变长数组（Variable Length Array）

```
int a[10];
```

声明了一个对象a
其类型为int[10]

```
int m;
scanf("%d", &m);
int a[m];
```

声明了一个对象a
其类型为？

```
int m=1;
int a[m];
```

声明了一个对象a
其类型为？

变长数组（Variable Length Array）

变长数组类型：T[M]

变长数组VLA是一个数组类型，元素类型为T，元素个数是M

(M不是一个整数常量或者整数常量表达式) || (T是一个变长数组)

数组的长度(M的rvalue)*sizeof(T)必须在运行时才能确定

首次执行到T[M]时数组大小确定，确定之后不能再修改数组大小

变长数组（Variable Length Array）

```
int m;
scanf("%d", &m);
```

int a[m];	一维数组，大小为m，元素类型为int
int a[m][2];	一维数组，大小为m，元素类型为int[2]
int a[2][m];	一维数组，大小为2，元素类型为int[m]
int a[m][m];	一维数组，大小为m，元素类型为int[m]

常量

C语言包括5种常量（constant)

整数常量、浮点数常量、枚举常量、字符常量、预定义常量（C23）

integer-constant	10
floating-constant	10.0
enumeration-constant	enum DAYS {MONDAY, TUESDAY, WENSDAY};
character-constant	‘a’
predefined-constant	false, true, nullptr

A member of an enumeration is called an enumeration constant

常量表达式

常量表达式能在编译时进行evaluate，像常量一样使用

限制条件：

1、常量表达式不能包含赋值、自增/自减，函数调用，逗号运算符
除非他们包含在不被evaluate的子表达中

2、常量表达式evaluate之后的rvalue的取值范围，应该在该表达
rvalue类型的表征范围之内

整数常量表达式

1、表达式rvalue类型为整数类型

2、operand是

1）整数常量 1+2

2）字符常量 1+’a’

3）浮点数常量用cast转换成整数 1+(int)(5.0)

4）返回结果是整数常量的sizeof表达式 1+sizeof(int[2])

5）alignof表达式 1+alignof(int)

。。。（本课程不涉及）

思考题

1、int m = 1; m是不是整数常量表达式？

2、(int)(+5.0)是不是整数常量表达式？

都不是

3、int m = 1; sizeof(int[m])是不是整数常量表达式？

也不是，为什么？

常量表达式能在编译时进行evaluate

进一步理解sizeof

sizeof后面可以跟两大类，三种operand

1、sizeof(type-name)

2、sizeof(exp) / sizeof exp

2.1、sizeof(lvalue-exp) / sizeof lvalue-exp

2.2、sizeof(non-lvalue-exp) /sizeof non-lvalue-exp

理解sizeof(type-name)

1、如果type-name是Non-VLA，则
sizeof(type-name)在编译时得到type-name的大小，
其返回值是整数常量
type-name作为operand，整体不做evaluate

sizeof(int[2+3])

2、如果type-name是VLA，则
sizeof(type-name)不能在编译时得到type-name的大小，
其返回值不是整数常量，需要在运行时得到类型大小
type-name作为operand，子表达式需要evaluate

int m = 5; sizeof(int[m]); sizeof(int[++m]);

思考题

```
int i=1;
int j=3;

size_t s;

s = sizeof(int[++i]);
printf("i=%d\n", i);
printf("s=%zd\n", s);

s = sizeof(int[++i][++j]);
printf("i=%d, j=%d\n", i, j);
printf("s=%zd\n", s);
```

```
i=2
s=8
i=3, j=4
s=48

Process returned 0 (0x0)
Press any key to continue.
```

1、如果type-name是Non-VLA，则
sizeof(type-name)在编译时得到type-name的大小，
其返回值是整数常量

2、如果type-name是VLA，则
sizeof(type-name)不能在编译时得到type-name的大小，
其返回值不是整数常量，需要在运行时得到类型大小
意味着type-name作为operand，子表达式需要evaluate

sizeof(int[++i][++i])有什么问题？

理解sizeof(lvalue)

1、如果lvalue定位的对象类型是Non-VLA，则
sizeof(lvalue)在编译时得到lvalue定位对象的大小，
其返回值是整数常量
lvalue作为operand，整体不做evaluate

int a[10][10]; sizeof(a); sizeof(a[0]);

sizeof(a[0][0]);

2、如果lvalue定位的对象类型是VLA，则
sizeof(lvalue)不能在编译时得到lvalue定位对象的大小，
其返回值不是整数常量，需要在运行时得到对象大小
意味着lvalue作为operand，子表达式需要evaluate

int m = 5; int n=10; int a[m][n]; sizeof(a); sizeof(a[0]);

sizeof(a[0][0]);
a[0][0]是VLA？

思考题

```
int main()
{
    int i=1;
    int m=10;

    int a[10][10];
    int b[m][10];
    int c[10][m];

    size_t s;

    s = sizeof(a[++i]);
    printf("%d\n", i);
    s = sizeof(b[++i]);
    printf("%d\n", i);
    s = sizeof(c[++i]);
    printf("%d\n", i);

    return 0;
}
```

1、如果exp是lvalue，且对象类型是Non-VLA，则exp作为operand不做evaluate，意味着所有子表达式都不会做evaluate

2、如果exp是lvalue，且对象类型是VLA，则exp作为operand按语法做evaluate，意味着子表达式需要做evaluate，exp本身不做evaluate

```
1
1
2

Process returned 0 (0x0)
Press any key to continue.
```

进一步理解整数常量表达式

int a[1+2]; int b[1+'a']; int c[1+(int)5.0];

都不是VLA

int a[2][1+2]; int b[2][1+'a']; int c[2][1+(int)5.0];

用高维数组加深理解

给定int i=1;

sizeof(a[++i]); sizeof(b[++i]); sizeof(c[++i]);

a[++i]
b[++i]
c[++i]
都是lvalue
且都不是VLA

i等于多少，为什么？

i等于1，因为++i都没有被evaluate

sizeof(type-name)是否是常量表达式

```
int i=1;
int a[2][sizeof(int[5])];
sizeof(a[++i]);
printf("%d\n", i);
printf("%zd\n", sizeof(a));
```

sizeof(int[5])是整数常量表达式

a[++i]是lvalue
定位的对象类型是int[20]，非VLA
++i不做evaluate

```
1
160

Process returned 0 (0x0)
Press any key to continue.
```

sizeof(type-name)是否是常量表达式

```
int i=1;
int m=5;
int a[2][sizeof(int[m])];
sizeof(a[++i]);
printf("%d\n", i);
printf("%zd\n", sizeof(a));
```

sizeof(int[m])不是整数常量表达式

a[++i]是lvalue
定位的对象类型是VLA
++i就需要做evaluate

```
2
160

Process returned 0 (0x0)
Press any key to continue.
```

sizeof(type-name)是否是常量表达式

```
int i=1;
int m=5;
int a[2][sizeof(int[++m])];
sizeof(a[++i]);
printf("%d\n", i);
printf("%zd\n", sizeof(a));
```

sizeof(int[++m])不是整数常量表达式
++m要做evaluate，++m的rvalue等于6

a[++i]是lvalue
定位的对象类型是VLA
++i就需要做evaluate

```
2
192

Process returned 0 (0x0)
Press any key to continue.
```

进一步理解整数常量表达式

```
int i=1;
int m=5;
int a[sizeof(int[++m])][2];
sizeof(a[++i]);
printf("%d\n", i);
printf("%zd\n", sizeof(a));
```

sizeof(int[++m])不是整数常量表达式
++m要做evaluate，++m的rvalue等于6

a[++i]是lvalue
定位的对象类型是int[2]，非VLA
++i就不需要做evaluate

```
1
192

Process returned 0 (0x0)
Press any key to continue.
```

进一步理解整数常量表达式

```
int i=1;
int a[2][1+(int)(+5.0)];
sizeof(a[++i]);
printf("%d\n", i);
printf("%zd\n", sizeof(a));
```

1+(int)(+5.0)不是整数常量表达式

…that are the immediate operands of casts

现在cast的operand是+5.0，不是5.0

1+(int)5.0是常量表达式，注意区别

```
2
48

Process returned 0 (0x0)
Press any key to continue.
```

再思考：const int m=5;m是不是常量

```
int i=1;
const int m=5;
int a[2][m];
int b[2][sizeof(int[m])];
sizeof(a[++i]);
sizeof(b[++i]);
printf("%d\n", i);
printf("%zd\n", sizeof(a));
printf("%zd\n", sizeof(b));
```

如果m是整数常量，则：

1、m应该是整数常量表达式

2、sizeof(int[m])应该是整数常量表达式

3、a[++i]这个lvalue定位对象类型应该是非VLA

4、b[++i]这个lvalue定位对象类型应该是非VLA

5、++i则不应该被evaluate

6、i的值应该是1

实际上

```
3
40
160

Process returned 0 (0x0)
Press any key to continue.
```

alignof(type-name)是整数常量表达式

```
int i=1;
int m=5;
int a[2][alignof(int[++m])];
sizeof(a[++i]);
printf("%d\n", m);
printf("%d\n", i);
```

alignof(operand)返回的就是operand的对齐要求
返回值是一个整数常量表达式
operand不会做evaluate

alignof ⇔ _Alignof

```
5
1

Process returned 0 (0x0)
Press any key to continue.
```

理解sizeof(non-lvalue)

如果表达式是一个non-lvalue，则
1、sizeof(non-lvalue)返回这个non-lvalue做evaluate之后rvalue类型的大小
2、这个non-lvalue并不会真的做evaluate

```
int i=1;
int a[2][5];
sizeof(a[++i]);
printf("%d\n", i);
sizeof(a[++i]+1);
printf("%d\n", i);
```

a[++i]是lvalue，定位对象为非VLA
a[++i]+1为non-lvalue

```
1
1

Process returned 0 (0x0)
Press any key to continue.
```

理解sizeof(non-lvalue)

如果表达式是一个non-lvalue，则

1、sizeof(non-lvalue)返回这个non-lvalue做evaluate之后rvalue类型的大小

2、这个non-lvalue并不会真的做evaluate

```
int i=1;
const int m=5;
int a[2][sizeof(int[m])];
sizeof(a[++i]);
printf("%d\n", i);
sizeof(a[++i]+1);
printf("%d\n", i);
```

a[++i]是lvalue，定位对象为VLA

a[++i]+1为non-lvalue

```
2
2

Process returned 0 (0x0)
Press any key to continue.
```

sizeof用于char的疑惑

1、对象类型的大小

sizeof(type_name)

例如：

sizeof(char)的结果为 1

2、表达式返回值类型的大小

sizeof(expression) 或 sizeof expression

例如：给定char c = ‘a’;

sizeof(c)的结果为 1

sizeof(c++)的结果为 1

sizeof(‘a’)的结果是?

‘a’是什么类型？

‘a’, ‘b’, ‘c’在C语言中视为Integer Character Constant

An integer character constant has type int

sizeof(‘a’)的结果是4

sizeof(‘a’)在C和C++环境下表现不一致
C++中，sizeof(‘a’)返回是1

VLA当函数参数

```
void foo(int row, int column, int a[row][column])
{
    int i=1;
    sizeof(a[++i]);
    printf("%d", i);
}

int main()
{
    int m=1;
    int n=5;
    int a[m][n];

    foo(m, n, a);

    return 0;
}
```

```
2
Process returned 0 (0x0)
Press any key to continue.
```

函数foo中的a是VLA吗？ 不是

main函数中调用foo函数的时候，a做evaluate，其rvalue类型是int(*)[n]

foo函数中的a类型也是一个指针类型int(*)[column]，其Referenced Type是一个VLA类型int[column]，因此a[++i]是一个VLA-lvalue，定位对象类型为int[column]

不同数组当函数参数的表现对比

```c
void foo(int row, int column, int a[1][5])
{
    int i=1;
    sizeof(a[++i]);
    printf("%d", i);
}

int main()
{
    int a[1][5];

    foo(1, 5, a);

    return 0;
}
```

```
1
Process returned 0 (0x0)
Press any key to continue.
```

```c
void foo(int row, int column, int a[row][column])
{
    int i=1;
    sizeof(a[++i]);
    printf("%d", i);
}

int main()
{
    int a[1][5];

    foo(1, 5, a);

    return 0;
}
```

```
2
Process returned 0 (0x0)
Press any key to continue.
```

```c
void foo(int row, int column, int a[1][5])
{
    int i=1;
    sizeof(a[++i]);
    printf("%d", i);
}

int main()
{
    int m=1;
    int n=5;

    int a[m][n];

    foo(m, n, a);

    return 0;
}
```

```
1
Process returned 0 (0x0)
Press any key to continue.
```

针对VLA，typeof处理跟sizeof逻辑一样

```
int m = 10;
int a[5][m];
int i=1;

typeof(a[++i]) c;
printf("%zd\n", sizeof(c));
printf("%d\n", i);
```

```
MinGW clang 16.0.2
Program returned: 0
Program stdout
40
2
```

变长数组总结

变长数组类型：T[M]

变长数组是不完全对象类型，长度确定的时候完全化

变长数组的长度一旦确定，就不能再更改

变长数组不是长度可以一直变来变去的数组

VLA-lvalue作为operand是否要evaluate是一个知识难点

思考题

```
int i=1;
int j;
int a[5][j=10];

sizeof(a[++i]);
printf("%d", i);
```

i = 2
为什么

j=10不是常量表达式

Constant expressions shall not contain assignment, increment, decrement, function-call, or comma operators, except when they are contained within a subexpression that is not evaluated

进一步了解size、padding和alignment

1、了解对象的size、padding和alignment的基本概念

2、通过结构体的实际应用来加深size、padding和alignment的理解

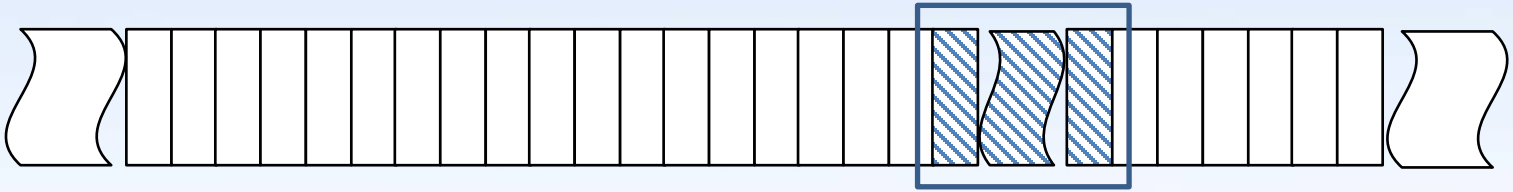
回顾一下：1 byte一定等于8 bit？

int的一定占4个字节吗？

声明一个char* p，(int*)p会有问题吗？

size的内涵

给定一个完全对象类型*T*，声明一个对象*O*（即*T O*），其含义是什么呢？



分配了一系列字节，这段内存的对象类型为*T*，且用*O*这个标识符可以定位

注意：*O*不是对象（*O*是标识符，是表达式），这段内存才是对象

sizeof(*T*) vs. sizeof(*O*)

sizeof(*T*)返回值的含义是为一个类型*T*的对象分配空间需要多少个字节

sizeof(*O*)返回值的含义是用标识符*O*来定位的那个对象共占用了多少个字节

理解padding：无符号整数

假设一个无符号整数类型*T*，一共占用*n* × CHAR_BIT个bit位

sizeof(*T*)返回值为<*n*, size_t>

这个*n*×CHAR_BIT个bit分为以下两个部分：

1、value bits
2、padding bits

示意图，不意味着必须这么摆放

假设value bits的个数是*N*，该无符号整数对象表值范围为0 ~ 2^N-1
这个*N*值就被称为这个无符号整数类型*T*的宽度（width）

unsigned char类型不允许有padding bits，其他无符号整数类型可以没有padding bits

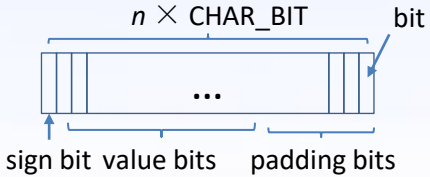
理解padding：有符号整数

假设一个有符号整数类型*T*，一共占用*n* × CHAR_BIT个bit位

sizeof(*T*)返回值为<*n*, size_t>

这个*n*×CHAR_BIT个bit分为以下三个部分：

1、sign bit
2、value bits
3、padding bits



示意图，不意味着必须这么摆放

假设sign bit+value bits的个数是*N*，该有符号整数对象表值范围为$-(2^{N-1})\sim 2^{N-1}-1$
这个*N*值就被称为这个有符号整数类型*T*的宽度（width）

signed char类型不允许有padding bits，其他有符号整数类型可以有padding bits

char和int在size和padding上的区别

C语言规定：

1、sizeof(unsigned char)/sizeof(signed char)恒等于1

2、unsigned char/signed char类型不允许有padding bits

结论：signed/unsigned char的大小一定是1个byte，且没有padding

例如对于一个unsigned char类型的对象，其取值范围为 0 to $2^{CHAR\_BIT} - 1$

对于司空见惯的int类型，思考一下下面两个问题

1、sizeof(int)=?

2、int类型允许包含padding bits吗？

深入了解unsigned int/int

对于int类型，思考一下下面两个问题

1、sizeof(int)=?
2、int类型允许包含padding bits吗？

C语言规定，int类型的sign bit+value bits长度必须大于等于16（新的INT_WIDTH宏）

相应的，INT_MAX必须大于等于32767，INT_MIN必须小于等于-32768

试试你自己电脑上编译器的INT_MAX等于多少？

不能假设sizeof(int)一定等于4，有些系统（例如Turbo C）中sizeof(int)就等于2

目前主流编译器的int类型都没有padding bits，但是依然不能假设所有int没有padding bits

了解int*N*_t/uint*N*_t

C语言标准规定编译器可以定义一类1）没有padding和2）确定宽度的整数类型
Exact-width integer types，形如int*N*_t，uint*N*_t

例如int16_t：意味着这个有符号整数类型没有padding bits且width恰好等于16

C语言标准不要求编译器必须提供int*N*_t类型

但如果编译器提供了宽度为8，16，32和64，且没有padding的整数类型
则应该通过typedef提供相应的int*N*_t类型

例如某平台int类型width是32且没有padding bit
则应该通过typedef int int32_t定义出int32_t类型

了解int_least*N*_t/uint_least*N*_t

C语言标准规定编译器需要定义一类宽度至少是某个N值的整数类型
Minimum-width integer types，形如int_least*N*_t，uint_least*N*_t

例如int_least16_t：意味着这个有符号整数类型的width大于等于16

以下类型是所有编译器都必须定义的

int_least8_t，int_least16_t，int_least32_t，int_least64_t
uint_least8_t，uint_least16_t，uint_least32_t，uint_least64_t

注意：如果编译器定义了int*N*_t，那么int_least*N*_t和int*N*_t一样

了解int_fast*N*_t/uint_fast*N*_t

C语言标准规定编译器需要定义一类宽度至少是某个N值且处理速度最快的整数类型
Fastest minimum-width integer types，形如int_fast*N*_t，uint_fast*N*_t

例如int_fast16_t：意味着这个有符号整数类型的width大于等于16，且处理速度最快

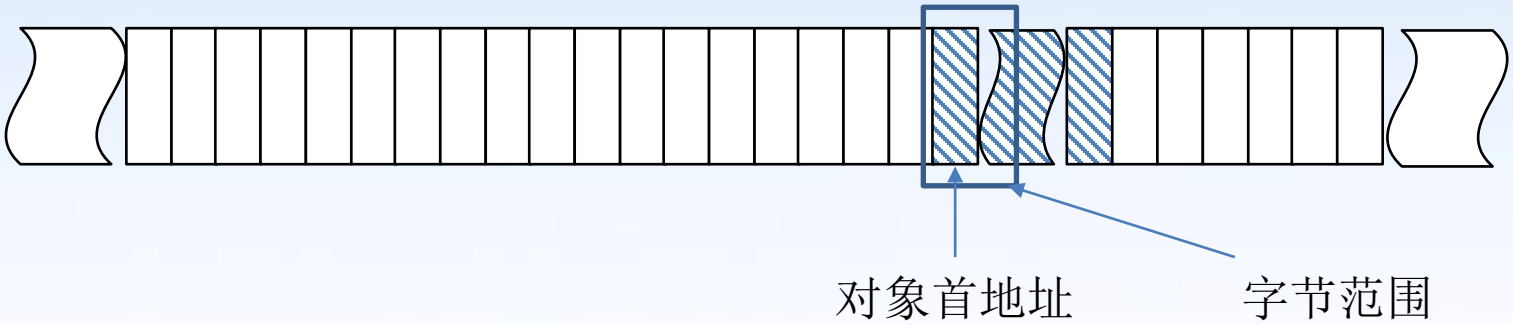
以下类型是所有编译器都必须定义的

int_fast8_t，int_fast16_t，int_fast32_t，int_fast64_t
uint_fast8_t，uint_fast16_t，uint_fast32_t，uint_fast64_t

注意：这个fast并不保证所有情况下处理速度都是最快，编译器可以简单选择满足符号要求和宽度要求的整数类型来进行typedef

什么是alignment（对齐）

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



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

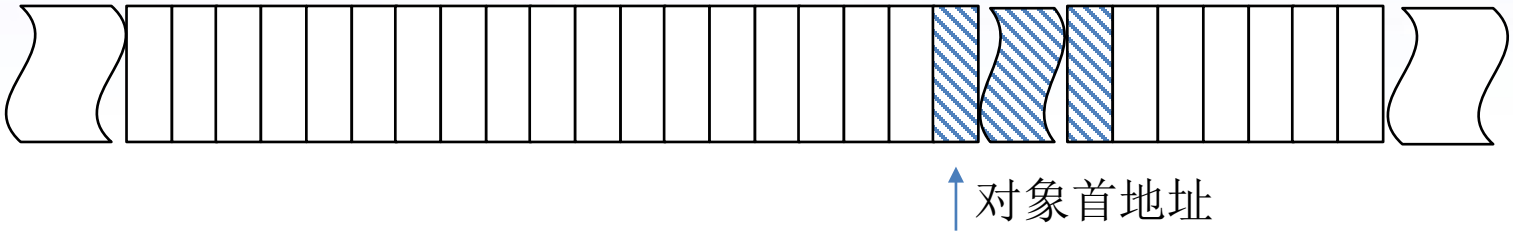
对齐的要求和编译器、硬件系统等紧密相关

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alingof(*T*)/_Alignof(*T*)可以获得对象类型*T*的Alignment

利用alignof(*O*)/_Alignof(*O*)可以获得对象*O*的Alignment（标准不支持）

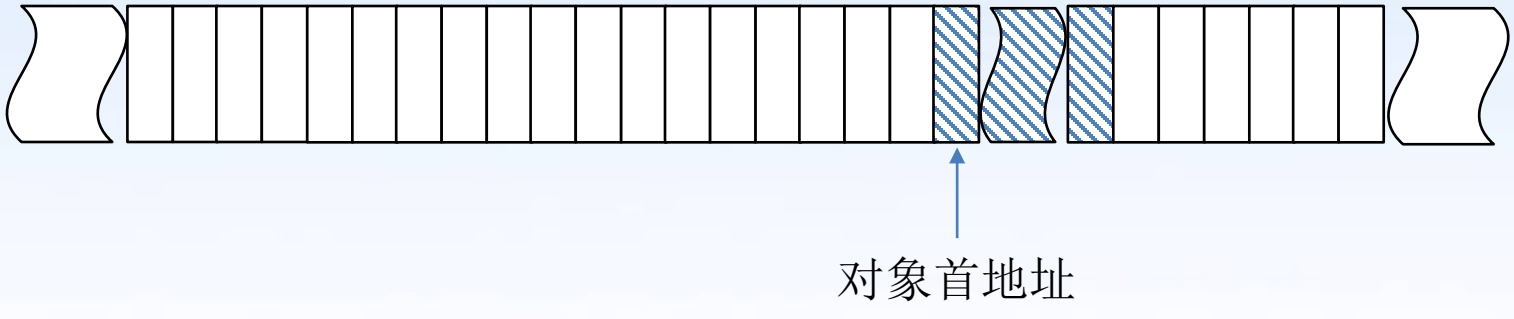
对象*O*的Alignment缺省等于对象类型*T*的Alignment



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*），标识符*O*对应那个对象的首地址有要求



char a	short a	int a	double a
alignof(char)返回1	alignof(short)返回2	alignof(int)返回4	alignof(double)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

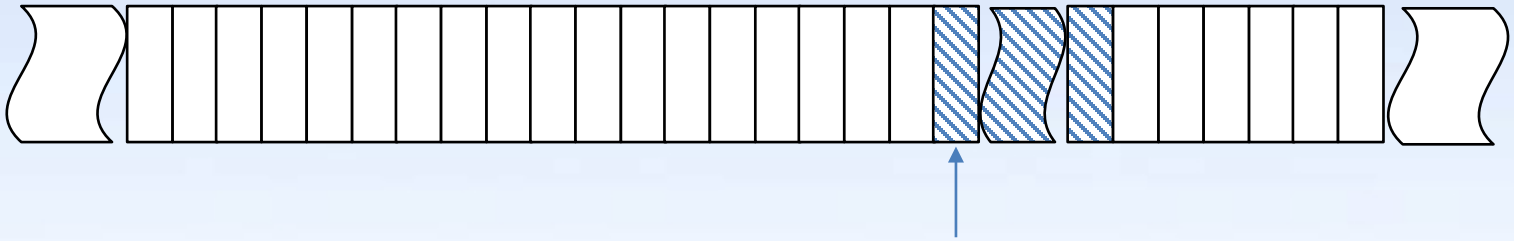
当声明一个*T*类型的对象*O*

可以为该对象设置更大（Stricter）的对齐要求：alignas(*N*)/_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



_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 += offset % _Alignof(E_2) ==0 ? 0: (_Alignof(E_2) - offset % _Alignof(E_2)

...

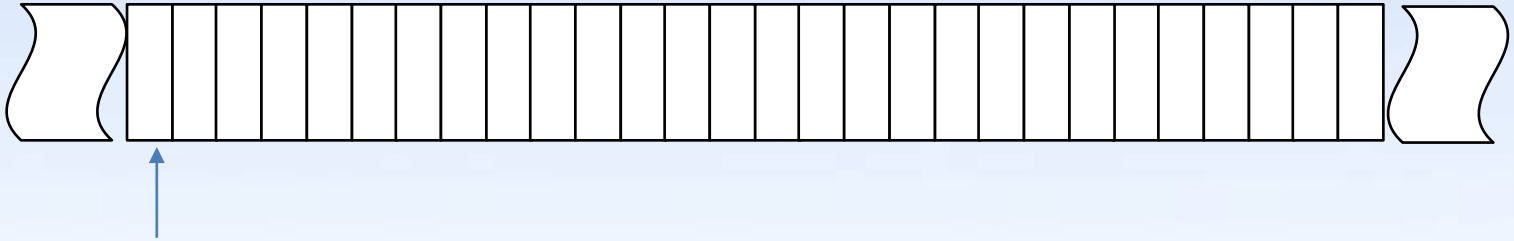
offset += sizeof(E_n)　　trailing padding

sizeof(T) = offset + offset % _Alignof(T) ==0 ? 0: (_Alignof(T) - 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
```

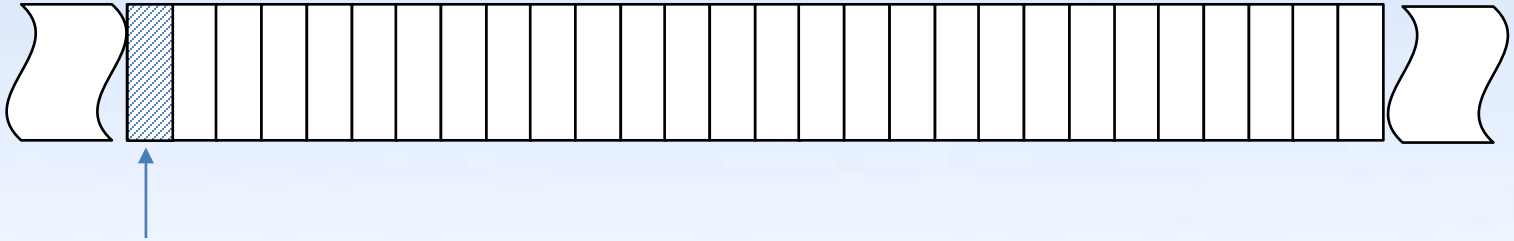


假设对象s的首地址是0x0061FD80（可以被64整除）

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
```



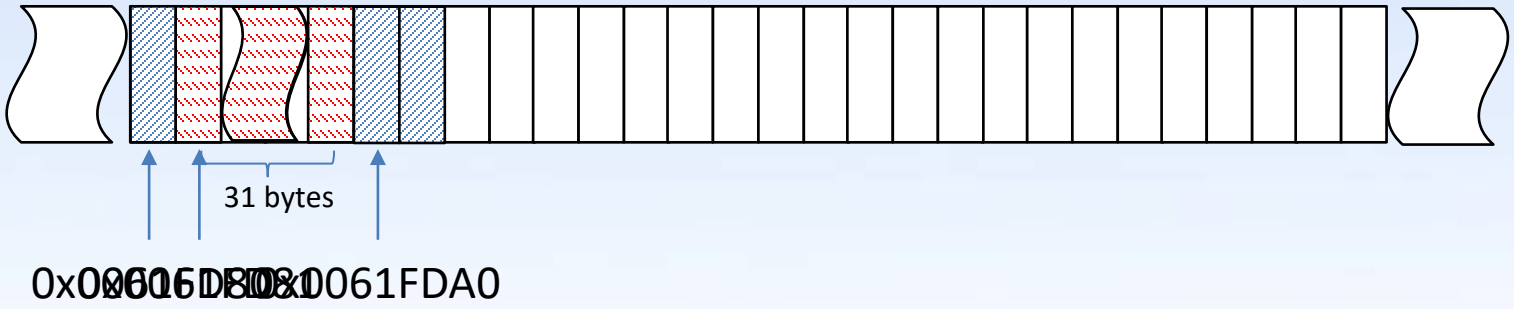
1、s.a的地址偏移量offset=0，即s.a的地址也是0x0061FD80

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

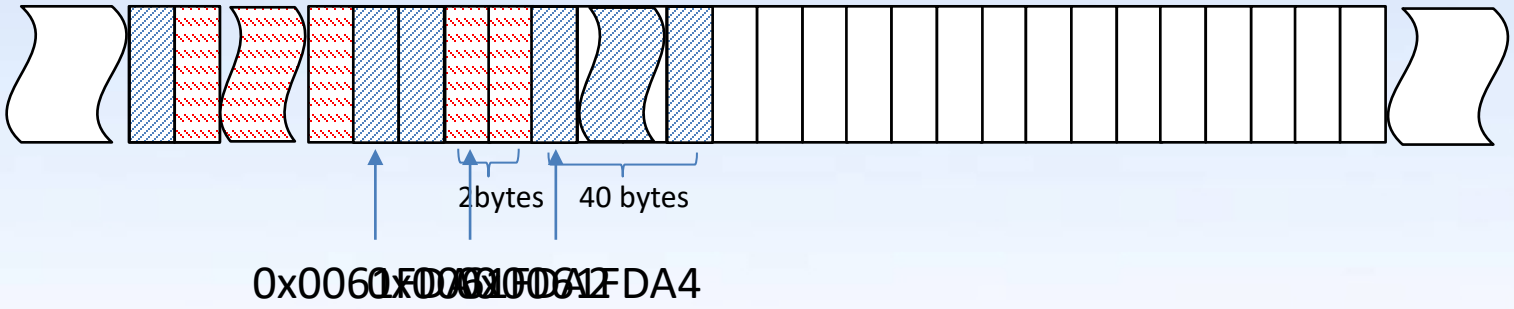
3、offset += offset % _Alignof(s.b) ==0 ? 0: (_Alignof(s.b) - offset % _Alignof(s.b))

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

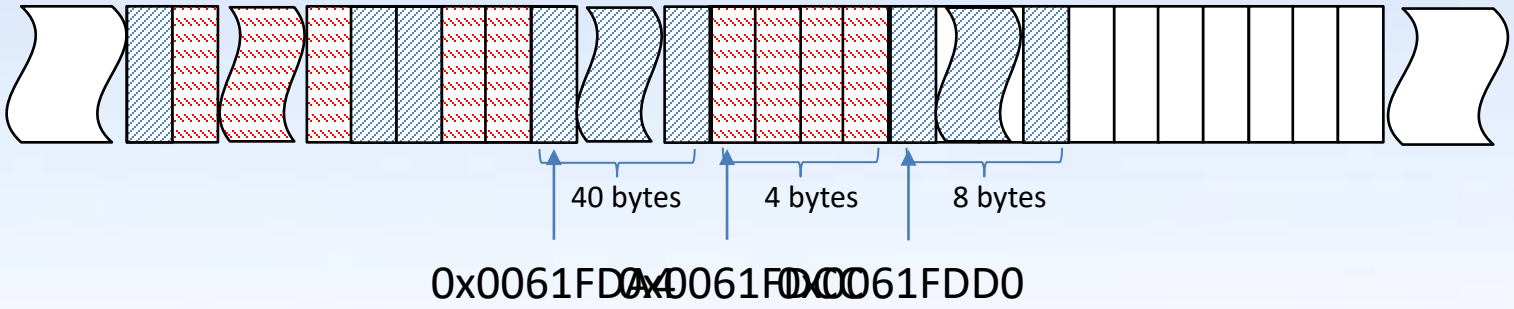
3、offset += offset % _Alignof(s.c) ==0 ? 0: (_Alignof(s.c) - offset % _Alignof(s.c))

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

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))

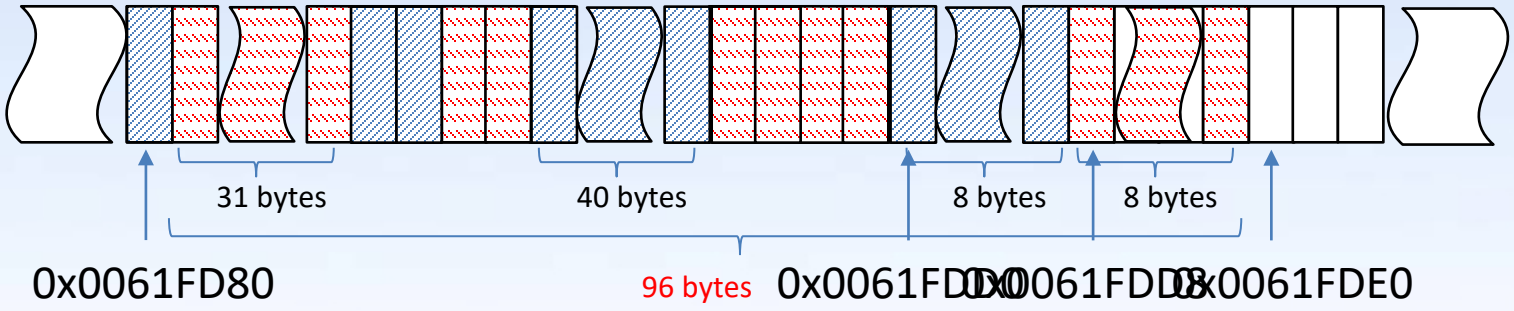
4、offset结果为80

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 += 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4，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 % fundamental alignment = 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