#include <stdio.h>
main() {
  char *pwest="west", *pnorth="north", *peast="east", *psouth="south";
  enum location { east=1, west=2, south=3, north=4};
  enum location direction;
  direction = east;
  if( direction == east )
    printf("Cannot go %s\n", peast); }

  enum month { jan = 1, feb=2, mar=3, apr=4, may=5, jun=6, jul=7, 
  aug=8, sep=9, oct=10, nov=11, dec=12 } this_month;
  this_month = feb;
  printf("What month is it? %d\n",this_month);
}
An enumerated type is one whose values are symbolic constant rather than literal.

Declaration example:

```c
enum Jar_Type {CUP, PINT, QUART, HALF_GALLON, GALLON};
```

The above example declares a type called `Jar_Type`

Variables of this type are declared like this:

```c
enum Jar_Type milk_jug, gas_can, medicine_bottle;
```

If there is only one declaration of variables of a particular enumerated type (i.e. no type name), both statements may be combined:

```c
enum { CUP, PINT, QUARTER, HALF_GALLON, GALLON} milk_jug, gas_can, medicine_bottle;
```
Enumerated Data Type (cont)

- Variables declared with an enumerated type are actually stored as integers.
- Internally, the symbolic names are treated as integer constants
  - By default, CUP = 0, PINT = 1, QUART = 2, etc.
- Caution: don’t mix them indiscriminately with integers – even though it is viable.
  - milk_jug = -623;
  - int a = PINT;
- The variable cannot be assigned any values outside those specified in the initialization list for the declaration of the enum type
The values of some variable may be required to remain constant throughout the program.

- using the qualifier `const` at the time of initialization in 2 ways:
  - `const int size = 40;`
  - `int const size = 40;`

The `const` data type qualifier tells the compiler that the value of the int variable `size` may not be modified in the program.

Assignment
- At declaration
- During function call for const parameters

Different from `#define` – both creating named constants
- `#define MAX_ELEMENTS 50       // literal constant`
- `int const max_elements = 50;  // variable constant`

Constant variable can only be used where variables are allowed; literal constant is allowed wherever a constant is allowed, such as in declaring the size of arrays.
Pointers and Constants

- int *pi;
  - pointer to int

- int const *pci;
  - point to constant int

- int *const cpi;
  - constant pointer to int

- int const *const cpci;
  - constant point to constant int
Storage Class

- Refers to the type of memory in which the variable’s value is stored which in turn defines different characters for the variable
  - Ordinary memory
  - Runtime stack
  - Hardware registers

- Determines when the variable is created and destroyed and how long it will retain its value

Introduction
- Auto → automatic
  - Local variable known only to the function in which it is declared; default storage class; stored in RAM (i.e. on the stack).
- Static
  - Local variable which exists and retains its value even after the control is transferred to the calling function; automatically initialized to zero; initialized only once during compilation; commonly used along with functions; stored in ordinary memory.
- Register
  - Local variables that are stored in the CPU memory register; must be initialized explicitly;

The default storage class for a variable depends on where it is declared.
Storage Classes - blocks

- Outside any blocks
  - Always stored in **static** memory (ordinary memory)
  - No way to specify any other storage class for these variables
  - Static variables are created before the program begins to run and exist throughout its entire execution.
  - They retain whatever value they were assigned until a different value is assigned or until the program completes

- Within a block
  - Default storage class is automatic
  - Stored on the stack
  - Keyword **auto** rarely used because doesn’t change default
  - Created just before the program execution enters the block in which they are declared;
  - Discarded just as execution leaves that block
  - New copies created each time the block is executed
Storage Classes – static

- Variables declared within a block but with the keyword static changes storage class from automatic to static
- Static storage class exists for the entire duration of the program, rather than just the duration of the block in which it is declared
- NOTE: the changing of the storage class of a variable does not change its scope; it is still accessible by name only from within the block
- FYI: formal parameters to a function cannot be declared static, because arguments are always passed on the stack to support recursion
When used in function definitions, or declarations of variables that appear outside of blocks

- The keyword `static` changes the linkage from external to internal*
- The storage class and scope are not affected
- Accessible only from within the source file in which they were declared

* Explain when go over linkage
Storage classes - register

- Can be used on automatic variables to indicate that they should be stored in the machine’s hardware registers rather than in memory.
- WHY? To be accessed more efficiently
- FYI – compiler can ignore if necessary i.e. too many designated as register (first come first served) → rest are automatic
- Smart Compiler? One that does its own register optimization so may ignore register class altogether and decide for itself skynet, is that you?
- Typically declare heavily used variables as register variables
- Created and destroyed at the same time as automatic variables (long story – previous values? will revisit; note: not allowed to take the address of a register variable)
Identifier Storage Class Summary

- Designated before the type
  - Automatic
    - Default
    - Local to a block
    - Discarded on exit from block
    - Can have `auto` specifier
    - Can have `register` specifier
      - Stored in fast registers of the machine if possible instead of RAM

- Static
  - Default for global variables
    - Declared prior to the main() function
  - Can also be defined within a function
  - Initialized at compile time and retains its value between calls... initial value must be a constant... be careful!
Suppose you want to write two functions, x and y, in the same source file, that use the variables given below. How and where would you write the declarations? NOTE: all initializations must be made in the declarations themselves, not by any executable statements in the functions.

The trick is to realize that function y can be put ahead of x in the file; after that, the rest is straightforward. Watch for assignment statements though; the problem specifies no executable statements in the functions.

```
static char b = 2;
void y( void )
{
}
int a = 1;

void x( void )
{
int c = 3;
static float d = 4;
}
```