Advanced Pointers in C

Master Class at Sunbeam Infotech

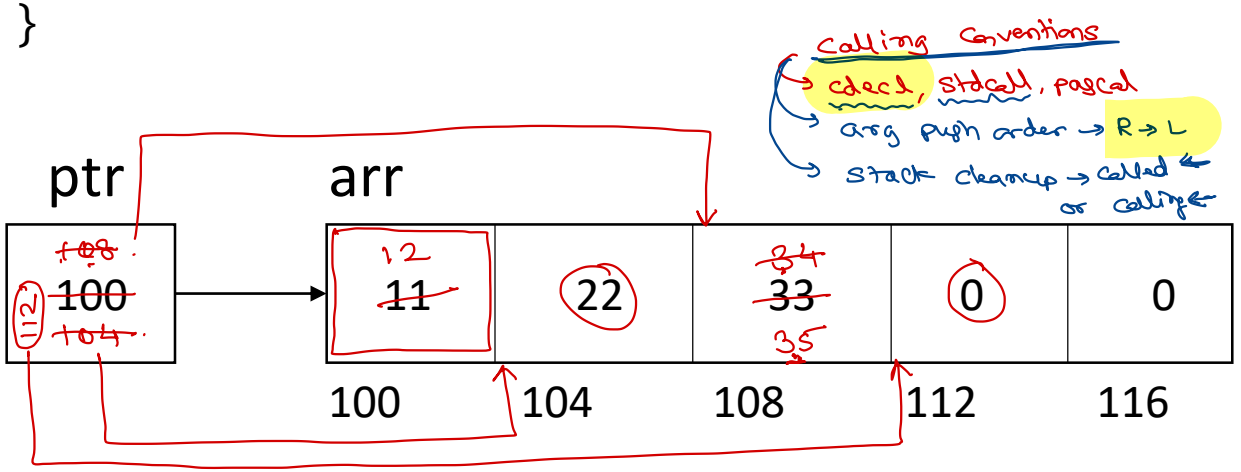
Agenda

- Array of pointers
- Command line arguments
- Function pointers
- Complex pointer declarations
- Using typedef
- Structure pointers
- Dangling pointers
- Near, far & huge pointers

Operators with pointers

Operators	Associativity
() [] -> .	left to right
! ~ ++ -- + - * (type) sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= %= &= ^= |= <<= >>=	right to left
,	left to right

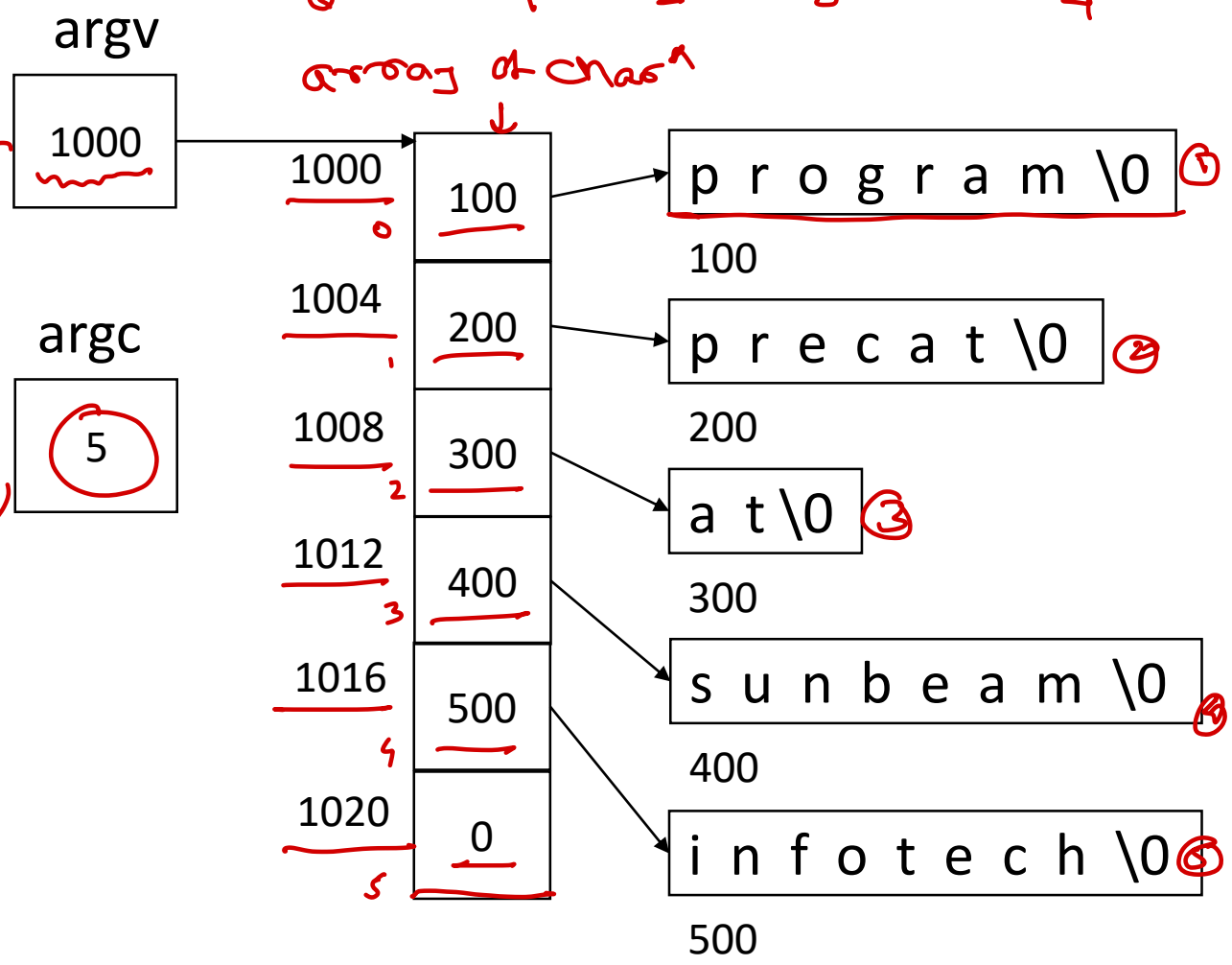
```
int main() {
    int arr[5] = {11, 22, 33};
    int *ptr = arr;
    printf("%d, %d, %d, %d, %d, %d\n",
        *ptr, ++*ptr++, (*ptr)++, *ptr++, *++ptr, ++*ptr);
    return 0;
}
```



Command line arguments

- The additional information passed to the program while running it from command line is called as command line arguments.
- e.g. ls –l –a
- e.g. notepad.exe D:\hello.txt
- Cmdline args are collected in args to main()
  - int main(int argc, char *argv[]);
  - argc = number of args
  - argv = pointer to array of arg pointers
  - argv[0] = name of executable
  - argv[argc] = NULL

- cmd> program precat at sunbeam infotech

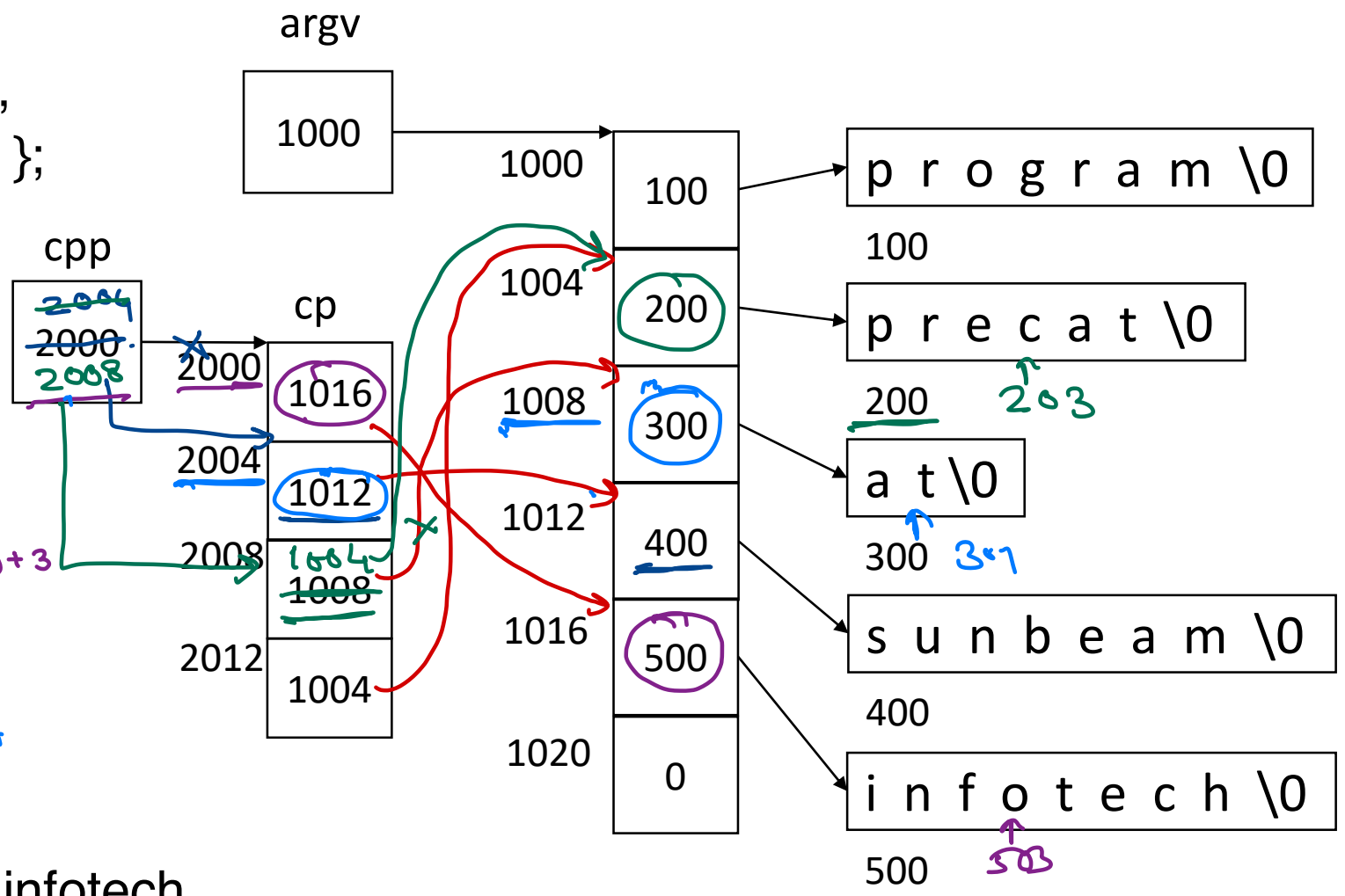


Array of Pointers

```
int main(int argc, char *argv[]) {
    char** cp[] = { argv + 4, argv + 3,
                    argv + 2, argv + 1 };

    char*** cpp = cp;

    printf("%s\n", **++cpp);
    printf("%s\n", *--*(++cpp+3));
    printf("%s\n", *cpp[-2]+3);
    printf("%s\n", cpp[-1][-1]+1);
    return 0;
}
```




```
//run> program precat at sunbeam infotech
```

*(ptr[1]+1)

((ptr+1)+1) = 30

((ptr+1)+0) = 20

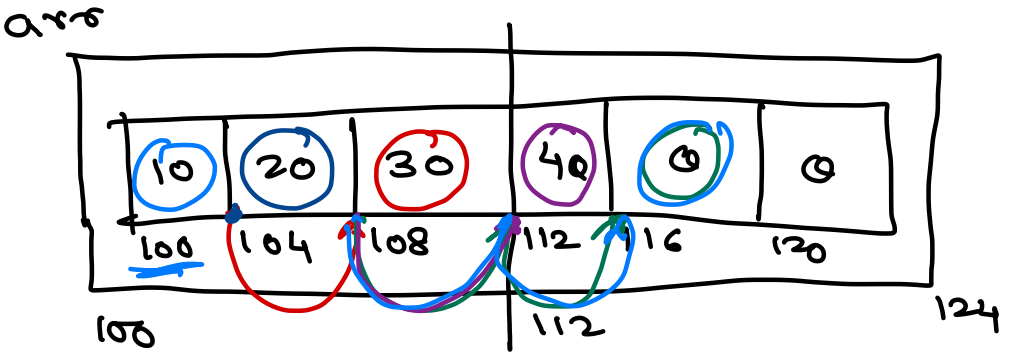
((ptr+0)+2) = 0

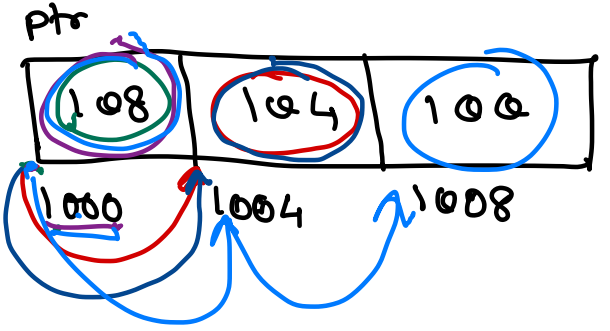
ptr[0][1]

((ptr+0)+1) = 40

~~*(ptr+0)[2]~~

~~*(*(ptr+0)+2) = 0~~





*(ptr+0)[2]

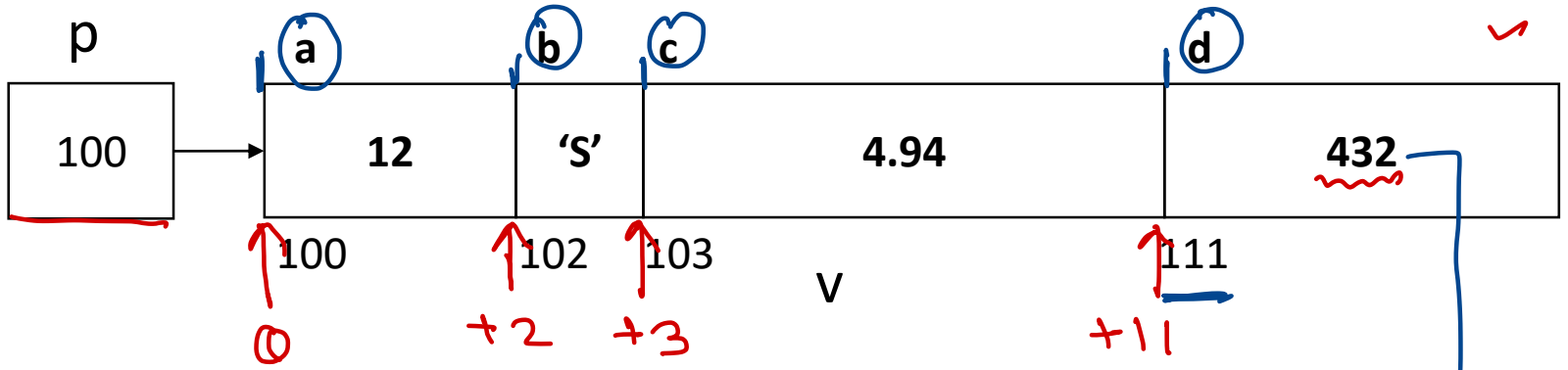
((ptr+0)+2)

Pointer to Structure

→ arr[i] = *(arr + i) #pragma pack(1)

```
int main() {
  struct test {
    short a;
    char b;
    double c;
    int d;
  };
  struct tes v = { 12, 'S', 4.94, 432}, *p;
  p = &v;
  printf("%hd, %hd\n", v.a, p->a);
  printf("%c, %c\n", v.b, p->b);
  printf("%lf, %lf\n", v.c, p->c);
  printf("%d, %d\n", v.d, p->d);
  return 0;
}
```

S.f. = 15



- How dot and arrow operator works?

v.d → value at (&v + 11) →

((int)(((char*) &v) + 11))

p→d → value at (p + 11) →

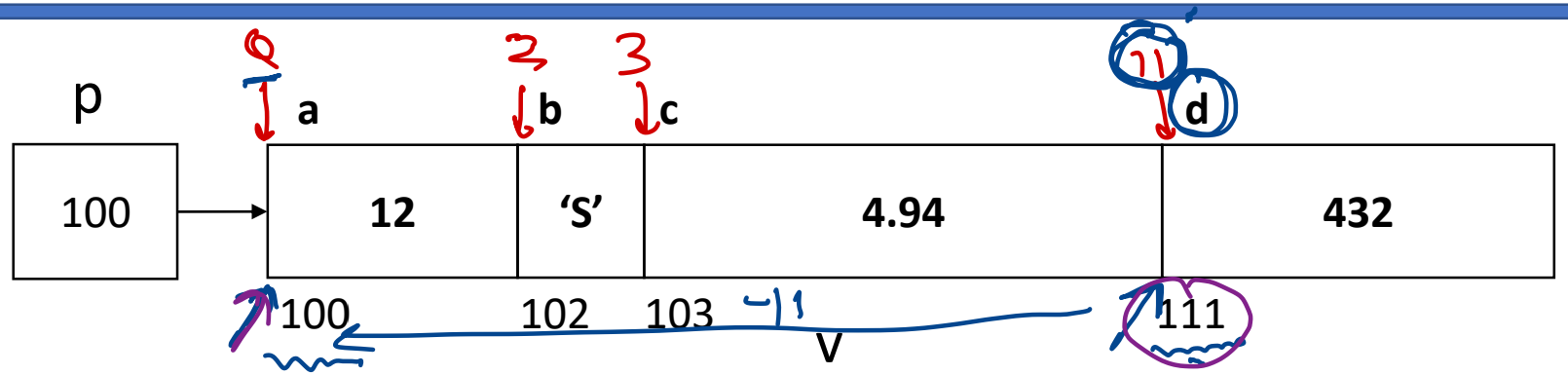
((int)(((char*) p) + 11))

Pointer to Structure

p = NULL;

&p->c

(char*)p + 11



#define offset_of(type, member) ((long) &((type*)0)->member)

struct test v = {-3};

int *d = &v.d;

test *q = (struct test*) ((char*)d - offset_of(struct test, d));

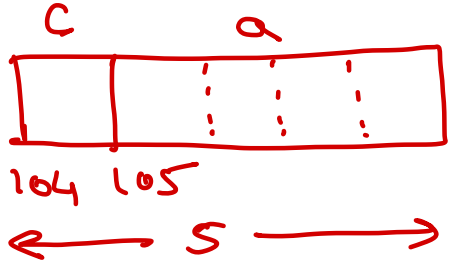
#define container_of(ptr, type, member)

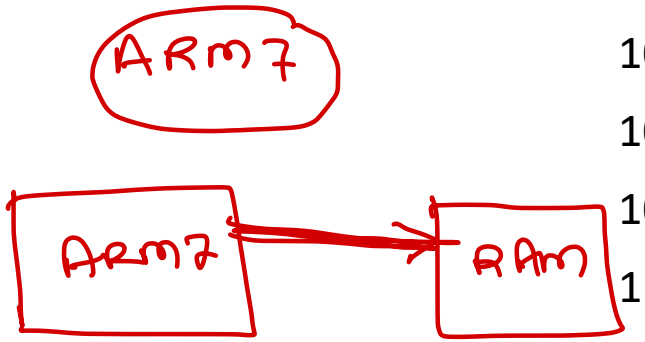
(type*)((char*)ptr - offset_of(type, member))

Slack bytes / Struct padding / Byte alignment.

```
struct test {
    char c;   -1
    int a;    -4
};
```

p = t.a;





100				
104	A	78	56	34
108	12			
112				
116				

x ← char
x2 ← short
x4 ← int

Compiler add slack bytes to make address of each member multiple of its size

```
struct test x = { 'A' , 0x12345678};
```

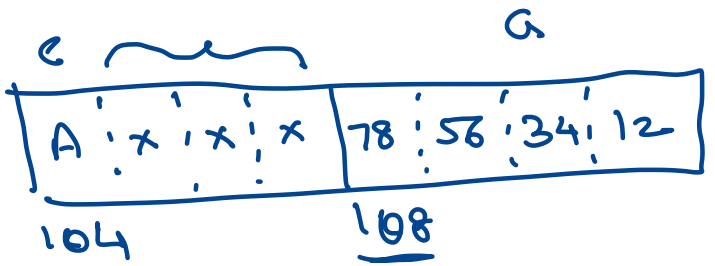
sizeof(x) = 8

pf("%u", &x); → 104

pf("%u", &x.c); → 104

pf("%u", &x.a); → 108 ← +3

p = t.a;



100				
104	A	x	x	x
108	78	56	34	12
112				
116				

const keyword

- const int i = 10;
- *const* keyword informs compiler that value of i cannot be modified.
- Compiler doesn't allow using any operator on i, which may modify the value of the variable (e.g. =, ++, --, +=)

- *const* variables are NOT stored in read-only section.
- They can be modified using pointers.
- int *ptr = &i;
- *ptr = 20;
- printf("%d", i);

Constant pointers

- In order to prevent accidental mistakes and making code more readable we use const pointers.

- int a = 10;
- const int *p = &a;
- int const *p = &a;

- int * const p = &a;

- const int * const p = &a;
- int const * const p = &a;

Complex pointer declarations

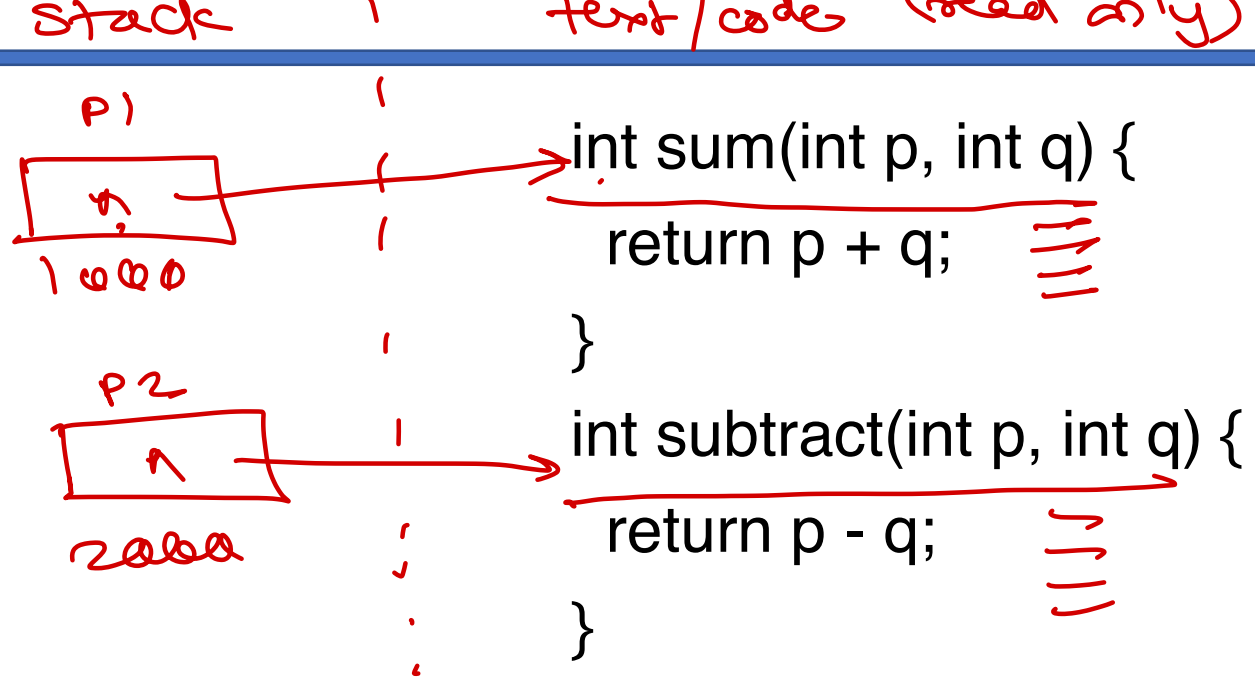
- Declarations should be read starting from the name and then following preceding order.
- Precedence Level1: Grouping parenthesis.
- Precedence Level2: Postfix operators i.e. () indicating function, [] indicating array.
- Precedence Level3: Prefix operator i.e. * indicating pointer.
- *const* or *volatile* next to type, applies to type. In other cases, *const* or *volatile* applies to pointer asterisk before it.

Complex pointer declarations

- int *x[5]; vs int* x[5]; → x is array of 5 int*.
  grouping
- int (*y)[5]; → y is pointer to array of 5 ints.
- int (*z)[2][3]; → z is pointer to array of int 2×3.
- void f(int (*y)[4]); → f is a func that takes pointer to array of 4 int as arg & return void.
- int (* f())[4]; → f is a func that returns pointer to array of 4 int.
- int (* f(int (*)[4]))[4]; →

Pointer to Function

```
int main() {
    int (*p1)(int,int);
    int (*p2)(int,int);
    int res;
    p1 = sum;
    p2 = subtract;
    res = p1(12, 4);
    printf("%d\n", res);
    res = p2(12, 4);
    printf("%d\n", res);
    return 0;
}
```



int (*p1)(int,int);

res = sum(12, 4);
p1

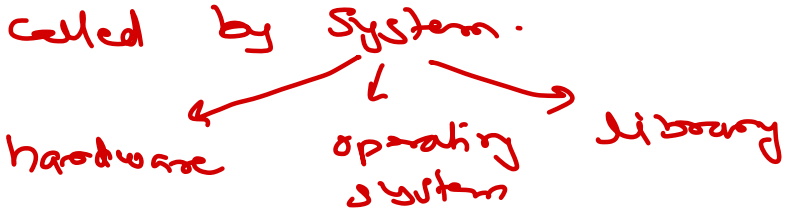
Pointer to Function

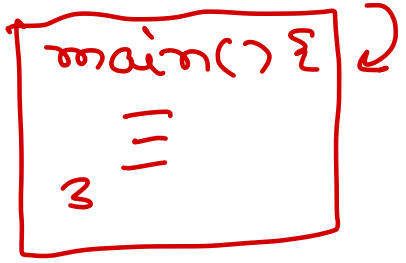
```
int main() {
    int (*arr[2])(int,int);
    int res, i;
    arr[0] = sum;
    arr[1] = subtract;
    for(i=0; i<2; i++) {
        res = arr[i](12, 4);
        printf("%d\n", res);
    }
    return 0;
}
```


```
int sum(int p, int q) {
    return p + q;
}
int subtract(int p, int q) {
    return p - q;
}
```

Applications of Function pointers

- Call-back functions → fn defined by programmer but called by System.
  - Interrupt Service Routines ← CPU / hw
  - Window Procedure ← win32 SDK
  - Thread Functions ← CreateThread(), pthread_create()
  - Signal handlers ← linux - SIGINT, ...
  - Linux device drivers





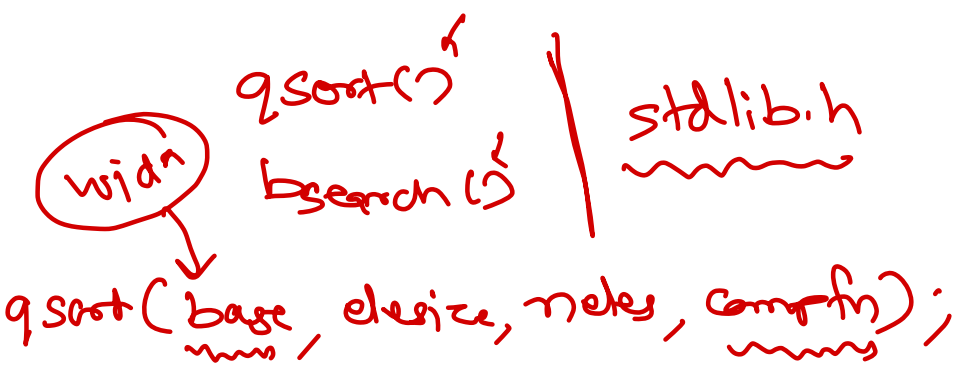

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    ...
};
```

char device driver

28 = 27 + 1

27

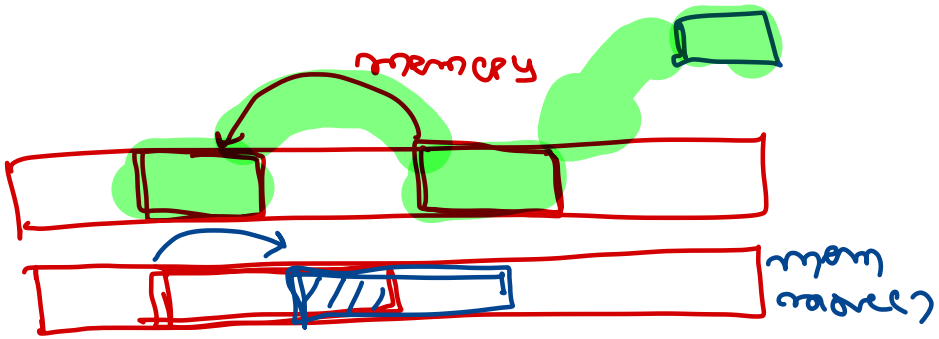
- C++ virtual functions → vtable → fun ptrs.



Void pointer

- Void pointer is generic pointer i.e. it can keep address of any data type.
- It is used for generic programming / algorithm implementation in C language.
- Void pointer do not have scale factor. It is common to cast into char pointer and process data byte by byte.
- Commonly used functions:
  - memset (ptr, value, nbytes);
  - memcpy(dest, src, nbytes);
  - memmove(dest, src, nbytes);

```
void sort(void* arr, int n, int elesize, int (*comp)(const void*,
 const void*)) {
    void* temp = malloc(elesize);
    int i, j;
    char* a = (char*)arr;
    for (i = 0; i < n - 1; i++) {
        for (j = i + 1; j < n; j++) {
            if (comp(a + i * elesize, a + j * elesize) > 0) {
                memcpy(temp, a + i * elesize, elesize);
                memcpy(a + i * elesize, a + j * elesize, elesize);
                memcpy(a + j * elesize, temp, elesize);
            }
        }
    }
    free(temp);
}
```



Pointer to Function

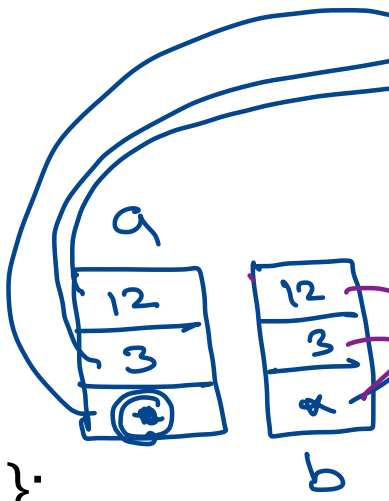
```
struct op {
  int x, y;
  int (*fn)(int,int);
};

int main() {
  struct op a = { 12, 3, sum };
  struct op b = {12, 3, subtract };
  printf("%d\n", calculate(&a));
  printf("%d\n", calculate(&b));
  return 0;
}

int sum(int p, int q) {
  return p + q;
}

int subtract(int p, int q) {
  return p - q;
}

int calculate(struct op *ptr) {
  return ptr->fn(ptr->x, ptr->y);
}
```



closure → python / swift / scala / ...

LDD :→ tasklets, timer

Complex pointer declarations

- void (*p)(int);
- char* (*q)(char*, const char*);
- void* (*q[3])(void*);
- int (* (*f)(int (*)[4]))[4];

Complex function pointers

- char* f(int **);
- char* (*p[10])(int **);

- char (* f(int **))[10];
- char (* (*p)(int **x))[10];

- void (* signal(int, void(*)(int)))(int, void (*)(int));

typedef

- typedef is not a macro to replace type.
  - #define char_ptr_t char*
  - ~~char_ptr_t~~ p1, p2; → char *p1, p2;
  - typedef char* char_ptr_t;
  - char_ptr_t p3, p4;
- typedef simplify the declaration.

declare types for portability.

- int (*ptr)[5];

uint32_t
uint16_t

- char* f(int **); → func takes int** & other char*
- char* (*p[10])(int **); → p is array of fn ptr

typedef char* fun(int**);
fun* ptr;
fun* p[10];

typedef char* (*ptrfun)(int**)
ptrfun ptr;
ptrfun p[10];

- char (* f(int **))[10];
- char (* (*p)(int **x))[10];

typedef char arr[10];
arr *ptr; → char(*ptr)[10];

arr* f(int**);

- ~~void (* signal(int, void(*)(int)))(int, void(*)(int));~~

void (* signal(int, void(*)(int)))(int);

typedef void(*sighandler_t)(int);

man page → sighandler_t signal(int, sighandler_t); → signal sys call

near, far and huge pointers

- Memory models on 8086
  - Tiny
  - Small
  - Compact
  - Medium
  - Large
  - Huge
- Pointers
  - near → 2 bytes → max 64 KB
  - far → 4 bytes → seg 2 + offset 2
    VDU mem: 0xB8000
  - huge → normalized far pointer.
    offset → 15 bits

- Converting far pointer to physical address

- Memory model & pointers

*P1 = 'A' *P2 = 'B'

Model	Code	Data
Tiny/Small	64 KB	
Compact	64 KB	1 MB
Medium	1 MB	64 KB
Large	1 MB	1 MB
Huge	1 MB	> 1 MB

Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>