5 structs, unions, and bitfields

Roll your own structures

Most things in life are more complex than a simple number.

So far, you've looked at the basic data types of the C language, but what if you want to go beyond numbers and pieces of text, and **model things in the real world**? **`structs`** allow you to model **real-world complexities** by writing your own structures. In this chapter, you'll learn how to **combine the basic data types** into `structs`, and even **handle life's uncertainties** with **`unions`**. And if you're after a simple yes or no, ***bitfields*** may be just what you need.

Sometimes you need to hand around a lot of data

You've seen that C can handle a lot of different types of data: small numbers and large numbers, floating-point numbers, characters, and text. But quite often, when you are recording data about something in the real world, you'll find that you need to use more than one piece of data. Take a look at this example. Here you have two functions that *both* need the same set of data, because they are both dealing with the same real-world *thing*:

"const char *" just means you're going to pass string literals.

Both of these functions take the same set of parameters.

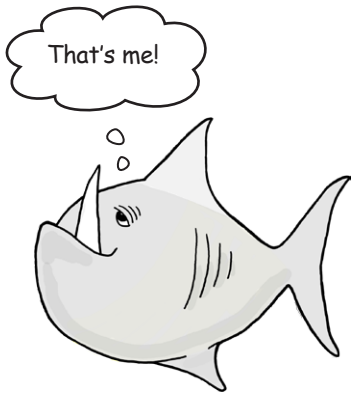
```
/* Print out the catalog entry */
void catalog(const char *name, const char *species, int teeth, int age)
{
  printf("%s is a %s with %i teeth. He is %i\n",
    name, species, teeth, age);
}

/* Print the label for the tank */
void label(const char *name, const char *species, int teeth, int age)
{
  printf("Name:%s\nSpecies:%s\n%i years old, %i teeth\n",
    name, species, teeth, age);
}
```

Now that's not really so bad, is it? But even though you're just passing four pieces of data, the code's starting to look a little messy:

You are passing the same four pieces of data twice.

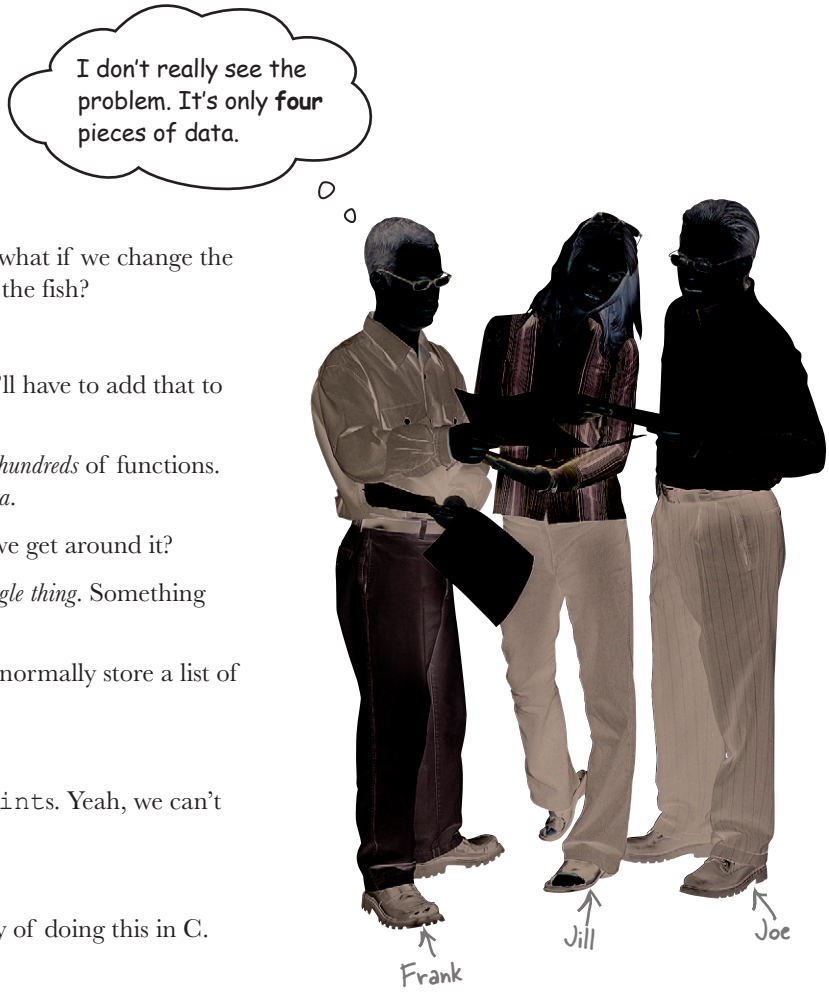
```
int main()
{
  catalog("Snappy", "Piranha", 69, 4);
  label("Snappy", "Piranha", 69, 4);
  return 0;
}
```

There's only one fish, but you're passing four pieces of data.



So how do you get around this problem? What can you do to avoid passing around lots and lots of data if you're really only using it to describe a single thing?

Cubicle conversation



Joe: Sure, it's four pieces of data *now*, but what if we change the system to record another piece of data for the fish?

Frank: That's only *one more parameter*.

Jill: Yes, it's just one piece of data, but we'll have to add that to *every function* that needs data about a fish.

Joe: Yeah, for a big system, that might be *hundreds* of functions. And all because we add *one more piece of data*.

Frank: That's a good point. But how do we get around it?

Joe: Easy, we just group the data into a *single thing*. Something like an array.

Jill: I'm not sure that would work. Arrays normally store a list of data of the *same type*.

Joe: Good point.

Frank: I see. We're recording strings and `int`s. Yeah, we can't put those into the same array.

Jill: I don't think we can.

Joe: But come on, there must be some way of doing this in C. Let's think about what we need.

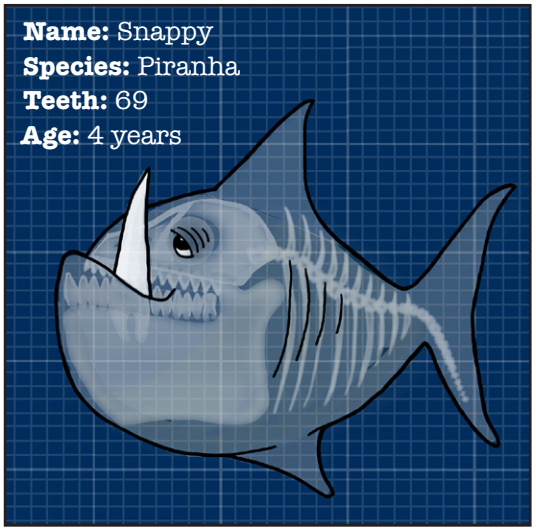
Frank: OK, we want something that lets us refer to a whole set of data of different types all at once, as if it were a single piece of data.

Jill: I don't think we've seen anything like that yet, have we?

What you need is something that will let you record several pieces of data into *one large piece of data*.

Create your own structured data types with a struct

If you have a set of data that you need to bundle together into a *single thing*, then you can use a **`struct`**. The word `struct` is short for **structured data type**. A `struct` will let you take all of those different pieces of data into the code and wrap them up into one large new data type, like this:

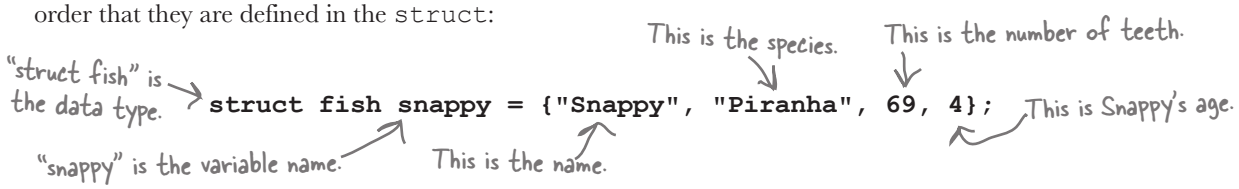



```
struct fish {
  const char *name;
  const char *species;
  int teeth;
  int age;
};
```

This will create a new custom data type that is made up of a collection of other pieces of data. In fact, it's a little bit like an array, except:

- **It's fixed length.**
- **The pieces of data inside the struct are given names.**

But once you've defined what your new `struct` looks like, how do you create pieces of data that use it? Well, it's quite similar to creating a new array. You just need to make sure the individual pieces of data are in the order that they are defined in the `struct`:



there are no Dumb Questions

Q: Hey, wait a minute. What's that `const char` thing again?

A: `const char *` is used for strings that you don't want to change. That means it's often used to record string literals.

Q: OK. So does this `struct` store the string?

A: In this case, no. The `struct` here just stores a pointer to a string. That means it's just recording an address, and the string lives somewhere else in memory.

Q: But you can store the whole string in there if you want?

A: Yes, if you define a `char` array in the string, like `char name[20];`.

Just give them the fish

Now, instead of having to pass around a whole collection of individual pieces of data to the functions, you can just pass your new custom piece of data:

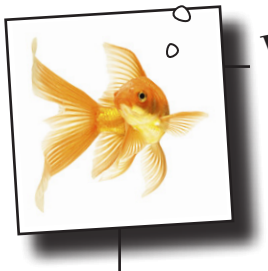
```
/* Print out the catalog entry */
void catalog(struct fish f)
{
  ...
}

/* Print the label for the tank */
void label(struct fish f)
{
  ...
}
```

Looks a lot simpler, doesn't it? Not only does it mean the functions now only need a *single piece of data*, but the code that calls them is easier to read:

```
struct fish snappy = {"Snappy", "Piranha", 69, 4};
catalog(snappy);
label(snappy);
```

So that's how you can define your custom data type, but how do you *use* it? How will our functions be able to read the individual pieces of data stored inside the `struct`?

Wrapping parameters in a struct makes your code more stable.



Why the fish is good for you

One of the great things about data passing around inside `struct`s is that you can change the contents of your `struct` without having to change the functions that use it. For example, let's say you want to add an extra field to `fish`:

```
struct fish {
    const char *name;
    const char *species;
    int teeth;
    int age;
    int favorite_music;
};
```

All the `catalog()` and `label()` functions have been told is they they're going to be handed a `fish`. They don't know (and don't care) that the `fish` now contains more data, so long as it has all the fields they need.

That means that `struct`s don't just make your code easier to read, they also make it better able to cope with change.

Read a struct's fields with the "." operator

Because a `struct`'s a little like an array, you might think you can read its fields like an array:

```
struct fish snappy = {"Snappy", "piranha", 69, 4};
printf("Name = %s\n", snappy[0]);
```

If snappy was a pointer to an array, you would access the first field like this.

You get an error if you try to read a struct field like it's an array.

```
File Edit Window Help Fish
> gcc fish.c -o fish
fish.c: In function 'main':
fish.c:12: error: subscripted value is neither array nor pointer
>
```

But you can't. Even though a `struct` stores fields like an array, the only way to access them is ***by name***. You can do this using the "." operator. If you've used another language, like JavaScript or Ruby, this will look familiar:

```
struct fish snappy = {"Snappy", "piranha", 69, 4};
printf("Name = %s\n", snappy.name);
```

This is the name attribute in snappy.

This will return the string "Snappy."

```
File Edit Window Help Fish
> gcc fish.c -o fish
> ./fish
Name = Snappy
>
```

OK, now that you know a few things about using structs, let's see if you can go back and update that code...

Piranha ~~Pool~~ Puzzle

Your job is to write a new version of the `catalog()` function using the `fish struct`. Take fragments of code from the pool and place them in the blank lines below. You may not use the same fragment more than once, and you won't need to use all the fragments.

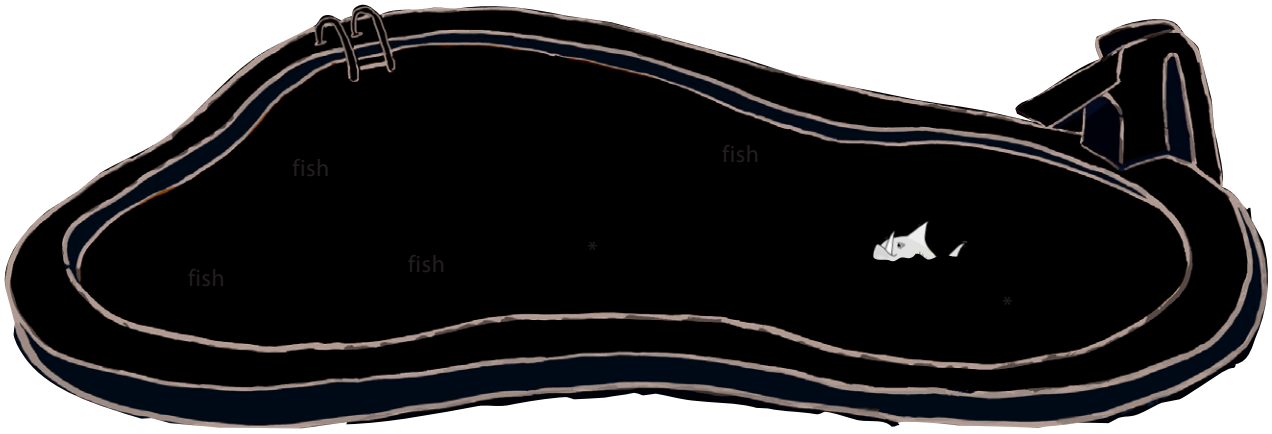
```
void catalog(struct fish f)
{
  printf("%s is a %s with %i teeth. He is %i\n",
    ......., ......., ......., ....... );
}

int main()
{
  struct fish snappy = {"Snappy", "Piranha", 69, 4};
  catalog(snappy);
  /* We're skipping calling label for now */
  return 0;
}
```

Note: each thing from the pool can be used only once!

Piranha ~~Pool~~ Puzzle Solution

Your job was to write a new version of the `catalog()` function using the `fish struct`. You were to take fragments of code from the pool and place them in the blank lines below.

```
void catalog(struct fish f)
{
  printf("%s is a %s with %i teeth. He is %i\n",
     f.name, f.species, f.teeth, f.age );
}

int main()
{
  struct fish snappy = {"Snappy", "Piranha", 69, 4};
  catalog(snappy);
  /* We're skipping calling label for now */
  return 0;
}
```



Test Drive

You've rewritten the `catalog()` function, so it's pretty easy to rewrite the `label()` function as well. Once you've done that, you can compile the program and check that it still works:

Hey, look, someone's using make...

This line is printed out by the catalog() function.

These lines are printed by the label() function.

```
File  Edit  Window  Help  FishAreFriendsNotFood
> make pool_puzzle && ./pool_puzzle
gcc pool_puzzle.c -o pool_puzzle
Snappy is a Piranha with 69 teeth. He is 4
Name:Snappy
Species:Piranha
4 years old, 69 teeth
>
```

That's great. The code works the same as it did before, but now you have really simple lines of code that call the two functions:

```
catalog(snappy);

label(snappy);
```

Not only is the code more readable, but if you ever decide to record some extra data in the `struct`, you won't have to change anything in the functions that use it.

there are no Dumb Questions

Q: So is a `struct` just an array?

A: No, but *like* an array, it groups a number of pieces of data together.

Q: An array variable is just a pointer to the array. Is a `struct` variable a pointer to a `struct`?

A: No, a `struct` variable is a name for the `struct` itself.

Q: I know I don't have to, but could I use `[0]`, `[1]`,... to access the fields of a `struct`?

A: No, you can only access fields by name.

Q: Are `structs` like classes in other languages?

A: They're similar, but it's not so easy to add methods to `structs`.

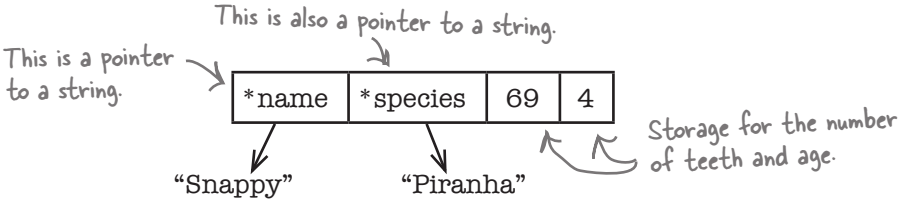
Structs In Memory Up Close

When you define a `struct`, you're not telling the computer to create anything in memory. You're just giving it a **template** for how you want a new type of data to look.

```
struct fish {
  const char *name;
  const char *species;
  int teeth;
  int age;
};
```

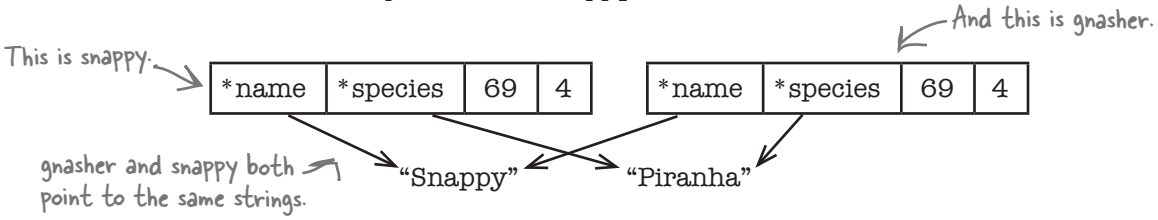
But when you define a new variable, the computer will need to create some space in memory for an **instance** of the `struct`. That space in memory will need to be big enough to contain all of the fields within the `struct`:

```
struct fish snappy = {"Snappy", "Piranha", 69, 4};
```



So what do you think happens when you assign a `struct` to another variable? Well, the computer will create a **brand-new copy of the struct**. That means it will need to allocate another piece of memory of the same size, and then copy over each of the fields.

```
struct fish snappy = {"Snappy", "Piranha", 69, 4};
struct fish gnasher = snappy;
```



Remember: when you're assigning struct variables, you are telling the computer to *copy* data.

Watch it!

The assignment copies the pointers to strings, not the strings themselves.

*When you assign one `struct` to another, the contents of the `struct` will be copied. But if, as here, that includes **pointers**, the assignment will just copy the pointer values. That means the `name` and `species` fields of `gnasher` and `snappy` both point to the same strings.*

Can you put one struct inside another?

Remember that when you define a `struct`, you're actually creating a *new data type*. C gives us lots of built-in data types like `int`s and `short`s, but a `struct` lets us combine existing types together so that you can describe *more complex objects* to the computer.

But if a `struct` creates a data type from existing data types, that means you can also **create `struct`s from other `struct`s**. To see how this works, let's look at an example.

```
struct preferences {
  const char *food;
  float exercise_hours;
};

struct fish {
  const char *name;
  const char *species;
  int teeth;
  int age;
  struct preferences care;
};
```

These are things our fish likes.

This is a struct inside a struct.

This is a new field.

This is called nesting.

Our new field is called "care," but it will contain fields defined by the "preferences" struct.

Why nest structs?

Why would you want to do this? So you can cope with **complexity**. `struct`s give us bigger *building blocks* of data. By combining `struct`s together, you can create larger and larger data structures. You might have to begin with just `int`s and `short`s, but with `struct`s, you can describe hugely complex things, like **network streams** or **video images**.

This code tells the computer one `struct` will contain another `struct`. You can then create variables using the same array-like code as before, but now you can include the data for one `struct` *inside another*:

```
struct fish snappy = {"Snappy", "Piranha", 69, 4, {"Meat", 7.5}};
```

This is the struct data for the care field.

This is the value for care.food.

This is the value for care.exercise_hours.

Once you've combined `struct`s together, you can access the fields using a *chain* of "." operators:

```
printf("Snappy likes to eat %s", snappy.care.food);
printf("Snappy likes to exercise for %f hours", snappy.care.exercise_hours);
```

OK, let's try out your new struct skillz...

Long Exercise

The guys at the Head First Aquarium are starting to record lots of data about each of their fish guests. Here are their `struct`s:

```
struct exercise {
  const char *description;
  float duration;
};

struct meal {
  const char *ingredients;
  float weight;
};

struct preferences {
  struct meal food;
  struct exercise exercise;
};

struct fish {
  const char *name;
  const char *species;
  int teeth;
  int age;
  struct preferences care;
};
```

This is the data that will be recorded for one of the fish:

```
Name: Snappy
Species: Piranha
Food ingredients: meat
Food weight: 0.2 lbs
Exercise description: swim in the jacuzzi
Exercise duration 7.5 hours
```

Question 0: How would you write this data in C?

```
struct fish snappy = ..........;
```

Question 1: Complete the code of the `label()` function so it produces output like this:

```
Name:Snappy
Species:Piranha
4 years old, 69 teeth
Feed with 0.20 lbs of meat and allow to swim in the jacuzzi for 7.50 hours
```

```
void label(struct fish a)
{
  printf("Name:%s\nSpecies:%s\n%i years old, %i teeth\n",
         a.name, a.species, a.teeth, a.age);
  printf("Feed with %2.2f lbs of %s and allow to %s for %2.2f hours\n",
         ..........., ...........,
         ..........., ........... );
}
```

The guys at the Head First Aquarium are starting to record lots of data about each of their fish guests. Here are their `struct`s:

```
struct exercise {
  const char *description;
  float duration;
};

struct meal {
  const char *ingredients;
  float weight;
};

struct preferences {
  struct meal food;
  struct exercise exercise;
};

struct fish {
  const char *name;
  const char *species;
  int teeth;
  int age;
  struct preferences care;
};
```

This is the data that will be recorded for one of the fish:

```
Name: Snappy
Species: Piranha
Food ingredients: meat
Food weight: 0.2 lbs
Exercise description: swim in the jacuzzi
Exercise duration 7.5 hours
```

Question 0: How would you write this data in C?

```
struct fish snappy = {"Snappy", "Piranha", 69, 4, {{"meat", 0.2}, {"swim in the jacuzzi", 7.5}}};
```

Question 1: Complete the code of the `label()` function so it produces output like this:

```
Name:Snappy
Species:Piranha
4 years old, 69 teeth
Feed with 0.20 lbs of meat and allow to swim in the jacuzzi for 7.50 hours
```

```
void label(struct fish a)
{
  printf("Name:%s\nSpecies:%s\n%i years old, %i teeth\n",
         a.name, a.species, a.teeth, a.age);
  printf("Feed with %2.2f lbs of %s and allow to %s for %2.2f hours\n",
         a.care.food.weight , a.care.food.ingredients ,
         a.care.exercise.description , a.care.exercise.duration );
}
```

Hmmm...all these struct commands seem kind of wordy. I have to use the struct keyword when I define a struct, and then I have to use it again when I define a variable. I wonder if there's some way of simplifying this.

You can give your struct a proper name using typedef.
When you create variables for built-in data types, you can use simple short names like `int` or `double`, but so far, every time you've created a variable containing a `struct` you've had to include the `struct` keyword.

```
struct cell_phone {
  int cell_no;
  const char *wallpaper;
  float minutes_of_charge;
};
...
struct cell_phone p = {5557879, "sinatra.png", 1.35};
```

But C allows you to create an **alias** for any `struct` that you create. If you add the word **`typedef`** *before* the `struct` keyword, and a **type name** *after* the closing brace, you can call the new type whatever you like:

typedef means you are going to give the struct type a new name.

```
typedef struct cell_phone {
  int cell_no;
  const char *wallpaper;
  float minutes_of_charge;
} phone;
```

phone will become an alias for "struct cell_phone."

```
...
phone p = {5557879, "sinatra.png", 1.35};
```

Now, when the compiler sees "phone," it will treat it like "struct cell_phone."

`typedef`s can shorten your code and make it easier to read. Let's see what your code will look like if you start to add `typedef`s to it…

What should I call my new type?

If you use `typedef` to create an alias for a `struct`, you will need to decide what your ***alias*** will be. The alias is just the name of your type. That means there are *two names* to think about: the name of the `struct` (`struct` **`cell_phone`**) and the name of the ***type*** (**`phone`**). Why have two names? You usually don't need both. The compiler is quite happy for you to skip the `struct` name, like this:

```
typedef struct {
  int cell_no;
  const char *wallpaper;
  float minutes_of_charge;
} phone;
phone p = {5557879, "s.png", 1.35};
```

This is the alias.

It's time for the scuba diver to make his daily round of the tanks, and he needs a new label on his suit. Trouble is, it looks like some of the code has gone missing. Can you work out what the missing words are?

```
#include <stdio.h>

  ..........struct {
   float tank_capacity;
   int tank_psi;
   const char *suit_material;
} ..........;

  ..........struct scuba {
   const char *name;
   equipment kit;
} diver;

void badge(.......... d)
{
  printf("Name: %s Tank: %2.2f(%i) Suit: %s\n",
    d.name, d.kit.tank_capacity, d.kit.tank_psi, d.kit.suit_material);
}

int main()
{
  .......... randy = {"Randy", {5.5, 3500, "Neoprene"}};
  badge(randy);
  return 0;
}
```

Exercise Solution

It's time for the scuba diver to make his daily round of the tanks, and he needs a new label on his suit. Trouble is, it looks like some of the code has gone missing. Could you work out what the missing words were?

```
#include <stdio.h>

  typedef ..........struct {
    float tank_capacity;
    int tank_psi;
    const char *suit_material;
} equipment ..........;

  typedef ..........struct scuba {
    const char *name;
    equipment kit;
} diver;

void badge( diver ..... d)
{
  printf("Name: %s Tank: %2.2f(%i) Suit: %s\n",
    d.name, d.kit.tank_capacity, d.kit.tank_psi, d.kit.suit_material);
}

int main()
{
  diver ..... randy = {"Randy", {5.5, 3500, "Neoprene"}};
  badge(randy);
  return 0;
}
```

The coder decided to give the struct the name "scuba" here. But you'll just use the diver type name.

BULLET POINTS

- A `struct` is a data type made from a sequence of other data types.
- `struct`s are fixed length.
- `struct` *fields* are accessed by name, using the <*`struct`*>.<*`field name`*> syntax (aka *dot notation*).
- `struct` fields are stored in memory in the same order they appear in the code.
- You can nest `struct`s.
- `typedef` creates an *alias* for a data type.
- If you use `typedef` with a `struct`, then you can skip giving the `struct` a name.

there are no Dumb Questions

Q: Do `struct` fields get placed next to each other in memory?

A: Sometimes there are small gaps between the fields.

Q: Why's that?

A: The computer likes data to fit inside word boundaries. So if a computer uses 32-bit words, it won't want a `short`, say, to be split over a 32-bit boundary.

Q: So it would leave a gap and start the `short` in the next 32-bit word?

A: Yes.

Q: Does that mean each field takes up a whole word?

A: No. The computer leaves gaps only to prevent fields from splitting across word boundaries. If it can fit several fields into a single word, it will.

Q: Why does the computer care so much about word boundaries?

A: It will read complete words from the memory. If a field was split across more than one word, the CPU would have to read several locations and somehow stitch the value together.

Q: And that'd be slow?

A: That'd be slow.

Q: In languages like Java, if I assign an object to a variable, it doesn't copy the object, it just copies a reference. Why is it different in C?

A: In C, *all* assignments copy data. If you want to copy a reference to a piece of data, you should assign a pointer.

Q: I'm really confused about `struct` names. What's the `struct` name and what's the alias?

A: The `struct` name is the word that follows the `struct` keyword. If you write `struct peter_parker { ... }`, then the name is `peter_parker`, and when you create variables, you would say `struct peter_parker x`.

Q: And the alias?

A: Sometimes you don't want to keep using the `struct` keyword when you declare variables, so `typedef` allows you to create a single word alias. In `typedef struct peter_parker { ... } spider_man;`, `spider_man` is the alias.

Q: So what's an anonymous `struct`?

A: One without a name. So `typedef struct { ... } spider_man;` has an alias of `spider_man`, but no name. Most of the time, if you create an alias, you don't need a name.

How do you update a struct?

A `struct` is really just a bundle of variables, grouped together and treated like a single piece of data. You've already seen how to create a `struct` object, and how to access its values using dot notation. But how do you *change* the value of a `struct` that already exists? Well, you can change the fields just like any other variable:

This creates a struct.

```
fish snappy = {"Snappy", "piranha", 69, 4};
printf("Hello %s\n", snappy.name);
snappy.teeth = 68;
```

This reads the value of the name field.

This sets the value of the teeth field.

Ouch! Looks like Snappy bit something hard.

That means if you look at this piece of code, you should be able to work out what it does, right?

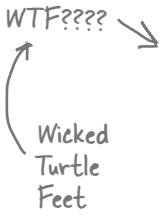
```
#include <stdio.h>

typedef struct {
  const char *name;
  const char *species;
  int age;
} turtle;

void happy_birthday(turtle t)
{
  t.age = t.age + 1;
  printf("Happy Birthday %s! You are now %i years old!\n",
    t.name, t.age);
}

int main()
{
  turtle myrtle = {"Myrtle", "Leatherback sea turtle", 99};
  happy_birthday(myrtle);
  printf("%s's age is now %i\n", myrtle.name, myrtle.age);
  return 0;
}
```

Myrtle the turtle

But there's something odd about this code...

Test Drive

This is what happens when you compile and run the code.




```
File Edit Window Help ILikeTurtles
> gcc turtle.c -o turtle && ./turtle
Happy Birthday Myrtle! You are now 100 years old!
Myrtle's age is now 99
>
```

Something weird has happened.

The code creates a new `struct` and then passes it to a function that was *supposed* to increase the value of one of the fields by 1. And *that's exactly what the code did*…at least, for a while.

Inside the `happy_birthday()` function, the `age` field was updated, and you know that it worked because the `printf()` function displayed the new increased `age` value. But that's when the weird thing happened. Even though the `age` was updated by the function, when the code returned to the `main()` function, the `age` seemed to reset itself.

This code is doing something weird. But you've already been given enough information to tell you exactly ***what*** happened. Can you work out what it is?

The code is cloning the turtle

Let's take a closer look at the code that called the `happy_birthday()` function:

```
void happy_birthday(turtle t)
{
   ...
}

...
happy_birthday(myrtle);
```

This is the turtle that we are passing to the function.

The myrtle struct will be copied to this parameter.

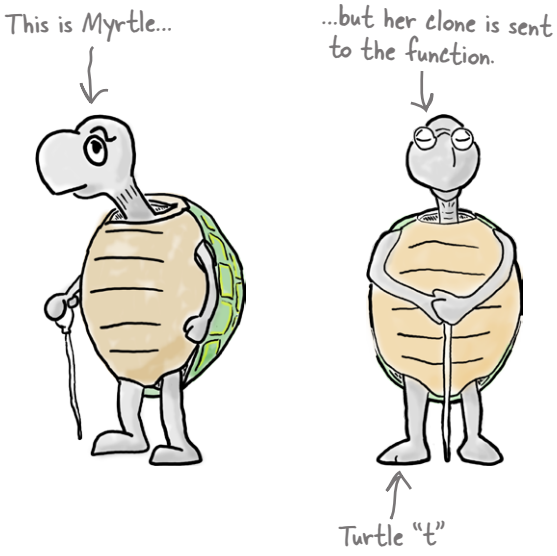
When you assign a struct, its values get copied to the new struct.

In C, parameters are passed to functions **by value**. That means that when you call a function, the values you pass into it are *assigned* to the parameters. So in this code, it's almost as if you had written something like this:

```
turtle t = myrtle;
```

But *remember*: when you assign `structs` in C, the values are copied. When you call the function, the parameter `t` will contain a *copy* of the `myrtle struct`. It's as if the function ***has a clone of the original turtle***. So the code inside the function *does* update the age of the turtle, **but it's a different turtle**.

What happens when the function returns? The `t` parameter disappears, and the rest of the code in `main()` uses the `myrtle struct`. But the value of `myrtle` was never changed by the code. It was always a completely separate piece of data.



So what do you do if you want pass a struct to a function that needs to update it?

You need a pointer to the struct

When you passed a variable to the `scanf()` function, you couldn't pass the variable itself to `scanf()`; you had to pass a **pointer**:

```
scanf("%f", &length_of_run);
```

Why did you do that? Because if you tell the `scanf()` function where the variable lives in memory, then the function will be able to update the data stored at that place in memory, which means it can update the variable.

And you can do just the same with `structs`. If you want a function to update a `struct` variable, you can't just pass the `struct` as a parameter because that will simply send a *copy* of the data to the function. Instead, you can pass the address of the `struct`:

```
void happy_birthday(turtle *t)
{
   ...
}

...
happy_birthday(&myrtle);
```

This means "Someone is going to give me a pointer to a struct."

Remember: an address is a pointer.

This means you will pass the address of the myrtle variable to the function.

See if you can figure out what *expression* needs to fit into each of the gaps in this new version of the `happy_birthday()` function.

Be careful. Don't forget that `t` is now a **pointer variable**.

```
void happy_birthday(turtle *t)
{
   ..........age = ..........age + 1;
  printf("Happy Birthday %s! You are now %i years old!\n",
         ..........name, ..........age);
}
```

You were to figure out what *expression* needs to fit into each of the gaps in this new version of the `happy_birthday()` function.

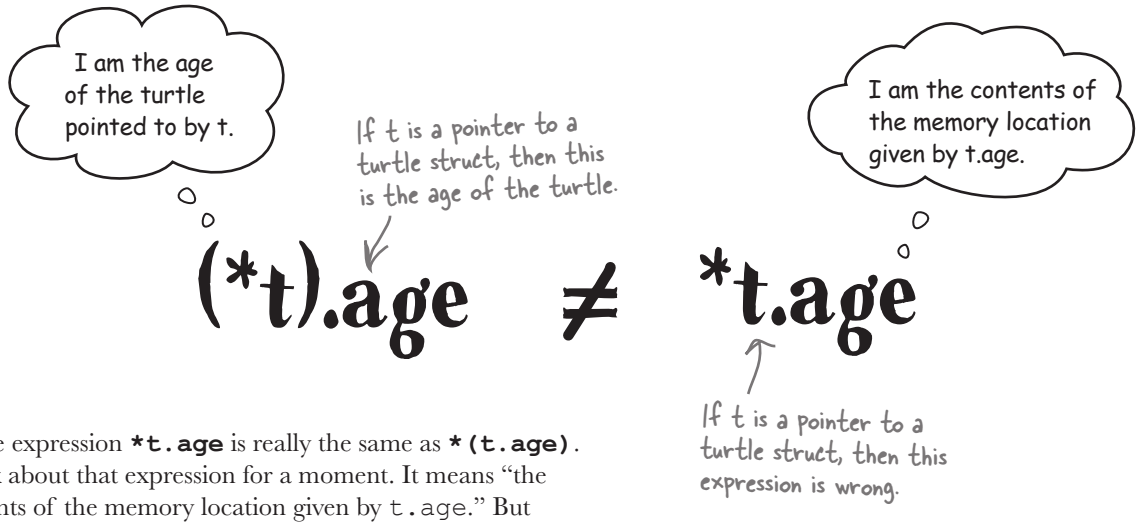
```
void happy_birthday(turtle *t)
{
    (*t).age = (*t).age + 1;
    printf("Happy Birthday %s! You are now %i years old!\n",
           (*t).name, (*t).age);
}
```

You need to put a * before the variable name, because you want the value it points to.

The parentheses are really important. The code will break without them.

(*t).age vs. *t.age

So why did you need to make sure that `*t` was wrapped in parentheses? It's because the two expressions, `(*t).age` and `*t.age`, are very different.



So the expression **`*t.age`** is really the same as **`*(t.age)`**. Think about that expression for a moment. It means "the contents of the memory location given by `t.age`." But `t.age` isn't a memory location.

So be careful with your parentheses when using structs—parentheses really matter.

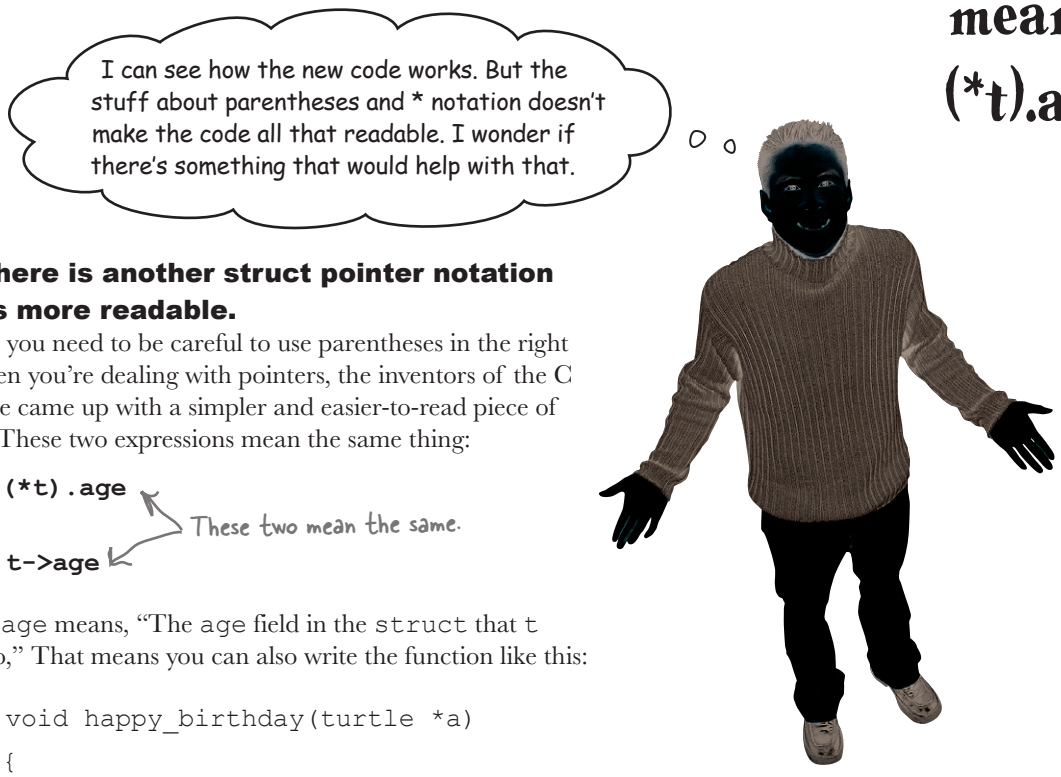
Test Drive

Let's check if you got around the bug:

```
File Edit Window Help ILikeTurtles
> gcc happy_birthday_turtle_works.c -o happy_birthday_turtle_works
Happy Birthday Myrtle! You are now 100 years old!
Myrtle's age is now 100
>
```

That's great. The function now works.
By passing a pointer to the `struct`, you allowed the function to update the *original data* rather than taking a local copy.

t->age means (*t).age



Yes, there is another struct pointer notation that is more readable.
Because you need to be careful to use parentheses in the right way when you're dealing with pointers, the inventors of the C language came up with a simpler and easier-to-read piece of syntax. These two expressions mean the same thing:

```
(*t).age
t->age
```

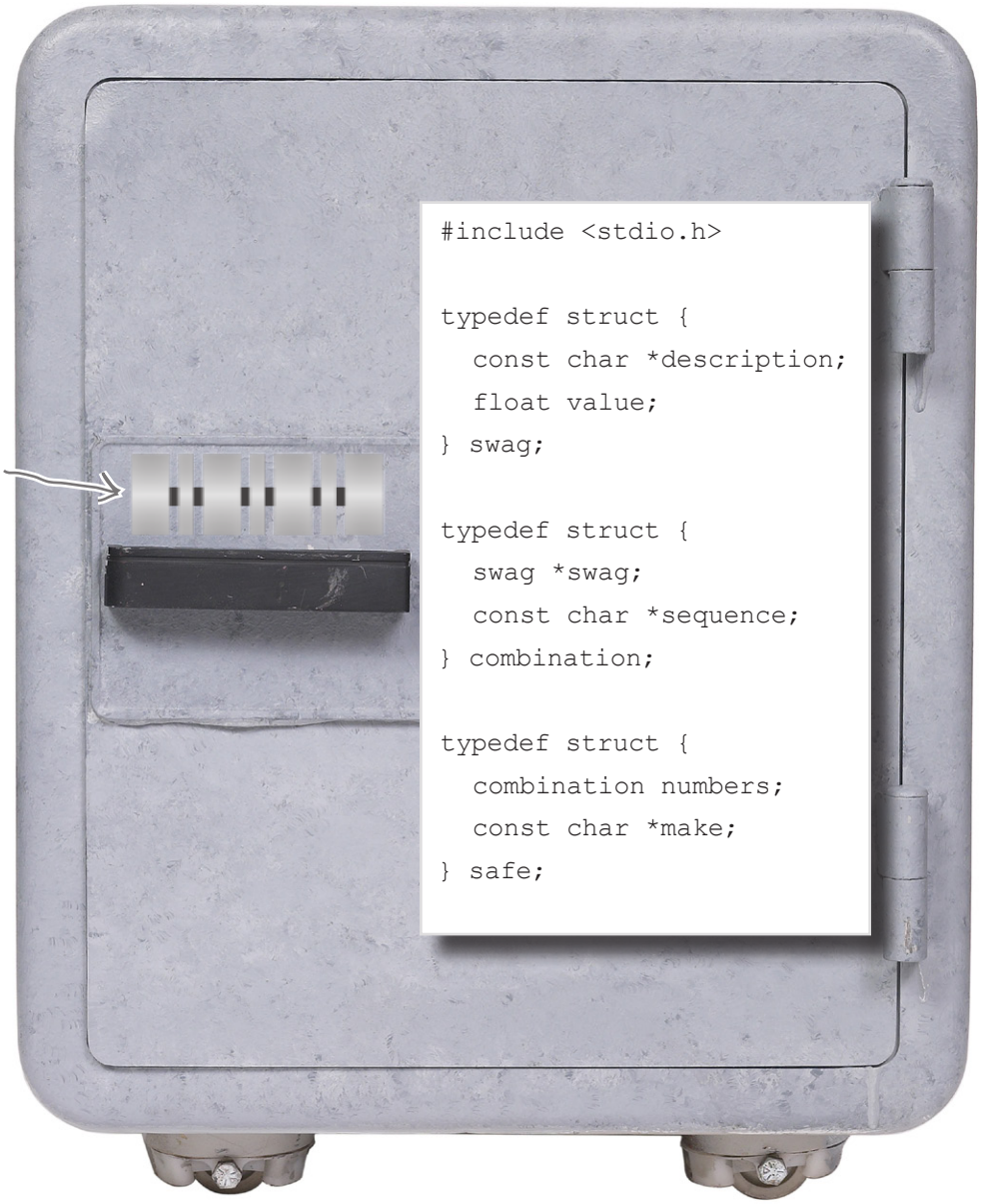
These two mean the same.

So, `t->age` means, "The `age` field in the `struct` that `t` points to," That means you can also write the function like this:

```
void happy_birthday(turtle *a)
{
  a->age = a->age + 1;
  printf("Happy Birthday %s! You are now %i years old!\n",
         a->name, a->age);
}
```

Safe Cracker

Shhh…it's late at night in the bank vault. Can you spin the correct combination to crack the safe? Study these pieces of code and then see if you can find the correct combination that will allow you to get to the gold. Be careful! There's a `swag` type *and* a `swag` field.



The bank created its safe like this:

```
swag gold = {"GOLD!", 1000000.0};
combination numbers = {&gold, "6502"};
safe s = {numbers, "RAMACON250"};
```

What combination will get you to the string "GOLD!"? Select one symbol or word from each column to assemble the expression.

con	.	s	+	swag	.	value
s	->	numbers	.	description	-	swag
numbers	:	swag	->	value	->	description
swap	-	gold	-	sequence	+	gold

there are no Dumb Questions

Q: Why are values copied to parameter variables?

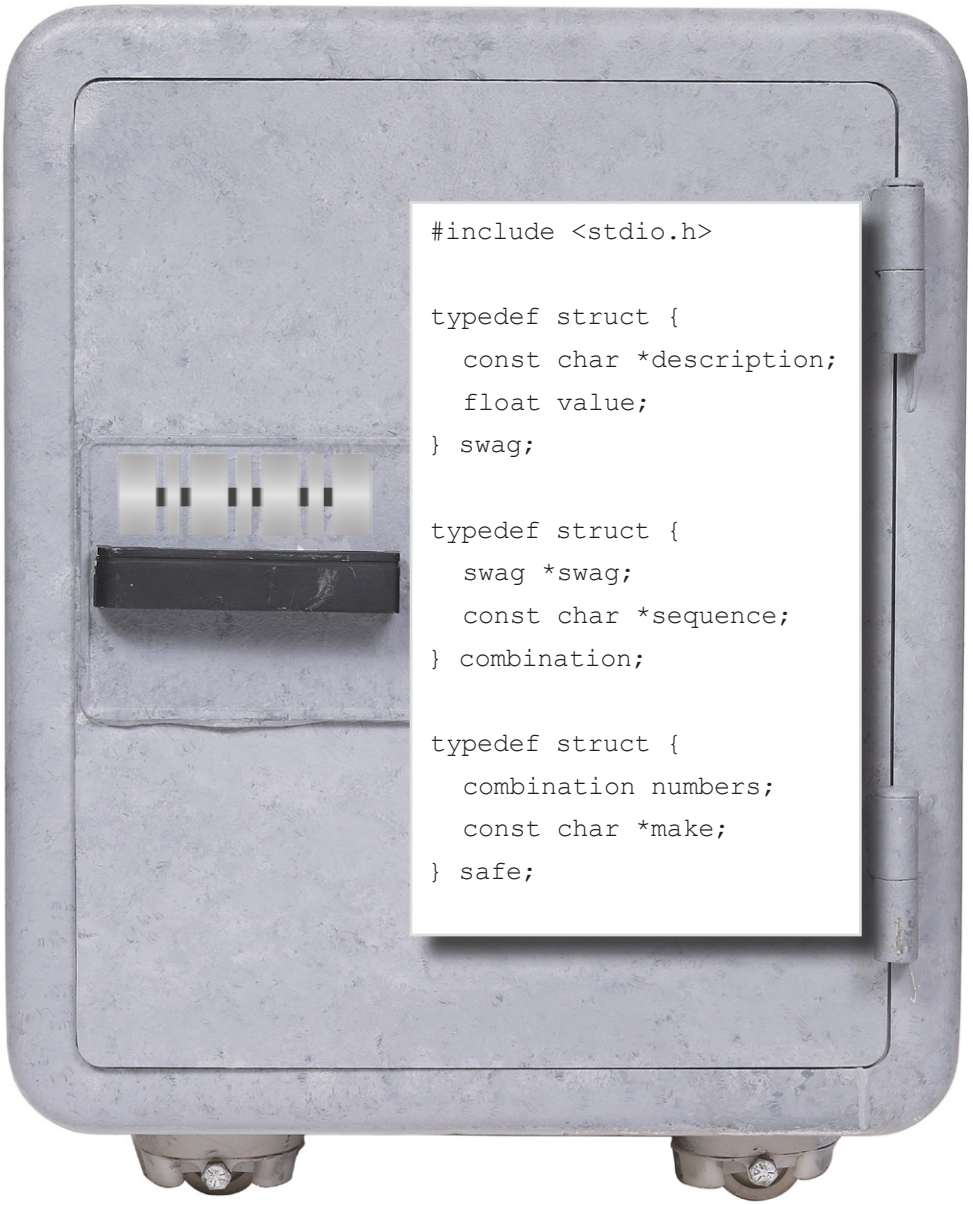
A: The computer will pass values to a function by assigning values to the function's parameters. And all assignments copy values.

Q: Why isn't `*t.age` just read as `(*t).age`?

A: Because the computer evaluates the dot operator before it evaluates the *.

Safe Cracker Solution

Shhh…it's late at night in the bank vault. You were to spin the correct combination to crack the safe. You needed to study these pieces of code and then find the correct combination that would allow you to get to the gold.

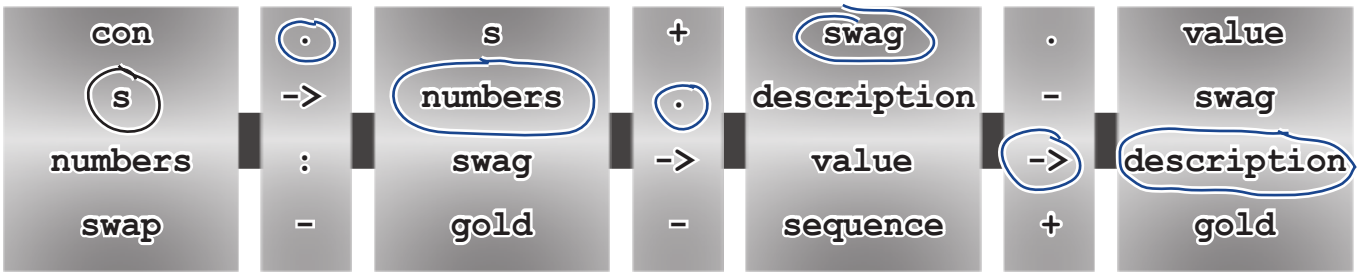


```
#include <stdio.h>

typedef struct {
  const char *description;
  float value;
} swag;

typedef struct {
  swag *swag;
  const char *sequence;
} combination;

typedef struct {
  combination numbers;
  const char *make;
} safe;
```

The bank created its safe like this:

```
swag gold = {"GOLD!", 1000000.0};
combination numbers = {&gold, "6502"};
safe s = {numbers, "RAMACON250"};
```

What combination will get you to the string "GOLD!"? You were to select one symbol or word from each column to assemble the expression.



So you can display the gold in the safe with:

```
printf("Contents = %s\n", s.numbers.swag->description);
```

BULLET POINTS

- When you call a function, the values are ***copied*** to the parameter variables.
- You can create pointers to `structs`, just like any other type.
- `pointer->field` is the same as `(*pointer).field`.
- The `->` notation cuts down on parentheses and makes the code more readable.

Sometimes the same type of thing needs different types of data

`structs` enable you to model more complex things from the real world. But there are pieces of data that don't have a single data type:

So if you want to record, say, a *quantity* of something, and that quantity might be a **count**, a **weight**, or a **volume**, how would you do that? Well, you *could* create several fields with a `struct`, like this:

```
typedef struct {
  ...
  short count;
  float weight;
  float volume;
  ...
} fruit;
```

But there are a few reasons why this is not a good idea:

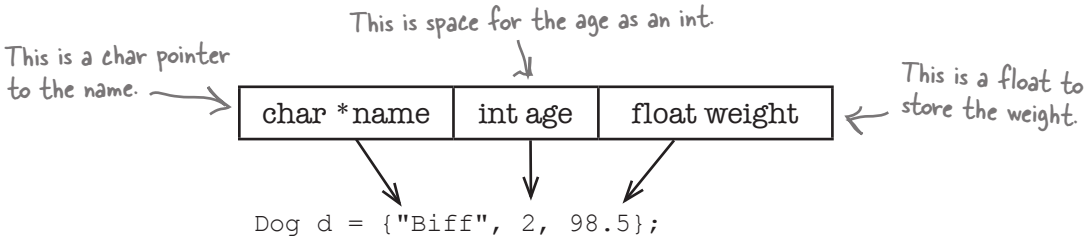
- **It will take up more space in memory.**
- **Someone might set more than one value.**
- **There's nothing called "quantity."**

It would be *really useful* if you could specify something called `quantity` in a data type and then decide for each particular piece of data whether you are going to record a count, a weight, or a volume against it.

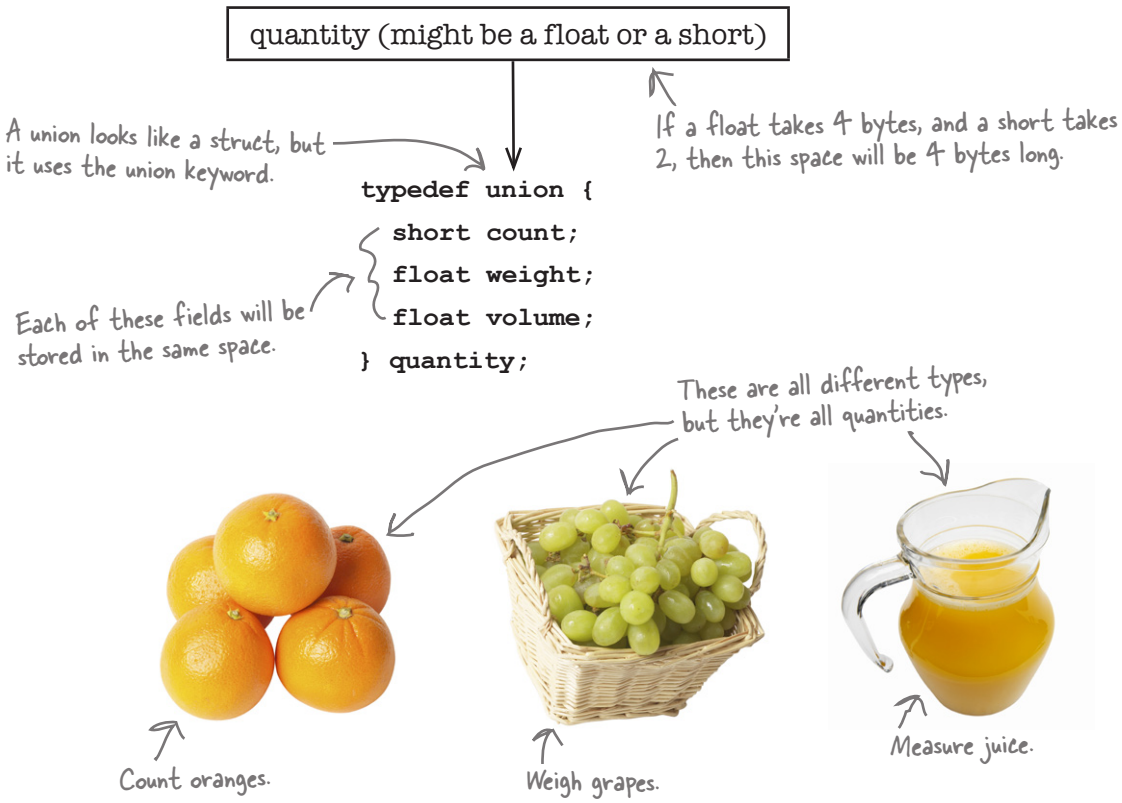
In C, you can do just that by using a union.

A union lets you reuse memory space

Every time you create an instance of a `struct`, the computer will lay out the fields in memory, one after the other:



A **union** is different. A `union` will use the space for just one of the fields in its definition. So, if you have a `union` called `quantity`, with fields called `count`, `weight`, and `volume`, the computer will give the `union` enough space for its largest field, and then leave it up to you which value you will store in there. Whether you set the `count`, `weight`, or `volume` field, the data will go into the same space in memory:



How do you use a union?

When you declare a `union` variable, there are a few ways of setting its value.

C89 style for the first field

If the `union` is going to store a value for the **first field**, then you can use C89 notation. To give the `union` a value for its first field, just wrap the value in braces:

```
quantity q = {4};
```

This means the quantity is a count of 4.

Designated initializers set other values

A **designated initializer** sets a `union` field value by **name**, like this:

```
quantity q = {.weight=1.5};
```

This will set the union for a floating-point weight value.

Set the value with dot notation

The third way of setting a `union` value is by creating the variable on one line, and setting a field value on another line:

```
quantity q;
q.volume = 3.7;
```

Remember: whichever way you set the `union`'s value, there will only ever be **one piece of data stored**. The `union` just gives you a way of creating a variable that supports ***several different data types***.

there are no Dumb Questions

Q: Why is a `union` always set to the size of the *largest* field?

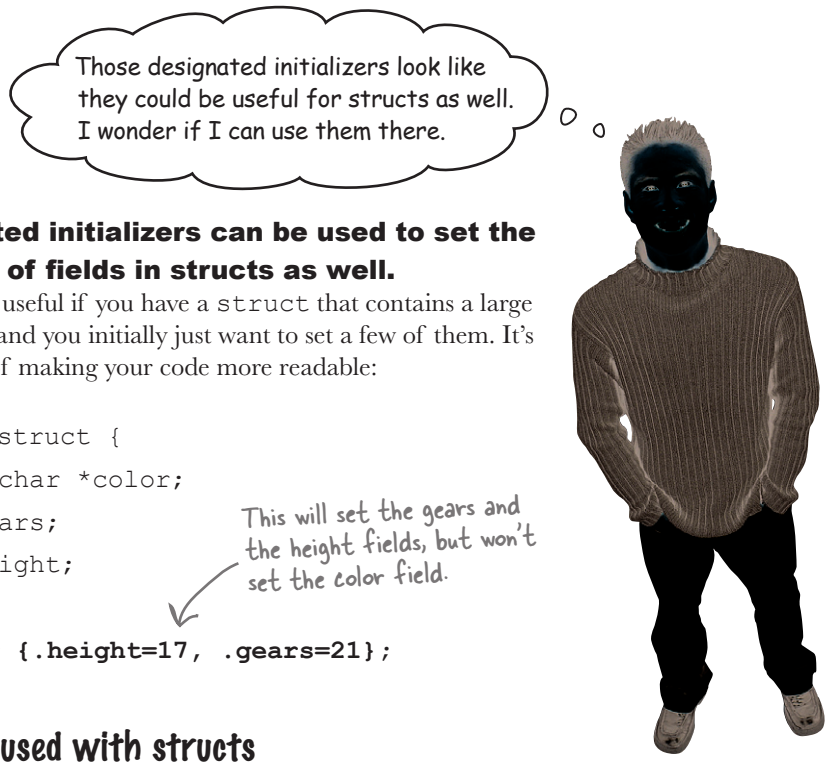
A: The computer needs to make sure that a `union` is always the same size. The only way it can do that is by making sure it is large enough to contain any of the fields.

Q: Why does the C89 notation only set the first field? Why not set it to the first `float` if I pass it a `float` value?

A: To avoid ambiguity. If you had, say, a `float` and a `double` field, should the computer store `{2.1}` as a `float` or a `double`? By always storing the value in the first field, you know exactly how the data will be initialized.

The Polite Guide to Standards

Designated initializers allow you to set `struct` and `union` fields by name and are part of the C99 C standard. They are supported by most modern compilers, but be careful if you are using some *variant* of the C language. For example, *Objective C* supports designated initializers, but *C++* **does not**.



Yes, designated initializers can be used to set the initial values of fields in structs as well.
They can be very useful if you have a `struct` that contains a large number of fields and you initially just want to set a few of them. It's also a good way of making your code more readable:

```
typedef struct {
  const char *color;
  int gears;
  int height;
} bike;
bike b = {.height=17, .gears=21};
```

This will set the gears and the height fields, but won't set the color field.

unions are often used with structs

Once you've created a `union`, you've created a *new data type*. That means you can use its values anywhere you would use another data type like an `int` or a `struct`. For example, you can combine them with `structs`:

```
typedef struct {
  const char *name;
  const char *country;
  quantity amount;
} fruit_order;
```

And you can access the values in the `struct/union` combination using the dot or `->` notation you used before:

Here, you're using a double designated identifier. .amount for the struct and .weight for the .amount.

It's .amount because that's the name of the struct quantity variable.

```
fruit_order apples = {"apples", "England", .amount.weight=4.2};
printf("This order contains %2.2f lbs of %s\n", apples.amount.weight, apples.name);
```

This will print "This order contains 4.20 lbs of apples."

Mixed-UP Mixers

It's Margarita Night at the Head First Lounge, but after one too many samples, it looks like the guys have mixed up their recipes. See if you can find the matching code fragments for the different margarita mixes.

Here are the basic ingredients:

```
typedef union {
  float lemon;
  int lime_pieces;
} lemon_lime;

typedef struct {
  float tequila;
  float cointreau;
  lemon_lime citrus;
} margarita;
```

Here are the different margaritas:

```
margarita m = {2.0, 1.0, {0.5}};
```

```
margarita m = {2.0, 1.0, .citrus.lemon=2};
```

```
margarita m = {2.0, 1.0, 0.5};
```

```
margarita m = {2.0, 1.0, {.lime_pieces=1}};
```

```
margarita m = {2.0, 1.0, {1}};
```

```
margarita m = {2.0, 1.0, {2}};
```

And finally, here are the different mixes and the drink recipes they produce. Which of the margaritas need to be added to these pieces of code to generate the correct recipes?

```
.....
printf("%2.1f measures of tequila\n%2.1f measures of cointreau\n%2.1f
    measures of juice\n", m.tequila, m.cointreau, m.citrus.lemon);

2.0 measures of tequila
1.0 measures of cointreau
2.0 measures of juice
```

```
.....
printf("%2.1f measures of tequila\n%2.1f measures of cointreau\n%2.1f
    measures of juice\n", m.tequila, m.cointreau, m.citrus.lemon);

2.0 measures of tequila
1.0 measures of cointreau
0.5 measures of juice
```

```
.....
printf("%2.1f measures of tequila\n%2.1f measures of cointreau\n%i pieces
    of lime\n", m.tequila, m.cointreau, m.citrus.lime_pieces);

2.0 measures of tequila
1.0 measures of cointreau
1 pieces of lime
```

BE the Compiler

One of these pieces of code compiles; the other doesn't. Your job is to play like you're the compiler and say which one compiles, and why the other one doesn't.

```
margarita m = {2.0, 1.0, {0.5}};
```

```
margarita m;
m = {2.0, 1.0, {0.5}};
```

Mixed-UP Mixers Solution

It's Margarita Night at the Head First Lounge, but after one too many samples, it looks like the guys have mixed up their recipes. You were to find the matching code fragments for the different margarita mixes.

Here are the basic ingredients:

```
typedef union {
  float lemon;
  int lime_pieces;
} lemon_lime;

typedef struct {
  float tequila;
  float cointreau;
  lemon_lime citrus;
} margarita;
```

Here are the different margaritas:

```
margarita m = {2.0, 1.0, .citrus.lemon=2};
```

```
margarita m = {2.0, 1.0, 0.5};
```

None of these lines was used.

```
margarita m = {2.0, 1.0, {1}};
```

And finally, here are the different mixes and the drink recipes they produce. Which of the margaritas need to be added to these pieces of code to generate the correct recipes?

```
margarita m = {2.0, 1.0, {2}};
...
printf("%2.1f measures of tequila\n%2.1f measures of cointreau\n%2.1f
    measures of juice\n", m.tequila, m.cointreau, m.citrus.lemon);

2.0 measures of tequila
1.0 measures of cointreau
2.0 measures of juice
```

```
margarita m = {2.0, 1.0, {0.5}};
...
printf("%2.1f measures of tequila\n%2.1f measures of cointreau\n%2.1f
    measures of juice\n", m.tequila, m.cointreau, m.citrus.lemon);

2.0 measures of tequila
1.0 measures of cointreau
0.5 measures of juice
```

```
margarita m = {2.0, 1.0, {.lime_pieces=1}};
...
printf("%2.1f measures of tequila\n%2.1f measures of cointreau\n%i pieces
    of lime\n", m.tequila, m.cointreau, m.citrus.lime_pieces);

2.0 measures of tequila
1.0 measures of cointreau
1 pieces of lime
```

BE the Compiler Solution

One of these pieces of code compiles; the other doesn't. Your job is to play like you're the compiler and say which one compiles, and why the other one doesn't.

```
margarita m = {2.0, 1.0, {0.5}};
```

This one compiles perfectly. It's actually just one of the drinks above!

```
margarita m;
m = {2.0, 1.0, {0.5}};
```

This one doesn't compile because the compiler will only know that {2.0, 1.0, {0.5}} represents a struct if it's used on the same line that a struct is declared. When it's on a separate line, the compiler thinks it's an array.

Hey, wait a minute... You're setting **all** these different values with all these different types and you're storing them in **the same place in memory**... How do I know if I stored a float in there once I've stored it? What's to stop me from reading it as a short or something??? **Hello?**

That's a really good point: you can store lots of possible values in a union, but you have *no way of knowing* what type it was once it's stored.
The compiler won't be able to keep track of the fields that are set and read in a `union`, so there's nothing to stop us setting one field and reading another. Is that a problem? Sometimes it can be a **BIG PROBLEM**.

```
#include <stdio.h>
typedef union {
  float weight;
  int count;
} cupcake;

int main()
{
  cupcake order = {2};
  printf("Cupcakes quantity: %i\n", order.count);
  return 0;
}
```

By mistake, the programmer has set the weight, not the count.

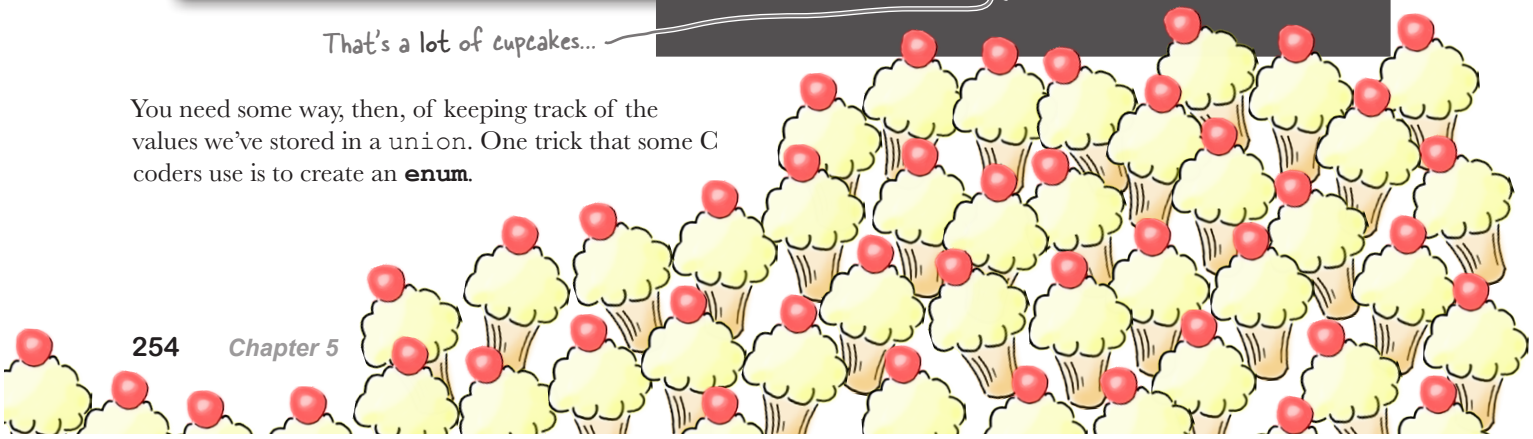
She set the weight, but she's reading the count.

This is what the program did.

```
File Edit Window Help
> gcc badunion.c -o badunion && ./badunion
Cupcakes quantity: 1073741824
```

That's a lot of cupcakes...

You need some way, then, of keeping track of the values we've stored in a `union`. One trick that some C coders use is to create an **`enum`**.



An enum variable stores a symbol

Sometimes you don't want to store a number or a piece of text. Instead, you want to store something from a list of **symbols**. If you want to record a day of the week, you only want to store MONDAY, TUESDAY, WEDNESDAY, etc. You don't need to store the text, because there are only ever going to be seven different values to choose from.

That's why enums were invented.

`enum` lets you create a list of symbols, like this:

Possible colors in your enum.

The values are separated by commas.

```
enum colors {RED, GREEN, PUCE};
```

You could have given the type a proper name with typedef.

Any variable that is defined with a type of **`enum colors`** can then only be set to one of the keywords in the list. So you might define an `enum colors` variable like this:

```
enum colors favorite = PUCE;
```

Watch it!

structs and unions separate items with semicolons (;), but enums use commas.

Under the covers, the computer will just assign numbers to each of the symbols in your list, and the `enum` will just store a number. But you don't need to worry about what the numbers are; your C code can just refer to the symbols. That'll make your code easier to read, and it will prevent storing values like `REB` or `PUSE`:

The computer will spot that this is not a legal value, so it won't compile.

Nope; I'm not compiling that; it's not on my list.

```
enum colors favorite = PUSE;
```

So that's how enums work, but how do they help you keep track of unions? Let's look at an example...

Code Magnets

Because you can create new data types with `enum`s, you can store them inside `struct`s and `union`s. In this program, an `enum` is being used to track the kinds of quantities being stored. Do you think you can work out where the missing pieces of code go?

```
#include <stdio.h>

typedef enum {
  COUNT, POUNDS, PINTS
} unit_of_measure;

typedef union {
  short count;
  float weight;
  float volume;
} quantity;

typedef struct {
  const char *name;
  const char *country;
  quantity amount;
  unit_of_measure units;
} fruit_order;

void display(fruit_order order)
{
  printf("This order contains ");

  if (.....== PINTS)

    printf("%2.2f pints of %s\n", order.amount......, order.name);
```

```
  else if (..... == .....)
    printf("%2.2f lbs of %s\n", order.amount.weight, order.name);
  else

    printf("%i %s\n", order.amount. ..... , order.name);
}

int main()
{

  fruit_order apples = {"apples", "England", .amount.count=144, .....};

  fruit_order strawberries = {"strawberries", "Spain", .amount. .....=17.6, POUNDS};

  fruit_order oj = {"orange juice", "U.S.A.", .amount.volume=10.5, .....};
  display(apples);
  display(strawberries);
  display(oj);
  return 0;
}
```

POUNDS

order.units

COUNT

weight

volume

count

PINTS

order.units

Code Magnets Solution

Because you can create new data types with `enum`s, you can store them inside `struct`s and `union`s. In this program, an `enum` is being used to track the kinds of quantities being stored. Were you able to work out where the missing pieces of code go?

```
#include <stdio.h>

typedef enum {
  COUNT, POUNDS, PINTS
} unit_of_measure;

typedef union {
  short count;
  float weight;
  float volume;
} quantity;

typedef struct {
  const char *name;
  const char *country;
  quantity amount;
  unit_of_measure units;
} fruit_order;

void display(fruit_order order)
{
  printf("This order contains ");

  if (order.units == PINTS)

    printf("%2.2f pints of %s\n", order.amount.volume, order.name);
```

order.units

volume

```
  else if ( order.units == POUNDS )
    printf("%2.2f lbs of %s\n", order.amount.weight, order.name);
  else
    printf("%i %s\n", order.amount. count , order.name);
}

int main()
{
  fruit_order apples = {"apples", "England", .amount.count=144, COUNT };
  fruit_order strawberries = {"strawberries", "Spain", .amount. weight =17.6, POUNDS};
  fruit_order oj = {"orange juice", "U.S.A.", .amount.volume=10.5, PINTS };
  display(apples);
  display(strawberries);
  display(oj);
  return 0;
}
```

When you run the program, you get this:

```
File Edit Window Help
> gcc enumtest.c -o enumtest
This order contains 144 apples
This order contains 17.60 lbs of strawberries
This order contains 10.50 pints of orange juice
```

union: …so I said to the code, "Hey, look. I don't care if you gave me a `float` or not. You asked for an `int`. You got an `int`."

struct: Dude, that was totally uncalled for.

union: That's what I said. It's totally uncalled for.

struct: Everyone knows you only have one storage location.

union: Exactly. Everything is one. I'm, like, Zen that way…

enum: What happened, dude?

struct: Shut up, `enum`. I mean, the guy was crossing the line.

union: I mean, if he had just left a record. You know, said, I stored this as an `int`. It just needed an `enum` or something.

enum: You want me to do what?

struct: Shut up, `enum`.

union: I mean, if he'd wanted to store several things at once, he should have called you, am I right?

struct: Order. That's what these people don't grasp.

enum: Ordering what?

struct: Separation and sequencing. I keep several things alongside each other. All at the same time, dude.

union: That's just my point.

struct: All. At. The. Same. Time.

enum: (Pause) So has there been a problem?

union: Please, `enum`? I mean these people just need to make a decision. Wanna store several things, use you. But store just one thing with different possible types? Dude's your man.

struct: I'm calling him.

union: Hey, wait…

enum: Who's he calling, dude?

struct/union: Shut up, `enum`.

union: Look, let's not cause any more problems here.

struct: Hello? Could I speak to the Bluetooth service, please?

union: Hey, let's just think about this.

struct: What do you mean, he'll give me a callback?

union: I'm just. This doesn't seem like a good idea.

struct: No, let me leave you a message, my friend.

union: Please, just put the phone down.

enum: Who's on the phone, dude?

struct: Be quiet, `enum`. Can't you see I'm on the phone here? Listen, you just tell him that if he wants to store a `float` and an `int`, he needs to come see me. Or I'm going to come see him. Understand me? Hello? Hello?

union: Easy, man. Just try to keep calm.

struct: On hold? They put me on ^*&^ing hold!

union: They what? Pass me the phone… Oh…that… man. The Eagles! I hate the Eagles…

enum: So if you pack your fields, is that why you're so fat?

struct: You are entering a world of pain, my friend.

Sometimes you want control at the bit level

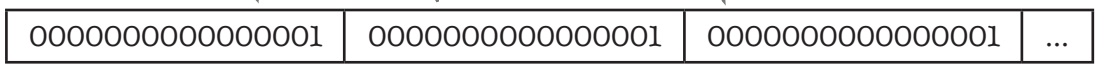
Let's say you need a `struct` that will contain a lot of yes/no values. You *could* create the `struct` with a series of `shorts` or `ints`:

```
typedef struct {
  short low_pass_vcf;
  short filter_coupler;
  short reverb;
  short sequential;
  ...
} synth;
```

Each of these fields will contain 1 for true or 0 for false.

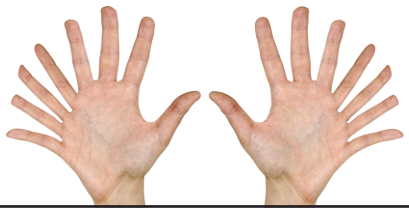
There are a lot more fields that follow this.

Each field will use many bits.



And that would work. The problem? The `short` fields will take up a lot more space than the *single bit* that you need for **true/false** values. It's wasteful. It would be much better if you could create a `struct` that could hold a sequence of single bits for the values.

That's why **`bitfields`** were created.



Geek Binary Digits

When you're dealing with binary value, it would be great if you had some way of specifying the 1s and 0s in a literal, like:

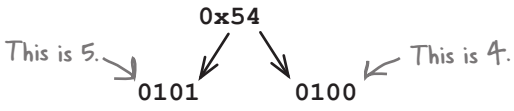
```
int x = 01010100;
```

Unfortunately, C doesn't support **binary literals**, but it *does* support **hexadecimal literals**. Every time C sees a number beginning with 0x, it treats the number as **base 16**:

This is not decimal 54.

```
int x = 0x54;
```

But how do you convert back and forth between hexadecimal and binary? And is it any easier than converting binary and **decimal**? The good news is that you can convert hex to binary **one digit at a time**:



Each hexadecimal digit matches a binary digit of length 4. All you need to learn are the binary patterns for the numbers 0–15, and you will soon be able to convert binary to hex and back again in your head within seconds.

Bitfields store a custom number of bits

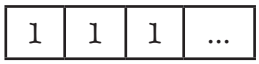
A **bitfield** lets you specify *how many bits* an individual field will store. For example, you could write your `struct` like this:

```
typedef struct {
  unsigned int low_pass_vcf:1;
  unsigned int filter_coupler:1;
  unsigned int reverb:1;
  unsigned int sequential:1;
  ...
} synth;
```

Each field should be an unsigned int.

This means the field will only use 1 bit of storage.

By using bitfields, you can make sure each field takes up only one bit.



If you have a sequence of bitfields, the computer can **squash them together** to save space. So if you have eight single-bit bitfields, the computer can store them in a single byte.

Watch it!

Bitfields can save space if they are collected together in a struct.

But if the compiler finds a single bitfield on its own, it might still have to pad it out to the size of a word. That's why bitfields are usually grouped together.

Let's see how how good you are at using bitfields.

How many bits do I need?

Bitfields can be used to store a sequence of true/false values, but they're also useful for other short-range values, like months of the year. If you want to store a month number in a `struct`, you know it will have a value of, say, 0–11. You can store those values in **4 bits**. Why? Because 4 bits let you store 0–15, but 3 bits only store 0–7.

```
...
 unsigned int month_no:4;
...
```

Back at the Head First Aquarium, they're creating a customer satisfaction survey. Let's see if you can use bitfields to create a matching `struct`.

Head First Aquarium

Aquarium Questionnaire

Is this your first visit?	
Will you come again?	
Number of fingers lost in the piranha tank:	
Did you lose a child in the shark exhibit?	
How many days a week would you visit if you could?	

```
typedef struct {
  unsigned int first_visit:      ;
  unsigned int come_again:       ;
  unsigned int fingers_lost:     ;
  unsigned int shark_attack:     ;
  unsigned int days_a_week:      ;
} survey;
```

You need to decide how many bits to use.

Back at the Head First Aquarium, they're creating a customer satisfaction survey. You were to use bitfields to create a matching `struct`.

Head First Aquarium

Aquarium Questionnaire

Is this your first visit?	
Will you come again?	
Number of fingers lost in the piranha tank:	
Did you lose a child in the shark exhibit?	
How many days a week would you visit if you could?	

```
typedef struct {
  unsigned int first_visit: 1 ;
  unsigned int come_again: 1 ;
  unsigned int fingers_lost: 4 ;
  unsigned int shark_attack: 1 ;
  unsigned int days_a_week: 3 ;
} survey;
```

1 bit can store 2 values: true/false.

4 bits are needed to store up to 10.

3 bits can store numbers up to 7.

there are no Dumb Questions

Q: Why doesn't C support binary literals?

A: Because they take up a lot of space, and it's usually more efficient to write hex values.

Q: Why do I need 4 bits to store a value up to 10?

A: Four bits can store values from 0 to binary 1111, which is 15. But 3 bits can only store values up to binary 111, which is 7.

Q: So what if I try to put the value 9 into a 3-bit field?

A: The computer will store a value of 1 in it, because 9 is 1001 in binary, so the computer transfers 001.

Q: Are bitfields really just used to save space?

A: No. They're important if you need to read low-level binary information.

Q: Such as?

A: If you're reading or writing some sort of custom binary file.

BULLET POINTS

- A `union` allows you to store different data types in the same memory location.
- A designated initializer sets a field value by name.
- Designated initializers are part of the C99 standard. They are not supported in C++.
- If you declare a `union` with a value in {braces}, it will be stored with the type of the first field.
- The compiler will let you store one field in a `union` and read a completely different field. But be careful! This can cause bugs.
- `enum`s store symbols.
- Bitfields allow you to store a field with a custom number of bits.
- Bitfields should be declared as `unsigned int`.

Your C Toolbox

You've got Chapter 5 under your belt, and now you've added structs, unions, and bitfields to your toolbox. For a complete list of tooltips in the book, see Appendix ii.

typedef lets you create an alias for a data type.

A struct combines data types together.

You can read struct fields with dot notation.

You can initialize structs with {array, like, notation}.

-> notation lets you easily update fields using a struct pointer.

Designated initializers let you set struct and union fields by name.

unions can hold different data types in one location.

enums let you create a set of symbols.

Bitfields give you control over the exact bits stored in a struct.