Warning

CEng 140

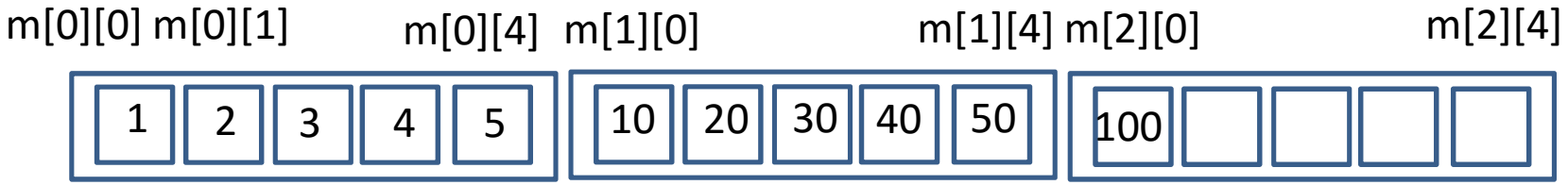
Multi Dimensional Arrays and Pointers

Multi-dimensional Arrays: Storage

- A multi-dim array in C is really a one-dim array [a contiguous area],
  - whose elements are themselves arrays (i.e., **arrays of arrays**)
  - and stored such that the **last subscript varies most rapidly** (i.e., row-order storage)
- Name of the multi-dim array is a **pointer to the first array**!

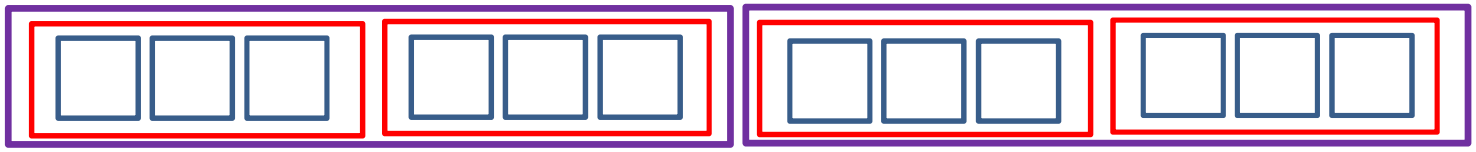
- A multi-dim array in C is really a one-dim array [a contiguous area],
  - whose elements are themselves arrays (i.e., **arrays of arrays**)
  - and stored such that the **last subscript varies most rapidly** (i.e., row-order storage)

```
int m[3][5] = {{1,2,3,4,5},
               {10,20, 30, 40,50},
               {100, 200, 300, 400, 500}};
```



- A multi-dim array in C is really a one-dim array [a contiguous area],
  - whose elements are themselves arrays (i.e., **arrays of arrays**)
  - and stored such that the **last subscript varies most rapidly** (i.e., row-order storage)

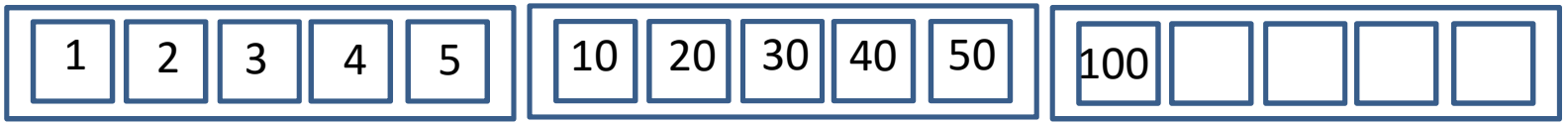
```
int m[2][2][3] = {
        { {1, 2, 3},  {2, 3, 4}},
        { {5, 6, 3},  {7, 8, 4}}  };
```



m[0][0][0] m[0][0][1] m[0][0][2] m[0][1][0] m[0][1][1] m[0][1][2]
m[1][0][0] m[1][0][1] m[1][0][2] m[1][1][0] m[1][1][1] m[1][1][2]

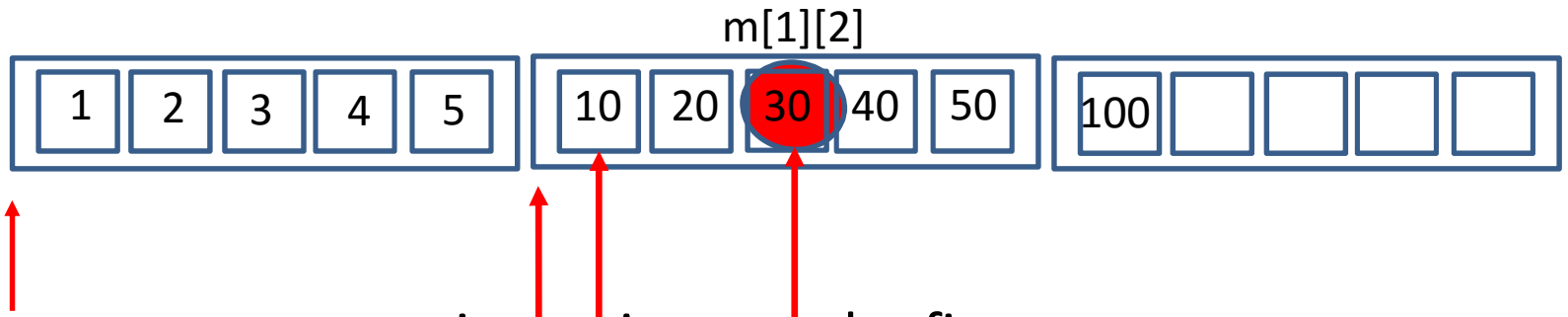
```
int m[3][5] = {{1,2,3,4,5},
               {10,20, 30, 40,50},
               {100, 200, 300, 400, 500}};
```

What does m[i][j] really mean?



m	is a pointer to the first array (i.e., a ptr to array of 5 elements)
m+i	is a pointer to the i-th array
*(m+i)	is the i-th array pointer converted into a pointer to the first element in the i-th array
*(m+i) + j	is a pointer to the j-th element in the i-th array
* (*(m+i) +j)	is m[i][j]

m[1][2]



m is a pointer to the first array
(i.e., a ptr to array of 5 elements)

m+1 is a pointer to the 1-th array

*(m+1) is the 1-th array pointer converted into a
pointer to the first element in the 1-th array

*(m+1) + 2 is a pointer to the 2-th element in the 1-th array

* (*(m+1) +2) is m[1][2]

Imagine what happens for a 3 dim array!

A 2D array

int main()

{ int i, j, scores[3][5] = {1, 2, …,15};

What is the value of scores?

What is the type of scores?

What is the value of scores+1?

What is the type of scores+1?

What is the value of *(scores+1)?

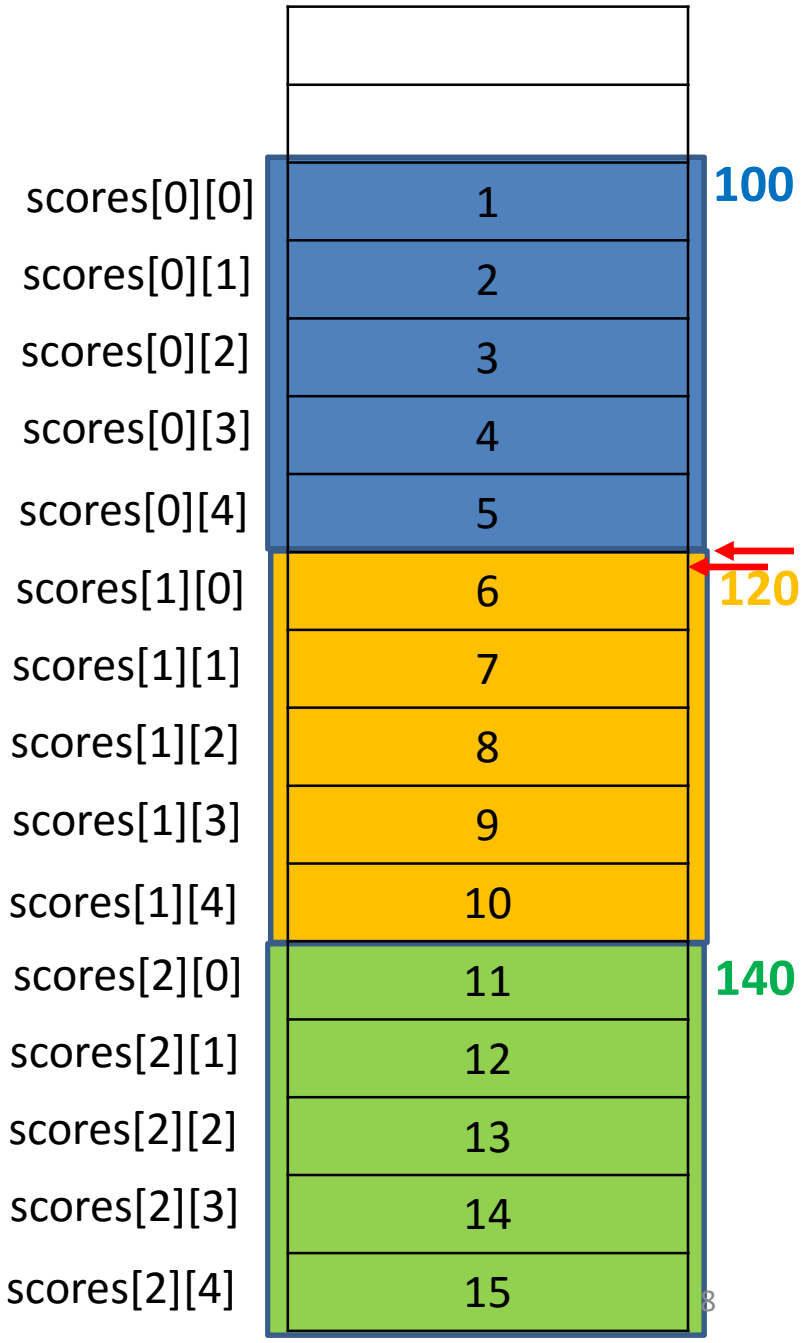
What is the type of *(scores+1)?

What is the value of *(scores+1)+1?

What is the type of *(scores+1)+1?

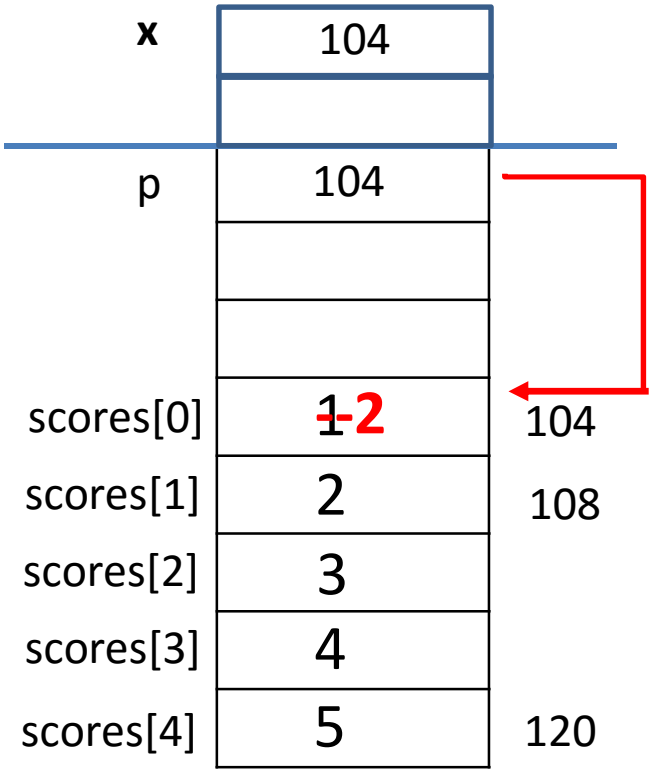
What is the value of *(*(scores+1)+1)?

What is the type of *(*(scores+1)+1)?



Accessing 1D array via a pointer

```
int main()
{ int scores[5] = {1,2,3,4,5};
   int *p, i;
   p=scores;
   for (i=0; i<5; i++)
        *(p+i) *= 2;   }
```



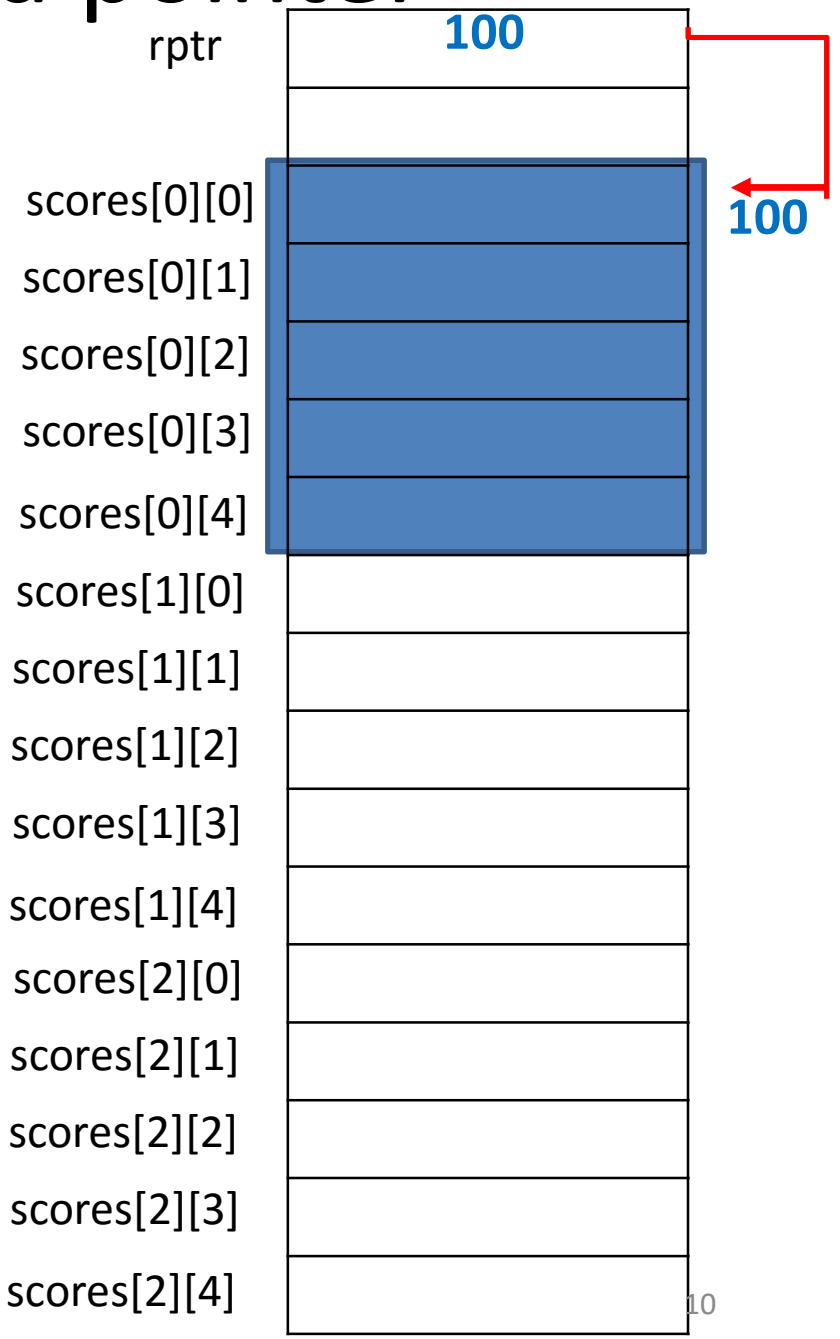

```
            int *x
  void f(int x[],int len)
  { int i;
   for (i=0; i<len; i++)
         *(x+i) *= 2;  /* x[i] *= 2; */ } }
```

CALL (in main here):

f(scores, 5);

Accessing 2D array via a pointer

```
int main()
{ int i, j, scores[3][5] = {…};
  /* Declare a pointer to the
       rows; i.e, array of 5 ints */
  int  (*rptr)[5];
  rptr=scores;
  for (i=0; i<3; i++)
     for (j=0; j<5; j++)
        *(*(rptr+i) + j) *= 2;   }
```



Accessing 2D array via a pointer

```
int main()
{ int i, j, scores[3][5] = {…};
  /* Declare a pointer to the
       rows; i.e, array of 5 ints */
  int  (*rptr)[5];
  rptr=scores;
  for (i=0; i<3; i++)
     for (j=0; j<5; j++)
       *(*(rptr+i) + j) *= 2;   }
```

Parameter is pointer to first array element, i.e., pointer to an array of 5 integers!

```
int (*x)[5]
void f(int x[][5],int nr, int nc)
{ int  (*rptr)[5];
  rptr=x;
  for (i=0; i<nr; i++)
     for (j=0; j<nc; j++)
       *(*(rptr+i) + j) *= 2;}
```

CALL (in main here):

```
f(scores, 3, 5);
```

Mystery solved!

- Recall from the previous weeks:

"When we declare a **multi-dim array** as a **parameter**, we must still specify **all** but the first dimension!"

- This is needed, so that when compiler sees m[i][j], it can compute the pointer arithmetics for **m+i**; i.e., it will go **i** "arrays" ahead from the base address **m**.
- Of course, you should still separately pass as paramaters the length of array for each dimension, to know the array boundaries.

Accessing 2D array via a pointer

```
        int (*x)[5]
void f(int x[][5],int nr, int nc)
{ int  (*rptr)[5];
   rptr=x;
   for (i=0; i<nr; i++)
      for (j=0; j<nc; j++)
         *(*(rptr+i) + j) *= 2; OR
          rptr[i][j] *= 2; OR
         *(rptr[i]+ j) *= 2; OR
         (*(rptr+i))[j] *= 2;
```

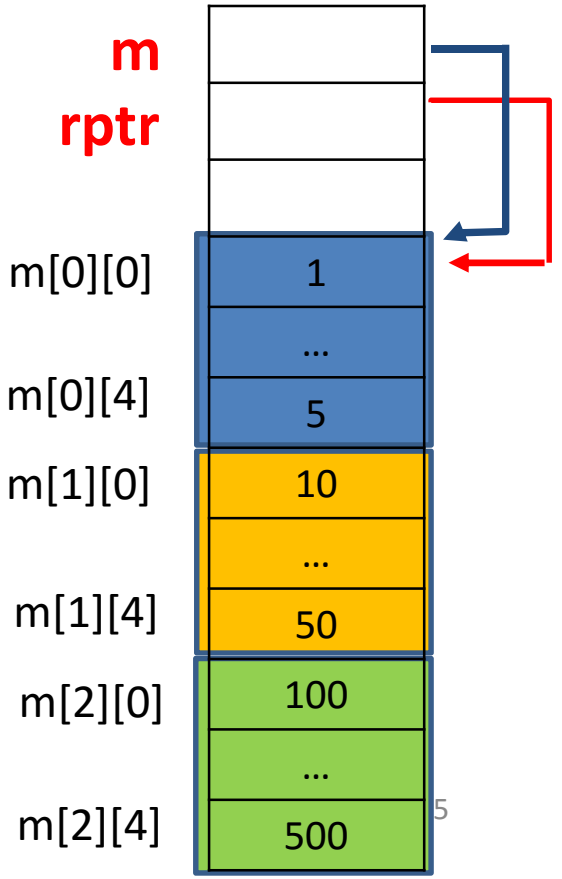
Reminder

Operator	Type	Associativity
Fucntion call: () Array subscript: []		Left to right
(type) + - ++ -- **! & * sizeof**	Unary	Right to left
* / %	Binary	Left to right
+ -	Binary	Left to right
< <= > >=	Binary	Left to right
== !=	Binary	Left to right
&&	Binary	Left to right
||	Binary	Left to right
= *= /= %= += -=	Binary	Right to left
,		Left to right

Example: Column Sum

```
void f(int m[][5],int no_rows, int no_cols, int target)
{ int  i, (*rptr)[5] = m, sum =0;
  for (i=0; i<no_rows; i++)
        sum +=  rptr[i][target]; OR
        sum +=  *(rptr[i]+ target); OR
        sum += *(*(rptr+i) + target); OR
        sum +=    (*(rptr+i))[target]; OR

       sum +=  (*rptr)[target];
       rptr++; OR
       sum +=  (*rptr++)[target];
```

```
int m[3][5] ={
{1,    2,    3,    4,    5},
{10,  20,   30,   40,   50},
{100, 200, 300, 400, 500}};
f(m, 3, 5, 1);
```



CEng 140

Multi Dimensional Arrays and Pointers

Dynamic 2D Arrays

Creating Dynamic 2D Arrays

- True 2D:
  - Both dimension lengths are known at compile time
  - Ex: I have 20 students and 5 int grades per student:
  - int stu_grades[20][5];
- Dynamic:
  - First dimension length is determined dynamically
  - Second dimension length is determined dynamically
  - Both dimension lengths are determined dynamically

Case 1: First dim dynamic

- I have 5 int grades per student, but number of students wil be determined during run-time

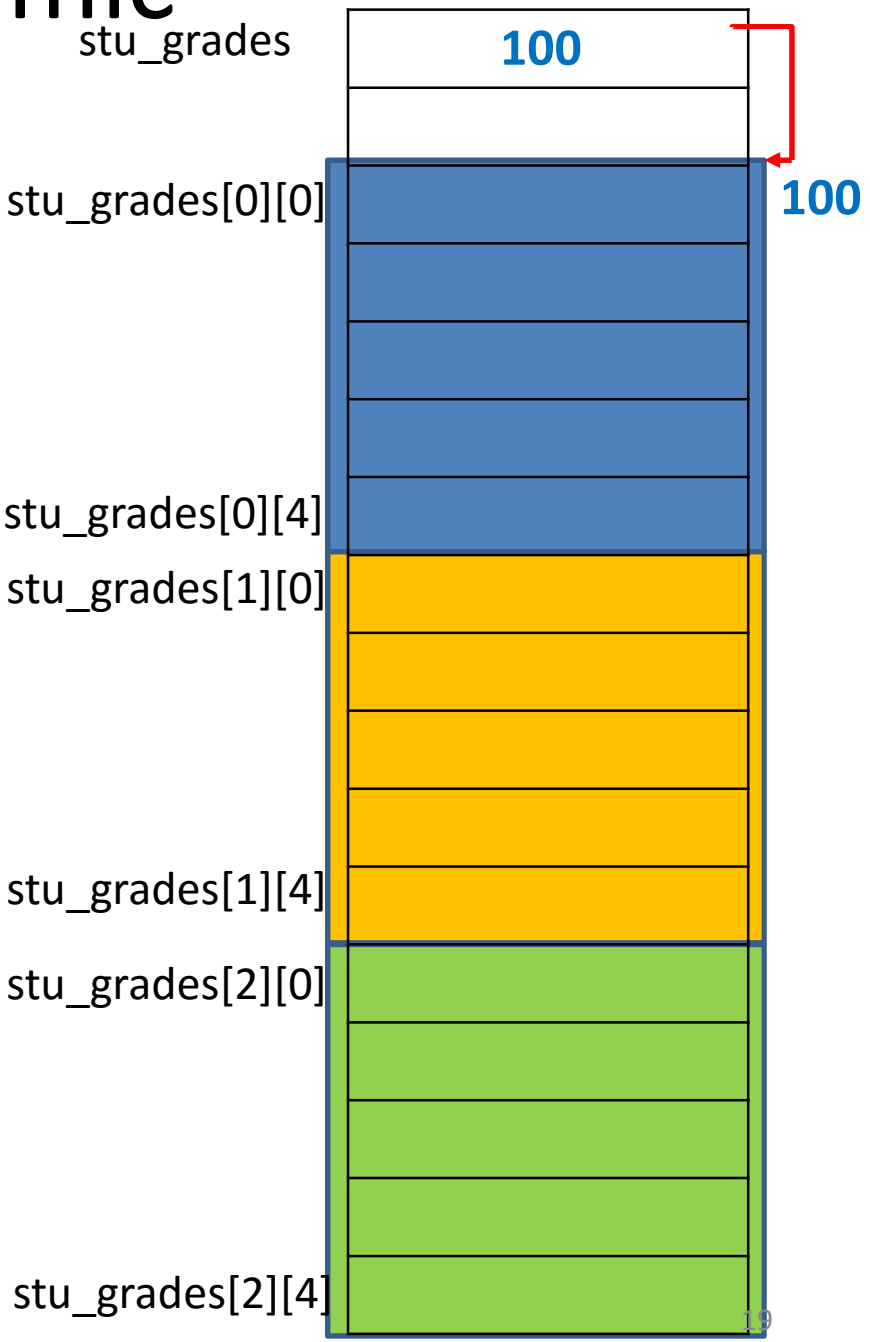
```
int main()
{  int (*stu_grades)[5]; /* ptr to a block of 5 ints */
   int no_of_stu, i, j, temp;
   scanf("%d", &no_of_stu);
   stu_grades = (int (*)[5]) malloc(sizeof(int [5])*no_of_stu);
   for (i=0; i< no_of_stu; i++)
     for (j=0; j<5; j++)
     {   scanf("%d", &temp);
         stu_grades[i][j] = temp; }
}        /* Lets draw this on board */
```

sizeof(***stu_grades**)

Allocated area is CONTIGUOUS

First dim dynamic

```
int main()
{  int (*stu_grades)[5];
   int no_of_stu, i, j, temp;
    scanf("%d", &no_of_stu);
    // assume 3
    stu_grades = (int (*)[5])
          malloc(sizeof(int [5])*3);


for (i=0; i< no_of_stu; i++)
      for (j=0; j<5; j++)
      {   scanf("%d", &temp);
          stu_grades[i][j] = temp; }}
```

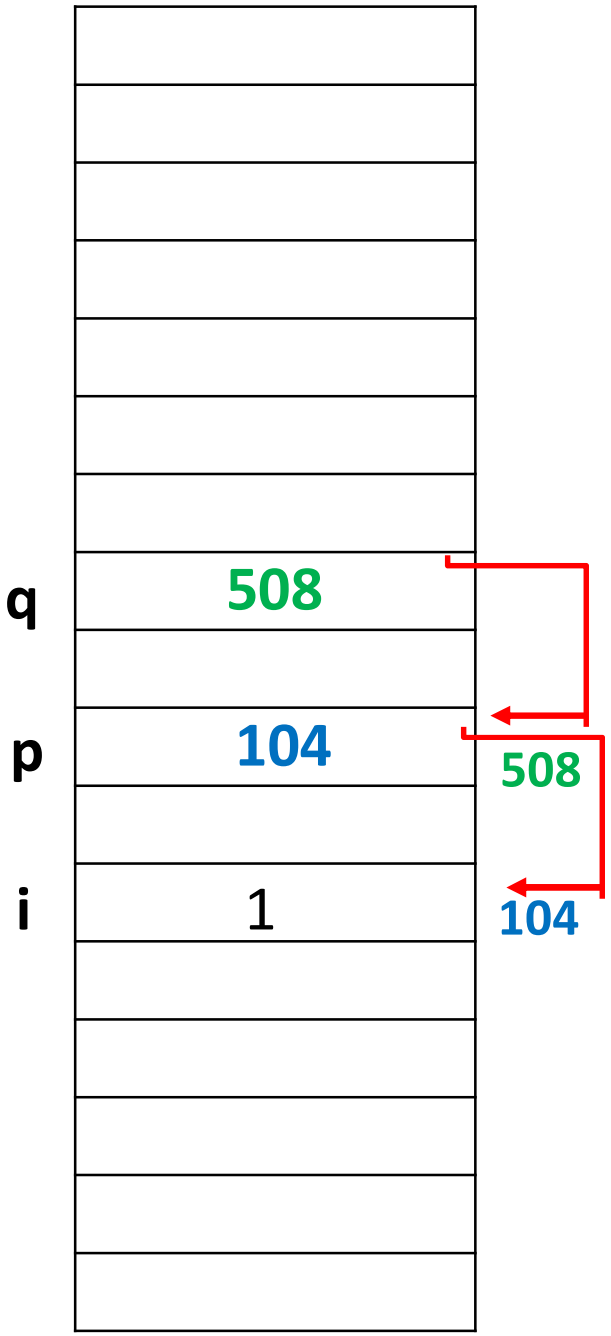


Case 1: First dim dynamic

```
int main()
{  int (*stu_grades)[5]; /* ptr to a block of 5 ints */
   …
   stu_grades = (int (*)[5]) malloc(sizeof(int [5])*no_of_stu);
   f(stu_grades, no_of_stu, 5);
}
```

Sending this array as a **parameter** to a function

```
  void f(int stu_grades[][5], int no_of_stu, int no_of_gra)
  void f(int (*stu_grades)[5], int no_of_stu, int no_of_gra)
  { stu_grades[i][j] = … }
```



Pointers to Pointers

int i =1;

int *p;

p = &i;

Assume some variable q

q=&p;

What is the type of q?

A ptr to ptr to integer!

printf("%d", i) OR

printf("%d", *p) OR

printf("%d", **q)

How to declare it?

int **q;

No limits on the levels of indirection!

Case 2: Second dim dynamic

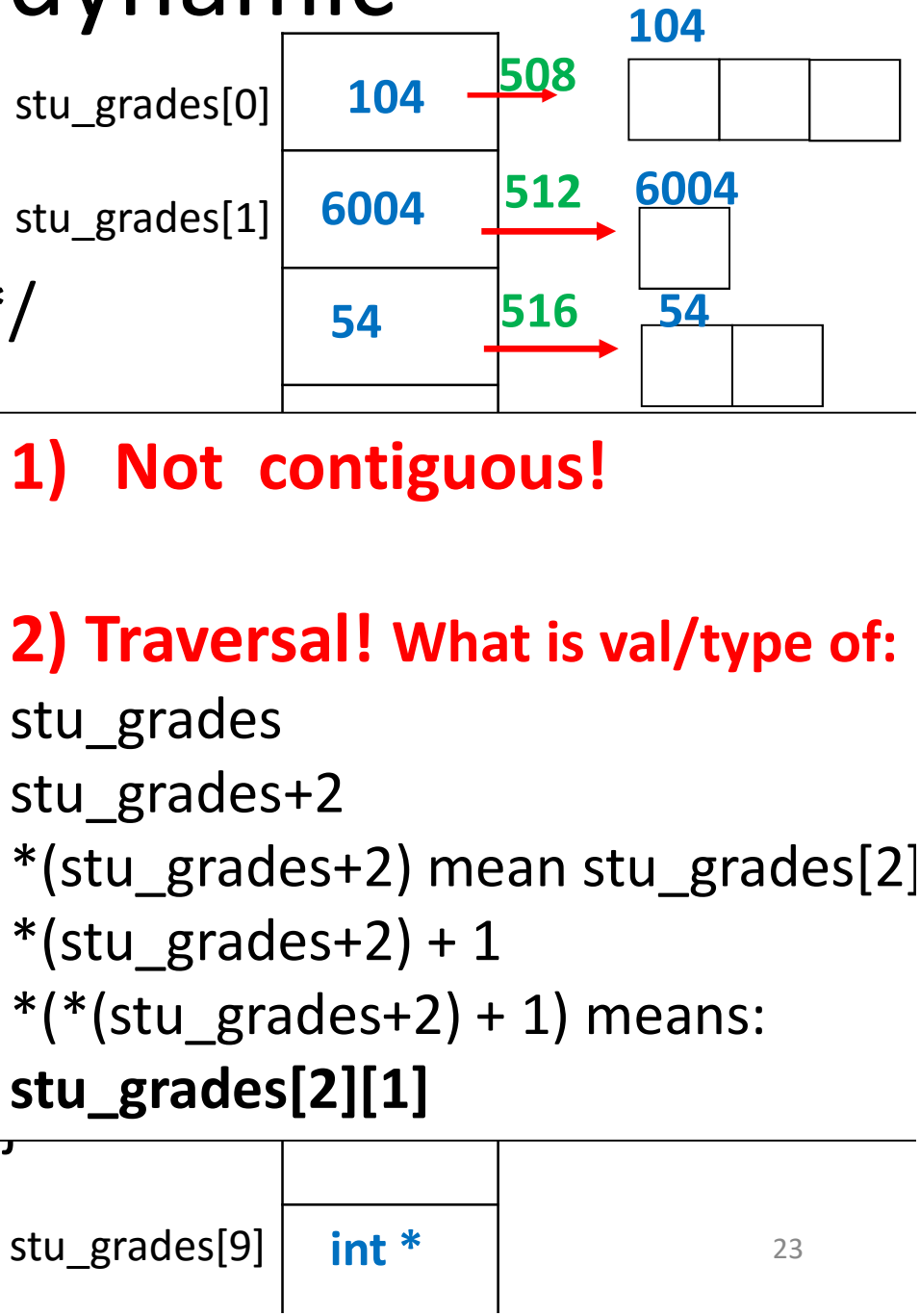
- I have 10 students, but number of grades per student wil be determined during run-time → iliffe (ragged) vector

Not contiguous!

```
int main()
{  int *stu_grades[10]; /* array of 10 pointers to int */
   int no_of_gra, i, j, temp;
   for (i=0; i< 10; i++)
   {  scanf("%d", & no_of_gra);
      stu_grades[i] = (int *) malloc(sizeof(int)*no_of_gra);
      for (j=0; j< no_of_gra; j++)
      {   scanf("%d", &temp);
          stu_grades[i][j] = temp; }
   }
}      /* Lets draw this on the board */
```

Case 2: Second dim dynamic

```
int main()
{  int *stu_grades[10];
   /* array of 10 pointers to int */
    int no_of_gra, i, j, temp;
    for (i=0; i< 10; i++)
    {  scanf("%d", & no_of_gra);
       stu_grades[i] = (int *)
malloc(sizeof(int)*no_of_gra);
       for (j=0; j< no_of_gra; j++)
       {   scanf("%d", &temp);
           stu_grades[i][j] = temp;
    } }
```



1) Not contiguous!

2) Traversal! What is val/type of:

stu_grades

stu_grades+2

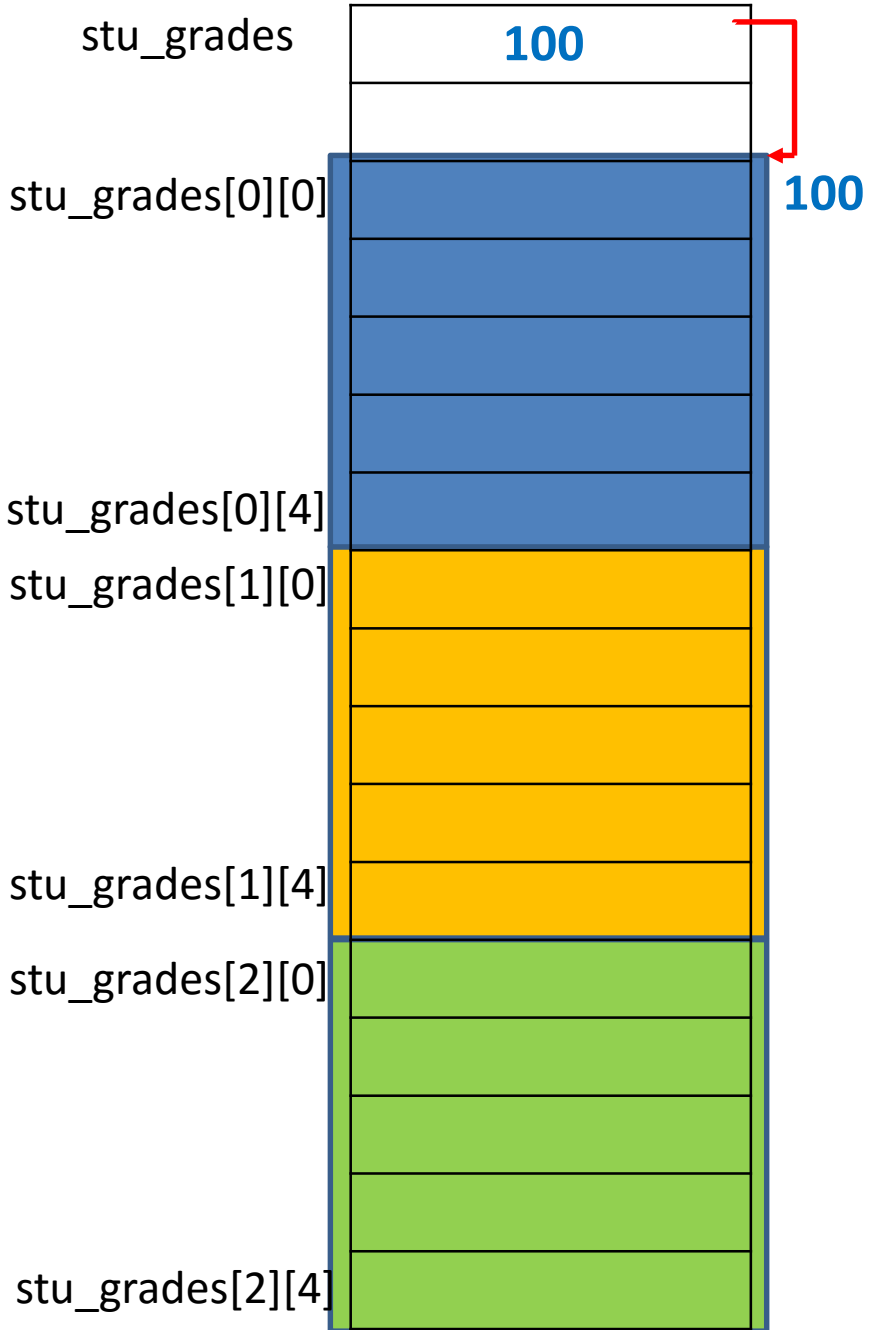
*(stu_grades+2) mean stu_grades[2]

*(stu_grades+2) + 1

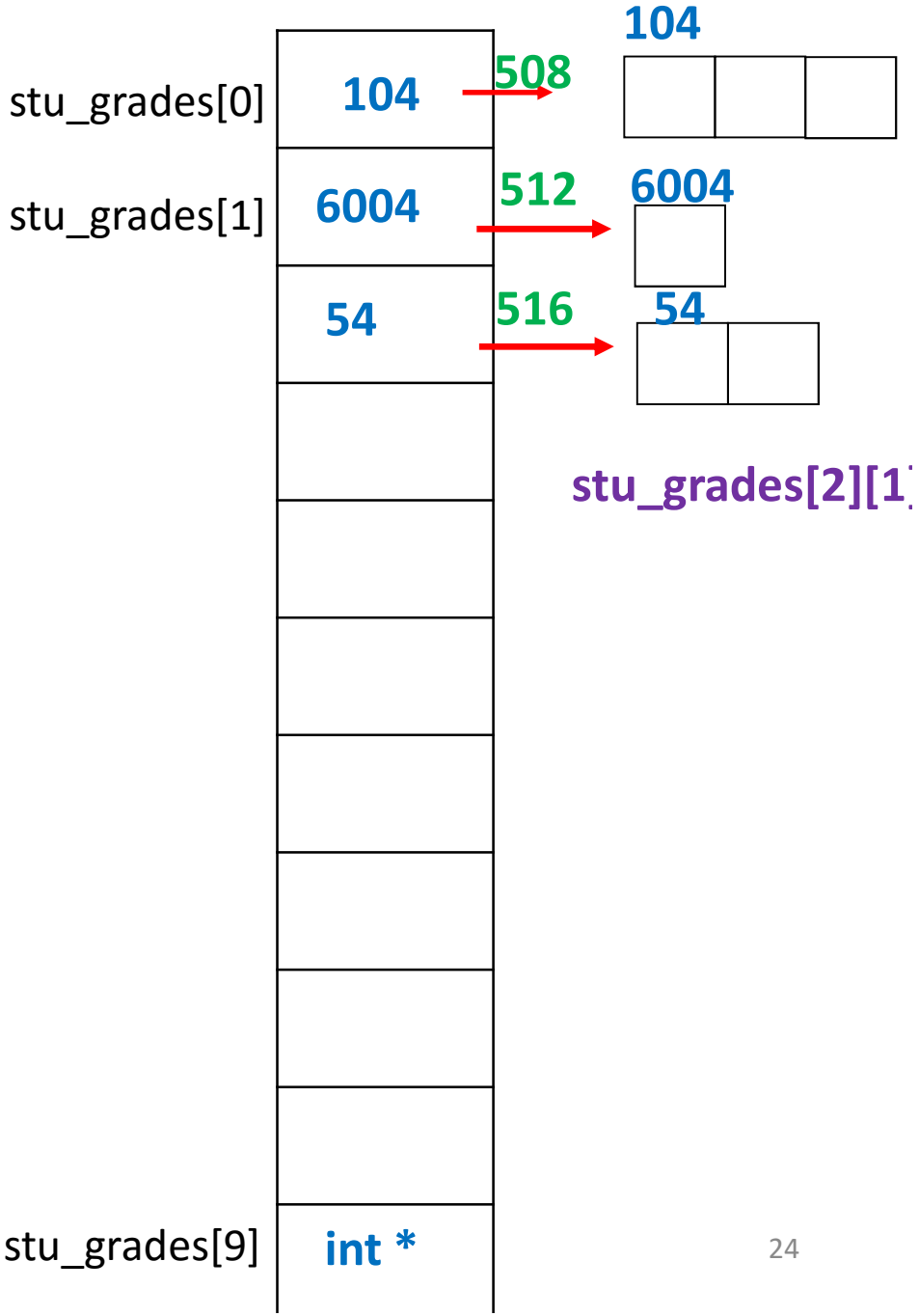
((stu_grades+2) + 1) means:

stu_grades[2][1]

int (*stu_grades)[5];



int *stu_grades[10];



Case 2: Second dim dynamic

```
int main()
{  int *stu_grades[10]; /* array of 10 pointers to int */
   …
  f(stu_grades, 10, …);
}
```

Sending this array as a **parameter** to a function

void f(**int *stu_grades[]**, …) → stu_grades is the ptr to first arrays element, which is of type int * so:

```
void f(int **stu_grades, …)
{ stu_grades[i][j] = … }
```

In main or f(), notation is the same while accessing array elements, but what really happens is slightly different than Case1 or true-2D array

Case 3: Both dim lenghts dynamic

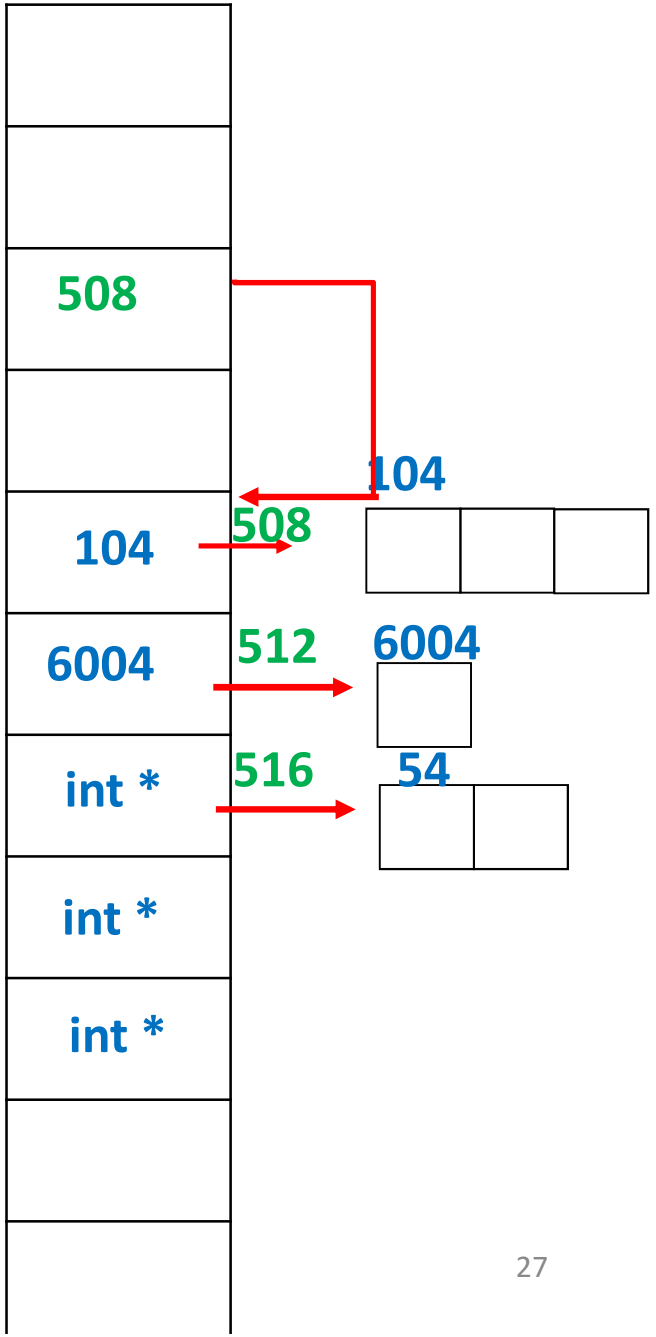
- Both number of students and grades per student wil be determined during run-time

Not contiguous!

```
int main()
{  int **stu_grades;
   int no_of_gra, no_of_stu, i, j, temp;
   scanf("%d", &no_of_stu);
   stu_grades = (int **) malloc(sizeof(int *)*no_of_stu);
   for (i=0; i< no_of_stu; i++)
   {  scanf("%d", & no_of_gra);
      stu_grades[i] = (int *) malloc(sizeof(int)*no_of_gra);
      for (j=0; j< no_of_gra; j++)
      {   scanf("%d", &temp);  stu_grades[i][j] = temp; }
   } }      /* Lets draw this on the board */
```

Case 3: Both dynamic

```
int main()
{  int **stu_grades;
   int no_of_gra, no_of_stu, i, j, temp;

   scanf("%d", &no_of_stu);
   stu_grades = (int **) malloc(sizeof(int *)*no_of_stu);

   for (i=0; i< no_of_stu; i++)
   {  scanf("%d", & no_of_gra);
      stu_grades[i] = (int *) malloc(sizeof(int)*no_of_gra);
      for (j=0; j< no_of_gra; j++)
      {   scanf("%d", &temp);  stu_grades[i][j] = temp; }
    }
}
```



Case 3: Both dim lenghts dynamic

```
int main()
{  int **stu_grades;
    …
   f(stu_grades, …, …);
}
```

Sending this array as a **parameter** to a function

```
   void f(int **stu_grades, …)
   { stu_grades[i][j] = … }
```

Recall: In Case 2 and Case 3 we should also store the no of grades of grades per student to be able to access them correctly later…

CEng 140

Strings and Pointers

Strings (and Pointers)

- [As we know] C uses NULL terminated arrays of chars to represent strings
- To create a string variable you must **allocate** sufficient space for the number of characters and the NULL character '\0'.
  - Using arrays
  - Using pointers

Using arrays for strings

char robot[5]; // declaration

- **Assignment** of a string to an array: two ways
- First way: each array element assigned to a char

robot[0] = ‘g’;

robot[1] = ‘o’;

robot[2] = ‘o’;

robot[3] = ‘d’;

robot[4] = ‘\0’;

robot[0]	g
robot[1]	o
robot[2]	o
robot[3]	d
robot[4]	\0

Using arrays for strings

char robot[5]; // declaration

- **Assignment** of a string to an array: two ways
- second way: via strcpy func

//strcpy copies the chars one by one from

// source str to destination str

strcpy(robot, "good");

String constant

robot[0]	g
robot[1]	o
robot[2]	o
robot[3]	d
robot[4]	\0

Using arrays for strings

- You can also store a string in an array during the **initialization**

```
char robot[5];   // declaration
char robot[5] = {‘g’, ‘o’, ‘o’, ’d’, ‘\0’};   // or
char robot[5] = "good";
```

robot[0]	g
robot[1]	o
robot[2]	o
robot[3]	d
robot[4]	\0

When a char array is intialized to a string constant:

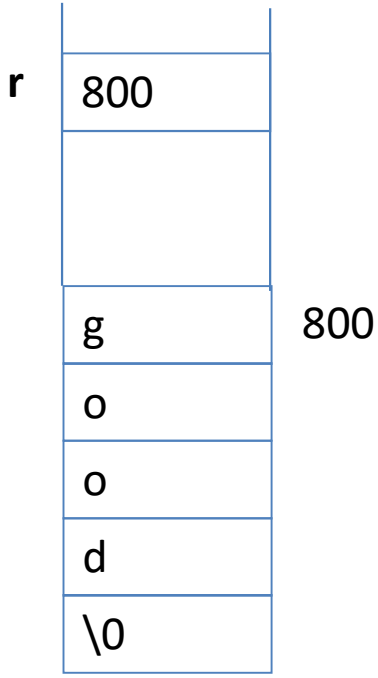
- Same name (robot) always refers to the same storage
- Individual chars can be modified by assignments!

```
robot[0] = ‘w’; // works
```

Using pointers for arrays

```
char *r;  // declaration
          // normally, alloc space for string before assignment
r = (char *) malloc(sizeof(char) * 5);
// Assignment: first way
r[0] = 'g';
r[1] = 'o';
...
r[4] = '\0';
// Assignment: second way
strcpy(r, "good");
```

r[0] = 'w'; **// works**



Using pointers for arrays

- You can also store a **string constant** in a ptr via **(initialization, or) direct assignment**

```
char *r;  // declaration
char *r = "good";  //or
char *r;
r = "good";
```

Hey! You did not allocate any storage for the string, how is this possible?

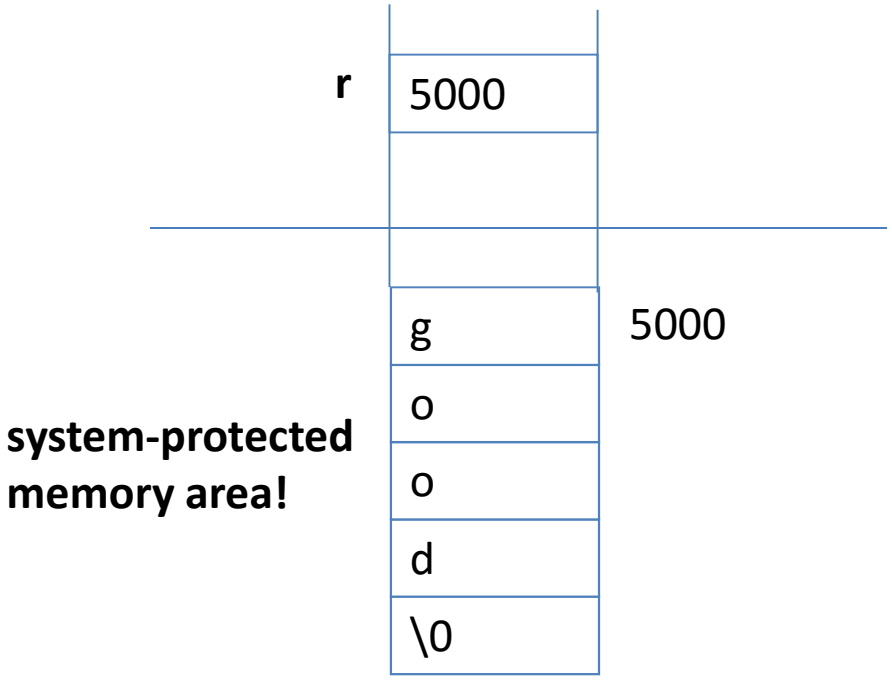
[More about the] String Constants

- A string constant is a sequence of chars in " " and compiler automatically adds NULL character at the end.
- When a string constant appears anywhere (**except** as an initializer of a char array or an argument to the sizeof operator) the chars making up the string (together with NULL) are stored in contiguous memory locations, and **string contant** becomes a **pointer** to the first char of the stored string.
  - Usually stored in a **system-protected memory area!**

Mystery solved!

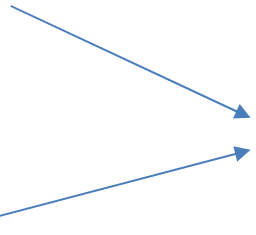
```
char *r = "good";  //or
char *r;
r = "good";
```

String constant;
a pointer to where "good" is stored!
Both sides are of type **pointer to char**!



Mystery solved!

```
char *r = "good";  //or
char *r;
r = "good";
```



String constant;
a pointer to where "good" is stored!
Both sides are of type **pointer to char**!

When a char pointer is intialized/assigned to a string constant:

- Pointer var may be assigned to point somewhere else
- But can **NOT** modify the string pointed by it!

  r [0] = ‘w’; // fails! **Result is undefined!**

Let’s recall again cases with a string constant:

- If your variable has its own memory and you copy string constant there, you can modify it as you wish, as in:
  - char robot[5];

    strcpy(robot, "good");
  - char robot[5] = "good";
  - char *r;

    r = (char *) malloc(sizeof(char) * 5);

    strcpy(r, "good");

Otherwise…!

```
char *r;  //
r = (char *) malloc(sizeof(char) * 5);
r = "good";
r [0] = ‘w’; // What will happen?
```

Result is undefined! Bec you are not using the allocated memory but pointing to a string constant, which is not modifiable!

Otherwise…!

```
char robot[5];
robot = "good"; // What will happen?
```

RECALL this is not array initialization (where string constant behaves exceptionally), so you are simply trying to change where an array name points to!

→ compile-time error!

strcpy()

- Now that we know what a string constant really is (i.e., a ptr to char)…
- what should be the prototype of strcpy() function?
  - char robot[5];

    strcpy(robot, "good");
  - char *r;

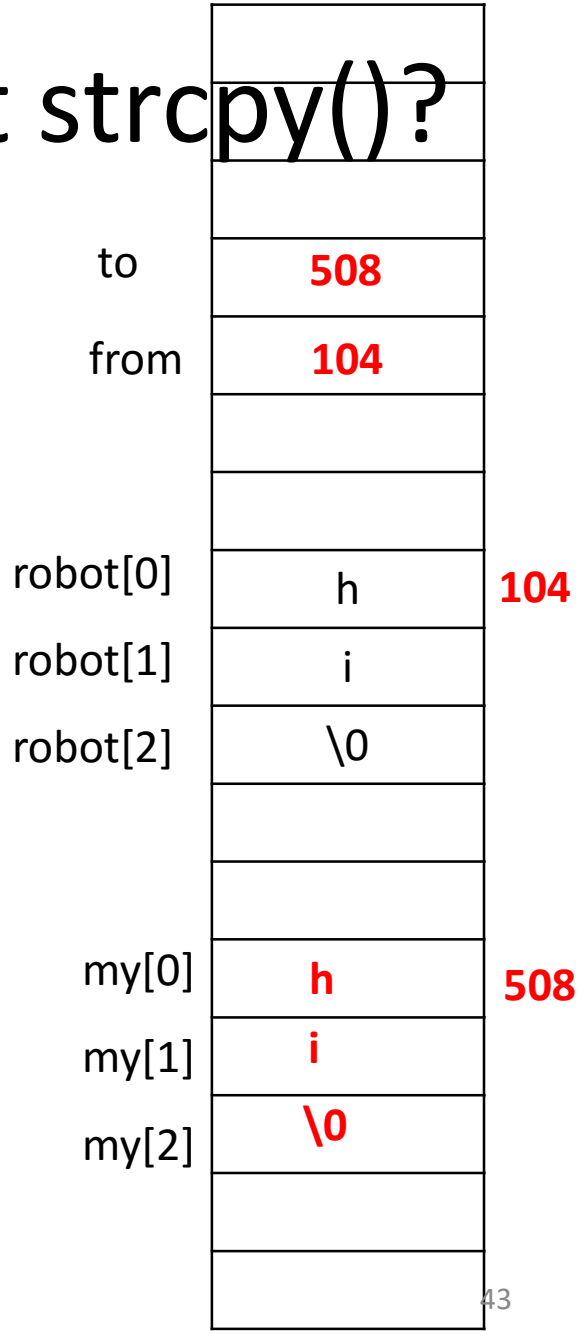
    r = (char *) malloc(sizeof(char) * 5);

    strcpy(r, "good");

    or, strcpy(r, robot);

How can we implement strcpy()?

```
void strcpy(char *to, char *from)
{  while (*to = *from)
        to++ , from++ ; }

int main()
{
char my[3], robot[3]="hi";
strcpy(my, robot); }
```



How can we implement strcpy()?

```
void strcpy(char *to, char *from)
{  while (*to = *from)
        to++ , from++ ; }
```

Shorter:

```
void strcpy(char *to, char *from)
{  while (*to++ = *from++) ; }
```

```
char robot[5], my[8];
strcpy(robot, "good"); strcpy(my, robot); …
```

How can we implement strlen()?

```
int my_strlen(char str[])
{ int i;
   for (i=0; str[i] != '\0'; i++) ;
   return i;
}
```

C Library Functions

Declared in string.h

size_t → unsigned integral type

size_t **strlen**(const char *s); (length of s w.o. NULL)

char ***strcpy**(char *s1, const char *s2);

(copies s2 to s1 including NULL, returns s1)

Sec. 7.4.1:

…strncpy..

…strcat..

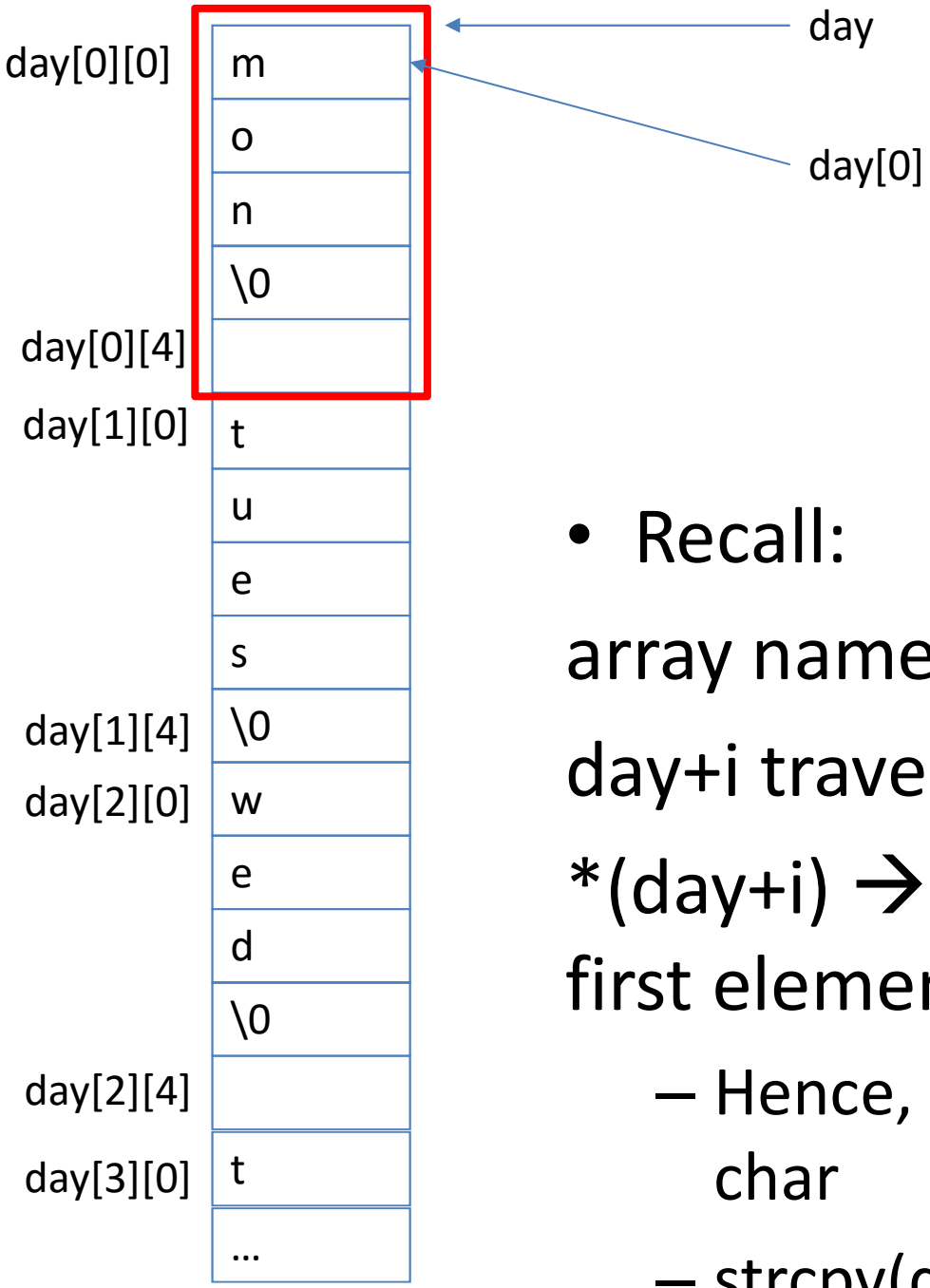
…strcmp…

Array of strings

- True 2D, initialized

```
char robot[5] = {‘g’, ‘o’, ‘o’, ’d’,
‘\0’};   // or
char robot[5] = "good";
```

```
char day[7][5] = {"mon",..., "sun"};
// OR, I could first declare array and then assign as:
char day[7][5];
day[0][0] = ‘m’; …
// OR, assign as:
strcpy(day[0], "mon");
// In all cases, strings in the array are modifiable!
```



- Recall:

array name is a ptr to first array!

day+i traverses arrays

*(day+i) → day[i] is a ptr to the first element in the ith array!

  - Hence, day[i] is of type ptr to char
  - strcpy(day[0], "mon"); is OK!

Pop-up quiz

```
char day[7][5] = {"mon",…, "sun"}; OK

char day[7][5];

day[0]="hey";

day[0][1]= ‘m’; What will happen?
```

a) Compile-time error

b) Run-time error: string is not modifiable

c) Undefined: string is not modifiable

d) String becomes mey

We have seen

```
char robot[5] = "good"; OK

char robot[5] ;

robot= "good"; COMPILE ERROR
```

Array of strings

- True 2D, passing as a parameter:

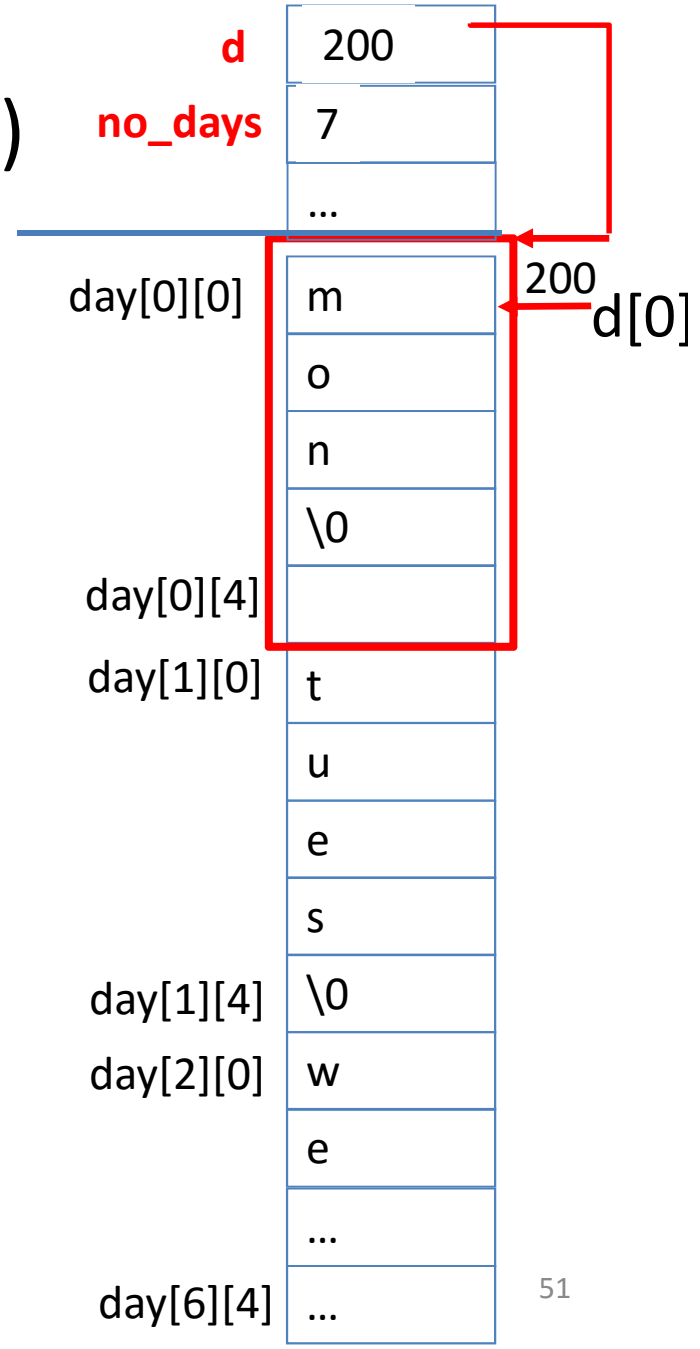
Rewritten as: **char (*d)[5]**

```
void list_days (char d[][5], no_days)
{ int i;
  for (i=0; i<no_days; i++)
     printf("%s\n", d[i]); }
```

No need for second dim length,
as each array is ended with NULL!

Rewritten as: **char (*d)[5]**

```
void list_days (char d[][5], no_days)
{ int i;
   for (i=0; i<no_days; i++)
       printf("%s\n", d[i]); }

int main(void)
{char day[7][5] = {"mon",..., "sun"};
  list_days(day,7);}
```



Array of strings

- Dynamic 2D, iliffe vector, can be:

```
char *day[7] = {"mon",…, "sun"};
```

Not-modifiable

Or:

```
char *day[7];
day[0] = "mon";
```

Not-modifiable

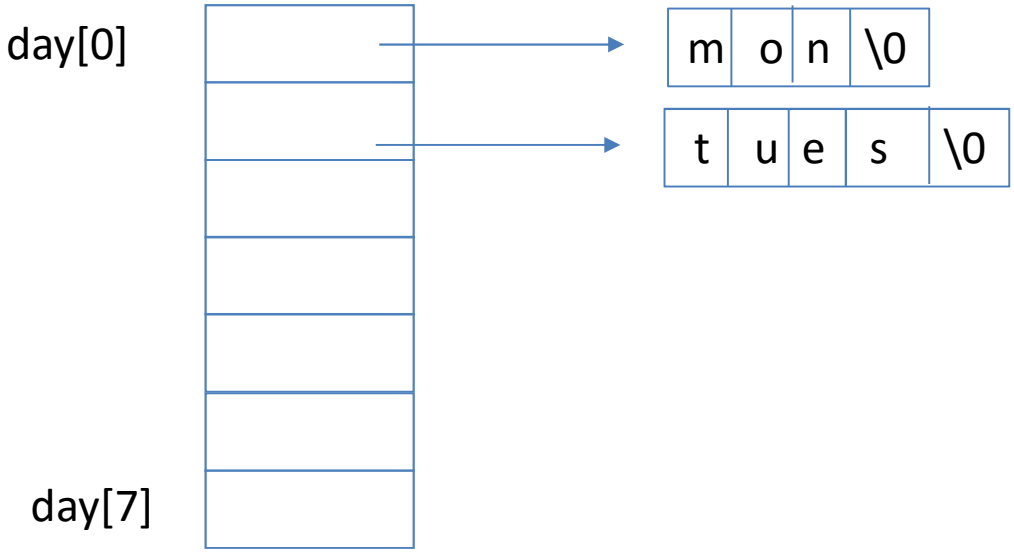
In what cases, strings are **modifiable**?

Or:

```
char *day[7];
day[0] = (char *) malloc (sizeof("mon"));
strcpy(day[0], "mon"); // or: day[0][0] = 'm'; …
```

modifiable!

Array of strings



- day[i] is of type ptr to char
- Note that pointed memory space is either explicitly allocated, or system-area (if ptr is assigned to a str constant)

Array of strings

- Dynamic 2D, iliffe vector, passing as a parameter:

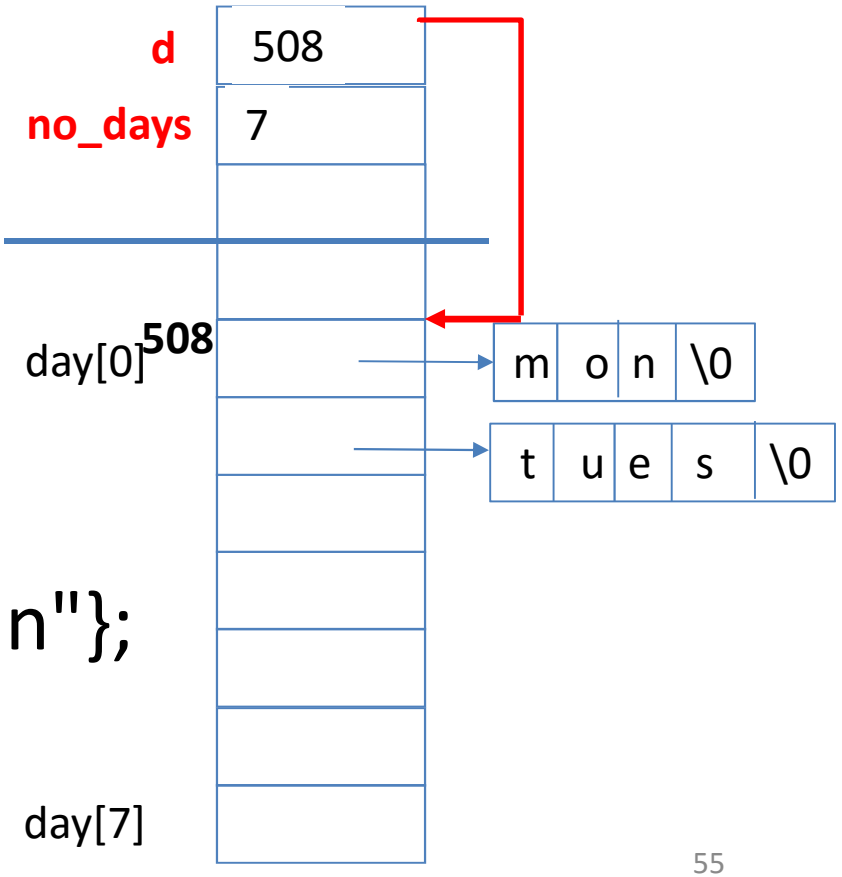
Rewritten as: **char **d**

```
void list_days (char *d[], no_days)
{ int i;
   for (i=0; i<no_days; i++)
       printf("%s\n", d[i]); }
```

Array of strings

Rewritten as: **char **d**

```
void list_days (char *d[], no_days)
{ int i;
   for (i=0; i<no_days; i++)
       printf("%s\n", d[i]); }

int main(void)
{char *day[7] = {"mon",…, "sun"};
  list_days(day,7);}
```



Parameters of main()

- main can be defined with formal parameters so that it can accept command-line arguments
  - main defined as having two parameters, typically called as argc and argv, as follows:

Rewritten as: **char **argv**

int main(int argc, char *argv[])

number of command line args

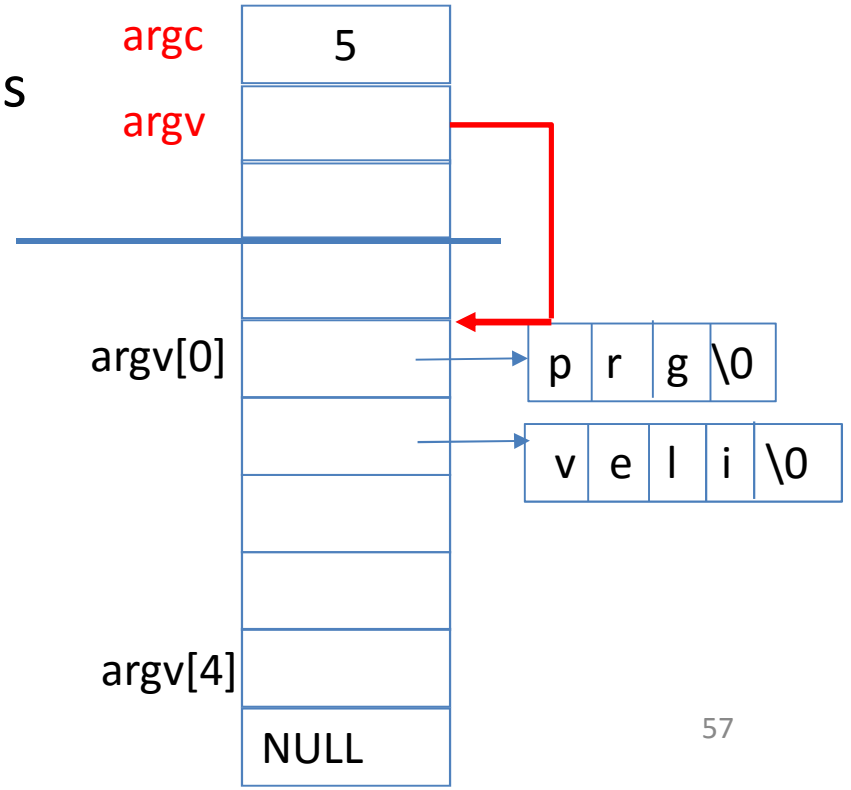
an array of pointers to chars (strings representing args)

Parameters of main()

- Compile your prg.c as executable prg

  ./prg veli ali ayse fatma

- argc: 5, argv is as shown in figure:
  - argv[0] points to the name of the program
  - argv[1] to argv[argc-1] point to args
  - argv[argc] is NULL by convention

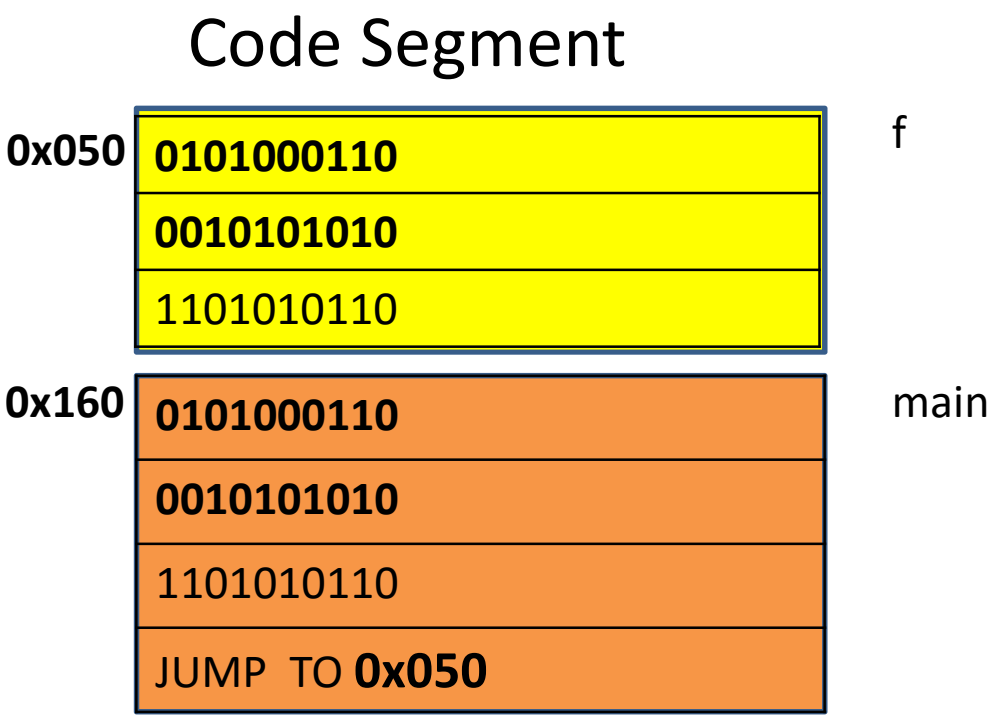


Parameters of main()

- So, the command line arguments are strings
  - if needed you can convert them to other types
  - long int atoi(char *) → string to int
  - more functions in Section Appendix A.6 of the textbook

Pointers to Functions

- It is possible to define a pointer to a function.
  - Best thought as the address of the code executed when the func is called



Declaration

- While declaring, the pointed-to functions's type together with parameters is specified
- int (*fp)(int i, int j);
- fp is a pointer to function that takes two int (as arguments) and returns an int
- int **(***fp**)**(int i, int j);
  - paranheses are necessary: what is

    int *fp(int i, int j); /* declaration of func fp */
  - parameter names İ an j may be omitted

What do these declare?

- int i(void);
  - declares func i with no parameters and returns int
- int *i(void);
  - declares func i with no parameters and returns a pointer to an int
- int (*i)(void);
  - declares i is a pointer to a function with no parameters and returns an int
- int * (*i)(void)
  - declares i is a pointer to a function with no parameters and returns a pointer to an int

Assignment

- To make a pointer point to a specific function, just assign the pointer to the function name (**without** parentheses/params etc).

```
int gcd(int, int);
int (*fp)(int, int);
fp = gcd;
```

Dereferencing and Func. Call

- To call the pointed function, dereference it (with arguments)

```
int gcd(int, int);
int (*fp)(int, int);
fp = gcd;
(*fp)(42, 56);
```

- Parentheses are necessary to dereference **before** the func call

Reminder

Operator	Type	Associativity
Fucntion call: () Array subscript: []		Left to right
(type) + - ++ -- **! & * sizeof**	Unary	Right to left
* / %	Binary	Left to right
+ -	Binary	Left to right
< <= > >=	Binary	Left to right
== !=	Binary	Left to right
&&	Binary	Left to right
||	Binary	Left to right
= *= /= %= += -=	Binary	Right to left
,		Left to right

Dereferencing and Func. Call

void (*gp)(void);

void initialize(void);

- First one is a ptr, second one is a func decl.
- Could write as

  void (*gp)(void), initialize(void);

gp = initialize;

(*gp); Is this ok?

NO (*gp)();

Why are they useful?

- For passing funcitons as arguments to functions

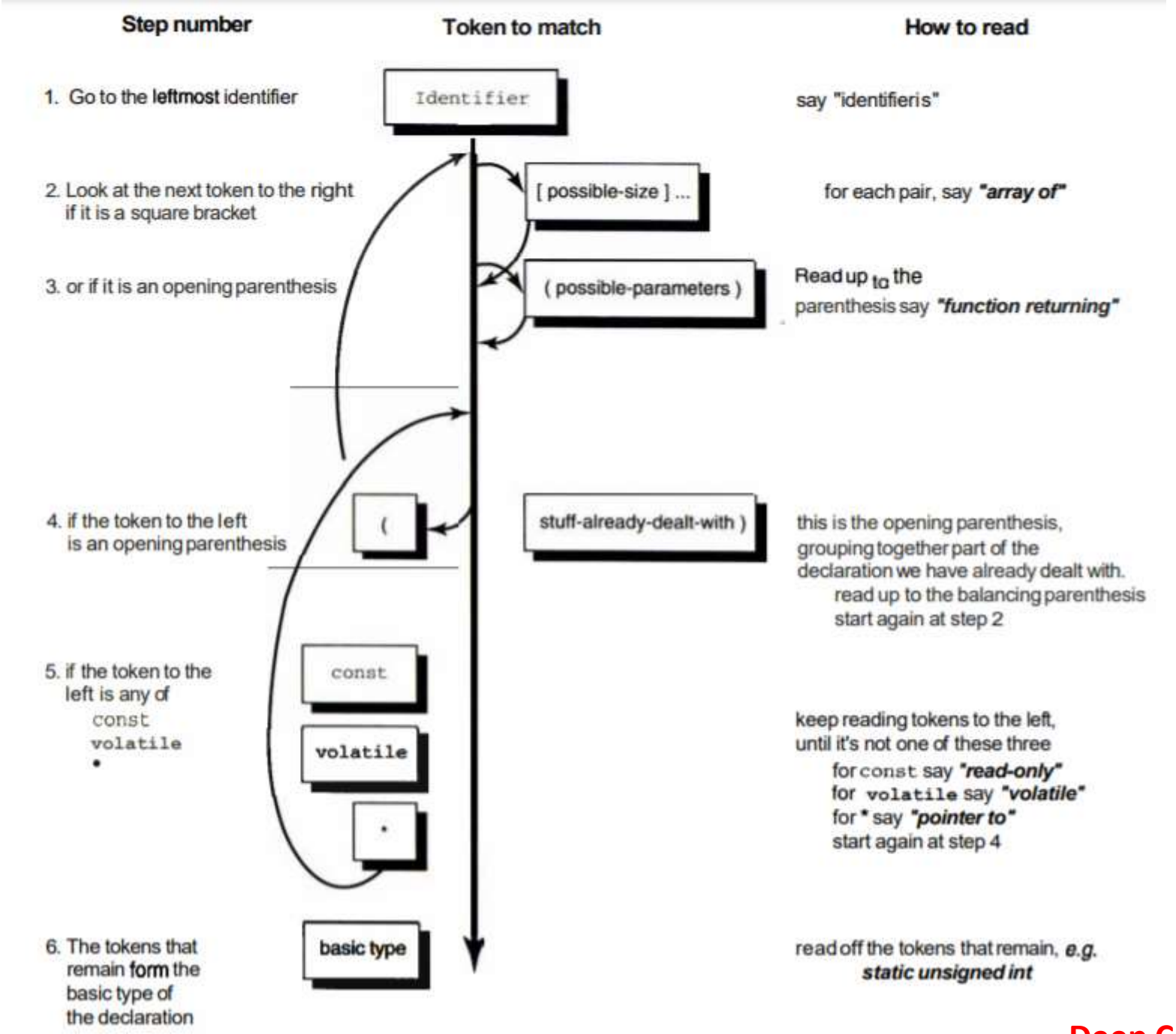
```
void table(int (*fp)(int, int), int *x, int *y, len)
{ int i;
  for (i=0; i<len; i++)
      printf("%d &d %d", x[i], y[i], (*fp)(x[i], y[i]) );  }

int gcd(int x, int y)
{ …. }
int lcm(int x, int y);
{ … }
int x[10] = {10, 24, …}, y[10] = {5, 120, …};
/*we want to have  a table of gcd (or lcm) of each pair in arrays*/
table(gcd, x, y, 10);
table(lcm, x, y, 10);
```

What is this?

- int * (*foo[])();

- See the **Magic Decoder Ring for C Declarations** ("Deep C Secrets« book)
  - Also available temporarily at:

  http://user.ceng.metu.edu.tr/~ceng140/c_decl.pdf


Step number
Token to match
How to read
1. Go to the leftmost identifier
Identifier
say "identifier is"
2. Look at the next token to the right if it is a square bracket
[possible-size] ...
for each pair, say "array of"
3. or if it is an opening parenthesis
(possible-parameters)
Read up to the parenthesis say "function returning"
4. if the token to the left is an opening parenthesis
(
stuff-already-dealt-with)
this is the opening parenthesis, grouping together part of the declaration we have already dealt with.
read up to the balancing parenthesis
start again at step 2
5. if the token to the left is any of
const
volatile
*
const
volatile
*
keep reading tokens to the left, until it's not one of these three
for const say "read-only"
for volatile say "volatile"
for * say "pointer to"
start again at step 4
6. The tokens that remain form the basic type of the declaration
basic type
read off the tokens that remain, e.g.
static unsigned int

Warning