Finished Pointer

Storage Class
Type Qualifier

```
float table[2][3]={{1.1,1.2,1.3},{2.1,2.2,2.3}};
```

1. what's the meaning of table	&table[0][0]
2. what's the meaning of (table+1)	&table[1][0]
3. what's the meaning of *(table+1)	&table[1][0]
4. what's the meaning of (*(table+1)+1)	&table[1][1]
5. What's the meaning of (*(table)+1)	&table[0][1]
6. what's the value of *(*(table+1)+1)	table[1][1]
7. what's the value of *(*(table)+1)	table[0][1]
8. what's the value of *(*(table+1))	table[1][0]
9. what's the value of *(*(table)+1)+1	table[0][1]+1

A variable's storage class determines when the cell is allocated and how long the cell remains in existence.

- Storage Classes in A Single-Source File:
  auto, extern, static

A **block** – a section of C code bounded by brace { }.
containing block – the smallest block that contains the definition of a variable which is contained in a block.

```
#include <stdio.h>
#include <stdlib.h>
int I = 0;                                  /* not contained in the block */
main( ) {
        char c;                             /* in the block */
        while ( scanf( "%c", &c ) != EOF )
                I++;
        printf( "file length = %d\n", I );
        return EXIT_SUCCESS;
}
```

auto

When we write

int num;

inside the body of a function, the variable **num** receives the default storage class **auto**. It can be written as

auto int num;

or

int num;

- If both the storage class and the data type are given, the storage class must come first.
- An **auto** variable must be defined inside a function's body.
- Storage for an **auto** variable is allocated when control enters the variable's containing block and is released when control leaves its containing block. The term **auto** underscores the fact that storage is allocated and released automatically.
- An auto variable is visible only in its containing block.
- If an auto variable is simultaneously defined and initialized, the initialization is repeated each time storage is allocated.
- If an auto variable is defined but not initialized, the value of the variable is undefined when control enters its containing block.

```
void err_handler( void );
main( ) {
        …
        err_handler( );
        …
}
/* We want to count the number of times the function err_handler is invoked */
void err_handler( void ) {
        int err_code;   /* storage for err_code is allocated when the function
                           err_handler is invoked and released when err_handler returns
                           to the invoking function main( ). */
        int count = 0;  /* dubious, because the initialization is repeated each time
                           the storage is allocated. */
        ++count;                    …
}
```

extern

extern - the default storage class for a variable defined outside a function's body.

- Must be defined outside all function bodies.
- Storage for an **extern** variable is allocated for the life time of the program.
- If an **extern** variable is simultaneously defined and initialized, it is initialized only once– when storage is allocated.
- If an extern variable is defined but not initialized, the system initializes it to zero once
- An extern visible to all functions that ***follow*** its definition

```
#include <stdio.h>
#include <stdlib.h>
int table[80];      /* extern variable Initializes each of table's 80 cells to zero.*/
main( ) {
        …
        err_handler( );
        …
}
char err_flag;          /* Initialize err_flag to 0 extern variable */
char read_table ( void ) {
        …
}
char verify_table ( void ) {
        …
}
```

```
int table[80];

Block 1

char err_flag;

Block 2

Block 3
```

table is visible to all functions in all blocks

err_flag is visible only to blocks 2 and 3

Static

To define the **static** variable i to be of type int, we write

static int i; /*can be inside or outside a function's body*/

- Storage for a **static** variable is allocated for the life of the program – conceptually just before the program as a whole begins executing.
- If a **static** variable is simultaneously defined and initialized, it is initialized only once – when storage is allocated.
- If a **static** variable is defined but not initialized, the system initializes it to zero once.
- A **static** variable that is defined inside a function's body is visible only in its containing block.

```
main( ) {
        static char* ptr = "I am a student.";
        static short list[ 100 ];
}
/* improve previous err_handler function */
void err_handler( void ) {
        int err_code;
```

```
        static int count = 0; /* count is initialized only once. */
        ++count;
        …
}
```

- An **auto** or **static** variable that is defined inside a block is visible only in its containing block.

```
float compute_fed_tax ( float gross, float rate ) {
        float taxes;      /* visible only in this function */
        …
}
float compute_state_tax( float gross, float rate ) {
        float taxes;      /* visible only in this function */
        …
}
```

- A reference to an **auto** or **static** variable defined inside a block that has the same name as an extern variable is resolved in favor of the auto or static variable.

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

```
int I = 0;
void val( void );
main( ) {
        printf( “main’s I = %d\n”, I++ );
        val( );
        printf( “main’s I = %d\n”, I );
        val( );
        return EXIT_SUCCESS;
}
void val( void ) {
        static int I = 100;
        printf( “val’s I = %d\n”, I++ );
}
```

The output is

```
main’s I = 0
val’s I = 100
main’s I = 1
val’s I = 101
```

The Storage Class **register**

- register int i /* store in CPU register*/
  - advantage: faster access
  - limitations:
    - Compiler may ignore it and treats it as an auto
    - Must be in a block
    - Allocated when entering the block and released when leaving
    - Cannot not be addressed by &i

```
main( ) {
        register int i;                    /* use register as variable i's cell. */
        for( i = 0; i < 1000; i++ )
                …
        }
```

Storage Classes in Multiple-Source Files

```
#include <stdio.h>
#include <stdlib.h>
int p( void );
int q( void );
main( ) {
        int k;
        k = p( );
        printf( "%d\n", k );
        k = q( );
        printf( "%d\n", k );
        return EXIT_SUCCESS;
}
int count = 0;
int p( void ) {
        count++;
        return count;
}
```

```
#include <stdio.h>
#include <stdlib.h>
int p( void );
int q( void );
main( ) {
        extern int count;
        int k;
        count - = 10;
        printf( "%d\n", count );
        k = p( );
        printf( "%d\n", k );
        k = q( );
        printf( "%d\n", k );
        return EXIT_SUCCESS;
}
int count = 0;
```

```
int q( void ) {
        count += 5;
        return count;
}
```

The output is
1
6

Two cases:(1) Invoke precedes definition;(2) use in another file. We should define:extern datatype name of varable;

```
int p( void ) {
        count++;
        return count;
}
int q( void ) {
        count += 5;
        return count;
}
```

The output is
-10
-9
-4

A more common reason to declare an **extern** variable is to make it visible in another file.

```
/***** Begin file A *****/
#include <stdio.h>
#include <stdlib.h>
int p( void );
int q( void );
```

```
/***** Begin file B *****/
int q( void ) {
        count += 5; /* illegal */
        return count;}
/***** End of File B *****/
```

```
main( ) {
        int k;
        k = p( );
        printf( "%d\n", k );
        k = q( );
        printf( "%d\n", k );
        return EXIT_SUCCESS;
}
int count = 0;
int p( void ) {
        count++;
        return count;
}
/***** End of File A *****/
```

```
/*****New file B *****/
```

```
/***** Begin file B *****/
int q( void ) {
        extern int count;
        count += 5; /* legal */
        return count;
}
/***** End of file B *****/
```

Summary: the storage class **extern** may be used to allow different functions to access the same variable. Such a variable must be created exactly once by defining the variable outside all blocks. There are two ways to define an **extern** variable. First, the keyword **extern** must be omitted if the **extern** variable is not initialized at definition time. Second, the keyword extern may be either present or absent if the variable is initialized at definition time.（usage just before main function）

```
int no_init_var ;                 /* legal definition */
extern float init_var = 120.0;    /* legal definition */
extern int no_init_var2;          /* illegal definition */
main( ) {
        …
}
```

Recommend that the keyword **extern** be omitted when defining **extern** variables.

Once defined outside all blocks, an **extern** variable can be made visible in a block by declaring the variable in the block. Each declaration must include the keyword **extern** and **cannot be initialized**.

```
main( ) {
        extern int flag;          /* legal */
        extern int I_flag = 0;    /* illegal */
        …
}
int flag = 0;
int I_flag;
```

```
/********** file A ************/
        void clear( void ) {
                dist = 111;             /* illegal */
                …
        }
        external int dist;              /* be visible in all functions that follow the
                                           definition, but not precedes its definition in the
                                           same file*/
        extern int pos;                 /* defined in another file */
        void draw( void ) {
                dist = 222;             /* legal */
                pos = 333;
        }
/******** End of file A *********/
```

Caution: A **static** variable is never visible in more than one file.

Summary: extern and static classes

extern

- omit (extern) outside a block
- can be initialized outside a block
- visible to all functions that follow
- extern int num inside a block makes num visible in that block and across files

static

- located inside a block
- allocated and initialized once
- visible only in containing block
- if static variable has same name as an extern variable, the static variable is favored
- is never visible across files

Nested Blocks

```
#include <stdio.h>
main( ) {                                   /* block 1 */
        int var1 = 10;
        var1 += 20;                         /* var1 = 30 */
        {                                   /* block 2 */
                int var2 = 20;
                var2 += 30;
                var1 += var2;               /* var1 = 30 + 50 = 80 */
                printf( "var2 = %d\n", var2);  /* print out var2 = 50 */
                {                           /* block 3 */
                        int var3 = 30;
                        var3 += 40;         /* var3 = 70 */
                        var2 += var3;       /* var2 = 50 + 70 = 120 */
                        var1 += var2;       /* var1 = 80 + 120 = 200 */
                        printf( "var3 = %d\n", var3);  /* print out var3 = 70 */
                }       /* end of block 3 */
                /* printf( "var3 = %d\n", var3);          illegal: var3 is invisible */
        }               /* end of block 2 */
        printf( "var1 = %d\n", var1);       /* print out: var1 = 200 */
        /*printf( "var2 = %d\n", var2);             illegal: var2 is invisible */
}       /* end of block 1 */
```

Variables with the same name

```
main( ) {                          /* block 1 */
        int var1 = 10;
        var1 += 20;                /* var1 = 30 */
        {                          /* block 2 */
                int var1 = 100;
                var1 += 50;        /* var1 = 150 */

                {                  /* block 3 */
                        int var1 = 500;
                        var1 += 100;   /* var1 = 600 */

                }                  /* end of block 3 */
        }                          /* end of block 2 */
        var1+ = 100;               /* var1 = 130 */
}                                  /* end of block 1 */
```

Variables with the same name

adding extern

```
int var1;
main( ) {                          /* block 1 */
        int var1 = 10;
        var1 += 20;                /* var1 = 30 */
        {                          /* block 2 */
                int var1 = 100;
                var1 += 50;        /* var1 = 150 */

                {                  /* block 3 */
                        int var1 = 500;
                        var1 += 100;   /* var1 = 600 */

                }                  /* end of block 3 */
        }                          /* end of block 2 */
        var1+ = 100;               /* var1 = 130 */
}                                  /* end of block 1 */
```

Results are not changed

Conflict in names: auto class is favored

Storage Classes for Function

Storage classes for function ----- **extern** and **static.** **(extern** is default storage class.)
For **extern**, the syntax is

```
extern int fun1( ) {…}   /* it returns an int, instead of extern int. */
```

Or

```
int fun1( ) {…}          /* usually way */
```

- An **extern** function can be invoked by any other function in any file.

An static function ‘s definition

```
static int fun2( ) { … }
```

It only can be invoked by functions in a same file.

```
void bid( void );
int deal( void );
main ( )
{ … }
void bid( void )
{ … }
static int deal( void )      /* only function main() and bid() may invoke deal() */
{ … }
```

Visible to all files

extern int fun1 (void)

int fun1 (void)

Visible to one file

static int fun2 (void)

Type Qualifiers: **const** And **volatile**

const int x=100;

- x is a constant and its value cannot be changed
- x must be initialized in its definition

- x= 101;
- ++x;
- x--;
- x++;
- --x;

} illegal

const int array[ ] ={1,2,3}

- all elements in the array are constants

Pointers and constants

```
char s[ ] =“ Houston”;
const char* ptr = s;          /*const applies to char*/
    *ptr =‘D’                 /* error */
    ptr = “Boston”            /*ok, ptr is not const */

char s[ ] =“ Houston”;
char* const ptr = s;          /*const applies to ptr/
    *ptr =‘D’                 /*  ok */
    ptr = “Boston”            /* error; ptr is const */
```

Usage and Limitation

- Protects data
  - void f (const int* p) /*protects cells pointed to by p*/
- Do not use const in array size
  - const int n=100;
  - float b[n] /*error */
- Use
  - #define n 100;
  - float b[n];

```
const int pi = 3.141593;        /* ok --- initialization in definition */
pi = 111;                       /* Error: illegal lvalue */
pi++;                           /* Error: illegal lvalue */
pi--;                           /* Error: illegal lvalue */
--pi;                           /* Error: illegal lvalue */
++pi;                           /* Error: illegal lvalue */
```

```
void f( const double cd ) {
        cd = 1.234;             /* Error: illegal lvalue */
        …    }
```

```
const ca[ 3] = { 1, 2, 3 }      /* ok---initialization in definition */
ca[ 0 ]++;                      /* Error: illegal lvalue */
ca[ 1 ] = 8;                    /* Error: illegal lvalue */
--ca[ 0 ];                      /* Error: illegal lvalue */
```

```
char s[ ] = “Happy New Year”;
const char* ptr1 = s;                   /* const char* */
char* const ptr2 = s;                   /* const ptr2 */
*ptr1 = ‘F’;                    /* Error: ptr1 points to is const */
*ptr2 = ‘F’;                    /* ok: *ptr2 is not const */
ptr1 = “E=mc^2”;                /* ok: ptr1 is not const */
ptr2 = “E=mc^2”;                /* Error: illegal lvalue */
```

volatile

volatile -- indicates that a variable's storage cell might be referenced by something besides statements in the program that defines the variable.

```
volatile int wait_flag = 1;      /* in order to signal the programmer's expectation
                                    that something outside the C program eventually
                                    alters wait_flag' s value; something = OS */

void busy_wait(void);
main( ) {
        busy_wait();
        …
}
void busy_wait( void ) {
        while( wait_flag);
}
```

The keyword volatile tells the compiler to re-read the object value each time it is used, even if the program itself has not changed its value since it was last obtained.
This qualifier is commonly used in hardware interface programming to prevent variable values from being updated in a timely manner after being modified by external events.

Type Qualifiers in Combination

volatile const char c = ‘z’

c is const in this program but it may be changed by OS.

Type Qualifiers and Compiler Optimization

Useful in optimizing speed of execution
Memory units: CPU registers, ROM, RAM, disks, tapes