In which our hero learns of the most basic way to build a new data collection type in C, the ubiquitous struct, and learns a little bit about how they are placed in memory.

31.1 Funny or Interesting Introduction

We try to start each chapter with a funny or interesting introduction. Why? Well, to draw you, dear reader, into the material, in a light and enjoyable way, and perhaps spark some other interests in you as well. The best of these relate directly to the material and also stand on their own. The worst are just a few paragraphs of mildly interesting diversion.

Sometimes, though, we lack inspiration. And thus you get what we call here the generic funny and interesting introduction. Imagine this introduction was funny or interesting or (better yet!) both. Finished with that trip to Imagination Land? Good. And now we proceed into the material.

31.2 The struct: Definition

Thus far we’ve only seen basic data types, like integers, characters, and strings. However, many programs that you write will want to group many of these basic data types together into a larger related collection of information. Thus, most programming languages, including C, provide some kind of facility for doing so.

For example, let’s say you want to track some information about a group of people. Specifically, you want to be able to record a person’s name (first and last), the person’s age (in years), and the person’s height (in inches). To do so, you need new language support for grouping this information as well as accessing it. In C, the support you get is provided by the struct (short for structure).

Here is a simple example of a struct that creates a new type, called struct person_info, and lists its three member fields: age, height, and id.
struct person_info {
    int age;     // age in years
    int height;  // height in inches
    int id;      // unique id number
};

Now, each time you want to track information about a person, you can define a new variable to be of this type and use it throughout your program as needed. For example, you might define one of these new structures (on the stack, in the stack frame of `main()`) as follows:

```c
int main(int argc, char *argv[]) {
    struct person_info p;
    ...
}
```

### 31.3 Accessing Struct Fields

As we showed above, a `struct` has a number of constituent **fields**, each of which is some kind of variable in its own right. But how do we access these fields?

To read or write a field, C provides you with a new operator, known as the **dot operator** (\`). For example, if we want to initialize the fields of the `struct person_info` above, all we need to do is this:

```c
int main(int argc, char *argv[]) {
    struct person_info p;

    p.age = 45;
    p.height = 75;
    p.id = 1001;
    ...
}
```

With statically-declared structures (such as `struct person_info` above), you can also use some shortcuts when initializing, thanks to more recent versions of C. For example, you can initialize `p` in one line as follows:

```c
struct person_info p = { 45, 75, 1001 };  
```

Also, if you wish to make your code a little clearer, you can add the field names:

```c
struct person_info p = { .age=45, .height=75, .id=1001 };  
```
All are fine ways to set the initial values of a struct. The important thing to remember: always initialize variables (including fields of a struct) before using them.

### 31.4 Typing Less With typedef

The masters of early UNIX and C programming didn’t like typing a lot. One reason is likely that fancy editors and integrated development environments didn’t exist yet, so a programmer really had to type every long word, letter by excruciating letter.

As a result, shortcuts were pretty popular. One particular shortcut enabled by C is the typedef keyword, which allows you to provide different names for types. The syntax of typedef is simple:

```c
typedef existing_type new_type;
```

For example, you might be getting tired of typing (or perhaps, just reading) `struct person_info` over and over. Thus, use typedef and save those precious keystrokes!

```c
typedef struct person_info pinfo_t;
```

Here, we create a new name (`pinfo_t`) for this type, and can use it any place we previously would have used a `struct person_info`. For example, one example of initialization code above could be replaced with:

```c
pinfo_t p = { 45, 75, 1001 };
```

The _t at the end of the type name is just a common convention that programmers often use when creating a new type name. However, any legal C name would be acceptable.

Another common practice is to combine the typedef with the definition of the new struct itself. For example, the longer form of separate typedef and struct usage looks like this:

```c
typedef struct person_info pinfo_t;
struct person_info {
    int age;       // age in years
    int height;   // height in inches
    int id;       // unique id number
};
```

However, we could instead combine those two ever so slightly more concisely in this manner:
typedef struct person_info {
    int age;        // age in years
    int height;     // height in inches
    int id;         // unique id number
} pinfo_t;

Which way you decide depends on your own style, or the style your employer forces upon you. However, as is the case with all style issues, **pick one style and stick to it**. Not doing so makes code less consistent, and thus less readable.

### 31.5 Pointers To structs

As with any variable type, you can use pointers to point to struct types too, and manipulate them mostly as you would think. Here’s an example that creates a struct on the stack, and then refers to it indirectly via a pointer:

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    struct person_info p;

    p.age = 45;
    p.height = 75;
    p.id = 1001;

    struct person_info* q;
    q = &p;
    (*q).age = 21;

    // now prints '21': you're younger!
    printf("age: %d\n", p.age);
}
```

The assignment statement works exactly as before. The code `q=&p;` assigns the address of `p` to `q`, but then we have the hard part: how to use the . operator to access the structure’s fields?

The dot operator only works on struct types, and unfortunately, `q` is not a struct but rather a pointer to one. Thus, to use the dot operator to access fields of this struct, you must first dereference `q`, and only then can you use the dot operator to access the age field. But this leads to one more problem: what precedence is the dot operator . as compared to the dereference operator *?

As you can see in Figure 31.1, the dot operator has a very high level of precedence, even higher than dereferencing. Thus, if you
simply write the following, without parentheses, the dot operator takes precedence:

\[ \texttt{+q.age = 21;} \]

The result is that the compiler will complain, perhaps like this:

```
main.c:12 error: member reference type 'struct
person_info *' is a pointer
```

This means that the compiler knows \( q \) is a pointer, and that you can’t apply the dot operator to it. Thus, you must force a different grouping, with parentheses:

\[ \texttt{(*q).age = 21;} \]

Here again the masters of C decided to do something about this. Here the reason is mostly aesthetics: \( (*q)\).age is simply an ugly way to refer to the age field of the struct \( q \) points to. To make this look prettier, C provides another operator, which looks like a little rightward-pointing arrow (\( \rightarrow \)). Wherever you used to write \( (*q)\).age you can now instead write \( q\rightarrow\text{age} \). Thus, we can rewrite the sequence that defines a new pointer \( q \) and sets the age field into one short two-statement sequence, completely equivalent to the previous version, only a little nicer to look at:

```
struct person_info* q = &p;
q->age = 21;
```

And now you know why when someone writes \( q\rightarrow\text{age} \) in their code, we also say that they are dereferencing the pointer \( q \).

A depiction of this code in action is shown in Figure 31.2 (page 6). We now highlight a few major points. First, you can see that a struct, when placed in memory, is nothing special; in this case, there is just room for the three integers that comprise the struct. That’s it! When you write \( p\).age in your code, you are just referring to the first of those integers, \( p\).height the second, and \( p\).id the third.
Figure 31.2: Structure Layout And Pointer Access

Second, you can see that \( q \) itself does not take up much room in memory. It is just a pointer! (here, we assume it is a 32-bit pointer, or 4 bytes). While it can refer to a large structure, \( q \) itself doesn’t take up much space at all.

Third, you can see that assigning a pointer \( q \) to the address of \( p \) just sets \( q \) to the address of the first element of the structure. In the example in the figure, \( q \) gets set to address 1016. While we say this is the address of the structure, you should note it is also the address of the first element (age).

Finally, you can see how a pointer dereference operates with a pointer to a structure. By using the arrow operator, we dereference \( q \), and thus, indirectly, update one of the fields in the structure it points to by writing \( q->\text{age}=21 \) in the code.

31.6 Structures And Copying

One nice things about struct types is that it is easy to make a copy of one via simple assignment. For example, here we just fill in one copy of a structure, makes a copy of it, and then updates the height field:

```c
struct person_info p = { 45, 75, 1001 };
struct person_info c = p;
c.height = 80; // your twin is taller!
```

The variable \( c \) now holds the same values in it, as shown in Figure 31.3 (page 7). Because \( p \) and \( c \) are distinct copies, changing one (e.g., setting the height field of \( c \) to 80, as in Line 3 above) does not affect the other copy.
31.7 Passing Structures To Functions

C, as you recall, has pass-by-value semantics, meaning that when a parameter is passed to a function, a copy of that value is placed on the stack and this copy is what the function operates upon when running. These semantics have strong implications for us when passing a struct to a function.

Specifically, when you pass a struct to a function, you are implicitly passing a copy of that entire struct. Pass-by-value thus has two strong implications. First, it can be quite inefficient: if the struct is large, every element of it is copied onto the stack. Second, it may not be what you are looking for, especially if you wish to change any of the values within that struct. To achieve this end, you must instead pass a pointer to a struct.

Let’s look at an example. Here is a routine that tries to change the fields of a struct person_info, but not very effectively:

```c
// this code is pointless and broken and makes us sad
void change_fields(struct person_info arg, int new_age,
                   int new_height, int new_id) {
    arg.age = new_age;
    arg.height = new_height;
    arg.id = new_id;
}
```

When change_fields is called, C places a copy of each argument onto the stack, including a copy of the entire structure, and thus when p is modified within the function, it only changes that copy, leaving the original unmodified. Perhaps we should have called the function doesnt_change_fields()?
Figure 31.4 shows the call to `change_fields(p, 21, 80, 1)`, focusing on how a copy of all the arguments (including the structure itself) are put on the stack. As you can see in the time sequence, when the (broken) code updates the structure `arg`, it is simply modifying the copy on the stack.

The more proper way to write `change_fields()` is shown below. Here, you can see small but critical difference: a pointer to the structure is passed to the routine, and thus each update to `arg` is indirectly updating the caller’s structure, as desired.

```c
// this code works now!
void change_fields(struct person_info* arg, int new_age, int new_height, int new_id) {
    arg->age = new_age;
    arg->height = new_height;
    arg->id = new_id;
}
```

Figure 31.5 (page 9) shows the difference on the stack when we call the modified `change_fields()` function (in the example shown here, with arguments `&p, 21, 80, 1`). Here you can see the immediate difference; because `arg` is now a pointer, a copy of the address is placed on the stack. Then, within the function, that indirect reference is used to reach down into the original `struct` and change its values.
You can also see how much more efficient it is to pass a pointer to a structure than a copy of the structure itself – just one address instead of a copy of every field of the (potentially quite large) structure. Thus, you should almost always pass a pointer to a structure rather than the structure itself.

Of course, this could be error prone: you might accidentally write code to change the value of the structure now without really meaning to. To avoid this problem, you can use the keyword `const` in the parameter list of the function. For example, you can use `const` as follows:

```c
void print_fields(const struct person_info* arg) {
    printf("age: %d\n", arg->age);
    printf("height: %d\n", arg->height);
    printf("id: %d\n", arg->id);
}
```

If, for some reason within this function, you try to update a field of the pointed-to structure (e.g., if you include the code `arg->age=0`), the compiler will generate an error, thus preventing a successful compilation:

```c
error: cannot assign to variable 'arg' with
       const-qualified type 'const struct person_info *'
```

As you can see, the relatively readable error tells you that you are doing something wrong; you’ll either need to remove the `const` keyword from the parameter list, or change the function itself not to update the structure.
For efficiency reasons, (almost) always pass a pointer to a `struct` to a function, not the `struct` itself. If the function does not modify the `struct`, use the `const` keyword to enforce this at compile time.

In functions that take values only as inputs (and thus, they, or what they point to, should not be modified), use the keyword `const` to tell the compiler of this fact. Once written, the compiler will help enforce this rule, thus reducing the chances you will make silly mistakes. However, don’t overuse this, rather sticking to cases where you are passing pointers to objects that should not be modified by the function in question.

**31.8 Nested Structures**

C structures are quite flexible, allowing arbitrary nesting of one structure inside of another (and another, etc.) to form large and complicated user-defined data types. For example, let’s say we want to build a larger structure out of the previously-defined `struct person_info` about a student, which adds a little bit of auxiliary information:

```c
#define MAX_CLASSES (4)
#define MAX_NAME (40)

struct student_info {
    // some basic info about the student
    struct person_info basics;

    // classes: array of classes student is taking
    // num_classes: count of valid classes in array
    // 0... num_classes-1 are valid
    int classes[MAX_CLASSES];
    int num_classes;

    // housing unit
    char housing_unit[MAX_NAME];
};
```

To access the fields of this structure, we use the same dot operator as before. For example, here is some code that sets the values of each field; you can see, in particular, that the dot operator can be repeatedly applied in order to access fields within fields (and, if the structure had more sub-structures, fields within those, ad infinitum):
struct student_info s;
s.basics.age = 10;  // young student
s.basics.height = 50;  // also short!
s.basics.id = 10001;
s.classes[0] = 240;  // discrete math class
s.classes[1] = 354;  // intro to systems
s.num_classes = 2;  // only two; slacker
sprintf(s.housing_unit, "Union South");

We can even use a static initializer, which follows the nested structure of the struct itself, if we like:

```
struct student_info s = { { 10, 50, 10001 },
                        { 240, 354, -1, -1 }, 2,
                        "Union South" };  
```

31.9 Arrays Of Structures

Structures are also incredibly useful when combined with arrays. For example, let’s define an array of person_info structures, one per person that we’re interested in:

```
struct person_info people[10];
```

We can then access any particular record by first using the bracket operators to specify which element of the array, and then the dot operator to specify which field:

```
people[3].age = 10;
```

We could also write a simple initialization routine as follows:

```
void clear_fields(struct person_info a[], int num) {
    int i;
    for (i = 0; i < num; i++) {
        a[i].age = 0;
        a[i].height = 0;
        a[i].id = 0;
    }
}
```

This function works because, as we learned earlier, passing an array to a function is really just passing the starting address, and thus accessing each element is just like dereferencing that pointer. For this reason, we could keep the same function body and write the function parameter definition like this:
void clear_fields(struct person_info* a, int num) {
    ...
}

In either form, the function works as expected, clearing each field of each structure within the array.

31.10 Structure Packing

The last point we’ll touch on is an important one, and relates to how C places structures in memory. Most of this discussion is based on Raymond’s excellent treatise found on the Internet [R14]; here we just present a shortened version of the same.

The first piece of knowledge you need is that each scalar variable type (such as int, double, short) as an alignment requirement. This means that when C allocates space for a variable in memory, it cannot start at any particular address but rather is more constrained.

For example, a typical 4-byte C int has a 4-byte alignment requirement, which means that any C integer must start on an address that is evenly divisible by 4 (e.g., address%4 equals zero). The requirements for other types are similar: a 2-byte short must be 2-byte aligned, a 4-byte type (such as an int or float) must be 4-byte aligned, and an 8-byte type (such as a long long or double) must be 8-byte aligned. A char, being 1 byte, has no such requirement and can be placed anywhere in memory.

You can actually see the effects of these requirements without using structures at all. For example, imagine the following function:

```c
void show_alignment() {
    int x;
    int y;
    char z;
    printf("x:%p
", &x);
    printf("y:%p
", &y);
    printf("z:%p
", &z);
}
```

When we run this function (assuming we compiled for 32 bits, with the -m32 flag to gcc), we see the following three addresses:

x:0xbffed8fc
y:0xbffed8f8
z:0xbffed8f7

From this output, you can observe that the compiler ensured that both x and y were aligned to 4-byte boundaries (the last digit of each’s address being c and 8, respectively, which are both divisible evenly by 4), while z in this case was not (the last digit being 7).
The alignment requirement has strong implications for structure layout, both within a single `struct` as well as in an array of structures. Specifically, C will guarantee that if you define an array of structures, the layout of the array will guarantee that even if you pack together a number of structures into an array, these alignment requirements will be met.

Why is this challenging? Well, imagine a `struct` called `record` composed of the two integers (4 bytes) and one character (1 byte) (total size needed: 9 bytes):

```c
struct record {
    int a1;
    int a2;
    char c;
};
```

If we repeat this structure in memory (without too much thought), we’ll quickly see that the alignment requirements will not be met. Figure 31.6 shows the problem when two of these structures are placed back to back in memory; while the first structure aligns as needed, the second does not (e.g., the integer `a` in `p[1]` is placed at address 1009, which is clearly not evenly divisible by 4 (1009%4=1).

To solve this problem, C will insert `padding` into structures to make them meet alignment requirements, even when they are used in arrays as above. Padding is just wasted space inside each structure, but is useful in creating the desired alignments. Figure 31.7
Figure 31.7: Alignment With Padding In A struct

shows how C will align struct record to meet the alignment requirements listed above.

31.11 Alignment: Why It’s There, How To Avoid Waste

A few questions naturally fall out of this discussion. The first is this: why does C enforce alignment? The second: how can we avoid wasting space with lots of padding?

The reason for alignment requirements has to do with performance [D07]. Although memory is logically byte addressable, processors often fetch data from memory in 4-byte, 8-byte, or even larger sized chunks. Without alignment, the result is that a basic data type like an integer could span these natural fetching boundaries.

For example, imagine if a processor naturally fetched data from memory in 32-byte sized chunks. Imagine further that a 4-byte integer was not aligned properly and spanned these two chunks, instead of fitting neatly within one of them, as in Figure 31.8.

As a result, every time you read the integer, the processor has to fetch two large chunks, wasting memory bandwidth. Every time you write the integer, it’s even worse: the processor has to read both chunks in, modify the integer (spanning the end of one chunk and the beginning of the next), and then write the two chunks out, wast-
ing even more memory bandwidth. By aligning basic data types, C generally avoids bandwidth waste and generally improves performance\(^1\).

Finally, we answer the second question: how can you, as a programmer, avoid wasting space in structures unnecessarily? As Raymond calls it, this is the “lost art of structure packing” [R14], and is actually quite simple to understand.

The most important thing to know is that C absolutely lays out member fields of the structure in the order you specify in the struct. Specifically, imagine you define a structure as follows:

```c
struct record {
    char c1; // yes, these are creative names
    int i1;
    char c2;
    int i2;
    char c3;
    int i3;
    char c4;
    int i4; // especially this one
};
```

C guarantees that it will place such a structure in memory in exactly the order your structure definition specifies, i.e., the address of c1 will be lower than the address of i1 will be lower than the address of c2, etc. However, C also fills in padding to meet alignment requirements. In the case above, 3 bytes of padding will be placed after each char, resulting in a total size for the record of 32 bytes. You can use the handy operator `sizeof()` to check this is true:

```c
printf("size of 'struct record': %lu\n",
    sizeof(struct record));
```

\(^1\)Note that on some platforms, alignment is even required for correctness (e.g., Sparc and MIPS CPUs don’t allow arbitrary byte address accesses; rather, loads and stores from memories must be 4-byte aligned). However, x86 and other architectures allow all possible bytes to be accessed, just performing worse when alignment constraints are not followed.
So now you have a challenge: how can you rearrange the member fields of the structure above to minimize space usage? Think about it, and see if you can come up with an answer. (pause here)

Here is one way you could do better. By placing all the integers first, followed by the characters, `struct record_packed` guarantees that no padding at all is needed, and only 20 bytes are needed for each record, a 37.5% savings compared to the sloppy original. Nice!

```c
struct record_packed {
    int i1;
    int i2;
    int i3;
    int i4;
    char c1;
    char c2;
    char c3;
    char c4;
};
```

You can also peer into each structure to see the byte offsets of each field with the newer C macro `offsetof()` (please do make sure to `#include <stddef.h>` to use it). Here’s an example:

```c
printf("offset of i1: %lu\n",
    offsetof(struct record, i1));
```

One last point: if you don’t want to try to change the order of fields in a structure but for some reason really want to save space, you can direct the C compiler to pack the structure in as tightly as possible, hence ignoring the performance implications of doing so. With `gcc`, this feat is accomplished by adding the non-standardized `__attribute__((packed))` after the last brace of the structure definition but before the semi-colon. We’ll leave it to the reader to learn more about this if interested.

### 31.12 Common Mistakes

One common mistake made by programmers is not to realize that in order to use the arrow operator on a pointer to a structure, the pointer must indeed first point to something. For example, you might see code like this:

```c
struct foo* ptr;
ptr->field = 10;
```

When you see this code, you should immediately know that it is buggy (and indeed, the outcome is `undefined`, although it will likely...
crash). Try it and see! And then learn the lesson: **do not dereference an uninitialized pointer via the arrow operator.** Of course, you shouldn’t ever dereference any sort of uninitialized pointer, but somehow the arrow operator seems to hide the dereference from those new to C.

Another mistake, as discussed earlier, is to pass a copy of a structure to a function, instead of a pointer to the structure. Doing so is both inefficient (especially for large structures) and prevents changing the fields of the structure from within the function. Thus: **do not pass structures to functions; rather, pass pointers to structures.**

One last mistake, as discussed in the last part of this chapter, is not understanding how structures are laid out in memory. Without care, a programmer can waste a great deal of memory. In many ways, this is the worst kind of error, as it is silent – the program “works” but less efficiently. Thus, **be cognizant of structure packing and alignment rules** or be doomed to waste memory needlessly.

### 31.13 Summary

Structures are one of the most important aspects of C programs. Indeed, some programmers say that the first thing they do to understand a large piece of complex code is study the data structures, in order to understand how data is stored; then, when studying functions, it is easier to grok what the intent of the code is. In this chapter, we’ve introduced C structures and the `struct` keyword, and shown some of its basic usages. As you move forward with more complex programs of your own, you will spend a great deal more time defining such structures and thus taking the time to carefully digest this material will have been well worth it.
References

[R14] “The Lost Art of C Structure Packing”
Eric S. Raymond
January, 2014
http://www.catb.org/esr/structure-packing
An excellent overview of how C packs data into structures, with many detailed examples.

[D07] “What Every Programmer Should Know About Memory”
Ulrich Drepper
November 21, 2007
We referenced this earlier and will again later. It has a little bit about alignment and a lot about a lot of other topics.