Introduction to C

Programming Workshop in C (67316)
Fall 2018
Lecture 5
1.11.2018

Memory allocation summary

```
void* malloc(size_t n)
```

- malloc() allocates blocks of memory
- returns a pointer to uninitialized block of memory on success (NULL on failure)
- the returned value should be cast to appropriate type:

```
int *p = (int*) malloc (sizeof(int)*length);
```

```
void* calloc(size_t n, size_t n)
```

- calloc() allocates an array of n elements each of 'size' bytes
- initializes memory to 0

```
int *p = (int*) calloc (length, sizeof(int));
```

```
void* realloc(void *ptr, size_t new_size );
```

- attemts to resize the memory block pointed to by ptr that was previously allocated with a call to malloc or calloc.

free - deallocates the memory previously allocated by malloc, calloc, or realloc

Structs

The origin of classes

Structs

- A structure is a collection of related variables (possibly of different types) grouped together under a single name
- This is an example of composition - building complex structures out of simple ones

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

;

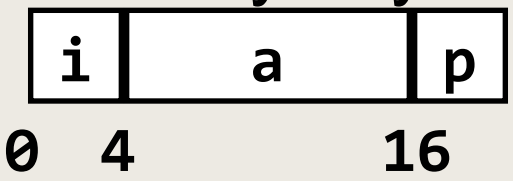
```
struct Student
{
    char fname[100];
    char lname[100];
    int id;
};
```

Structs

- Contiguously-allocated region of memory
- Refer to *members* within structure by names (rather than order in arrays)
- Members may be of different types
- Example:

```
struct MyStruct
{
    int i;
    int a[3];
    int *p;
};
```

Memory Layout



Struct initialization

Structs can be initialized in a way similar to arrays:

```
struct MyStruct
{
    int i;
    int a[3];
    int *p;
};
int k;
struct MyStruct s = { 5, 0, 1, 2, &k };
s.i = 1; // access using ‘.’ operator
s.a[0] = 5;
s.p = &k;
```

Structs are like any other type

- variables of type struct
- a pointer to a struct
- arrays of structs
- pass a struct to a function
- return a struct from a function
- …

More examples

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

struct Triangle
{
    struct Point a;
    struct Point b;
    struct Point c;
};
```

```
struct Triangle t;

int ax = t.a.x;

int by = t.b.y;
```

multiple '.' are required due to nesting

members are structs too

More examples

```
struct ChainElement
{
    int data;
    struct ChainElement *next;
};
```

self referential member

Access to struct members via pointers

```
struct MyStr
{
    int _a[10];
};
```

```
main()
{
    struct MyStr x;
    struct MyStr *p_x = &x;

    x._a[2] = 3;

    (*p_x)._a[2] = 3; // same

}
```

Access to struct members via pointers

```
struct MyStr
{
    int _a[10];
};
```

```
main()
{
    struct MyStr x;
    struct MyStr *p_x = &x;

    x._a[2] = 3;

    (*p_x)._a[2] = 3; // same

    p_x->_a[2] = 3; // same
}
```

The -> operator

typedef

typedef

- Synonyms for variable types – make your program more readable
- Can be used also for built-in types

```
typedef <existing_type_name> <new_type_name>
```

```
typedef struct MyStr MyStrStruct;
typedef struct MyStr MyStr;
typedef unsigned long size_t;

size_t l = strlen(“abc”);
```

typedef

- Defining struct typedef while defining the struct

complex.h

```
typedef struct Complex
{
    double _real, _imag;
} Complex;

Complex addComplex(Complex, Complex);
Complex subComplex(Complex, Complex);
```

typedef

- Defining struct typedef while defining the struct

complex.h

```
typedef struct Complex
{
    double _real, _imag;
} Complex;

Complex addComplex(Complex, Complex);
Complex subComplex(Complex, Complex);
```

optional

Data alignment

Further reading:

https://en.wikipedia.org/wiki/Data_structure_alignment

Data alignment

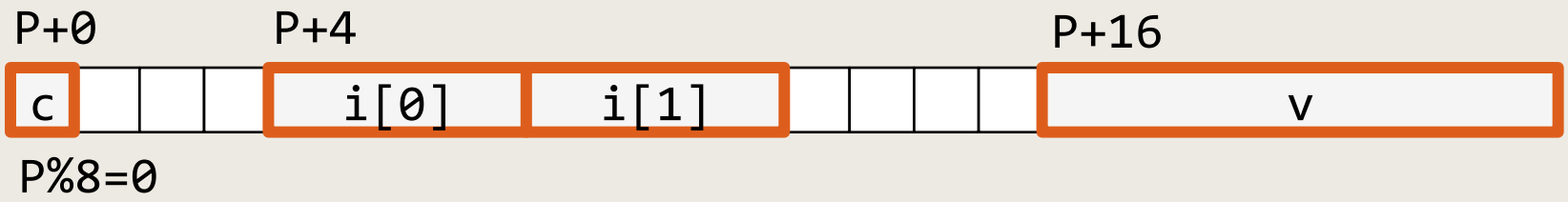
- sizeof (struct) ≥ sizeof (its members)
- Each member is aligned to the lowest address after the previous member that satisfies:

```
mod(address/sizeof(member)) == 0
```

```
struct S {
    char c;
    int i[2];
    double v;
}
```

e.g.:

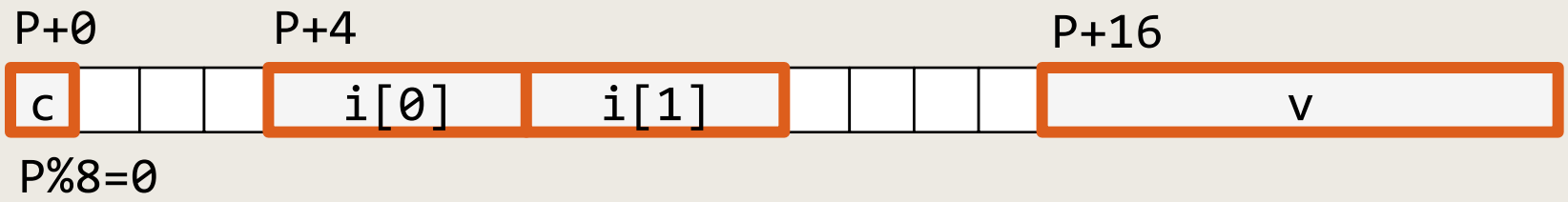
int (4 bytes) will be aligned to 8/12/…



Data alignment – why?

- Hardware fetches memory in chunks (words), and thus **fetching misaligned vars may be slower**
- Some platforms (CPUs) don't support misaligned memory access

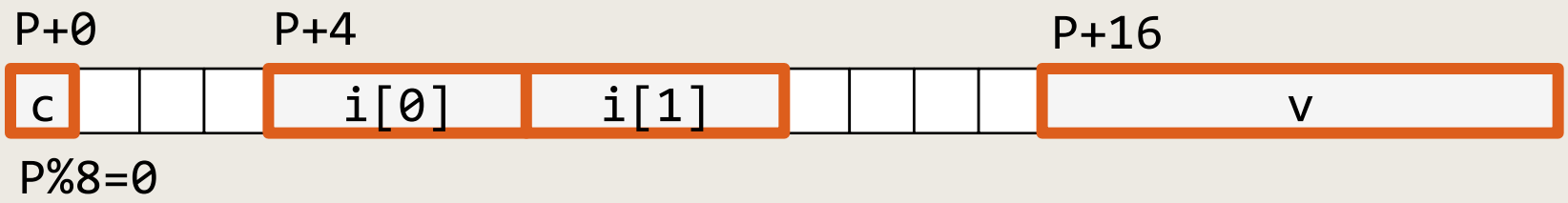
```
struct S {
    char c;
    int i[2];
    double v;
}
```



Data alignment – structure padding

- Padding bytes (unnamed) are added where needed, to make sure all members are properly aligned
- A struct may be padded at the end such that its total size would be a multiple of its largest member – needed for proper alignment of each element in an array of structures

```
struct S {
    char c;
    int i[2];
    double v;
}
```



sizeof struct / structs padding / alignment

- The compiler takes care of that for us (by padding)
- We can plan a compact struct, e.g.:

  `{char, char, short, int}`
- We can tell the compiler to “pack” the structs

```
struct S {
    char c;
    int i[2];
    double v;
} __attribute__((packed));
```

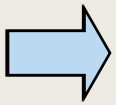
Copying structs with different members

structs copying

Copy structs using '=':
copies struct values (byte-by-byte)

just

```
Complex a,b;
a._real = 5;
a._imag = 3;
b = a;
```



a:

```
_real = 5
_imag = 3
```

b:

```
_real = 5
_imag = 3
```

Arrays in structs copying

struct definition:

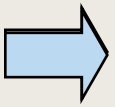
vec.h

```
typedef struct Vec
{
    double _arr [MAX_SIZE];
} Vec;

Vec addVec(Vec, Vec);
...
```

Arrays in structs copying

copy struct using ‘=‘:

```
Vec a,b;
a._arr[0] = 5;
a._arr[1] = 3;
b = a;
```



a:

_arr =

{5,3,…}

b:

_arr =

{5,3,…}

But !!!

Pointers in structs copying

struct definition:

vec.h

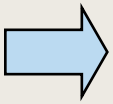
```
typedef struct Vec
{
   double _arr[MAX_SIZE];
   double * _p_arr;
} Vec;

Vec addVec(Vec, Vec);
...
```

Pointers in structs copying

Copy structs using ‘=‘:

copies **just** struct values!!!

```
Vec a,b;
a._arr[0] = 5;
a._arr[1] = 3;
a._p_arr = a._arr;
b = a;
```



a:
_arr = {5,3,?,…}

_p_arr = 0x55

b:
_arr = {?,?,?,…}

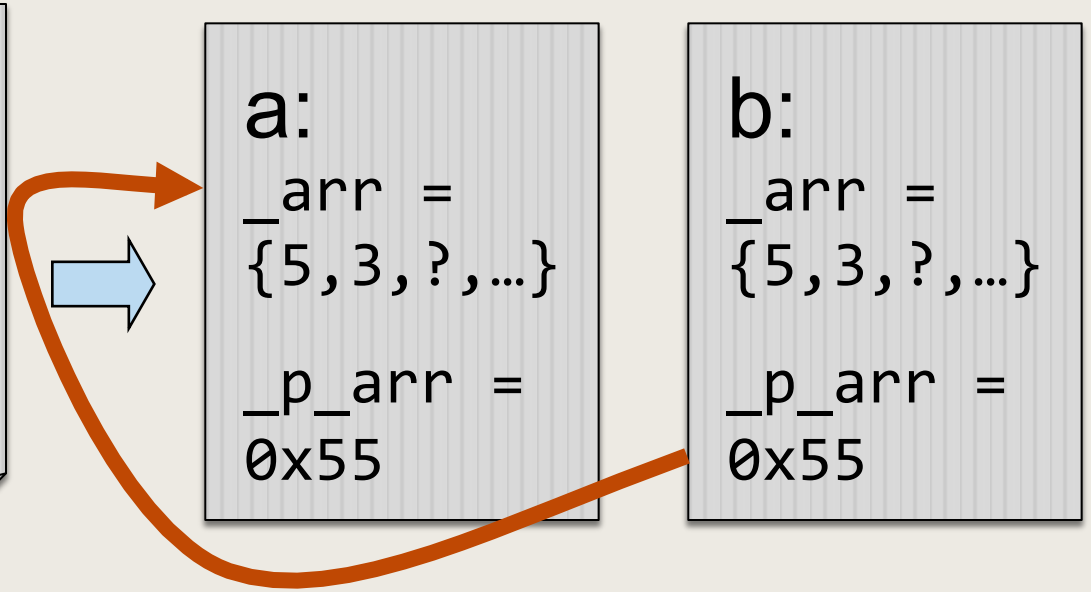
_p_arr = ?

Pointers in structs copying

Copy structs using ‘=‘:

copies **just** struct values!!!

```
Vec a,b;
a._arr[0] = 5;
a._arr[1] = 3;
a._p_arr = a._arr;
b = a;
```



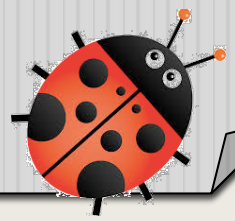
Pointers copied by value!!!

Pointers in structs copying

The result:

```
Vec a,b;
a._arr[0] = 5;
a._arr[1] = 3;
a._p_arr = a._arr;
b = a;
*(b._p_arr) = 8;
printf ("%f", a._arr[0]);
```

```
// output
8
```



How to copy structs with pointer members correctly?

Implement a clone function:

```
void cloneVec (Vec *a, Vec *b)
{
    int i = 0;
    for (i = 0; i < MAX_SIZE; i++)
    {
        b->_arr[i] = a->_arr[i];
    }
    b->_p_arr = b->_arr;
}
```

Arrays & structs as arguments

When an **array** is passed as an argument to a function, the **address of the 1st element** is passed.

Structs are passed **by value**, exactly as the basic types.

Arrays & structs as arguments

```
typedef struct MyStr
{
   int _a[10];
} MyStr;

void f(int a[])
{
   a[7] = 89;
}

void g(MyStr s)
{
   s._a[7] = 84;
}
```

```
main()
{
   MyStr x;
   x._a[7] = 0;
   f(x._a);
   printf("%d\n", x._a[7]);
   g(x);
   printf("%d\n", x._a[7]);
}
```

Output:

Arrays & structs as arguments

```
typedef struct MyStr
{
   int _a[10];
} MyStr;

void f(int a[])
{
   a[7] = 89;
}

void g(MyStr s)
{
   s._a[7] = 84;
}
```

```
main()
{
   MyStr x;
   x._a[7] = 0;
   f(x._a);
   printf("%d\n", x._a[7]);
   g(x);
   printf("%d\n", x._a[7]);
}
```

Output:

```
89
89
```

Structs and Memory Management

Malloc example

```
void *malloc( size_t Size );

int* iptr =
     (int*) malloc(sizeof(int));

struct Complex* complex_ptr =
     (struct Complex*)
     malloc(sizeof(struct Complex));
```

initialization function

```
int* iptr = (int*) malloc(sizeof(int));
struct Complex* complex_ptr =
              (struct Complex*)malloc(sizeof(struct Complex));
```

*iptr = not initialized
*complex_ptr = not initialized

Good design – an initialization function for a struct (poor constructor):

```
struct Complex *complex_ptr =
                    newComplex (1.0, 2.1);
```

initialization function

```
struct Complex
       *newComplex(double r, double i)
{
   struct Complex *p =
     (struct Complex*)
     malloc (sizeof (Complex));
   p->_real = r;
   p->_imag = i;
   return p;
}
```

free also works for structs, of course…

```
void foo( double r, double i )
{
    struct Complex* p_c =
                    newComplex (r,i);
    // do something with p_c
    free(p_c);
}
```

This version frees all allocated memory (good)