{
    double a = 0.1; // imprecise: (binary) 0.000110...

    // may be false or true
    if (a + a + a + a + a + a + a + a + a + a == 1.0) {
        printf("10 * 0.1 is indeed 1.0. This is not guaranteed in the general case.\n");
    }

    // Using a small delta value.
    if (fabs(a + a + a + a + a + a + a + a + a + a - 1.0) < 0.000001) {
        // C99 5.2.4.2.2p8 guarantees at least 10 decimal digits
        // of precision for the double type.
        printf("10 * 0.1 is almost 1.0.\n");
    }

    return 0;
}
```

Another example:

```c
gcc -O3   -g   -I./inc   -std=c11   -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes
 -Wold-style-definition       rd11.c -o rd11 -L./lib -lsoq
#include <stdio.h>
#include <math.h>

static inline double rel_diff(double a, double b)
{
    return fabs(a - b) / fmax(fabs(a), fabs(b));
}

int main(void)
{
    double d1 = 3.14159265358979;
    double d2 = 355.0 / 113.0;

    double epsilon = 1.0;
    for (int i = 0; i < 10; i++)
    {
        if (rel_diff(d1, d2) < epsilon)
            printf("%d:%.10f <=> %.10f within tolerance %.10f (rel diff %.4E)\n",
                   i, d1, d2, epsilon, rel_diff(d1, d2));
        else
            printf("%d:%.10f <=> %.10f out of tolerance %.10f (rel diff %.4E)\n",
                   i, d1, d2, epsilon, rel_diff(d1, d2));
        epsilon /= 10.0;
    }
    return 0;
}
```

Output:

```
0:3.1415926536 <=> 3.1415929204 within tolerance 1.0000000000 (rel diff 8.4914E-08)
1:3.1415926536 <=> 3.1415929204 within tolerance 0.1000000000 (rel diff 8.4914E-08)
2:3.1415926536 <=> 3.1415929204 within tolerance 0.0100000000 (rel diff 8.4914E-08)
3:3.1415926536 <=> 3.1415929204 within tolerance 0.0010000000 (rel diff 8.4914E-08)
4:3.1415926536 <=> 3.1415929204 within tolerance 0.0001000000 (rel diff 8.4914E-08)
```

```
5:3.1415926536 <=> 3.1415929204 within tolerance 0.0000100000 (rel diff 8.4914E-08)
6:3.1415926536 <=> 3.1415929204 within tolerance 0.0000010000 (rel diff 8.4914E-08)
7:3.1415926536 <=> 3.1415929204 within tolerance 0.0000001000 (rel diff 8.4914E-08)
8:3.1415926536 <=> 3.1415929204 out of tolerance 0.0000000100 (rel diff 8.4914E-08)
9:3.1415926536 <=> 3.1415929204 out of tolerance 0.0000000010 (rel diff 8.4914E-08)
```

Section 63.18: Floating point literals are of type double by default

Care must be taken when initializing variables of type `float` to literal values or comparing them with literal values, because regular floating point literals like `0.1` are of type `double`. This may lead to surprises:

```
#include <stdio.h>
int main() {
    float  n;
    n = 0.1;
    if (n > 0.1) printf("Wierd\n");
    return 0;
}
// Prints "Wierd" when n is float
```

Here, `n` gets initialized and rounded to single precision, resulting in value 0.10000000149011612. Then, `n` is converted back to double precision to be compared with `0.1` literal (which equals to 0.10000000000000001), resulting in a mismatch.

Besides rounding errors, mixing `float` variables with `double` literals will result in poor performance on platforms which don't have hardware support for double precision.

Section 63.19: Using character constants instead of string literals, and vice versa

In C, character constants and string literals are different things.

A character surrounded by single quotes like `'a'` is a *character constant*. A character constant is an integer whose value is the character code that stands for the character. How to interpret character constants with multiple characters like `'abc'` is implementation-defined.

Zero or more characters surrounded by double quotes like `"abc"` is a *string literal*. A string literal is an unmodifiable array whose elements are type `char`. The string in the double quotes plus terminating null-character are the contents, so `"abc"` has 4 elements (`{'a', 'b', 'c', '\0'}`)

In this example, a character constant is used where a string literal should be used. This character constant will be converted to a pointer in an implementation-defined manner and there is little chance for the converted pointer to be valid, so this example will invoke *undefined behavior*.

```
#include <stdio.h>

int main(void) {
    const char *hello = 'hello, world'; /* bad */
    puts(hello);
    return 0;
}
```

In this example, a string literal is used where a character constant should be used. The pointer converted from the string literal will be converted to an integer in an implementation-defined manner, and it will be converted to `char`

in an implementation-defined manner. (How to convert an integer to a signed type which cannot represent the value to convert is implementation-defined, and whether `char` is signed is also implementation-defined.) The output will be some meaningless thing.

```
#include <stdio.h>

int main(void) {
    char c = "a"; /* bad */
    printf("%c\n", c);
    return 0;
}
```

In almost all cases, the compiler will complain about these mix-ups. If it doesn't, you need to use more compiler warning options, or it is recommended that you use a better compiler.

Section 63.20: Recursive function — missing out the base condition

Calculating the factorial of a number is a classic example of a recursive function.

Missing the Base Condition:

```
#include <stdio.h>

int factorial(int n)
{
       return n * factorial(n - 1);
}

int main()
{
    printf("Factorial %d = %d\n", 3, factorial(3));
    return 0;
}
```

Typical output: `Segmentation fault: 11`

The problem with this function is it would loop infinitely, causing a segmentation fault — it needs a base condition to stop the recursion.

Base Condition Declared:

```
#include <stdio.h>

int factorial(int n)
{
    if (n == 1) // Base Condition, very crucial in designing the recursive functions.
    {
       return 1;
    }
    else
    {
       return n * factorial(n - 1);
    }
}

int main()
{
```

```
    printf("Factorial %d = %d\n", 3, factorial(3));
    return 0;
}
```

Sample output

```
Factorial 3 = 6
```

This function will terminate as soon as it hits the condition `n` is equal to 1 (provided the initial value of `n` is small enough — the upper bound is `12` when `int` is a 32-bit quantity).

Rules to be followed:

1. Initialize the algorithm. Recursive programs often need a seed value to start with. This is accomplished either by using a parameter passed to the function or by providing a gateway function that is non-recursive but that sets up the seed values for the recursive calculation.
2. Check to see whether the current value(s) being processed match the base case. If so, process and return the value.
3. Redefine the answer in terms of a smaller or simpler sub-problem or sub-problems.
4. Run the algorithm on the sub-problem.
5. Combine the results in the formulation of the answer.
6. Return the results.

Source: Recursive Function

Section 63.21: Overstepping array boundaries

Arrays are zero-based, that is the index always starts at 0 and ends with index array length minus 1. Thus the following code will not output the first element of the array and will output garbage for the final value that it prints.

```
#include <stdio.h>

int main(void)
{
    int x = 0;
    int myArray[5] = {1, 2, 3, 4, 5}; //Declaring 5 elements

    for(x = 1; x <= 5; x++) //Looping from 1 till 5.
       printf("%d\t", myArray[x]);

    printf("\n");
    return 0;
}
```

Output: `2 3 4 5 GarbageValue`

The following demonstrates the correct way to achieve the desired output:

```
#include <stdio.h>

int main(void)
{
    int x = 0;
    int myArray[5] = {1, 2, 3, 4, 5}; //Declaring 5 elements

    for(x = 0; x < 5; x++) //Looping from 0 till 4.
       printf("%d\t", myArray[x]);
```

```
    printf("\n");
    return 0;
}
```

Output: `1 2 3 4 5`

It is important to know the length of an array before working with it as otherwise you may corrupt the buffer or cause a segmentation fault by accessing memory locations that are out of bounds.

Section 63.22: Passing unadjacent arrays to functions expecting "real" multidimensional arrays

When allocating multidimensional arrays with `malloc`, `calloc`, and `realloc`, a common pattern is to allocate the inner arrays with multiple calls (even if the call only appears once, it may be in a loop):

```
/* Could also be `int **` with malloc used to allocate outer array. */
int *array[4];
int i;

/* Allocate 4 arrays of 16 ints. */
for (i = 0; i < 4; i++)
    array[i] = malloc(16 * sizeof(*array[i]));
```

The difference in bytes between the last element of one of the inner arrays and the first element of the next inner array may not be 0 as they would be with a "real" multidimensional array (e.g. `int array[4][16];`):

```
/* 0x40003c, 0x402000 */
printf("%p, %p\n", (void *)(array[0] + 15), (void *)array[1]);
```

Taking into account the size of `int`, you get a difference of 8128 bytes (8132-4), which is 2032 `int`-sized array elements, and that is the problem: a "real" multidimensional array has no gaps between elements.

If you need to use a dynamically allocated array with a function expecting a "real" multidimensional array, you should allocate an object of type `int *` and use arithmetic to perform calculations:

```
void func(int M, int N, int *array);
...

/* Equivalent to declaring `int array[M][N] = {{0}};` and assigning to array4_16[i][j]. */
int *array;
int M = 4, N = 16;
array = calloc(M, N * sizeof(*array));
array[i * N + j] = 1;
func(M, N, array);
```

If `N` is a macro or an integer literal rather than a variable, the code can simply use the more natural 2-D array notation after allocating a pointer to an array:

```
void func(int M, int N, int *array);
#define N 16
void func_N(int M, int (*array)[N]);
...

int M = 4;
int (*array)[N];
array = calloc(M, sizeof(*array));
array[i][j] = 1;
```

```
/* Cast to `int *` works here because `array` is a single block of M*N ints with no gaps,
   just like `int array2[M * N];` and `int array3[M][N];` would be. */
func(M, N, (int *)array);
func_N(M, array);
```

Version ≥ C99

If `N` is not a macro or an integer literal, then `array` will point to a variable-length array (VLA). This can still be used with `func` by casting to `int *` and a new function `func_vla` would replace `func_N`:

```
void func(int M, int N, int *array);
void func_vla(int M, int N, int array[M][N]);
...

int M = 4, N = 16;
int (*array)[N];
array = calloc(M, sizeof(*array));
array[i][j] = 1;
func(M, N, (int *)array);
func_vla(M, N, array);
```

Version ≥ C11

Note: VLAs are optional as of C11. If your implementation supports C11 and defines the macro `__STDC_NO_VLA__` to 1, you are stuck with the pre-C99 methods.