Exercise 1 (size of pointer type): write a program to show size of pointer type like Figure 1,

```
#include <stdio.h>

int main(int argc, char* argv[] )
{
    printf("size of char* = %d\n",   sizeof( char*  ) );
    printf("size of int* = %d\n",    sizeof( int*   ) );
    printf("size of float* = %d\n",  sizeof( float* ) );
    printf("size of double* = %d\n", sizeof( double* ) );
    return 0 ;
}
```

Figure 1: show size of 4 pointer type

(1) execute this program and list processor in your desktop PC and workstation, what's the difference between them.

(2) We say content of a pointer is address of some variable, in 32-bit machine, system use 32-bit (4 bytes) to address a variable, so we expect size of a pointer is 4 (bytes), what's the value of size of pointer type in 64-bit machine?

(3) Search "EM64t" in google, what do you find?

Exercise 2 (fetch address of an array): consider statement "ip = &z[0] ;" in Figure 2, why we interpret it as "ip now points to z[0]", not "ip = (&z)[0]". Hint: use precedence of operator.

```
#include <stdio.h>

int main(int argc, char* argv[] )
{
    int  x = 1 ;
    int  y = 2 ;
    int  z[10] ;  // z is an integer array
    int  *ip ;   // ip is a pointer to int

    ip = &x ;    // ip now points to x

    y = *ip ;    // y is now 1

    *ip = 0 ;    // x is now 0

    ip = &z[0] ; // ip now points to z[0]

    return 0 ;
}
```

Figure 2: reference and dereference operaiton

Exercise 3 (dynamic allocation): in the course we introduce a pair of library functions, malloc and free to do dynamic memory allocation and de-allocation. Write test code like Figure 3, you can modify symbolic constant ALLOC_SIZE to allocate different size of memory block. Note that operating system must return you a contiguous memory block, if OS cannot find one, then it return NULL (no space).

(1) remove casting term (int*) in statement "ip = (int*) malloc(sizeof(int)*ALLOC_SIZE) ;", what's error message when using g++ and icpc

(2) What value of ALLOC_SIZE should you take to exhaust all memory such that function

malloc returns NULL? Report your answer for PC and workstation.

```c
#include <stdio.h>
#include <stdlib.h>  // prototype of malloc and free

#define  ALLOC_SIZE  10

int main(int argc, char* argv[] )
{
    int  x = 1, y = 2 ;
    int  z[10] ;  // z is an integer array, static
    int  *ip ;   // ip is a pointer to int

// dynamic allocate integer array with 10 elements from OS
// and return address of first element to pointer ip
    ip =  (int*) malloc( sizeof(int)*ALLOC_SIZE ) ;
    if ( NULL == ip ){
        printf("Error: allocation fails\n");
    }
    printf("ip(0x%p) = 0x%p\n", &ip, ip);

    free( ip ) ; // release integer array to OS
    return 0 ;
}
```

Figure 3: use library function mallc to do dynamic memory allocation.

(3) try to allocate more memory blocks to exhaust system memory, you can refer source code in Figure 4 or write your own version. Note that when you allocate a memory block, do something for this memory block, for example, random a value or arithmetic operation.
Can you exhaust system memory?

```c
#include <stdio.h>
#include <stdlib.h>  // prototype of malloc and free

#define  ALLOC_SIZE  10

int main(int argc, char* argv[] )
{
    int i ;
    double  *ip[10] ;   // ip is a pointer array

// dynamic allocate integer array with 10 elements from OS
// and return address of first element to pointer ip
    for ( i = 0 ; i < 10 ; i++ ){
        ip[i] =  (double*) malloc( sizeof(double)*ALLOC_SIZE ) ;
        if ( NULL == ip[i] ){
            printf("Error: allocation fails\n");
        }
        printf("ip[%d] = 0x%p\n", i, ip[i] );
// do some computation here

    }

    for ( i = 0 ; i < 10 ; i++ ){
        free( ip[i] ) ; // release integer array to OS
    }

    return 0 ;
}
```

Figure 4: use pointer array to exhaust system memory.

Exercise 4 (pointers and function argument): in the course we introduce call-by-value and use swap as an example to show how to use pointer to avoid copy of function arguments.

(1) write test code in Figure 5 and use Visual Studio debugger to trace change of memory, record configuration of all variables, is the configuration the same as we talk in the course?

(2) Do the same experiment in Linux machine, since we don't have debugger in Linux machine (in fact, GDB is debugger in unix system, but we don't introduce it), we need to report address explicitly like Figure 6. Use g++ and icpc to compile and execute source code in

Figure 6 respectively, record address of all variables and compare configuration between these two compilers.

```c
#include <stdio.h>

void swap(int x, int y) ;

int main(int argc, char* argv[] )
{
    int x = 1 ;
    int y = 5 ;

    printf("Before swap, x = %d, y= %d\n", x, y) ;
    swap( x, y ) ;
    printf("After swap, x = %d, y= %d\n", x, y) ;
    return 0 ;
}

/* this is wrong version */
void swap(int x, int y)
{
    int temp ;

    temp = x ;
    x = y ;
    y = temp ;
}
```

Figure 5: wrong version of swap function

```c
#include <stdio.h>

void swap(int x, int y) ;

int main(int argc, char* argv[] )
{
    int x = 1 ;
    int y = 5 ;
    printf("main: x(0x%p)\n", &x ) ;
    printf("main: y(0x%p)\n", &y ) ;

    printf("Before swap, x = %d, y= %d\n", x, y) ;
    swap( x, y ) ;
    printf("After swap, x = %d, y= %d\n", x, y) ;
    return 0 ;
}

/* this is wrong version */
void swap(int x, int y)
{
    int temp ;
    printf("swap: x(0x%p)\n", &x ) ;
    printf("swap: y(0x%p)\n", &y ) ;
    printf("swap: temp(0x%p)\n", &temp ) ;
    temp = x ;
    x = y ;
    y = temp ;
}
```

Figure 6: report address of all variables explicitly, we need this kind of code in Linux machine.

Exercise 5 (string operation): standard C library provides basic operation for string, in this lecture, we introduce two among them, one is ***strcpy*** (string copy), the other is ***strcmp*** (string comparison).

(1) in Figure 7, we implement ***strcpy*** function void strcpy(char *s, char *t), when pointer ***s*** and ***t*** move in function ***strcpy***, array ***A*** and ***B*** in function main are fixed, why?

(2) In textbook, the author provides two version of string compare (see Figure 8), one is based on array whereas the other is based on pointer. Write a code to test this two versions, explain why they are equivalent.

```c
#include <stdio.h>

// strcpy: copy t to s : pointer version
void strcpy( char *s, char *t )
{
    while ( '\0' != (*s = *t) ){
        s++ ;
        t++ ;
    }
}

int main( int argc, char* argv[] )
{
    char A[] = "now is the time" ; // an array
    char B[] = "This is a book!" ; // an array

    strcpy( B, A ) ; // copy A to B
    printf("array B = %s \n", B ) ;

    return 0 ;
}
```

Figure 7: string copy

```c
/* strcmp : return < 0   if s < t
                  = 0   if s = t
                  > 0   if s > t
  */
int strcmp( char *s, char *t )
{
    int i ;

    for ( i = 0 ; s[i] == t[i] ; i++){
        if ( '\0' == s[i] )
            return 0 ; // s = t
    }
    return s[i] - t[i] ; // s != t, compare character
}
```

```c
/* strcmp : return < 0   if s < t
                  = 0   if s = t
                  > 0   if s > t
  */
int strcmp( char *s, char *t )
{
    for (  ; *s == *t ; s++, t++){
        if ( '\0' == *s )
            return 0 ; // s = t
    }
    return *s - *t ; // s != t, compare character
}
```

Figure 8: string comparison: left panel is array version and right panel is pointer version.

‘

Exercise 6 (command-line argument): write program in Figure 9 and execute it in Linux machine, trace the address of each arguments, is the result the same as we talk in the course?

```c
#include <stdio.h>

int main( int argc, char* argv[] )
{
    printf("argc (%p) = %d\n", &argc, argc );
    printf("argv (%p) = %p\n", &argv, argv );
    while( *argv ){
        printf("argv= %p, *argv(%p) = %s\n", argv,
            *argv, *argv );
        argv++ ;
    }
    return 0 ;
}
```

Figure 9: display command-line argument.

Second, if we restrict size of array ***argv*** as you see in Figure 10, does the program work? Write the program and test it in the Linux machine.

```c
#include <stdio.h>

int main( int argc, char* argv[3] )
{
    int i ;

    printf("argc = %d\n", argc );
    for( i = 0 ; i <= argc ; i++ ){
        printf("argv[%d] = %s\n", i, argv[i] );
    }
    return 0 ;
}
```

Figure 10: restrict size of array ***argv*** as 3

Exercise 7 (diagnosis, assert): experienced C-programmer use a lot of assertions to help them write /debug program, why? If some error activates assertion, then such assertion would print File name and Line number which assertion occurs, this is so important that we can locate the bugs. However if program is correct, then assertions are redundant (they never execute), we can remove them by adding symbolic constant **NDEBUG.** The popular assertion in C-language is macro assert which is defined in ***assert.h***, and you can find description in page 253 of textbook or MSDN library. Here we provide an example (given by MSDN library)

```c
#include <stdio.h>
#include <assert.h>
#include <string.h>

void analyze_string( char *string );   // Prototype

int main( int argc, char *argv[] )
{
   char  test1[] = "abc" ;
   char  *test2 = NULL ; // no string

   printf ( "Analyzing string '%s'\n", test1 );
   analyze_string( test1 );
   printf ( "Analyzing string '%s'\n", test2 );
   analyze_string( test2 );

   return 0 ;
}
// Tests a string to see if it is NULL,
// empty, or longer than 0 characters.
void analyze_string( char * string )
{
   assert( string != NULL );        // Cannot be NULL
   assert( *string != '\0' );       // Cannot be empty
   assert( strlen( string ) > 2 );  // Length must exceed 2
}
```

Figure 11: assertion example

(1) Test code of Figure 11 in your PC and Linux workstation, what's message in the terminal?

(2) Add symbolic constant **NDEBUG** to remove assertion, note that definition of **NDEBUG** must be in front of **#include <assert.h>**, see Figure 12. What's the result in PC and Linux workstation?

(3) If you don't explicit define symbolic constant **NDEBUG** in code of Figure 12, do you have other choice to ***disable*** assert operation? Hint: add macro in compiler's option.

```c
#define NDEBUG   // disable assertion

#include <stdio.h>
#include <assert.h>
#include <string.h>

void analyze_string( char *string );   // Prototype

int main( int argc, char *argv[] )
{
   char  test1[] = "abc" ;
   char  *test2 = NULL ; // no string

   printf ( "Analyzing string '%s'\n", test1 );
   analyze_string( test1 );
   printf ( "Analyzing string '%s'\n", test2 );
   analyze_string( test2 );

   return 0 ;
}
// Tests a string to see if it is NULL,
// empty, or longer than 0 characters.
void analyze_string( char * string )
{
   assert( string != NULL );        // Cannot be NULL
   assert( *string != '\0' );       // Cannot be empty
   assert( strlen( string ) > 2 );  // Length must exceed 2
}
```

Figure 12: use symbolic constant **NDEBUG** to disable assertion

Exercise 8 (quick sort): sorting is basic skill in computer science and we will use it frequently in scientific computation, in standard C library, it provide quick sort (a sorting algorithm) in **stdlib.h**, see page 253 in textbook

void ***qsort***(void ****base***, size_t ***n***, size_t ***size***,
int (****cmp***)(const void*, const void*))

However quick sort uses function pointer ***cmp*** as comparison operator provided by user.

(1) Use qsort to sort integer array, character array, floating point array and string array. You can refer source code in Figure 13 which sorts double array.

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

int  double_comp( double *x, double *y )
{
    return (int)(*x - *y) ;
}

int main( int argc, char* argv[] )
{
    int i ;
    double a[] = { 4.0, 3.0, 1.0, 2.0, 8.0, 7.0 } ;

    qsort( (void*) a, (size_t) 6, sizeof(double),

        (int (*)(const void*, const void*))  &double_comp  ) ;

    for(i = 0 ; 5 >= i ; i++){
        printf("%6.2f  ", a[i]);
    }
    printf("\n") ;

    return 0 ;
}
```

Figure 13: use quick sort to sort double array.

(2) read section 5.11 in textbook, also the author use another comparison operation for string array, the comparison is called ***numcmp***, see Figure 14. Take an example yourself and use this new comparison to do quick sort, what's your result? Can you explain it?

```c
#include <stdlib.h> // prototype of atof

/* numcmp: compare s1 and s2 numerically */
int numcmp( char *s1, char *s2 )
{
    double v1, v2 ;

    v1 = atof(s1) ;
    v2 = atof(s2) ;
    if ( v1 < v2 )
        return -1 ;
    else if ( v1 > v2 )
        return 1 ;
    else
        return 0 ;
}
```

Figure 14: another comparison operator on string, it use numerical value of a string.

Exercise 9 (bubble sort): in lecture note, we take bubble sort as our sorting algorithm, and we provide final version, which has the same function prototype as quick sort in standard C library.

(1) implement bubble sort we talk about in the course (see Figure 15) and use example you built in **Exercise 8** to test it.

(2) Compare performance between quick sort and bubble sort, which one is better? Note that in order to compare both algorithms, you need a large array as example.

```c
#include <stddef.h>
void swap( void  *x, void  *y , size_t size ) ;

// sort base[0], ... base[n-1], total n elements
void  bubble_sort( void* base, size_t n, size_t size,
                int (*comp)(void*, void*)   )
{
    int k, j ;
    char*  a = (char*) base ;
    char*  aj ;       // &a[j]
    char*  aj_plus1 ; // &a[j+1]

    for ( k = n-1 ; 0 < k ; k-- ){
        for ( j = 0 ; j < k ; j++ ){
            aj       = a + size*j  ;
            aj_plus1 = aj + size ;
           if ( (*comp)( aj,  aj_plus1 )  > 0 ){
               swap( (void*) aj, (void*) aj_plus1,
                     size ) ;
           }
        }// for j
    }// for k
}
```

```c
void swap(  void *x,  void *y, size_t size  )
{
    size_t  i ;
    char temp ;
    char*  px = (char*) x ;
    char*  py = (char*) y ;

    for ( i = 0 ; i < size ; i++){
        temp = *px ;
        *px = *py ;
        *py = temp ;
        px++ ;
        py++ ;
    }
}
```

Figure 15: bubble sort algorithm.

Exercise 10 (extract sign, exponent and fraction of a floating point): in the course we introduce how to use pointer to extract each field in a single precision floating point, the source code is in Figure 16

(1) interpret the value of sign, exponent and fraction after shift operation

(2) do you have other approach to find out sign, exponent and fraction of a single precision floating point number?

(3) Refer to source code in Figure 16, write a code to extract each field in a ***double*** precision floating point

```c
#include <stdio.h>
int main( int argc, char* argv[] )
{
    float a = 0.15625 ;
    int  sign = 0 ;
    int  exponent = 0 ;
    int  mantissa = 0 ;
    int  *aInt_ptr = (int*) &a ;

    printf("a (double ) = %25.16f\n", a) ;

    sign     =  *aInt_ptr & 0x80000000   ;
    exponent =  *aInt_ptr & 0x7F800000   ;
    mantissa =  *aInt_ptr & 0x0007FFFFF  ;

    printf("\t\t before shift\n") ;
    printf("sign     = %x\n", sign );
    printf("exponent = %x\n",  exponent );
    printf("mantissa = %x\n",  mantissa );

    sign     = ( *aInt_ptr & 0x80000000 ) >> 31 ;
    exponent = ( *aInt_ptr & 0x7F800000 ) >> 23 ;
    mantissa =   *aInt_ptr & 0x0007FFFFF  ;

    printf("\t\t after shift\n") ;
    printf("sign     = %x\n", sign );
    printf("exponent = %x\n", exponent );
    printf("mantissa = %x\n", mantissa );

    return 0 ;
}
```

Figure 16: extract sign, exponent and fraction fields of single precision floating point $a = 0.15625$