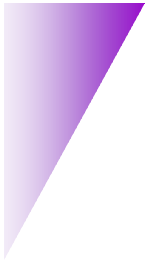
# Advanced Pointer Topics



# Pointers to Pointers

- A pointer variable **is a variable** that takes some memory address as its value. Therefore, you can have another pointer pointing to it.

```
int x;
int * px;
int ** ppx;
ppx = &px;
px = &x;        /* i.e. *ppx = &x */
**ppx = 10;  /*  i.e. *px =10;  i.e. x=10; */
ppx = (int **) malloc(sizeof(int *));
**ppx = 20;  /* Wrong, since *ppx is uninitialized! */
```

# Arrays of Pointers (1)

- If we have an array of structures, each structure can potentially be very big.
- To sort such an array, a lot of memory copying and movements are necessary, which can be expensive.
- For efficiency, we can use array of pointers instead:

```
struct book{
    float price;
    char abstract[5000];
};
struct book book_ary[1000];
struct book * pbook_ary[1000];
……
for(i=0;i<1000;i++)
    pbook_ary[i] = &book_ary[i];
```

# Arrays of Pointers (2)

```
void my_sort(struct book * pbook_ary[ ], int size)
{
    int i, j;
    struct book *p;
    for(i=1;i<size;i++){
      p=pbook_ary[i];
      for(j=i-1;j>=0;j--)
        if(pbook_ary[ j ] -> price > p -> price)
            pbook_ary[ j+1 ]= pbook_ary[ j ];
        else
            break;
      pbook_ary[ j+1 ] = p;
   }
}
```

# Arrays of Pointers (3)

```
struct book ** search_range(struct book * pbook_ary[ ],
    int size, float low, float high, int *num)
{
    int i, j;
    for(i=0;i<size;i++)
            if(pbook_ary[i] -> price >= low) break;
    for( j=size; j>0;j--)
            if(pbook_ary[ j] -> price <= high) break;
    /* i , i+1, …, j  are the elements in the range */
    *num = j – i + 1;
    return &pbook_ary[ i ];
}
```

# Dynamic Two Dimensional Arrays

```
int ** ary;
int m, n;
srand( time(NULL) );
m = rand( ) % 5000 +10;
ary = (int **) malloc( m * sizeof(int *) );
for( j =0; j< m; j++){
   ary[ j ]= (int *) malloc ( (j+1) *sizeof(int));
}
ary[3][4] = 6;
*( *( ary + 3) + 4) = 6;
ary->[3]->[4] = 6;   /* NO! You can not do this */
```

# const Pointers (1)

- The const keyword has a different meaning when applied to pointers.

```
void test( const int k, const int * m)
{
   k ++;      /* 1 */
   (*m) ++;  /*  2 */
   m ++;      /*  3 */
   printf("%d,%d", k, *m);
}
```

- The compiler will warn you about the 1st and 2nd increments, but not the 3rd .

# const Pointers (2)

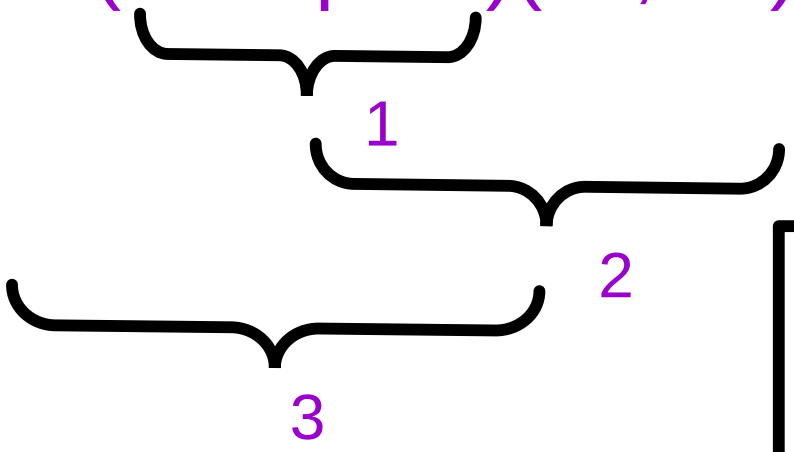
- The reason we use const before parameters is to indicate that we will not modify the value of the corresponding parameter inside the function.

- For example:  we would not worry about the format_str is going to be modified by printf when we look at its prototype:
  - int printf(const char * format_str, ……);

# Pointers to Functions (1)

- Since a pointer merely contains an address, it can point to anything.
- A function also has an address -- it must be loaded in to memory somewhere to be executed.
- So, we can also point a pointer to a function.

```
int (*compare)(int, int);
```




1. Compare is a pointer
2. To a function
3. That returns an int value

# Pointers to Functions (2)

```
typedef struct{
   float price;
   char title[100];
} book;
int (*ptr_comp)(const book *, const book *);
/* compare with
   int * ptr_comp(const book *, const book *);
*/
```

- Do not forget to initialize the pointer -- point the pointer to a real function!

# Pointers to Functions (3)

```
#include <string.h>
int compare_price(const
    book * p, const book *q)
{
    return p->price-q->price;
}
int compare_title(const
    book * p, const book *q)
{
    return strcmp(p->title,q->
    title);
}
```

```
int main( ){
    book a, b;
    a.price=19.99;
    strcpy(a.title, "unix");
    b.price=20.00;
    strcpy(b.title, "c");
    ptr_comp = compare_price;
    printf("%d", ptr_comp(&a,
    &b));
    ptr_comp = compare_title;
    printf("%d", ptr_comp(&a,
    &b));
    return 0;
}
```

# Example: The qsort() Function (1)

- Often, you want to sort something using the quick sort algorithm. C provides a qsort() function in its standard library. Here is the prototype:

```
SYNOPSIS
#include <stdlib.h>
void  qsort(void  *base,  size_t  nel,  size_t  width,
      int (*compar)(const void *, const void *) );
```

- The base argument points to the element at the base of the array to sort.
- The nel argument is the number of elements in the table. The width argument specifies the size of each element in bytes.
- The compar argument is a pointer to the comparison function, which is called with two arguments that point to the elements being compared.

# Example: The qsort() Function (2)

- An example:

```
#include <stdlib.h>

……
{
    book my_books[1000];
    ……
    qsort(my_books, 1000, sizeof(book), compare_price);
    ……
    qsort(my_books, 1000, sizeof(book), compare_title);
    ……
}
```

# Deallocating Dynamic Structures

- For every call to malloc used to build a dynamically allocated structure, there should be a corresponding call to free.
- A table inside malloc and free keeps track of the starting addresses of all allocated blocks from the heap, along with their sizes.
- When an address is passed to free, it is looked up in the table, and the correct amount of space is deallocated.
- You cannot deallocate just part of a string or any other allocated block!

# Example

```
#include <stdio.h>


int main(){
  char *p = malloc(100);
  free(p+1);
  printf("Finsished!\n");
  return 0;
}
```