# C Programming Notes, Tips & Tricks

Lorenzo Conti
lore.conti.g<at>gmail.com

2006

# Contents

# Chapter 1

# Introduction

"Hey, it is not nice, but it works!"

This is a collection of notes, examples, tips & tricks on C programming language I put together some time ago just for my reference to remember the lesson learned. Some notes and examples are mine, some I found in books and web pages I read (see references at the end of the document, all rights reserved to the relevant owner).
Something is obsolete, something still a good example. Take it as it is, if you want.
I hold no responsibility for any damage or failure caused by using this document. It is very likely that the contents of the document might change from time to time and might even be mistaken at some places.
Use this document as per your own discretion.

Lorenzo

# Chapter 2

# Optimization

## 2.1 Timing your program

To measure the time consumed by a program you can use "time":

```
time myprogram
```

Or, programmatically, call "clock()" which calculates the best available approximation of the cumulative amount of time used by your program since it started. To convert the result into seconds, divide by the macro "CLOCKS_PER_SEC".

## 2.2 Integers 1

> If you know the value will never be negative use unsigned integers instead of integers.

Some processors can handle unsigned integer arithmetic considerably faster than signed (this is also good practise, and helps make for self-documenting code). So the best declaration for an int variable in a tight loop would be:

```
register unsigned int var_name;
```

Although it is not guaranteed that the compiler will take any notice of "register", and "unsigned" may make no difference to the processor. The keyword "register" asks the compiler to place the variable into a general purpose register, rather than on the stack.

## 2.3 Integers 2

> If you use unsigned integers instead of integers be sure the value will never be negative.

Look at the following example and reflect...

```
#include <stdio.h>

#define DATALEN (6)

int main() {
    unsigned char result;
    int i;
    int data1[DATALEN] = {1,-1,1,-1,1,-1};
    int data2[DATALEN] = {1,2,3,4,5,6};

    for(i = 0; i < DATALEN; i++) {
        result = data1[i]*data2[i];
        printf("%3d %3d\n", data1[i]*data2[i], result);
    }
    return 0;
}
```

## 2.4 Integers 3

Integer arithmetic is much faster than floating-point arithmetic.

Integer arithmetic can be usually be done directly by the processor, rather than relying on external FPUs or floating point maths libraries. If you only need to be accurate to two decimal places (e.g. in a simple accounting package), scale everything up by 100, and convert it back to floating point as late as possible.

The following example compares the execution speed of a multiplication of three different data types:

```
#include <time.h>

long test_float() {
    long i;
    float j = 3.14, k = 13.0;

    for (i = 0 ; i < 10000000; i++) {
        j *= k;
    }
    return j;
}

long test_double() {
    long i;
    double j = 3.14, k = 13.0;

    for (i = 0 ; i < 10000000; i++) {
        j *= k;
    }
    return j;
}

long test_int() {
    long i;
    int j = 314, k = 13;

    for (i = 0 ; i < 10000000; i++) {
        j *= k;
    }
    return j;
}


unsigned long test(long (*function)(void), int n) {
    unsigned long average = 0;
    clock_t begin, end;
    int i;

    for (i = 0; i < n; i++) {
        begin = clock();
        function();
        end = clock();
        average += end - begin;
    }
    return ((unsigned long)average / n);
}

int main () {
    printf("test_float = %ld\n", test(test_float, 10));
    printf("test_double = %ld\n", test(test_double, 10));
    printf("test_int  = %ld\n", test(test_int, 10));
    return 0;

}
```

These are the results on my machine:

```
test_float = 77
test_double = 73
test_int   = 44
```

You can try comparing also other arithmetic operations.

## 2.5 Arithmetic operation

The strength of an arithmetic operation specifies how much processor time is needed to complete the operation. Strength reduction involves taking a relatively expensive operation and substituting it with a cheaper operation.

The next table provides a list of potential strength reduction techniques.

| Operation | Replace With | Example |
|---|---|---|
| Exponentiation | Multiplication | y = pow(x,3.0); becomes y = x*x*x; |
| Multiplication | Addition | y = x*5; becomes y = x+x+x+x+x; |
| Mult/Div by 2 | Bit-wise shift | y = x*34; becomes y = (x«5) + x + x; |
| Division | Multiplication | y = x/4; becomes y = x*(0.25); |

## 2.6 Loops: early breaking

It is often not necessary to process the entirety of a loop.

For example, if you are searching an array for a particular item, break out of the loop as soon as you have got what you need.

This loop searches a list of 10000 numbers to see if there is a -99 in it:

```
found = FALSE;
for(i=0; i < 10000; i++) {
    if( list[i] == -99 ) {
        found = TRUE;
    }
}
if( found ) {
    printf("Yes, there is a -99!!!\n");
}
```

This works well, but will process the entire array, no matter where the search item occurs in it. A better way is to abort the search as soon as you've found the desired entry:

```
found = FALSE;
for(i=0; i < 10000; i++){
    if( list[i] == -99 ) {
        found = TRUE;
        break;
    }
}
if( found ) {
    printf("Yes, there is a -99!!!\n");
}
```

If the item is at, say position 23, the loop will stop there and then, and skip the remaining 9977 iterations.

## 2.7 Loops: jamming

Never use two loops where one will suffice.

For example:

```
for(i=0; i < 100; i++) {
    stuff();
}
for(i=0; i < 100; i++) {
    morestuff();
}
```

It would be better to do:

```
for(i=0; i < 100; i++) {
    stuff();
    morestuff();
}
```

Note, however, that if you do a lot of work in the loop, it might not fit into your processor's instruction cache. In this case, two separate loops may actually be faster as each one can run completely in the cache.

## 2.8 Loops: unrolling

It is well known that unrolling loops can produce considerable savings.

The following example demonstrates this principle:

| Normal Loop | Unrolled Loop |
|---|---|
| execute loop 10 times | execute loop 2 times |
| step A | step A |
| end loop | step A |
| | step A |
| | step A |
| | step A |
| | end loop |

Both loops perform step A ten times. The unrolled code is definitely larger, but it is also faster because the overhead incurred by looping around (checking the end condition and jumping back to the beginning) is performed less often, thus taking up a smaller percentage of loop- execution time.

So the source code:

```
for(i=0; i < 3; i++) {
    something(i);
}
```

is less efficient than:

```
something(0);
something(1);
something(2);
```

because the code has to check and increment the value of 'i' each time round the loop.

Compilers will often unroll simple loops like this, where a fixed number of iterations is involved, but something like:

```
for(i=0;i<limit;i++){ ... }
```

is unlikely to be unrolled, as we don't know how many iterations there will be.

It is, however, possible to manually unroll this sort of loop and take advantage of the speed savings that can be gained.

Consider the example in the next listing: the loop-condition is tested once every 8 iterations, instead of on each one. If you know that you will working with arrays of a certain size, you could make the blocksize the same size as (or divisible into the size of) the array.

```
#include <stdio.h>

#define BLOCKSIZE (8)

int main(void) {
    int i = 0;
    int limit = 33; /* could be anything */
    int blocklimit;

    /*
     * The limit may not be divisible by BLOCKSIZE,
     * go as near as we can first, then tidy up.
     */
    blocklimit = (limit / BLOCKSIZE) * BLOCKSIZE;

    /* unroll the loop in blocks of 8 */
    while (i < blocklimit) {
        printf("process(%d)\n", i);
        printf("process(%d)\n", i+1);
        printf("process(%d)\n", i+2);
        printf("process(%d)\n", i+3);
        printf("process(%d)\n", i+4);
        printf("process(%d)\n", i+5);
        printf("process(%d)\n", i+6);
        printf("process(%d)\n", i+7);
        /* update the counter */
```

```
        i += 8;
    }

    /*
     * There may be some left to do.
     * This could be done as a simple for() loop,
     * but a switch is faster (and more interesting)
     */
    if (i < limit) {
        /*
         * Jump into the case at the place that will allow
         * us to finish off the appropriate number of items.
         */
        switch (limit - i) {
            case 7 : printf("process(%d)\n", i); i++;
            case 6 : printf("process(%d)\n", i); i++;
            case 5 : printf("process(%d)\n", i); i++;
            case 4 : printf("process(%d)\n", i); i++;
            case 3 : printf("process(%d)\n", i); i++;
            case 2 : printf("process(%d)\n", i); i++;
            case 1 : printf("process(%d)\n", i);
        }
    }
    return(0);
}
```

The next example counts the set bits in a data block:

```
#include <stdio.h>

#define DATALEN (4)
int main(void) {
    int k;
    long count = 0;
    int data[DATALEN] = {1,2,3,4};

    for (k = 0; k < DATALEN; k++) {
        count +=((data[k] & 128) >> 7) +
                ((data[k] & 64) >> 6) +
                ((data[k] & 32) >> 5) +
                ((data[k] & 16) >> 4) +
                ((data[k] &  8) >> 3) +
                ((data[k] &  4) >> 2) +
                ((data[k] &  2) >> 1) +
                ((data[k] &  1));
    }
    printf("count = %d\n", count);
    return 0;
}
```

## 2.9 Loops: go faster

Ordinarily, you would code a simple for() loop like this:

```
for (i = 0; i < 10; i++){ ... }
```

so that 'i' loops through the values 0,1,2,3,4,5,6,7,8,9.
If you don't care about the order of the loop counter, you can do this instead:

```
for (i = 10; i--;) { ... }
```

Using this code, 'i' loops through the values 9,8,7,6,5,4,3,2,1,0, and the loop should be faster.
This works because it is quicker to process "i- -" as the test condition, which says: "is i non-zero? If so, decrement it and continue.".

In tight loops, this make a considerable difference. The syntax is a little strange, but is perfectly legal. The third statement in the loop is optional (an infinite loop would be written as "for( ; ; )" ).

The same effect could also be gained by coding:

```
for (i = 10; i; i--) {...}
```

or (to expand it further)

```
for(i = 10; i != 0; i--) {...}
```

The only things you have to be careful of are remembering that the loop stops at 0 (so if you wanted to loop from 50-80, this wouldn't work), and the loop counter goes backwards. It's easy to get caught out if your code relies on an ascending loop counter.

The following examples shows the corresponding assembly translation of some kind of loops:

```
void plus() {
    int i;
    for(i = 0; i < 10; i++) {
        /* Nothing to do here */
    }
}

void minus1() {
    int i;
    for(i = 10; i--;) {
        /* Nothing to do here */
    }
}

void minus2() {
    int i;
    for(i = 10; i != 0; i--) {
        /* Nothing to do here */
    }
}
```

3: start function "plus"
8-9: reserve space for 'i' and init it to 0.
11-12: compare 'i' with 9.
12-17: if 'i' is less-equal to 9 increment and continue.

21: start function "minus1"
26-27: reserve space for 'i' and init it to 10.
29-30: decrement 'i'.
31-32: if 'i' is not equal to -1 continue.

35: start function "minus2"
40-41: reserve space for 'i' and init it to 10.
43: compare 'i' with 0.
44, 47-49: if 'i' is not equal to 0 decrement and continue.

```
1     .file  "loop_optimization.c"
2     .text
3  .globl _plus
4     .def   _plus; .scl   2;  .type  32; .endef
5  _plus:
6     pushl  %ebp
7     movl   %esp, %ebp
8     subl   $4, %esp
9     movl   $0, -4(%ebp)
10 L2:
11    cmpl   $9, -4(%ebp)
12    jle L4
13    jmp L1
14 L4:
15    leal   -4(%ebp), %eax
16    incl   (%eax)
17    jmp L2
18 L1:
19    leave
20    ret
21 .globl _minus1
22    .def   _minus1;   .scl   2;  .type  32; .endef
23 _minus1:
24    pushl  %ebp
25    movl   %esp, %ebp
26    subl   $4, %esp
27    movl   $10, -4(%ebp)
28 L7:
29    leal   -4(%ebp), %eax
30    decl   (%eax)
31    cmpl   $-1, -4(%ebp)
32    jne L7
33    leave
34    ret
35 .globl _minus2
36    .def   _minus2;   .scl   2;  .type  32; .endef
37 _minus2:
38    pushl  %ebp
39    movl   %esp, %ebp
40    subl   $4, %esp
41    movl   $10, -4(%ebp)
42 L12:
43    cmpl   $0, -4(%ebp)
44    jne L14
45    jmp L11
46 L14:
47    leal   -4(%ebp), %eax
48    decl   (%eax)
49    jmp L12
50 L11:
51    leave
52    ret
```

## 2.10 Common Case First, Infrequent Case Last

Always evaluate switch statement or if-else ladder in order of frequency.

In other words, evaluate the most common case first and the least common case last. The problem with this advice is that it assumes that you know how to determine the relative frequency of the different branching conditions. I can't offer you a general solution to this dilemma, but there are a few special situations. For example, if one case is normal and the others are all exceptional cases, put the normal case first.

```
if (type == NORMAL) { /*handle normal case*/ }
else if (type == EXCEPTION_1) { /*handle exception #1*/ }
else if (type == EXCEPYION_2) { /*handle exception #2*/ }
```

If the cases are all equally likely, then I would urge you to order things in a way that makes your code readable (e.g., alphabetical order, numeric order, etc.).

```
switch(char_type){
    case ALPHA: { /*...*/ }break;
    case COLON: { /*...*/ }break;
    case COMMA: { /*...*/ }break;
    case DIGIT: { /*...*/ }break;
    case LEFT_PAREN: { /*...*/ }break;
    case PERIOD: { /*...*/ }break;
    case RIGHT_PAREN: { /*...*/ }break;
    case SEMI_COLON: { /*...*/ }break;
    case WHITE_SPACE: { /*...*/ }break;
}
```

## 2.11 switch() instead of if-else

For large decisions involving if-else-else, like this:

```
if( type == NORMAL)
    handle_normal();
else if (type == EXCEPTION_1)
    handle_exception1();
else if (type == EXCEPTION_2)
    handle_exception2();f
```

it may be faster to use a switch:

```
switch (type) {
    case NORMAL: handle_normal(); break;
    case EXCEPTION_1: handle_exception1(); break;
    case EXCEPTION_1: handle_exception2(); break;
}
```

In the if() statement, if the last case is required, all the previous ones will be tested first. The switch lets us cut out this extra work. If you have to use a big if..else.. statement, test the most likely cases first.

## 2.12 Switch: fall-through

When a block of code has several labels, place the labels on separate lines. The fall-through feature of the C switch statement must be commented for future maintenance. If you've ever been "bitten" by this feature, you'll appreciate its importance!

```
switch (expr) {
    case ABC:
    case DEF:
        statement;
        break;
    case UVW:
        statement;   /*FALLTHROUGH*/
    case XYZ:
        statement;
          break;
}
```

While the last break is technically unnecessary, the consistency of its use prevents a fall-through error if another case is later added after the last one. The default case, if used, should always be last and does not require a final break statement if it is last.

## 2.13 Lookup table: function pointer

Also switch statement is a common programming technique to use with care: each test and jump uses processor time simply deciding what work should be done next. Putting individual cases in order by their relative

frequancy of occurrence will reduce the average execution time but it will not improve at all the worst case time. When the range of conditions for the choices can be translated to a continuous range of numbers (0, 1, 2, 3, 4, and so on), a jump table can be used. The idea behind a jump table is that the pointers of the functions to be called are inserted in a table, and the selector is used as an index in that table.

```
/* Creation of a table of pointers to functions. */
typedef int (*pfun)();
pfun jump_table[] = {processA, processB, processC} ;

/* The switch is replaced with the following next line: */
int result = jump_table[selector]();
```

The following is a workig example you can test:

```
#include <stdio.h>

int processA() {
    printf("processA() done\n");
    return 0;
}

int processB() {
    printf("processB() done\n");
    return 1;
}

int processC() {
    printf("processC() done\n");
    return 2;
}

typedef int (*pfun)();
pfun jump_table[] = { processA, processB, processC } ;


int main () {
    int selector = 0;

    printf("result = %d\n", jump_table[selector]());
    selector++;
    printf("result = %d\n", jump_table[selector]());
    selector++;
    printf("result = %d\n", jump_table[selector]());
    return 0;
}
```

## 2.14 Lookup table: array indices

If you wished to set a variable to a particular character, depending upon the value of something, you might do this :

```
switch ( queue ) {
    case 0:
        letter = 'W';
        break;
    case 1:
        letter = 'S';
        break;
    case 2 :
        letter = 'U';
        break;
}
```

or maybe:

```
if (queue == 0)
    letter = 'W';
else if (queue == 1)
    letter = 'S';
```

```
else
    letter = 'U';
```

A neater (and quicker) method is to simply use the value as an index into a character array, eg.

```
static char *classes="WSU";
letter = classes[queue];
```

## 2.15 Programming with data

> Algorithms, or details of algorithms, can often be encoded compactly, efficiently and expressively as data rather than, say, as lots of if statements.

The reason is that the complexity of the job at hand, if it is due to a combination of independent details, can be encoded. A classic example of this is parsing tables, which encode the grammar of a programming language in a form interpretable by a fixed, fairly simple piece of code. Finite state machines are particularly amenable to this form of attack, but almost any program that involves the 'parsing' of some abstract sort of input into a sequence of some independent 'actions' can be constructed profitably as a data-driven algorithm.

Perhaps the most intriguing aspect of this kind of design is that the tables can sometimes be generated by another program - a parser generator, in the classical case. As a more earthy example, if an operating system is driven by a set of tables that connect I/O requests to the appropriate device drivers, the system may be 'configured' by a program that reads a description of the particular devices connected to the machine in question and prints the corresponding tables.

One of the reasons data-driven programs are not common, at least among beginners, is the tyranny of Pascal. Pascal, like its creator, believes firmly in the separation of code and data. It therefore (at least in its original form) has no ability to create initialized data. This flies in the face of the theories of Turing and von Neumann, which define the basic principles of the stored-program computer. Code and data are the same, or at least they can be.

If you can cache any often used data rather than recalculating or reloading it, it will help. Examples of this would be sine/cosine tables, or tables of pseudo-random numbers (calculate 1000 once at the start, and just reuse them if you don't need truly random numbers).

## 2.16 Recursion

> Do not use recursion.

Recursion can be very elegant and neat, but creates many more function calls which can become a large overhead.

## 2.17 File access

Binary/unformatted file access is faster than formatted access, as the machine does not have to convert between human-readable ASCII and machine-readable binary. If you don't actually need to read the data in a file yourself, consider making it a binary file.

## 2.18 Compiler optimization 1

> Turn compiler optimization on, but do not assume that the optimized program will behave the same as the unoptimized one.

The compiler will be able to optimize at a much lower level than can be done in the source code, and perform optimizations specific to the target processor, reducing the code size and increasing the speed. Here are the optimization flags for the "gcc" compiler:

| Flag | Description |
|---|---|
| -O, -O1 | Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function. With -O, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time. |
| -O2 | Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. The compiler does not perform loop unrolling or function inlining when you specify -O2. As compared to -O, this option increases both compilation time and the performance of the generated code. |
| -O3 | Optimize yet more. |
| -O0 | Do not optimize. This is the default. |
| -Os | Optimize for size. -Os enables all -O2 optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size. |

One of the most common optimization is the "dead code elimination", which eliminate the code that the compiler believes to be either redundant or irrelevant. For example:

```
*pCommand = DISABLE;
*pData = 'c';
*pCommand = ENABLE;
```

Most compiler would remove the first statement because the value of "*pCommand" is not used before it is overwritten. But if those pointers are actually some memory mapped device register, probably your device will not work properly because it would not receive the DISABLE comand before the data is written. So, probably, after turning the optimization on, tou will have to completely test again your software.

To solve the problem of the example you should use the "volatile" keyword to indicate to the compiler that no optimization may be performed with those objects. Using "volatile" guarantees that the executable code will reload the value of the object each time it is used, allowing external changes to the value (system events/interrupts, OS states, other processes, and external devices) to be visible in your code.

So the following declaration:

```
/* Constant pointer to a volatile address value. */
volatile char const *p = 0xC000050;
```

tells the compiler that pointer 'p' should always point to address 0xC000050 (it is not allowed to change and point to any other address), but that the value in address 0xC000050 can change at any given time.

## 2.19 Compiler optimization 2

Compilers can often optimize a whole file.

Avoid splitting off closely related functions into separate files, the compiler will do better if can see both of them together (it might be able to inline the code, for example).

## 2.20 Global variables

Global variables are created placing the declaration outside all functions. The compiler place them in the data segment and initialize to "zero".

Many C programming standard ban the use of global variables, since access cannot be controlled or restricted so it is difficult to track who and why modified it.

So, well... if I wear the software engineer hat I should say: in the name of modularity and reentrancy minimize the use of global variables!

But... using global variables is more efficient than to pass a parameter to a function. In fact this eliminates the need to push and pop parameters and status for the context switch.

## 2.21 Local Variables

A program running in memory is made by several parts:

- Code Segment: where all the code is placed. This segment is read only.

- Stack: where all local variables are stored.
- Data Segment: a fixed area where all global variables are stored.
- Heap: where all dynamic memory is allocated.

Special keyword exist to specify where to store a local variable:

| Keyword | Place | Initialized | Notes |
|---|---|---|---|
| auto | Stack | NO | This is the default |
| static | Data Segment | YES (zero) | The scope is local |
| register | CPU register | NO | |

So a little but important lesson learned:

Do not rely on the initialization of auto and register variables.

## 2.22 The Stack

C uses the stack to store variables declared in functions and to pass parameters to functions. This is roughly how it works:

1. The caller pushes the parameters.
2. The function is called.
3. The called picks up the parameters.
4. The called pushes its local variables.
5. The called pops its local variables.
6. The called jumps back to the caller.
7. The caller pops the parameters
8. The return value is used.

The next example show this mechanism:

```
int sum(int a, int b) {
    int c;

    c = a + b;

    return c;
}


int main () {
    int result;

    result = sum(5, 4);

    return 0;
}
```

28-29: The caller pushes the parameters.
30 : The function is called.
9 : The called picks up the parameters.
8 : The called pushes its local variables.
12 : The called pops its local variables.
13-14: The called jumps back to the caller.
31 : The caller pops the parameters.

```
1      .file   "stack.c"
2      .text
3   .globl _sum
4      .def    _sum;   .scl    2;  .type   32; .endef
5   _sum:
6      pushl   %ebp
7      movl    %esp, %ebp
8      subl    $4, %esp
9      movl    12(%ebp), %eax
10     addl    8(%ebp), %eax
11     movl    %eax, -4(%ebp)
12     movl    -4(%ebp), %eax
13     leave
14     ret
15     .def    ___main;    .scl    2;  .type   32; .endef
16  .globl _main
17     .def    _main; .scl    2;  .type   32; .endef
18  _main:
19     pushl   %ebp
20     movl    %esp, %ebp
21     subl    $24, %esp
22     andl    $-16, %esp
23     movl    $0, %eax
24     movl    %eax, -8(%ebp)
25     movl    -8(%ebp), %eax
26     call    __alloca
27     call    ___main
28     movl    $4, 4(%esp)
29     movl    $5, (%esp)
30     call    _sum
31     movl    %eax, -4(%ebp)
32     movl    $0, %eax
33     leave
34     ret
```

Managing the stack to facilitate a function call is the responsibility of both the procedure that is invoking the function and the function being invoked. Both entities must work together in order to pass information back and forth on the stack. I will start with the responsibilities that belong to the invoking function.

The following steps can be used to invoke a procedure and pass it arguments

1. Push the current function's state onto the stack.
2. Push the return value onto the stack.
3. Push function arguments onto the stack.
4. Push the return address onto the stack and jump to the location of the procedure

The function being invoked must also take a few steps to ensure that it can access the parameters passed to it and create local storage (function's prologue):

1. Push EBP on to the stack (to save its value).
2. Copy the current ESP value into EBP.
3. Decrement ESP to allocate local storage.
4. Execute the function's instructions.

Intel machines, the EBP register is pushed on the stack so that it can serve as a reference point. EBP is known as the stack frame pointer, and it is used so that elements in the activation record can be referenced via indirect addressing (i.e., MOV AX, [EBP+8]).

When the function has done its thing and is ready to return, it must perform the following stack maintenance steps (function's epilogue):

1. Reclaim local storage.
2. Pop EBP off the stack.

3. Pop the return address off the stack and jump to the return address.

Once the invoked function has returned, the invoking function will need to take the following steps to get its hands on the return value and clean up the stack:

1. Pop the function arguments off the stack.
2. Pop the return value off the stack.
3. Pop the saved program state off the stack.

Another way to handle the arguments is to simply increment the stack pointer. We really have no use for the function arguments once the invoked function has returned, so this is a cleaner and more efficient way to reclaim the corresponding stack space.

This whole process can be seen in terms of four compound steps:

1. Invoking function sets up stack
2. Prologue: function invoked sets up EBP and local storage
3. Called function executes
4. Epilogue:unction invoked frees local storage and restores EBP
5. Invoking function extracts return value and cleans up stack

## 2.23 Aliases

Consider the following:

```
void func1(int *data) {
    int i;
    for(i = 0; i < 10; i++) {
        func2(*data, i);
    }
}
```

Even though "*data" may never change, the compiler does not know that func2() did not alter it, and so the program must read it from memory each time it is used - it may be an alias for some other variable that is altered elsewhere.

If you know it won't be altered, you could code it like this instead:

```
void func1(int *data) {
    int i;
    int localdata;
    localdata = *data;
    for(i = 0; i < 10; i++) {
        func2(localdata, i);
    }
}
```

This gives the compiler better opportunity for optimization.

## 2.24 Embedded statement

There is a time and a place for embedded assignment statements. In some constructs there is no better way to accomplish the results without resulting in bulkier and less readable code:

```
while ((c = getchar()) != EOF) {
    /* process the character */
}
```

Using embedded assignment statements to improve run-time performance is possible.

However, you should consider the tradeoff between increased speed and decreased maintainability that results when embedded assignments are used in artificial places. For example:

```
x = y + z;
d = x + r;
```

should not be replaced by:

```
d = (x = y + z) + r;
```

even though the latter may save one cycle. In the long run the time difference between the two will decrease as the optimizer is enhanced, while the difference in ease of maintenance will increase.

# Chapter 3

# Not only a matter of style...

## 3.1 Put the constant on the left in a conditional

We've all experienced bugs like this:

```
while (continue = TRUE) {
    /* ... */
}
```

This type of problem can be solved by putting the constant on the left, so if you leave out an = in a conditional, you will get a compiler error instead of a program bug (because constants are non lvalues, of course):

```
while (TRUE = continue) {
    /* compile error! */
}
```

This is another example: instead of writing:

```
while (c == ' ' || c == '\t' || c == '\n')
    c = getchar();
```

You can say:

```
while (' ' == c || '\t' == c || '\n' == c)
    c = getchar();
```

This way you will get a compiler diagnostic:

```
while (' ' = c || '\t' == c || '\n' == c)
    c = getchar();
```

This style lets the compiler find problems; the above statement is invalid because it tries to assign a value to ' '.

## 3.2 Tell the truth

There is no boolean data type in C, integers are used instead:

- The value of 0 is FALSE.
- Any other value is TRUE.

So TRUE is defined as 1, but TRUE is logically any non-zero value!

```
if (result == TRUE) /* WRONG */
    {...}

if (result)        /* correct, if result != 0 */
    {...}
```

## 3.3 Check a boolean value

Do not check a boolean value for equality with 1 (TRUE, YES, etc.)
. Instead test for inequality with 0 (FALSE, NO, etc.).

Most functions are guaranteed to return 0 if false, but only non-zero if true. Thus,

```
if (TRUE == func()) { ... }
```

is better written

```
if (FALSE != func()) { ... }
```

## 3.4 Defining logical opposites

How many times have you seen the following?

```
#define TRUE (1)
#define FALSE (0)
```

When defining logical opposites, first define one of the logical states and then define the opposite state as a macro based on the original state:

```
#define TRUE (1)
#define FALSE (!TRUE)
```

This method has two advantages:

- First, if the logical state for TRUE is changed, the logical states that are based on TRUE are automatically changed. By design, this increases the readability of the code, provides a better understanding of the intent of the original software writer, and helps document the code. The software engineer doesn't have to search and guess which states are logical opposites.
- Second, some optimizers will perform better if the logical opposite states are based on each other.

## 3.5 Are you sure it works?

Typically testing, and therefore test case design, is performed at four different levels:

1. Unit Testing: this is the "smallest" piece of software that a developer creates. It is typically the work of one programmer and is stored in a single disk file. Different programming languages have different units: in C++ and Java the unit is the class; in C the unit is the function; in less structured languages like Basic and COBOL the unit may be the entire program.
2. Integration Testing: we assemble units together into subsystems and finally into systems. It is possible for units to function perfectly in isolation but to fail when integrated. A classic example is this C program and its subsidiary function:

```
/* main program */
void func(int);

int main() {
    func(42); /* call the function passing an integer */
    return 0;
}

/* function (in a separate file) */
#include <stdio.h>

void func(double x) { /* expects a double, not an int! */
    printf ("%f\n",x); /* Will print garbage (0 is most likely) */
}
```

   If these units were tested individually, each would appear to function correctly. In this case, the defect only appears when the two units are integrated. The main program passes an integer to function oops but "func" expects a double length integer and trouble ensues. It is vital to perform integration testing as the integration process proceeds. Mandatory and rigorous unit testing is an essential tool for eliminating bugs during implementation.

3. System Testing - A system consists of all of the software (and possibly hardware, user manuals, training materials, etc.) that make up the product delivered to the customer. System testing focuses on defects that arise at this highest level of integration. Typically system testing includes many types of testing: functionality, usability, security, internationalization and localization, reliability and availability, capacity, performance, backup and recovery, portability, and many more.

| System Test | Purpose |
|---|---|
| Functional test | Determines if the application does what it's supposed to do. |
| Regression test | Verifies that previously fixed bugs have not reappeared. |
| Stress test | Examines behavior under a large number of service requests. |
| Performance test | Measures execution speed and memory footprint. |
| Security test | Verifies that an application can withstand intrusion attempts. |
| Installation test | Determines if the installation subsystem works. |
| Usability test | Measures the ability of a user to interact with the application. |
| Conformance test | Checks to see if the application conforms to a specified standard. |

4. Acceptance Testing - Acceptance testing is defined as that testing, which when completed successfully, will result in the customer accepting the software and giving us their money. From the customer's point of view, they would generally like the most exhaustive acceptance testing possible (equivalent to the level of system testing). From the vendor's point of view, we would generally like the minimum level of testing possible that would result in money changing hands. Typical strategic questions that should be addressed before acceptance testing are: Who defines the level of the acceptance testing? Who creates the test scripts? Who executes the tests? What is the pass/fail criteria for the acceptance test? When and how do we get paid?

## 3.6 Style

What's the best style for code layout in C?

K&R, while providing the example most often copied, also supply good excuse for disregarding it: The position of braces is less important, although people hold passionate beliefs. We have chosen one of several popular styles. Pick a style that suits you, then use it consistently. It is more important that the layout chosen be consistent (with itself, and with nearby or common code) than that it be "perfect." If your coding environment (i.e. local custom or company policy) does not suggest a style, and you don't feel like inventing your own, just copy K&R.

## 3.7 Be consistent

Be consistent in the way you write your code.

Use the same indentation and bracketing style everywhere. If you put the constant on the left in a conditional, do it everywhere. If you assert on your pointers, do it everywhere.

Use the same kind of comment style for the same kind of comments. If you are the type to go in for a naming convention (like Hungarian notation), then you have to stick to it everywhere. Don't do int iCount in one place and int nCount in another.

## 3.8 Don't ignore API function return values

Most API functions will return a particular value which represents an error.

You should test for these values every time you call the API function. If you don't want want to clutter your code with error-testing, then wrap the API call in another function (do this when you are thinking about portability, too) which tests the return value and either asserts, handles the problem, or throws an exception.

## 3.9 Use Make

Utilities for compiling and linking such as Make simplify considerably the task of moving an application from one environment to another. During development, make recompiles only those modules that have been changed since the last time make was used.

## 3.10 Use Lint

Use lint frequently. lint is a C program checker that examines C source files to detect and report type incompatibilities, inconsistencies between function definitions and calls, potential program bugs, etc.

## 3.11 Test for empty input

All programs should be tested for empty input.

Many programs fail when their input is missing. This is also likely to help you understand how the program is working

## 3.12 Test for all input

All programs should be tested for all input.

Don't assume any more about your users or your implementation than you have to. Things that "cannot happen" sometimes do happen. A robust program will defend against them. If there's a boundary condition to be found, your users will somehow find it!

## 3.13 Test for true or false

Do not default the test for non-zero

That is:

```
if (f() != FAIL) { ... }
```

is better than:

```
if (f()) { ... }
```

even though FAIL may have the value 0 which C considers to be FALSE. An explicit test will help you out later when somebody decides that a failure return should be -1 instead of 0.

## 3.14 Be clear

Make your program's purpose and structure clear.

Keep in mind that you or someone else will likely be asked to modify your code or make it run on a different machine sometime in the future. Craft your code so that it is portable to other machines.

## 3.15 Indentation

Establish an indentation standard and be consistent throughout the work product.

Structured indentation makes software easier to read, thereby reducing the possibility of coding error. Consider the following example:

```
for (x=0; x
<
= MAX_X; ++x) {for (y=1; y
<
= MAX_Y; ++y){z+= x*y;
   if (x == TEST) {
     printf("x value pass");
     t=x/AVERAGE;
   }
z /= 4;}
   }
```

Nerves of steel would be required to prepare for a code inspection if you had to review page after page of poorly indented code. This piece of code could give the visual illusion that the line of code z /= 4; executes as part of the x for loop.

Remember, you're a software professional, not a magician. Leave the illusions to management.

Now consider the more palatable solution:

```
for (x = 0; x <= MAX_X; ++x) {
    for (y = 1; y <= MAX_Y; ++y) {
        z += x*y;
        if (x == TEST) {
            printf("x value pass");
            t = x/AVERAGE;
        }
        z /= 4;
    }
}
```

Good indentation helps to visually define the structure of the code. In this example, all open and closed braces were put on separate lines with the open braces lining up with the first character of the conditional/loop expression. Both versions of the source code execute identically; the only difference is the visual interpretation.

A good rule of thumb is to allow only one statement per line of code and insert braces on their own line.

You be the judge. Which piece of code would you rather inspect? Whatever form of indentation you decide to use, you must be consistent throughout your code. Remember, one of the objectives is to make the code more readable.

## 3.16 Magic numbers

Do not use hard coded numbers directly in expressions.

Maintainability becomes a horrendous task when the code is contaminated with magic numbers.

A magic number has meaning to the original programmer at the time of coding, but when the programmer leaves the company due to an evolutionary process called career development, the significance of the number (unless defined in a comment or definition table) can lose meaning. Let's take the simple line equation to help illustrate this point.

The equation is:

```
y = m * x + b
```

where m is the slope of the line, 'b' is the 'y' intercept (at x = 0), and 'x' (input) and 'y' (output) are the points on the x - and y -axes. You can write the following line of code to satisfy the requirements:

```
y = (13.432 * x) + 4.232;
```

If this line of code is out in the field for many years and then a requirement change is issued forcing a modification to the code, at first glance a person would try to determine the origins of the numbers.

If the code is properly commented, this becomes less of an issue (although the same constant might be used in a different part of the software, increasing complexity). Contrast that with the following line of code:

```
y = (SS_PRESSURE_SLOPE * x) + SS_PRESSURE_INTERCEPT;
```

The first observation you can make is that the equation deals with some type of pressure constants. The key words SLOPE and INTERCEPT indicate to the reader that the expression might take on the format of a line equation.

The constants can be defined in a parameter or constant definition file or in the local header file. Any changes in the value of the constant would propagate throughout the definition scope of the software product, sparing the programmer the burden of searching the entire software product for all occurrences of the constant.

## 3.17 Treat functions with care

Functions are the most general structuring concept in C. They should be used to implement "top-down" problem solving - namely breaking up a problem into smaller and smaller subproblems until each piece is readily expressed in code. This aids modularity and documentation of programs. Moreover, programs composed of many small functions are easier to debug.

Cast all function arguments to the expected type if they are not of that type already, even when you are convinced that this is unnecessary since they may hurt you when you least expect it. In other words, the compiler will often promote and convert data types to conform to the declaration of the function parameters. But doing so manually in the code clearly explains the intent of the programmer, and may ensure correct results if the code is ever ported to another platform.

If the header files fail to declare the return types of the library functions, declare them yourself. Surround your declarations with #ifdef/#endif statements in case the code is ever ported to another platform.

Function prototypes should be used to make code more robust and to make it run faster.

## 3.18 Dangling else

Stay away from "dangling else" problem unless you know what you're doing:

```
if (a == 1)
    if (b == 2)
        printf("***\n");
    else
        printf("###\n");
```

The rule is:

> else attaches to the nearest if.

When in doubt, or if there is a potential for ambiguity, add curly braces to illuminate the block structure of the code.

## 3.19 Don't let yourself believe you see what isn't there

Look at the following example:

```
while (c == ' ' || c = '\t' || c == '\n')
    c = getchar();
```

The statement in the while clause appears at first glance to be valid C. The use of the assignment operator, rather than the comparison operator, results in syntactically incorrect code. The precedence of '=' is lowest of any operator so it would have to be interpreted this way (parentheses added for clarity):

```
while ((c == ' ' || c) = ('\t' || c == '\n'))
    c = getchar();
```

The clause on the left side of the assignment operator is:

```
(c == ' ' || c)
```

which does not result in an lvalue. If 'c' contains the space character, the result is "true" and no further evaluation is performed, and "true" cannot stand on the left-hand side of an assignment.

## 3.20 Be clear in your intentions

When you write one thing that could be interpreted for something else, use parentheses or other methods to make sure your intent is clear.

This helps you understand what you meant if you ever have to deal with the program at a later date. And it makes things easier if someone else has to maintain the code. It is sometimes possible to code in a way that anticipates likely mistakes.

## 3.21 Semicolons and null statement

Semicolons are very important: they form a statement terminator, they tell the compiler where one statement ends and the next one begins. If you forget to place one after each statement, you will get compilation errors.

The null body of a for or while loop should be alone on a line and commented so that it is clear that the null body is intentional and not missing code.

```
while (*dest++ = *src++)
    ; /* VOID */
```

On the contrary remember that a semicolon after the condition forms a void statement:

```
printf("input an integer: ");
scanf("%i", %i);
if (i > 0);
    printf("The number you entered is POSITIVE.");
```

results in:

```
input an integer: -2

The number you entered is POSITIVE.
```

## 3.22 Make String Comparison Look More Natural

One of the problems with the strcmp () routine to compare two strings is that it returns zero if the strings are identical. This leads to convoluted code when the comparison is part of a conditional statement:

```
if (!strcmp(s,"volatile")) {
    return QUALIFIER;
}
```

a zero result indicates false, so we have to negate it to get what we want. Here's a better way. A possible solution is to define a macro STREQ:

```
#define STREQ(str1, str2) (strcmp((str1), (str2)) == 0)
```

Using this, a statement such as:

```
If (STREQ(inputstring, somestring))
    { ... }
```

carries with it an implied behavior that is unlikely to change under the covers.

Another approach can set up the definition:

```
#define STRCMP(a,R,b) (strcmp(a,b) R 0)
```

Now you can write a string in the natural style

```
if (STRCMP(s, ==, "volatile"))
    { ... }
```

Using this definition, the code expresses what is happening in a more natural style. Try rewriting the cdecl program to use this style of string comparison, and see if you prefer it.

## 3.23 Constants

> Symbolic constants make code easier to read.

Numerical constants should generally be avoided; use the #define function of the C preprocessor to give constants meaningful names. Defining the value in one place (preferably a header file) also makes it easier to administer large programs since the constant value can be changed uniformly by changing only the define. Consider using the enumeration data type as an improved way to declare variables that take on only a discrete set of values. Using enumerations also lets the compiler warn you of any misuse of an enumerated type. At the very least, any directly-coded numerical constant must have a comment explaining the derivation of the value.

Constants should be defined consistently with their use; e.g. use 540.0 for a float instead of 540 with an implicit float cast. That said, there are some cases where the constants 0 and 1 may appear as themselves instead of as defines. For example if a for loop indexes through an array, then:

```
for (i = 0; i < arraysub; i++) { ... }
```

is quite reasonable, while the code:

```
gate_t *front_gate = opens(gate[i], 7);
if (front_gate == 0) {
    error("can't open %s\n", gate[i]);
}
```

is not. In the second example front_gate is a pointer; when a value is a pointer it should be compared to NULL instead of 0. Even simple values like 1 or 0 are often better expressed using defines like TRUE and FALSE (and sometimes YES and NO read better).

Don't use floating-point variables where discrete values are needed. This is due to the inexact representation of floating point numbers (see the second test in scanf, above). Test floating-point numbers using <= or >=; an exact comparison (== or !=) may not detect an "acceptable" equality.

Simple character constants should be defined as character literals rather than numbers. Non-text characters are discouraged as non-portable. If non-text characters are necessary, particularly if they are used in strings, they should be written using a escape character of three octal digits rather than one (for example, '
007'). Even so, such usage should be considered machine-dependent and treated as such.

## 3.24 Function names

Function names must be unique and should reflect what they do and what they return.

Functions are used in expressions, often in an if clause, so they need to read appropriately. For example:

```
if (checksize(x)) {...}
```

is unhelpful because it does not tell us whether checksize returns true on error or non-error; instead:

```
if (validsize(x)) {...}
```

makes the point clear and makes a future mistake in using the routine less likely.

## 3.25 Variable names

When choosing a variable name, length is not important but clarity of expression is.

A long name can be used for a global variable which is rarely used but an array index used on every line of a loop need not be named any more elaborately than i. Using 'index' or 'elementnumber' instead is not only more to type but also can obscure the details of the computation. With long variable names sometimes it is harder to see what is going on.

A global variable rarely used may deserve a long name, maxphysaddr say. An array index used on every line of a loop needn't be named any more elaborately than i. Saying index or elementnumber is more to type (or calls upon your text editor) and obscures the details of the computation.

When the variable names are huge, it's harder to see what's going on. This is partly a typographic issue; consider

```
for(i = 0 to 100) {
    array[i] = 0;
}
```

versus:

```
for(elementnumber = 0 to 100) {
    array[elementnumber] = 0;
}
```

The problem gets worse fast with real examples. Indices are just notation, so treat them as such.

Pointers also require sensible notation. "np" is just as mnemonic as "nodepointer" if you consistently use a naming convention from which "np" means "node pointer" is easily derived.

As in all other aspects of readable programming, consistency is important in naming. If you call one variable "maxphysaddr", don't call its cousin "lowestaddress".

## 3.26 goto statements

goto should be used sparingly (or never...).

The one place where they can be usefully employed is to break out of several levels of switch, for, and while nesting, although the need to do such a thing may indicate that the inner constructs should be broken out into a separate function.

```
for (...) {
    while (...) {
        ...
        if (wrong)
            goto error;
    }
}
...
error:
print a message
```

When a goto is necessary the accompanying label should be alone on a line and either tabbed one stop to the left of the code that follows, or set at the beginning of the line. Both the goto statement and target should be commented to their utility and purpose.

## 3.27 Declaration vs. Definition

Don't define variables in header files.

You only want to declare them in the header file, and define them (once only) in the appropriate C source file, which should #include the header file of course for type checking.

A header file is literally substituted into your C code in place of the #include statement. Consequently, if the header file is included in more than one source file all the definitions in the header file will occur in both source files. This causes them to be defined more than once, which gives a linker error.

The distinction between a declaration and a definition is easy to miss for beginners:

- A *declaration* tells the compiler that the named symbol should exist and should have the specified type, but it does not cause the compiler to allocate storage space for it.
- A *definition* does allocate the space, and provides an initialization value, if any.

So there can be many "declarations" (and in many translation units) of a single global variable or function, but there must be exactly one "definition".

To make a declaration rather than a definition, put the keyword "extern" before the definition. So, if we have an integer called "counter" which we want to be publicly available, we:

- Define it in a source file (one only) as "int counter;" at top level.
- Declare it in a header file as "extern int counter;".

## 3.28 Shorthand operator

I have often seen the "op=" shorthand operators abused by engineers who thought it would make their code faster. This is not necessarily true. If anything, all it does is hurt readability. For example, consider the following:

```
int i;
i = i + 0xFF;
i += 0xFF;
```

It just so happens that there is no tangible performance differential between these two statements. Both of these statements get translated into the same assembly code:

```
; i = i + 0xFF;
leal    -4(%ebp), %eax
addl    $255, (%eax)

; i += 0xFF;
leal    -4(%ebp), %eax
addl    $255, (%eax)
```

## 3.29 Header Files

Some considerations about header files:

- Header files should be functionally organized, that is, declarations for separate subsystems should be in separate header files.
- Declarations that are likely to change when code is ported from one platform to another should be in a separate header file.
- Avoid private header filenames that are the same as library header filenames. The statement

  ```
  #include "math.h"
  ```

  includes the standard library math header file if the intended one is not found in the current directory. If this is what you want to happen, comment this fact.
- Using absolute pathnames for header files is not a good idea. The "include-path" option of the C compiler (-I (capital "eye") on many systems) is the preferred method for handling extensive private libraries of header files; it permits reorganizing the directory structure without having to alter source files. For example, instead of doing:

  ```
  #include "c:\development\myproj\include\header.h"
  ```

  you can do as follow:

  ```
  #include "header.h"

  gcc -I c:\development\myproj\include myproj.c
  ```

## 3.30 #include < > and #include " "

The < > syntax is typically used with standard or system-supplied headers.
The " " syntax is typically used for a program's own header files.

## 3.31 Identifier clashes between source files

In C, variables and functions are by default public, so that any C source file may refer to global variables and functions from another C source file.

This is true even if the file in question does not have a declaration or prototype for the variable or function. You must, therefore, ensure that the same symbol name is not used in two different files. If you don't do this you will get linker errors and possibly warnings during compilation.

One way of doing this is to prefix public symbols with some string which depends on the source file they appear in. For example, all the routines in "gfx.c" might begin with the prefix "gfx".

If you are careful with the way you split up your program, use sensible function names, and don't go overboard with global variables, this shouldn't be a problem anyway.

To prevent a symbol from being visible from outside the source file it is defined in, prefix its definition with the keyword "static". This is useful for small functions which are used internally by a file, and won't be needed by any other file. So you can declare anything within a file (external to functions) as "static", unless it is intended to be global.

## 3.32 Redefinitions, redelarations, conflicting types

Simple rule: include files should never include include files.

Multiple inclusions are a bane of systems programming. It's not rare to have files included five or more times to compile a single C source file. The Unix "/usr/include/sys" stuff is terrible this way.

Consider what happens if a C source file includes both "a.h" and "b.h", and also "a.h" includes "b.h" (which is perfectly sensible: "b.h" might define some types that "a.h" needs). Now, the C source file includes "b.h" twice. So every #define in "b.h" occurs twice, every declaration occurs twice (not actually a problem), every typedef occurs twice, etc. In theory, since they are exact duplicates it shouldn't matter, but in practice it is not valid C and you will probably get compiler errors or at least warnings.

The solution to this problem is to ensure that the body of each header file is included only once per source file. This is generally achieved using preprocessor directives.

We will define a macro for each header file, as we enter the header file, and only use the body of the file if the macro is not already defined. In practice it is as simple as putting this at the start of each header file:

```
#ifndef FILENAME_H
#define FILENAME_H
```

and then putting this at the end of it:

```
#endif
```

replacing FILENAME_H with the (capitalised) filename of the header file, using an underline instead of a dot. Some people like to put a comment after the #endif to remind them what it is referring to, e.g.

```
#endif /* #ifndef FILENAME_H */
```

This dance involving #ifdef's can prevent a file being read twice, but the problem is that the #ifdef's are in the file itself, not the file that includes it. The result is often thousands of needless lines of code passing through the lexical analyzer, which is (in good compilers) the most expensive phase.

So... follow the simple rule.

## 3.33 What to put in...

I'm splitting up a program into multiple source files for the first time, and I'm wondering what to put in ".c" files and what to put in ".h" files.

As a general rule, you should put these things in header files:

- macro definitions (preprocessor #defines)
- structure, union, and enumeration declarations
- typedef declarations
- external function declarations
- global variable declarations

It's especially important to put a declaration or definition in a header file when it will be shared between several other files. In particular, never put external function prototypes in ".c" files.

On the other hand, when a definition or declaration should remain private to one ".c" file, it's fine to leave it there. The best arrangement is:

- Place each definition in some relevant ".c" file.
- Put an external declaration in a header (".h") file.
- The header file is included wherever the declaration is needed.
- The ".c" file containing the definition should also #include the same header file, so the compiler can check that the definition matches the declarations.

This rule promotes a high degree of portability: it is consistent with the requirements of the ANSI C Standard, and is also consistent with most pre-ANSI compilers and linkers.

It's especially important to put global declarations in header files if you want the compiler to catch inconsistent declarations for you.

In particular, never place a prototype for an external function in a ".c" file: it wouldn't generally be checked for consistency with the definition, and an incompatible prototype is worse than useless.

## 3.34 Pointers to Functions

Use function pointers to encode complexity.

Some of the complexity is passed to the routine pointed to. The routine must obey some standard protocol - it's one of a set of routines invoked identically - but beyond that, what it does is its business alone. The complexity is distributed.

There is this idea of a protocol, in that all functions used similarly must behave similarly. This makes for easy documentation, testing, growth and even making the program run distributed over a network - the protocol can be encoded as remote procedure calls.

I argue that clear use of function pointers is the heart of object-oriented programming.

Given a set of operations you want to perform on data, and a set of data types you want to respond to those operations, the easiest way to put the program together is with a group of function pointers for each type. This, in a nutshell, defines class and method. The OO languages give you more of course - prettier syntax, derived types and so on - but conceptually they provide little extra.

Combining data-driven programs with function pointers leads to an astonishingly expressive way of working, a way that, in my experience, has often led to pleasant surprises. Even without a special OO language, you can get 90% of the benefit for no extra work and be more in control of the result. I cannot recommend an implementation style more highly. Maybe that's it: the discipline it forces pays off in the long run.

```
#include <stdio.h>

int addition (int a, int b) {
    printf("addition(%d,%d)\n", a, b);
    return (a+b);
}

int subtraction (int a, int b) {
    printf("subtraction(%d,%d)\n", a, b);
    return (a-b);
}

/*
* minus is a global pointer to a function that has two parameters of type int,
* it is immediately assigned to point to the function subtraction.
*/
int (*minus)(int,int) = subtraction;

int operation (int x, int y, int (*functocall)(int,int)) {
    int g;

    g = (*functocall)(x,y);
    return (g);
}

int main () {
    int m, n;

    m = operation (7, 5, addition);
    n = operation (20, m, minus);
    printf("%d\n", n);
    return 0;
}
```

## 3.35 Macros for the interface

A good use for macros is in developing reusable interfaces.

For example:

```
#define GET_SENSOR_SIGNAL() (signed short int)(RAW_ANALOG_INPUT.SENSOR1)
#define GET_NVRAM_DATA() (NVRAM_STRUCTURE.DATA_REGISTER)
#define SET_NVRAM_DATA(x) (NVRAM_STRUCTURE.DATA_PORT = (x))
```

The interface is defined by Get and Set macros and the definition of these macros are hardware-, machine-, and/or architecture-dependent.

In this example, the GET_SENSOR_SIGNAL macro looks for a data structure (RAW_ANALOG_INPUT) that is memory mapped to the A/D converters and picks out the SENSOR1 port of the A/D converter.

As an algorithm developer, my only concern is to get the sensor data from the first sensor ( SENSOR1 ). This means that what is hidden or encapsulated into the macro (similar to C++) is of little interest once the interface is designed.

The sensor data might come from a telemetric signal that is bounced off of a satellite and then uploaded through the World Wide Web via an HTML page. It does not really matter. What's important is the value of the sensor signal at a given time. Therefore, the interface in the main code would consist of GET and SET macro statements. When the hardware architecture changes, only the macro definition is modified to correspond to the change. The software is completely void of any interface code with the exception of an interface header that links the outside world to the inside.

Macros are useful when applied correctly. Conversely, they can be your worst enemy, unmercifully enslaving you to agonizing hours of debugging. Ignorance is not bliss in this case.

## 3.36 Using the preprocessor

Don't use the preprocessor for defining complex macros.

Of course, the word "complex" is open for interpretation. It should therefore be defined within your development team. The operation of the preprocessor is poorly defined in the C coding standard, so its operation is at the mercy of the compiler writers. The preprocessor does have some valid uses, which, when applied properly, can help increase the maintainability and readability of the code.

The preprocessor is useful for defining "manifest constants" and for pre-calculating constants. This feature allows the user to force the preprocessor to calculate mathematical expressions into a constant type. The value of that type then gets replaced throughout the scope of the code. Scaling and conversions are common uses for this feature.

For example, the conversion from miles per hour (MPH) to kilometers per hour (KPH) is required for a vehicle parameter that is given in miles per hour:

```
#define MPH_TOP_VEHICLE_SPEED (154) /*MPH*/
#define KPH_TOP_VEHICLE_SPEED (MPH_TOP_VEHICLE_SPEED * 1.6) /*KPH*/
```

The calculation is carried out using the highest precision of the preprocessor, usually floating point, and can then be cast to the required type. Despite the simplistic example, one can envision multiple conversions and scalings within one #define . When the parameter for top vehicle speed changes, the intermediate constant calculations are automated by the preprocessor and the change would be propagated throughout the scope of the code.

Let's consider another example. We will allow the preprocessor to do calculations in floating point and then convert the constant to the format in which it will be used. The formula for the area of a triangle is half of the base times the height. Let's assume that for this application, the base and height are known system parameters. The base is equal to 10cm. The height is equal to 5cm. You could either code the area of the triangle directly:

```
#define TRI_AREA (25)
```

Or you could define the base and height separately:

```
#define TRI_BASE (10.0) /* cm */
#define TRI_HEIGHT (5.0) /* cm */
```

and then define TRI_AREA as:

```
#define TRI_AREA (0.5 * TRI_BASE * TRI_HEIGHT)
```

The value TRI_AREA, is calculated by the preprocessor with the maximum precision available. TRI_AREA can then be type cast in the code or in the #define as follows:

```
#define US_TRI_AREA (unsigned short) (0.5 * TRI_BASE * TRI_HEIGHT)
```

Since the area will always be positive, an unsigned short type is used. The advantage to this method is that if the base or height of the triangle changes, you don't have to recalculate the area. All you have to do is change the parameters of the triangle in the definition. This technique becomes highly valuable when you have constants that are a direct result of complex formulas based on physical system parameters.

Limit the definition of macros in the preprocessor to highly reuseable, simple functions.

## 3.37 Conditional Compilation

Conditional compilation is useful for things like machine-dependencies, debugging, and for setting certain options at compile-time.
Various controls can easily combine in unforeseen ways:

- If you use #ifdef for machine dependencies, make sure that when no machine is specified, the result is an error, not a default machine. The #error directive comes in handy for this purpose.
- If you use #ifdef for optimizations, the default should be the unoptimized code rather than an uncompilable or incorrect program. Be sure to test the unoptimized code.

## 3.38 Parenthesizing

Always use full parenthesizing for macros and equations.

To reduce the possibility of error when using the preprocessor, always use full parenthesizing. Most people who use parentheses when defining negative numbers don't use them when defining positive numbers. Parenthesizing all numbers is beneficial for the following reasons. First, nobody has ever gotten into trouble by using too many parentheses; it's the placement of the parentheses that introduces errors.

Second, let's consider the following example of defining a positive number:

```
#define POS_NO 10
```

When the preprocessor executes it will find the location of POS_NO and replace it with the value of 10. Since the operation of the preprocessor is weakly defined, the compilation of the following line of code is entirely dependent on the compiler:

```
y = .POS_NO;
```

The decimal point before POS_NO might be treated as just that, a decimal point and the line would appear to the compiler as:

```
y = .10;
```

If you're fortunate, your compiler would freak out at this implementation and generate an error message (in capital letters because it likes to shout when you do stupid things).

This is the best case scenario. The compiler might very well compile this line of code and grant you the luxury of finding the problem in the debugging phase. The whole problem could be easily avoided by defining the POS_NO parameter in parentheses:

```
#define POS_NO (10)
```

In all cases, the preprocessor would replace POS_NO with (10) and the code would appear to the compiler as:

```
y = .(10);
```

In this case the compiler should flag an error. It's worth noting here that the period or decimal point (.) is defined in the C language as an operator with similar attributes and constraints as other operators.
Now look at the following example:

```
#define MAX(a,b) a > b ? a : b /* WRONG */

x = 1;
y = 2;
/* WRONG: result=1 ('>' has more precedence than '&') */
result = MAX(x & 0xffff, y);
```

## 3.39 Macro instead of tiny functions

Use #defined macros instead of commonly used tiny functions.

Sometimes the bulk of CPU usage can be tracked down to a small external function being called thousands of times in a tight loop. Replacing it with a macro to perform the same job will remove the overhead of all those function calls, and allow the compiler to be more aggressive in it's optimization.

## 3.40 #ifdef and #if

A word of caution for those who use #ifdef instead of #if. A common use for the #if statement is to strap in and out portions of code that are configuration dependent. It would seem tempting to use the #ifdef , which calculates to TRUE if the variable is defined. Consider the following example.

In a configuration header file, the following define statements exist:

```
#define FOUR_WHEEL_DRIVE (0)
#define REAR_WHEEL_DRIVE (0)
#define FRONT_WHEEL_DRIVE (1)
```

The configuration file allows the programmer to choose between the three different vehicle configurations that are available for the software product. Each configuration consists of unique straps or patches that must be included at compile time. Now suppose the programmer wants to strap out a piece of code using the defined configuration as follows:

```
#ifdef REAR_WHEEL_DRIVE
    { code block A }
#endif
#ifdef FRONT_WHEEL_DRIVE
    { code block B }
#endif
```

Both blocks of code would be included at compile time because the #ifdef only interrogates the variable name for a definition. A definition of zero (0) would still flag a logic TRUE to the preprocessor.
Avoid this by eliminating the use of the #ifdef and #ifndef and use #if instead. A #undef operator is provided to undefine a variable if for some reason you crave the use of the #ifdef operator.

## 3.41 System parameters

When possible, tie your constants and parameters back to some physical unit.

Then if the dimensions of your external system change, you can change the physical unit constant defined in your header file so that all constants based on the physical unit will be recalculated at compile time. Of course, you have to check for overflows, but hopefully the software is defined well and you are given limits for the physical unit.

For example, let's say we have a 10-bit analog-to-digital voltage converter (A to D) with a full swing of zero to five volts. Let's say that the zero-point for the sensor in which we're interested is at +3.0V. You could manually calculate the zero point A-to-D value by taking 3/5 * 1023 and hard code the value 614 into the software:

```
#define SENSOR_ZERO_POINT (614)
```

If any part of the external architecture changes (voltage range, zero point of the sensor, A-to-D resolution) the software engineer must recalculate the A-to-D zero-point value manually. The value in which we're interested is the SENSOR_ZERO_POINT . If any part of the parameter of the system changes, the value of the SENSOR_ZERO_POINT is automatically calculated.

The same concept holds true for constants that are time-based. You should tie them back to the physical oscillator or internal clock frequency. When you change processing speeds, all you have to do is alter the clock frequency and all time-based constants are recalculated.

# Chapter 4

# How it works

## 4.1 Endianess

There are two different ways to store multibyte data values in memory: big-endian and little-endian.

- The big-endian convention dictates that the most significant byte of a value has the lowest address in memory.
- The little-endian convention is just the opposite: the least significant byte of a value must have the lowest address in memory.

So the terms "little-endian" and "big-endian" indicate which end of the multibyte value, the little end or the big end, is stored at the starting address of the value.

```
increasing memory address
---------------->
    A       A+1     <- BIG ENDIAN
+---+---+---+---+
| M S B | L S B |  16 bit integer
+---+---+---+---+
   A+1      A      <- LITTLE ENDIAN
<----------------
increasing memory address
```

The storage method used will vary according to the hardware platform you're on. For example, the Intel family of 32-bit processors is a little-endian platform, while PowerPC family is a big-endian platform.

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    unsigned long value = 0xABCDEF12;
    unsigned char *arr;

    arr = (unsigned char *)&value;

    printf("little-endian(x86): LSB has the lowest address in memory.\n");
    printf("big-endian (68k): MSB has the lowest address in memory.\n");

    printf("MSB-> %X <-LSB\n", value);

    printf("VAL=%X %X %X %X\n", arr[0], arr[1], arr[2], arr[3]);
    printf("ADR=%p %p %p %p\n", &arr[0], &arr[1], &arr[2], &arr[3]);

    return (0);
}
```

Arrays in C are always indexed from a low address to a high address. Thus, the first element of the array (i.e., arr[0]) also has the lowest address. The reason behind this is that arr[3] is the same as arr+3. In other words, the index is really an offset from the first element of the array.

Endianess is important because it makes transferring data between platforms difficult. Here is an example: suppose you are running an application that stores configuration data in a binary file. Also suppose that your current machine (an RS6000 workstation) crashes and you are forced to take the source code of the application and build a new copy on a backup machine (an Intel laptop). The backup machine's endianness is different from that of the original computer. When dealing with networking protocols the sending protocol stack and the receiving protocol stack must agree on the order in which the bytes of these multibyte fields will be transmitted. The Internet protocols use big-endian byte ordering for these multibyte integers, this order is also called "network byte order".

Another simple example to test the endianness of a platform by simply storing the value 1 (0x000001) in a long word and checking in which byte the bit is set:

```
#include <stdio.h>

int main() {
    unsigned int i1 = 0x01;
    unsigned int i2 = *(unsigned char *)&i1;

    if (i1 == i2) {
        printf("LITTLE ENDIAN\n");
    } else {
        printf("BIG ENDIAN\n");
    }
    return 0;
}
```

## 4.2 The balancing rule

Most of the time we don't care how the compiler internally balances and promotes the different variable types, as long as we get the desired result.

When we don't get the desired result we are rudely awakened to the tune of assembly-level debugging, trying to determine why a simple, straightforward equation does not function correctly.

In C, the promotion rule is as follows:

> Integral types of character, short integer, or an integer bit field and their signed and unsigned varieties will be converted to signed integer if the signed integer can represent all values of the original type; otherwise it is converted to unsigned integer.

So chars, shorts, and bit fields get converted to some type of integer (signed or unsigned) before an expression is executed internally. To avoid potential problems, explicitly cast non integer operands in expressions.

When using char, short, or bit fields types in expressions, always cast them to the appropriate integer type to reduce the possibility of conversion errors. If you don't, C automatically converts these integral types to integer before executing the expressions, and the conversion might not give the desired result.

Even if you verify the behavior of your compiler for conversions of these types, when you switch to another compiler for a different program, its behavior might be different. Let's look at another example that will help illustrate the problem. Let's consider the following example:

```
unsigned char UC;
UC = 0;
if (UC == ~0xFF) { ... }
```

Will the conditional expression result in true or false? From the C balancing rule, both UC and the hex value 0xFF are converted to type signed integer (because their values can be contained in the signed data type) before being evaluated. So the following steps are performed:

- UC is cast to signed short 00:00.
- The hex value 0xFF is cast to a signed short 00:FF.
- The not operator functions on the integer type, which results in FF:00.
- The comparison is then done between 00:00 and FF:00

So the expression from the view of the compiler takes on the following after the negate operator is done:

```
if ((signed short)0x0000 == (signed short)0xFF00) { ... }
```

This expression would result in an evaluation of false. The problem could easily be avoided by casting the constant to the desired type:

```
if (UC == (unsigned char)~0xFF) { ... }
```

This will effectively truncate the top byte of the integer and the comparison will be done on character types. When in doubt, cast it out.

## 4.3 Precedence: use full parenthesizing

The "equal" (=) assignment operator is treated as any other operator in C, unlike other programming languages (such as Pascal and Ada) in which the assignment operator is unique. Consider the following equation:

```
y = m * x + b;
```

Assuming that the algorithm author's intent is that the multiplication should execute first, the line of code should be written as:

```
y = ((m * x) + b;
```

By full parenthesizing every expression, you'll eliminate the question of when to and when not to do it. It also aids in documenting the programmer's intent. Maybe the programmer wanted the sum of 'x' and 'b' to be executed before the multiplication. The equation would then be written as:

```
y = (m * (x + b));
```

Full parenthesizing should be second nature, especially in a safety-critical system. Too many software bugs are a direct result of incorrect precedence assumption and/or poor parenthesizing. The solution for eliminating precedence problems is easy: use full parenthesizing.

## 4.4 Floating Point

When I set a float variable to, say, 3.1, why is printf printing it as 3.0999999?

Most computers use base 2 for floating-point numbers as well as for integers. Although 0.1 is a nice, polite-looking fraction in base 10, its base-2 representation is an infinitely-repeating fraction (0.0001100110011...), so exact decimal fractions such as 3.1 cannot be represented exactly in binary.

Depending on how carefully your compiler's binary/decimal conversion routines (such as those used by printf) have been written, you may see discrepancies when numbers not exactly epresentable in base 2 are assigned or read in and then printed (i.e. converted from base 10 to base 2 and back again).

## 4.5 Floating Point Equality

What's a good way to check for "close enough" floating-point equality?

Since the absolute accuracy of floating point values varies, by definition, with their magnitude, the best way of comparing two floating point values is to use an accuracy threshold which is relative to the magnitude of the numbers being compared. Rather than

```
double a, b;
...
if(a == b) { ... } /* WRONG */
```

use something like:

```
#include <math.h>
if ( fabs(a - b) <= epsilon * fabs(a) ) { ... }
```

where "epsilon" is a value chosen to set the degree of "closeness" (and where you know that a will not be zero).

## 4.6 memcpy() and memmove()

Remember:

- memmove() offers guaranteed behavior if the source and destination arguments overlap.
- memcpy() makes no such guarantee, and may therefore be more efficiently implementable.

When in doubt, it's safer to use memmove().

## 4.7 ++i and i++

Remember:

- ++i adds one to the stored value of 'i' and "returns" the incremented value to the surrounding expression.
- i++ adds one to 'i' but returns the prior unincremented value.

So if n is 3, then:

- x = n++ sets x to 3
- x = ++n sets x to 4

in both cases 'n' becomes 4.

## 4.8 Logical operators: && || !

These operators give TRUE (i.e. non zero value) when the condition succeeds and FALSE (i.e. zero) when the condition fails.

There is a special "short-circuiting" exception for these operators: the right-hand side is not evaluated if the left-hand side determines the outcome (i.e. is true for || or false for &&).

Therefore, left-to-right evaluation is guaranteed, as it also is for the comma operator. Furthermore, all of these operators introduce an extra internal sequence point.

By the way, if you are working with a language or some oddball compiler that does not support short circuit evaluation, then you can simulate it as follows:

```
if((value < 11) && (value%2 == 0)){
    /* do some work */
}
```

can be translated in:

```
if(value < 11){
    if(value%2 == 0){
        /* do some work* /
    }
```

## 4.9 Bitwise operators

The followings are bitwise operators that can only be applied to integers:

```
\&  bitwise and
|   bitwise inclusive or
\^  bitwise exclusive or
\~  one's complement
>>  right shift
<<  left shift
```

## 4.10 main()

The correct declaration of main() is either:

```
int main()
int main(void)
int main(int argc, char *argv[])
```

## 4.11 cdecl program

Use the cdecl program, which turns English into C and viceversa:

```
cdecl> declare a as array of pointer to function returning pointer to function returning pointer to char
char *(*(*a[])())()
```

cdecl can also explain complicated declarations, help with casts, and indicate which set of parentheses the parameters go in (for complicated function definitions, like the one above).

## 4.12 ++ and - -

When the increment or decrement operator is used on a variable in a statement, that variable should not appear more than once in the statement because order of evaluation is compiler-dependent.

Do not write code that assumes an order, or that functions as desired on one machine but does not have a clearly defined behavior:

```
int i = 0, a[5];
a[i] = i++;  /* assign to a[0] or a[1]? */
```

## 4.13 The Precedence Rule for Understanding C Declarations

Here it is:

1. Declarations are read by starting with the name and then reading in precedence order.
2. The precedence, from high to low, is:
   (a) parentheses grouping together parts of a declaration
   (b) the postfix operators:
      - parentheses ( ) indicating a function, and
      - square brackets [ ] indicating an array.
   (c) the prefix operator: the asterisk denoting "pointer to".
3. 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.

An example of solving a declaration using the Precedence Rule:

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

Solving a Declaration Using the Precedence Rule

1. First, go to the variable name, "next", and note that it is directly enclosed by parentheses.
2. So we group it with what else is in the parentheses, to get "next is a pointer to..."
3. Then we go outside the parentheses, and have a choice of a prefix asterisk, or a postfix pair of parentheses.
4. Rule 2.b tells us the highest precedence thing is the function parentheses at the right, so we have "next is a pointer to a function returning"
5. Then process the prefix "*" to get "pointer to".
6. Finally, take the "char * const", as a constant pointer to a character.

Then put it all together to read:

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

and we're done.

## 4.14 Operator precedence

Equality operators have precedence over bitwise operators.

```
if ( x & MASK == VALUE ) /* WRONG: equivalent to (x & (MASK==VALUE)) */
if ( (x & MASK) == VALUE ) /* right */
```

# Chapter 5

# Array and Pointers

## 5.1 Array?

Well, an array is a collecion of data items all of the same type. C always allocates the array in a single block of memory. Array initialization can be done in various manner:

```
/*
int a[5];

/* The number of init values is exactly equal to the number of array elements. */
int b[5] = {1, 2, 3, 4, 5};

/* The number of init values is less than to the number of array elements.
* The remaining elements are initilized to zero. */
int c[5] = {1, 2};

/* The number of array elements is not specified.
* The size of the array is fixed by the ini values. */
int d[5] = {1, 2, 3, 4, 5};
```

The elements are accessed by integers ranging from 0 to size-1, and there is no bounds checking.

So remember:

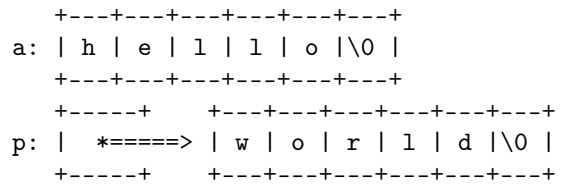
> The first element in an array is indexed by 0.

## 5.2 Pointers?

As usual, a picture is worth a thousand words. The declarations

```
char a[] = "hello";
char *p = "world";
```

Would initialize data structures which could be represented like this:




A string literal can be used in two slightly different ways:

- As an array initializer (as in the above declaration of char a[]), it specifies the initial values of the characters in that array.
- Anywhere else, it turns into an unnamed, static array of characters, which may be stored in read-only memory, and which therefore cannot necessarily be modified.

In an expression context, the array is converted at once to a pointer, as usual, so the second declaration initializes 'p' to point to the unnamed array's first element.

```
#include <stdio.h>

int main() {
    float fl = 3.14;
    unsigned int fl_addr = (unsigned int)&fl;
    float fl_content = *(float*)fl_addr;
    float* fl_ptr = &fl;

    printf("float fl = 3.14;\n");
    printf("unsigned int fl_addr = (unsigned int)&fl;\n");
    printf("float fl_content = *(float*)fl_addr;\n");
    printf("float* fl_ptr = &fl;\n");
    printf("fl       =%f\n",fl);
    printf("fl_addr  =%d\n",fl_addr);
    printf("fl_content =%f\n",fl_content);
    printf("fl_ptr   =%d\n",fl_ptr);

    return 0;
}
```

## 5.3 const char *p and char * const p

- "const char *p" (which can also be written "char const *p") declares a pointer to a constant character (you can't change any pointed-to characters).
- "char * const p" declares a constant pointer to a (variable) character (you can't change the pointer).

Read these "inside out" to understand them.

## 5.4 Pointers and Arrays

Much of the confusion surrounding arrays and pointers in C can be traced to a misunderstanding of the "equivalence of pointers and arrays".

Saying that arrays and pointers are "equivalent" means neither that they are identical nor even interchangeable. What it means is that array and pointer arithmetic is defined such that a pointer can be conveniently used to access an array or to simulate an array. Specifically, the cornerstone of the equivalence is this key definition:

An lvalue of type array-of-T which appears in an expression decays (with three exceptions) into a pointer to its first element; the type of the resultant pointer is pointer-to-T.

That is, whenever an array appears in an expression, the compiler implicitly generates a pointer to the array's first element, just as if the programmer had written &a[0]. (The exceptions are when the array is the operand of a sizeof or & operator, or is a string literal initializer for a character array.)

As a consequence of this definition, the compiler doesn't apply the array subscripting operator [] that differently to arrays and pointers, after all. In an expression of the form a[i], the array decays into a pointer, following the rule above, and is then subscripted just as would be a pointer variable in the expression p[i] (although the eventual memory accesses will be different).

If you were to assign the array's address to the pointer:

```
p = a;
```

then p[3] and a[3] would access the same element.

## 5.5 null pointer

A null pointer does not point to any object. Thus it is illegal to use a null pointer for any purpose other than assignment and comparison.

Never redefine the NULL symbol. The NULL symbol should always have a constant value of zero. A null pointer of any given type will always compare equal to the constant zero, whereas comparison with a variable with value zero or to some non-zero constant has implementation-defined behaviour.

Dereferencing a null pointer may cause strange things to happen.

## 5.6 Pointer Arithmetic

C allows pointer arithmetic (addition and subtraction). Suppose we have:

```
char array[5];
char *array_ptr = &array[0];
```

In this example:

```
 *array_ptr   is the same as array[0]
*(array_ptr+1) is the same as array[1]
*(array_ptr+2) is the same as array[2]
```

and so on. Note the use of parentheses. However, (*array_ptr)+1 is not the same as array[1]: the +1 is outside the parentheses, so it is added after the dereference. So (*array_ptr)+1 is the same as array[0]+1.

At first glance, this method may seem like a complex way of representing simple array indices. The elements of an array are assigned to consecutive addresses.

For example: array[0] may be placed at address 0xff000024, then array[1] would be placed at address 0xff000025, and so on.

This structure means that a pointer can be used to find each element of the array. The next example prints out the elements and addresses of a simple character array.

```
#include <stdio.h>

#define ARRAY_SIZE 10
char array[ARRAY_SIZE] = "0123456789";

int main() {
    int index;

    for (index = 0; index < ARRAY_SIZE; ++index) {
        printf("&array[index]=0x%p (array+index)=0x%p array[index]=0x%x\n",
            &array[index], (array+index), array[index]);
    }
    return (0);
}
```

## 5.7 Dangling pointer

Be aware that if you return a pointer to an object allocated within a function, that object will be destroyed when the function returns:

```
char* test() {
    char str[100];
    [..]
    return str;   /* WRONG! str points within temporary stack */
}
```

## 5.8 Array subscripting is commutative

> Array subscripting is commutative in C.

This curious fact follows from the pointer definition of array subscripting, namely that "a[e]" is identical to "*((a)+(e))", for *any* two expressions 'a' and 'e', as long as one of them is a pointer expression and one is integral.

## 5.9 Pointers as Function Arguments

```
#include <stdio.h>

void inc_count(int *count_ptr) {
    (*count_ptr)++;
}
```

```
int main(){
    int  count = 0;

    while (count < 10) {
        inc_count(&count);
        printf("%d\n",count);
    }
    return (0);
}
```

## 5.10 Pointer to buffer as Function Argument

Somethimes you have a function which reads some buffer and you need that function to return that buffer. If you don't want to use a global buffer, you can do as the following example:

```
#include <stdio.h>

/* buffer is a pointer to a pointer of char */
int fill_buffer(char **buffer) {
    char *content = "0123456789-----987654321";
    int len = 25;

    (*buffer) = (char *)calloc(len, sizeof(char));
    memcpy(*buffer, content, len);
    return len;
}


int main(){
    int len;
    char *buffer;

    len = fill_buffer(&buffer);
    printf("%d:%s\n", len, buffer);
    free(&buffer);
    return (0);
}
```

## 5.11 The use of pointers

C is unusual in that it allows pointers to point to anything. Pointers are sharp tools, and like any such tool, used well they can be delightfully productive, but used badly they can do great damage. Pointers have a bad reputation in academia, because they are considered too dangerous, dirty somehow. But I think they are powerful notation, which means they can help us express ourselves clearly.

Consider: when you have a pointer to an object, it is a name for exactly that object and no other. That sounds trivial, but look at the following two expressions:

```
np
node[i]
```

The first points to a node, the second evaluates to (say) the same node. But the second form is an expression; it is not so simple.

To interpret it, we must know what node is, what 'i' is, and that 'i' and 'node' are related by the (probably unspecified) rules of the surrounding program. Nothing about the expression in isolation can show that 'i' is a valid index of 'node', let alone the index of the element we want.

If 'i' and 'j' and 'k' are all indices into the node array, it's very easy to slip up, and the compiler cannot help. It's particularly easy to make mistakes when passing things to subroutines: a pointer is a single thing; an array and an index must be believed to belong together in the receiving subroutine.

An expression that evaluates to an object is inherently more subtle and error-prone than the address of that object. Correct use of pointers can simplify code:

```
parent->link[i].type
vs.
lp->type.
```

If we want the next element's type, it's

```
parent->link[++i].type
or
(++lp)->type
```

i advances but the rest of the expression must stay constant; with pointers, there's only one thing to advance.

Typographic considerations enter here, too. Stepping through structures using pointers can be much easier to read than with expressions: less ink is needed and less effort is expended by the compiler and computer.

A related issue is that the type of the pointer affects how it can be used correctly, which allows some helpful compile-time error checking that array indices cannot share. Also, if the objects are structures, their tag fields are reminders of their type, so

```
np->left
```

is sufficiently evocative; if an array is being indexed the array will have some well-chosen name and the expression will end up longer:

```
node[i].left
```

Again, the extra characters become more irritating as the examples become larger.

As a rule, if you find code containing many similar, complex expressions that evaluate to elements of a data structure, judicious use of pointers can clear things up. Consider what

```
if(goleft)
    p->left=p->right->left;
else
    p->right=p->left->right;
```

would look like using a compound expression for p. Sometimes it's worth a temporary variable (here p) or a macro to distill the calculation.

## 5.12 Pass structures by reference

Whenever possible, pass structures by reference (i.e. pass a pointer to the structure), otherwise the whole darn thing will be copied onto the stack and passed, which will slow things down a tad (I've seen programs that pass structures several megabytes in size by value, when a simple pointer will do the same thing).

Functions receiving pointers to structures as arguments should declare them as "pointer to const" if the function is not going to alter the contents of the structure, e.g.

```
void print_data(const bigstruct *data_pointer) {
    ...printf contents of structure...
}
```

This example informs the compiler that the function does not alter the contents (pointer to constant structure) of the external structure, and does not need to keep re-reading the contents each time they are accessed. It also ensures that the compiler will trap any accidental attempts by your code to write to the read-only structure.

## 5.13 const pointers

Declaring constant pointers is a little tricky. For example, the declaration:

```
const int result = 5;
```

tells C that result is a constant so that:

```
result = 10; /* Illegal */
```

is illegal. The declaration:

```
const char *my_ptr = "One-Two-Three";
```

does not tell C that the variable my_ptr is a constant. Instead, it tells C that the data pointed to by my_ptr is a constant. The data cannot be changed, but the pointer can.

Again we need to make sure we know the difference between "things" and "pointers to things".

- What's my_ptr? A pointer.
- Can it be changed? Yes, it's just a pointer.

- What does it point to? A const char array.
- Can the data pointed to by my_ptr be changed? No, it's constant.

In C this is:

```
my_ptr = "Fifty-One"; /* Legal (my_ptr is a variable) */
*my_ptr = 'X';       /* Illegal (*my_ptr is a constant) */
```

If we put the const after the * we tell C that the pointer is constant. For example:

```
char *const aname_ptr = "Test";
```

- What's aname_ptr? It is a constant pointer.
- Can it be changed? No.
- What does it point to? A character.
- Can the data we pointed to by aname_ptr be changed? Yes.

```
aname_ptr = "New"; /* Illegal (aname_ptr is constant) */
*aname_ptr = 'B'; /* Legal (*aname_ptr is a char) */
```

Finally, we can put const in both places, creating a pointer that cannot be changed to a data item that cannot be changed:

```
const char *const atitle_ptr = "Title";
```

## 5.14 Strings

In C strings are array of char.

The last character of a string must the special character called "null" and written as '
0'. The "null" character is guaranteed to be zero. Depending on how strings are initialized you need to manually add the "null" character:

```
   +---+---+---+---+
a: | f | o | o |\0 |           char a[4] = {'f','o','o','\0'};
   +---+---+---+---+
   +---+---+---+---+
b: | b | a | r |\0 |           char b[4] = "bar";
   +---+---+---+---+
   +---+---+---+---+---+---+---+
c: | o | t | h | e | r |\0 |   char c[] = "other";
   +---+---+---+---+---+---+---+
   +---+---+---+
d: | b | b | b |               char d[3] = "bbb";
   +---+---+---+
```

Note that "d" is not a string but only an array of char.

## 5.15 How Not to Use Pointers

The major goal of this book is to teach you how to create clear, readable, maintainable code. Unfortunately, not everyone has read this book and some people still believe that you should make your code as compact as possible. This belief can result in programmers using the ++ and – operators inside other statements.

The program copies a string from the source (p) to the destination (q).

```
Example: Cryptic Use of Pointers.
void copy_string(char *p, char *q) {
   while (*p++ = *q++);
}
```

Given time, a good programmer will decode this. However, understanding the program is much easier when we are a bit more verbose, as the next example.

Example: Readable Use of Pointers.

```
/*****************************************************
 * copy_string -- Copies one string to another.
 *     dest -- Where to put the string
 *     source -- Where to get it
 *****************************************************/
void copy_string(char *dest, char *source) {
    while (1) {
        *dest = *source;
        /* Exit if we copied the end of string */
        if (*dest == '\0')
            return;
        ++dest;
        ++source;
    }
}
```

## 5.16 Array bounds

Check the array bounds of all arrays, including strings, since where you type "fubar" today someone someday may type "floccinaucinihilipilification". Robust production software should not use gets().

The fact that C subscripts start from zero makes all kinds of counting problems easier. However, it requires some effort to learn to handle them.

## 5.17 Pointer qualifier

The pointer qualifier, '*', should be with the variable name rather than with the type.

```
char *s, *t, *u;
```

instead of

```
char* s, t, u;
```

The latter statement is not wrong, but is probably not what is desired since 't' and 'u' do not get declared as pointers.

# Chapter 6

# Type Conversion

## 6.1 A number beginning with 0 (zero)

A number beginning with 0 (zero) is interpreted as octal, not decimal.

```
int limit = 0100; /* oops, limit is assigned 64 (decimal) */
```

## 6.2 sizeof()

sizeof() means how many bytes (not elements) and includes trailing null byte of a string.

## 6.3 char

char is signed, and will be promoted to a signed integer (not unsigned).

## 6.4 Conversions

Be careful when doing conversions from strings to numbers, and ensure that you are using the correct function. At one point I wrote some code that looked like:

```
unsigned long bytes;
...
bytes = (unsigned long)atoi(str); /* save bytes for later */
```

Even though I was casting as a long, the atoi() function converts to int, and not long. The code worked for a fair amount of time, until the value stored in 'str' grew more than the size of an int, and then suddenly 0 was returned.

The atoi() will not produce an error on fail, but simply bails and returns 0. To solve this problem use atol() instead.

## 6.5 Explicit signed or unsigned

> Explicitly define your types as signed or unsigned.

If you don't specify the signed or unsigned type, the C compiler will select one for you, which might not achieve the desired result.

In most cases, when defining a variable as an integer, the compiler assigns the default type of signed short integer unless the value cannot be contained by that data type. When defining a bit field or character type, the default type is entirely compiler dependent.

Explicitly defining your data types as signed or unsigned increases the portability of the code. What you might consider to be an unsigned character type in your code might actually be viewed by the compiler as a signed character type. By allowing the compiler to choose for you, you're essentially giving up your first amendment right as a software developer, which states that software professionals are more capable than compilers at making decisions. What about a signed bit field of length one?

## 6.6 Explicit cast

Explicitly cast mixed precision arithmetic operands in expressions.

In expressions, the sub-expressions are evaluated at the appropriate operand precision. The desired result may not be achieved if the resultant precision is greater than the expressions operand precision. To eliminate this error, explicitly cast the operands to the final precision of the result.

For example:

```
signed short v1 = 1;
signed short v = 2;
float result;
result = v1 / v2;
```

Here "result" incorrectly calculates to zero because the sub-expression is evaluated with integer precision. The expression:

```
result = (float) (v1 / v2);
```

also incorrectly evaluates to zero for the same reason. The sub-expression "v1 / v2" is calculated with integer precision and then the integer result is cast to floating precision. Consider the following:

```
result = (float) v1 / v2;
or:
result = (float) v1 / (float) v2;
```

Both evaluate to the correct result (0.5). Explicitly casting one of the operands in the sub-expression to the desired precision of the result forces the sub-expression to be evaluated at the resultant precision.

XXX: INSERIRE CODICE ESEMPIO.

## 6.7 Portability

Reusable code-everyone talks about it, yet when you're asked to reuse someone else's code, a chill runs up your spine. Let's face it, at one time or another most of us had to reuse a piece of code that was supposed to be portable. In reality, it wasn't.

And we all remember the joy of debugging someone else's code. This section will address some guidelines for designing code that is targeted for portability. One of the weaknesses of the C standard is in defining the sizes of types. C specifies that:

```
char >= 8 bits
short >= 16 bits
int  >= 16 bits
long >= 32 bits
char <= short <= int <= long
```

Portability of types becomes an issue in the C language because the compiler writers must determine the size of types based on these constraints. The solution is to define replacements for standard C types.

The replacement types are used throughout the code for casting and defining variables. When you switch to a different compiler, just change the header file to reflect the type sizes.

## 6.8 Max and Min limits

"limits.h": this standard header contains the definition of a number of costants giving the maximum and minimum sizes of various kinds of integers.

"float.h": contains constants relating to floating point types, giving the maximum and minimum sizes and accuracy of each kind of floats.

If you don't have a "limits.h" file for your compiler, define the limits of your types in a separate header file. Even if you have a "limits.h" file, it would be good to redefine your limits based on your defined types.

If your limits change due to type size changes, you need only change the header file for these limits. A typical limits file definition consists of the following:

```
#define BYTE_MIN (-127)      /* 8-bit type */
#define BYTE_MAX (127)
#define UBYTE_MAX (255)
#define WORD_MIN (-32767)    /* 16-bit type */
```

```
#define WORD_MAX (32767)
#define UWORD_MAX (65535)
#define LONG_MIN (-2147483647) /* 32-bit type */
#define LONG_MAX (2147483647)
#define ULONG_MAX (4294967295)
```

## 6.9 Floating Point Rounding

The simplest and most straightforward way to round numbers is:

```
(int)(x + 0.5)
```

This technique won't work properly for negative numbers, though. So you could use something like this:

```
(int)(x < 0 ? x - 0.5 : x + 0.5)
```

## 6.10 Floating point to integer

When you allow the macros to pre-calculate constants using the maximum resolution of the preprocessor, which is typically done in floating point, you must either truncate the result or round it up or down before stuffing it into an integer variable (assuming that you are using integer math in your software).

If the accuracy of the variable or constant forces you to round to the nearest integer you would either add or subtract 0.5 from the number and then truncate it.

The possibility for error arises if you're trying to round signed numbers. Let's take the following example:

```
#define TRAJECTORY (SPEED - (WEIGHT * 1.67584))
```

Realistically, the shuttle's orbital trajectory would probably be done in floating point, but stick with me here while I use the absurd to illustrate the practical. If we desired to round the number and cram it into an integer constant, the following rules would apply:

- If the number $>= 0$, then add 0.5 and truncate.
- If the number $< 0$, then subtract 0.5 and truncate.

Let's assume that the algorithm designer uses constants for the shuttle cruise speed and weight that result in a positive trajectory. The #define could be coded as:

```
#define SL_TRAJECTORY (signed long) (SPEED - (WEIGHT * 1.67584) + 0.5)
```

Years later, the shuttle cruise speed is reduced resulting in a negative shuttle orbital trajectory constant. The rounding in this example is done incorrectly and the shuttle crashes into the earth at about 2000 MPH. Oops! To eliminate this problem, create macros for doing the rounding.

Now when the preprocessor calculates the equation, the macro checks for the sign of the result and then adds or subtracts 0.5 as appropriate, and the government doesn't have to raise our taxes to pay for the mistake:

```
#define SL_TRAJECTORY SL_RND(SPEED - (WEIGHT * 1.67584))
```

## 6.11 No assumptions about type size

> Never make any assumptions about the size of a given type, especially pointers.

When char types are used in expressions most implementations will treat them as unsigned but there are others which treat them as signed. It is advisable to always cast them when used in arithmetic expressions.

# Chapter 7

# Tips

## 7.1 sprintf(): itoa()

To convert numbers to strings (the opposite of atoi) just use sprintf(). You can obviously use sprintf() to convert long or floating point numbers to strings as well (using %ld or %f).

## 7.2 printf(): variable field width

To implement a variable field width with printf you can do as follow:

```
printf("%*d", width, x);.
```

## 7.3 printf(): flush

To force printf() to immediately display a string: Use fprintf():

```
fprintf( stdout, ... );
fflush( stdout );
```

## 7.4 Flushing Output Buffer

When an application terminates abnormally, the tail end of its output is often lost.

The application may not have the opportunity to completely flush its output buffers. Part of the output may still be sitting in memory somewhere and is never written out. On some systems, this output could be several pages long.

Losing output this way can be misleading because it may give the impression that the program failed much earlier than it actually did. The way to address this problem is to force the output to be unbuffered, especially when debugging. The exact incantation for this varies from system to system but usually looks something like this:

```
setbuf(stdout, (char *) 0);
```

This must be executed before anything is written to stdout. Ideally this could be the first statement in the main program.

## 7.5 Bit-field structs

Bit-field structs aren't portable.

## 7.6 Debugging: lots of printf() statements

C Elements of Style suggests prefixing all comments with "##" for easy search and removal. This is a good idea. I also like to keep lots of:

```
if (debug) printf("...")
```

around as well, so that when my program is started with the "-debug" flag, useful information is displayed. Either way, if you don't know where something is crashing, or what the value is in a variable (never mind what it should be), use a printf().

## 7.7 Debugging: use #ifdef

Remove large sections of code for with #ifdef QQQ. This is another suggestion from C Elements of Style, and a good one. Instead of commenting out, or worse, deleting, wrap it as:

```
#ifdef QQQ
    /* code that you don't want to run */
#endif QQQ
```

A couple of notes:
If you don't like QQQ (suggested for the fact there is little chance of it being defined and it's search-ability), use #ifdef UNDEF, which should never be defined!
The QQQ on the #endif is not needed, and is only there for a visual reminder of what you are ending. Some older compilers may puke on this however. If they do, just stick with plain old #endif.

You can resort to using exit(0) to mark points of arrival.

If you have some particularily odd code (graphics, games or something) and you are having trouble tracking down exactly where the code is getting to, stick an exit(0) in.

If the program exits with error code of 0, you'll know it hit your statement. Linus Torvalds used a simliar technique when creating the first linux video drivers in assembler. He'd put reboot commands in and if the system just froze he knew his reboot statement hadn't been reached.

## 7.8 Debugging: use asserts

> Use asserts liberally in debug builds.

You normally don't want to put assert code in release builds, because you don't want the user to see your bug messages.

ANSI C provides assertion functions in <assert.h> if the symbol NDEBUG is defined somewhere before <assert.h> is included, then assert() will have no effect. Otherwise, it will print out a diagnostic message and abort the program if its argument evaluates to FALSE.

Since "0 == FALSE", you can use assert() on pointers to test them for non-NULL:

```
void my_function(char* foo) {
    assert(foo); /* same as assert(NULL != foo); */
}
```

## 7.9 Debugging: some useful macros

```
#if defined(DEBUG)
#define GOT_HERE    printf("Reached %i in %s\n", __LINE__, __FILE__)
#define SHOW(FMT, ARG) printf(#ARG " = " FMT "\n", ARG)
#else
#define GOT_HERE
#define SHOW(FMT, ARG)
#endif
```

## 7.10 Complexity

Most programs are too complicated - that is, more complex than they need to be to solve their problems efficiently.

Why?

Mostly it's because of bad design, but I will skip that issue here because it's a big one. But programs are often complicated at the microscopic level, and that is something I can address here.

- You can't tell where a program is going to spend its time. Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you've proven that's where the bottleneck is.
- Measure. Don't tune for speed until you've measured, and even then don't unless one part of the code overwhelms the rest.
- Fancy algorithms are slow when n is small, and n is usually small. Fancy algorithms have big constants. Until you know that n is frequently going to be big, don't get fancy. For example, binary trees are always faster than splay trees for workaday problems.
- Fancy algorithms are buggier than simple ones, and they're much harder to implement. Use simple algorithms as well as simple data structures. The following data structures are a complete list for almost all practical programs: array, linked list, hash table, binary tree.
  Of course, you must also be prepared to collect these into compound data structures. For instance, a symbol table might be implemented as a hash table containing linked lists of arrays of characters.
- Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming.

## 7.11 Handle errors and not bugs

There are lots of error conditions that happen in the normal life of a program. For instance, file not found, out of memory, or invalid user input. You should always handle these conditions gracefully (by re-prompting for a filename, by freeing memory or telling the user to quit other applications, or by telling the user there is an error in his input, respectively). However, there are other conditions which are not real error conditions, but are the result of bugs.

For example, say you have a routine which copies a string into a buffer, and no one is supposed to pass in a NULL pointer to the routine. You do not want to do something like this:

```
void copy_string(char* buffer, int size) {
    if (NULL == buffer) {
        return; /* quietly fail if NULL pointer */
    } else {
        strncpy(buffer, "Hello", size);
    }
}
```

Also, don't do something like this (some extremely fault-tolerant systems may be able to justify this, but 99% of applications can't):

```
void copy_string(char* buffer, int size) {
    if (NULL == buffer) {
        fprintf(stderr, "Error: NULL pointer passed to copy_string");
    } else {
        strncpy(buffer, "Hello", size);
    }
}
```

Instead, do something like this:

```
void copy_string(char* buffer, int size) {
    assert(buffer); /* complains and aborts in debug builds */
    strncpy(buffer, "Hello", size);
};
```

In a release build, if the user passes in a NULL pointer, this will crash. But rather than think "I never want my application to crash, therefore I will test all pointers for NULL", think "I never want my applications to crash, therefore I will put in asserts and find bugs that result in NULL pointers being passed to routines, so I can fix them before I ship this software".

## 7.12 scanf

scanf should never be used in serious applications. Its error detection is inadequate. Look at the example below:

```
#include <stdio.h>

int main(void) {
    int i;
    float f;

    printf("Enter an integer and a float: ");
    scanf("%d %f", &i, &f);

    printf("I read %d and %f\n", i, f);
    return 0;
}
```

Test run:
Enter an integer and a float: 182 52.38
I read 182 and 52.380001

Another test run:
Enter an integer and a float: 6713247896 4.4
I read -1876686696 and 4.400000

## 7.13 calloc() and malloc()

calloc(m, n) is essentially equivalent to:

```
p = malloc(m * n);
memset(p, 0, m * n);
```

The zero fill is all-bits-zero, and does *not* therefore guarantee useful null pointer values or floating-point zero values. free() is properly used to free the memory allocated by calloc().

# Bibliography


[1] Kernighan and Ritchie, *The C Programming Language*, Prentice-Hall, 2ndEd.

[2] Steve Oualline, *Practical C Programming*, O'Reilly, 3rdEd.

[3] Andrew Koenig, *C Traps and Pitfalls*, AT&T Bell Laboratories.

[4] Steve Summit, *C-FAQ-list*, http://www.eskimo.com/∼scs/C-faq/top.html.

[5] Shiv Dutta and Gary Hook, *Best practices for programming in C*, IBM, http://www-106.ibm.com.

[6] Rob Pike, *Notes on Programming in C*, http://www.di-mgt.com.au/src/pikestyle.html

[7] Peter van der Linden, *Expert C Programming - Deep C Secrets*, Prentice Hall

[8] Dinu Madau, *Rules for Defensive C Programming*, http://www.embedded.com

[9] Barnett, *C optimisation tutorial*

[10] Vishal Patil, *C, C++ programming tips*