Outline

- Primitive datatypes
- do-it-yourself (i.e., user-defined) datatypes
- flow of control

C provides primitive datatypes. I

- void <=> nothing, occupies no bytes
- void* Warning: void pointer can point to anything, needs to be cast
- int <=> integers, signed or unsigned, long, short
- int* <=> pointer to an integer
- The number of bytes read from an address pointed to by an int* corresponds to the number of bytes in the int.
- char <=> characters, signed or unsigned, occupy one byte per character.
- char* pointer to a byte
- float <=> a number represented by sign, exponent and mantissa
- double <=> a number represented by sign, exponent and mantissa, using twice as many bits as float
- double*, float*

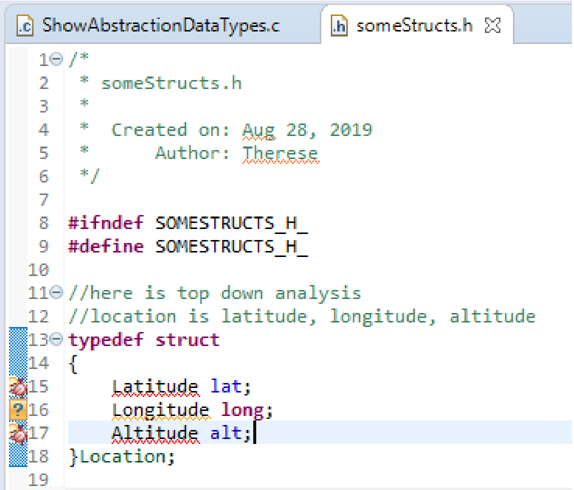
Using <stdbool.h>, C provides bool datatype.

- bool is used in logic, as you have seen (e.g., if(tests()) . . .)

It can be very useful to create our own datatypes.

- We can build levels of abstraction in our code with datatypes.
- Suppose we have the idea of location near planet Earth.
- We could analyze such a location using latitude, longitude and altitude.
- C gives us the syntax for this.

We can define a type that is a struct containing latitude, longitude and altitude.



```c
/*
 * someStructs.h
 *
 *  Created on: Aug 28, 2019
 *      Author: Therese
 */

#ifndef SOMESTRUCTS_H_
#define SOMESTRUCTS_H_

//here is top down analysis
//location is latitude, longitude, altitude
typedef struct
{
    Latitude lat;
    Longitude long;
    Altitude alt;
}Location;

```

Figure: Eclipse's C editor is producing warnings because it does not yet know these datatypes, specifically Latitude, Longitude, and Altitude.

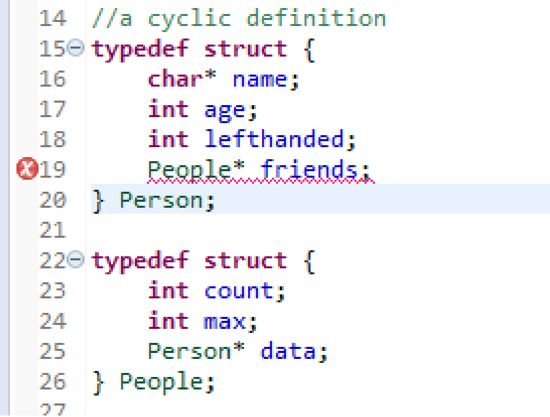
We can solve this particular problem by defining these before using them. I

```
//here is bottom up analysis
//location is latitude, longitude, altitude
typedef struct
{
    int hours;
    int minutes;
    int seconds;
}Latitude;
typedef struct
{
    int hours;
    int minutes;
    int seconds;
}Longitude;
```

We can solve this particular problem by defining these before using them. II

```
typedef struct
{
    double height;
}Altitude;
typedef struct
{
    Latitude lat;
     Longitude longit;
    Altitude alt;
}Location;
```

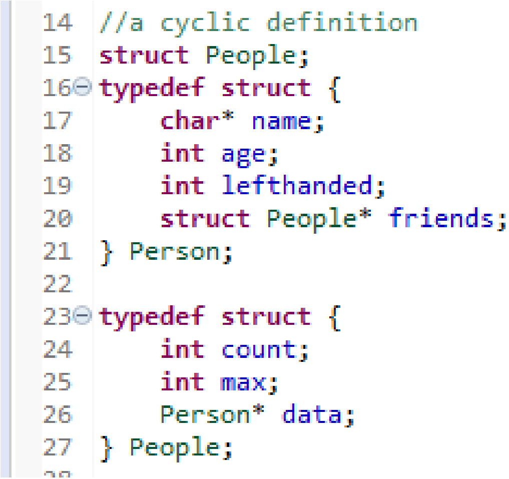
It is possible to have cyclic definitions. I



```c
//a cyclic definition
typedef struct {
    char* name;
    int age;
    int lefthanded;
    People* friends;
} Person;

typedef struct {
    int count;
    int max;
    Person* data;
} People;
```

Figure: Changing the order does not solve cyclic definitions.

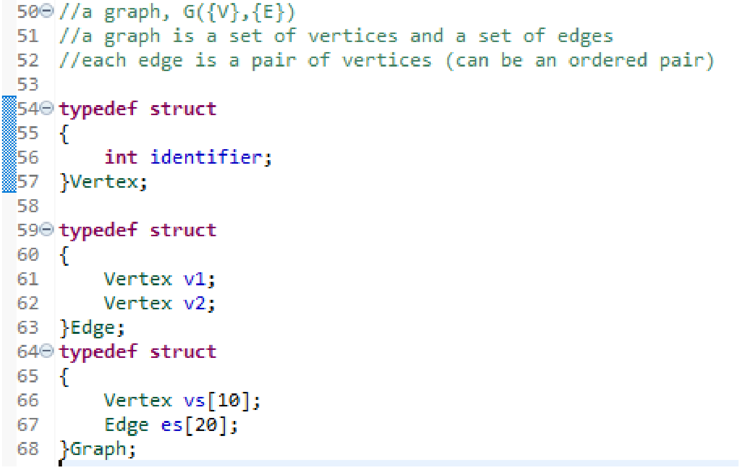
It is possible to have cyclic definitions. II



```c
//a cyclic definition
struct People;
typedef struct {
    char* name;
    int age;
    int lefthanded;
    struct People* friends;
} Person;

typedef struct {
    int count;
    int max;
    Person* data;
} People;
```

Figure: We solve it by informing the compiler that a definition of the struct will follow. We must thereafter use the keyword struct.

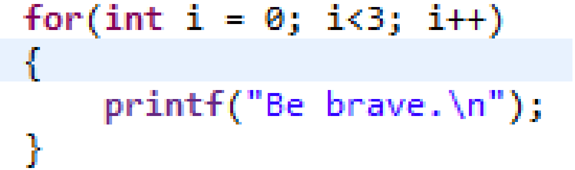
We can represent a graph.



```c
//a graph, G({V},{E})
//a graph is a set of vertices and a set of edges
//each edge is a pair of vertices (can be an ordered pair)

typedef struct
{
    int identifier;
}Vertex;

typedef struct
{
    Vertex v1;
    Vertex v2;
}Edge;
typedef struct
{
    Vertex vs[10];
    Edge es[20];
}Graph;
```

We often wish to control the flow of instruction execution. I

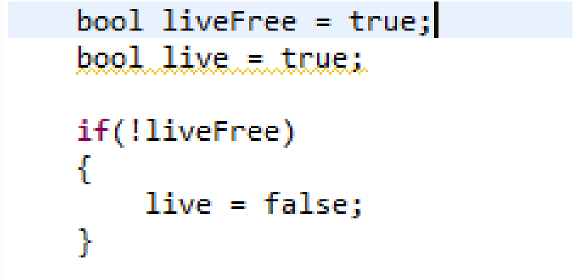
- We might know in advance how many times something is to be done.



```
for(int i = 0; i<3; i++)
{
    printf("Be brave.\n");
}
```

We often wish to control the flow of instruction execution. II

- We might not know the number of times, but we might know the deciding factor.



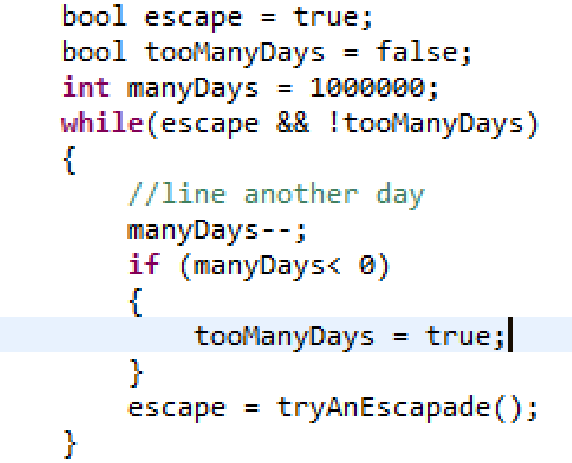
```
bool liveFree = true;
bool live = true;

if(!liveFree)
{
    live = false;
}
```

Figure: The motto of the state of New Hampshire is “Live free or die.”

We often wish to control the flow of instruction execution. III

- We might not know the number of times, but it could be determined during execution.

We often wish to control the flow of instruction execution. IV

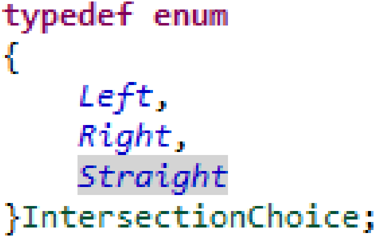


```
bool escape = true;
bool tooManyDays = false;
int manyDays = 1000000;
while(escape && !tooManyDays)
{
    //line another day
    manyDays--;
    if (manyDays< 0)
    {
        tooManyDays = true;
    }
    escape = tryAnEscapade();
}
```

Go To's are considered harmful.

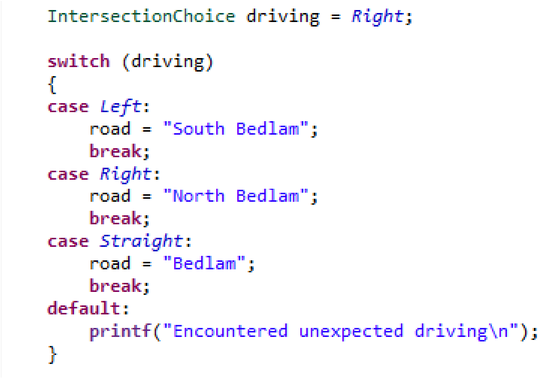
- Except for the switch construct, which we are about to see, the looping and conditional flow of control instructions seen above are sufficient.
- Your code will be better if you restrict yourself to the switch construct, and for/while/if.
- There are other instructions, including break and continue.
- Points will be lost from your homework and/or final if you use them. Only in switch may you use break.

Life includes multiple choice; the switch construct expresses that. I



```
typedef enum
{
    Left,
    Right,
    Straight
}IntersectionChoice;
```

Life includes multiple choice; the switch construct expresses that. II



```
IntersectionChoice driving = Right;

switch (driving)
{
case Left:
    road = "South Bedlam";
    break;
case Right:
    road = "North Bedlam";
    break;
case Straight:
    road = "Bedlam";
    break;
default:
    printf("Encountered unexpected driving\n");
}
```

Edsger Dijkstra wrote “Go To’s Considered Harmful”.

- You can read it here: `https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf`

There is higher level flow of control.

- Functions can call themselves (recursion), and
- other functions.
- If those other function carry on by calling the first function,
- we have mutual recursion.
- These are also flow of control, but
- flow of control often refers to the level of statements.
- The sequence diagram is another example of expressing higher level flow of control.