User-Defined Types and Pre-Processor Directives

Enum

Enum provides a way to create integer constants. For instance, in Java we might declare:

```
public static final int GOLD = 1;
public static final int SILVER = 2;
public static final int BRONZE = 3;
```

..but really what we want is to create a "medal" type that can take on the values GOLD, SILVER, and BRONZE. Moreover, we want GOLD to mean "1", SILVER to mean "2", and BRONZE to mean "3".

Defining Enums

We can do just that in C using an enum. Here's the format:

```
enum typeName {
     value1,
     value2,
     ...
     valueN
} objectList;
```

Here, `typeName` is the name of the new data type we're creating. `value1, value2, ..., valueN` are the different values this type can take. Finally, `objectList` is a list of variables we want to create of type `typeName`. In this definition, `typeName` and `objectList` are both optional.

Here's how we would represent medals using an enum:

```
enum medal {gold, silver, bronze};
```

Enum Variables

Now, the type of this enum is "`enum medal`". All variables of this type can have a value of either `gold`, `silver`, or `bronze`. Here's how to declare a variable:

```
enum medal medal100m;
```

And here's how to give this variable the value "`gold`"

```
medal100m = gold;
```

We can also use the possible values "`gold`", "`silver`", and "`bronze`" in comparisons:

```
if (medal100m == gold) {
     printf("You won!\n");
}
```

What's Going On?
When you declare an enum type, what you're really doing is making the enum values into integer constants. For example, in the medal enum, we're really creating the integer constants:

```
gold = 0;
silver = 1;
bronze = 2;
```

So, for instance, to set `medal100m` to `gold`, we could also say:

```
medal100m = 0;
```

Then the test:

```
if (medal100m == gold) {..}
```

would still evaluate to true, since `gold` holds the value 0.

Changing Enum Values
Our medal example isn't quite right. It has `gold` with the value 0, `silver` with 1, and `bronze` with 2. But we would like `gold` to have the value 1, `silver` 2, and `bronze` 3. Fortunately, there's a way to do that:

```
enum medal {gold = 1, silver, bronze};
```

Setting `gold` to 1 resets the numbering in an enum, so that `silver` gets set to 2 and `bronze` to 3. We also could have explicitly set each value:

```
enum medal {gold = 1, silver = 2, bronze = 3};
```

Here's another example:

```
enum example {val1, val2 = 9, val3, val4 = 12, val5};
```

Here's how the integer values were assigned:

```
val1 = 0;
val2 = 9;
val3 = 10;
```

```
val4 = 12;
val5 = 13;
```

An enum tries to keep numbering in order, unless we explicitly set a value. Then, it uses that value to reset the numbering.

Examples
There is no boolean type in C. However, we can create our own boolean type using an enum:

```
enum boolean {false, true};
```

(Notice that `false` has the value 0 and `true` has the value 1). Now we can use "boolean variables" like we do in Java:

```
enum boolean flag;
flag = true;
if (flag == true) {...}
```

We can also use an enum to represent grade levels in high school. We'll start freshman out at 9, and continue the numbering from there:

```
enum grade {freshman = 9, sophomore, junior, senior};
```

Now we can use this type:

```
enum grade class;
class = freshman;
class = 10;          //same as sophomore
```

Enums inside Structs
Since an enum is a valid type, we can put an enum field inside a struct. For example, suppose we want to create a student struct that has a name and a grade level. Here's how:

```
//define the enum type
enum grade {freshman = 9, sophomore, junior, senior};

//define the struct
struct student {
     char name[20];
     enum grade class;
};
```

This code works, but it's overkill. Instead of defining the enum separately, we want to be able to define it inside the struct. In fact, ALL we want is a variable of that enum type – we don't even

care about naming the type. The enum name is optional, so we'll leave it off. Here's our revised definition:

```
struct student {
     char name[20];
     enum {freshman = 9, sophomore, junior, senior} class;
};
```

Now, suppose we want to create a student named Mary who is a junior. Here's how:

```
struct student mary;
strcpy(mary.name, "Mary");
mary.class = junior;
```

Union

A union is a construct in C that can hold one of several types. A union variable can only hold one value at a time, unlike a struct, but that value is not restricted to a single type. Here's the format for declaring a union:

```
union modelName {
     type1 name1;
     type2 name2;
     ...
     typeN nameN;
} objectList;
```

Here, `modelName` is the name of the type you're creating, each "`type name`" is a type you want to be able to store in this union plus a name for it, and `objectList` is a list of variables you want to create with this union type. Both `modelName` and `objectList` are optional.

Defining Unions

Suppose I want a type that will allow me to store money as dollars (in a double field) or as yen (in an int field). Here's how I'd do that:

```
union money {
     double dollars;
     int yen;
} price;
```

This creates a variable called `price` that has type `union money`. I could also create a variable called price2 like this:

```
union money price2;
```

Union Fields
To access a field in a union, use the `.` notation – just like with a struct:

```
variableName.fieldName
```

(`variableName` is the name of the union variable, and `fieldName` is the name of the field that we want to access.) If we had a pointer to a union variable, we would use the `->` notation to access a field.

For example, I can make the `price` variable hold 17 yen:

```
price.yen = 17;
```

Or I can make the `price` variable hold $4.20:

```
price.dollars = 4.20;
```

Note that when I change the value of `price` from 17 yen to $4.20, I lose the information about 17 yen. A union variable can only hold one value at a time, but that value can have different types.

Unions and Enums
Suppose we have a variable of type `union money`, and we want to print out its value (in either dollars or yen, depending on what is stored). With what we've done so far, there is no way to tell WHICH type is stored in a union variable – it might be `yen` and it might be `dollars`. But if we do:

```
union money cost;
cost.yen = 17;
printf("Dollars: %lf\n", cost.dollars);
```

which sets the `yen` field but then prints the dollars `field`, the behavior is undefined. This code is likely to crash or at least have unpredictable results.

To solve this problem, we need to keep track of which type we're storing in a union variable. The most common way to do this is with an enum. For example, we could declare:

```
enum costK {dollarsK, yenK};
```

Then if we did:

```
union money cost;
cost.yen = 17;
```

We could also do:

```
enum costK costEnum;
costEnum = yenK;
```

Then we could tell which type was stored in our union variable by checking the value of our enum variable:

```
if (costEnum == yenK) }
     printf("Yen: %d\n", cost.yen);
}
else if (costEnum == dollarsK) {
     printf("Dollars: %lf\n", cost.dollars);
}
```

We will print the value of the field that we're currently using in the union.

Structs inside Unions

A struct is a valid type, so we can have a struct be a field in a union. For example, suppose we wanted to create an animal union that could be either a dog (struct type) or a cat (struct type). Furthermore, we want to keep track of the name and breed of the dog, and the name and color of the cat. Here's how:

```
union animal {
     struct {
          char name[20];
          char breed[20];
     } dog;

     struct {
          char name[20];
          char color[20];
     } cat;
};
```

Notice that we don't name the dog struct or the cat struct because we only want a single variable of that type. Instead, we create a single dog variable and a single cat variable. Here's how we create a union variable that holds an orange cat called Tiger:

```
union animal kitty;
strcpy(kitty.cat.name, "Tiger");
strcpy(kitty.cat.color, "orange");
```

Unions inside Structs

Similarly, we can put a union inside a struct. For example, suppose we wanted to keep track of someone's name, what country they're from, and how much their bill is (in the appropriate

currency). We could store the country with an enum, and the bill (with different currency) as a union. Here's how:

```
struct guest {
     char name[20];
     enum {USA, France, Japan} country;
     union {
          double dollars;
          double euro;
          int yen;
     } cost;
};
```

Now if we want the guest Maria from France, who paid 100 euro, here's what we'd do:

```
struct guest maria;
strcpy(maria.name, "Maria");
maria.country = France;
maria.cost.euro = 100.0;
```

Unions as Inheritance

The union construct allows us to simulate inheritance in Java. For example, suppose we defined the following abstract class in Java:

```
public abstract class Person {
     public String name;
     public int age;
}
```

Suppose as well that we have two Java classes, `Student` and `Employee`, which extend `Person`:

```
public class Student extends Person {
     public String major;
     public double gpa;
}
public class Employee extends Person {
     public String division;
     public int yearsWorked;
}
```

The equivalent in C would be a person structure that holds a `name`, `age`, and a union field. The union field could hold either a student struct (which has a `major` and `gpa`) or an employee struct (which has a `division` and a `yearsWorked`). We would also need an enum field that keeps track of which type in the union is active. Here's what it would look like:

```
struct person {
     char name[20];
     int age;

     union {
          struct {
               char major[20];
               double gpa;
          } student;
          struct {
               char division[20];
               int yearsWorked;
          } employee;
     } type;

     enum {employeeK, studentK} typeK;
};
```

Suppose a student has the following information:

- **Name:** Bob Jones
- **Age:** 18
- **Major:** EECE
- **GPA:** 3.2

Here's how we would create a variable to represent that student:

```
struct person bob;
strcpy(bob.name, "Bob Jones");
bob.age = 18;
strcpy(bob.type.student.major, "EECE");
bob.type.student.gpa = 3.2;
bob.typeK = studentK;
```

Typedef

It gets a bit cumbersome to use types called "`struct person`" and "`enum grade`". It would be much nicer to be able to call them just "`person`" or "`grade`". With C's typedef construct, we can do just that. The format of typedef is:

```
typedef oldType newType;
```

Here, we rename type `oldType` to the new name `newType`. We can now create variables using the name newType. For example:

```
typedef char letterGrade;
```

```
letterGrade g;
g = 'A';
```

Notice that we treat our new `letterGrade` type just like it was a char. The only difference is that when we declare the variable, we can use `letterGrade` as the type instead of char.

Typedef with Structs, Unions, and Enums
We can do a similar thing to rename structs, unions, and enums. For example, consider the following struct:

```
struct person {
     char name[20];
     int age;
};
```

The formal type of the struct is "`struct person`". Now, we want to rename the type to be just "person":

```
typedef struct person person;          //old type new type
```

Alternatively, we can declare the struct and rename it with typedef all on the same line:

```
//old type new type

typedef struct person {
     char name[20];
     int age;
} person;
```

However, we don't need to name the struct now, since we're always going to be using the new type name when creating variables of this type:

```
typedef struct {
     char name[20];
     int age;
} person;
```

Now we can declare and use a `person` variable:

```
person p;
strcpy(p.name, "Bill");
p.age = 22;
```

We can do a similar thing to rename unions and enums. Consider the following union:

```
union money {
```

```
        double dollars;
        int yen;
};
```

We can rename the union type to `money`:

```
typedef union {
        double dollars;
        int yen;
} money;
```

Now we can use "`money`" as the type name instead of "`union money`". Similarly, consider the following enum:

```
enum grade{freshman = 9, sophomore, junior, senior};
```

We can rename the enum type to `grade`:

```
typedef enum {freshman = 9, sophomore, junior, senior} grade;
```

Now we can use "`grade`" as the type name instead of "`enum grade`".

Typedef with Linked Lists
Consider the following structure for a node in a linked list:

```
struct node {
        int data;
        struct node *next;
};
```

We can try to rename the "`struct node`" type to "`node`" using typedef:

```
typedef struct {
        int data;
        node *next;
} node;
```

However, this will give us a compile error. The reason is that this struct is self-referential. Thus, when we declare the field "`node *next`" in the struct, the compiler hasn't yet seen that we're renaming the type to "`node`". If instead we list the field as

```
struct node *next;
```

we will also get a complaint, as we left off the name of the struct.

If you're using typedef on a self-referential struct, you need to include BOTH the name of the struct and the name of the renamed type. The fixed node struct looks like:

```
typedef struct node {
     int data;
     struct node *next;
} node;
```

Here's how we'd use it:

```
node *head = malloc(sizeof(node));
head->data = 4;
head->next = malloc(sizeof(node));
head->next->data = 7;
head->next->next = NULL;
```

This creates the linked list `4->7`.

User-Defined Types and Pointers

Consider the following struct:

```
typedef struct {
     char name[20];
     int age;

     union {
          struct {
               char major[20];
               double gpa;
          } student;
          struct {
               char division[20];
               int yearsWorked;
          } employee;
     } type;

     enum {employeeK, studentK} typeK;
} person;
```

Suppose we want to create a pointer to a struct variable with the following information:

- **Name:** Bob Jones
- **Student**
- **Age:** 18
- **Major:** EECE
- **GPA:** 3.2

First, we'd create a pointer of type person:

```
person *p;
```

Then we'd allocate memory:

```
p = malloc(sizeof(person));
```

And then we'd initialize the fields:

```
strcpy(p->name, "Bob Jones");
p->age = 18;
strcpy(p->type.student.major, "EECE");
p->type.student.gpa = 3.2;
p->typeK = studentK;
```

Notice that we use a `->` to access any fields in the struct (since our variable is a pointer). After we are inside an internal field like a union, we switch to `.` notation (since the union is not a pointer).

The C Pre-Processor

The C pre-processor is a special program that runs before the C compiler. It processes every line that begins with a #, such as a `#include` statement. The pre-processor may add, remove, or change your code when handling the # statements.

#include

We've been using the `#include` statement since our first program in order to get access to C's library functions (like `printf` in `stdio.h`). When the pre-processor sees a `#include` statement, it replaces the `#include` line with the contents of the included file. For example, when the pre-processor sees

```
#include <stdio.h>
```

then it replaces that line with the contents of `stdio.h` (`printf`, `scanf`, `fprintf`, etc.). Then, when the compiler sees a call to `printf`, it knows what you mean because `printf` is defined in your file.

Including a file like

```
#include <stdio.h>
```

tells the pre-processor to look for `stdio.h` in the standard include directory (which is where all the library files live). If instead you say

```
#include "stdio.h"
```

the pre-processor will first look for `stdio.h` in the current directory. If it can't find the file, it will hen look in the standard include directory.

#define

The `#define` statement tells the pre-processor to find all occurrences of one value in your code, and to replace them with another value. This can be used in two ways: to define *constants* and to define *macros* (simplified functions).

Constants

A constant in C is defined like this:

```
#define name value
```

Here, `name` is the name we're giving the constant, and `value` is the value we'd like it to have. Here's an example:

```
#define PI 3.14159
```

Now, we can use `PI` in our code when we want the value 3.14159. For example:

```
int radius = 10;
double area = PI * radius * radius;
```

When the pre-processor sees a `#define` constant, it replaces all occurrences of the constant's name with it's specified value. So, by the time the pre-processor gets done with it, the above code looks like:

```
int radius = 10;
double area = 3.14159 * radius * radius;
```

Macros

A macro is simplified function that includes a name and a list of arguments. They are defined using the `#define` statement, just like constants. For example:

```
#define SUM(a, b) a+b
```

When the preprocessor sees a call to the macro, it will replace the macro call with the macro formula. For example, if we do:

```
int x = 3;
int y = 4;
int result = SUM(x, y);
```

Then the pre-processor will change the last line to be:

```
int result = x + y;          //Uses x as a and y as b in the macro formula
```

Just like other pre-processor directives, the compiler never sees the macro itself. It only sees the final result, after the pre-processor has replaced the macro call with the macro formula.

As another example, let's try to write a macro that computes the difference of two squares:

```
#define DIFF_SQUARE(a, b) a*a - b*b        //This is not right!
```

This macro may look right (and this is how we would write a diffSquare function), but it has some problems. For example:

```
int x = 4;
int y = 3;
int c = 2;
int d = 1;

int result = DIFF_SQUARE(x-c, y-d);
```

The pre-processor will replace the call to `DIFF_SQUARE` with the macro value. It will use "`x-c`" as the value for `a`, and "`y-d`" as the value for `b`. Here's what the last line will become:

```
int result = x-c*x-c - y-d*y-d;
```

However, what we WANT is:

```
int result = (x-c)*(x-c) - (y-d)*(y-d);
```

To get this substitution, we need to add parentheses to the original macro:

```
#define DIFF_SQUARE(a, b) (a)*(a) - (b)*(b)
```

Here's a macro that returns the minimum of two numbers:

```
#define MIN(a, b) a < b ? a : b
```

This uses the *ternary conditional operator*. It means: if `a` is less then `b`, then `a` is the minimum. Otherwise, `b` is the minimum.

Here's a macro that eats all input left in the input buffer. This can be VERY useful as any unexpected input will stay in the input buffer. It reads one character at a time until it runs out of input.

```
#define FLUSH() while (getchar() != '\n')
```

#ifdef, #ifndef

There is also a pre-processor if-statement that checks to see whether or not a constant has been defined. Here's the format:

```
#ifdef (pre-processor constant)

code

#endif
```

The code inside this `#ifdef/#endif` block will only be compiled if the pre-processor constant was defined. For example:

```
#define SIZE 10

//(elsewhere)
#ifdef SIZE

int nums[SIZE];

#endif
```

In this example, we only include the "`int nums[SIZE];`" line in our program if the `SIZE` constant has already been defined.

There is a similar construct called `#ifndef`, which evaluates to true if a constant has NOT been defined. For example:

```
#ifndef SIZE

printf("Define SIZE!\n");

#endif
```

The `printf` statement will only be part of the program if the `SIZE` constant has not been defined. The `#ifndef` statement will be very useful when we start to break our programs into several files – it will help ensure that we don't end up including the same information twice when we link our files together.