Final C

C Structures, Unions, Example Code

Review

- Introduction to C
- Functions and Macros
- Separate Compilation
- Arrays
- Strings
- Pointers
- Structs and Unions
- Advanced Pointers

Variable Quantifiers - auto

- *auto* is the default for function/block variables
  - auto int a is the same as int a
  - As a default, you will almost never see it used
- Storage is automatically allocated on function/block entry and freed when the function/block is exited
- May not be used with global Variables

Variable Quantifiers - register

- *register* provides a hint to the compiler that you think a variable will be used frequently
- Compiler is free to ignore register hint
- If ignored, variable is equivalent to an auto with the exception that you cannot get the address
  - If it were stored as a register, you can't get its memory address
- Rarely used since any modern compiler will do a better job of optimization than most programmers

Variable Quantifiers - extern

- 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
- Note that the extern keyword is for *declarations*, not *definitions*
  - An extern declaration does not create any storage; must be done with global definition

Variable Quantifiers - static

- Variables declared as static may not be accessed outside of its declaring file even if the variable is global
- Often used when a group of functions need the same variable, but do not want to risk other functions changing their internal variables

Variable Initialization

- *auto*, *register*, and *static* variables may be initialized at creation
- 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 allocated to it
  - Therefore, *auto* and *register* variables should **always** be initialized before they are read
  - Compiler may provide a switch to warn about uninitialized variables

Variable Quantifiers - volatile

- Declaring a variable as volatile indicates that a variables value may be affected outside of the context sequential execution of code
- Two primary scenarios we must consider:
  - When using memory-mapped devices or registers
  - Global variables accessed by multiple tasks or by and interrupt service routine
- Prevents the compiler optimizations based on relationships of sequential code

Volatile Example

```
int main(){
    DDRA = 1;
    while (PINB!= 0) {
        //do nothing and wait
        //volatile PINB forces the compiler
        //not to remove this loop
    }
    PORTA= 1;
    return;
}
```

- PINB is defined as *(volatile uint8_t *)(0x20+0x03)
- 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

Software Delays

- Volatile is often used to create a software delay loop that won't disappear after optimization

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

Variable Quantifiers - const

- *const* is used with a datatype declaration or definition to specify an unchanging value
- *const* objects may not be changed

  - const int five = 5;
  - const double pi = 3.141593;
  - pi = pi + 1; //Illegal

Interpreting const qualifier

Better to read declarations from right to left:

Declaration	Read as:	Usage Notes
const int x int const x	x is an integer constant x is a constant integer	Cannot modify x
const int * x int const * x	X is a pointer to an integer constant X is a pointer to a constant integer	Can modify where x points, but can't dereference it to modify the value it points to
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 int const * const x	X is a constant pointer to an integer constant X is a constant pointer to a constant integer	Can't modify where x points; can't dereference it to modify the value it points to

const variables and const pointers

Concept	Declarations, definitions, intitializations	Attempted Code	Allowance by a sample compiler	
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;
	const int y=1;
	int * ptrX = &x;
		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=&x; (*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	

Slide borrowed from Dr. Ryan Robucci

Pass by value/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 puroposes

Generic Pointers

- Pointers of type (void *) can be useful so that one pointer can point to different data types at different types
- Also the type returned by malloc

```
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

- Function identifiers are like array identifiers in that they refer to a location in memory where code is stored
- Able to create pointers to functions as variables and even pass them as arguments to other functions
  - int (*ptrFunction) (float, char);
  - Declares a pointer to a function that takes a float and char and returns an integer
  - * has lower precedence than ( ), so you cannot do the same with int * ptrFunction(float,char)
    - This returns an integer pointer

Function Pointer Example

```
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);
}
```

Function Pointers

- Since functions are all pointers anyway, the explicit dereferencing is optional
  - return (*ptrFunction)(f,c); can be replaced with
  - return ptrFunction(f,c); in the previous example

- Similarly
  - ptrFunction = &function; can be replaced with
  - ptrFunction = function;

Passing Function Pointers

- Let's create a function to find a value in an array satisfying some condition

- First, lets define the following:
  - _Bool IsOdd(int x)(return x%2);
  - _Bool IsNegative(int x)(return x<0);

Passing Pointer Functions Example

```
Prototype (declaration):
int FindIndexOfFirst(const int array[],int length, int (* SomeCheckFunction)(int));
Definition:
int FindIndexOfFirst(const int array[], int length,  int (* SomeCheckFunction)(int,int)) {
   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;
}
int a[] = {1,2,3};
i = FindIndexOfFirst(a,3,IsOdd);  //Returns 1
```