Variable Qualifiers to be Discussed:

- Storage Classes:
  - `auto`
  - `register`
  - `static`
  - `extern`
- \- and others:
  - `restrict`
  - `const`
  - `volatile`

The `auto` storage class – 'Automatic'

- **`auto`** is the default for function/block variables
  - **`auto int a`** is the same as **`int a`**
  - *because it is the default, the keyword is almost never used*
- storage is automatically allocated on function/block entry and automatically freed when the function/block is exited
- may not be used with global variables (which have storage space that exists for the life of the program)
- From <http://icecube.wisc.edu/~dglo/c_class/vstorage.html>

```
{
   int Inc_1;
   auto int Inc_2;  // both variables are of the same class...
}
```

The `register` *storage class – 'Optimization Hint'*

- The register storage class is used to define local variables that should be stored in a register instead of RAM. This means that:
  - The variable has a maximum size equal to the register size (usually one word)
  - The variable can't have the unary '&' operator applied to it (as it does not have a memory location).

```
{
   register int  miles;
}
```

- From: <http://www.tutorialspoint.com/cprogramming/c_storage_classes.htm>

- **`register`** provides a hint to the compiler that you think a variable will be frequently used
- The compiler is free to ignore register hint
- if ignored, the variable is equivalent to an auto variable with the exception that you may not take the address of a register (since, if put in a register, the variable will not have an address)
- It is rarely used, since any modern compiler will do a better job of optimization than most programmers
- From <http://icecube.wisc.edu/~dglo/c_class/vstorage.html>

The `static` storage class – 'File-Private Variables'

- The `static` storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.
- Another use for the static keyword is to ensure that code outside this file (code linked in later) cannot modify variables that are globally declared inside this file
  - If declare.c had declared farvar as:
    ```
    static int farvar;
    ```
    then the **`extern int farvar`** statement in use.c would cause an error
  - This use of static is commonly used in situations where a group of functions need to share information but do not want to risk other functions changing their internal variables

```
static int do_ping = 1;/* start with `PING' */
void ping(void){
  if (do_ping == 1) {
    printf("PING ");
    do_ping = 0;
  }
}

void pong(void) {
  if (do_ping == 0) {
    printf("PONG\n");
    do_ping = 1;
  }
}
```

- From: <http://www.tutorialspoint.com/cprogramming/c_storage_classes.htm>
- From <http://icecube.wisc.edu/~dglo/c_class/vstorage.html>

The `extern` *storage class – 'External References'*

- If a variable is declared (with global scope) in one file but referenced in another, the extern keyword is used to inform the compiler of the variable's existence.
- The extern storage class is used to give a reference of a global variable that is visible to ALL the program files.
- Note that the extern keyword is for declarations, not definitions
  - An extern declaration does not create any storage; that must be done with a global definition

- First File: main.c:

```
#include <stdio.h>
int count ;
extern void write_extern();

main()
{
   count = 5;
   write_extern();
}
```

- Second File: support.c:

```
#include <stdio.h>
extern int count;

void write_extern(void)
{
   printf("count is %d\n", count);
}
```

- The compiler command then:

```
$gcc main.c support.c
```

- Will produce the result when running a.out:

```
count is 5
```

- From: <http://www.tutorialspoint.com/cprogramming/c_storage_classes.htm>
- From <http://icecube.wisc.edu/~dglo/c_class/vstorage.html>

Variable Initialization

- auto, register, and static variables may be initialized at creation:

```
int main(void)
  {
  int a = 0;
  register int start = 1234;
  static float pi = 3.141593;
  }
```

- Any global and static variables which have not been explicitly initialized by the programmer are set to zero
- If an auto or register variable has not been explicitly initialized, it contains whatever was previously stored in the space that is allocated to it
  - this means that auto and register variables should always be initialized before being used (read)
  - compiler may provide a switch to warn about uninitialized variables
- From <http://icecube.wisc.edu/~dglo/c_class/vstorage.html>

Volatile

- The keyword volatile indicates that a variables value may be affected outside of the context sequential execution of code
- There are two primary situations we consider :
  - When using memory-mapped devices or registers
  - Global variables accessed by multiple tasks or by interrupt service routines

- It prevents the compiler optimizations based on relationships of sequential code

```
int a,b; // Basically does nothing…
int main(){
  while (b != 0) {
    //do nothing
  }
  a = 1;
  return;
}
```

```
int main(){
  DDRA = 1;
  while (PINB!= 0) {
    //do nothing and wait
  }
  PORTA= 1;
  return;
}
```

- `PINB` is defined as `*(volatile uint8_t *)(0x20+0x03)`
- It is substituted with a dereferencing of a pointer to a volatile `uint8_t` located at memory address `(0x20+0x03)`
- `PORTA` and `DDRA` are similar with a different address

- Another example to create a loop that won't disappear after optimization:

```
void delay(void){
  volatile int count;
  for (count=255;count>0;count--) {
  }
return;
}
```

- From <http://icecube.wisc.edu/~dglo/c_class/vstorage.html>

`const`

- `const` is used with a datatype declaration or definition to specify an unchanging value
  - Examples:
```
const int five = 5;
const double pi = 3.141593;
```
- const objects may not be changed
  - The following are illegal:
```
const int five = 5;
const double pi = 3.141593;
pi = 3.2;
five = 6;
```
- From <http://icecube.wisc.edu/~dglo/c_class/vstorage.html>

declaration	How to read right to left	Usage Notes
const int x	x is an integer constant	Can't modify x
int const x	x is a constant integer	
const int * x	x is a pointer to an integer constant	Can modify where x points, but can't dereference it to modify the value it points to
int const * x	x is a pointer to a constant integer	
int * const x	x is a constant pointer to an integer	Can't modify where x points, but can dereference it to modify the value it points to
const int * const x	x is a constant pointer to an integer constant	Can't modify where x points; can't dereference it to modify the value it points to
int const * const x	x is a constant pointer to constant integer	

- Review C++ examples here: http://www.possibility.com/Cpp/const.html

const variables and const pointer variables

Concept	Declarations, definitions, intitializations	Attempted Code	Allowance by a sample compiler	Notes:
Mutable Variable	int x=1;	x=2;	Allowed **ok**	
Const Variable	const int y=1;	y=2	Not allowed **bad**	

(int *) pointing to (const int)	int x=1;
	int * ptrX = &x;
	const int y=1;
		ptrX=&y;
		*ptrX=2;
			Allowed*
			Allowed **bad**
			*Should not be allowed but sometimes is
				C++ and even C supposedly doesn't allow this implicit conversion between from const int * to int * But at least some C compilers allow the modification of y. The the behavior here should be undefined as constants may not even be stored in writeable memory.

	Same as previous	ptrX=(int *) &y; *ptrX=2;	Allowed **bad**	Even with explicit type casting, the behavior is undefined

(const int *) pointing to int	int x=1;
	const int y=1;
	const int * ptrZ = &y;
		ptrZ=&x;	Allowed **ok**	

	Same as previous	ptrZ=&y; (*ptrZ) = 4;	Not allowed **bad**	

(int * const) pointing to int	int x=1;
	const int y=1;
	int * const ptrZ = &x;
		ptrZ=&y;	Not allowed **bad**	

	Same as previous	*ptrZ = 2;	Allowed **ok**	

Const, pointers and functions pass by value and reference

- Passing by reference (using pointers) is required to modify data structures from the caller in the function.
- Passing by reference (using pointers) is better than passing large structures by value to functions even when not modifying contents.
- For safety and clarity of function intent, always use the const keyword when intending to pass by reference for read-only purposes:

```
function AddPoly(poly_t * polySum,
                 const poly_t * polyA,
                 const poly_t * polyB);
function AddPoly(poly_t polySum[],
                 const poly_t polyA[],
                 const poly_t polyB[]);
```

Generic Pointers

- Pointers of type (void *) can be useful to use one pointer to point to different data types at different times
- This is also the type returned by malloc
- Example:

```
void PrintArray(void * ptr, char type, int numElements){
  char * c = (char *) ptr;
  int * i= (int *) ptr;
  float * f= (float *) ptr;

  int j=0;

  switch (type) {
    case 'c': for(;j<numElements; j++) printf("%c\n",*(c+j)); break;
    case 'i': for(;j<numElements; j++) printf("%d\n",*(i+j)); break;
    case 'f': for(;j<numElements; j++) printf("%f\n",*(f+j)); break;
  }
}
```

Function Pointers

- As it turns out functions identifiers are like array identifiers in that they refer to a location in memory where code is stored like an array identifier refers to a location in memory where an array of data is stored.
- We can create pointers to functions as variables and even pass them as arguments to other functions.
- Lets look at some declarations and assignments and then an example function that has a function pointer as an argument:

`int (* ptrFunction)(float,char);`	declares a pointer to a function that returns an int and take parameters of type float and call

- Since * has a lower precedence than the () that is appended to function names, we can't do the equivalent like this:

`int * ptrFunction(float,char);`	same as (int *) ptrFunction (float,char); not same as int (* ptrFunction)(float,char);

```
int function(float,char);                //prototype

int main(){
  int (* ptrFunction)(float,char) = NULL;//declaration and initialization
                                         //  of pointer to NULL
  char c = 1;
  float f = 2;

  ptrFunction = &function;     //actually the same as ptrFunction = function;
  return (*ptrFunction)(f,c) ; //dereference pointer to call function
}

int function(float f,char c){ //definition
  return (int f)/(int c);
}
```

- A good tutorial can be found at: http://www.newty.de/fpt/fpt.html

Function Pointers (2)

- line

```
return (*ptrFunction)(f,c) ; //dereference pointer to
                             //  call function
```

- Can be replaced with

```
return ptrFunction(f,c) ;    //dereference pointer to
                             //  call function
```

- since functions are all pointers anyway, the explicit deferencing is optional
  - Up until now, the prof would have said C is complex and intricate but at least consistent. This is, perhaps, the most inconsistent syntax feature of C, but it is good to not have to write (*function)(f,c) instead of just function(f,c)

- For the same reason, this
  - `ptrFunction = &function;`

- is the same as this
  - `ptrFunction = function;`

Passing Function Pointers Example

- Create a function to find a value in an array satisfying some condition:
- First, define some conditions:

```
_Bool IsOdd(int x)( return x%2);
_Bool IsNegative(int x)( return x<0);
_Bool IsPrime(int x)( r = 0;);
```

- Protype (declaration):

```
int FindIndexOfFirst(const int array[],
                     int length,
                     int (* SomeCheckFunction)(int));
```

- Or

```
int FindIndexOfFirst(const int *, int, int (*)(int));
```

- Definition:

```
int FindIndexOfFirst(const int array[],
                     int length,
                     int (* SomeCheckFunction)(int,int)) { //definition
  int i,savedIndex;

  assert(SomeCheckFunction!= NULL);//check for null pointer and
                                   //exiting is better than seg fault
                                   //NULL defined in <stddef.h> as (void*)0
  savedIndex = -1;
  i==0;
  while (i<length && savedIndex==-1){
    if ( (*SomeCheckFunction)(array[i]) ){
      savedIndex = i;
    }
    i++;
  }

return savedIndex;
}
```

- And, the function call:

```
int a[] = {1,2,3};
i = FindIndexOfFirst(a,3,IsOdd);
```

Function Pointer and Generic Use Case: Qsort

```
void qsort(void *base, size_t nmemb, size_t size,
           int(*compare)(const void *, const void *));
```

- **`base`** is the pointer to the start of the array to be sorted
- **`nmemb`** is the number of elements in the array
- **`size`** is the size of each element in the array
- **`compare`** is a pointer to a function that is provided to compare the array elements which must return an integer less than zero if the first argument is “less than” the second argument, greater than zero if the first argument is “greater than” the second argument and zero if the arguments are equal

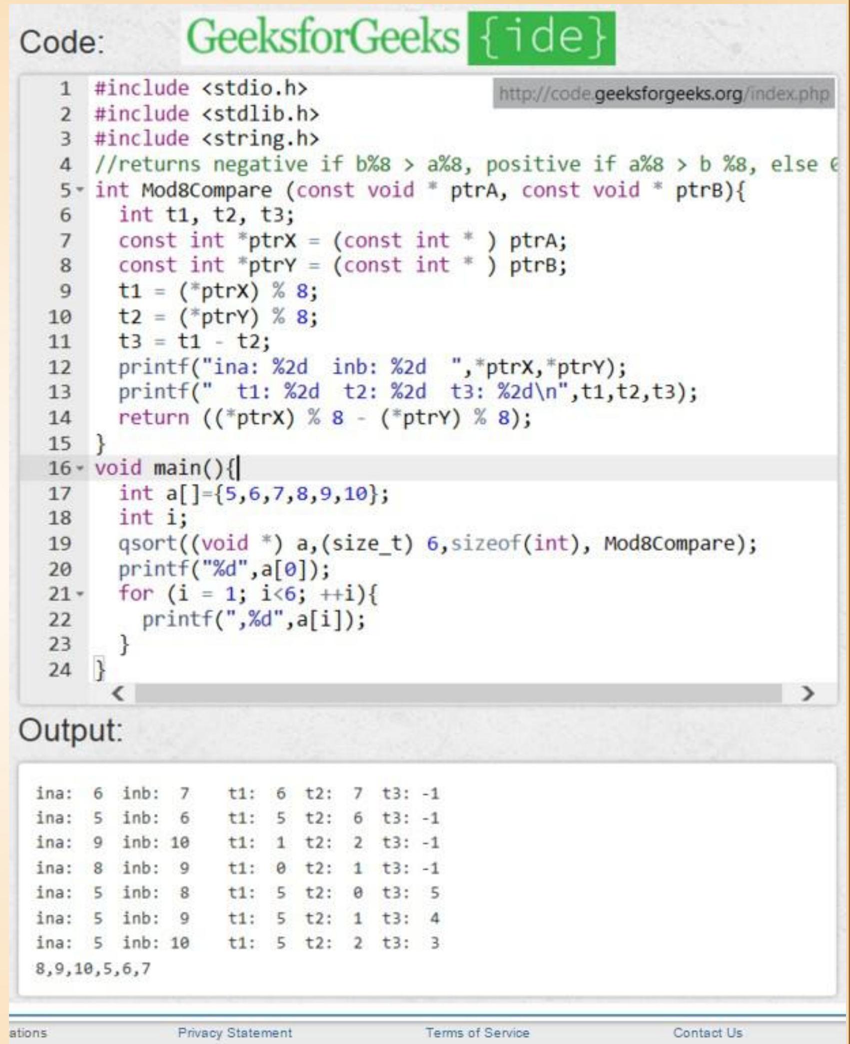
```
-------------------Code Listing -----------------------
   --
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

//returns negative if b%8 > a%8,
//        positive if a%8 > b %8, else 0
int Mod8Compare (const void * ptrA, const void * ptrB){
  const int *ptrX = (const int * ) ptrA;
  const int *ptrY = (const int * ) ptrB;
  return ((*ptrX)%8 -(*ptrY) % 8);
}

void main(){
  int a[]={5,6,7,8,9,10};
  int i;
  qsort((void *)a,(size_t)6,sizeof(int),Mod8Compare);

  printf("%d",a[0]);
  for (i = 1; i<6; ++i){
    printf(",%d",a[i]);
  }
}

----------------------Run----------------------------
   --
$ ./a.out
8,9,10,5,6,7
```



`_Bool`

- `_Bool` variables have the value 1 or 0

- Unlike ints,
  - `_Bool` flag variables may always be safely compared to 1 to test for truth
  - bitwise operations on `_Bool` values can only result 1 or 0
  - Any scalar assignment to a `_Bool` results in 0for assignment to zero or 1otherwise. (`_Bool f=2;` sets `f` to `1`)
  - You may include <stdbool.h>to get the following:

```
#define bool _Bool
#define true 1
#define false 0
```

Restrict

- The class is not responsible for understanding restrict, only to know that it is a keyword.
- **For the curious, continue:**
  - It is intended only to qualify pointers. The compiler will assume that two restricted pointers do not point to the same memory address. It is the coders responsibility to ensure this. The compiler's assumption allows it more aggressively optimize code by not having to account for pointers changing each other's pointee's values.
- Example:

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

int funcR( int * restrict ptr1,
           int * restrict ptr2){
  if (*ptr1 >= *ptr2){ // compiler will fetch the
                       //   first and second value
    *ptr1 = *ptr2-1;   // modify first value, MAY
                       //   be modifying second value
  }
  if (*ptr1 >= *ptr2){ // without restrict assumption
                       //   the compiler must include
                       // conditional code to re-fetch
                       //    second value
    *ptr1 = *ptr1-1;
  }
}

int func( int *ptr1, int * ptr2){
  if (*ptr1 >= *ptr2){
    *ptr1 = *ptr2-1;
  }

  if (*ptr1 > *ptr2){
    *ptr1 = *ptr1-1;
  }
}
```

```
void main(){
  int a=2, b=2, c=2, d=2;

  funcR(&a,&b);
  printf("a=%d,b=%d\n",a,b);

  funcR(&c,&c); //invalid
  printf("c=%d\n",c);

  func(&d,&d);
  printf("d=%d\n",d);
  return;
}
```

-------------------Run:-----------------

```
$ ./a.exe
a=1,b=2
c=0
d=1
```

Note: c and d are different - why?

C99 C Keywords

- Keywords add with C99 (not in C89) are in **`bold blue`**
  - `auto`
  - `break`
  - `case`
  - `char`
  - `const`
  - `continue`
  - `default`
  - `do`
  - `double`
  - `else`
  - `enum`
  - `extern`
  - `float`
  - `for`
  - `goto`
  - `if`
  - **`inline`**
  - `int`
  - `long`
  - `register`
  - **`restrict`**
  - `return`
  - `short`
  - `signed`
  - `sizeof`
  - `static`
  - `struct`
  - `switch`
  - `typedef`
  - `union`
  - `unsigned`
  - `void`
  - `volatile`
  - `while`
  - **`_Bool`**
  - **`_Complex`**
  - **`_Imaginary`**

- You are not responsible for understanding **`restrict`**, **`_Complex`**, **`_Imaginary`**

Materials to Review for Quiz 3:

class: 13, slide: 9 → Syntax of Pointers in C
class: 13, slide: 10 → Pointer Caution
class: 13, slide: 13 → Pointer Examples - especially incrementing
class: 13, slide: 14 → Pointer and Variable types
class: 13, slide: 17-19 → More Pointer Examples
class: 14, slide: 4 → More Pointers & Arrays - understand offsets
class: 14, slide: 9 → ptrAdd.c Example
class: 14, slide: 14-15 → Array of Pointers
class: 15, slide: 3-6 → Basic Structures
class: 15, slide: 9-10 → Arrays of struct
class: 15, slide: 15 → Union vs struct
class: 16, slide: 4 → Pointer in a struct
class: 16, slide: 7 → Dynamic Memory
class: 16, slide: 10 → Testing the returned pointer
class: 16, slide: 11 → assert
class: 16, slide: 12 → free
class: 16, slide: 16 → 2D arrays
class: 17, slide: 2-4 → register, static, extern
class: 17, slide: 7 → chart of const
class: 17, slide: 11-12 → Function Pointers
class: 17, slide: 17 → Keywords