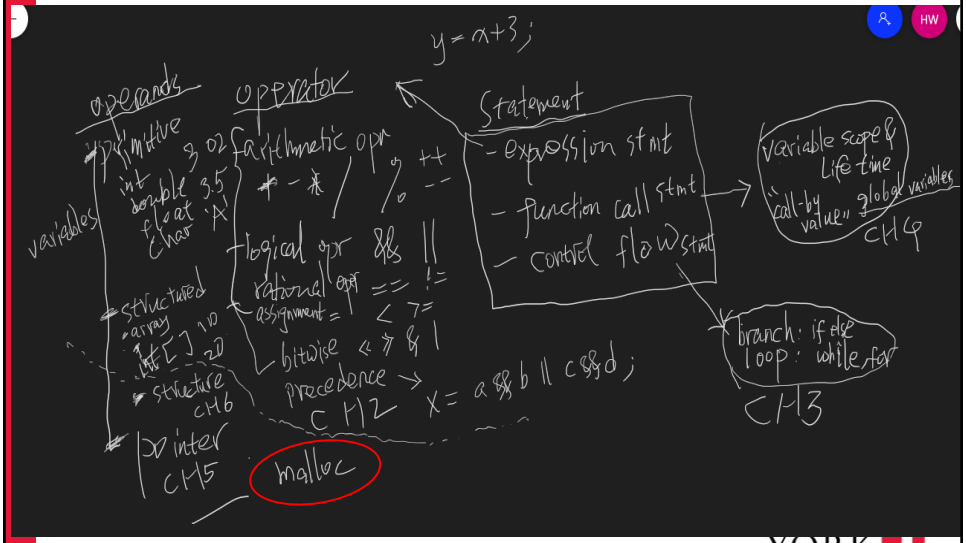
- Pointers (Ch5)
  - Basics: Declaration and assignment (5.1)
  - Pointer to Pointer (5.6)
  - Pointer and functions (pass pointer by value) (5.2)
  - Pointer arithmetic +- ++ -- (5.4)
  - Pointers and arrays (5.3)
    - Stored consecutively
    - Pointer to array elements p + i = &a[i] *(p+i) = a[i]
    - Array name contains address of 1st element a = &a[0]
    - Pointer arithmetic on array (extension) p1-p2 p1<>!= p2
    - Array as function argument – "decay"
    - Pass sub_array
  - Array of pointers (5.6-5.9)
  - Command line arguments (5.10)
  - **Memory allocation (extra)**

Previous lecture

today

- Structures (Ch6)
  - Pointer to structures (6.4)
  - Self-referential structures (extra)



Dynamic memory allocation scenario / motivation 1

- When we define an array, we allocate memory for it

```
int arr[20];
```

sets aside space for 20 ints (80 bytes)

- This space is allocated at compile-time (i.e. when the program is compiled)

```
int arr[20];           20*4 bytes
char arr[20][30];      20*30*1 bytes
int arr[] = {3,5,6};   3*4 bytes
char arr[] = "Hello"   6*1 bytes
```


Dynamic memory allocation scenario / motivation 1

- What if we do not know how large our array should be?
- length is determined at runtime rather than compile time
- In other words, we need to be able to allocate memory at run-time (i.e. while the program is running)

- How?

```
int n;
printf("How many elements in int array? ");
scanf("%d", &n);
int my_array[n];   /* but not allowed in ANSI-C */
```

```
gcc -ansi -pedantic varArray.c
gcc -ansi -pedantic-errors varArray.c
   ISO C90 forbids variable length array 'my_array'
```

- Fortunately, C supports ***dynamic storage allocation*:** the ability to allocate storage during program execution.
- Using dynamic storage allocation, we can design data structures that grow (and shrink) as needed.

- The **<stdlib.h>** header declares three memory allocation functions:
  - **malloc** Allocates a block of memory but doesn't initialize it.
  - **calloc** Allocates a block of memory and clears it.
  - **realloc** Resizes a previously allocated block of memory.
- These functions return a value of type **void *** (a "generic" pointer).
  - function has no idea what type of data to store in the block.

83

Common library functions [Appendix of K+R]

<stdio.h>

```
printf()
scanf()
getchar()
putchar()

sscanf()
sprintf()

gets()  puts()
fgets() fputs()

fprintf()
fscanf()
```

<string.h>

```
strlen(s)
strcpy(s,s)
strcat(s,s)
strcmp(s,s)
strtok(s,s)
```

<math.h>

```
sin() cos()
exp()
log()
pow()
sqrt()
ceil()
floor()
```

<stdlib.h>

```
double atof(s)
int    atoi(s)
long   atol(s)
void    rand()
void   system()
void    exit()
int    abs(int)

void* malloc()
void* calloc()
void* realloc()
void  free()
```

<ctype.h>

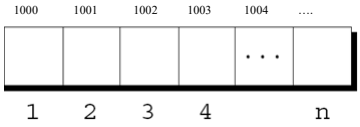
```
int islower(int)
int isupper(int)
int isdigit(int)
int isxdigit(int)
int isalpha(int)

int tolower(int)
int toupper(int)
```

<assert.h>

```
assert()
```

84

malloc()



- **"stdlib.h"** defines:

```
void * malloc (int n);
```

- allocates memory at run-time
- returns a void pointer to the memory that has at least n bytes available (just allocated for you).
  - Address of first byte e.g., 1000
  - Can be casted to any type

Summary of pointer operations

RECALL

- Legal:
  - assignment of pointers of the same type `p2 = p1`

  - adding or subtracting a pointer with an integer `p++ , p+2, p-2`
  - subtracting or comparing two pointers to members of the same array `p2- p1` `if (p1 < p2)` `while (p1 != p2)`
  - assigning or comparing to zero (NULL) (later) `p = NULL` `p==NULL`

- Illegal: `p1+p2; p1*p2; p*3`
  - add two pointers, multiply or divide two pointers, integers
  - add or subtract float or double to pointers `p + 1.23`
  - shift or mask pointer variables `p << 2` `p|3`
  - assign a pointer of one type to a pointer of another type (except for void *) without a cast used in OS course

malloc()

Dangling Pointers


```
#include <stdlib.h>

int main() {
  int *p;  // uninitialized, not point to anywhere

  *p = 52;
  printf("%d\n", *p);
}
```

```
segmentation fault
core dump
```

87

87

Whenever you need to set a pointer's pointee

e.g.,

- `*ptr = var;`
- `scanf("%s", ptr);`
- `strcpy(ptr, "hello");`
- `fgets(ptr, 10, STDIN);`
- ……
- `*ptrArr[2] = var;  // pointer array`

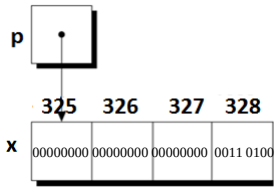
Ask yourself: Have you done one of the following

1. `ptr = &var.  /* direct */`
   `arr[20]; ptr=&arr[0];`
2. `ptr = ptr2  /* indirect, assuming ptr2 is good */`
3. `ptr = (..)malloc(....)  /* now */`

88

88

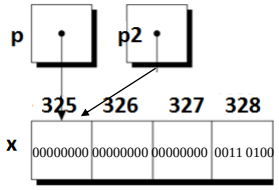
malloc()




```
#include <stdlib.h>

int main() {
  int *p, x;
  p = &x;
  *p = 52;  // x=52
  printf("%d\n", *p);
}
```

fix1

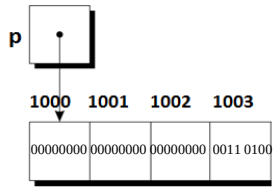
malloc()




```
#include <stdlib.h>

int main() {
  int *p, x;
  int *p2 = &x;  p = p2;
  *p = 52;  // x=52
  printf("%d\n", *p);
}
```

fix2

malloc()



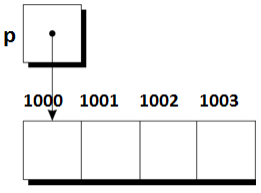

```
#include <stdlib.h>

int main() {
  int *p;
  p = (int *) malloc(4);
  *p = 52;
  printf("%d\n", *p);
}
```

fix3

Improve?

- Note: type conversion (cast) on result of malloc
  `p = malloc(4);` also works. Will convert

Improve1 sizeof



- A better approach to ensure portability

```
int *p;
p = (int *) malloc(4);
```

⬇

```
p = (int *) malloc( sizeof(int) );
*p = 52;
```

Improve 2 NULL

- Allocation not always successful
- malloc() returns **NULL** when it cannot fulfill the request, i.e., memory allocation fails (e.g. no enough space)

```
int *p;
p = (int *)malloc(100000000);// malloc returns NULL
p = (int *)malloc(-10);      // malloc returns NULL
```

93

93

NULL

- <stdlib.h> <stdio.h> <string.h> …defines macro
  NULL a special pointer constant with value 0
- 0 (zero) is never a valid address
- **NULL** == "0 as a pointer" == "points to nothing"
  - `int * p;  // p == NULL?  Not really`
  - `p == 0 ?  // better use NULL   like EOF`

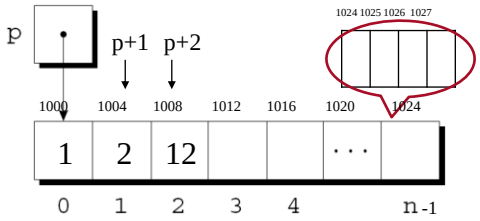
```
p = malloc(10000000);
if (p == NULL) {  // an "exception"
  exit(0) /* allocation failed; take appropriate action *
}
else …
```

```
if ( (p = malloc(10000000)) == NULL) {
  exit(0) /* allocation failed; take appropriate action *
}else ….
```

94

94

malloc()

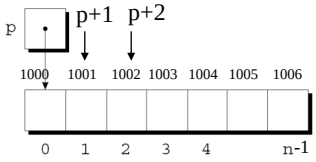



```
#include <stdlib.h>

int main() {
  int n;
  printf("How many elements in int array? ");
  scanf("%d", &n);

  int * p = (int *)malloc(n * sizeof(int));
  if (p == NULL)
     exit(0);

  // else
  *p = 1;          // p[0] = 1     second +1 +4?
  *(p+1) = 2;      // p+1 = 1004   p[1]= 2
  *(p+2) = 12;     // p+2 = 1008   p[2] = 12
}                    pointer arithmetic!!!
```

4n bytes allocated.
n=7 28 bytes 1000~1027 allocated

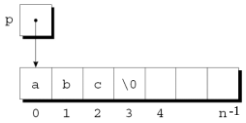
95

malloc()




```
#include <stdlib.h>

int main() {
  int n;
  printf("chars in array: ");
  scanf("%d", &n);

  char * p = (char *)malloc(n * sizeof(char)); //n+1?
  if (p == NULL)
     exit(0);

  strcpy(p, "abc");

  *(p+1) = 'x';

  printf("%s", p); // axc
  printf("%d", strlen(p)); /
} printf("%s", p+1);
```

n bytes allocated. Include for \0
n=7 7 bytes 1000~1006 allocated



96