

Struct, union, bitfields

Struct

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

Example:

```

// example.c
struct point {
    int x;
    int y;
};

int main() {
    struct point p;
    p.x = 10;
    p.y = 20;
}
    
```

Priority of different types?

Pointers - review

Review: memory address of variable

- To declare a pointer
- To declare a pointer to a pointer
- Dereferencing: `*ptr`
- Pointer arithmetic
- Array of pointers
- Array of array of pointers
- Array of pointer to array of pointers
- Array of pointer to pointer
- Array of array of pointer

Bit fields

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

Example:

```

// bitfields.c
struct bitfield {
    unsigned int a: 5;
    unsigned int b: 5;
    unsigned int c: 5;
};

int main() {
    struct bitfield b;
    b.a = 10;
    b.b = 20;
    b.c = 30;
}
    
```

The location after the column specifies the width in bits. Each variable has the size of its width.

Struct, union, bitfields

- review pointers
- structures
- union
- bitfields

Union

Definition: A union is a data type that can store one of several different types of data.

Example:

```

// union.c
union data {
    int i;
    float f;
};

int main() {
    union data u;
    u.i = 10;
    u.f = 3.14;
}
    
```

The variable declared as the union can hold any of the types listed.

Union

Definition: A union is a data type that can store one of several different types of data.

Example:

```

// union.c
union data {
    int i;
    float f;
};

int main() {
    union data u;
    u.i = 10;
    u.f = 3.14;
}
    
```

Union

Definition: A union is a data type that can store one of several different types of data.

Structs and pointers

Definition: A pointer to a structure is a variable that stores the memory address of a structure.

Example:

```

// struct_ptr.c
struct point {
    int x;
    int y;
};

int main() {
    struct point p;
    struct point *ptr;
    ptr = &p;
    ptr->x = 10;
    ptr->y = 20;
}
    
```

Example

```

// example.c
struct point {
    int x;
    int y;
};

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

int main() {
    struct point p;
    struct triangle t;
    t.A.x = 10;
    t.A.y = 20;
    t.B.x = 30;
    t.B.y = 40;
    t.C.x = 50;
    t.C.y = 60;
}
    
```

Structs and unions

Definition: A union is a data type that can store one of several different types of data.

Example:

```

// struct_union.c
struct point {
    int x;
    int y;
};

union data {
    struct point p;
    int i;
};

int main() {
    union data u;
    u.p.x = 10;
    u.p.y = 20;
    u.i = 30;
}
    
```

Struct, union, bitfields

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*/
```

pointers - review

- Pointers: memory addresses of variables
- `&` (address of) operator
- Declaring: `int i=10; int *p=&i;`
- Dereferencing: `*p=20;`
- Pointer arithmetic:
 - `++` (increment)
 - `--` (decrement)
 - Addition/subtraction of an integer to/from a pointer
 - Subtraction of a pointer from another pointer
 - Comparison of two pointers

```
int i=10;
int *p=&i;
*p=20;
```

```
int *p=&i;
int *q=p;
```

Bit-fields

Definition: A bit-field is a set of adjacent bits with a single "word".

```
struct bitfield
{
    unsigned int flag:5;
    unsigned int minutes:6;
    unsigned int seconds:5;
};
```

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



Struct, union, bitfields

- review pointers
- structures
- union
- bitfields

struct

- `struct` defines a new datatype.
- The name of the structure is optional.
- `struct L...` e.g.:
- The variables declared within a structure are called its **members**
- Variables of a structure can be declared like any other data type (e.g. `int`).
- Initialization is done by specifying values of every member.
- `struct point p(10,20);`
- `*` pointer operator (`*`) points every member of the structure (be careful with pointers to structures).

unions

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

```
union data
{
    int i;
    char c;
};
/* sizeof union data is 4 bytes */
/* sizeof char is 1 byte */
/* sizeof int is 4 bytes */
```



nesting

- Functions call context
- Relative positions
- Named arguments
- Info about pointers

Access the struct members

- Individual members can be accessed using `*` (the operator)

```
struct point A(10,20);
int main()
{
    int x,y;
    struct point p;
    p.x=A.x;
    p.y=A.y;
}
```



examples:

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

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

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



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 data
{
    char c;
    /* padding */
    int i;
};
```



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

```
__attribute__((aligned(16)))
__attribute__((packed))
__attribute__((packed))
```

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

```
initialize at run time =  
dynamic allocation  
  
stdlib.h  
allocate memory at run time  
void * malloc(size_t, bytes);  
  
deallocate memory  
void free(void *);  
  
TYPE1 *p1, *p2;  
TYPE2 *p3, *p4;  
...  
p1=malloc(sizeof(TYPE1));  
if (p1 == NULL)  
...  
p2=malloc(sizeof(TYPE2));  
if (p2 == NULL)  
...  
free(p1); /* no more need? */  
p2=malloc(sizeof(TYPE1));  
if (p2 == NULL)  
...  
free(p2);  
if (malloc(sizeof(TYPE2)) == NULL)  
{ /* handle error */ }
```

"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 * p); struct point pt; foo(&pt);
- Members can be accessed 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 0 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}};

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 accessed 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?

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

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*/
```

pointers - review

- Pointers: memory addresses of variables
- `&` (address of) operator
- Declaring: `int i=10; int *p=&i;`
- Dereferencing: `*p=20;`
- Pointer arithmetic:
 - `++` (increment)
 - `--` (decrement)
 - Addition/subtraction of an integer to/from a pointer
 - Subtraction of a pointer from another pointer
 - Comparison of two pointers

```
int i=10;
int *p=&i;
*p=20;
```

```
int *p=&i;
int *q=p;
```

Bit-fields

Definition: A bit-field is a set of adjacent bits with a single "word".

```
struct bitfield
{
    unsigned int flag:5;
    unsigned int minutes:6;
    unsigned int seconds:5;
};
```

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

```
struct bitfield
{
    unsigned int flag:5;
    unsigned int minutes:6;
    unsigned int seconds:5;
};
```

Struct, union, bitfields

- review pointers
- structures
- union
- bitfields

struct

- `struct` defines a new datatype.
- The name of the structure is optional.
- `struct L...` e.g.:
- The variables declared within a structure are called its **members**
- Variables of a structure can be declared like any other data type (e.g. `int`).
- Initialization is done by specifying values of every member.
- `struct point p(10,20);`
- `*` pointer operator (`*`) copies every member of the structure (be careful with pointers to structures).

unions

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

```
union data
{
    int i;
    char c;
};
/* sizeof union = 4 bytes */
/* sizeof char = 1 byte */
/* sizeof int = 4 bytes */
```

```
union data
{
    int i;
    char c;
};
/* sizeof union = 4 bytes */
/* sizeof char = 1 byte */
/* sizeof int = 4 bytes */
```

nesting

- Functions call context
- Relative functions
- Named arguments
- Note about pointers

Access the struct members

- Individual members can be accessed using `*` (dot) operator.

```
struct point A(10,20);
int main()
{
    int x,y;
```

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

```
    T.A.x=10;
    T.A.y=20;
    T.B.x=20;
    T.B.y=30;
    T.C.x=30;
    T.C.y=40;
}
```

examples:

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

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

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

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

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 data
{
    char c;
    /* padding */
    int i;
};
```

```
struct data
{
    char c;
    /* padding */
    int i;
};
```

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

```
__attribute__((aligned(16)))
__attribute__((packed))
struct data
{
    char c;
    /* padding */
    int i;
};
```