

## Overview of the C Storage Model

### 1 Memory, Variables, Values, and Types

*Memory* consists of storage cells, each of which is capable of storing a single byte of information. Each cell is identified by a unique *address*. Values are stored in *variables*. A variable is a consecutive block of storage cells with an associated type. The number of cells is called the *size* of the variable. The type of a variable determines its size and the kind of values it can store.

Note that the memory consists only of the cells. The grouping of cells into variables and the type information concerning variables belong to the beholder; they are not stored in the memory. This means that if you look directly at memory, you will only see the storage cells and their contents. There will be no indication of how they are grouped into variables nor how they are supposed to be interpreted.

*Values* (also called *rvalues*) are data items that can be manipulated by operators in C. Values are classified according to *type*. The type of a value describes the size of the value and how it is to be interpreted by the C operators.

Values come in several flavors: numeric, compound, reference, and functions.

- *Numeric values* are values that represent numbers such as `int`, `char`, `long int`, `double`, and so forth.
- *Compound values* are collections of values formed using `struct`.
- *References values* are addresses of memory cells and have types such as `int*` or `char*`. The base type of a reference (the part of the type name before the `*`) describes the type and size of the variable to be found at the referenced memory location.
- *Functions* are code to be executed and have types such as `int()` (`double`, `int`). Function “values” are special in that they cannot be stored in variable. They can only be named, referenced, and called.

The types corresponding to the several flavors of values describe various properties of those values. The type of a numeric value describes its size and encoding. For example, values of both `float` and `int` (on the Zoo machines) have size 4, but the way they represent numbers is different. The type of a compound value describes the type and name of each component. The type of a reference value indicates both that it is a reference and describes the type of the variable which it references. The type of a function value describes the type of each argument to the function and the type of the value that the function returns.

### 2 Type Expressions and Declarations

A *type expression* is a syntactic construct to describe a type. Type expressions are used in various places in C program. The most common usage is to declare a variable as in the simple declaration `int x;`. This declaration actually does two distinct things:

1. It defines the program identifier `x` as being meaningful within its scope.
2. It allocates a variable on the stack of type `int` and names it `x`. To allocate a variable of type `int` means to reserve `sizeof(int)` consecutive bytes of storage for it.

The C programmer has three distinct storage areas available: static, stack, and heap. Stack variables are allocated automatically when a variable is declared. If the declaration is prefixed with the word *static*, the variable is allocated in the static storage area instead. The difference between static and stack storage is with the lifetime of the variable. The lifetime of static variables is the entire program. The lifetime of stack variables is the same as the lifetime of the block in which they are declared.

Variables are allocated in the heap using `malloc()` or `calloc()`. The lifetime of heap variables is controlled by the programmer. Once allocated, they persist over time until deallocated by an explicit call on `free()`.

Heap variables differ from static and stack in that they are always anonymous. The only way to access heap variables is through references. `malloc()` (and `calloc()`) returns a reference to a newly-allocated heap variable. `free()` deallocates a variable previously allocated by `malloc()`. If `free()` is passed a reference that was not created by `malloc()`, its behavior is undefined. (Read: this is a bug.)

Arrays are different from all other types in C. The biggest difference is that there are no array *values*. Since there are no array values, there are also no array *variables*. So what is there, you might ask? There are array types, but their use is highly restricted. This is best illustrated by some examples. The declaration `int array tab[5]` does the following:

1. It defines the program identifier `tab` as being meaningful within its scope.
2. It allocates 5 consecutive variables on the stack of type `int`.
3. It constructs a *reference value* of type `int*` to the first of those stack variables and gives it the name `tab`.

Comparing these actions to what happens with other types, we see two differences. First, the array declaration allocates several variables all at once. Second, the array identifier becomes the name of a reference value rather than the name of a variable. Because `tab` in the above example is a value of type `int*`, `tab` can be used anywhere that a value of type `int*` is expected, for example, as the actual argument to a function that takes a reference parameter, or on the right side of an assignment statement that stores into a variable of type `int*`. Because `tab` is *not* a variable, it cannot be used where a variable is expected, such as on the left side of an assignment statement.

There is one other difference between the identifiers defined by array declarations and other identifiers. For identifiers that name a variable, `sizeof` returns the number of bytes in the variable. For an array name, `sizeof` returns the number of bytes in the whole array, so `sizeof tab` returns 20 in the above example (assuming 4-byte `ints`), not the size of the reference value which `tab` names (which is 4 in my implementation).

### 3 Operations on Reference Values and Variables

A variable `x` can be interpreted in two distinct ways depending on the context in which it appears. When `x` is used on the left side of an assignment statement, it means that a value is to be stored in the variable named `x`. This is called the *lvalue* of `x`. When used on the right side of an assignment statement (or in another value-producing context, such as the argument to a function), it means that

the value stored in the associated variable is to be fetched and used. This is called the *rvalue* of  $x$ . The process of fetching a stored value from a variable is called *dereferencing*.

Literals such as the numeric constant `123` are always considered to be rvalues. That's why one can't write something like `123 = y`, since assignment needs an lvalue for its left operand and `123` has no lvalue.

### 3.1 Operators `*` and `&`

The unary operator `*` applied to a reference value returns the lvalue of the variable that the reference value references. Suppose  $e$  is a reference value of type  $T^*$ , that is,  $e$  is a reference to a variable of type  $T$ . Then  $*e$  is the variable referenced by  $e$ , and its type is  $T$ .  $*e$  can be used just like any other variable of type  $T$ . In particular, it has both an lvalue and an rvalue, so it can appear on either the left or right side of an assignment statement.

The unary operator `&` applied to a variable constructs a reference to that variable. Suppose  $v$  is a variable of type  $T$ . Then `& $v$`  is a reference value that refers to  $v$ , and its type is  $T^*$ .

Operators `*` and `&` are inverses of each other. If  $e$  and  $v$  are as above, then `& $*e$`  is the same reference value as  $e$ , and  `$*\&v$`  is the same variable as  $v$ .

Here's a concrete example.

```
1  int* p;
2  int x = 3;
3  p = &x;
4  *p = 4;
5  printf("%d\n", x);
```

Line 1 declares a reference variable `p` (also called a *pointer*). Line 2 declares an integer variable `x` and initializes it to 3. The right side of line 3 constructs a reference to `x` which is then stored in `p` by the assignment operator `=`. The left side of line 4 turns the reference value stored in `p` into an `int` variable (which is the same as `x`) and then stores 4 into it. Because `*p` and `x` are different names for the *same* variable, the value of `x` is now also 4, as can be demonstrated by observing the output of line 5.

### 3.2 Reference-valued expressions

There are four methods for constructing reference values.

1. The `&` operator applied to an existing variable returns a reference to that variable. In the above example, `&x` is a reference to `x`.
2. `malloc()` and `calloc()` allocate storage in the heap and return a reference to newly-allocated storage. For example, `malloc(10*sizeof(int))` allocates storage for 10 consecutive integer variables on the heap and returns a reference to the first such variable.
3. An array declaration allocates storage on the stack, creates a reference to it, and gives a name to the newly-created reference. For example, `int tab[10];` allocates storage for 10 consecutive integer variables on the stack and creates a reference named `tab` to the first such variable. The name `tab` evaluates to that reference value in all contexts except for the argument to `sizeof` (see above). If `p` is as described above, then `p=tab` causes the reference value `tab` to be stored in the pointer variable `p`, and `*tab` can be used as a name for the first variable in the array.

4. A new reference can be constructed from an existing reference using reference arithmetic. Let  $e$  be a reference value of type  $T^*$  that refers to memory cell with address  $a$ . Let  $k$  is an integer value (of any integral type). Then  $e+n$  is a new reference value of type  $T^*$  that refers to memory cell with address  $a + k * \text{sizeof}(T)$ . If one regards the memory beginning with cell  $a$  as a sequence of variables of type  $T$ , each occupying  $\text{sizeof}(T)$  cells, then  $e+n$  is a reference to variable number  $k$  in this sequence, where the first variable is numbered 0, the next is numbered 1, and so forth.

Reference arithmetic is the mechanism that allows the elements of an array to be accessed by the program. Here's an example of code that creates an array of three `int` variables and uses pointer arithmetic to set them to 2, 4, and 6, respectively.

```
1  int tab[3];
2  *tab = 2;
3  *(tab+1) = 4;
4  *(tab+2) = 6;
```

Line 1 allocates three `int` variables on the stack, creates a reference to the first of them, and names that reference `tab`. Hence, `*tab` is a name for that first integer variable and can be used with the assignment operator in line 2 to store 2 into that variable. `tab+1` is a reference to the second variable, so line 3 stores 4 there. Similarly, line 4 stores 6 into the third variable.

### 3.3 Subscripts

C provides the subscript notation as a shorthand for referring to array elements. If  $e$  is a reference value and  $k$  an integer, then  $e[k]$  is completely equivalent to  $*(e+k)$ . Thus, the above example could be equivalently written using subscripts as:

```
1  int tab[3];
2  tab[0] = 2;
3  tab[1] = 4;
4  tab[2] = 6;
```

Note that `tab[0]` is equivalent to `*(tab+0)` which in turn is equivalent to `*tab`, since `tab+0` is the same reference value as `tab`.

Note that `&` can be combined with the subscript notation to obtain references to arbitrary array elements. In the above example, `&tab[2]` is a reference to the third variable in the array. To see that this is correct, we note that `tab[2]` is equivalent to `*(tab+2)`, so `&tab[2]` is equivalent to `&*(tab+2)`. But as noted above, `&` is the inverse of `*`, so `&*(tab+2)` is equivalent to `tab+2`, a pointer to the third variable in the `tab` array.

## 4 Dangling References

No discussion of the C storage model would be complete without a discussion of dangling references. We say that a reference of type  $T$  is *valid* if it references a place in storage in which a variable of type  $T$  is currently allocated. Otherwise, it is said to be *dangling*. We think of a dangling reference as one that points off into the blue somewhere.

It is always an error to attempt to access the storage at the address referred to by a dangling reference. If one tries, one of two things will happen. If the storage at that address has not been assigned by the operating system to the running program, the attempted access will be caught as

invalid by the operating system, the program will be terminated, and a *segmentation fault* will be reported to the user. If the referenced storage is assigned to the running program, then the access will succeed just as if the reference were valid and a value will be read from or written to the memory byte(s) at the referenced address. Since that memory may be in use by other parts of the program, this unexpected sharing of memory will likely cause the program to fail, often in mysterious ways. These kinds of bugs tend to be the most difficult to find because the effects of the error may only become apparent long after the invalid access was made and will often be noticed first in an unrelated part of the code.

Here are some common ways that dangling references occur, so these are things to watch out for in your own programming.

**Uninitialized pointer variables** When one first declares a pointer variable, its contents is undefined. Of course, there is something in the memory occupied by the pointer variable; that something is a dangling reference. The common error is to forget to allocate storage and initialize the variable before use. For example, the following kind of incorrect code is all too common.

```
1 char buf[256];
2 char* mystring;
3 fgets( buf, 256, stdin );
4 strcpy( mystring, buf );
```

Here, line 3 is a safe way to read a string from standard input (unlike `gets()`, which should never be used). It reads at most 255 characters from `stdin` and places them in `buf`, followed by a NUL character. Line 4 is supposed to save the string just read in `mystring`. But no storage has been allocated for `mystring`, nor has a valid reference value been stored into the pointer `mystring`. Here is a correct way to accomplish this.

```
1 char buf[256];
2 char* mystring;
3 fgets( buf, 256, stdin );
4 mystring = (char*) malloc( strlen(buf)+1 );
5 strcpy( mystring, buf );
```

Line 4 allocates just enough storage to contain the string read in by line 3. Because `strlen()` does not count the terminating NUL character, 1 must be added to the value that it returns when calling `malloc()`.

**Buffer overrun** References to elements beyond the end of a valid array are dangling. For example, if `tab` is an array of length 10, then `tab[10]` refers to the first storage cell *beyond the end of* `tab`. In this example, `tab` and `tab+9` are both valid reference values, but `tab+10` is invalid.

**Explicit deallocation** Once `free()` is called to deallocate storage, any references to that storage that remain in the program become dangling. Unless the program is carefully structured, it may not be clear to the programmer whether any references remain to the newly freed storage. Here's an example.

```
1 char buf[256];
```

```

2 char* mystring;
3 fgets( buf, 256, stdin );
4 mystring = (char*) malloc( strlen(buf)+1 );
5 strcpy( mystring, buf );
6 puts( mystring );
7 free( mystring );

```

Lines 1–5 read a string into heap-allocated storage. Line 6 correctly prints the string to standard output. Line 7 deallocated the memory allocated in line 4 under the assumption that it is no longer needed. At this point, the reference to that storage remains in `mystring`, but it is no longer valid. Hence, `mystring` is now a dangling pointer.

**Automatic deallocation** References can also be made to stack-allocated variables. Because such variables are automatically deallocated when control exits from the block in which they were declared, the programmer may not be aware that any pointers to them have become invalid. Here's a typical example of this error.

```

1 char* append_doc( char* name )
2 {
3     char newname[100];
4     if (strlen( name ) > 95) return NULL;
5     strcpy( newname, name );
6     strcat( newname, ".doc" );
7     return newname;
8 }
9 ...
10 char* fname = append_doc( basename );
11 if (fname != NULL) fopen( fname, "r" );

```

This function carefully checks the length of its argument so as to avoid a buffer overrun, but the problem is that the reference it returns becomes dangling as soon as the function returns. As with other cases of dangling pointers, this may or may not work depending on whether the storage is immediately reused or not, but in any case it is incorrect and should be avoided.

One correct way to do this is to change line 3 to

```

3 char* newname = (char*) malloc( 100 );

```

But now the calling program is responsible for freeing the storage. If it forgets to, the program will likely have a memory leak.

Another correct way to do this is to make the caller responsible for providing the storage, e.g.,

```

1 char* append_doc( char* newname, int len, char* name )
2 {
3     if (strlen( name ) > len-5) return NULL;
4     strcpy( newname, name );
5     strcat( newname, ".doc" );
6     return newname;
7 }

```

```
8  ...
9  char newname[100];
10 char* fname = append_doc( newname, 100, basename );
11 if (fname != NULL) fopen( fname, "r" );
```

**NULL** The NULL reference itself is dangling, so one must never access \*NULL. However, it is perfectly legal to compare a pointer with NULL using == or !=. Also, NULL tests false and any non-NULL pointer tests as true when used as the condition in a Boolean context such as in the condition part of an if or while statement, so one can test if a pointer p is NULL by writing `if (!p) { ... }` instead of the more straightforward `if (p!=NULL) { ... }`.

Note that references are ordered, so one can also write `p < q`, for example, to test whether or not pointer p is smaller than pointer q. Sometimes it makes sense to use dangling pointers other than NULL in comparisons. Consider the following code to zero an array:

```
1  int tab[10];
2  int* top = &tab[10];
3  int* p;
4  for (p=tab; p<top; p++) *p=0;
```

Line 5 is a straightforward for loop except that instead of using integer subscripts to run through the array, it uses the pointer p. The loop terminates when p becomes equal to top. Even though &tab[10] is a dangling reference, it is guaranteed to exist, to be greater than &top[9], and to equal &top[9]+1. Such guarantees apply only to the reference that falls just beyond the end of the array. No guarantees are made about the reference &top[11] or how it compares with top, for example.