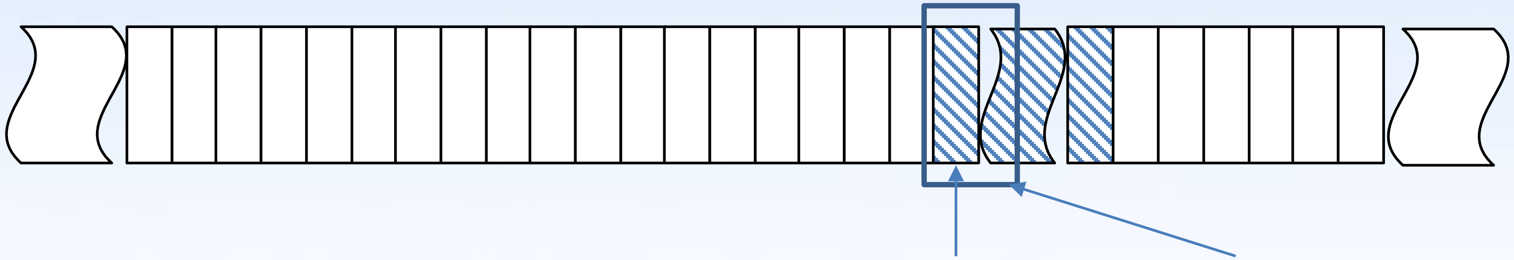
什么是alignment（对齐）

当声明一个类型为*T*的完全对象（例如*T O*），变量*O*对应那个对象如下



对象首地址　　　　字节范围

objects ... with addresses that are particular multiples of a byte address

An implementation-defined integer value representing the number of bytes between successive addresses at which a given object can be allocated

对齐值必须是2的n次方，例如1、2、4、8、16、32。。。

大多数编译器对alignment的处理

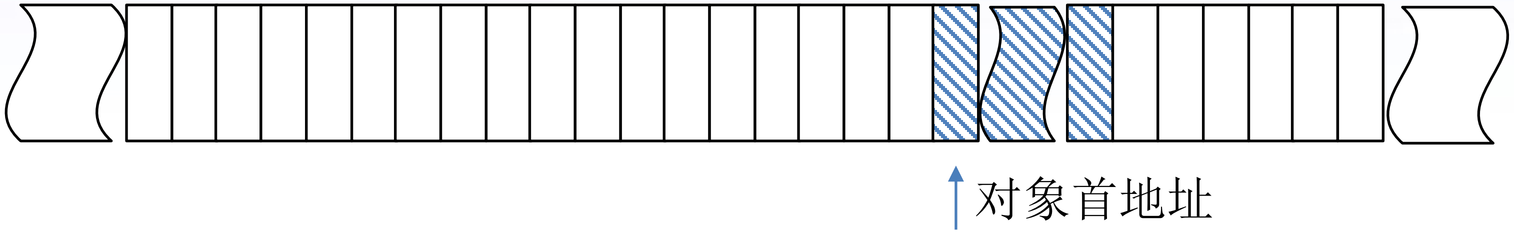
对齐的要求和编译器、硬件系统等紧密相关

各类编译器对同样的数据类型可能有不同的对齐要求

利用_Alignof(*T*)可以获得对象类型*T*的Alignment（对齐要求）

利用_Alignof(*O*)可以获得对象*O*的Alignment（对齐要求）

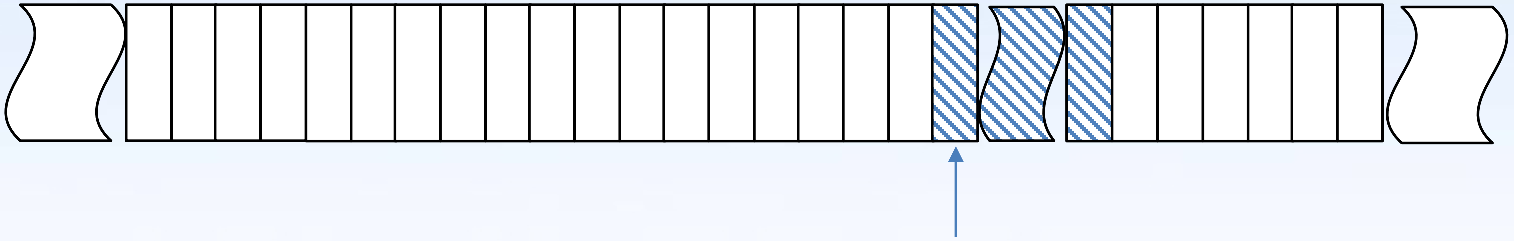
对象*O*的Alignment缺省等于对象类型*T*的Alignment



对象首地址 % 对象的Alignment = 0

alignment示例

当声明一个类型为*T*的完全对象（例如*T O*），变量*O*对应那个对象的首地址有要求

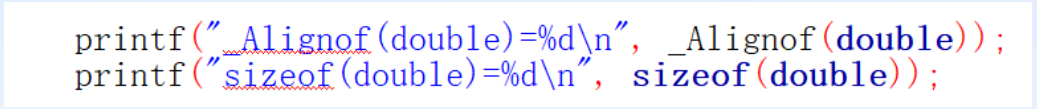


对象首地址

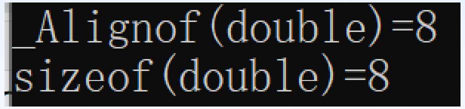
char a	short a	int a	double a
_Alignof(a)返回1	_Alignof(a)返回2	_Alignof(a)返回4	_Alignof(a)返回?
对象首地址 % 1 = 0	对象首地址 % 2 = 0	对象首地址 % 4 = 0	对象首地址 % ? = 0

对齐与具体实现紧密相关

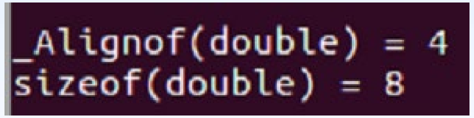
以double为例



```
printf("_Alignof(double)=%d\n", _Alignof(double));
printf("sizeof(double)=%d\n", sizeof(double));
```



```
_Alignof(double)=8
sizeof(double)=8
```



```
_Alignof(double) = 4
sizeof(double) = 8
```

_Alignof(char)一定等于1吗？

char, signed char, and unsigned char shall have the weakest alignment requirement

后续我们假设：_Alignof(char)=1, _Alignof(short)=2, _Alignof(int)=4, _Alignof(double)=8

_Alignas修改对齐要求

C语言标准规定编译器必须支持的对齐叫做fundamental alignment

fundamental alignment <= _Alignof(max_align_t)

max_align_t是一个类型，拥有最大的基础对齐要求

这个值由实现去约定，目前主流编译器一般是8或16

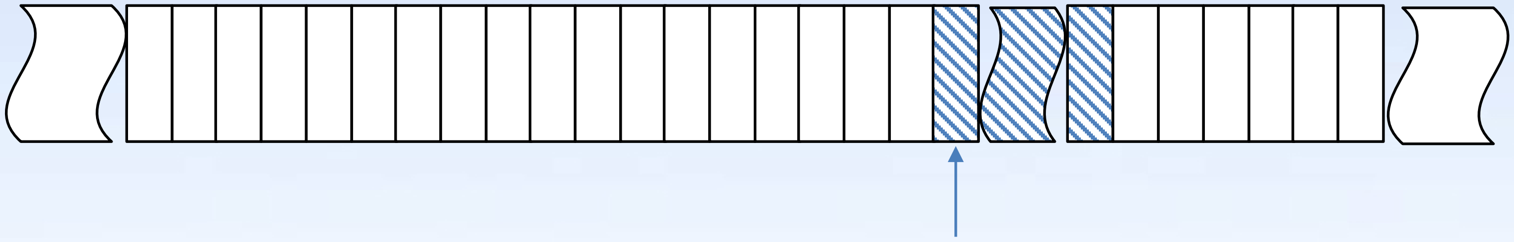
当声明一个*T*类型的对象*O*

可以为该对象设置更大（Stricter）的对齐要求：_Alignas(*N*) *T O*

N >= _Alignof(*T*) （注意：如果N大于_Alignof(max_align_t)，编译器决定是否支持）

思考：_Alignof(*O*) 总是等于_Alignof(*T*) ？

_Alignas修改对齐要求示例：int



对象首地址

_Alignas(64) int a

_Alignof(a)返回<64, size_t>

对象首地址 % 64 = 0

sizeof(a) = ? _Alignof(int) = ?

sizeof(a) = 4, _Alignof(int) = 4

修改对象的alignment不改变对象类型的对齐大小，也不改变对象的大小

_Alignas修改对齐要求示例：int[2]

当声明一个*T*类型的对象*O*，如果*T*是一个数组类型，元素类型为*E*
_Alignof(*T*) = _Alignof(*E*)

例如：_Alignof(int[3][4][5]) = _Alignof(int[4][5]) = _Alignof(int[5]) = _Alignof(int)

_Alignas(64) int a[2]

_Alignof(a)返回<64, size_t>

对象首地址 % 64 = 0

sizeof(a) = ?, _Alignof(int[2]) = ?

sizeof(a) = 8, _Alignof(int[2]) = 4

修改对象的alignment不改变对象类型的对齐大小，也不改变对象的大小

结构体类型的对齐要求

当声明一个*T*类型的对象*O*，如果*T*是一个结构体类型，成员对象分别是E_i

_Alignof(*T*) = max{_Alignof(E_i), 1<i<=结构体成员个数}

```
struct stru {
  char a;
  short b;
  int c;
  double d;
} s;
```

_Alignof(struct stru)

返回<8, size_t>

```
struct stru {
  char a;
  short b;
  int c[10];
  double d;
} s;
```

Alignof(struct stru)

返回<8, size_t>

```
struct stru {
  char a;
  _Alignas(32) short b;
  int c[10];
  double d;
} s;
```

_Alignof(struct stru)

返回<32, size_t>

修改结构体对象的对齐要求

```
_Alignas(64) struct stru {
  char a;
  _Alignas(32) short b;
  int c[10];
  double d;
} s;
```

_Alignof(struct stru) 返回<32, size_t>

_Alignof(s) 返回<64, size_t>

修改对象的alignment不改变对象类型的对齐大小，也不改变对象的大小

结构体类型的大小怎么计算的？

结构体对象的size

一个结构体类型T，成员对象分别是E_i，$1<i<=n$ （假设有n个成员对象）

_Alignof(T)是这个结构体类型T的对齐要求

结构体第1个成员对象E_1的首地址就是结构体的首地址，地址偏移量offset为0

结构体第2个成员对象E_2的地址偏移量确定方法如下

offset += sizeof(E_1)

internal padding

E_2的首地址偏移量 : offset += $\boxed{\text{offset \% \_Alignof}(E_2) == 0\ ?\ 0 : (\text{\_Alignof}(E_2) - \text{offset \% \_Alignof}(E_2)}$

...

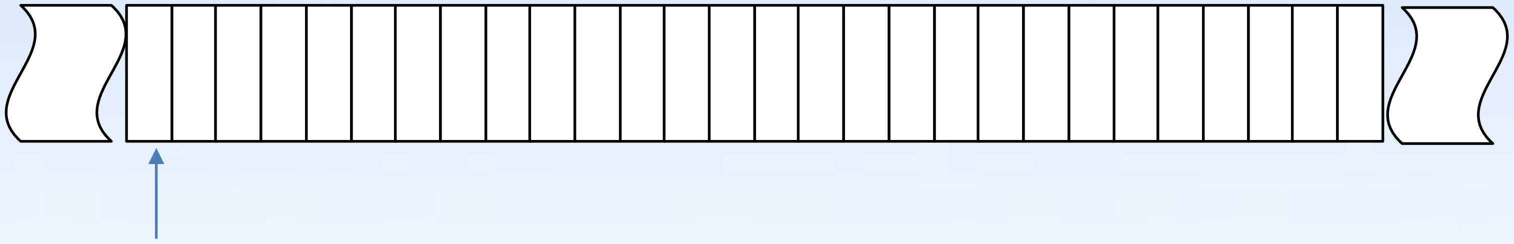
offset += sizeof(E_n)

trailing padding

sizeof(T) = offset + $\boxed{\text{offset \% \_Alignof}(T) == 0\ ?\ 0 : (\text{\_Alignof}(T) - \text{offset \% \_Alignof}(T)}$

结构体对象的size示例

```
_Alignas(64) struct stru {
  char a;
 _Alignas(32) short b;
  int c[10];
  double d;
} s;

_Alignof(struct stru)=32
_Alignof(s)=64
```



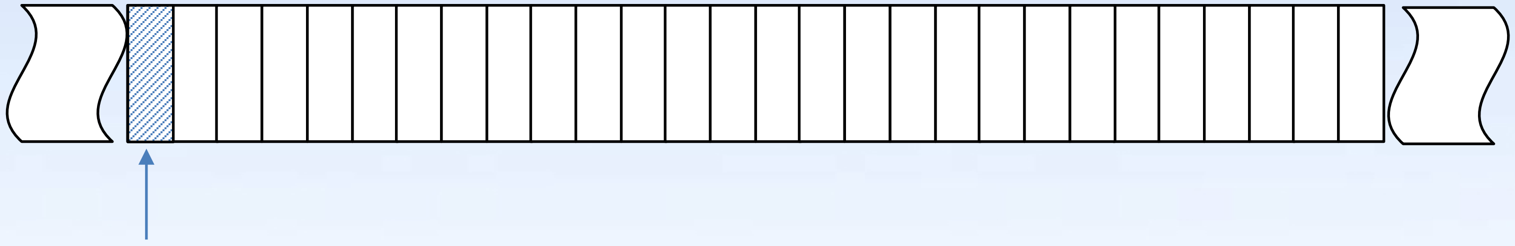
0x0061FD80

假设对象s的首地址是0x0061FD80（可以被64整除）

结构体对象的size示例：确定s. a的offset

```
_Alignas(64) struct stru {
char a;
 _Alignas(32) short b;
 int c[10];
 double d;
} s;
```

_Alignof(struct stru)=32
_Alignof(s)=64



0x0061FD80

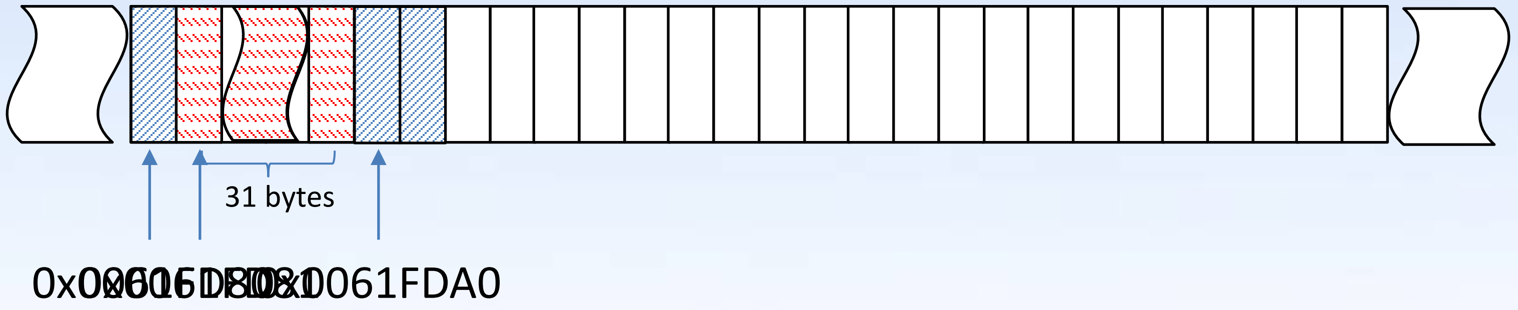
1、s.a的地址偏移量offset=0，即s.a的地址也是0x0061FD80

2、sizeof(s.a) = 1

结构体对象的size示例：确定s.b的offset

```
_Alignas(64) struct stru {
  char a;
  _Alignas(32) short b;
  int c[10];
  double d;
} s;
```

_Alignof(struct stru)=32
_Alignof(s)=64



1、 offset += sizeof(s.a), offset结果为1

2、 _Alignof(s.b) = 32

internal padding

3、 offset += $\boxed{\text{offset \% \_Alignof(s.b) ==0 ? 0: (\_Alignof(s.b) - offset \% \_Alignof(s.b))}}$

4、 offset结果为32

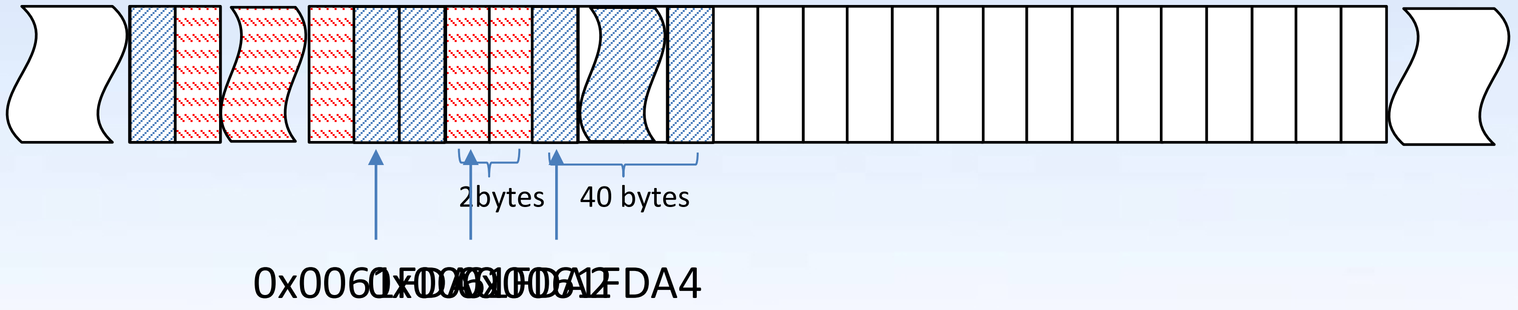
5、 sizeof(s.b) = 2

结构体对象的size示例:确定s.c的offset

```
_Alignas(64) struct stru {
  char a;
  _Alignas(32) short b;
  int c[10];
  double d;
} s;
```

_Alignof(struct stru)=32

_Alignof(s)=64



1、 offset += sizeof(s.b), offset结果为34

2、 _Alignof(s.c) = 4

internal padding

3、 offset += $\boxed{\text{offset \% \_Alignof(s.c) ==0 ? 0: (\_Alignof(s.c) - offset \% \_Alignof(s.c))}}$

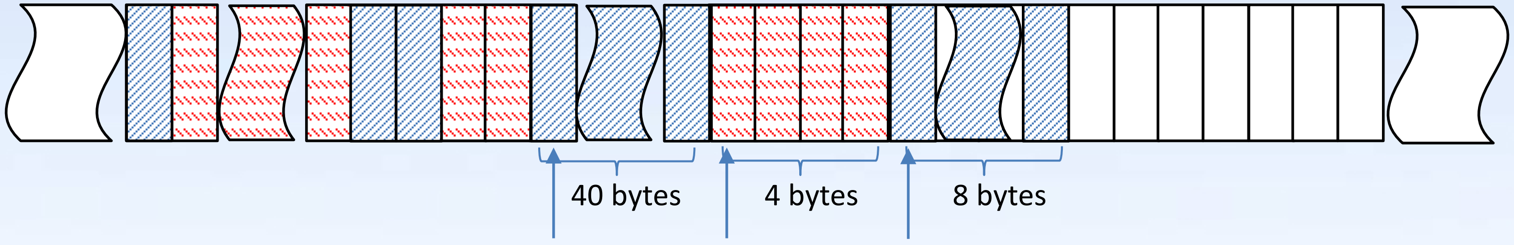
4、 offset结果为36

5、 sizeof(s.c) = 40

结构体对象的size示例:确定s. d的offset

```
_Alignas(64) struct stru {
  char a;
  _Alignas(32) short b;
  int c[10];
  double d;
} s;
```

_Alignof(struct stru)=32
_Alignof(s)=64



0x0061FDA4 0x0061FDCC 0x0061FDD0

1、 offset += sizeof(s.c), offset结果为76

2、 _Alignof(s.d) = 8

internal padding

3、 offset += offset % _Alignof(s.d) ==0 ? 0: (_Alignof(s.d) - offset % _Alignof(s.d))

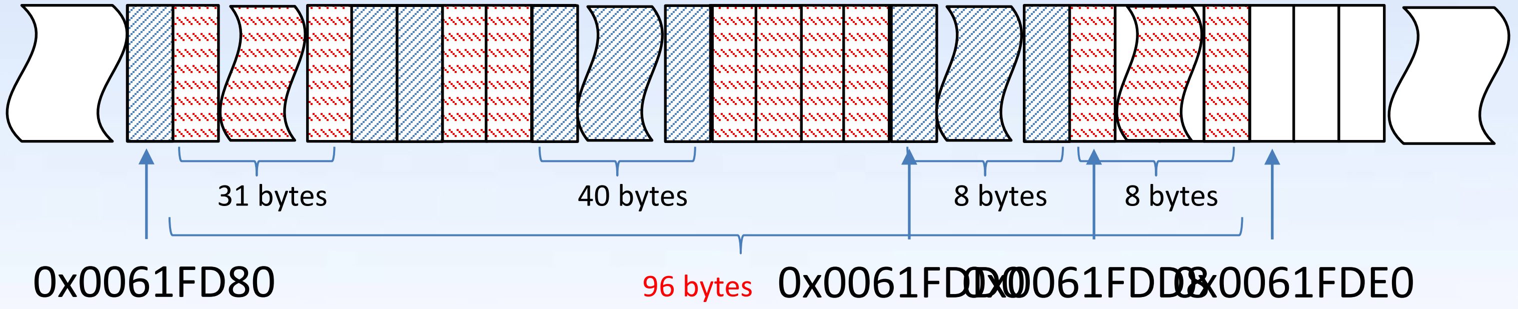
4、 offset结果为80

5、 sizeof(s.d) = 8

结构体对象的size示例：确定s的size

```
_Alignas(64) struct stru {
  char a;
 _Alignas(32) short b;
  int c[10];
  double d;
} s;
```

_Alignof(struct stru)=32
_Alignof(s)=64



1、 offset += sizeof(s.d), offset结果为88

2、 _Alignof(struct stru) = 32

trailing padding

3、 offset += $\boxed{\text{offset \% \_Alignof(struct stru) ==0 ? 0: (\_Alignof(struct stru) - offset \% \_Alignof(struct stru))}}$

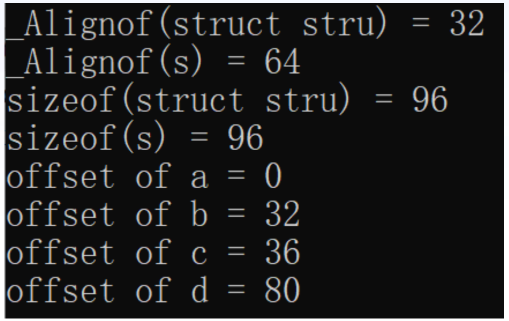
4、 offset结果为96

5、 sizeof(s) = 96

结构体对象的size示例：确定s的size

```
_Alignas(64) struct stru {
  char a;
 _Alignas(32) short b;
  int c[10];
  double d;
} s;
```

```
printf("_Alignof(struct stru) = %d\n", _Alignof(struct stru));
printf("_Alignof(s) = %d\n", _Alignof(s));
printf("sizeof(struct stru) = %d\n", sizeof(struct stru));
printf("sizeof(s) = %d\n", sizeof(s));
printf("offset of a = %d\n", offsetof(struct stru, a));
printf("offset of b = %d\n", offsetof(struct stru, b));
printf("offset of c = %d\n", offsetof(struct stru, c));
printf("offset of d = %d\n", offsetof(struct stru, d));
```



```
_Alignof(struct stru) = 32
_Alignof(s) = 64
sizeof(struct stru) = 96
sizeof(s) = 96
offset of a = 0
offset of b = 32
offset of c = 36
offset of d = 80
```

#pragma pack(*n*)调整结构体对象的对齐

利用#pragma pack(*n*)可以进一步调整结构体的对齐要求，规则如下

当声明一个*T*类型的对象*O*，如果*T*是一个结构体类型，成员对象分别是E_i

1、_Alignof(E_i) = min{_Alignof (E_i) , *n*}

2、_Alignof(*T*) = max{_Alignof(E_i), 1<i<=结构体成员个数}

```
#pragma pack(1)

_Alignas(64) struct stru {
  char a;
 _Alignas(32) short b;
  int c[10];
  double d;
} s;
```

_Alignof(s.a) = 1 _Alignof(s.b)=1 _Alignof(s.c) =1 _Alignof(s.d) = 1

_Alignof(struct stru) = 1

_Alignof(s) = 64

pack(*n*)针对的是结构体内部的对象对齐要求
结构体对象不受影响

#pragma pack(*n*)示例1

```
#pragma pack(1)

_Alignas(64) struct stru {
  char a;
  _Alignas(32) short b;
  int c[10];
  double d;
} s;
```

_Alignof(struct stru) = 1   _Alignof(s) = 64

_Alignof(s.a) = 1 _Alignof(s.b)=1 _Alignof(s.c) =1 _Alignof(s.d) = 1

假设对象s的首地址是0x0061FD80（可以被64整除）

1、s.a的offset等于0，且sizeof(s.a)=1
2、offset += sizeof(s.a)，结果为1
3、offset % _Alignof(s.b)，结果为0，s.b的offset为1
4、offset += sizeof(s.b)，结果为3
5、offset % _Alignof(s.c)，结果为0，s.c的offset为3
6、offset += sizeof(s.c)，结果为43
7、offset % _Alignof(s.d)，结果为0，s.d的offset为43
8、offset += sizeof(s.d)，结果为51
9、offset % _Alignof(struct stru)，结果为0，sizeof(s)为51

#pragma pack(*n*)示例2

```
#pragma pack(4)

_Alignas(64) struct stru {
  char a;
 _Alignas(32) short b;
  int c[10];
  double d;
} s;
```

_Alignof(struct stru) = 4 _Alignof(s) = 64

_Alignof(s.a) = 1 _Alignof(s.b)=4 _Alignof(s.c) =4 _Alignof(s.d) = 4

假设对象s的首地址是0x0061FD80（可以被64整除）

1、s.a的offset等于0， 且sizeof(s.a)=1
2、offset += sizeof(s.a)， 结果为1
3、offset % _Alignof(s.b)， 结果为1， s.b的offset为4 (3个padding字节)
4、offset += sizeof(s.b)， 结果为6
5、offset % _Alignof(s.c)， 结果为2， s.c的offset为8 (2个padding字节)
6、offset += sizeof(s.c)， 结果为48
7、offset % _Alignof(s.d)， 结果为0， s.d的offset为48 (没有padding)
8、offset += sizeof(s.d)， 结果为56
9、offset % _Alignof(struct stru)， 结果为0， sizeof(s)为56 (没有padding)

#pragma pack(*n*)示例3

```
#pragma pack(8)

_Alignas(64) struct stru {
  char a;
 _Alignas(32) short b;
  int c[10];
  double d;
} s;
```

_Alignof(struct stru) = 8 _Alignof(s) = 64

_Alignof(s.a) = 1 _Alignof(s.b)=8 _Alignof(s.c) =4 _Alignof(s.d) = 8

假设对象s的首地址是0x0061FD80（可以被64整除）

1、s.a的offset=? sizeof(s.a)=?

2、s.b的offset=? sizeof(s.b)=?

3、s.c的offset=? sizeof(s.c)=?

4、s.d的offset=? sizeof(s.d)=?

5、sizeof(s) =?

#pragma pack(*n*)示例3

```
#pragma pack(8)

_Alignas(64) struct stru {
  char a;
 _Alignas(32) short b;
  int c[10];
  double d;
} s;
```

_Alignof(struct stru) = 8 _Alignof(s) = 64

_Alignof(s.a) = 1 _Alignof(s.b)=8 _Alignof(s.c) =4 _Alignof(s.d) = 8

假设对象s的首地址是0x0061FD80（可以被64整除）

1、s.a的offset等于0， 且sizeof(s.a)=1
2、offset += sizeof(s.a)， 结果为1
3、offset % _Alignof(s.b)， 结果为1， s.b的offset为8 (7个padding字节)
4、offset += sizeof(s.b)， 结果为10
5、offset % _Alignof(s.c)， 结果为2， s.c的offset为12 (2个padding字节)
6、offset += sizeof(s.c)， 结果为52
7、offset % _Alignof(s.d)， 结果为0， s.d的offset为56 (4个padding字节)
8、offset += sizeof(s.d)， 结果为64
9、offset % _Alignof(struct stru)， 结果为0， sizeof(s)为64 (没有padding)

不要随意使用#pragma pack(*n*)

除非有确定的需求，充分了解带来的潜在效率风险
否则不要随意使用#pragma pack(*n*)

malloc返回的指针对齐

void* p = malloc(size)

指针p指向的这块内存满足基础对齐要求，因此一般来说

p的地址 % _Alignof(max_align_t) = 0

如果需要获得指定对齐要求的空间，可以使用

void *aligned_alloc(size_t alignment, size_t size);

指针转换中的对齐问题

由于对齐的问题，指针的强制转换需要注意

指针的强制转换隐含着地址对应的对象类型发生了变化

如果转换后的指针对应的对象对齐方式不正确，则指针的强制转换行为是未定义行为

char a[10]，假设对象a的首地址 % 2 = 0

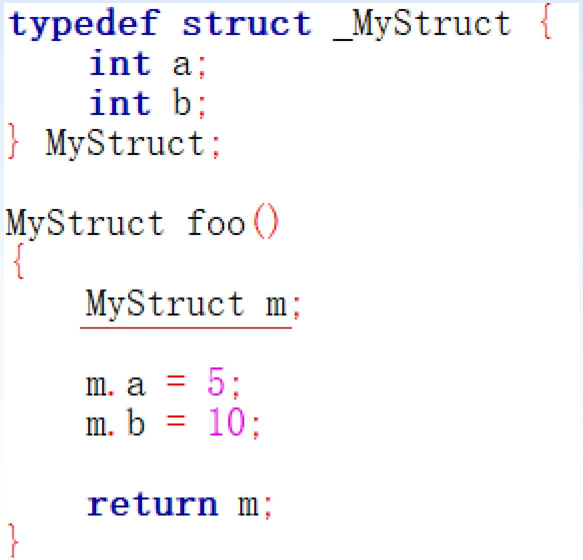
a+1一定是奇数，除以4的余数肯定不为0

(int*)(a+1)是未定义行为

如果对象a的首地址 % 2 = 0，但 % 4 ≠ 0

(int*)(a)也是未定义行为

关于结构体lvalue的问题



```
typedef struct _MyStruct {
    int a;
    int b;
} MyStruct;

MyStruct foo()
{
    MyStruct m;

    m.a = 5;
    m.b = 10;

    return m;
}
```

对于foo函数中的变量m

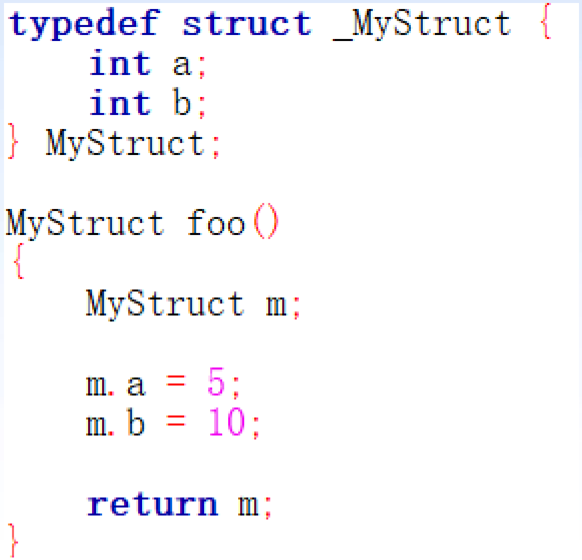
1、m是lvalue吗？
2、m.a是lvalue吗？
3、m.b是lvalue吗？

m、m.a、m.b都是lvalue
&m, &m.a, &m.b都是合法的

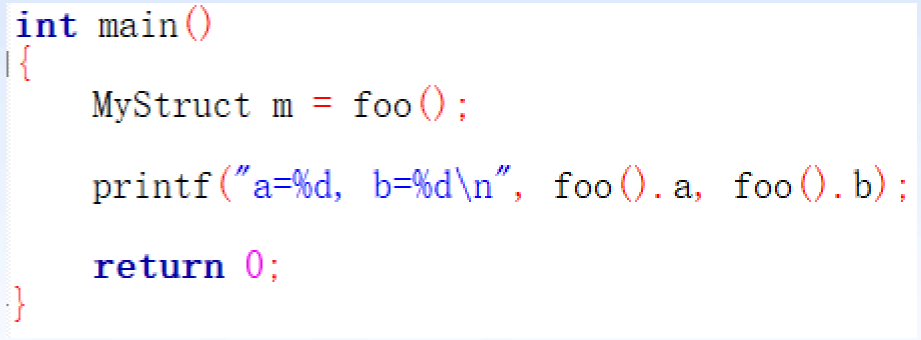
A postfix expression followed by the . operator and an identifier designates a member of a structure or union object. The value is that of the named member, and is an lvalue if the first expression is an lvalue.

A postfix expression followed by the-> operator and an identifier designates a member of a structure or union object. The value is that of the named member of the object to which the first expression points, and is an lvalue.

关于结构体lvalue的问题（续）



```
typedef struct _MyStruct {
    int a;
    int b;
} MyStruct;

MyStruct foo()
{
    MyStruct m;

    m.a = 5;
    m.b = 10;

    return m;
}
```



```
int main()
{
    MyStruct m = foo();

    printf("a=%d, b=%d\n", foo().a, foo().b);

    return 0;
}
```

foo()是lvalue吗?　　　不是

If f is a function returning a structure or union, and x is a member of that structure or union, f().x is a valid postfix expression but is not an lvalue.

对象的存储周期(Storage Duration)

任何一个对象都有生命周期（lifetime)，决定生命周期的就是对象的
Storage Duration

在生命周期内
1、系统确保对象的内存有效
2、对象的地址是constant address，在对象声明周期内，地址不变
3、对象保有最后赋值(last-stored value)不变

超出对象生命周期之外对对象的访问是未定义行为

对象的4种存储周期

C语言定义了4种存储周期： **static**, **thread**, **automatic**, **allocated**

Static存储周期的对象

Its lifetime is the entire execution of the program and its stored value is initialized only once, prior to program startup.

Automatic存储周期的对象

An object whose identifier is declared with no linkage and without the storage-class specifier static has automatic storage duration

allocated存储周期的对象

依靠malloc这些内存管理函数分配的空间， 需要调用free释放

Storage-class Specifier

C语言定义了如下Storage-class specifier

typedef
extern
static
thread_local
auto
register

Storage-class specifier

typedef、extern、static、thread_local、auto、register

The typedef specifier is called a "storage-class specifier" for syntactic convenience only
Typedef没有定义新对象类型，只是提供更好的书写的便利性

register告诉编译器越快越好，但编译器可以不理会

register修饰的对象不能取地址

```
register int a;
&a
```

✗

标识符的Scope

Identifier（标识符）是C语言中最基础的一种Symbol

标识符可以用来识别：

an object; a function; a tag or a member of a structure, union, or enumeration; a typedef name; a label name; a macro name; or a macro parameter.

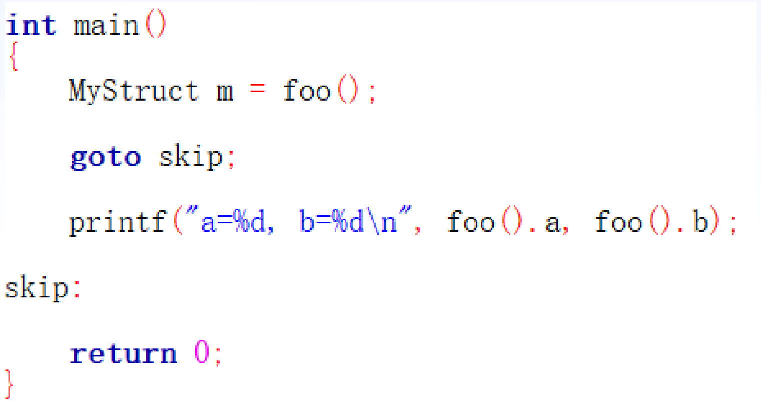
For each different entity that an identifier designates, the identifier is visible (i.e., can be used) only within a region of program text called its **scope**.

Scope的分类:Function Scope

C语言一共有4种Scope

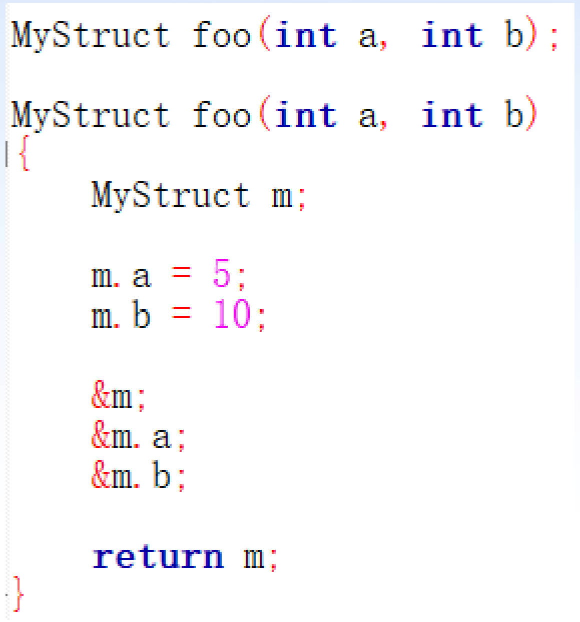
function, file, block, and function prototype

A label name is the only kind of identifier that has function scope.



```
int main()
{
    MyStruct m = foo();

    goto skip;

    printf("a=%d, b=%d\n", foo().a, foo().b);

skip:

    return 0;
}
```

Scope的分类：Function Prototype Scope



```
MyStruct foo(int a, int b);

MyStruct foo(int a, int b)
{
    MyStruct m;

    m.a = 5;
    m.b = 10;

    &m;
    &m.a;
    &m.b;

    return m;
}
```

the identifier appears within the list of parameter declarations in a function prototype (not part of a function definition), the identifier has function prototype scope, which terminates at the end of the function declarator.

注意区分函数原型说明和函数定义

Scope的分类：Block Scope vs. File Scope

If the declarator or type specifier that declares the identifier appears outside of any block or list of parameters, the identifier has file scope, which terminates at the end of the translation unit. (A C program need not all be translated at the same time. The text of the program is kept in units called source files. A source file together with all the headers and source files included via the preprocessing directive #include is known as a preprocessing translation unit. After preprocessing, a preprocessing translation unit is called a translation unit).

If the declarator or type specifier that declares the identifier appears inside a block or within the list of parameter declarations in a function definition, the identifier has block scope, which terminates at the end of the associated block.

Identifier是否位于一个block或者作为函数实参

了解Linkage

相同的标识符出现在不同的地方，他们是否表示同一个实体呢？

C语言依靠一个叫做linkage的机制来进行分辨

An identifier declared in different scopes or in the same scope more than once can be made to refer to the same object or function by a process called linkage.

C语言有三种Linkage：external、internal、none

no linkage

The following identifiers have no linkage:

1、 an identifier declared to be anything other than an object or a function;

2、 an identifier declared to be a function parameter;

3、 a block scope identifier for an object declared without the storage-class specifier extern.

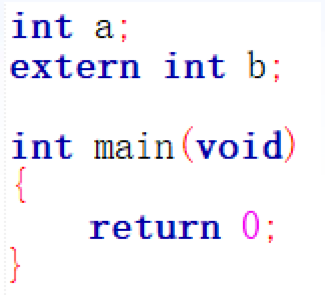
```c
int main(void)
{
    int a;

    return 0;
}
```

```c
int main(void)
{
    static int a;

    return 0;
}
```

no linkage的标识符都指向不同的对象

external linkage

For an identifier declared with the storage-class specifier extern in a scope in which a prior declaration of that identifier is visible, if the prior declaration specifies internal or external linkage, the linkage of the identifier at the later declaration is the same as the linkage specified at the prior declaration. If no prior declaration is visible, or if the prior declaration specifies no linkage, then the identifier has external linkage.

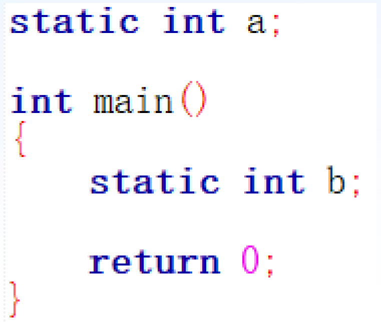


```c
int a;
extern int b;

int main(void)
{
    return 0;
}
```

Internal Linkage

具有file Scope的对象/函数标识符+static，具有internal linkage

If the declaration of a file scope identifier for an object or a function contains the storage-class specifier static, the identifier has internal linkage.



```
static int a;

int main()
{
    static int b;

    return 0;
}
```

A function declaration can contain the storage-class specifier static only if it is at file scope