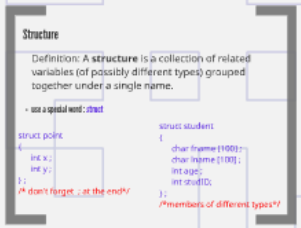
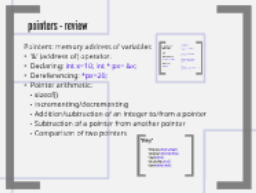
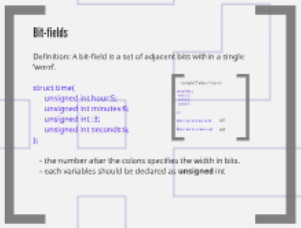
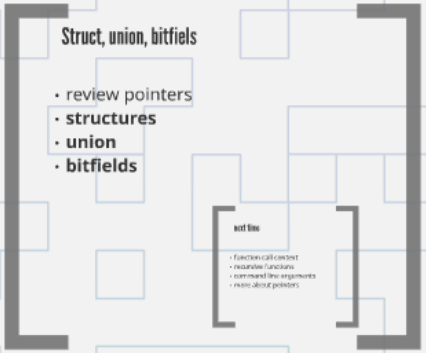
Struct, union, bitfields

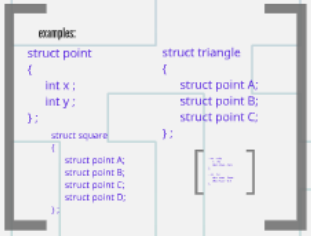
Struct, union, bitfields

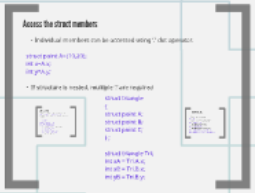


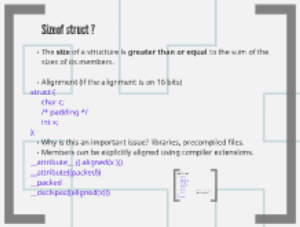












Struct, union, bitfiels

- review pointers
- **structures**
- **union**
- **bitfields**

next time

- function call context
- recursive functions
- command line arguments
- more about pointers

pointers - review

Pointers: memory address of variables

- '&' (address of) operator.
- Declaring: int x=10; int * px= &x;
- Dereferencing: *px=20;
- Pointer arithmetic:
  - sizeof()
  - incrementing/decrementing
  - Addition/subtraction of an integer to/from a pointer
  - Subtraction of a pointer from another pointer
  - Comparison of two pointers

"Strings"

- String copy: strcpy(),strncpy()
- Comparison: strcmp(),strncmp()
- Length: strlen()
- Concatenation: strcat()
- Search: strchr(),strstr()

initialize at run time = dynamic allocation

stdlib.h

allocate memory at run time

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

deallocate memory

```
void free(void *);
```

```
TYPE1 *p1, *p2;
TYPE2 *p3, *p4;
...
p1=malloc(n*sizeof(TYPE1));
if( p1 != NULL)
...

p3=malloc(m*sizeof(TYPE2));
if(p3 != NULL)
...

free(p3); /*p3 no more need*/
p2=malloc(n*sizeof(TYPE1));
if (p2 != NULL)

...

free(p1);
if((p4=malloc(m*sizeof(TYPE2)) == NULL)
 { /* handle error */ }
...
```

"Strings"

- String copy: strcpy(),strncpy()
- Comparison: strcmp(),strncmp()
- Length: strlen()
- Concatenation: strcat()
- Search: strchr(),strstr()

Structure

Definition: A **structure** is a collection of related variables (of possibly different types) grouped together under a single name.

- use a special word : struct

```
struct point
{
    int x ;
    int y ;
} ;
/* don't forget  ; at the end*/
```

```
struct student
{
    char fname [100] ;
    char lname [100] ;
    int age ;
    int studID;
} ;
/*members of different types*/
```

struct

struct defines a new datatype.

- The name of the structure is optional.

```
struct {...} x,y,z;
```

- The variables declared within a structure are called its **members**

- Variables of a structure can be declared like any other

```
struct point stPointA;
```

- Initialization is done by specifying values of every member.

```
struct point stPointA={10,20};
```

- Assignment operator (=) copies every member of the structure (be careful with pointers to structures).

examples:

```
struct point
{
    int x ;
    int y ;
} ;
```

```
struct square
{
    struct point A;
    struct point B;
    struct point C;
    struct point D;
} ;
```

```
struct triangle
{
    struct point A;
    struct point B;
    struct point C;
} ;
```

```
struct node{
    int info;
    struct node * next;
};

struct list{
    struct node * head;
    struct node * tail;
};
```

```
struct node{
    int info;
    struct node * next;
};

struct list{
    struct node * head;
    struct node * tail;
};
```

Access the struct members

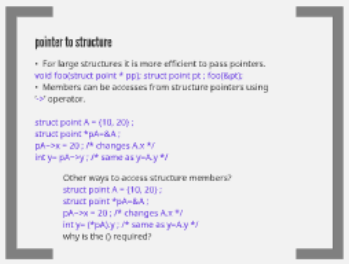
- Individual members can be accessed using '.' dot operator.

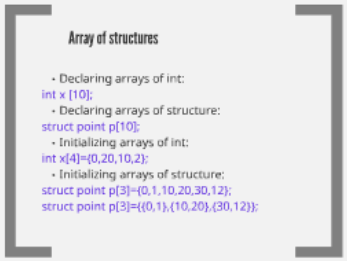
```
struct point A={10,20};
int x=A.x;
int y=A.y;
```

- If structure is nested, multiple '.' are required

```
struct triangle
{
struct point A;
struct point B;
struct point C;
} ;

struct triangle Tri;
int xA = Tri.A.x;
int xB = Tri.B.x;
int yB = Tri.B.y;
```





pointer to structure

- For large structures it is more efficient to pass pointers.

```
void foo(struct point * pp); struct point pt ; foo(&pt);
```

- Members can be accesses from structure pointers using '->' operator.

```
struct point A = {10, 20} ;
struct point *pA=&A ;
pA−>x = 20 ; /* changes A.x */
int y= pA−>y ; /* same as y=A.y */
```

Other ways to access structure members?

```
struct point A = {10, 20} ;
struct point *pA=&A ;
pA−>x = 20 ; /* changes A.x */
int y= (*pA).y ; /* same as y=A.y */
```

why is the () required?

Array of structures

- Declaring arrays of int:

```
int x [10];
```

- Declaring arrays of structure:

```
struct point p[10];
```

- Initializing arrays of int:

```
int x[4]={0,20,10,2};
```

- Initializing arrays of structure:

```
struct point p[3]={0,1,10,20,30,12};
struct point p[3]={{0,1},{10,20},{30,12}};
```

Sizeof struct ?

- The **size** of a structure is **greater than or equal** to the sum of the sizes of its members.

- Alignment (if the alignment is on 16 bits)

```
struct {
    char c;
    /* padding */
    int x;
};
```

- Why is this an important issue? libraries, precompiled files.
- Members can be explicitly aligned using compiler extensions.

```
__attribute__ (( aligned(x )))
__attribute((packed))
__packed
__declspec((aligned(x)))
```

consider 16 bits alignment

```
struct foo{
    char v1;
    short int v2;
    char v3;
    short int v4;
    int v5;
    char v6;
    int v7;
};
struct foo x;
```

How much space is allocated for x?

unions

A **union** is a variable that may hold objects of different types/sizes in **the same memory location**.
Example:

```
union data
{
    int idata ;
    float fdata ;
    char * sdata ;
} d1 ;

d1.idata =10;

d1.fdata =3.14F ;

d1.sdata = (char *)malloc(20);
strcpy(d1.sdata, "hello world") ;
```

sizeof union

The size of the union variable is equal to the size of its **largest element.**

- Important: **The compiler does not test if the data is being read in the correct format.**

```
union data d;
d.idata=10;
float f=d.fdata; /*what will be the value of f?*/
```

- A common solution is to maintain a separate variable.

```
enum dtype { INT , FLOAT, CHAR } ;
struct variant{
    union data d ;
    enum dtype t ;
} ;
```

sizeof union

The size of the union variable is equal to the size of its **largest element.**

- Important: **The compiler does not test if the data is being read in the correct format.**

```
union data d;
d.idata=10;
float f=d.fdata; /*what will be the value of f?*/
```

- A common solution is to maintain a separate variable.

```
enum dtype { INT , FLOAT, CHAR } ;
struct variant{
    union data d ;
    enum dtype t ;
} ;
```

Bit-fields

Definition: A bit-field is a set of adjacent bits within a single 'word'.

```
struct time{
    unsigned int hour:5;
    unsigned int minutes:6;
    unsigned int :3;
    unsigned int seconds:6;
};
```

using bit-fields or masks?

```
struct bits {
  int b1:1;
  int b2:1;
  int b3:1;
  ...
} x;

if(x.b1 && x.b2 && x.b3)        x&7
...
if(x.b1 && !x.b2 && x.b3)       x&5
```

- the number after the colons specifies the width in bits.
- each variables should be declared as **unsigned** int

using bit-fields or masks?

```
struct bits {
  int b1:1;
  int b2:1;
  int b3:1;
  ...
} x;

if(x.b1 && x.b2 && x.b3)        x&7
...
if(x.b1 && !x.b2 && x.b3)       x&5
```

next time

- function call context
- recursive functions
- command line arguments
- more about pointers

Struct, union, bitfields