数组指针与函数指针

①数组相关的数据类型(坑太多，C++不建议使用)

```
int arr[5]={1,2,3,4,5}   arr数据类型为int [5]   而非int*
int *p = arr;   数组退化为指针
```

补充：[]的运算优先级是大于*的

1°数组指针 本质为指针，指向某特定大小的数组

```
int (*arrp)[5] =&arr;
```

数组名取地址的类型 = 数组名类型

int 数组指针 = int 指针

特点：arrp为指针，是arr数组的首地址

可用(*arrp)[i](解引用)访问数组元素

2°指针数组 本质为数组，内部储存指针(地址)

```
int *parr[5]={&a,&b,&c,&d,&e};
```

特点：parr为数组，每个元素为指针

可用*parr[i]访问指针所指对象

3°数组引用

```
int (&arrq)[5]= arr;   相当于取别名
```

注：有指针数组，但不存在引用数组

"Hello World"字符串字面量(左值) 有12个字符 '\0'结束符

```
const char *p = "hello world";
```

居然可以去掉

```
const char (*p)[12] = "hello world";
```

```
const char (&p)[12] = "hello world";
char str[12] = "hello world";
```
执行拷贝操作

② 数组名作函数参数传递时会发生退化

```
void fun(int a[10]);
void fun(int a[5]);
void fun(int *a);
```
⇒等价 传入类型为(int *)

```
void fun2(int (*a)[100]);   void fun2(int (*a)[5]);
```
与上不等价

传入数组指针且拥有100个元素

```
int (*b)[2] = {2, 1};
fun2(b);
```
✗ 报错

③ 函数相关的数据类型

```
bool fun(int a, int b);
```
⇒ bool(int, int); 函数数据类型

```
bool (*funp)(int a, int b);
```
⇒ bool (*)(int, int); 指针

函数指针赋值: `funp = &fun;` 可省略 函数名退化为指针

函数指针使用: `bool c = (*funp)(1, 2);`

函数指针作形参:

```
void fun2(int c, bool(*funp)(int, int));
```

函数指针作返回值：

该指针无法省略

bool (*fun3(int c))(int, int);

函数的引用：bool (&funref)(int, int) = fun; 仅能如此写

④类型别名 typedef / using

基本使用：

1° typedef int int_name;

原有 别名

2° using int_name = int; 更推荐使用且支持模板别名

别名 原有

int_name ooo = 911; 等同于 int ooo = 911;

定义函数类型：

typedef bool funx(int, int);

typedef bool (*funp)p(int, int);

using fun3x = bool(int, int);

using funp3p = bool (*)(int, int);

} fun3x = funp3p*

模板类：

template <typename T>

using MyVec = std::vector<T>;

MyVec<int> vec; 等价于 std::vector<int>