# John Romero Programming Proverbs

- 3. “Keep your code absolutely simple. Keep looking at your functions and figure out how you can simplify further.”

- John Romero, “The Early Days of Id Software - John Romero @ WeAreDevelopers Conference 2017”

# C compound data structures

- in C we can declare a `struct` using the following mechanism:

```
struct name {
  int x;
  int y;
}

main ()
{
   struct name foo;

   foo.x = 1;
   foo.y = 2;
}
```

# Using typedefs

- we can introduce a `typedef` to clean up the previous code:

```
typedef struct name_s {
  int x;
  int y;
} name;

main ()
{
   name foo;

   foo.x = 1;
   foo.y = 2;
}
```

- notice how we still declare the `struct`, but wrap it in a `typedef`.

# Forward declaring structs

- sometimes a data structure needs to refer to itself
  - such as a binary tree
  - linked list etc
- consider a single linked list with a `value` and a `next` field
- this could be declared as follows

# Forward declaring structs

```
#include <stdlib.h>

typedef struct node_s node_t;
typedef node_t *node;

struct node_s {
               int value;
               node next;
             };

static node n;

main ()
{
  node foo = (node) malloc (sizeof (*foo));
  foo->value = 1;
  foo->next = foo;
}
```

# Simplified node declaration

```
#include <stdlib.h>

typedef struct node_s *node;

struct node_s {
                int value;
                node next;
              };

static node n;

main ()
{
  node foo = (node) malloc (sizeof (*foo));

  foo->value = 1;
  foo->next = foo;
}
```

# Simplified node declaration

- notice how we have re factored

- ```
  typedef struct node_s node_t;

  typedef node_t *node;
  ```
  ```

# Simplified node declaration

- to

```
typedef struct node_s *node;

struct node_s {
                int value;
                node next;
              };
```

- also note that the `typedef` is using a forward declaration of `struct`
  - which is completed in the next code line

# struct's

- are the only data type in C which can be partially declared (or forward declared)
- they must be completed though at some later point
- `structs` can be used to wrap up other data types
  - which can be useful if these other data types are self referential

# Example 1: declare a pointer to an array of 10 elements to itself

```
#include <stdlib.h>

typedef struct mydecl_a *mydecl;

struct mydecl_a { mydecl array[10+1]; };
static mydecl a;

main ()
{
  a = (mydecl) malloc (sizeof *a);
  a->array[0] = a;
}
```

- of course this is a pedalogical example as it does not reference anything else!

# Example 1: declare a pointer to an array of 10 elements to itself

- notice that we cannot join the `typedef` with the `struct` declaration like this:

```
typedef struct mydecl_a { mydecl array[10+1]; } *mydecl;

/* this will not work.  */
```

- as the definition for `mydecl` involves a use of `mydecl`