

Lecture 8

- Review
- Pointers
 - Void pointers
 - Function pointers
- Hash table

Review:Pointers

- pointers: `int x; int *p=&x;`
- pointers to pointer: `int x; int *p=&x;int **pp=&p;`
- Array of pointers: `char *names[]={ "abba","u2" };`
- Multidimensional arrays: `int x [20][20];`

Review: Stacks

- LIFO: last in first out data structure.
- items are inserted and removed from the same end.
- operations: `push()`, `pop()`, `top()`
- can be implemented using arrays, linked list

Review: Queues

- FIFO: first in first out
- items are inserted at the rear and removed from the front.
- operations: `queue()`, `dequeue()`
- can be implemented using arrays, linked list

Review: Expressions

- Infix: $(A+B) * (C-D)$
- prefix: $*+AB-CD$
- postfix: $AB+CD-*$

6.087 Lecture 8 – January 21, 2010

- Review
- Pointers
 - Void pointers
 - Function pointers
- Hash table

Void pointers

- C does not allow us to declare and use void **variables**.
- void can be used only as return type or parameter of a function.
- C allows void **pointers**
- Question: What are some scenarios where you want to pass void pointers?
- void pointers can be used to point to any data type
 - **int** x; **void***p=&x; /*points to int */
 - **float** f; **void***p=&f; /*points to float */
- void pointers cannot be dereferenced. The pointers should always be cast before dereferencing.

```
void*p; printf ("%d", *p); /*invalid */
```

```
void*p; int *px=(int*)p; printf ("%d", *px); /*valid */
```

Function pointers

- In some programming languages, functions are first class variables (can be passed to functions, returned from functions etc.).
- In C, function itself is not a variable. But it is possible to declare pointer to functions.
- Question: What are some scenarios where you want to pass pointers to functions?
- Declaration examples:
 - `int (*p)(int)` /*notice the () */
 - `int (*p)(void*,void*)`
- Function pointers can be assigned, pass to and from functions, placed in arrays etc.

Callbacks

Definition: Callback is a piece of executable code passed to functions. In C, callbacks are implemented by passing function pointers.

Example:

```
void qsort(void*arr, int num,int size, int (*p)(void*pa,void*pb))
```

- `qsort()` function from the standard library can be sort an array of any datatype.
- Question: How does it do that? callbacks.
- `qsort()` calls a function whenever a comparison needs to be done.
- The function takes two arguments and returns ($<0,0,>0$) depending on the relative order of the two items.

Callback (cont.)

```
int arr []={10,9,8,1,2,3,5};
/*callback */
int asc ( void * pa , void * pb )
{
    return ( * ( int *) pa - * ( int *) pb );
}
/*callback */
int desc ( void * pa , void * pb )
{
    return ( * ( int *) pb - * ( int *) pa );
}
/*sort in ascending order */
qsort ( arr , sizeof ( arr ) / sizeof ( int ) , sizeof ( int ) , asc );
/*sort in descending order */
qsort ( arr , sizeof ( arr ) / sizeof ( int ) , sizeof ( int ) , desc );
```

Callback (cont.)

Consider a linked list with nodes defined as follows:

```
struct node {  
    int data ;  
    struct node * next ;  
};
```

Also consider the function 'apply' defined as follows:

```
void apply ( struct node * phead ,  
            void (*fp)( void *, void * ),  
            void * arg) /*only fp has to be named */  
{  
    struct node * p=phead ;  
    while ( p !=NULL)  
    {  
        fp(p,arg); /*can also use (*fp)(p,arg) */  
        p=p->next ;  
    }  
}
```

Callback (cont.)

Iterating:

```
struct node * phead ;  
/*populate somewhere */  
void print ( void * p, void * arg )  
{  
    struct node * np =( struct node *)p;  
    printf ( "%d ",np ->data );  
}  
apply(phead,print ,NULL);
```

Callback (cont.)

Counting nodes:

```
void dototal ( void * p, void * arg )
{
    struct node * np=( struct node *)p;
    int * ptotal      =( int *)arg ;
    *ptotal  += np ->data ;
}
int total =0;
apply(phead,dototal ,&total );
```

Array of function pointers

Example: Consider the case where different functions are called based on a value.

```
enum TYPE{SQUARE,RECT,CIRCILE ,POLYGON};
struct  shape {
    float  params[MAX] ;
    enum TYPE type ;
};
void draw ( struct  shape * ps )
{
    switch ( ps->type )
    {
        case SQUARE:
            draw_square(ps );    break ;
        case RECT:
            draw_rect (ps );    break ;
        ...
    }
}
```

Array of function pointers

The same can be done using an array of function pointers instead.

```
void (*fp [4])( struct shape * ps)=
{&draw_square,&draw_rec,&draw_circle ,&draw_poly };
typedef void (*fp )( struct shape * ps) drawfn;
drawfn fp[4] =
{&draw_square,&draw_rec,&draw_circle ,&draw_poly };
void draw ( struct shape * ps )
{
    (*fp [ps ->type])(ps);    /* call the correct function */
}
```

6.087 Lecture 8 – January 21, 2010

- Review
- Pointers
 - Void pointers
 - Function pointers
- Hash table

Hash table

Hash tables (hashmaps) combine linked list and arrays to provide an *efficient* data structure for storing dynamic data. Hash tables are commonly implemented as an array of linked lists (hash tables with chaining).

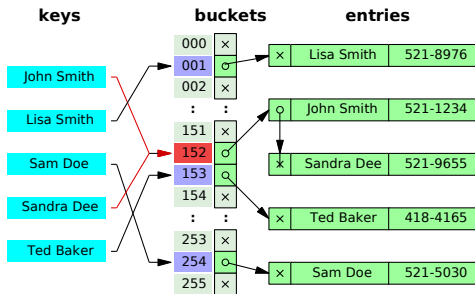


Figure: Example of a hash table with chaining (source: wikipedia)

Hash table

- Each data item is associated with a *key* that determines its location.
- *Hash functions* are used to generate an evenly distributed hash value.
- A *hash collision* is said to occur when two items have the same hash value.
- Items with the same hash keys are chained
- Retrieving an item is $O(1)$ operation.

Hash tables

Hash functions:

- A hash function maps its input into a finite range: hash value, hash code.
- The hash value should ideally have uniform distribution. why?
- Other uses of hash functions: cryptography, caches (computers/internet), bloom filters etc.
- Hash function types:
 - Division type
 - Multiplication type
- Other ways to avoid collision: linear probing, double hashing.

Hash table: example

```
#define MAX_BUCKETS 1000
#define MULTIPLIER 31
struct wordrec
{
    char * word ;
    unsigned long count ;
    struct wordrec * next ;
};

/*hash bucket */
struct wordrec * table [MAX_LEN];
```

Hash table: example

```
unsigned long hashstring ( const char * str )
{
    unsigned long hash =0;
    while ( *str )
        {
            hash= hash *MULTIPLIER+ *str ;
            str ++;
        }
    return hash%MAX_BUCKETS;
}
```

Hash table: example

```
struct wordrec * lookup ( const char * str , int create )
{
    struct wordrec * curr =NULL;
    unsigned long hash= hashstring(str );
    struct wordrec * wp= table [hash];
    for (curr =wp;curr!=NULL ;curr =curr ->next )
        /*search */;
notfound :
    if ( create )
        /*add to front */
    return curr ;
}
```

MIT OpenCourseWare
<http://ocw.mit.edu>

6.087 Practical Programming in C

January (IAP) 2010

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.