Welcome to text-only Counterstrike.
You are in a dark, outdoor map.
> GO NORTH
You have been pwned by a grue.

© xkcd
Global Variables

- Remember how we said to never make a variable "outside" a function? Well, about that...
- Sometimes you need to make something that is shared between multiple functions:

```c
int global = 42;
int get_global()
{
    return global;
}
int set_global( int new_value )
{
    global = new_value;
}
```
Why not all global?

- At first, it seems like a great idea...
- Why not make ALL variables outside functions
- Then you can reuse them all over and you won't forget to declare "int i=0;" in all your functions
- It turns out to be a problem, but not in the way you might think
- Your neurology
Brain Variables

- Your brain only has about 7 (+/- 2) variables
- That's it
  - http://en.wikipedia.org/wiki/The_Magical_Number_Seven,_Plus_or_Minus_Two
- It's why we write out telephone numbers in groups, like:
  1 555 618 9230
  rather than
  1 5 5 5 6 1 8 9 2 3 0
- And it's why we have functions, to a degree
All at once

/*@ this is a nonsense example to prove a point
  * 7 different words and 5 different numbers 
*/
#include <stdio.h>
int main( int argc, char** argv )
{
    int i=0;
    int j=0;
    while( i < 26 )
    {
        j=0;
        while( j < 13 )
        {
            fprintf( stdout, "%d\n", string_length(argv[argc-1]) );
            j++;
        }
        i++;
    }
    return -42;
}
Plucked out - harder?

/* this is a nonsense example to prove a point
 * 10 different words and 5 different numbers
 */

#include <stdio.h>
int main( int argc, char** argv )
{
    do_stuff( argv[argc-1] );
    return -42;
}

void do_more_stuff( char* string )
{
    int j=0;
    while( j < 13 )
    {
        fprintf( stdout, "%d\n", string_length(string) );
        j++;
    }
}

void do_stuff( char* string )
{
    int i=0;
    while( i < 26 )
    {
        do_more_stuff(string);
        i++;
    }
}
\#include <stdio.h>
int main( int argc, char** argv )
{
    do_stuff( argv[argc-1] );
    return -42;
}
/* this is a nonsense example to prove a point
* 4 different words and 2 different numbers
*/

void do_stuff( char* string )
{
    int i=0;
    while( i < 26 )
    {
        do_more_stuff(string);
        i++;
    }
}
void do_more_stuff( char* string )
{
  int j=0;
  while( j < 13 )
  {
    fprintf( stdout, "%d\n", string_length(string) );
    j++;
  }
}
But with globals

- If we decide to make everything global...
- That means that **every** variable is visible to **every** function
/* this is a nonsense example to prove a point
* 6 different words and 3 different numbers */

#include <stdio.h>
int main( int argc, char** argv )
{
    do_stuff( argv[argc-1] );
    return -42;
}

int i = 0;
int j = 0;
void do_stuff( char* string )
{
    while( i < 26 )
    {
        do_more_stuff(string);
        i++;
    }
}

int i = 0;
int j = 0;
Plucked out - harder

/* this is a nonsense example to prove a point
 * 6 different words and 2 different numbers */

void do_more_stuff( char* string )
{
    while( j < 13 )
    {
        printf( stdout, "%d\n",
                string_length( string ) );
        j++;
    }
}

int i = 0;
int j = 0;
Less Globalization

• Lesson:
  • Don't overload your brain with options
  • Global variables make for more things you may or may not have to pay attention to, when trying to limit yourself to one function
  • However, we can make them "less bad"...
  • Unadorned, global variables are visible to all functions in all files – that makes the number of visible things a mess all over
  • But if we can at least limit a global variable to one file...
Static Cling

• A technical point in C is that in order to achieve "not being able to access a global variable outside the file", you need an additional "adjective" next to the global variable:

```c
static int global;
static int hours;
static int P;
```
Pointing Fingers

• So far in class we have glossed over the meaning of "char*" or "int*" or anything-else-*

• Today the mystery mobile comes to CS102 and clears things up

• And Scooby is hungry
Scooby Snacks, Anyone?

- Poor Scooby has no snacks. He needs someone to fill in the number of treats he has.
- Kinda like fgets(scratchpad) and fill(matrix)
- Why will this not work?

```c
/* in main: */
int snacks = 0;
put_snacks(snacks);
...
void put_snacks(
    int snacks )
{
    snacks = 10;
}
```
Snack Cloning

- Because variables don't go on rides to functions and back, remember?

- A copy of the contents of mains "snacks", which is currently zero, travels to set_snacks and is put in the variable "snacks" there.

- They have nothing to do with each other. They are different variables, so changing the one that happens to be called "snacks" in the function set_snacks doesn't affect main's "snacks".

```c
/* in main: */
int snacks = 0;
put_snacks(snacks);
...
void put_snacks(
    int snacks )
{
    snacks = 10;
}
```
#include <stdio.h>

void get_snacks(int snacks);

int main( int argc, char** argv )
{
    int snacks = 0;
    fprintf(stdout,"main's snacks=%d\n",snacks);
    put_snacks(snacks);
    fprintf(stdout,"main's snacks=%d\n",snacks);
}

void put_snacks(int snacks)
{
    fprintf(stdout,"gremlin's snacks=%d\n",snacks);
    snacks = 10;
    fprintf(stdout,"gremlin's snacks=%d\n",snacks);
}
Snacks Gremlin

Here is a zero, put it in your snacks bucket

Now put a 10 in your snacks bucket
Just Point Me Where

• What we need to do is tell set_snacks where main's "snacks" are, so it can change it for us.

• Conceptually:

/* in main: */
int snacks = 0;
put_snacks(
    location_of_snacks);
...
void put_snacks(
    location_of_snacks )
{
    go to where snacks are
    and change them to 10
}
Snacks Gremlin

"yellow bucket"

location of snacks

here is a location, put it in your "location of snacks" bucket

YOINK!

snacks

10

location of snacks

put a 10 wherever location of snacks points to (which happens to be "yellow bucket")
Syntactic Sugar (My Favorite)

- This is a variable that stores an integer:
  ```
  int snacks;
  ```

- This is the location of snacks:
  ```
  &snacks
  ```

- This is the type of variable that stores where an integer (like snacks) lives:
  ```
  int* location_of_an_integer;
  ```

- And this could be where the snacks are:
  ```
  int* snack_location = &snacks;
  ```
• So we've solved half the mystery and Scooby is almost happy to find out that he has 10 Scooby snacks...

```c
/* in main: */
int snacks = 0;
put_snacks( &snacks );
...
void put_snacks(
   int* location_of_snacks
)
{
   somehow use
   location_of_snacks and change what's there to 10
}
```
int snacks
here is a memory address, put it in your location bucket

int snacks
put a 10 wherever location is pointing (which happens to be &snacks)
#include <stdio.h>
void put_snacks(int* location);
int main( int argc, char** argv )
{
    int snacks = 0;
    fprintf(stdout,"main's snacks=%d\n",snacks);
    fprintf(stdout,"main's &snacks=%p\n",&snacks);
    put_snacks(&snacks);
    fprintf(stdout,"main's snacks=%d\n",snacks);
    fprintf(stdout,"main's &snacks=%p\n",&snacks);
}
void put_snacks(int* location)
{
    fprintf(stdout,"gremlin's location=%p\n",location);
    fprintf(stdout,"what's at location=%d\n",*location);
    *location = 10;
    fprintf(stdout,"gremlin's location=%p\n",location);
    fprintf(stdout,"what's at location=%d\n",*location);
}
More Syntactic Sugar (Scooby's Favorite)

- If this is a variable that stores where an integer (like snacks) lives:
  \[
  \text{int* location_of_an_integer;}
  \]
- Then this is "what's at that location":
  \[
  *\text{location_of_an_integer}
  \]
- So this makes "what's at the location of snacks" equal to 10:
  \[
  *\text{snack_location} = 10;
  \]
• Yum...

    /* in main: */
    int snacks = 0;
    put_snacks( &snacks );
    /* now snacks is 10 */
...

    void put_snacks(
        int* location
    )
    {
        *location = 10;
    }
Identical

```c
int snacks;

snacks = 10;

*(where) = 10;
```

```c
int snacks;

*(&snacks) = 10;
```

```c
int snacks;

int* where;

where = &snacks;

*(where) = 10;
```
#include <stdio.h>
void put_snacks(int* location);
int main( int argc, char** argv )
{
    int snacks = 0;
    int* secret_stash;
    fprintf(stdout,"main's snacks=%d\n",snacks);
    fprintf(stdout,"main's &snacks=%p\n",&snacks);
    secret_stash = &snacks;
    put_snacks(secret_stash);
    fprintf(stdout,"main's snacks=%d\n",snacks);
    fprintf(stdout,"main's &snacks=%p\n",&snacks);
}
void put_snacks(int* location)
{
    fprintf(stdout,"gremlin's location=%p\n",location);
    fprintf(stdout,"what's at location=%d\n",*location);
    *location = 10;
    fprintf(stdout,"gremlin's location=%p\n",location);
    fprintf(stdout,"what's at location=%d\n",*location);
}


int get_snacks( int snacks )
{
    return snacks;
}

int get_snacks( int* where )
{
    return *where;
}

x = get_snacks(snacks);  
x = get_snacks(&snacks);
Here is a memory address, put it in your location bucket.

Return a copy of whatever location is pointing to (which happens to be &snacks).
#include <stdio.h>

int get_snacks(int* location);

int main( int argc, char** argv )
{
    int snacks = 10;
    int scoobys_mouth = 0;
    fprintf(stdout,"main's snacks=%d\n",snacks);
    fprintf(stdout,"scooby's mouth=%d\n",scoobys_mouth);
    scoobys_mouth = get_snacks(&snacks);
    fprintf(stdout,"main's snacks=%d\n",snacks);
    fprintf(stdout,"scooby's mouth=%d\n",scoobys_mouth);
}

int get_snacks(int* location)
{
    fprintf(stdout,"gremlin's location=%p\n",location);
    fprintf(stdout,"what's at location=%d\n",*location);
    return *location;
}
Redirect

• I taught you that so I can teach you this :-)

• \textit{The sscanf()} function is almost the opposite of fprintf()
  
  int i;
  char string[10] = "123";
  sscanf( string, "%d", &i );

• Note the \& next to the i
More Args

• Likewise, you can read two things:
  ```c
  int i;
  char string[10] = "123 sean";
  char name[10];
  sscanf( string, "%d %s", &i, name );
  ```

• Note the & next to the i and not the name!
Failure is an Option

- What if we gave sscanf bad data and then expected it to work?
  ```c
  int i;
  char string[10] = "sean";
  int num_found = sscanf( string, "%d", &i );
  ```

- Well, as far as I know, I'm not an integer, so this won't work. `Sscanf` returns the number of things it successfully converted. In this case 0.

- Whereas:
  ```c
  int i;
  char string[10] = "123";
  int num_found = sscanf( string, "%d", &i );
  ```

- Returns 1 because 1 placeholder was filled

- If you're looking to fill 2 placeholders, look for a 2
Static strings

- If you know that you're looking for a specific string and then a variable, you can bake that into the second argument.

- So if you are looking for a number after "foo":
  ```c
  int i;
  char string[10] = "foo 1";
  int num_found = sscanf( string, "foo %d", &i );
  ```

- Or for a string after "foo":
  ```c
  char name[10];
  char string[10] = "foo sean";
  int num_found = sscanf( string, "foo %s", name );
  ```

- (Bookmark this page for later in the labs)
Output Files

- Go back to the files lecture for a refresher
- Open the file as normal, but with "w" instead of "r"
- Now we get to fprintf to the filehandle instead of fgets'ing from it - syntax is the same as stdout, but use the FILE*'s name instead of stdout

```c
char filename[10] = "foo";
FILE* writeme = fopen(filename,"w");
if( writeme != NULL ) {
    fprintf( writeme, "hello!\n" );
    fclose( writeme );
}
```
# include <stdio.h>
int main( int argc, char** argv )
{
    char filename[10] = "foo";
    FILE* writeme = fopen( filename, "w" );
    if( writeme != NULL ) {
        fprintf( writeme, "hello!\n" );
        fclose( writeme );
    }
}

-bash-4.1$ gcc example5.c -o example5
-bash-4.1$ ./example5
-bash-4.1$ cat foo
hello!
Else If

- Note that you can stack if statements like this:
  ```c
  if( /* something */ )
  {
  }
  else if( /* another case */ )
  {
  }
  else if( /* yet another case */ )
  {
  }
  else
  {
  }
  ```
Incomparable

• If you have two strings ("things like this" or char*'s), you cannot compare them like with integers, for reasons we will get into another week:

```c
void f( char * s, char * s2 )
{
    if( s == "foo" ) /* WRONG */
    ...else if( s == s2 ) /* also WRONG */
    ...else if( "foo" == "bar" ) /* also WRONG */
    ...else if( s[0] == s2[0] )
    /* will only compare first char but does work */
```
Struct

- Just like in Python, where you can have an "object" that is full of several things
- You can do the same in C using a word called **struct**
Struct Definition

- To make a "structure" called a, containing two integers x and y:
  
  ```c
  struct {
    int x;
    int y;
  } a;
  ```

- And then make "a's x" -1 and "a's y" 42:
  
  ```c
  a.x = -1;
  a.y = 42;
  ```
Stereotyping Variables

- Let's say we are going to have a bunch of "iPad" variables in our program. Structs that have a color and a number of GB memory:

```c
struct {
    int gb;
    char color[32];
} ipad1;
```

- Which creates a thing with an int and a string in it called **ipad1**.

- And we're going to be making a ton of these, having to type all that stuff over and over and over...

```c
struct { int gb; char color[32]; } ipad2;
struct { int gb; char color[32]; } ipad3;
struct { int gb; char color[32]; } ipad4;
```
Typedef

- Instead, we can make an "alias" for all that junk that's long to type...

```c
typedef struct {
    int gb;
    char color[32];
} ipad_t;
```

- Now, we do not have a `thing` with an integer and a string in it... we have `type of thing`. (Hence `_t` – which stands for type.)
Much Easier

- This allows us to then be able to do this:
  
  ```
  ipad_t ipad;
  ipad_t bunch_of_ipads[100];
  ```

- Much easier!

- We can likewise do this for any variable type, if we really wanted:
  
  ```
  typedef int integer;
  integer foo;
  ```
Typedef location

- Incidentally, if we make typedefs in a 1-page program, they go up top with the prototypes:

```
#include <stdio.h>
typedef struct { int gb; char color[32] } ipad_t;
void some_function( int x );
int main( int argc, char **argv )...
```

- And if you are in a multi-page program (main.c, functions.c, functions.h), then it belongs in the header (.h) file:

```
/* functions.h */
typedef struct { int gb; char color[32] } ipad_t;
void some_function( int x );
```
Maze Game - The Breakdown

- For project 11, we're making a maze game
- It'll start out as a 1-D maze, then 2-D
- First problem: make a 1-D maze full of rooms that are either a wall or not
  - maze.c and maze.h - Simpler than matrix multiply
- Second problem: be able to navigate the maze
  - Much like reading stdin for your calculator
- Third problem: be able to save/load a maze
  - file.c and file.h - Similar to printing a matrix
This year's new experiment

- Every year I try something new
- This year I'm running the labs for project 11 differently
- I've actually done the labs for you...
- Sort of
- If you look in cs102e/lab-11/common
  You'll see a bunch of lab0, lab1, lab2 directories
- I've done your labs but for a parallel project that's not exactly identical, but it does follow the same syntax, and it has a lot of explanations for things in it
- So while trying to go from lab2 to lab3, follow along with the differences between common/lab2 and common/lab3 files
Lab 1 - Empty Maze

- All in main.c for now
  - typedef struct { int wall; } room_t;
  - make an array of 10 rooms called maze
- In main:
  - Fill each one's wall with 0 (meaning no wall)
  - Set room 3's wall to 1 (i.e. the fourth room)
  - In a loop, print each room:
    - If that room has a wall, print "##"
    - Otherwise print ". " (dot and space)
- Just for lab1, there's a lab0 and lab1 example in "common", to break it down even further
Lab 1 Output

#= ./maze

.... ##....

#>
Lab 2 - Maze and a Player

- Still all in main.c
- In addition to the maze, you need to keep track of "where you are" in the maze, so make an integer called location_x
- Fill it with 2
- When you're printing, change it to be:
  - If the room I'm printing is in location_x, print "x " (x, space)
  - Otherwise the other stuff from lab 1

git commit
Lab 2 Output

#> ./maze

.. x ###.......

#>
Lab 3 - Functions, still in main.c

- Make the **maze** array of room_t's a global variable and **static**
- 3 new functions:
  - `void clear_maze()` - Move the setting of all the walls to zero in here
  - `int is_wall(int x)` - Return room x's wall
  - `void set_wall(int x, int w)` - Set room x's wall to w
- You'll need to use these 3 functions in main in the place where you used to do the work in main:
  - clear the maze - using your new function to do so
  - set room 3's wall to 1 - using your new function to do so
  - in your printing loop, you'll need to use `is_wall()`
- (no change to program output)

`git add
git commit`
Lab 4 - Move maze to maze.c

- Move the typedef to maze.h
- Move the **maze** array of room_t's maze.c
- And all 3 functions
- You'll need to use these 3 functions in main in the place where you used to do the work in main
- (no change to program output)

git add
git commit
Lab 5 - Move player to maze.c

- Move `location_x` to maze.c
  - Also make it a global variable and `static`
- 3 new functions:
  - `void set_location_x( int x )` - Set `location_x` to the given `x`
  - `int get_location_x()` - Return the `x` location
  - `void print_maze()` - Print the maze
- You'll need to replace the stuff left over in main with calls to these functions. Main should be fairly empty now, just use the functions to:
  - clear the maze
  - make the room 3's wall 1
  - make the x location 2
  - print maze
- (again no change to program output)
Lab 6 - Moving Around

- New functions in maze.c
  - move_right()
  - move_left()

- Algorithm for moving right:
  - make a copy of your x location
  - make it bigger by 1
  - if it there is not a wall at that position AND that position is within range (not greater than 9), then make set your x location to that position

- Make the equivalent for moving left

- Meanwhile in main, instead of just printing the maze:
  - print the maze, move right, print the maze, move left, print the maze, move left, print the maze

git commit
Lab 6 Output

#> ./maze

. . x ###. . . . . .
. . x ###. . . . . .
. . x ###. . . . . .
. x . ###. . . . . .
. . . ###. . . . . .
. . . ###. . . . . .
. . . ###. . . . . .
. . . ###. . . . . .
. . . ###. . . . . .
#>
Lab 7 - Deliberate Input

- Remember reading from a file from last week?
- Now read from `stdin`
- You don't have to open it, it's always open (like stdout), only it reads from the screen instead of a file
- Instead of all the print/move/print/move stuff:
  - while you can read from stdin
    - if the first letter of your line is 'l' (ell), move left
    - otherwise if the first letter is 'r', move right
    - print the maze right before the loop ends
- Try it out, you have a moveable character in a very small maze
Lab 7 Output

Using the command:

```
#> ./maze
```

We get the following output:

```
. . x ##. . . . . .
l
. x . ##. . . . . .
```

And then:

```
. . x #. . . . . .
r
. . x #. . . . . .
```
Lab 8 - Map Edit

• In your input loop in main:
  • previously, you were just looking for an 'l' and an 'r' character
  • if those checks fail, use sscanf to look for "wall" and an integer,
    – if you find it, set the wall at that number's location to 1
  • otherwise, look for "empty" and an integer
    – if you find it, set the wall at that number's location to 0

  (Hint: look for the bookmarked slide)
Lab 8 Output

`> ./maze`

```
. . x ##. . . . .
empty 3
. x . . . . . . . .
wall 4
. . x . ##. . . . 
```

`r`

```
. . . x ##. . . . 
```
Lab 9 - Load

- 3 new files: file.c, file.h, test.maze
- 1 new function:
  - void load( char* filename )
    - Open that filename for reading
    - While you can read a line
      - Use sscanf to find "WALL" and a number
        - If you found it, set the wall at that room number to 1
- In main, after clearing the maze, don't set any walls any more, instead load "test.maze"

git add
git commit
test.maze

WALL 5
Lab 9 Output

#> ./maze

.. x .. ##. . . .

r

.. . x . ##. . . .
Lab 10 - Save

- 1 new function:
  - void save( char* filename )
    - Open that filename for writing
    - for each x from 0 to 9
      - if there's a wall at that room, print "WALL" and the number
- In main, after loading "test.maze", add a wall at position 9 and then save "test2.maze"
- (Note the slide example for writing a file)

git commit
Lab 10 Output

#> ./maze
.
.x.
.

r
.
.
.x.
.

^D

#> cat test2.maze

WALL 5

WALL 9
Lab 11 - Load and Save on Command

• Remove the load, the save, the wall-setting, and the location setting from main

• Make sure clear_maze sets the location to 0

• Instead, in addition to looking for l and r and wall and empty, also look for:
  • "save" and a string - and if you find it, save to that filename
  • "load" and a string - and if you find it, load from that filename
Lab 11 Output

```bash
#> ./maze
x . . . . . . . . .
load test2.maze
x . . . . ##. . . ##
empty 5
x . . . . . . . . ##
save test3.maze
x . . . . . . . . ##
^D
#> cat test2.maze
WALL 5
WALL 9
#> cat test3.maze
WALL 9
```
Lab 12 - Edit/Save/Load Location

- New in main.c:
  - If the command has an "x" and then a number, set the location to that number

- New in file.c:
  - If you find a line that has "X" and a number (much like "WALL" and a number), set the location to that number
  - When saving, save one line with "X #" where # is the current location
test4.maze

WALL 9

X 5
#.> ./maze
x . ......... 
load test4.maze
..... x .... ##
1
..... x .... ##
save test5.maze
..... x ..... ##
^D
#.> cat test5.maze
WALL 9
X 4