Crash Course in C and assembly

(v.2006-08-04)

Željko Vrba

These notes are intended to serve as coding guidelines for the Operating Systems course at the University of Oslo. The text focuses on subjects that students have most trouble with while coding their solutions.

1 Introduction

A word of warning: these notes are *not* a "coding cookbook". There is very little code that can be directly used in the projects, and the explanations are often very brief and, for the inexperienced C programmer, insufficient for thorough understanding of the matter at hand.

Rather, the reader should look upon these notes as a guide for further study. The text presents common problems and misunderstandings that are encountered during correcting assignments. The problems are pointed out on artificial examples and briefly explained. The readers are expected to carefully study the examples and references and invest much of their own effort. Exercises are *not* mandatory; they are intended just as "food for thought" to the reader.

The following are **elementary guidelines** for students who don't have the patience to read (and understand!) the whole document.

- **Code simplicity and correctness** should always be **before** performance. First make it work, then make it work fast(er). To quote D. E. Knuth: "Premature optimization is the root of all evil (or at least most of it) in programming." **You are graded for correctness, NOT performance! Leave all optimizations for the competition.**

- Do not do more than is required by the assignment. Always try to find out the minimum needed to **correctly** accomplish the assignment task. **Less code – less debugging.** Code for fun only **after** you have completely implemented the assignment.

- Do not use inline assembly unless you absolutely have to (and you don't). If you still think that you absolutely do need it, you are probably trying to do something contrary to the previous points.

- Use pointers instead of integers to deal with memory addresses.

- **Error-checking** is important, especially when writing an OS!

C language

The following sections describe some specifics of the C language. First, we introduce some basic facts.

Hosted implementation makes available all library functions defined by the standard. Most of these functions require some support from the operating system, which is not available in our OS. Therefore, in our programs, we are allowed to use only what is available in the freestanding implementation.

Freestanding implementation makes available only functions and macros defined in a subset of standard C headers. These are `<float.h>`, `<limits.h>`, `<stdarg.h>`, `<stddef.h>`, and (available only in C99) `<stdint.h>`.

The `<stddef.h>` header defines the following constants and macros.

1. The null pointer constant `NULL`,

2. the `size_t` type which is an unsigned integer type large enough to contain any size on the given architecture (usually 32 bits on 32-bit architectures), and

3. `offsetof` macro which is used to calculate the offset of a particular member in a structure. *Exercise: Study in detail what the `offsetof` macro does, and implement your own.*

2 Variables

2.1 Static variables `static` keyword is used to declare function-scope variables whose value persists across calls.[1] [2]

2.2 Automatic variables are function-scope variables (also sometimes called *local*) declared without the `static` keyword are called *automatic variables.*

Example: static vs. automatic. In function `f`, local variable (`x`) is automatic, and variable (`y`) is static.

```
void f(void)
{
  int x = 1;
  static int y = 3;

  printf("%d %d\n", ++x, ++y);
}
```

The function `f()` will on its first execution print `2 4`, and on its second execution `2 5`. Variable `x` is initialized on each entry to `f()`, while `y` is initialized *only once*, before the program starts up. `y` is accessible only within the function `f` and all changes to it persist across function calls to `f`.

2.3 Storage allocation.

- The storage for automatic variables is automatically allocated and initialized on each function entry, and deallocated on function exit. Automatic variables are usually stored on the processor's stack.[3]

- The storage for static variables is allocated only once, at compilation time. They are also initialized only once, before the `main` function starts to run.

2.4 Recursive functions. Each recursive invocation of a recursive function will get a freshly initialized *copy* of automatic variables. Note that *all* recursive invocations of the function share the same (only!) copy of static variables.

2.5 lifetime. `static` variables exist as long as the program is running. Automatic variables exist only as long as the function they are defined in has not returned. The latter point can be a source of nearly impossible to find bugs, which arise when a function returns pointer to an automatic variable.

Example: unsafe function. When the `unsafe` function returns, the `x` variable is deallocated, so the caller receives a pointer pointing to invalid data. *Exercise: Why is the call to `g` safe?*

```
int *unsafe(void)
{
  int x = 12;
  g(&x);      /* SAFE */
  return &x;  /* !UNSAFE! */
}
```

[1] Variables can have function- or file-scope. This usage (the only described here) affects the variable's *storage class.*
[2] Another use of `static` is to influence the symbol *linkage.*
[3] The C standard does not mention stack explicitly. It might not even exist on certain processor architectures. The standard just specifies the semantics of automatic variables.

Example: safe function. The code listed for the `safe` function is *valid* since the variable `x` is `static`. This is a way to make a local static variable visible outside of the function.

```
int *safe(void)
{
  static int x = 12;
  return &x;
}
```

3 Calling convention

This term refers to semantics and mechanism of passing arguments to and returning values from *functions.*[4]

3.1 Function-call semantics. In C there are two basic rules:

1. **All arguments are passed by value.** This means that a *copy* of the argument is pushed onto the stack. Any changes made to arguments within the function will not be visible to its caller. Care should be taken to distinguish between changing the pointer and the value pointed to.

2. Array *decays into pointer to the first element* instead of being copied.[5]

Example: argument-passing. Study the following code and explanation carefully, for it is essential to understand the C language.

```
void f1(int x, int *y)
{
  ++x; ++y; ++*y;
}

void f2(int **z)
{
  ++*z;
}

void g(void)
{
  int a[3] = { 1, 2, 3 };
  int x = 10, y = 11, *z = a+1;

  f1(x, a);
  f1(x, z);
  f2(&z);
  --*z;
}
```

1. After the first call to `f1` we have `x == 10` and `a[1] == 3`. Notice how an array has effectively decayed into a pointer. Had the function been declared like this: `void f1(int x, int y[])`, the effect would have been the same. These two function declarations are *equivalent.*

2. After the second call to `f1` we have `x == 10`, `a[2] == 4`, while `z == a+1`, i.e. it still points to the second element of array `a`. Notice how an element of an array is indirectly changed through the pointer, while the value of the pointer itself is unchanged on return.

3. After the call to `f2`, `z` is changed and equals `a+2`. Therefore, after the `--*z` statements is executed, we have `a[2] == 3`.

3.2 Function-call mechanism. Consider a function with the prototype `int f(int x, int *y)` having two integer local variables `a` and `b`. Suppose that it is called as `x = f(z, &c)`. Once the frame pointer is set up, arguments and local variables are at *fixed offsets* with respect to the `EBP` register. **Figure 1** shows stack layout only for the default case. The layout can be different, depending on the compiler options; `-fomit-frame-pointer` is particularly often used as it frees the `EBP` register for other uses. This option makes the code faster, but also harder to debug.

[4] The reader should distinguish functions from *preprocessor macros* which don't really pass arguments, but perform simple textual substitution.
[5] This is somewhat imprecise when multi-dimensional arrays are considered.

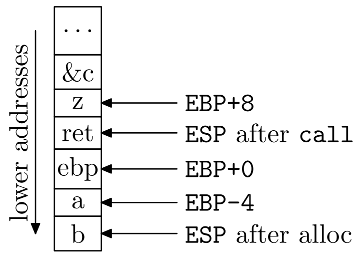

Figure 1 Stack diagram after executing the function prologue. Each "cell" is exactly 4 bytes (32 bits).

The caller pushes arguments in *right to left* order, and must clean them up after the function returns. The return address is automatically pushed by `call` and popped by `ret` instructions. Also, the called function must not modify certain registers.

4 Pointers.

Processors generally *do not distinguish* between integers and pointers. Interpretation of the register content depends on the instruction. In order to ease programming , integers and pointers are *distinct* data types in C. Do not convert between pointers and integers; always use pointers to access memory. Doing so has at least two benefits: (1) enhanced error checking, and (2) automatic pointer arithmetic.

4.1 Void pointer. In C, all pointers are implicitly convertible to an untyped pointer, `void*` and vice-versa. Explicit casts are neither needed nor desired. This esp. applies to storing the result of `malloc`. Pointer of type `void*` *cannot be dereferenced.*

4.2 Raw memory. If you need to access a portion of memory as a raw sequence of bytes, use `char*` or `unsigned char*` pointers.[6]

4.3 Size of char. The C standard *guarantees* that `sizeof(char) == sizeof(unsigned char) == 1`, so expression like `16*sizeof(char)` unnecessarily clutters the code as it *always* equals 16.

4.4 Integer types. When you *must* resort to conversion between pointers and integers, *always use* ***`unsigned`*** integer types. Otherwise, strange bugs can happen due to arithmetic sign-extensions. The recommended type to use is `uintptr_t`, defined in `<stdint.h>` when available.[7] Otherwise, the `size_t` type should be used. Both types are unsigned.

4.5 Initialization to fixed address. Sometimes, a pointer has to be initialized to a *specific memory location.* This is often the case when a program needs access to memory-mapped hardware registers.

Example: accessing video memory. The video memory starts at the physical address `0xB8000`, and is organized as a row-major array of *pairs* of bytes (16 bits) for one character. The bytes at even addresses specify *characters*, and bytes at odd addresses specify their *attributes*, e.g. color.

In this case a program might use the following definition:

```
unsigned short *videomem =
  (unsigned short*)0xB8000
```

4.6 Alignment. We say that the pointer is *aligned to the boundary of* n if its value is divisible by n. In almost all cases, n is a power of two.[8] For example, page tables on the x86 architecture must be aligned to the boundary of $2^{12} = 4096$ bytes. Since C does not allow bit operations on pointers, we must convert between pointers and integers.

The following functions align the pointer `p` to the next higher or lower address which is a mul-

[6] The difference between signed and unsigned integer types is not discussed here.
[7] This type is guaranteed to be large enough to store the value of a pointer without loss of information, whether it is a 32- or 64-bit architecture. This is a C99 feature, implemented by gcc, but might not be available in other compilers.
[8] Many RISC processors can perform only aligned loads from memory and throw exception when an unaligned load is attempted.

tiple of 2^n. If the pointer is already aligned, it is left unchanged.

```
void *ptr_align_down(void *p, unsigned n)
{
  uintptr_t pi = (uintptr_t)p;
  uintptr_t mask = (1 << n) - 1;
  return (void*)(pi & ~mask);
}

void *ptr_align_up(void *p, unsigned n)
{
  uintptr_t pi = (uintptr_t)p;
  uintptr_t  mask = (1 << n) - 1;
  return (void*)((pi+mask) & ~mask);
}
```

When the pointer is obtained as a result of `malloc`, it *must* be aligned to higher address. Aligning to lower address would corrupt `malloc` internal structures. You also have to *allocate* $2^n - 1$ *extra bytes* so that you don't access memory outside of the allocated block. When freeing, you must pass the *original* address returned by `malloc` to the `free` function , *and not the aligned one.*

5 Bit-fields

This is a feature of C which seems quite convenient to use for interfacing to hardware. Their main disadvantage is that they cannot be reliably used to write portable code, or to access hardware.

EXAMPLE: PITFALLS OF BIT-FIELDS. In order to access individual fields within an x86 page table entry, one may be tempted to declare a structure similar to the following:

```
struct pte {
  unsigned pba:20;
  unsigned avl:3;
  /* etc... */
};
```

This code *might not work*, depending on the compiler. Namely, the C standard does not mandate how the bits within a bit-field are allocated. The `pba` field might get assigned to the *highest* 20 bits, or to the *lowest* 20 bits of an `unsigned int`. The latter case does not conform to the PTE format expected by the CPU. When a strict bit-layout and cross-platform compatibility is needed, it is recommended not to use this feature and to manually manipulate the bits within a word.

Simple data structures

The following sections present data structures that are needed in coding assignments. You are allowed (and we recommend you!) to use and adapt code presented here for your own purposes.

An aggravating circumstance is that dynamic memory allocation routines (`malloc()` and others) are not available. Therefore, all memory must be *allocated at statically, compile-time.* One consequence of static memory allocation is that data structures *cannot grow* beyond a fixed number of elements which is predetermined at compile-time.

6 Arrays

Arrays store their elements *consecutively* in memory. An array holding `N` elements of type `T` is declared as `T arr[N]`. Array indices start at 0 and extend up to and including `N-1`. Accessing an array outside of its bounds is an *unchecked error* and more often than not it leads to problems that are extremely difficult to debug.

6.1 ARRAYS AND POINTERS. The array name itself is a pointer to the first element of the array.[9] Pointers themselves can be indexed. In fact, the indexing operator is just syntactic sugar, and the expression `arr[i]` is *equivalent* to `*(arr+i)`. However, the code in function `f1` is *invalid* because the pointer `p` is not initialized to valid memory.

```
void f1(void)
```

[9] The array is said to *decay* into a pointer.

```
{
  unsigned int *p;
  p[3] = 0;
}
```

6.2 Automatic arrays. Care has to be taken when declaring arrays within a function without the `static` storage specifier, like in function `f2`.

```
void f2(void)
{
  unsigned int arr[512];

  /* some code */
}
```

Such declaration uses *stack space* that is *automatically* allocated on function entry and deallocated on function exit. In this example, it amounts to `512 * sizeof(int)` bytes, or 2kB given the usual size of 4 bytes for `int`. When the available stack space is very limited, it is easily overflown if large automatic arrays are used. There are no checks and in the case of overflow, some other data will be overwritten. Again, this leads to very hard to find and debug problems. *Exercise: design an efficient way to detect stack overflows.*

7 Ring buffer

This is a data structure that supports storage and retrieval of bytes in FIFO manner. The total amount of data that can be stored is predetermined. Here is presented an implementation by circular buffer.

7.1 Types. The `ringbuf_t` structure includes basic fields needed to have a functional ring buffer. The ring buffer is *empty* when `rb->head == rb->tail`. Therefore, the ring buffer can hold at most `MAX_SIZE - 1` bytes.

```
struct ringbuf_t {
  unsigned int head, tail;
  unsigned char buffer[MAX_SIZE];
};
```

Elements are consumed from the head, and added to the tail of the buffer.

7.2 Storing/retrieving bytes. `rb_getchar` reads a single byte from the ring buffer `rb`. It returns -1 if the ring buffer is empty, otherwise an integer in range 0-255 is returned. `rb_putchar` stores a single byte `b` in the ring buffer `rb`. It returns -1 if the ring buffer is full, and 0 otherwise.

```
int rb_getchar(struct ringbuf_t *rb)
{
  if(rb->head == rb->tail)
    return -1;
  rb->head = (rb->head+1) % MAX_SIZE;
  return rb->buffer[rb->head];
}

int rb_putchar(
  struct ringbuf_t *rb, unsigned char b);
```

Exercise: Implement the `rb_putchar` function according to the given specification and prototype. Note that this is an "inverse" of `rb_getchar`, so use that function as a hint.

7.3 Larger objects. Larger objects can be stored and retrieved with the following functions:

```
int rb_write(
  struct ringbuf_t *rb, void *obj, size_t len);
int rb_read(
  struct ringbuf_t *rb, void *obj, size_t len);
```

where `obj` points to the object and `len` is the length of the buffer. The `rb_write` function tries to write `len` bytes in the ring buffer; it returns 0 on success and -1 if there is not enough space. The `rb_read` function tries to read *up to* `len` bytes from the ring buffer and returns the actual number of bytes read, which can be smaller than `len`. It should return -1 if the ring buffer is empty. *Exercise: implement these functions using `rb_getchar` and `rb_putchar`.*

8 Linked lists

There are many variants of linked lists. It is most convenient to use a circular, doubly-linked with a *dummy node*. The dummy node does-n't contain any useful data; its only purpose is to prevent the list from ever becoming empty. This greatly simplifies the code since it eliminates many special cases in insertion and removal code. The macros are given below.[10]

```
#define LINK_NEXT(node, newnode)        \
    do {                                \
        (newnode)->prev = node;         \
        (newnode)->next = (node)->next; \
        (node)->next->prev = newnode;   \
        (node)->next = newnode;         \
    } while(0)

#define LINK_PREV(node, newnode)        \
    do {                                \
        (newnode)->next = node;         \
        (newnode)->prev = (node)->prev; \
        (node)->prev->next = newnode;   \
        (node)->prev = newnode;         \
    } while(0)

#define LINK_REMOVE(node)                   \
    do {                                    \
        (node)->prev->next = (node)->next;  \
        (node)->next->prev = (node)->prev;  \
        (node)->next = (node)->prev = NULL; \
    } while(0)

#define QUE_IS_EMPTY(head) \
  ((head) == (head)->next)

#define QUE_INIT(head, dummy)               \
    do {                                    \
        head = dummy;                       \
        (head)->next = (head)->prev = head; \
    } while(0)
```

An advantage of using macros is that they are *untyped*: they can be used on *any* structure which defines `prev` and `next` fields as pointers.

8.1 Empty list. The list is *empty* when it contains only the dummy node. This situation is depicted in **Figure 2**, and justifies the implementation of `QUE_IS_EMPTY` and `QUE_INIT` macros.

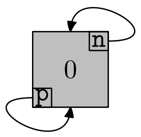

Figure 2 Empty list

Example: populating a list. The queue of tasks can be represented by a static array of task structures:

```
struct task {
  /* task data */
  struct task *next, *prev;
} tasks[16];
```

Note the addition of link fields in the structure. The following sequence of operations

```
/* initialize dummy node */
struct task *head;
QUE_INIT(head, &tasks[0]);

/* insert some nodes */
LINK_PREV(head, &task[1]);
LINK_PREV(head, &task[2]);
LINK_NEXT(head, &task[3]);
```

results with the list shown in **Figure 3**. *Exercise: The figure shows the final state of the list. Draw the whole list after each individual insertion.*

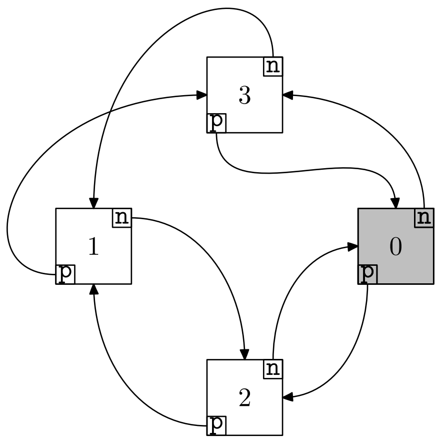

Figure 3 Populated list

8.2 Removal from a list. To remove a node, invoke the `LINK_REMOVE` macro on it. The node is not deallocated, it is just removed from the list. *Exercise: what happens when you remove the dummy node when the list is not empty? And when it is empty?*

[10] The `do while(0)` idiom enables macros to be used (almost) as functions.

8.3 Traversing a list. The dummy node is not used to store information; otherwise it wouldn't be possible to distinguish between an empty list and list with one data element. Therefore the traversal starts from `head->next`, and is done when the dummy node is encountered again. The loop should not be executed at all if the list is empty (i.e. contains only the dummy node). This code illustrates a possible way to accomplish the task:

```
struct task *t;
for(t = head->next; t != head; t = t->next) {
  /* ... */
}
```

Exercise: How to delete (in a safe way) the current element `t` *while traversing the list?*

9 Bit-vector

This is a data-structure in which individual bits can be addressed. It is usually implemented as an array of unsigned integers; for example this is an adequate definition of bit-vector holding up to $8 * 32 = 256$ bits:[11] `unsigned char bits[32];`

9.1 Addressing bits in an integer. The following macros can be used to test, clear and set individual bit `n` within an integer `w`. The argument `n` should be in the range from 0 to one less the number of bits used for `w` ($[0, 7]$ if `w` is `unsigned char`).[12] $n = 0$ operates on the least-significant bit.

```
#define BITMASK(n)        (1U << (n))
#define TESTBITw(w, n)    ((w) &   BITMASK(n))
#define CLEARBITw(w, n)  ((w) &= ~BITMASK(n))
#define SETBITw(w, n)     ((w) |=  BITMASK(n))
```

These macros *modify in place their first argument.* The key to understanding them is to notice that the `BITMASK(n)` macro evaluates to an unsigned integer having just the n-th bit set.

9.2 Addressing bits in a bit vector. The goal is to make macros `TESTBIT(v, n)`, etc., which work for the general case, where `v` is an array of integers, and `n` is a bit index within the bounds of an array. `n` is allowed to be larger than the number of bits in an integer. *Exercise: Using macros TESTBITw, etc., code the macros which work for the general case. Hint: you will need to use / and % operators.*

Assembler

The following section discuss topics related to the use of assembly language in the assignments.

10 Inline assembler

Mixing assembly with C code is **strongly discouraged**, since it makes it easy to make mistakes. Such code is also very hard to read.

Example. Function `clear_bit` is supposed to clear the `b`'th bit of `n` and return the result. Inline assembler implementation uses the `btr` instruction to accomplish the task:

```
unsigned clearbit(unsigned n, int b)
{
  unsigned r;

  asm("btrl %1, %0" : "+r" (n) : "r" (b));
  return r;
}
```

Exercise: The above function has deliberately been implemented incorrectly. Can you fix it? Compare it with the following pure C function; which one do you find easier to understand and see that it is correct?

```
unsigned clearbit(unsigned n, int b)
{
  return n & ~BITMASK(b);
}
```

[11] Here we quietly assume that `unsigned char` has 8 bits. This need not be the case; the actual size is given by the `CHAR_BITS` constant.

[12] `char` is also an integer type.

11 Memory operands.

Almost all x86 instructions accept memory operands. Exploiting these instructions can make the code much cleaner and easier to read, as illustrated in the following code snippets.

11.1 Read-modify-write instructions. The following problem is needed in the context switching code in one of the assignments. The task is to exchange the `%esp` register with a memory location named `stored_esp`. No other registers may be changed. The `%esp` is chosen on purpose so that the stack itself can't be used as a temporary storage.

The following code fragment uses only memory load and store instructions.

```
movl %eax, temp_eax
movl %esp, %eax
movl stored_esp, %esp
movl %eax, stored_esp
movl temp_eax, %eax
```

There are many *disadvantages* in this approach: (1) it uses an extra register and memory location, (2) it is non-atomic, and (3) it is non-reentrant. *Non-reentrancy* manifests itself in that fixed memory locations are used for temporary storage. *Exercise: consider what happens if the fragment is preempted and executed again from another thread, or when executed in parallel on an SMP system. Design a solution to this problem.*

The next fragment accomplishes the task in a straightforward way by using the `xchgl` instruction with a *memory operand.*

```
xchgl %esp, stored_esp
```

This approach has the following *advantages*: (1) it is easier to read and understand, (2) it is shorter and faster,[13] and (3) it is *atomic.*

11.2 A note on SMP systems. To guarantee atomicity of read-modify-write instructions on an SMP system, the `lock` prefix must be used; for example `lock; xchgl %esp, stored_esp`. Another consideration are `CLI/STI` instructions. They disable/enable interrupts *only* on the CPU which executes them – they have no effect on other CPUs. Thus, they *cannot* be used to implement *critical sections* on SMP systems.

11.3 Saving memory operands to the stack. The following is a possible solution to save and restore the contents of memory location `var_a` to the stack:

```
/* save var_a on the stack */
movl var_a, %eax
pushl %eax

/* restore var_a from the stack */
popl %eax
movl %eax, var_a
```

The simpler and recommended way is to do it directly:

```
pushl var_a /* save var_a */
popl var_a  /* restore var_a */
```

Example: pushing constants to the stack. *Always* write `pushl $1` instead of `movl $1, %eax ; pushl %eax` . The `$` is part of the AT&T syntax and is mandatory before an immediate constant.

12 Literature

When some issue is commented in the footnote, this is a signal to the reader that more extensive discussion can be found in the literature. The following resources cover issues mentioned here (and many others) in much more detail. Note that they are **not** obligatory reading for the course. They are an excellent reading if you want to gain an in-depth knowledge of system programming. When in doubt about some issue, it is most convenient to consult the C FAQ first.

[13] Read-modify-write instructions having memory operands are one of rare cases where clarity also yields better performance; at least on the x86 architecture.

1. C Frequently Asked Questions.
`http://www.c-faq.com`

2. Brian W. Kernighan and Dennis M. Ritchie: The C programming language. Prentice Hall, Inc., 1988. ISBN 0-13-110362-8 (paperback), 0-13-110370-9 (hardback).
`http://cm.bell-labs.com/cm/cs/cbook`

3. Peter van der Linden: Expert C Programming, Deep C Secrets. Pearson Education, 1994. ISBN 0131774298.
`http://www.taclug.org/booklist/development/C/Deep_C_Secrets.html`

4. John R. Levine: Linkers and Loaders. Morgan-Kauffman, 1999. ISBN 1-55860-496-0.
`http://www.iecc.com/linker`