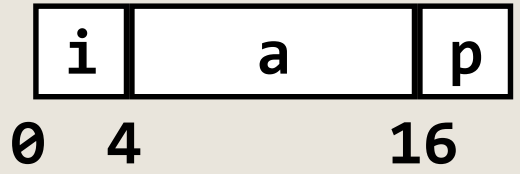
structs & typedef

Structs

- Contiguously-allocated region of memory
- Refer to members within structure by names
- Members may be of different types
- Example:

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

Possible Memory Layout



Struct initialization

`struct`s can be initialized in a way similar to arrays:

```
struct rec
{
   int i;
   int a[3];
   int *p;
};
...
int k;
struct rec r = { 5, { 0,1,2}, &k };
r.i=1;
r.a[0]=5;
r.p=&k;
```

Struct initialization

structs can be initialized in a way similar to arrays:

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

...

int k;
struct rec r = { 5, { 0,1,2}, &k };
r.i=1;
r.a[0]=5;
r.p=&k;
```

Do we really need to write struct rec???

typedef

- Synonyms for variable types – make your program more readable
- Syntax:

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

- Example:

```
typedef      unsigned int              size_t;
```

typedef

- Synonyms for variable types – make your program more readable

complex.h

```
struct _Complex
{
   double _real, _imag;
};

typedef struct _Complex Complex;

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

typedef

- Synonyms for variable types – make your program more readable

complex.h

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

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

typedef- why?

Readibilty: shorter names or dedicated names

```
typedef unsigned int size_t;
```

typedef- why?

Portability:

```
#ifdef INT_4_BYTES
   typedef int int32;
   typedef short int16;
#else
   typedef long int32;
   typedef int int16;
#endif
```

typedef- why?

Generic code:

- e.g. changing numbers in a data structure from char to int easily
- Not always recommended

Pointers to `structs`

- You can have a pointer to structs the same way you have pointers to built in type.

Access to struct members via pointers

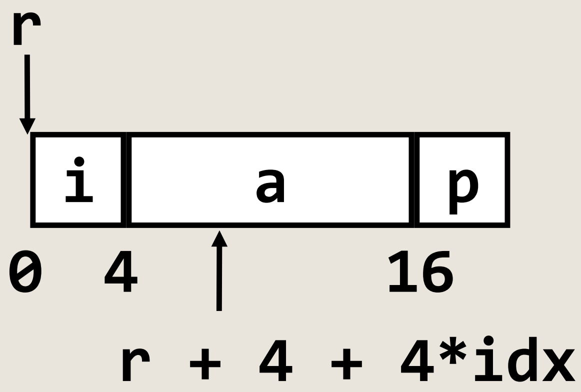
```
typedef struct
_MyStr
{
    int _a[10];
} MyStr;
```

```
main()
{
    MyStr x;
    MyStr *p_x = &x;
    x._a[2] = 3;
    (*p_x)._a[3] = 5;
    p_x->_a[4] = 6;
}
```

Structs - Code

Offset of each structure member determined at compile time

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

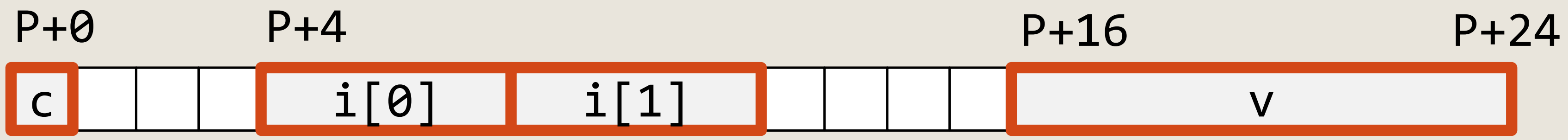


```
int* find_a(struct rec *r, int idx)
{
   return &(r->a[idx]);
}
```

Alignment in memory

Compiler specific. In MSVC 2012 default params:

```
struct S1
{
    char c;
    int i[2];
    double v;
};
```

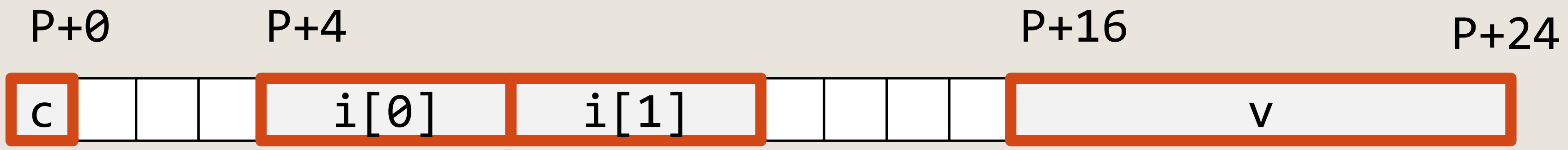


Note: MSVC compiler has flags to control this !

struct

- Sequential in memory, but may have gaps:
  sizeof(S1) != sizeof(char)+sizeof(double)+sizeof(int)*2
- Member offset determined in compile time
- Access member with "." e.g. a.i[0], a.v.
- Access member of struct pointer contents: (*p). or **p->**
- **As always, pass by value!**

```
struct S1 {
   char c;
   int i[2];
   double v;
};
struct S1 a,*p;
```



Structs – old object oriented design

Complex.h

```
struct Complex
{
   double _real, _imag;
};
struct Complex addComplex(struct Complex, struct Complex);
```

Complex.c

```
#include "Complex.h"
// implementation
struct Complex addComplex(struct Complex a, struct Complex b)
{
```

MyProg.c

```
#include "Complex.h"
int main()
{
   struct Complex c;
   ...
```

Structs – old object oriented design – Better design – more on this later

Complex.h

```
struct _Complex;
typedef struct _Complex Complex;

Complex* Complex_alloc();
```

Complex.c

```
#include "Complex.h"
struct _Complex {double _real, _imag}
Complex* Complex_alloc(double real, double imag) {...
```

MyProg.c

```
#include "Complex.h"
int main()
{
   Complex* c_ptr= Complex_alloc(3.0, -1.2);
   ...
```

#ifndef – for header safety

Complex.h:

```
struct Complex
{
   ...
```

MyStuff.h:

```
#include "Complex.h"
```

Main.c:

```
#include "MyStuff.h"
#include "Complex.h"
```

Error:
Complex.h:1: redefinition of `struct Complex'

#ifndef – header safety

Complex.h (revised):

```
#ifndef COMPLEX_H
#define COMPLEX_H
struct Complex
{
...
#endif
```

Main.c:

```
#include "MyStuff.h"
#include "Complex.h" // no error this time
```

#pragma once – header safety

Complex.h (revised):

```
#pragma once
struct Complex
{
...
```

Main.c:

```
#include "MyStuff.h"
#include "Complex.h" // no error this time
```

structs copying

Copy structs using ‘=‘:
copy just struct values!!!

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

a:

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

b:

```
_real = ?
_imag = ?
```

structs copying

Copy structs using ‘=‘:
copy just struct values!!!

```
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

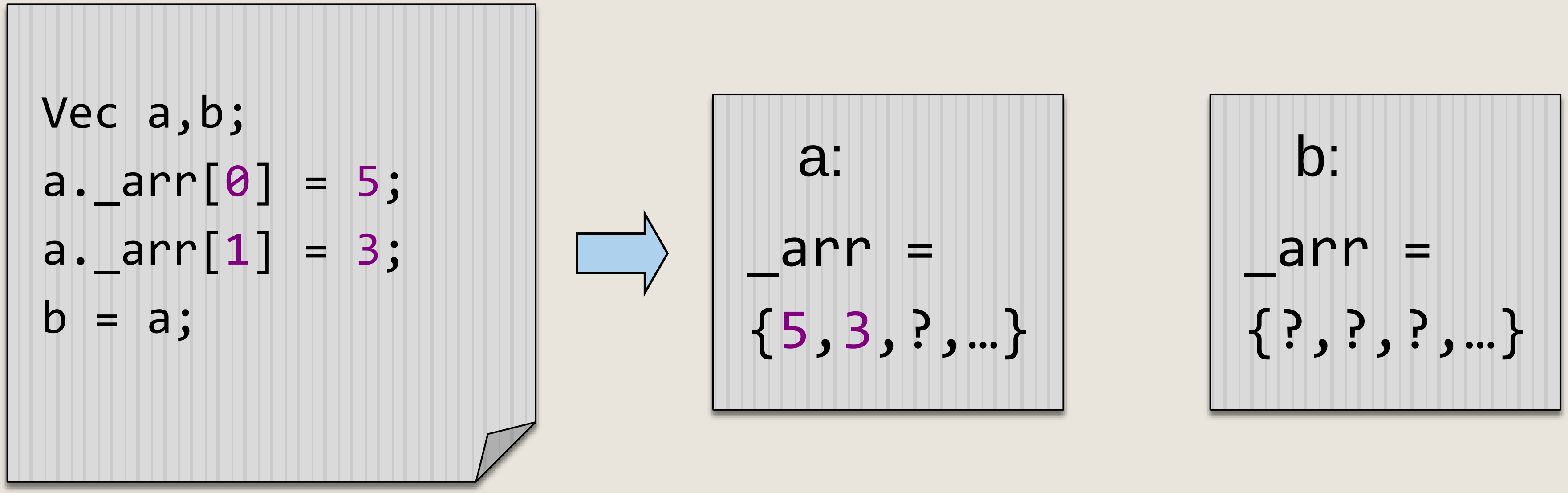
struct definition:

vec.h

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

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

Arrays in structs copying

copy struct using ‘=‘:



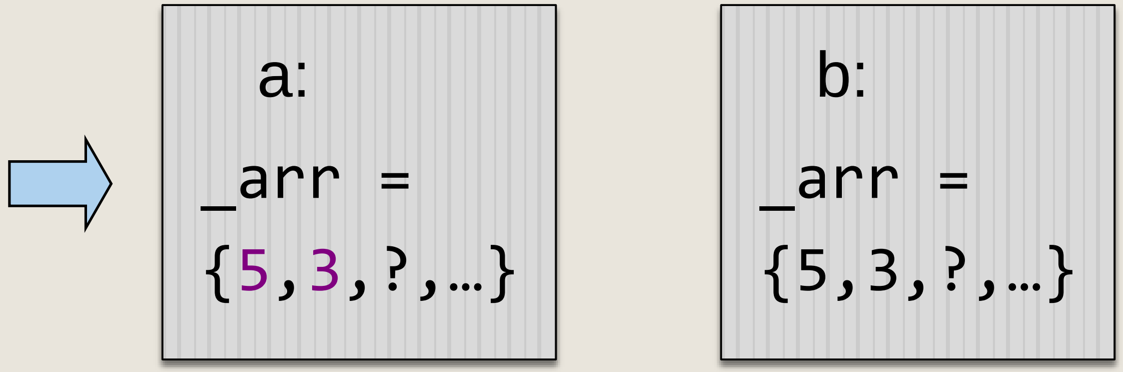
```
Vec a,b;
a._arr[0] = 5;
a._arr[1] = 3;
b = a;
```

Arrays in structs copying

Copy struct using ‘=‘:

copy just struct values!!!

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



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 '=':

copy 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:

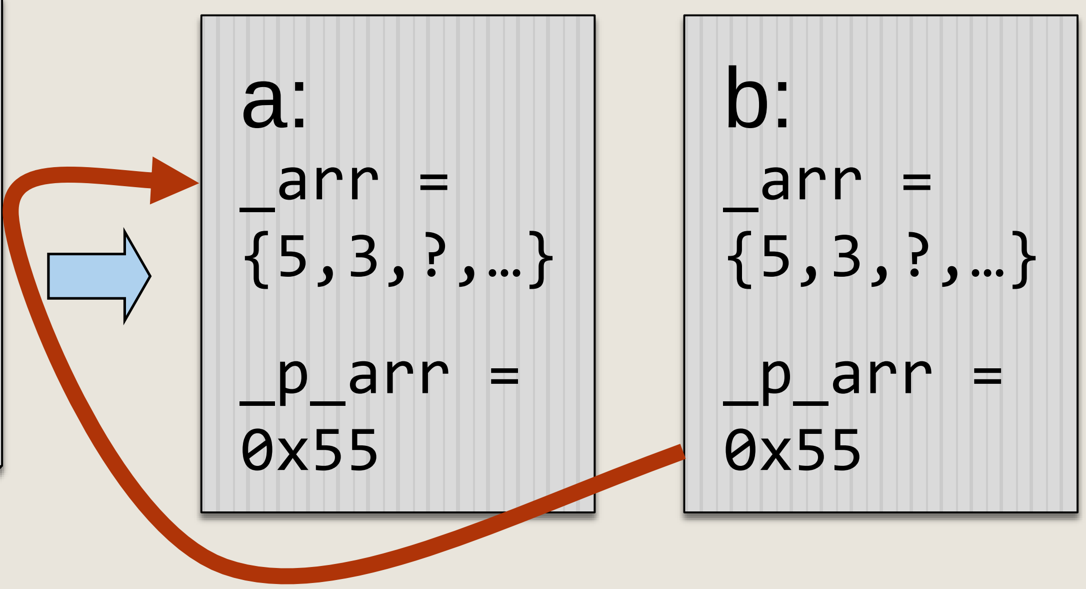
```
_arr =
{?,?,?,…}

_p_arr =
?
```

Pointers in structs copying

Copy structs using ‘=‘:

copy 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 deep copy structs correctly?

Implement a clone function:

```
Vec* Vec_clone (const Vec* v)
{
    ...
```

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] = -1;
}
```

```
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
```