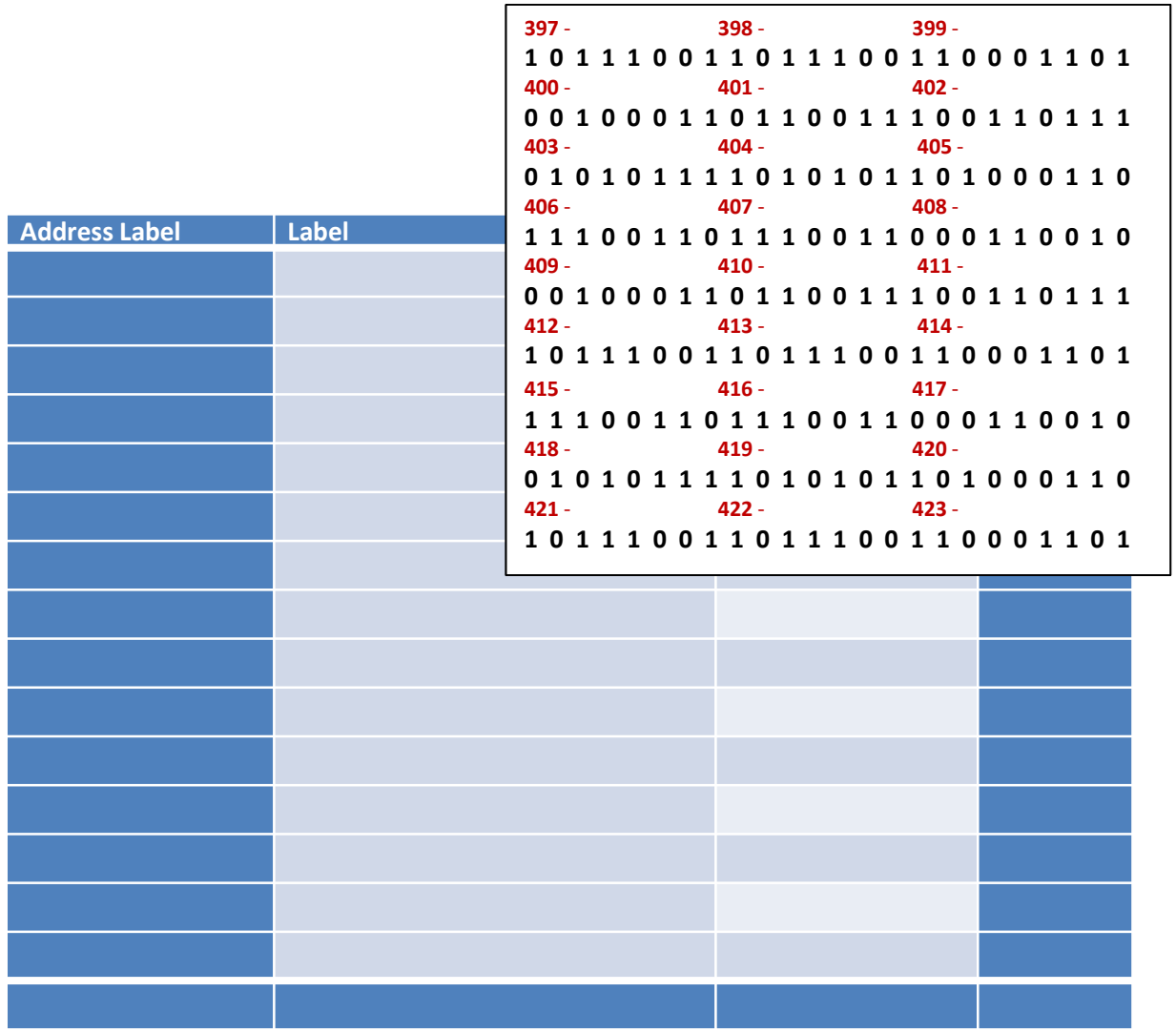
CS 2211
Systems Programming

Part Ten: Structures

MEMORY MAPPING - STRUCTURE



结构体中允许储存不同类型数据.

MEMORY MAPPING - STRUCTURE

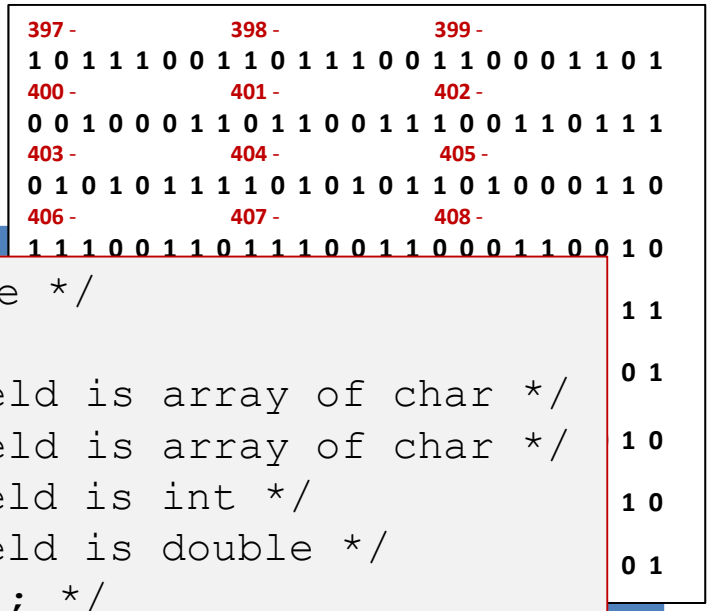
- a construct to group together dissimilar variables under **one name**

标签(tag).

```
/* "person" is name for structure type */
struct person {
        char    first[32];  /* 1st field is array of char */
        char    last[32];   /* 2nd field is array of char */
        int     year;       /* 3rd field is int */
        double  ppg;        /* 4th field is double */
        };                  /* ending ; */
/* means end of structure type definition */
```

member List →

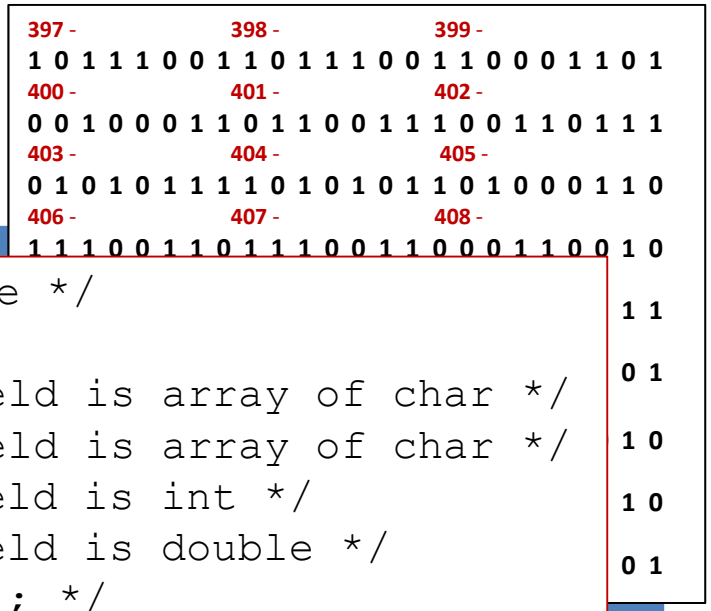
variable List



MEMORY MAPPING - STRUCTURE

- a construct to **group** together
  dissimilar variables under **one name**

非同类



Address Label | Label

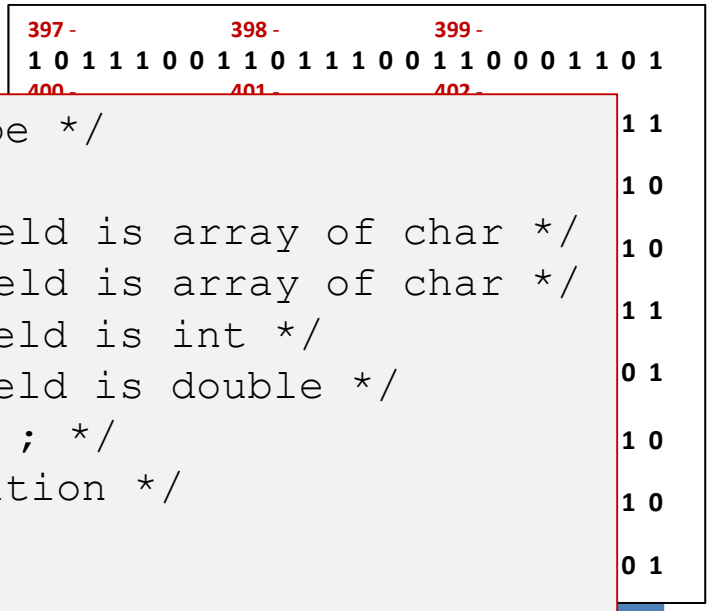
```
/* "person" is name for structure type */
struct person {
        char    first[32];  /* 1st field is array of char */
        char    last[32];   /* 2nd field is array of char */
        int     year;       /* 3rd field is int */
        double  ppg;        /* 4th field is double */
        };                  /* ending ; */
/* means end of structure type definition */
```

NOTE:

This code does NOT create a variable
(similar to Class declaration in OOP)
- grandfather of this concept
- this is ONLY a template for a variable declaration.
- this creates a **USER DEFINED** variable type

MEMORY MAPPING - STRUCTURE

- a construct to **group** together
  dissimilar variables under **one name**




```
/* "person" is name for structure type */
struct person {
        char    first[32];  /* 1st field is array of char */
        char    last[32];   /* 2nd field is array of char */
        int     year;       /* 3rd field is int */
        double  ppg;        /* 4th field is double */
        };                  /* ending ; */
/* means end of structure type definition */

int  i;
struct  person  teacher;
```

type.

creates an integer variable (space to store an integer) **i**
- and -
creates a person variable (space to store two character arrays, and integer and a double) **teacher**.

MEMORY MAPPING - STRUCTURE

```
/* "person" is name for structure type */
struct person {
            char    first[32];  /* 1st field is array of char */
            char    last[32];   /* 2nd field is array of char */
            int     year;               /* 3rd field is int */
            double      ppg;            /* 4th field is double */
            };                          /* ending ; */
/* means end of structure type definition */
```

Address Label	Label	Address	Value

MEMORY MAPPING - STRUCTURE

```
/* "person" is name for structure type */
struct person {
            char    first[32];  /* 1st field is array of char */
            char    last[32];   /* 2nd field is array of char */
            int     year;               /* 3rd field is int */
            double      ppg;            /* 4th field is double */
            };                          /* ending ; */
/* means end of structure type definition */

struct  person  teacher;
```

- memory 400 to 475

- the struct person takes up (requires) 76 bytes of data

Address Label	Label	Address	Value
teacher.first	teacher teacher.first[0]	400 -	
	teacher.first[1]	401 -	
	teacher.first[2]	402 -	
	teacher.first[3]	403 -	
	teacher.first[5]	404 -	
	teacher.first[6] – [31]	405 - 431	
teacher.last	teacher.last[0]	432 -	
	teacher.last[1]	433 -	
	teacher.last[2]	434 -	
	teacher.last[3]	435 -	
	teacher.last[4]	436 -	
	teacher.last[5]	437 -	
	teacher.last[5]	438 -	
	teacher.last[6] – [31]	439 - 463	
	teacher.year	464 - 467	
	teacher.ppg	468 - 475	

note:

```
teacher
teacher.first
&teacher.first[0]
```

all returns the address **400**

MEMORY MAPPING - STRUCTURE

```
/* "person" is name for structure type */
struct person {
            char    first[32];  /* 1st field is array of char */
            char    last[32];   /* 2nd field is array of char */
            int     year;               /* 3rd field is int */
            double      ppg;            /* 4th field is double */
            };                          /* ending ; */
/* means end of structure type definition */

struct  person  teacher;

teacher.year=2005;
teacher.ppg=10.4;
strcpy(teacher.first,"Sam");
strcpy(teacher.last,"Maggs");
```

Address Label	Label	Address	Value
teacher.first	teacher teacher.first[0]	400 -	S
	teacher.first[1]	401 -	a
	teacher.first[2]	402 -	m
	teacher.first[3]	403 -	\0
	teacher.first[5]	404 -	
	teacher.first[6] – [31]	405 - 431	
teacher.last	teacher.last[0]	432 -	M
	teacher.last[1]	433 -	a
	teacher.last[2]	434 -	g
	teacher.last[3]	435 -	g
	teacher.last[4]	436 -	s
	teacher.last[5]	437 -	\0
	teacher.last[5]	438 -	
	teacher.last[6] – [31]	439 - 463	
	teacher.year	464 - 467	2005
	teacher.ppg	468 - 475	10.4

MEMORY MAPPING - STRUCTURE

```
/* "person" is name for structure type */
```

notice memory location 400

```
teacher           // refers to the entire structure starting at 400
teacher.first     // refers to the character array starting at 400
teacher.first[0]  // refers to the single character byte at 400
```

Having a label that refers to the entire structure is used in two ways:

1.) allows assignment between two identical structure types:

```
struct  person    mailman, teacher;
mailman = teacher;
    // a byte by bytes transference of values (including nulls)
```

2.) passing a structure as a parameter to a function.

```
    // allows the easy transfer of large amounts of data
```

	teacher.last[5]	438 -	
	teacher.last[6] – [31]	439 - 463	
	teacher.year	464 - 467	2005
	teacher.ppg	**468 - 475**	**10.4**

MEMORY MAPPING - STRUCTURE

- passing a structure as a parameter to a function.

// allows the easy transfer of large amounts of data

```
DisplayStats(struct person Input) {
        Input.first = "Dunsul";
    }

/* "person" is name for structure type */
struct person {
         char    first[32];  /* 1st field is array of char */
         char    last[32];   /* 2nd field is array of char */
         int     year;         /* 3rd field is int */
         double   ppg;         /* 4th field is double */
         };                    /* ending ; */
/* means end of structure type definition */

struct  person  teacher;

teacher.year=2005;
teacher.ppg=10.4;
strcpy(teacher.first,"Sam");
strcpy(teacher.last,"Maggs");

 DisplayStats(teacher);
 printf("%s\n",teacher.first);
```

Sam

SIDE BAR:
- is the a call by value
-or-
a call by reference ?

MEMORY MAPPING - STRUCTURE

- passing a structure as a parameter to a function.

// allows the easy transfer of large amounts of data

```
DisplayStats(struct person Input) {
        Input.first = "Dunsul";
    }

/* "person" is name for structure type */
struct person {
        char    first[32];  /* 1st field is array of char */
        char    last[32];   /* 2nd field is array of char */
        int     year;         /* 3rd field is int */
        double   ppg;         /* 4th field is double */
        };                    /* ending ; */
/* means end of structure type definition */

struct  person  teacher;

teacher.year=2005;
teacher.ppg=10.4;
strcpy(teacher.first,"Sam");
strcpy(teacher.last,"Maggs");

 DisplayStats(teacher);
 printf("%s\n",teacher.first);
```

SIDE BAR:
- is the a call by value
-or-
a call by reference ?

Label	Address	Value

	teacher.ppg	468 - 475	10.4

MEMORY MAPPING - STRUCTURE

- passing a structure as a parameter to a function.

// allows the easy transfer of large amounts of data

```
DisplayStats(struct person Input) {
        Input.first = "Dunsul";
    }

/* "person" is name for structure type */
struct person {
         char    first[32];  /* 1st field is array of char */
         char    last[32];   /* 2nd field is array of char */
         int     year;         /* 3rd field is int */
         double   ppg;         /* 4th field is double */
         };                    /* ending ; */
/* means end of structure type definition */

struct  person  teacher;

teacher.year=2005;
teacher.ppg=10.4;
strcpy(teacher.first,"Sam");
strcpy(teacher.last,"Maggs");

 DisplayStats(teacher);
 printf("%s\n",teacher.first);
```

SIDE BAR:
- is the a call by value
-or-
a call by reference ?

Label	Address	Value
teacher	400 - 475	
	teacher.ppg	468 - 475	10.4

MEMORY MAPPING - STRUCTURE

- passing a structure as a parameter to a function.

```
// allows the easy transfer of large amounts of data
```

```
DisplayStats(struct person Input) {
        Input.first = "Dunsul";
    }

/* "person" is name for structure type */
struct person {
         char    first[32];  /* 1st field is array of char */
         char    last[32];   /* 2nd field is array of char */
         int     year;         /* 3rd field is int */
         double   ppg;         /* 4th field is double */
         };                    /* ending ; */
/* means end of structure type definition */

struct  person  teacher;

teacher.year=2005;
teacher.ppg=10.4;
strcpy(teacher.first,"Sam");
strcpy(teacher.last,"Maggs");

 DisplayStats(teacher);
 printf("%s\n",teacher.first);
```

SIDE BAR:
- is the a call by value
-or-
a call by reference ?

Label	Address	Value
teacher	400 - 475	[original]
teacher.ppg	468 - 475	10.4

MEMORY MAPPING - STRUCTURE

- passing a structure as a parameter to a function.

```
// allows the easy transfer of large amounts of data
```

```
DisplayStats(struct person Input) {
        Input.first = "Dunsul";
    }
```

Address Label	Label	Address	Value
teacher.first	teacher teacher.first[0]	400 -	S
	teacher.first[1]	401 -	a
	teacher.first[2]	402 -	m
	teacher.first[3]	403 -	\0
	teacher.first[5]	404 -	
	teacher.first[6] – [31]	405 - 431	
teacher.last	teacher.last[0]	432 -	M
	teacher.last[1]	433 -	a
	teacher.last[2]	434 -	g
	teacher.last[3]	435 -	g
	teacher.last[4]	436 -	s
	teacher.last[5]	437 -	\0
	teacher.last[5]	438 -	
	teacher.last[6] – [31]	439 - 463	
	teacher.year	464 - 467	2005
	teacher.ppg	468 - 475	10.4

SIDE BAR:
- is the a call by value
-or-
a call by reference ?

MEMORY MAPPING - STRUCTURE

- passing a structure as a parameter to a function.

// allows the easy transfer of large amounts of data

```
DisplayStats(struct person Input) {
        Input.first = "Dunsul";
    }

/* "person" is name for structure type */
struct person {
         char    first[32];  /* 1st field is array of char */
         char    last[32];   /* 2nd field is array of char */
         int     year;         /* 3rd field is int */
         double   ppg;         /* 4th field is double */
         };                    /* ending ; */
/* means end of structure type definition */

struct  person  teacher;

teacher.year=2005;
teacher.ppg=10.4;
strcpy(teacher.first,"Sam");
strcpy(teacher.last,"Maggs");

 DisplayStats(teacher);
 printf("%s\n",teacher.first);
```

SIDE BAR:
- is the a call by value
-or-
a call by reference ?

Label	Address	Value
teacher	400 - 475	[original]
Input	710 - 785	[copy]

	teacher.ppg	468 - 475	10.4

It is a copy only.

MEMORY MAPPING - STRUCTURE

- passing a structure as a parameter to a function.

```
// allows the easy transfer of large amounts of data
```

```
DisplayStats(struct person Input) {
        Input.first = "Dunsul";
    }

/* "person" is name for structure type */
struct person {
         char    first[32];  /* 1st field is array of char */
         char    last[32];   /* 2nd field is array of char */
         int     year;         /* 3rd field is int */
         double   ppg;                                double */
         };
/* means end of struc

struct  person  teacher;

teacher.year=2005;
teacher.ppg=10.4;
strcpy(teacher.first,"Sam");
strcpy(teacher.last,"Maggs");

 DisplayStats(teacher);
 printf("%s\n",teacher.first);
```

OUTPUT:

SIDE BAR:
- is the a call by value
-or-
a call by reference ?

Label	Address	Value
teacher	400 - 475	[original]
Input	710 - 785	[copy]

	teacher.ppg	468 - 475	10.4

MEMORY MAPPING - STRUCTURE

- passing a structure as a parameter to a function.

```
// allows the easy transfer of large amounts of data
```

```
DisplayStats(struct person Input) {
        Input.first = "Dunsul";
    }
```

return Input.

HOW TO CHANGE SO THE LINE PRINTS OUT:
Dunsul

(lab? or assignment?)

```
/* "pe                                re type */
struct

                                    1st field is array of char */
                                    2nd field is array of char */
                                    /* 3rd field is int */
                                    /* 4th field is double */
                                    /* ending ; */
/* means end of structure type definition */

struct  person  teacher;

teacher.year=2005;
teacher.ppg=10.4;
strcpy(teacher.first,"Sam");
strcpy(teacher.last,"Maggs");

 DisplayStats(teacher);
 printf("%s\n",teacher.first);
```

SIDE BAR:
- is the a call by value
-or-
a call by reference ?

Label	Address	Value
teacher	400 - 475	[original]
Input	710 - 785	[copy]

	teacher.ppg	468 - 475	10.4

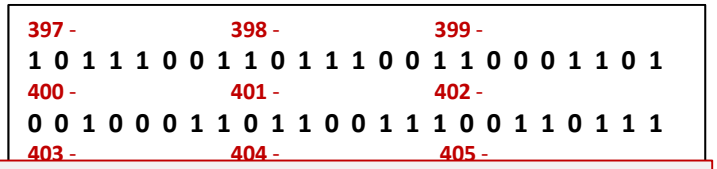
teacher = DisplayStats (struct teacher);

Structures in C

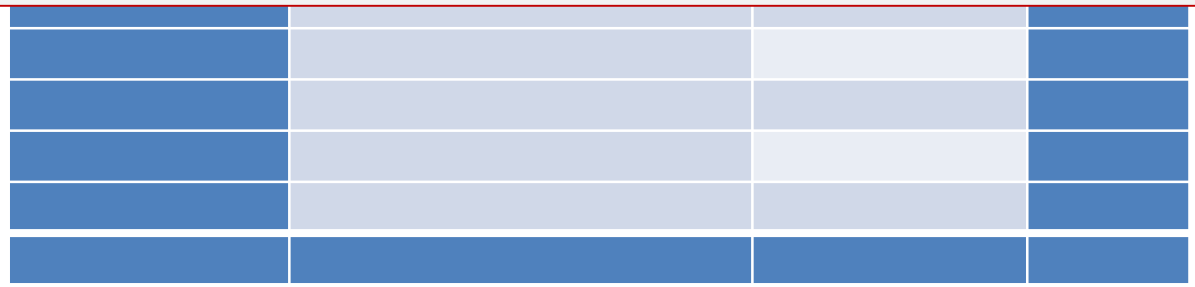
END OF PART 1

MEMORY MAPPING - STRUCTURE

- structures can be nested
- so that a field inside a structure can be another structure




```
struct name {
        char    first[30];
        char    last[30];
        };

struct person {
        int             age;
        float           ppg;
        struct name     title;  /* nested structure */
        };                      .       /* 'name' must be defined above */

/* the structure person now has a nested structure called name */
```



MEMORY MAPPING - STRUCTURE

```
struct name {
            char        first[30];
            char        last[30];
            };

struct person {
            int         age;
            float       ppg;
            struct name title;
            };
```

Address Label	Label	Address	Value

MEMORY MAPPING - STRUCTURE

```
struct name {
            char        first[30];
            char        last[30];
            };

struct person {
            int         age;
            float       ppg;
            struct name title;
            };

struct person boss;
```

Address Label	Label	Address	Value
	boss boss.age	400 - 403	
	boss.ppg	404 - 407	
boss.title.first	boss.title boss.title.first[0]	408 -	
	boss.title.first[1]	409 -	
	boss.title.first[2]	410 -	
	boss.title.first[3]	411 -	
	boss.title.first[4]	412 -	
	boss.title.first[5] – [31]	413 - 439	
boss.title.last	boss.title.last[0]	440 -	
	boss.title.last[1]	441 -	
	boss.title.last[2]	442 -	
	boss.title.last[3]	443 -	
	boss.title.last[4]	444 -	
	boss.title.last[5]	445 -	
	boss.title.last[6] – [31]	446 - 471	

MEMORY MAPPING - STRUCTURE

```
struct name {
            char        first[30];
            char        last[30];
            };

struct person {
            int         age;
            float       ppg;
            struct name title;
            };

struct person boss;

/* fill in some values
   for the nested
   structure */

boss.age=40;
```

boss. age.

Address Label	Label	Address	Value
	boss boss.age	400 - 403	40
	boss.ppg	404 - 407	
boss.title.first	boss.title boss.title.first[0]	408 -	
	boss.title.first[1]	409 -	
	boss.title.first[2]	410 -	
	boss.title.first[3]	411 -	
	boss.title.first[4]	412 -	
	boss.title.first[5] – [31]	413 - 439	
boss.title.last	boss.title.last[0]	440 -	
	boss.title.last[1]	441 -	
	boss.title.last[2]	442 -	
	boss.title.last[3]	443 -	
	boss.title.last[4]	444 -	
	boss.title.last[5]	445 -	
	boss.title.last[6] – [31]	446 - 471	

MEMORY MAPPING - STRUCTURE

```
struct name {
            char        first[30];
            char        last[30];
            };

struct person {
            int         age;
            float       ppg;
            struct name title;
            };

struct person boss;

/* fill in some values
   for the nested
   structure */

boss.age=40;
boss.ppg=0.1;
```

Address Label	Label	Address	Value
	boss boss.age	400 - 403	40
	boss.ppg	404 - 407	0.1
boss.title.first	boss.title boss.title.first[0]	408 -	
	boss.title.first[1]	409 -	
	boss.title.first[2]	410 -	
	boss.title.first[3]	411 -	
	boss.title.first[4]	412 -	
	boss.title.first[5] – [31]	413 - 439	
boss.title.last	boss.title.last[0]	440 -	
	boss.title.last[1]	441 -	
	boss.title.last[2]	442 -	
	boss.title.last[3]	443 -	
	boss.title.last[4]	444 -	
	boss.title.last[5]	445 -	
	boss.title.last[6] – [31]	446 - 471	

MEMORY MAPPING - STRUCTURE

```
struct name {
            char        first[30];
            char        last[30];
            };

struct person {
            int         age;
            float       ppg;
            struct name title;
            };

struct person boss;

/* fill in some values
   for the nested
   structure */

boss.age=40;
boss.ppg=0.1;
strcpy(boss.title.first,"Dean");
```

Dean\0

Label	Label	Address	Value
	boss boss.age	400 - 403	40
	boss.ppg	404 - 407	0.1
e.first	boss.title boss.title.first[0]	408 -	D
	boss.title.first[1]	409 -	e
	boss.title.first[2]	410 -	a
	boss.title.first[3]	411 -	n
	boss.title.first[4]	412 -	\0
	boss.title.first[5] – [31]	413 - 439	
boss.title.last	boss.title.last[0]	440 -	
	boss.title.last[1]	441 -	
	boss.title.last[2]	442 -	
	boss.title.last[3]	443 -	
	boss.title.last[4]	444 -	
	boss.title.last[5]	445 -	
	boss.title.last[6] – [31]	446 - 471	

MEMORY MAPPING - STRUCTURE

```
struct name {
            char        first[30];
            char        last[30];
            };

struct person {
            int         age;
            float       ppg;
            struct name title;
            };

struct person boss;

/* fill in some values
   for the nested
   structure */

boss.age=40;
boss.ppg=0.1;
strcpy(boss.title.first,"Dean");
strcpy(boss.title.last,"Smith");
```

Label	Label	Address	Value
	boss boss.age	400 - 403	40
	boss.ppg	404 - 407	0.1
e.first	boss.title boss.title.first[0]	408 -	D
	boss.title.first[1]	409 -	e
	boss.title.first[2]	410 -	a
	boss.title.first[3]	411 -	n
	boss.title.first[4]	412 -	\0
	boss.title.first[5] – [31]	413 - 439	
boss.title.last	boss.title.last[0]	440 -	S
	boss.title.last[1]	441 -	m
	boss.title.last[2]	442 -	i
	boss.title.last[3]	443 -	t
	boss.title.last[4]	444 -	h
	boss.title.last[5]	445 -	\0
	boss.title.last[6] – [31]	446 - 471	

Structures in C

END OF PART 2

MEMORY MAPPING - STRUCTURE

POINTERS and STRUCTURES

- can be stored as an address in a pointer variable
- just like any other variable

```
struct fraction {
                    int   wP;
                    int   fP;
                  };

struct fraction  f[3], *g;
```

Address Label	Label	Address	Value
f	f[0] f[0]. wP	400 - 403	
	f[0]. fP	404 - 407	
	f[1] f[1]. wP	408 - 411	
	f[1]. fP	412 - 415	
	f[2] f[2]. wP	416 - 419	
	f[2]. fP	420 - 423	
	g	424 - 427	

MEMORY MAPPING - STRUCTURE

POINTERS and STRUCTURES

- can be stored as an address in a pointer variable
- just like any other variable

```
struct fraction {
                    int   wP;
                    int   fP;
                  };

struct fraction  f[3], *g;

 f[0].wP = 3;
 f[0].fP = 7;
 g = &(f[0]);
```

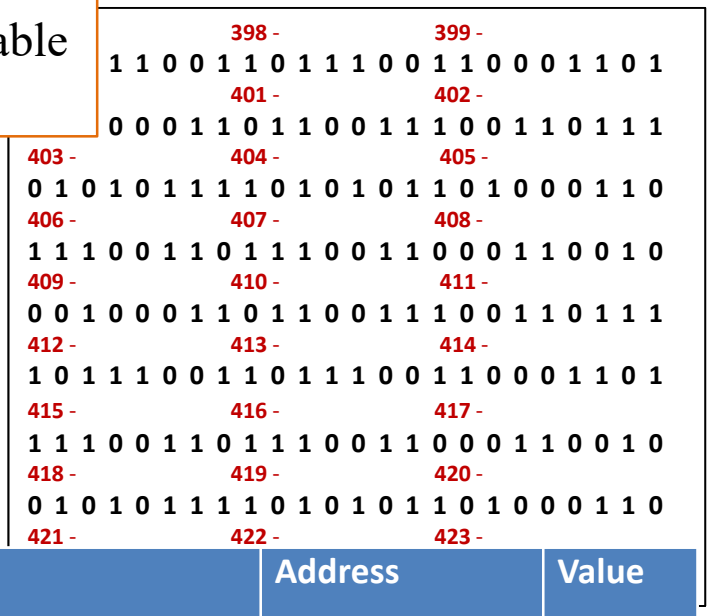
	Label	Address	Value
	f[0] f[0]. wP	400 - 403	3
	f[0]. fP	404 - 407	7
	f[1] f[1]. wP	408 - 411	
	f[1]. fP	412 - 415	
	f[2] f[2]. wP	416 - 419	
	f[2]. fP	420 - 423	
	g	424 - 427	400

MEMORY MAPPING - STRUCTURE

POINTERS and STRUCTURES

- can be stored as an address in a pointer variable
- just like any other variable

```
struct fraction {
                    int   wP;
                    int   fP;
                  };

struct fraction  f[3], *g;

 f[0].wP = 3;
 f[0].fP = 7;
 g = &(f[0]);

g++;
 /* g now
    contains
    400 + 8
    408 */
```



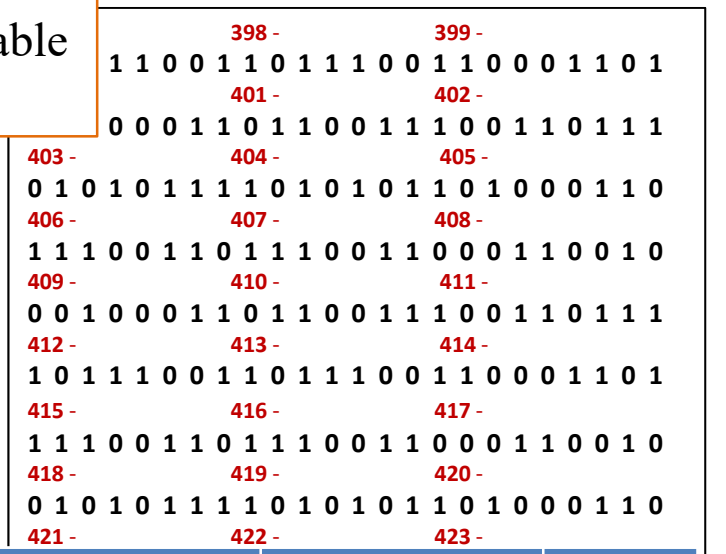
Address Label	Label	Address	Value
f	f[0] f[0]. wP	400 - 403	3
	f[0]. fP	404 - 407	7
	f[1] f[1]. wP	408 - 411	
	f[1]. fP	412 - 415	
	f[2] f[2]. wP	416 - 419	
	f[2]. fP	420 - 423	
	g	424 - 427	408

MEMORY MAPPING - STRUCTURE

POINTERS and STRUCTURES

- can be stored as an address in a pointer variable
- just like any other variable

```
struct fraction {
                    int   wP;
                    int   fP;
                  };

struct fraction  f[3], *g;

 f[0].wP= 3;
 f[0].fP= 7;
 g = &(f[0]);

g++;

(*g).wP = 5;
 g->fP = 11;
```



Address Label	Label	Address	Value
f	f[0] f[0]. wP	400 - 403	3
	f[0]. fP	404 - 407	7
	f[1] f[1]. wP	408 - 411	5
	f[1]. fP	412 - 415	11
	f[2] f[2]. wP	416 - 419	
	f[2]. fP	420 - 423	
	g	424 - 427	408

MEMORY MAPPING - STRUCTURE

POIN
- ca
- jus

- **two ways** of specifying the field to use.

what we have already seen:

```
(*g).wP = 5;
// * (at the address) of g go to the field at wP
```

C also has a **direction** symbol

```
->
g->fP = 11;
```

- which can be used on addresses of structures
  - go to the **structure** pointed to by **g**
  - go to the **fP variable** of that structure.

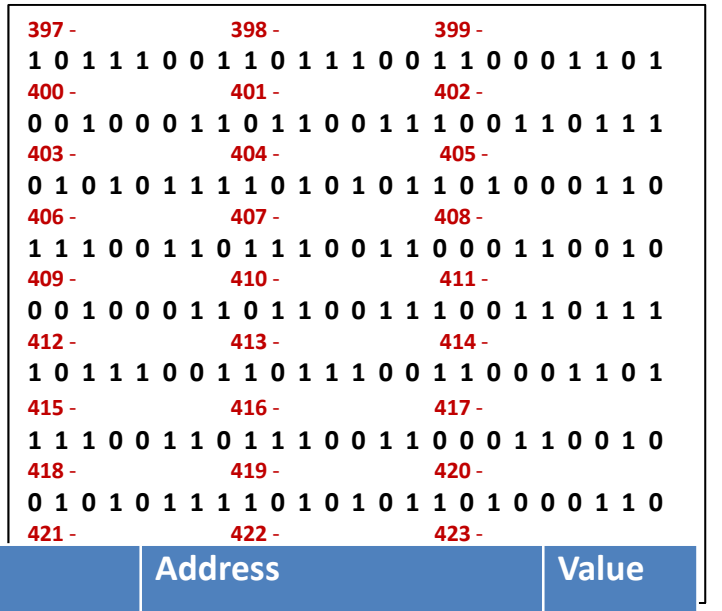
```
struct

struct

 f[0].w
 f[0].f
 g = &(

g++;

(*g).wP = 5;
 g->fP = 11;
```

	f[1]. fP	412 - 415	11
	f[2] f[2]. wP	416 - 419	
	f[2]. fP	420 - 423	
	g	424 - 427	408

MEMORY MAPPING - STRUCTURE

POINTERS and STRUCTURES
- a structure can be declared.

```
struct fraction {
                    int   wP;
                    int   fP;
                  } ;

struct fraction fract;
```

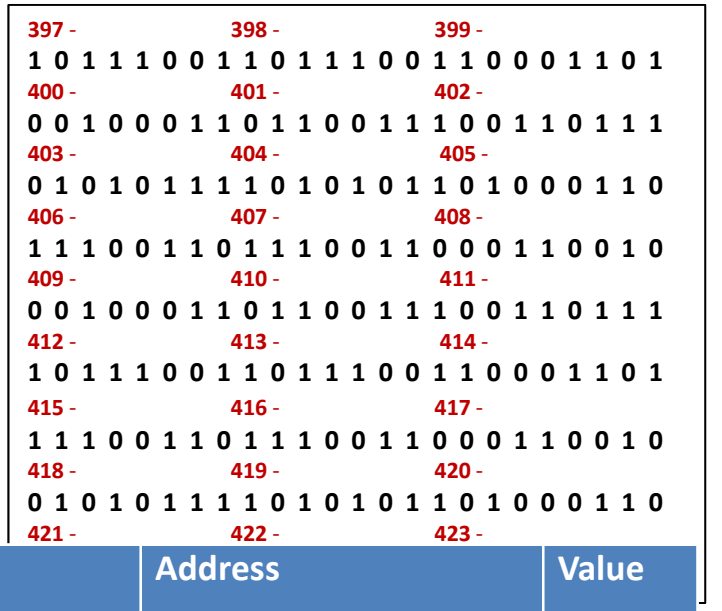


Address Label	Label	Address	Value
	fract.wP	400 - 403	
	fract.fP	404 - 407	

MEMORY MAPPING - STRUCTURE

POINTERS and STRUCTURES
- a structure can be declared.

```
struct fraction {
                    int   wP;
                    int   fP;
                  } ;

struct fraction fract;

fract.wp = 7;
fract.fP = 3;
```

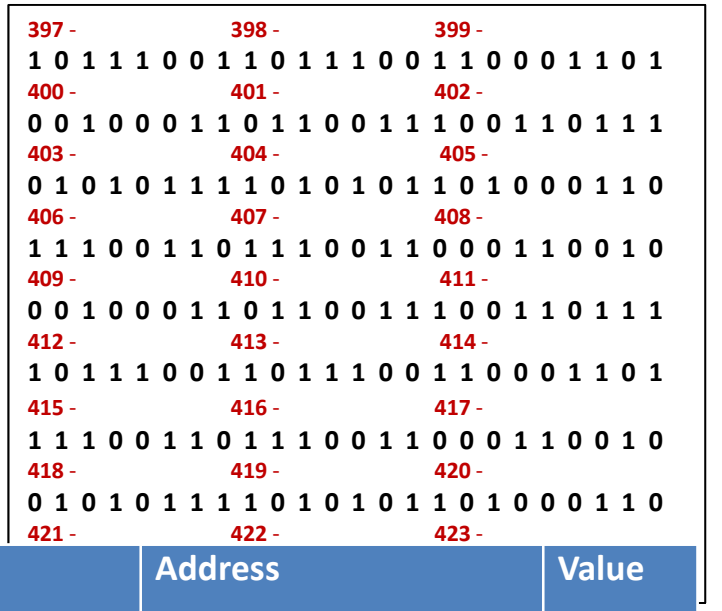


Address Label	Label	Address	Value
	fract.wP	400 - 403	7
	fract.fP	404 - 407	3

MEMORY MAPPING - STRUCTURE

POINTERS and STRUCTURES
- a structure can be declared.

```
struct fraction {
                    int   wP;
                    int   fP;
                  } fract;
```



Address Label	Label	Address	Value
	fract.wP	400 - 403	
	fract.fP	404 - 407	

MEMORY MAPPING - STRUCTURE

POINTERS and STRUCTURES
- a structure can be declared.

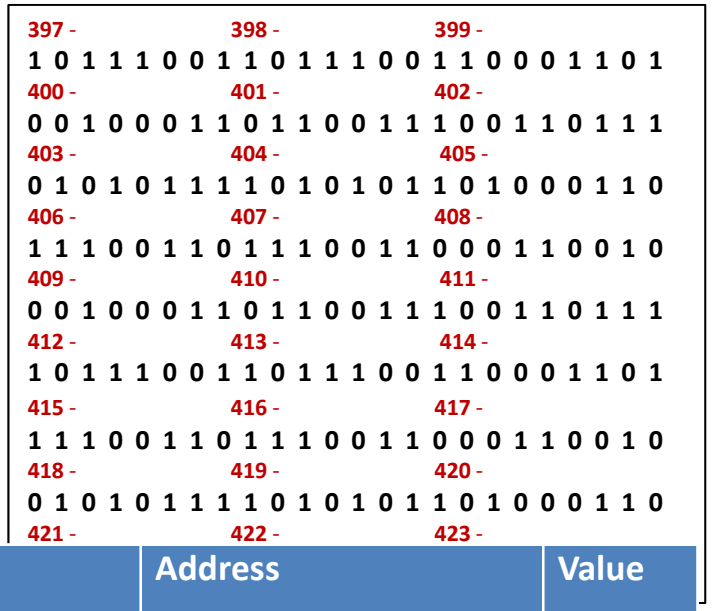
```
struct fraction {
                    int   wP;
                    int   fP;
                  } fract;

fract.wp = 7;
fract.fP = 3;
```

Address Label	Label	Address	Value
	fract.wP	400 - 403	7
	fract.fP	404 - 407	3

MEMORY MAPPING - STRUCTURE

POINTERS and STRUCTURES
- a structure can be declared.
- and/or declared later

```
struct fraction {
                    int   wP;
                    int   fP;
                  } fract;

struct fraction newNum;

fract.wp = 7;
fract.fP = 3;
```

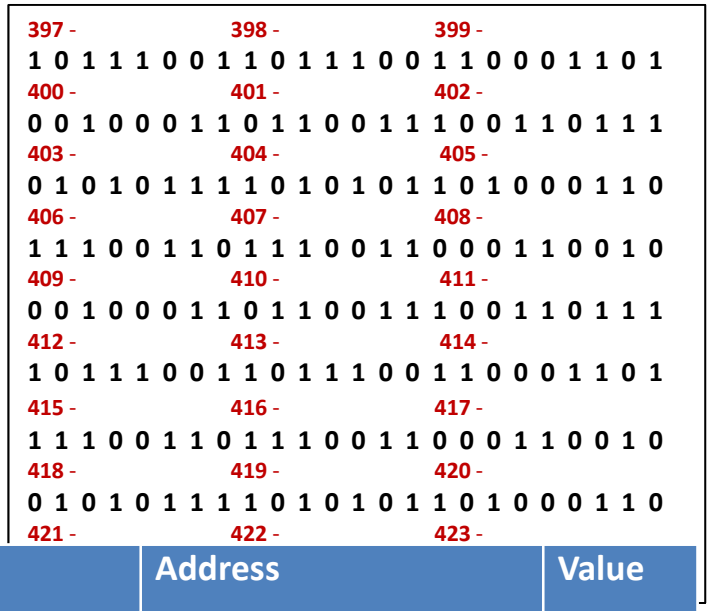


Address Label	Label	Address	Value
	fract.wP	400 - 403	
	fract.fP	404 - 407	
	newNum.wP	408 - 411	
	newNum.fP	412 - 415	

MEMORY MAPPING - STRUCTURE

POINTERS and STRUCTURES
- a structure can be declared.

```
struct fraction {
                    int   wP;
                    int   fP;
                  } fract;

struct fraction newNum;

fract.wp = 7;
fract.fP = 3;

newNum.wp = 2;
newNum.fP = 6;
```

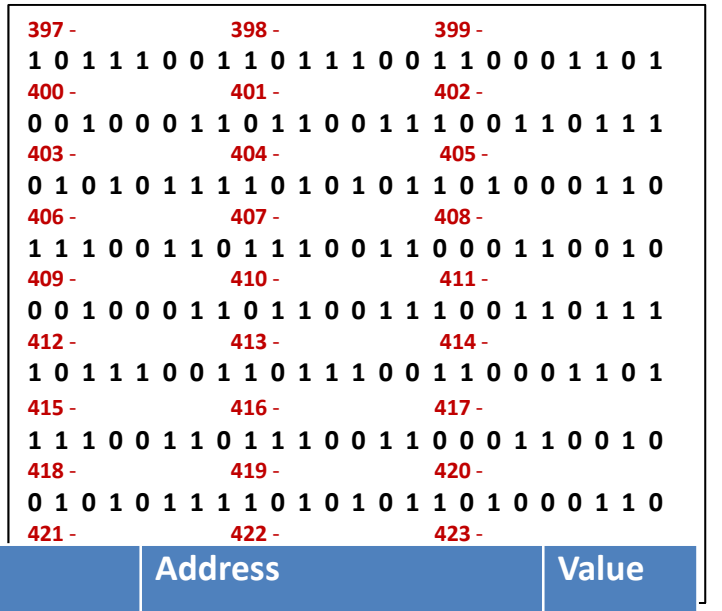


Address Label	Label	Address	Value
	fract.wP	400 - 403	7
	fract.fP	404 - 407	3
	newNum.wP	408 - 411	2
	newNum.fP	412 - 415	6

MEMORY MAPPING - STRUCTURE

POINTERS and STRUCTURES
- a structure can be declared.

```
typedef struct fraction {
                    int   wP;
                    int   fP;
                  } fract;
```



notice:
- the **typedef** does NOT allocate memory for the variable.

- **fract** is just the **alias** for this construct.

Address Label	Label	Address	Value

MEMORY MAPPING - STRUCTURE

POINTERS and STRUCTURES
- a structure can be declared.

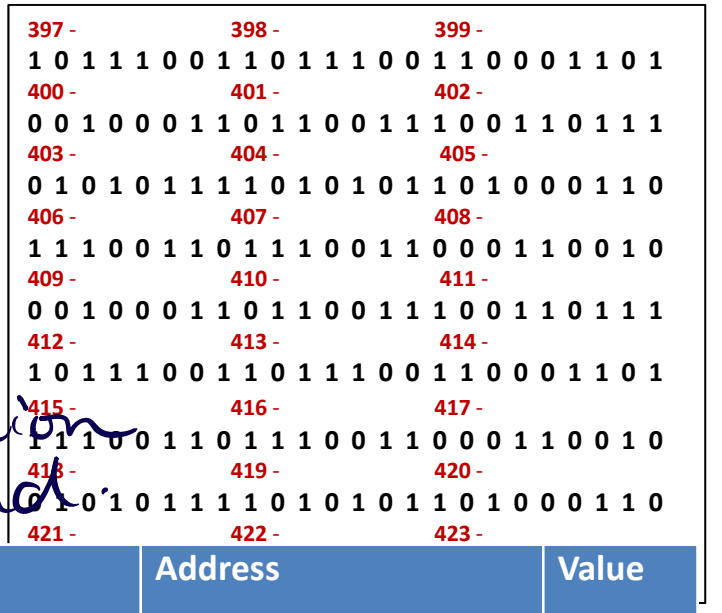
```
typedef struct fraction {
                    int   wP;
                    int   fP;
                  } fract;

fract first, second;
```

Address Label	Label	Address	Value
	first.wP	400 - 403	
	first.fP	404 - 407	
	second.wP	408 - 411	
	second.fP	412 - 415	

MEMORY MAPPING - STRUCTURE

POINTERS and STRUCTURES
- a structure can be declared.

```
typedef struct fraction {
                    int   wP;
                    int   fP;
                  } fract;

fract first, second;

first.wp = 7;
first.fP = 3;

second.wp = 2;
second.fP = 6;
```



struct fraction
first, second

Address Label	Label	Address	Value
	first.wP	400 - 403	7
	first.fP	404 - 407	3
	second.wP	408 - 411	2
	second.fP	412 - 415	6

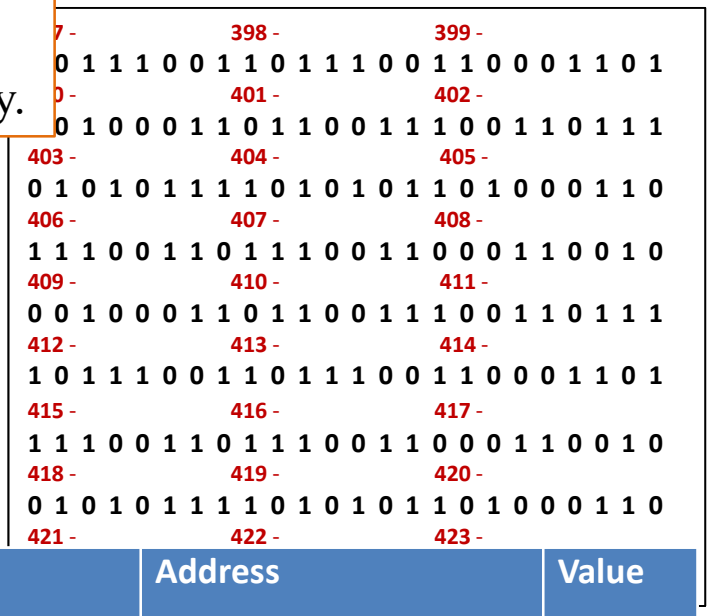
Structures in C

END OF PART 3

MEMORY MAPPING - STRUCTURE

POINTERS and STRUCTURES
- a pointer variable can also be used to **DYNAMICALLY** allocate memory.

```
struct fraction {
                    int   wP;
                    int   fP;
                  };
struct fraction  f[3], *g;
```

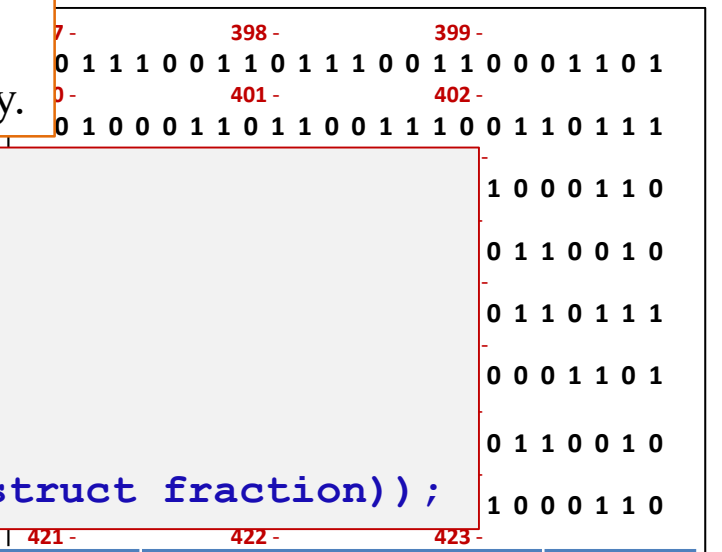


Address Label	Label	Address	Value
f	f[0] f[0]. wP	400 - 403	
	f[0]. fP	404 - 407	
	f[1] f[1]. wP	408 - 411	
	f[1]. fP	412 - 415	
	f[2] f[2]. wP	416 - 419	
	f[2]. fP	420 - 423	
	g	424 - 427	

MEMORY MAPPING - STRUCTURE

POINTERS and STRUCTURES
- a pointer variable can also be used to **DYNAMICALLY** allocate memory.

```
struct fraction {
                    int   wP;
                    int   fP;
                  };
struct fraction   f[3], *g;

g = (struct fraction *)malloc( sizeof(struct fraction));
```



Address Label	Label	Address	Value
f	f[0] f[0]. wP	400 - 403	
	f[0]. fP	404 - 407	
	f[1] f[1]. wP	408 - 411	
	f[1]. fP	412 - 415	
	f[2] f[2]. wP	416 - 419	
	f[2]. fP	420 - 423	
	g	424 - 427	10100
	{DM}	10100 - 10107	

MEMORY MAPPING - STRUCTURE

POINTERS and STRUCTURES

- a pointer variable can also be used to **DYNAMICALLY** allocate memory.

```
struct fr

struct fr

g = (stru
```

```
g = (struct fraction *)malloc( sizeof(struct fraction));
```

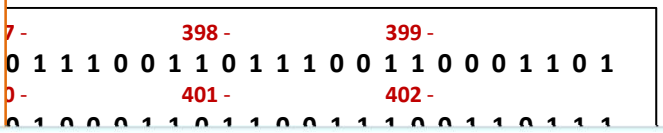
- what is happening here ?

  - a struct of type **fraction** is how many bytes in size ?
    - 8 bytes (2 integer values x 4 bytes)

    - so: (sizeof(struct fraction)) will return 8 bytes of memory

  - **malloc()** returns a pointer of type void
    UNLESS specifically type cast as above

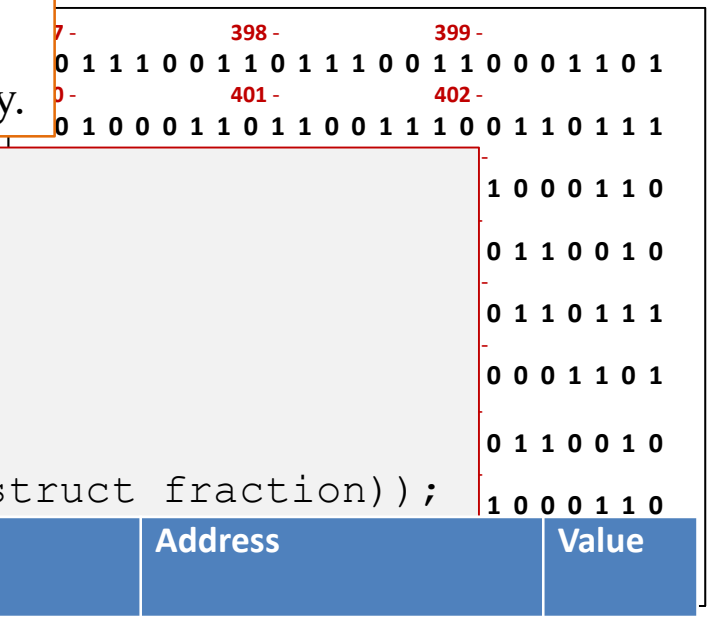
  - this tells the program the pointer references
    a memory location of a structure of the type **fraction**



	g	424 - 427	10100
	{DM}	10100 - 10107	

MEMORY MAPPING - STRUCTURE

POINTERS and STRUCTURES
- a pointer variable can also be used to **DYNAMICALLY** allocate memory.



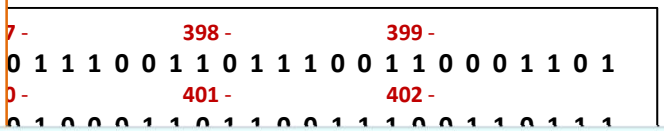

```
struct fraction {
                    int   wP;
                    int   fP;
                  };
struct fraction   f[3], *g;

g = (struct fraction *)malloc( sizeof(struct fraction));

 g->wP=6;
 (*g).fP=1;
```

Address Label	Label	Address	Value
f	f[0] f[0]. wP	400 - 403	
	f[0]. fP	404 - 407	
	f[1] f[1]. wP	408 - 411	
	f[1]. fP	412 - 415	
	f[2] f[2]. wP	416 - 419	
	f[2]. fP	420 - 423	
	g	424 - 427	10100
	g.wP	10100 - 10103	6
	g.fP	10104 - 10107	1

MEMORY MAPPING - STRUCTURE

POINTERS and STRUCTURES

- a pointer variable can also be used to **DYNAMICALLY** allocate memory.




```
struct fr

struct fr

g = (stru

 g->wP=6;
 (*g).fP=
```

```
g = (struct fraction *)malloc( sizeof(struct fraction));
```

malloc allocated structure size (8 bytes) of memory for use
malloc(sizeof(struct fraction))

- **g** (as a pointer variable) it is just a pointer.
  holds the address of this memory location

- **g** indirection allows the use of the memory

SO: as stated:

typecasting (struct fraction *) allows the O/S
to know how to deal with this memory allocation

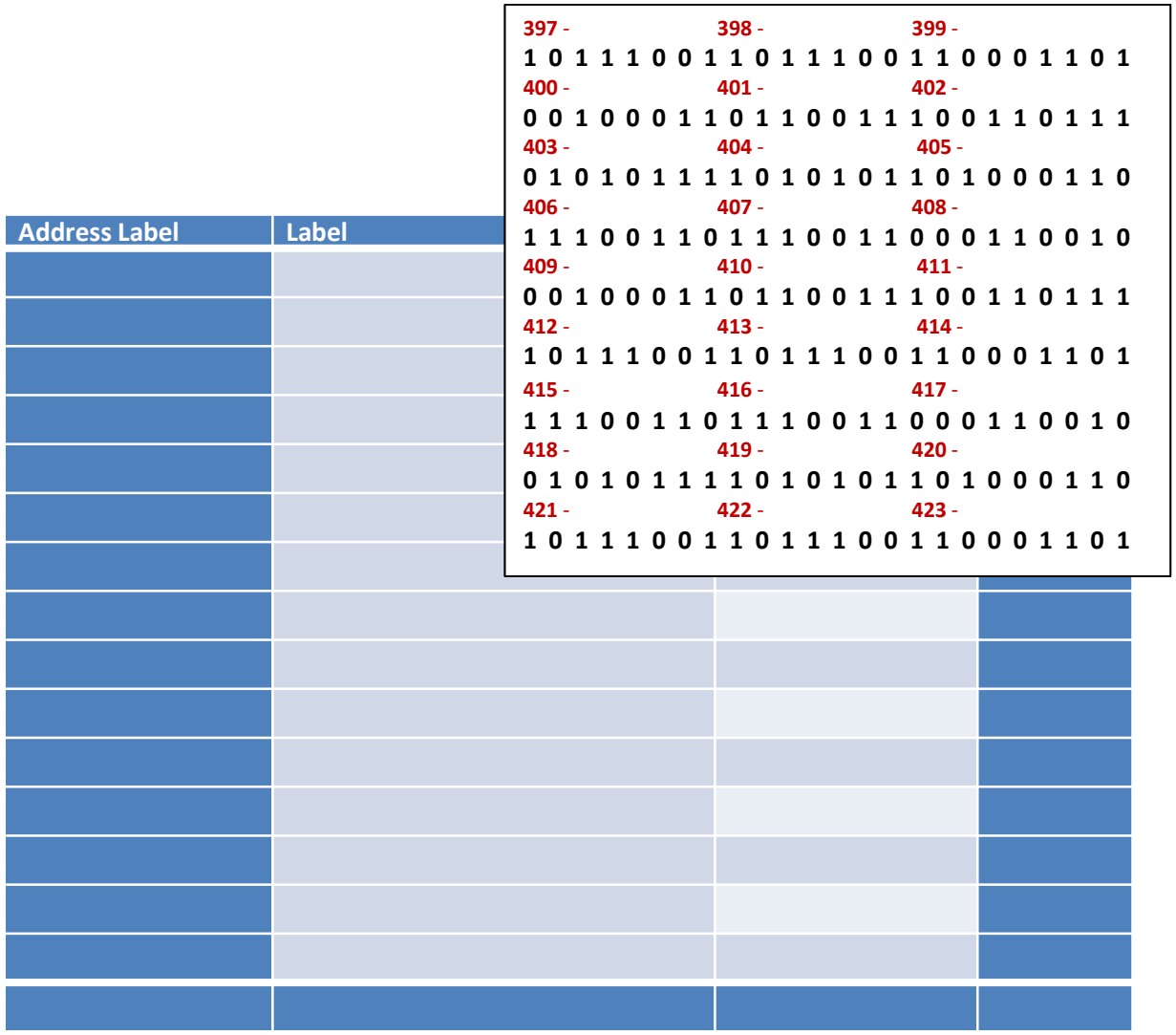
Structures in C

END OF PART 4

CS 2211
Systems Programming

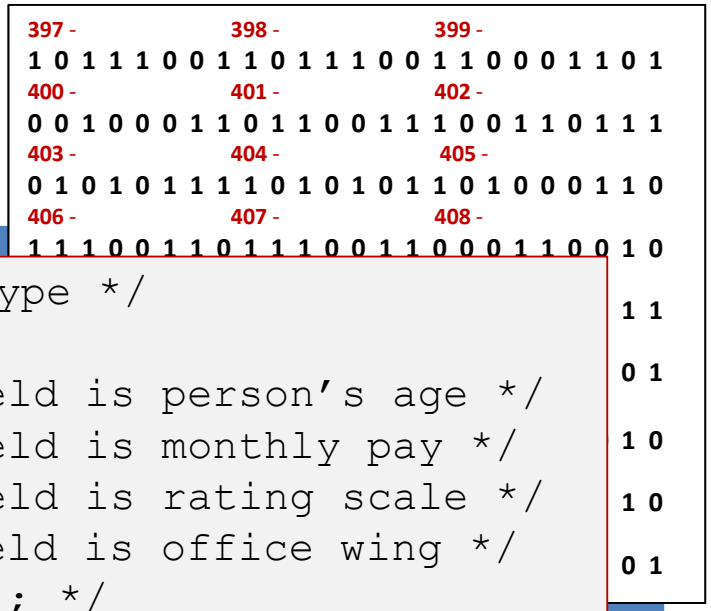
Part Ten: Unions

MEMORY MAPPING - UNIONS



MEMORY MAPPING - UNIONS

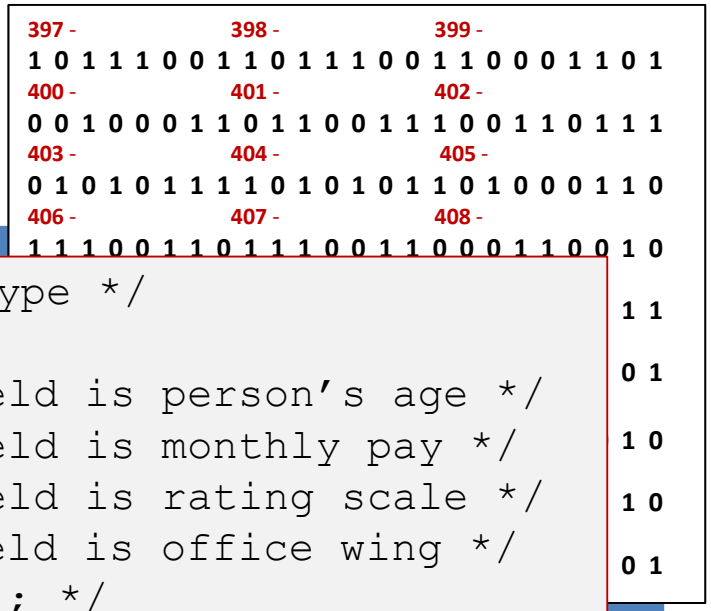
- a construct to group together dissimilar variables under **one name**




```
/* "demographics" is name for union type */
union demographics {
        int    age           /* 1st field is person's age */
        float  salary;       /* 2nd field is monthly pay */
        double emplevel;     /* 3rd field is rating scale */
        char    Owing;       /* 4th field is office wing */
        };                   /* ending ; */
/* means end of union type definition */
```

MEMORY MAPPING - UNIONS

- a construct to group together dissimilar variables under **one name**



Address Label | Label

```
/* "demographics" is name for union type */
union demographics {
        int    age           /* 1st field is person's age */
        float  salary;       /* 2nd field is monthly pay */
        double emplevel;     /* 3rd field is rating scale */
        char   Owing;        /* 4th field is office wing */
        };                   /* ending ; */
/* means end of union type definition */
```

NOTE:

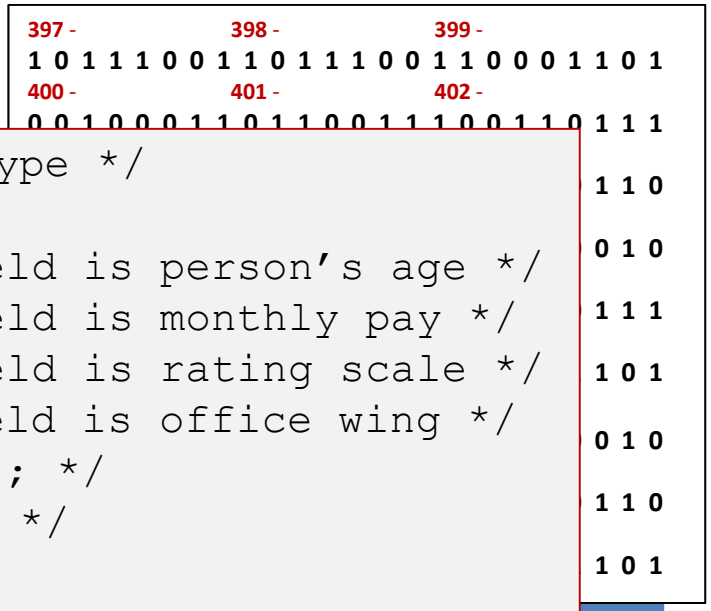
This code does NOT create a variable

(similar to Class declaration in OOP)

- grandfather of this concept
- this is ONLY a template for a variable declaration.
- this creates a **USER DEFINED** variable type

MEMORY MAPPING - UNIONS

- a construct to group together dissimilar variables under **one name**




```
/* "demographics" is name for union type */
union demographics {
        int    age           /* 1st field is person's age */
        float  salary;       /* 2nd field is monthly pay */
        double emplevel;     /* 3rd field is rating scale */
        char    Owing;       /* 4th field is office wing */
        };                   /* ending ; */
/* means end of union type definition */

int  i;
union  demographics employee;
```

creates an integer variable (space to store an integer) **i**

- and -

creates a demographic variable (space to store ONLY the largest of the variables in the UNION [in this case 8 bytes]) **employee**.

MEMORY MAPPING - UNIONS

```
/* "demographics" is name for union type */
union demographics {
        int    age          /* 1st field is person's age */
        float  salary;      /* 2nd field is monthly pay */
        double emplevel;    /* 3rd field is rating scale */
        char   Owing;               /* 4th field is office wing */
        };                           /* ending ; */
/* means end of union type definition */
```

4
4
8
1

Address Label	Label	Address	Value

MEMORY MAPPING - UNIONS

```
/* "demographics" is name for union type */
union demographics {
            int    age          /* 1st field is person's age */
            float  salary;      /* 2nd field is monthly pay */
            double emplevel;    /* 3rd field is rating scale */
            char   Owing;               /* 4th field is office wing */
            };                           /* ending ; */
/* means end of union type definition */

union demographics employee;
```

Address Label	Label	Address	Value
employee	employee.age	400 - 403	
	employee.salary	400 - 403	
	employee.emplevel	400 - 407	
	employee.Owing	400	

- memory 400 to 407

- the union employee takes up (requires) 8 bytes of data

note:

```
employee
employee.salary
employee.age
employee.emplevel
employee.Owing
```

all returns the address **400**

MEMORY MAPPING - UNIONS

```
/* "demographics" is name for union type */
union demographics {
            int    age          /* 1st field is person's age */
            float  salary;      /* 2nd field is monthly pay */
            double emplevel;    /* 3rd field is rating scale */
            char   Owing;               /* 4th field is office wing */
            };                           /* ending ; */
/* means end of union type definition */

union demographics employee;

employee.age = 24;
```

Address Label	Label	Address	Value
employee	employee.age	400 - 403	24
	employee.salary	400 - 403	
	employee.emplevel	400 - 407	
	employee.Owing	400	

MEMORY MAPPING - UNIONS

```
/* "demographics" is name for union type */
union demographics {
            int    age          /* 1st field is person's age */
            float  salary;      /* 2nd field is monthly pay */
            double emplevel;    /* 3rd field is rating scale */
            char   Owing;               /* 4th field is office wing */
            };                           /* ending ; */
/* means end of union type definition */

union demographics employee;

employee.age = 24;

employee.Owing = 'D';
```

Address Label	Label	Address	Value
employee	employee.age	400 - 403	
	employee.salary	400 - 403	
	employee.emplevel	400 - 407	
	employee.Owing	400	D

MEMORY MAPPING - UNIONS

```
/* "demographics" is name for union type */
union demographics {
            int    age          /* 1st field is person's age */
            float  salary;      /* 2nd field is monthly pay */
            double emplevel;    /* 3rd field is rating scale */
            char   Owing;                /* 4th field is office wing */

/* m

unio

empl

empl
```

CAUTION !

```
employee.Owing = 'D';
```

assigned the sequence 0100 0100 to the memory location 400

BUT !!! Address 401 and 402 and 403 are UNCHANGED

1.) these held the values from the previous assignment of:

```
employee.age = 24;
```

So the sequence that was:

0000 0000 0000 0000 0000 0000 0001 1000

Is now changed to

0100 0100 0000 0000 0000 0000 0001 1000

Now:

```
printf("age = %d",employee.age);
```

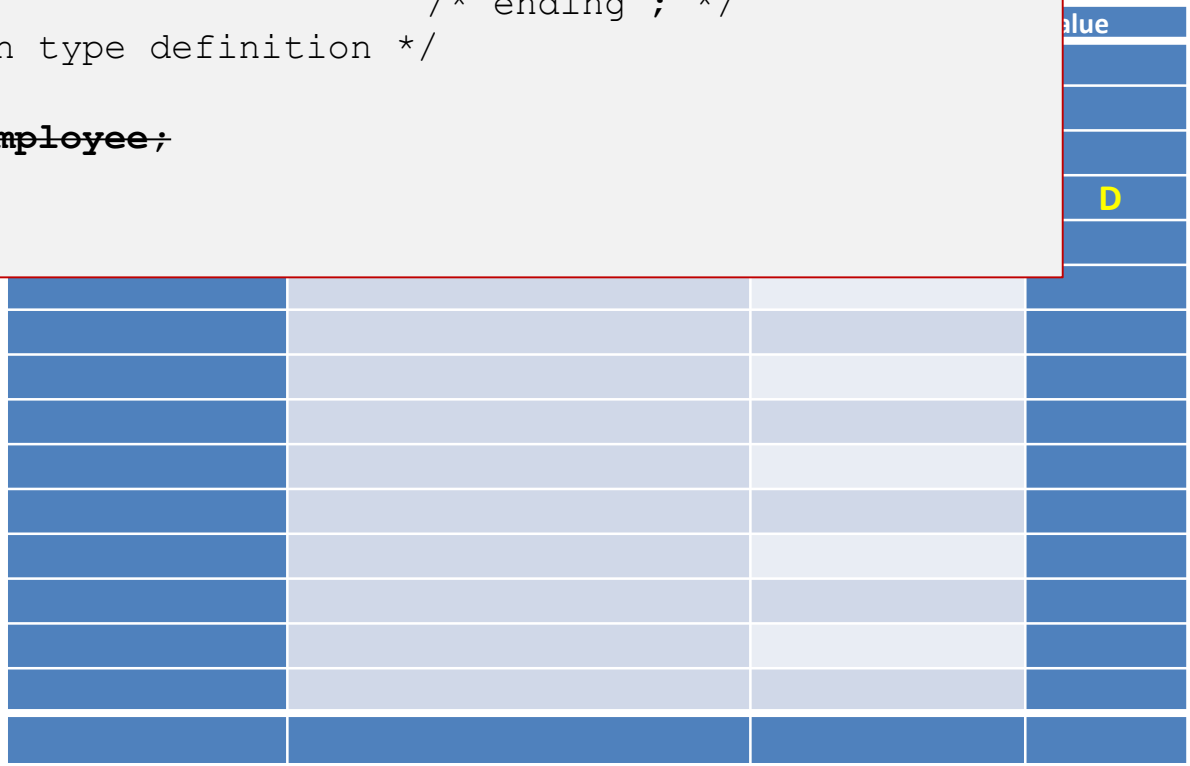
produces:

age = 570425356

	Value
3	
3	
7	
	D

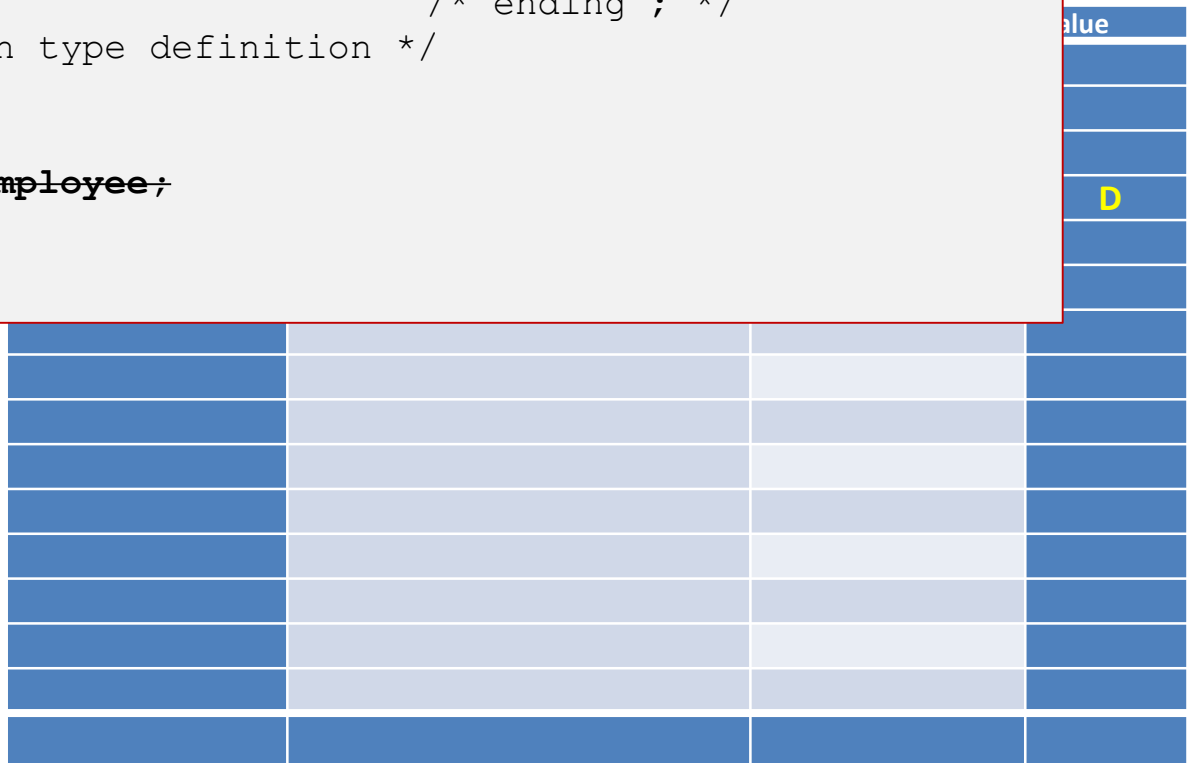
MEMORY MAPPING - UNIONS

```
/* "demographics" is name for union type */
union demographics {
        int    age          /* 1st field is person's age */
        float  salary;      /* 2nd field is monthly pay */
        double emplevel;    /* 3rd field is rating scale */
        char   Owing;        /* 4th field is office wing */
        } employee;                    /* ending ; */
/* means end of union type definition */

union demographics employee;
```



MEMORY MAPPING - UNIONS

```
/* "demographics" is name for union type */
typedef union demographics {
        int    age          /* 1st field is person's age */
        float  salary;      /* 2nd field is monthly pay */
        double emplevel;    /* 3rd field is rating scale */
        char   Owing;        /* 4th field is office wing */
        } dGraphics;                    /* ending ; */
/* means end of union type definition */

dGraphics employee;
union demographics employee;
```



MEMORY MAPPING - STRUCTURE vs UNION

	STRUCTURE	UNION
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is **greater than or equal to the sum of sizes of its members.**	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of **union is equal to the size of largest member.**
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

Unions in C

END OF PART 1