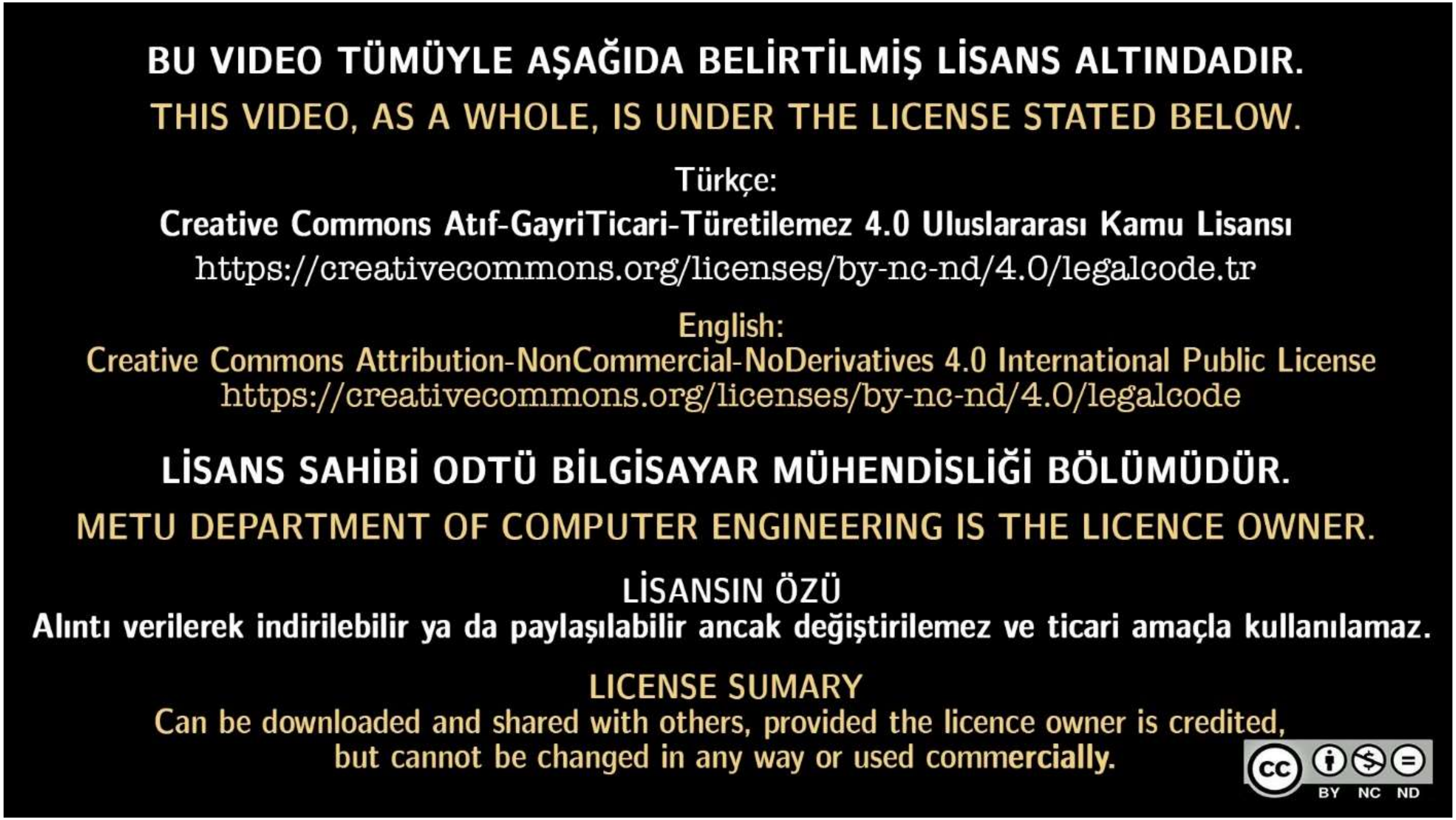
# Warning



# CEng 140

Structures

# Structures

- A **structure** is a collection of logically related data items grouped together under a single name, i.e., a **structure tag**.

- Why do we need them?
  1. Grouping
  2. Modularity
  3. Flexibility
  4. …

# Structures

- A **structure** is a collection of logically related data items grouped together under a single name, i.e., a **structure tag**.

- Data items of a structure are called its **members**, **components**, or **fields**; and can be of different types.

# Defining Structures

name of the structure (optional)

```
struct [tag]
{ variable declarations
};
```

type and name declarations
for member data items

# Example

```
struct person
 {
   int tc-id;
   int age;
   double weight;
   char gender;
 };
```

```
struct date
  { int day, month, year; };


struct course
{ int course_id;
   int student_no;
   double avg_grade;
};
```

# Naming Structure Tags & Member Variables

- Name of a member variable can be the same as its tag (as they will be differentiated by the context).

```
struct person
 {
    int person; /*no confusion with the tag name */
    int age;
    double weight;
    char gender;
};
```

# Naming Structure Tags & Member Variables

- Name of a member variable or a tag can be the same as that of some **non-member** variable.

```
struct person
  {  int person; /*no confusion with the tag name */
     int age;
     double weight;
     char gender;  };

int person;
double weight;
```

# Naming Structure Tags & Member Variables

- Two member variables in different structures can have the same name.

```
struct person
 {  int person;
    int age;
    double weight;
    char gender;  };
```

```
struct human
 {  int human;
    double weight;
    double height; };
```

```
int person;
double weight;
```

# A structure defines a **new type**

# Declaring structure variables

- Defining a structure defines a **new type**
- Variables of this type can be declared as:

```
struct date
  {
    int day, month, year;
  } order_date, arrival_date;
```

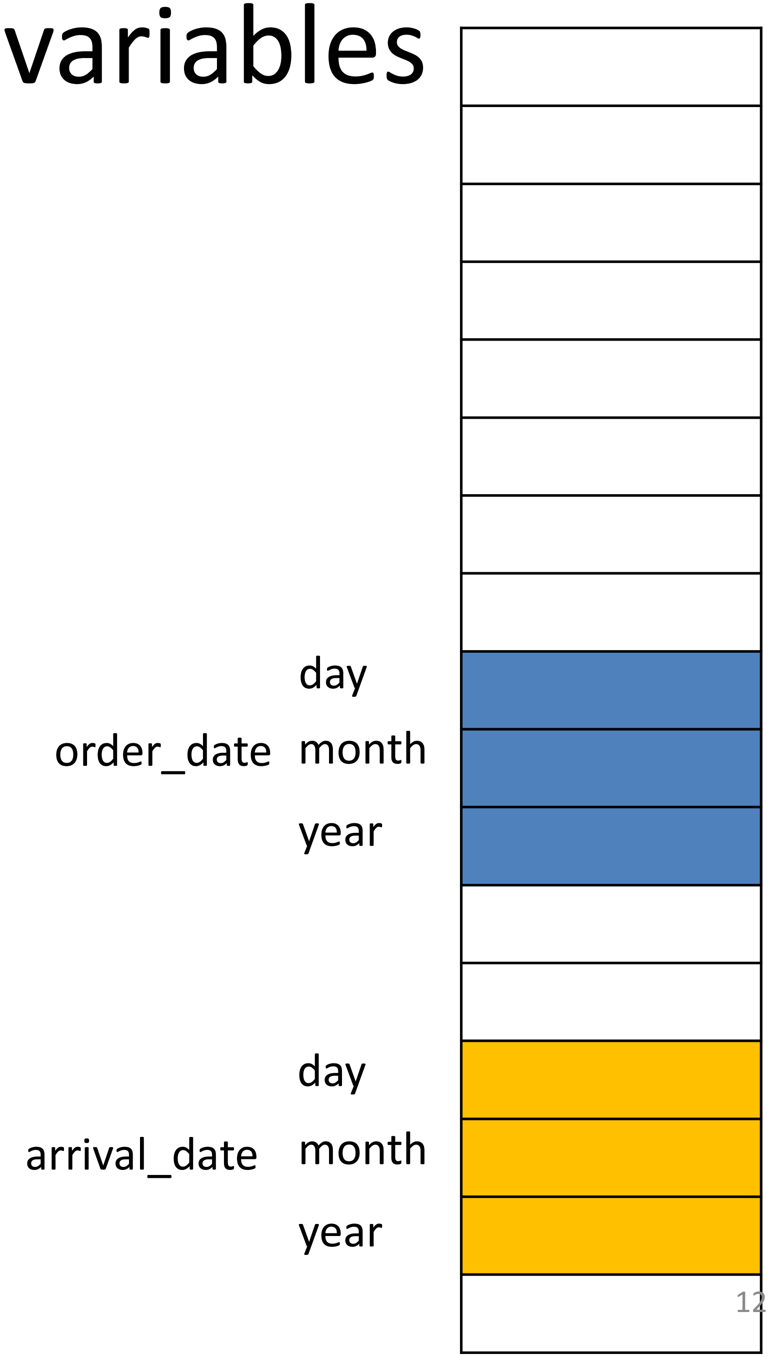
variables of type **struct date**

```
struct date
  { int day, month, year; };
struct date order_date;
struct date arrival_date;
```

variables of type **struct date**

# Declaring structure variables

```
struct date
  {
     int day, month, year;
  } order_date, arrival_date;
```



# Declaring structure variables

- Tag may be omitted, but then all variables of this type should be declared **when** the structure is defined:

```
struct
  { float r;
     float theta;
  } polar1, polar2;
```

tag omitted

Then, we cannot later subsequently define a variable polar3 with the same type of polar1 and polar2!

# Declaring structure variables

- **Each** occurrence of a structure definition introduces a **new** structure type that is neither the same nor equivalent to any other type!

```
struct { char c; int i; } u;
struct { char c; int i; } v;
struct s1 { char c; int i; } w;
struct s2 { char c; int i; } x;
struct s2 y;
```

Which of these variables are the same type?

a) ALL
b) NONE
c) u and v
d) w, x and y
e) x and y

# Declaring structure variables

- **Each** occurrence of a structure definition introduces a **new** structure type that is neither the same nor equivalent to any other type!

```
struct { char c; int i; } u;
struct { char c; int i; } v;
struct s1 { char c; int i; } w;
struct s2 { char c; int i; } x;
struct s2 y;
```

Types of the variables **u, v, w** and **x** are all **different!**

Types of the variables **x** and **y** are the **same**!

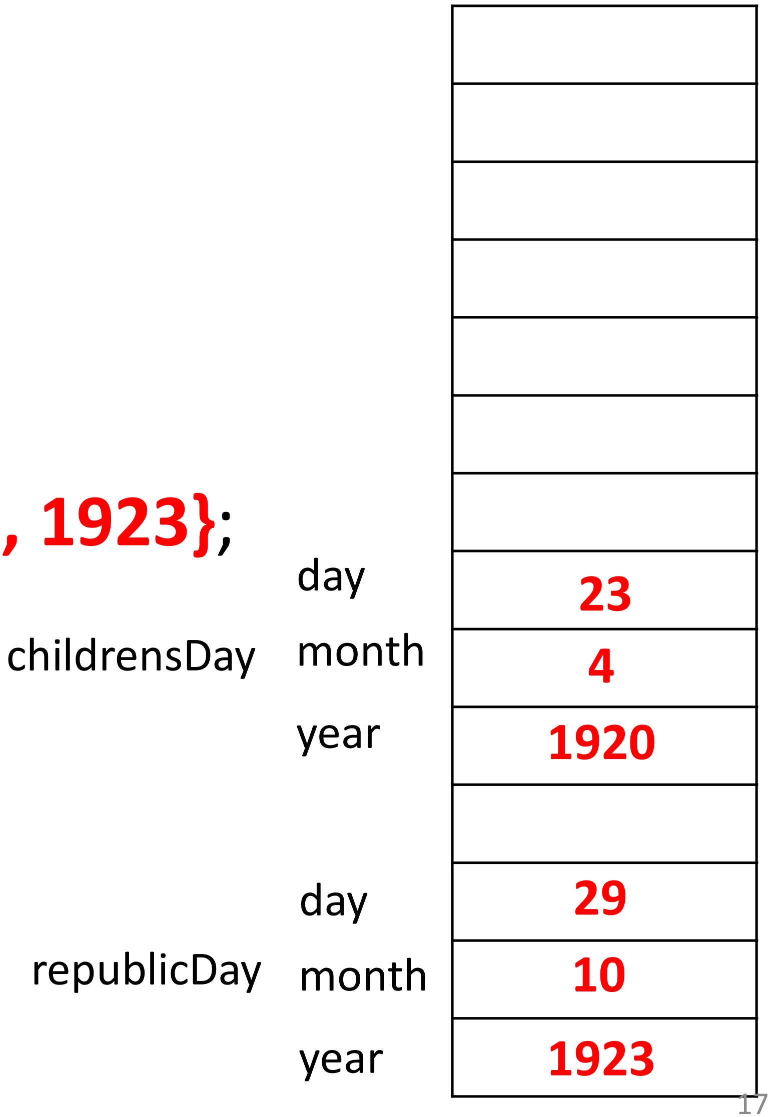
# Declaring structure variables

- A structure definition does **not** allocate any storage; it merely describes the type!

- Storage is allocated only when a variable of the corresponding type is declared!

# Initialization of structure variables

```
struct date
  {
    int day, month, year;
  } childrensDay = {23, 4, 1920};

struct date  republicDay = {29, 10, 1923};
```



# Initialization of structure variables

```
struct date
 {
    int day, month, year;
 } childrensDay = {23, 4, 1920};

struct date  republicDay = {29, 10, 1923};
```

- If there are fewer initializors, remaing ones are set to zero

```
struct date  mybDay = {21};
```

- If there are more initializors, error!

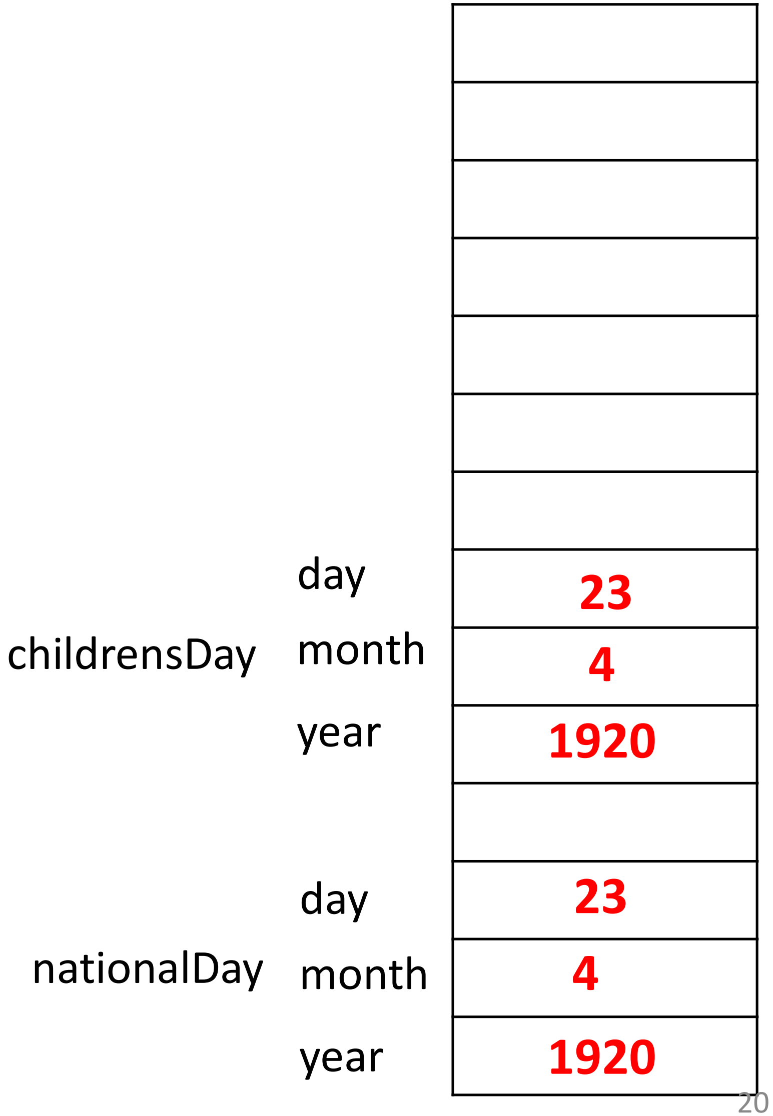
# Assignment of structure variables

- A structure variable may be assigned to another structure variable of the **same type.**

```
struct date
 {
    int day, month, year;
 } childrensDay = {23, 4, 1920};
struct date  nationalDay, republicDay = {29, 10, 1923};
nationalDay = republicDay;
```

# Assignment of structure variables

```
struct date
 {
    int day, month, year;
 } childrensDay = {23, 4, 1920};

struct date nationalDay;
nationalDay = childrensDay;
```



# Accesing Structure Members

- For accessing the members of a structure, we use the **dot operator**:

```
structure_var.member_name
struct date  nationalDay;
nationalDay.day = 29;
nationalDay.month=10;
nationalDay.year= 1923;
```

- The dot operator has the **same precedence** with function call (), array subscript [], and arrow operator -> (but **higher** than any other C operator)
- And, it is **left-to-right** associative.

# Size of a struct

- You can use the sizeof operator on structures.
- The size of a struct may be more than the sum of the sizes of its members.
- For example:

```
struct st_type {char a; int b;} var1;
```

➔ The size of var1 is probably more than 5 (due to **data alignment** with memory words)

- However, the following is probably 2 times the size of an int:

```
struct st_type2 {int a; int b;} var2;
```

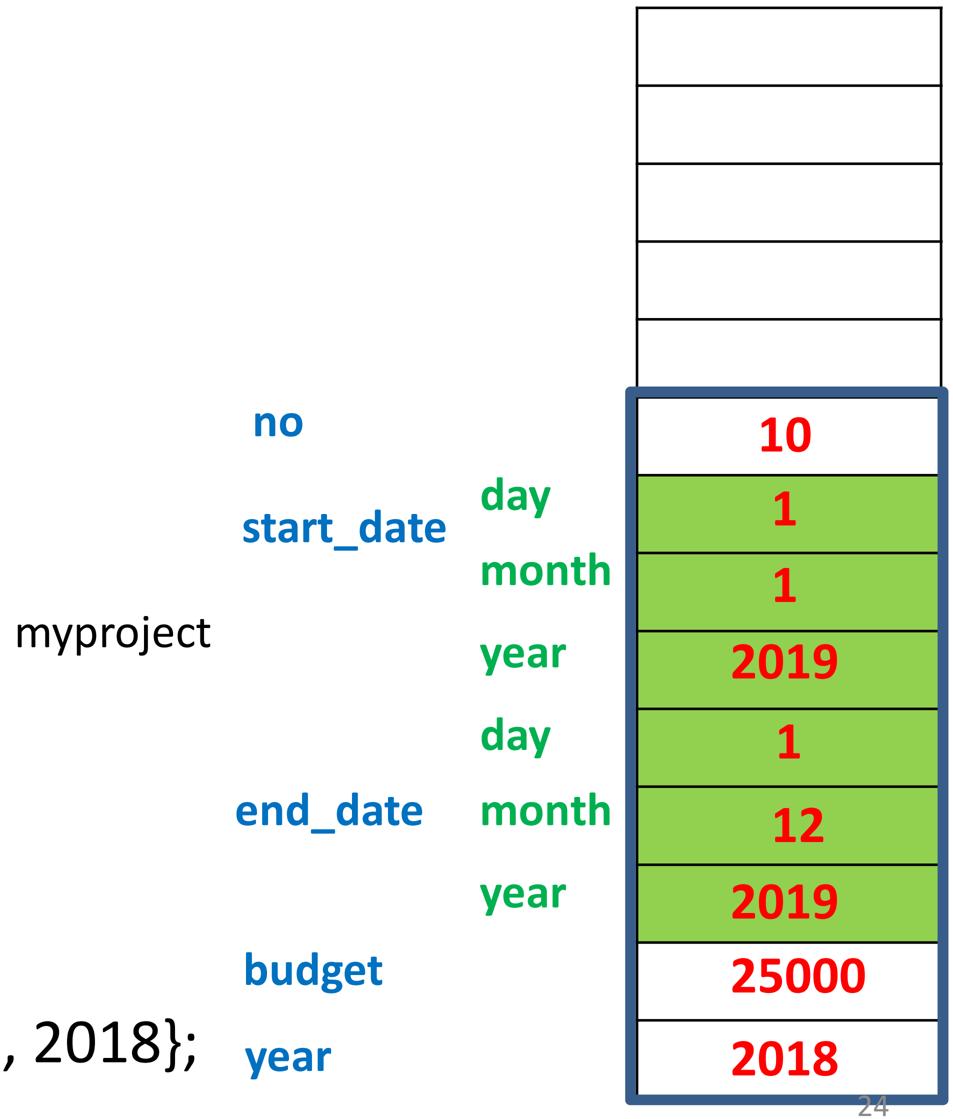
# Nested structures

- You can use one structure within another:

```
struct date
 { int day, month, year; };
struct project
 { int no;
   struct date start_date;
   struct date end_date;
   float budget;
   int year; };
struct project myproject = {10, {1,1,2019}, {1,12,2019}, 25000, 2018};
```

# Nested structures

```
struct date
  { int day, month, year; };
struct project
  { int no;
     struct date start_date;
     struct date end_date;
     float budget;
     int year; };
struct project myproject =
{10, {1,1,2019}, {1,12,2019}, 25000, 2018};
```



# Nested structures

- No limit on the depth of nesting!
- A member inside a nested structure can be accessed by **repeatedly applying** the dot operator

```
struct project myproject = {10, {1,1,2019}, {1,12,2019}, 25000, 2018}

myproject.no = 500;

myproject.start_date.month = 6;
struct date new_end_date = {1, 12, 2021};

myproject.end_date = new_end_date;
```

# Pointers to Structures

- A structure cannot be nested within itself (but it is possible to have a pointer to itself – LATER)
- A pointer to a structure var is created in the same way to a simple data type:

```
struct date  carnival, *mardigras;

mardigras = &carnival;
```

Pointer to a variable of type struct date

# Pointers to Structures

struct date

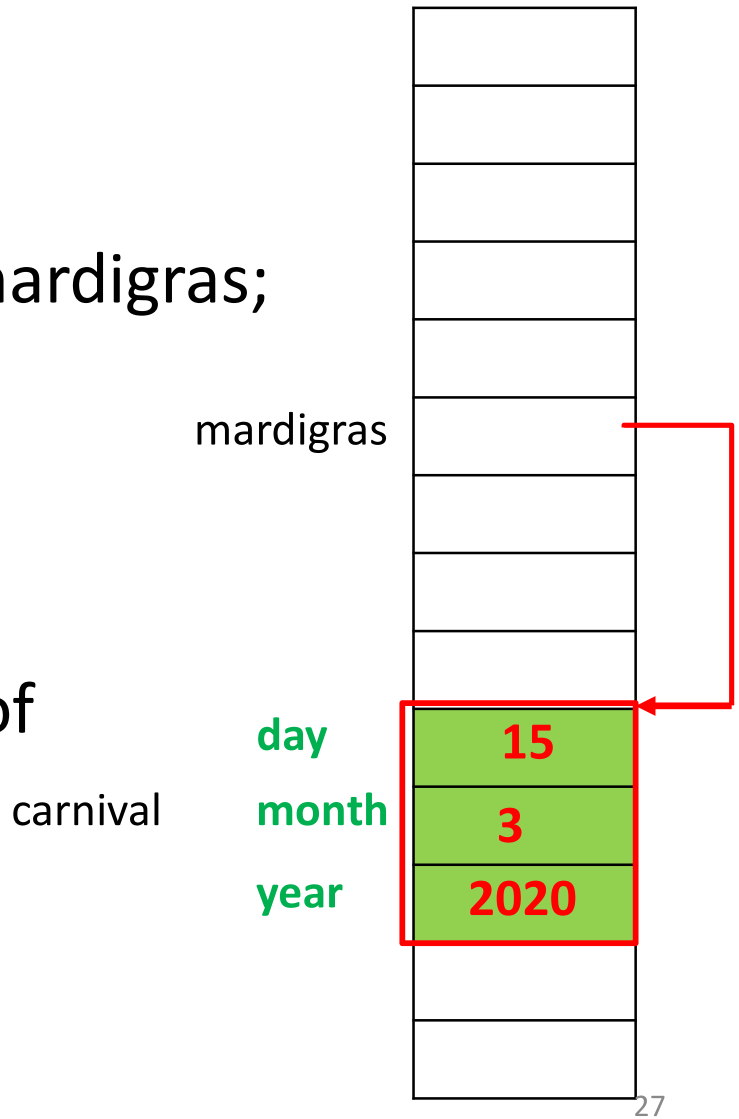
{ int **day, month, year**; };

struct date  carnival={15,3,2020}, *mardigras;

mardigras = &carnival;

**(***mardigras**)**.day  returns the value of

carnival.day



# Pointers to Structures

- A structure cannot be nested within itself (but it is possible to have a pointer to itself – LATER)
- A pointer to a structure var is created in the same way to a simple data type:

```
struct date  carnival, *mardigras;
```

Pointer to a variable of type struct date

```
mardigras = &carnival;
```

(*mardigras).day  returns the value of carnival.day

Parantheses are needed as dot op has higher precedence then deferenceing op *. Without parantheses, meaning is *(mardigras.day), which is an error!

# Pointers to Structures

(*mardigras).day

Arrow operator: A special operator for accessing members of a structure variable pointed to by a pointer!

**pointer_name → member_name**

Instead of (*mardigras).day we better write:

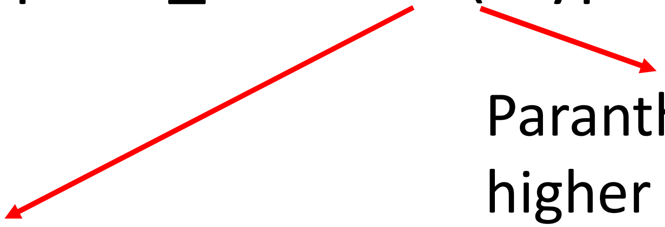
mardigras→day

→ has the same predence as dot operator and left-assoc.

# Pointers to Structures

- A pointer can point into the middle of a structure:

```
struct date  *complete_date;
complete_date = &(myproject.end_date);
```



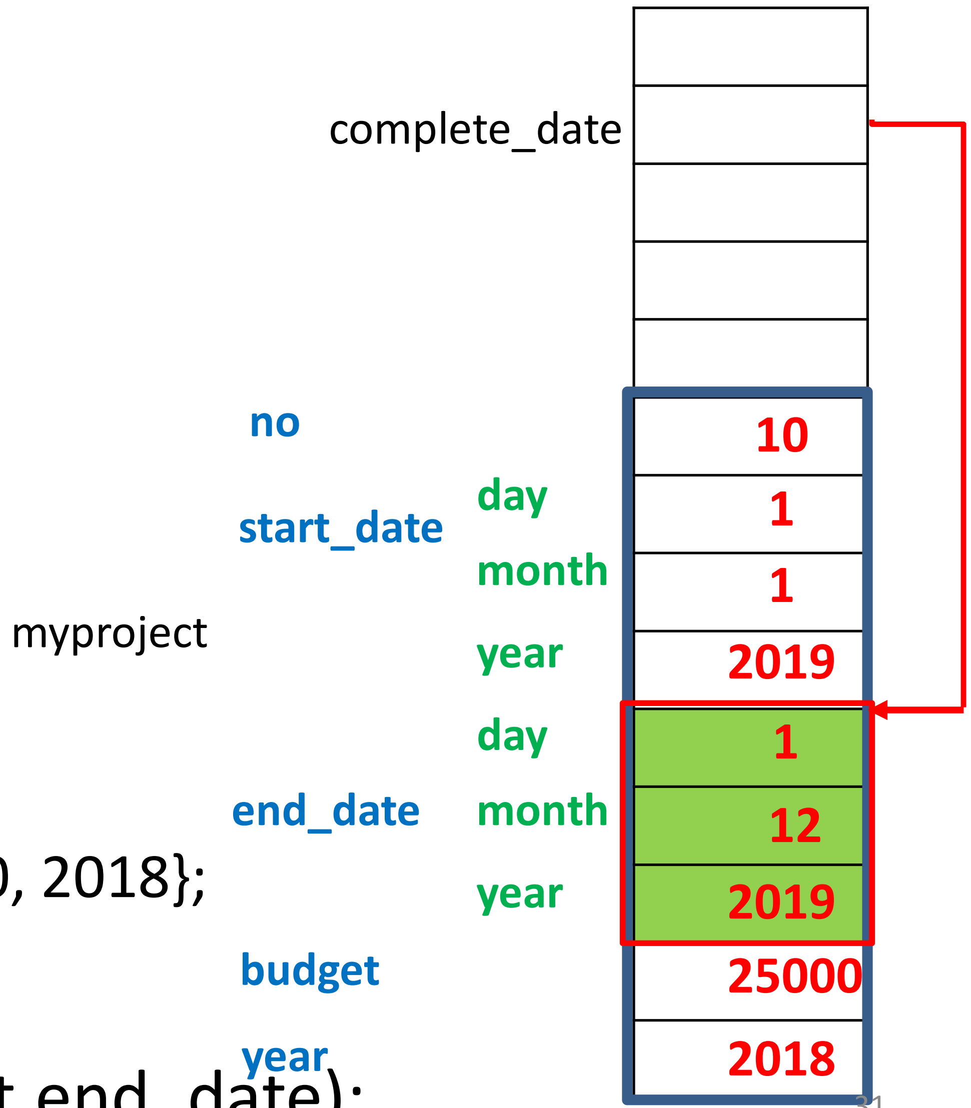
Parantheses **not** needed; dot operator has higher precedence

It is allowed to take the addresses of member vars of a structure variable

**See the example at p.285 of your textbook!**

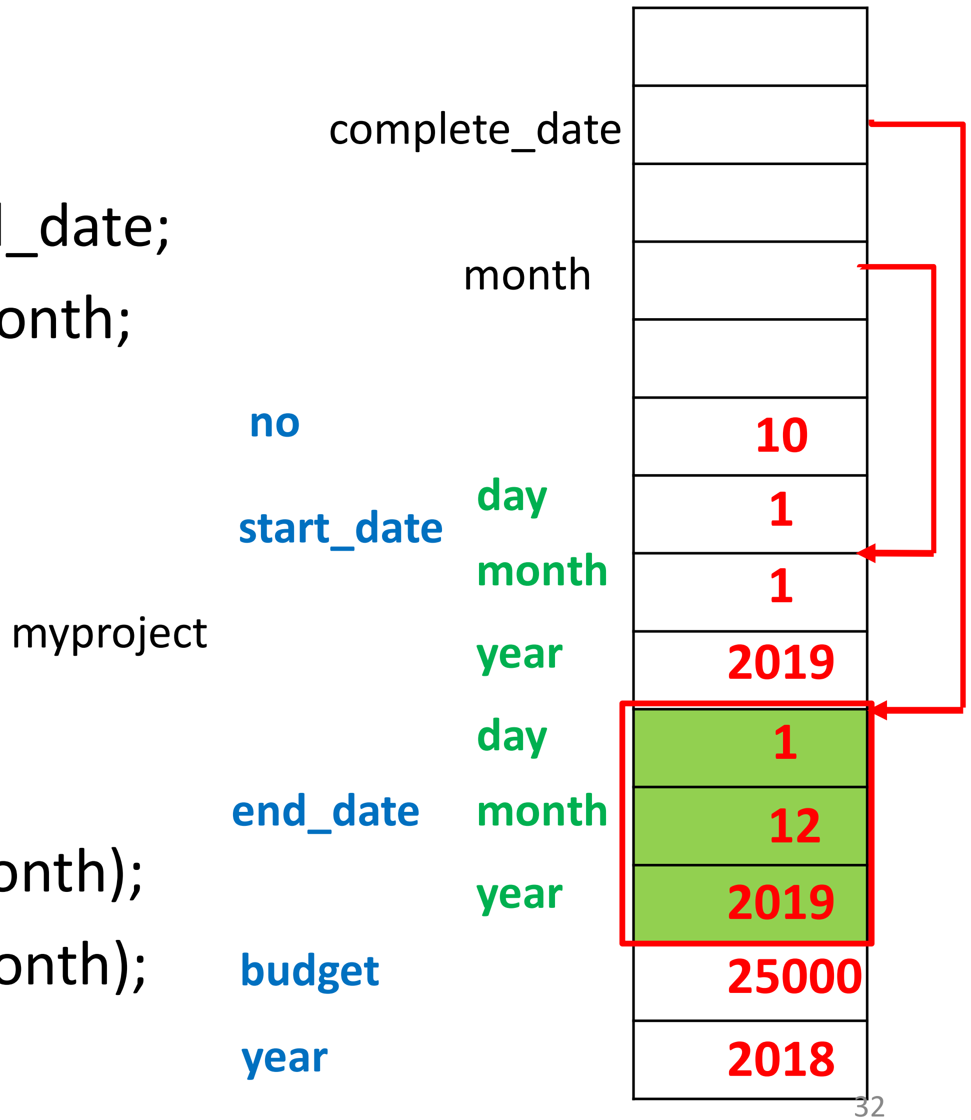
# Pointers to Structures

```
struct date
  { int day, month, year; } ;
struct project
  { int no;
     struct date start_date;
     struct date end_date;
     float budget;
     int year; };
struct project myproject =
{10, {1,1,2019}, {1,12,2019}, 25000, 2018};
struct date  *complete_date;
complete_date = &(myproject.end_date);
```



# Pointers to Structures

```
int *month;
struct date  *complete_date;
complete_date = &myproject.end_date;
month=&myproject.start_date.month;


/* print start month using ptr*/
printf("%d", *month);
/*print end month*/
printf("%d", (*complete_date).month);
printf("%d", complete_date → month);
```



# Pointers to Structures

- Using pointers to structures is better/faster than using structures directly especially in the case of function calls (for **passing parameters** or **returning values**).
- Using pointers to structures allows us to create **sophisticated data structures**.

# Reminder

| Operator | Type | Associativity |
|---|---|---|
| **Fucntion call: ()** **Array subscript: []**<br>**Dot operator: .** **Arrow operator: →** | | **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 |

# Structures & Functions

## Scope of a structure definition

The scoping rules for variable names apply also for struct definitions:

- With a local struct definition, only local variables of struct type can be defined.
- With a global definition (with a tag), you can have global & local variables of this struct type

# Example

```
struct global_date {int day, month, year; };
```

```
void f()
{ struct date {int d, m, y; };
  struct date valid = {01, 01, 2018}; /* var decl & init*/
  struct global_date next = {02, 01, 2018};  }
```

```
struct global_date my_date = {20, 2, 2020};
```

```
void g()
{ struct date today; /* illegal!!! */
  struct global_date tomorrow;  }
```

# Structures as Function Arg.s

- You can pass structures as parameters to a function, in 3 ways:

# Structures as Function Arg. (1)

1) Supply structure members as arguments in a function call separately; i.e.,treat as non-structs

```
struct point { float x, y; };
struct circle { float r; struct point o; };
```

Circle info

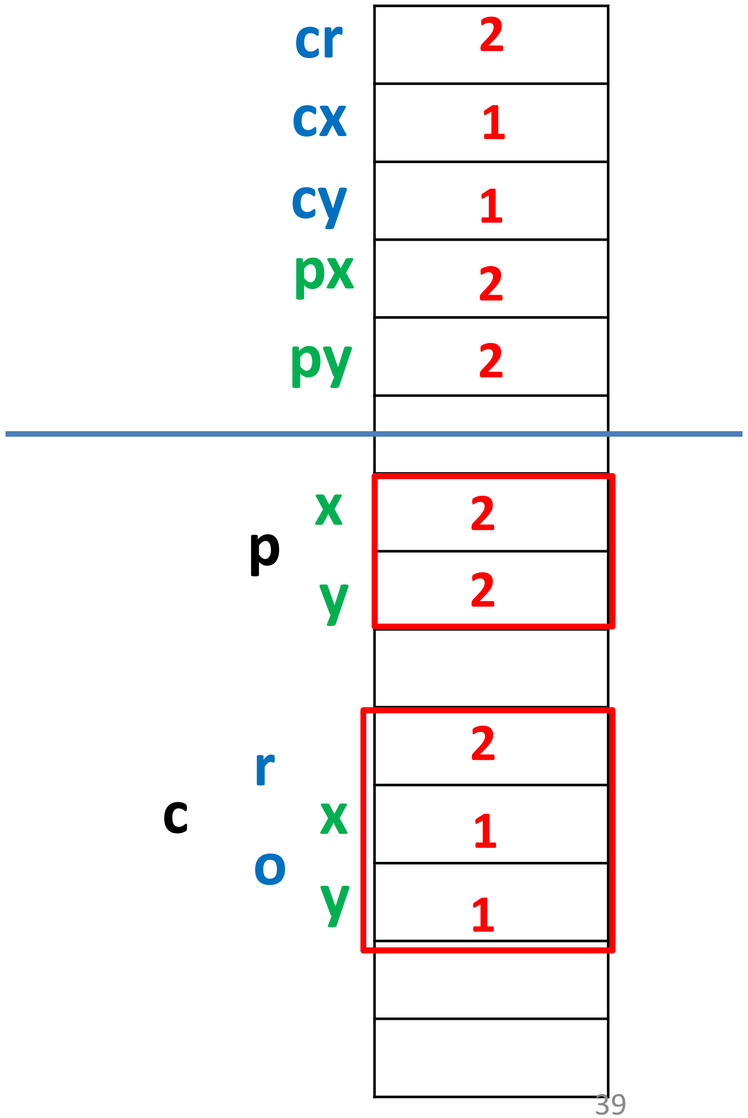
Point info

```
int contains (float cr, float cx, float cy, float px, float py)
{ return sqr(cx-px) + sqr(cy-py) > sqr(cr) ? 0 : 1 ; }
```

**In main:**

```
struct circle c = {2, {1 ,1} }; struct point p = {2,2};
contains(c.r, c.o.x., c.o.y, p.x, p.y); // function call
```

```
int contains (float cr, float cx, float cy, float px, float py)
{ return sqr(cx-px) + sqr(cy-py) > sqr(cr) ? 0 : 1 ; }
```



```
int main(void) {
struct circle c = {2, {1 ,1} };
struct point p = {2,2};
contains(c.r, c.o.x., c.o.y, p.x, p.y);  }
```

# Structures as Function Arg. (2)

2) Pass the complete structure by simply providing the name of the structure var as the argument in the function call

int contains (struct circle c, struct point p)

~~{ return sqr(cx-px) + sqr(cy-py) > sqr(cr) **? 0 : 1** ; }~~

{ return sqr(c.o.x-p.x) + sqr(c.o.y-p.y) > sqr(c.r) **? 0 : 1** ; }

**In main:**

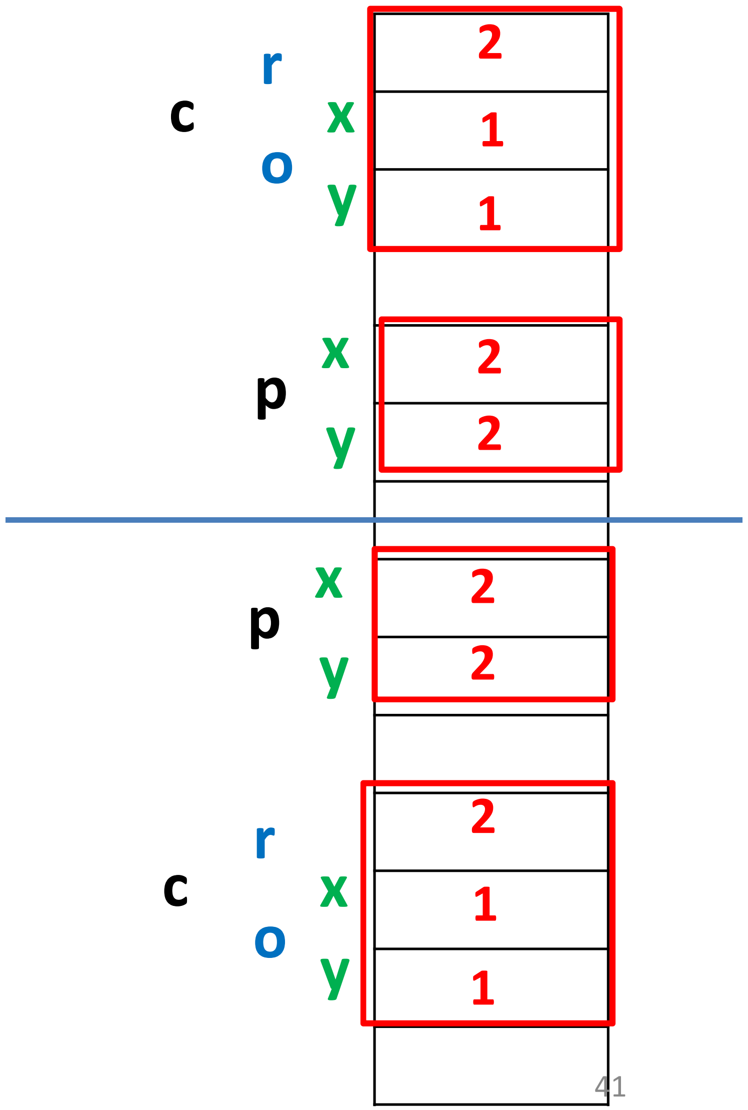
struct circle c = {2, {1 ,1} }; struct point p = {2,2};

~~contains(c.r, c.o.x., c.o.y, p.x, p.y); // function call~~

contains(c, p);

```
int contains (struct circle c, struct point p)
{ return sqr(c.o.x-p.x) + sqr(c.o.y-p.y) > sqr(c.r) ? 0 : 1 ; }
```

```
int main(void) {
struct circle c = {2, {1 ,1} };
struct point p = {2,2};
contains(c, p);}
```



# Structures as Function Arg. (2)

2) Pass the complete structure by simply providing the name of the structure var as the argument in the function call

Unlike array names, structure names are NOT pointers, and hence, they're **passed-by-value**

- When a struct name is provided as argument, entire struct is copied to the called function (and changes are **not** reflected to calling func)
- If there is an array in the struct, it is also copied!

# Structures as Function Arg. (3)

3) Pass a pointer to the structure variable as the argument in the function call

int contains (struct circle *c, struct point *p)

~~{ return sqr(c.o.x-p.x) + sqr(c.o.y-py) > sqr(c.r) **? 0 : 1** ; }~~

{ return sqr(c→o.x - p→x) + sqr(c→o.y - p→y) > sqr(c→r)
**? 0 : 1** ; }

**In main:**

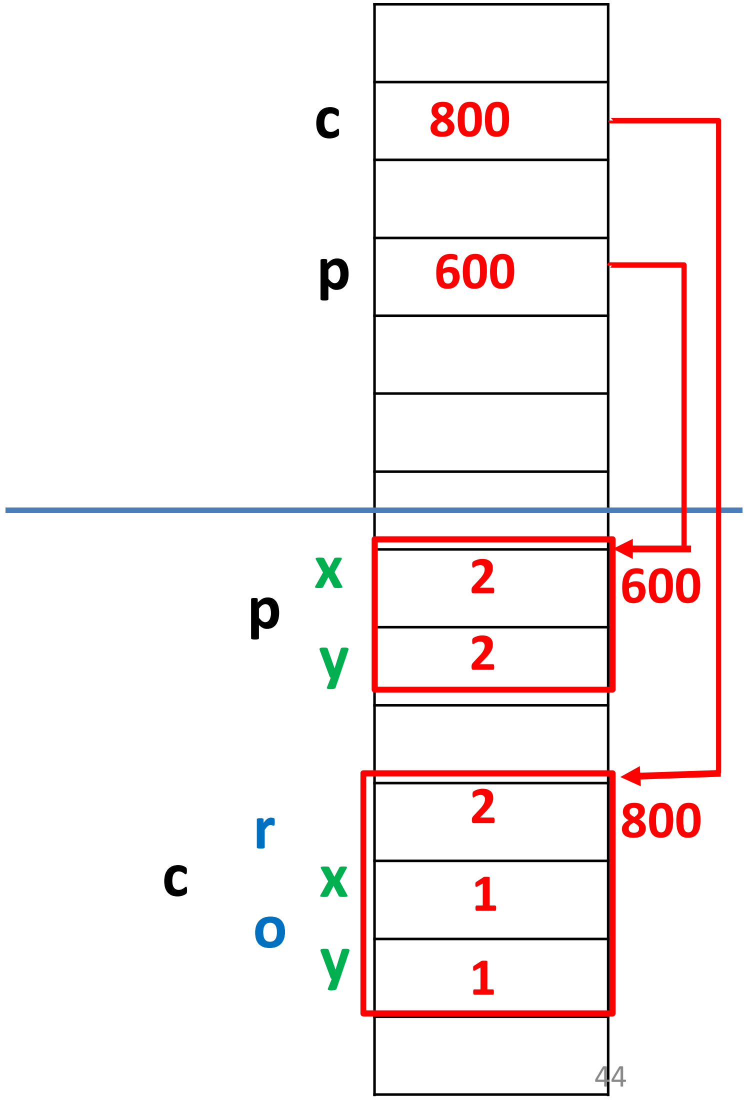
struct circle c = {2, {1 ,1} }; struct point p = {2,2};

~~contains(c.r, c.o.x., c.o.y, p.x, p.y); // function call~~

~~contains(c, p);~~

contains(&c, &p); // changes to args made in func are visible after returning to caller

```
int contains (struct circle *c, struct point *p)
{ return sqr(c→o.x - p→x) + sqr(c→o.y - p→y) > sqr(c→r)
        ? 0 : 1 ; }
```



```
int main(void) {
struct circle c = {2, {1 ,1} };
struct point p = {2,2};
contains(&c, &p);}
```

# Structures as Func. Values

- Structures may be returned as function values

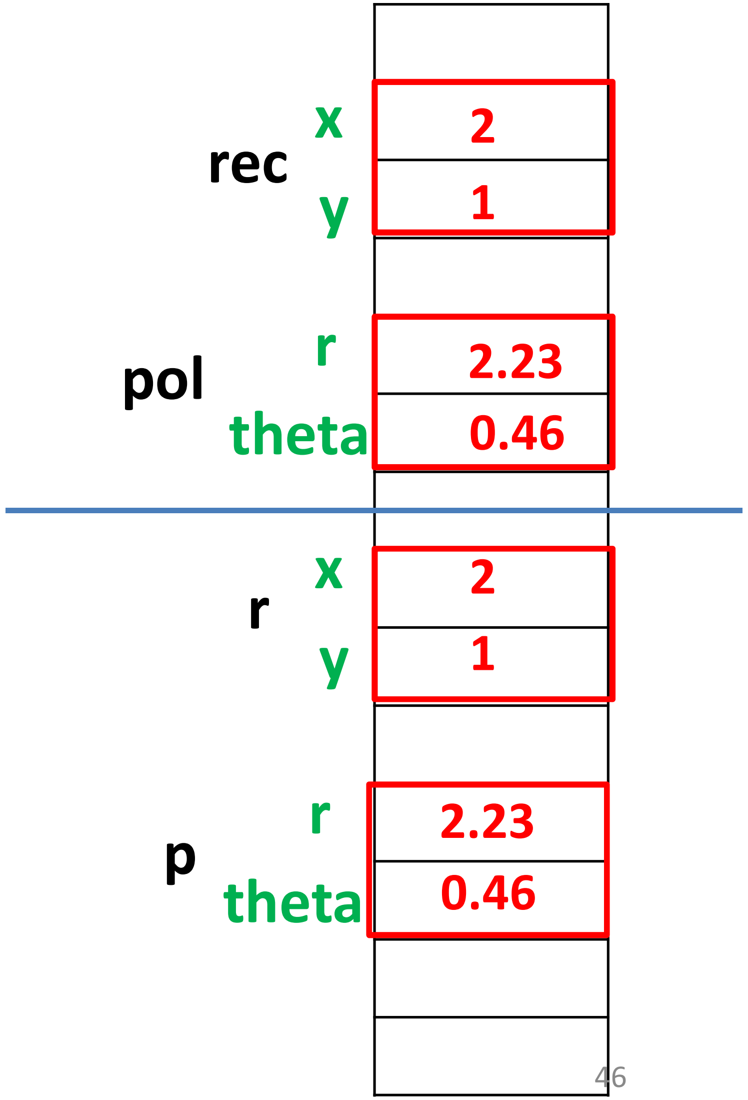
```
struct rectangular { float x, y; };
struct polar { float r, theta; };
struct polar convert (struct rectangular rec)
{ struct polar pol;
  if (rec.x == 0 && rec.y == 0)
     pol.r = pol.theta = 0;
  else
  { pol.r= sqrt(rec.x * rec.x +…);
    pol.theta = …; }
  return pol;
}
```

**In main:**

```
struct rectangular r={2 ,1};
struct polar p;
p = convert(r);
```

```
struct polar convert (struct rectangular rec)
{ struct polar pol;
   if (rec.x == 0 && rec.y == 0)
       pol.r = pol.theta = 0;
   else
   { pol.r= sqrt(rec.x * rec.x +…);
     pol.theta = …; }
   return pol;  }
int main(void) {
struct rectangular r={2 ,1};
struct polar p;
p = ~~convert(r)~~; }
```

Substitute with the returned value



# Structures as Func. Values

- A func may return a pointer to the structure

```
struct rectangular { float x, y; };
struct polar { float r, theta; };
struct polar * convert (struct rectangular rec)
{ struct polar *p;
  p = (struct polar *) malloc(sizeof(struct polar));
  if (p)
  { if (rec.x == 0 && rec.y == 0)
        p→r = p→theta = 0;
     else … }
  return p;
}
```

**In main:**

```
struct rectangular r={2 ,1};
struct polar *polp;
polp = convert(r);
```

```
struct polar * convert (struct rectangular rec)
{ struct polar *p;
  p = (struct polar *) malloc(sizeof(struct polar));
  if (p)
  { if (rec.x == 0 && rec.y == 0)
       p→r = p→theta = 0;
     else … }
  return p; }
```

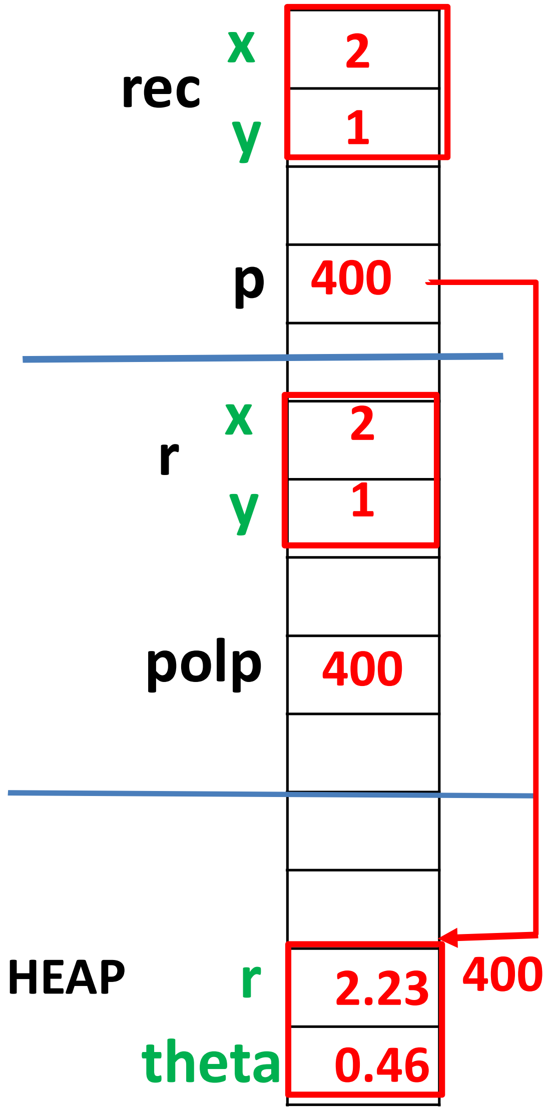
**int main(void) {**

struct rectangular r={2 ,1};

struct polar *polp;

polp = ~~convert(r)~~; **}**

Substitute with the returned value



# How about this version?

**WRONG!!!!!**

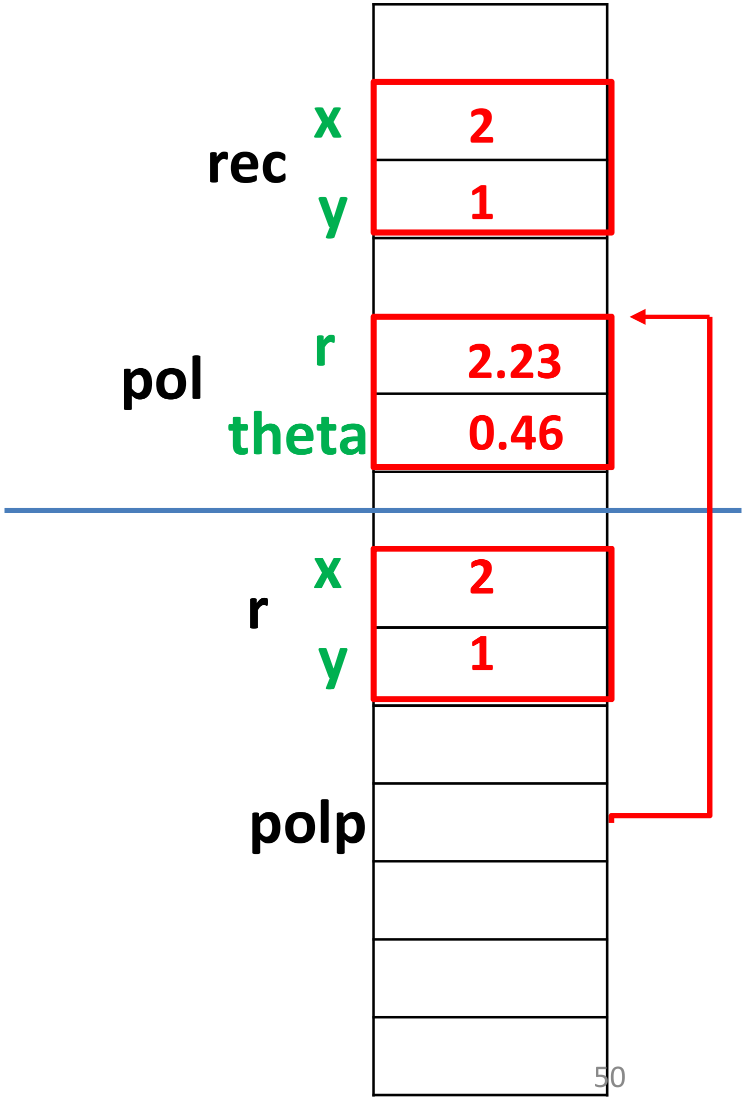
```
struct rectangular { float x, y; };
struct polar { float r, theta; };
struct polar * convert (struct rectangular rec)
{ struct polar pol;
   if (rec.x == 0 && rec.y == 0)
       pol.r = pol.theta = 0;
   else
   { pol.r= sqrt(rec.x * rec.x +…);
     pol.theta = …; }
   return &pol;
}
```

**In main:**

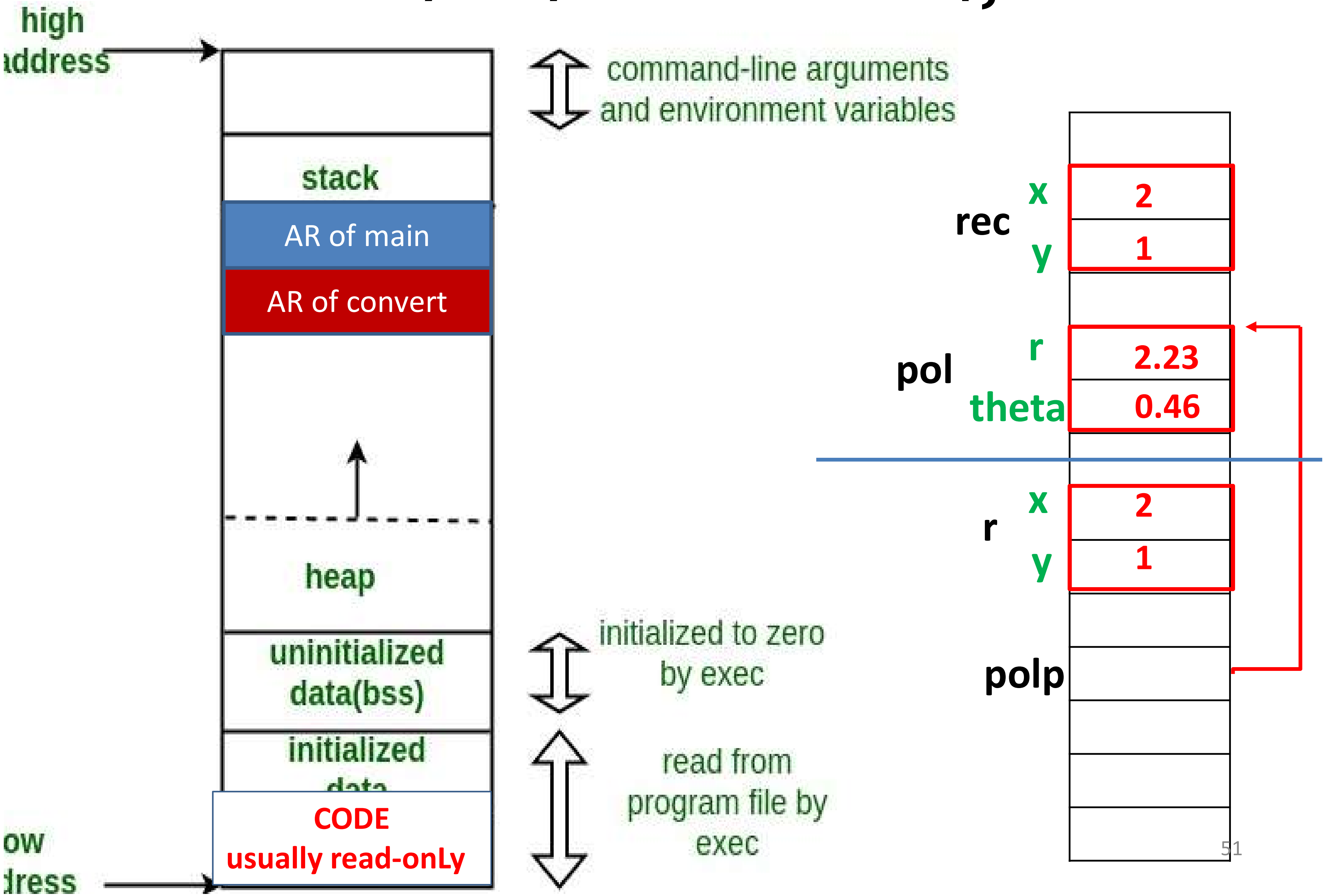
```
struct rectangular r={2 ,1};
struct polar *polp;
polp = convert(r);
```

```
struct polar * convert (struct rectangular rec)
{ struct polar pol;
   if (rec.x == 0 && rec.y == 0)
       pol.r = pol.theta = 0;
   else
   { pol.r= sqrt(rec.x * rec.x +…);
     pol.theta = …; }
   return &pol;  }
int main(void) {
struct rectangular r={2 ,1};
struct polar *polp;
polp = ~~convert(r)~~; }
```

Substitute with the returned value



# Memory Layout of C Programs



# Arrays & Structures

- Arrays and sturctures can be freely intermixed to create:
  - Arrays of structures
  - Structures containing arrays
  - Arrays of structures containing arrays…

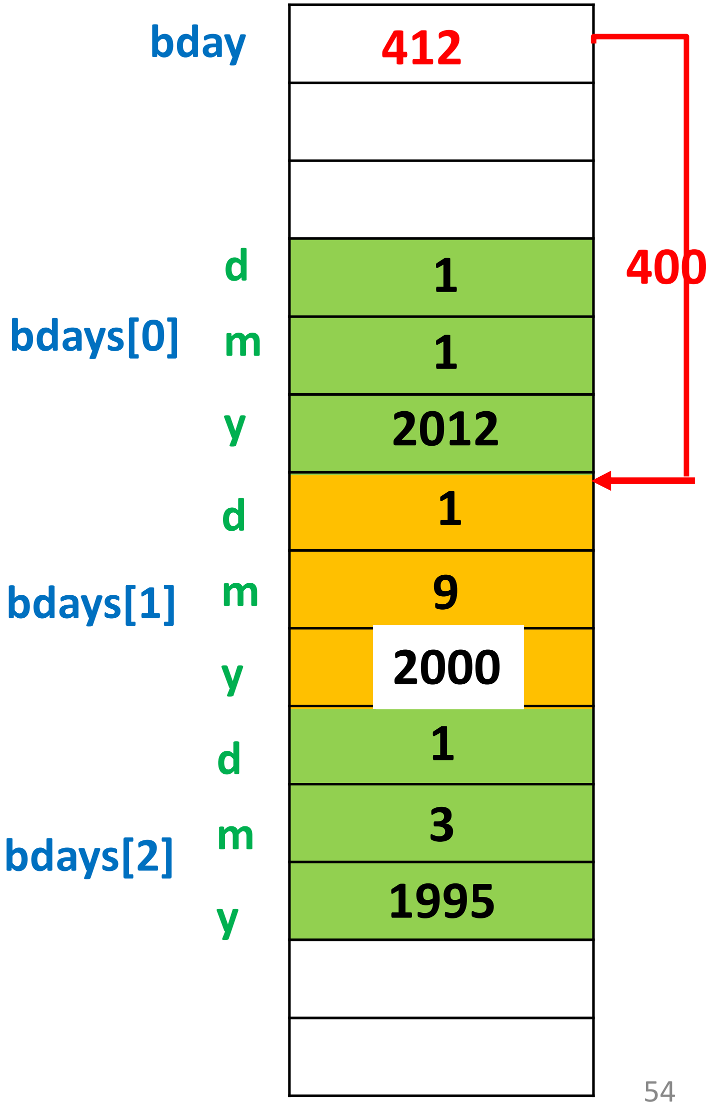
# Arrays of structures

- Used when a large no of similar records are required to be processed together

```
struct date { int d, m, y; };
struct date bdays[3] = {{1, 1, 2012}, {1,9,1980}, {1,3,1995}};
```

# Arrays of structures

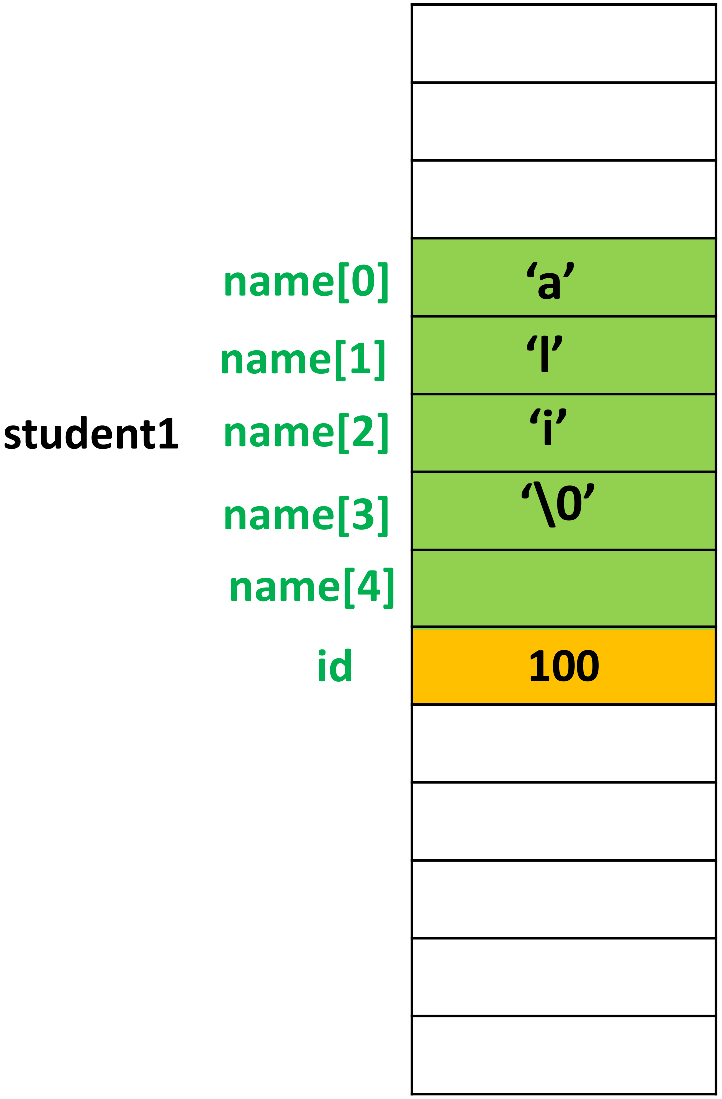
```
struct date { int d, m, y; };
struct date bdays[3] = {{1, 1, 2012},
   {1,9,1980}, {1,3,1995}};
struct date *bday;


bdays[1].y = 1990;
bday = &bdays[1];
bday → y = 2000;
```



# Structures containing arrays

- A struct can contain an array

```
struct person
{  char name[5];
   int id;
} student1 = {"ali" , 100};
//or {{‘a’, ’l’, ‘i’, ‘\0’}, 100};
```



student1

name[0] ‘a’

name[1] ‘l’

name[2] ‘i’

name[3] ‘\0’

name[4]

id 100

# Structures containing arrays

- A struct can contain an array

```
struct person
{  char name[5];
   int id;
} student1 = {"ali" , 100}; //or {{‘a’, ’l’, ‘i’, ‘\0’}, 100};
// btw, is this string modifiable or not?
// how do we assign name to "veli" ?
strcpy(student1.name, "veli");
//OR student1.name[0]=‘v’; student1.name[1]=‘e’; ..
printf("%c", student1.name[0]);
```

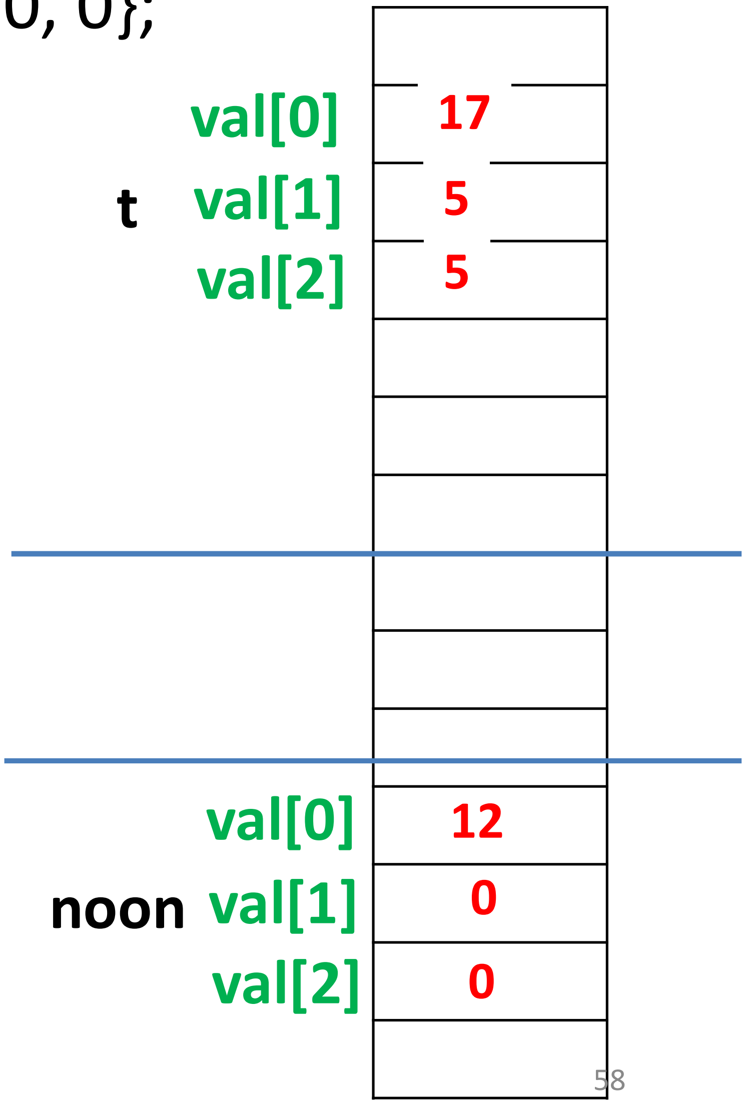
# Structures containing arrays

- When a struct containing an array is passed as an argument to a function, the member array is **passed-by-value**! (even when it is the only member)

```
struct time { int val[3]; } noon = {12, 0, 0};
void advanceTime(struct time t)
{ int i;
  for (i=0; i<3; i++)
          t.val[i] += 5; }
int main
{ advanceTime(noon);
  printf("%d",  noon.val[0]);}
```

# Structures containing arrays

```
struct time { int val[3]; } noon = {12, 0, 0};
void advanceTime(struct time t)
{ int i;
  for (i=0; i<3; i++)
        t.val[i] += 5; }
int main
{ advanceTime(noon);
  printf("%d",  noon.val[0]);}
```

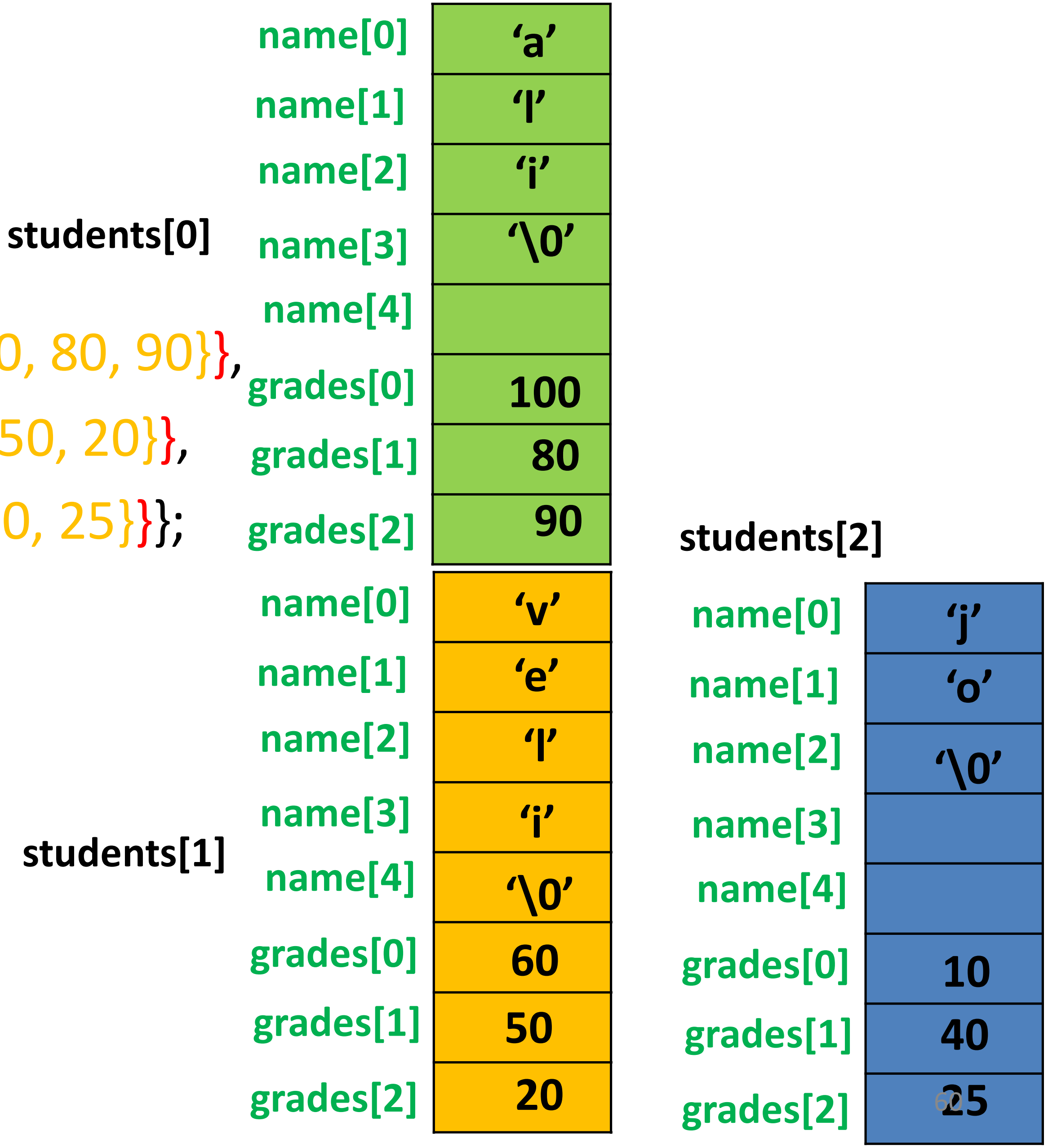


# Array of structs with arrays…

```
struct student
{  char name[5];
   int grades[3];
} students[3]={{"ali" , {100, 80, 90}},
                        {"veli" , {60, 50, 20}}, {"jo" , {10, 40, 25}}};
```

# Array of structs with arrays…

```
struct student
{  char name[5];
   int grades[3];
} students[3]={{"ali" , {100, 80, 90}},
              {"veli" , {60, 50, 20}},
               {"jo" , {10, 40, 25}}};
```



students[0]

| | |
|---|---|
| name[0] | ‘a’ |
| name[1] | ‘l’ |
| name[2] | ‘i’ |
| name[3] | ‘\0’ |
| name[4] | |
| grades[0] | 100 |
| grades[1] | 80 |
| grades[2] | 90 |

students[1]

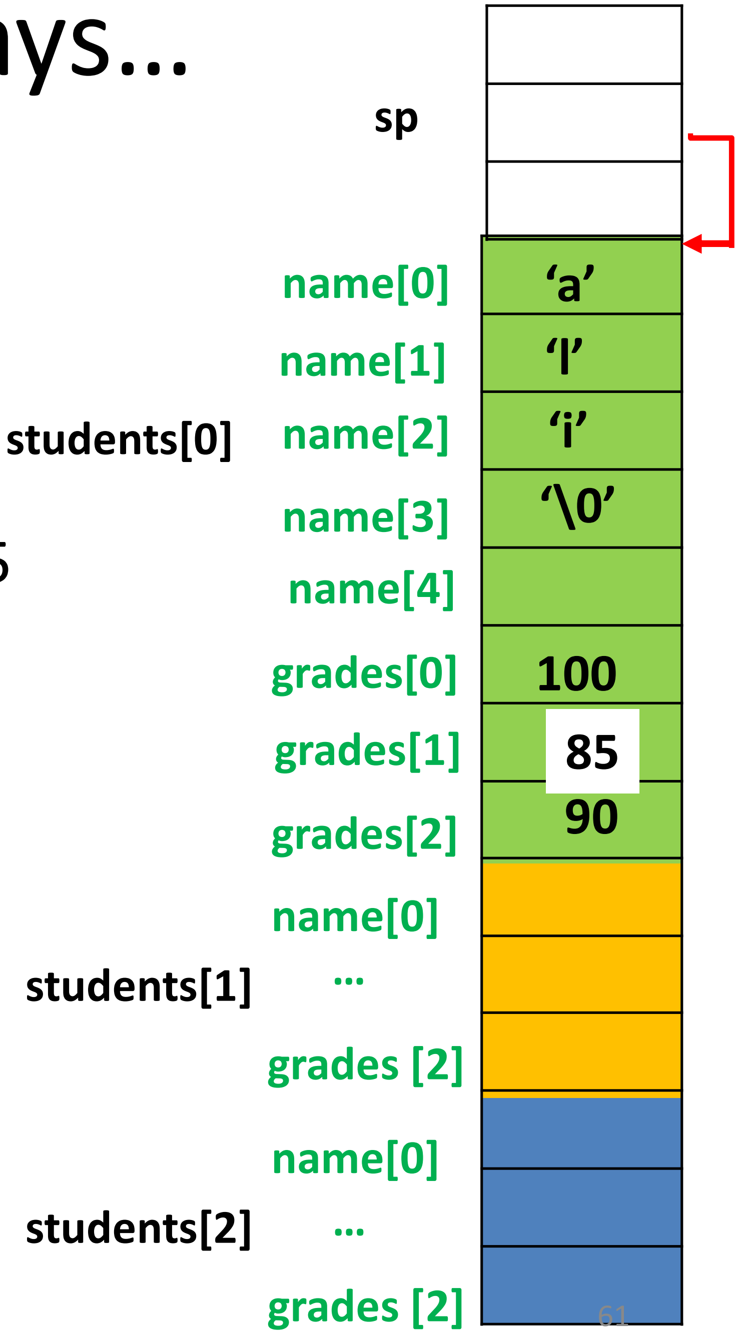
| | |
|---|---|
| name[0] | ‘v’ |
| name[1] | ‘e’ |
| name[2] | ‘l’ |
| name[3] | ‘i’ |
| name[4] | ‘\0’ |
| grades[0] | 60 |
| grades[1] | 50 |
| grades[2] | 20 |

students[2]

| | |
|---|---|
| name[0] | ‘j’ |
| name[1] | ‘o’ |
| name[2] | ‘\0’ |
| name[3] | |
| name[4] | |
| grades[0] | 10 |
| grades[1] | 40 |
| grades[2] | 25 |

# Array of structs with arrays…

```
  struct student
  {  char name[5];
     int grades[3];
  } students[3]={…};
students[0].grades[1] = 85; // 80 becomes 85
struct student *sp;
sp = &students[0]; // equivalent to?
// print first stu name’s first char
printf("%c" , sp→ name[0]
// print his full name and second grade,
printf("%s %d" , sp→ name, sp→ grades[1]);
                              *(sp→ grades + 1)
```



# A "ptr to a struct" is the parameter

```
struct student
{  int grade;
   char name[5];  };
```

```
void g(struct student *p)
{  p → grade = 0;
   strcpy(p → name, "TONY");}
```

```
int main()
{ struct student *sp;
  sp = (struct student *) malloc (sizeof(struct student));
  sp -> grade = 75;
  strcpy(sp → name, "JANE");
  printf("see %s %d\n", sp → name,sp → grade); // Output?
  g(sp);
 printf("see %s %d\n", sp → name,sp → grade); // Output?}
```

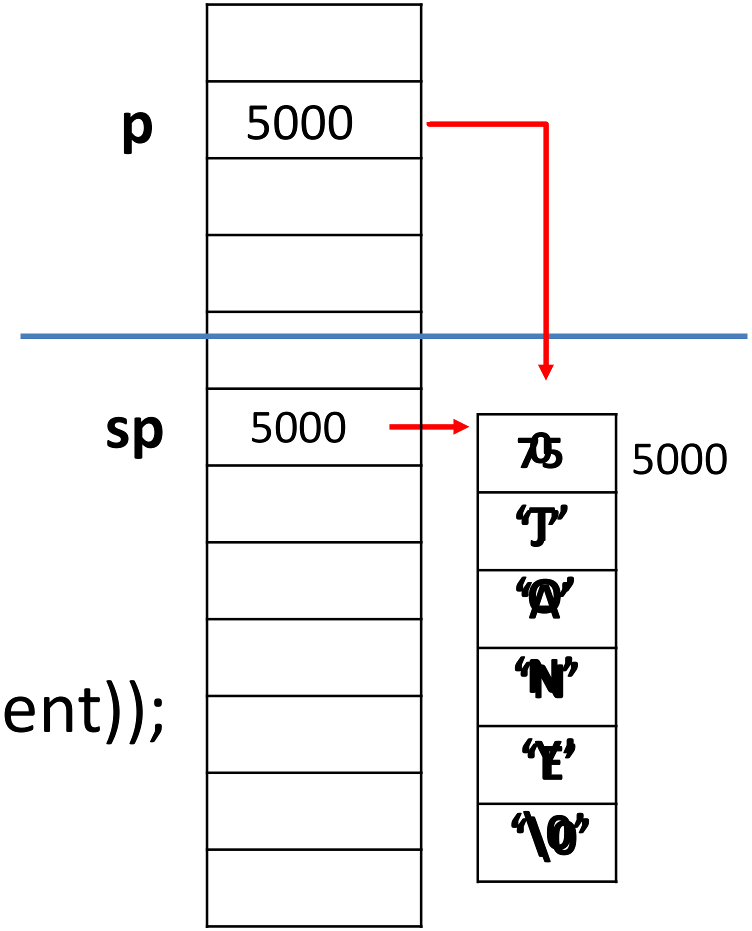
**A- 75 Jane**

**B- 0 Tony**

**C- Compile error**

**D- Run time error**

```
void g(struct student *p)
{  p → grade = 0;
   strcpy(p → name, "TONY");}

int main()
{ struct student *sp;
  sp = (struct student *)
                 malloc (sizeof(struct student));
  sp -> grade = 75;
  strcpy(sp → name, "JANE");
  printf("see %s %d\n", sp → name,sp → grade); // Output?
  g(sp);
 printf("see %s %d\n", sp → name,sp → grade); // Output?}
```



# A "ptr to a struct" is the parameter

```
struct student
{  int grade;
   char name[5];  };
```

```
void g(struct student *p)
{  p = (struct student *) malloc (...));
   p → grade = 0;
   strcpy(p → name, "TONY");}
```

```
int main()
{ struct student *sp;
  sp = (struct student *) malloc (sizeof(struct student));
  sp → grade = 75;
  strcpy(sp → name, "JANE");
  printf("see %s %d\n", sp → name,sp → grade); // Output?
  g(sp);
 printf("see %s %d\n", sp → name,sp → grade); // Output?
```

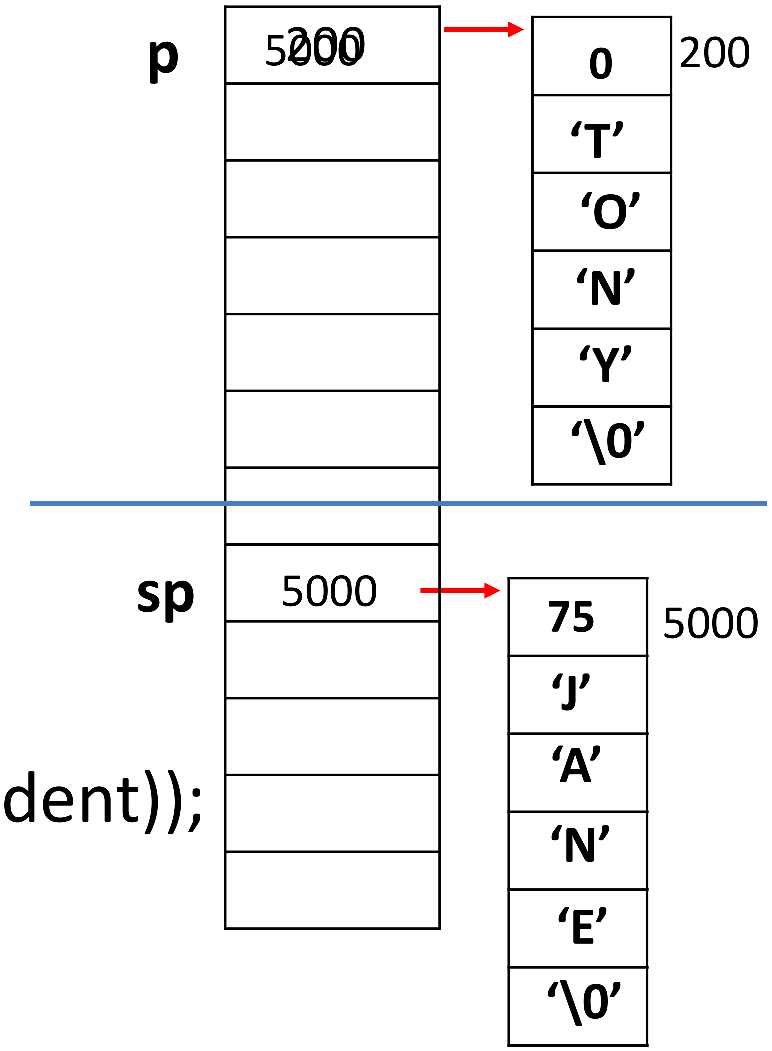
**A- 75 Jane**

**B- 0 Tony**

**C- Compile error**

**D- Run time erro**

```
void g(struct student *p)
{  p = (struct student *) malloc (…));
     p → grade = 0;
     strcpy(p → name, "TONY");}
int main()
{ struct student *sp;
   sp = (struct student *)
                    malloc (sizeof(struct student));
   sp -> grade = 75;
   strcpy(sp → name, "JANE");
   printf("see %s %d\n", sp → name,sp → grade); // Output?
   g(sp);
  printf("see %s %d\n", sp → name,sp → grade); // Output?}
```



# A "ptr to ptr to a struct" is parameter

```
struct student
{  int grade;
   char name[5];  };
```

```
void f(struct student **p)
{  (*p) → grade = 0;
   strcpy( (*p) → name, "TONY");}
```

```
int main()
{ struct student *sp;
  sp = (struct student *) malloc (sizeof(struct student));
  sp -> grade = 75;
  strcpy(sp → name, "JANE");
  printf("see %s %d\n", sp -> name,sp -> grade); // Output?
  f(&sp);
 printf("see %s %d\n", sp -> name,sp -> grade); // Output? }
```

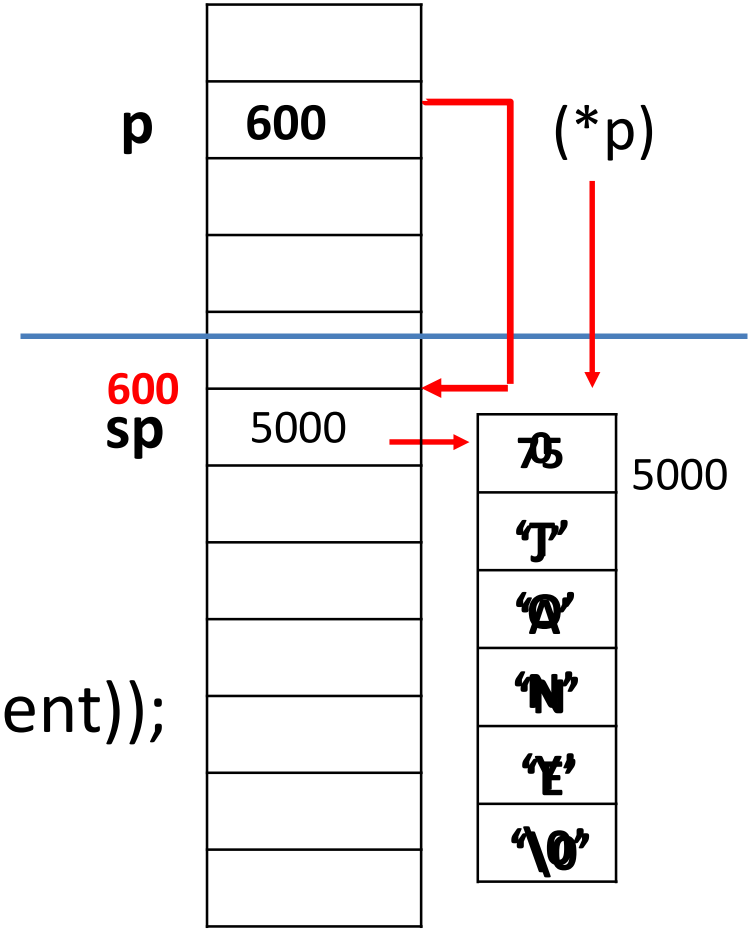
**A- 75 Jane**

**B- 0 Tony**

**C- Compile error**

**D- Run time error**

```
void g(struct student **p)
{(*p) → grade = 0;
   strcpy( (*p) → name, "TONY");}

int main()
{ struct student *sp;
  sp = (struct student *)
                    malloc (sizeof(struct student));
  sp -> grade = 75;
  strcpy(sp → name, "JANE");
  printf("see %s %d\n", sp → name,sp → grade); // Output?
  g(&sp);
 printf("see %s %d\n", sp → name,sp → grade); // Output?}
```



# A "ptr to ptr to a struct" is parameter

```
struct student
{  int grade;
   char name[5];  };
```

```
void f(struct student **p)
{ struct student *fp;
  fp = (struct student *) malloc (...));
  fp → grade = 0;
  strcpy( fp → name, "TONY");
  *p = fp; }
```

```
int main()
{ struct student *sp;
  sp = (struct student *) malloc (sizeof(struct student));
  sp -> grade = 75;
  strcpy(sp → name, "JANE");
  printf("see %s %d\n", sp -> name,sp -> grade); // Output?
  f(&sp);
 printf("see %s %d\n", sp -> name,sp -> grade); // Output? }
```

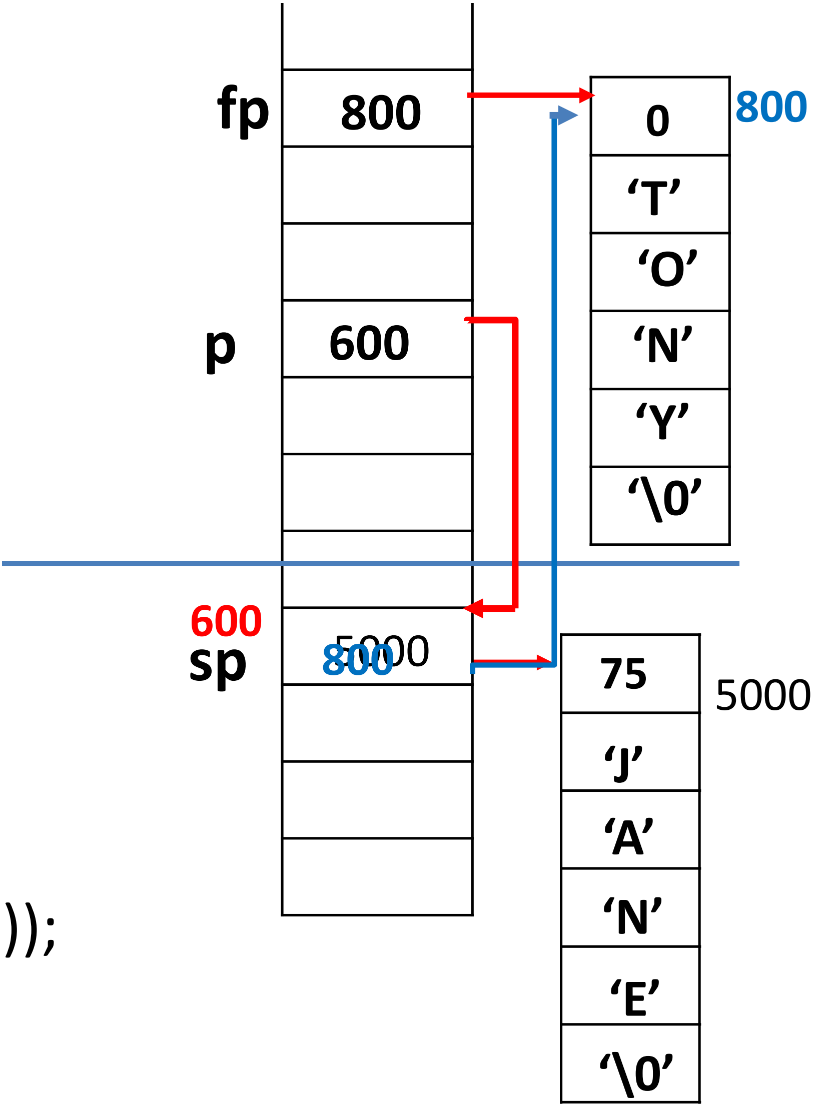
**A- 75 Jane**

**B- 0 Tony**

**C- Compile error**

**D- Run time error**

```
void g(struct student **p)
{ struct student *fp;
  fp = (struct student *) malloc (...));
   fp → grade = 0;
   strcpy( fp → name, "TONY");
 *p = fp; }

int main()
{ struct student *sp;
  sp = (struct student *)
                   malloc (sizeof(struct student));
  sp -> grade = 75;
  strcpy(sp → name, "JANE");
  printf("see %s %d\n", sp → name,sp → grade); // Output?
  g(&sp);
 printf("see %s %d\n", sp → name,sp → grade); // Output?}
```

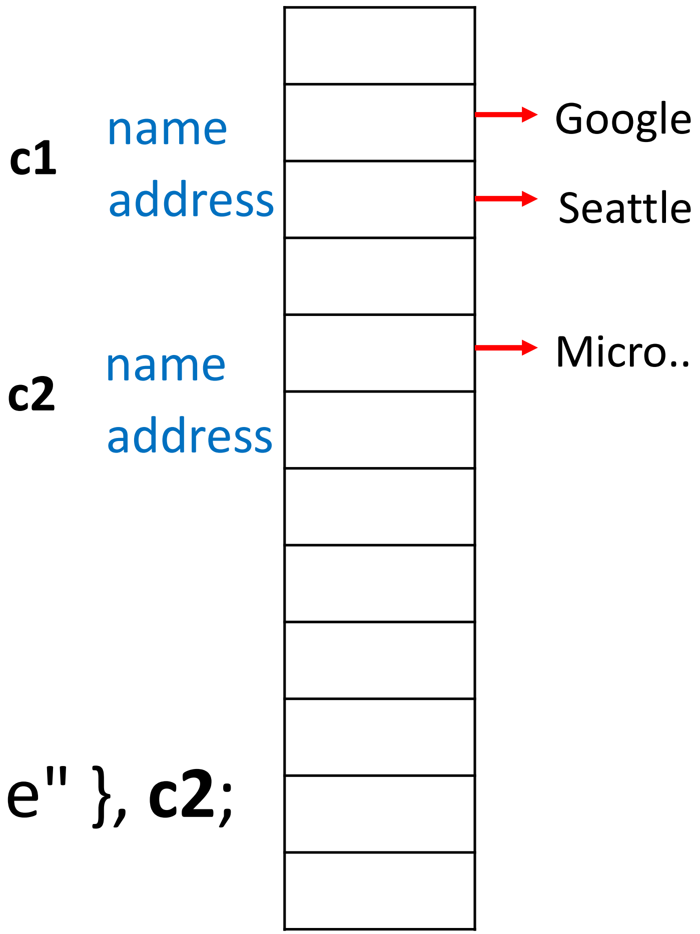


# Structs & Pointers

A struct can contain pointers as member variables

```
struct company
{  char *name;
   char *address;}

struct company c1 = {"Google", "Seattle" }, c2;
c2.name = "Microsoft";
```

// Are these modifiable? How can we make them modifiable?



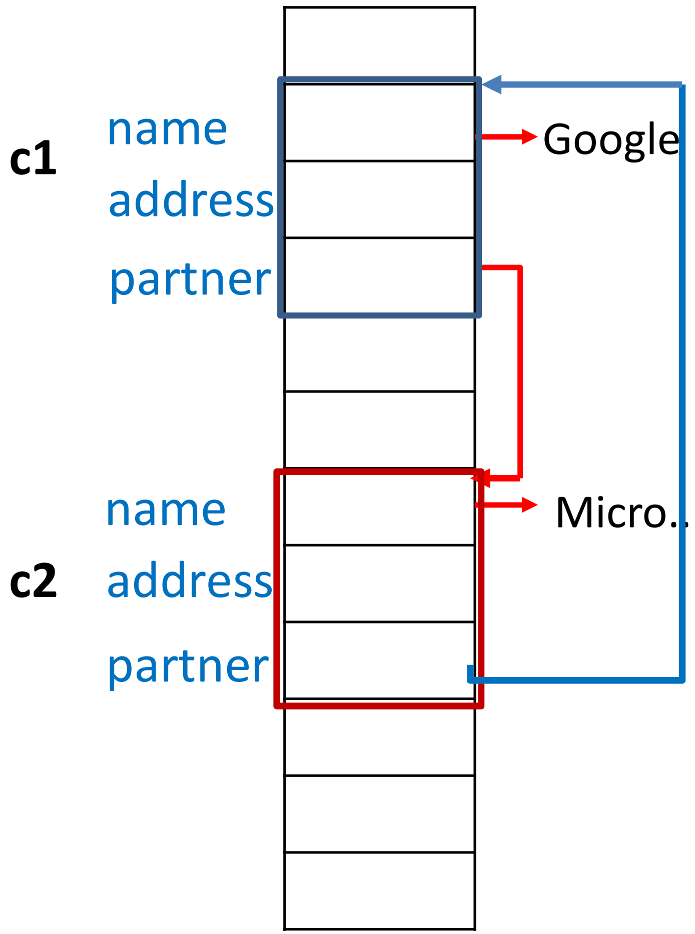
# Structs & Pointers

A struct can **not** be nested within itself, but may contain **pointers** to **structs** of their **own type**!

```
struct company
{  char *name;
   char *address;
   struct company *partner;}
struct company c1, c2;
c1.name = "Google";      c1.partner = &c2;
c2.name = "Microsoft"; c2.partner = &c1;
```

# Structs & Pointers

```
struct company
{  char *name;
   char *address;
   struct company *partner;}
```



```
struct company c1, c2;
c1.name = "Google"; c2.name = "Microsoft";
c1.partner = &c2;
c2.partner = &c1;
```