1.new操作符

new int;

2.操作符new

operator new

operator new[]

3.定位new

placement new

new和delete是用户进行动态内存申请和释放的操作符new在底层调用operator new全局函数来申请空间，delete在底层通过operator delete全局函数来释放空间。（operator new 和 operator delete是系统提供的全局函数）

new操作符

如果new操作符开辟一个自定义类型。如开辟一个对象

会先调用operator new(操作符new)------开辟空间

然后调用类的构造函数构造对象(初始化)

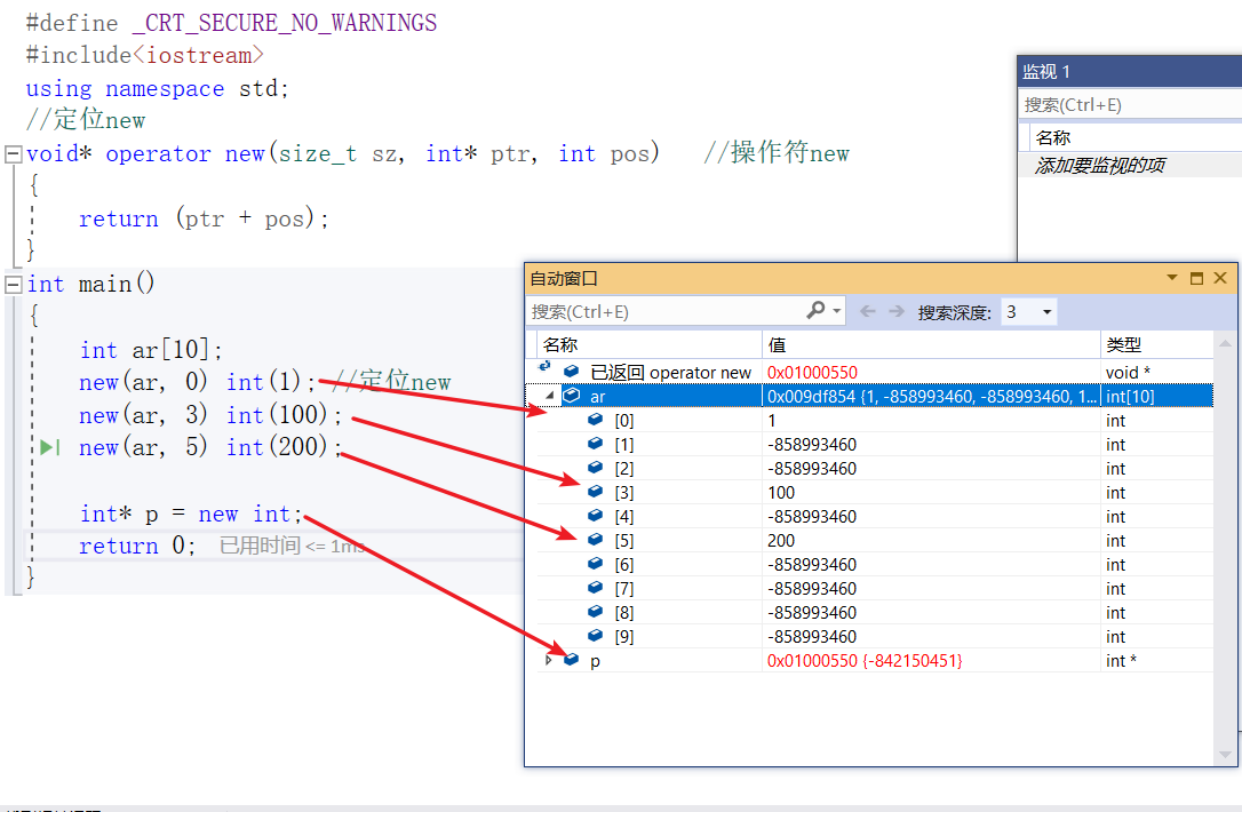
```
//new的底层模拟实现
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
using namespace std;
class Test
{
public:
	Test(int d = 0) : m_data(d)
	{
		cout << "Create Test Object : " << this << endl;
	}
	Test(const Test& t)
	{
		cout << "Copy Create Test Object : " << this << endl;
		m_data = t.m_data;
	}
	Test& operator=(const Test& t)
	{
		cout << "Assign:" << this << " = " << &t << endl;
		if (this != &t)
		{
			m_data = t.m_data;
		}
```

```
		return *this;
	}
	~Test()
	{
		cout << "Free Test Object : " << this << endl;
	}
public:
	void TestInit(int d = 0)
	{
		m_data = d;
	}
	void TestDestroy()
	{
		m_data = 0;
	}
public:
	int GetData()const
	{
		return m_data;
	}
public:
	void* operator new(size_t sz) //1
	{
		void* ptr = malloc(sz);
		return ptr;
	}
	void operator delete(void* ptr)
	{
		free(ptr);
	}
private:
	int m_data;
};

void* operator new(size_t sz) //1
{
	void* ptr = malloc(sz);
	return ptr;
}
void operator delete(void* ptr)
{
	free(ptr);
}

void* operator new[](size_t sz) //1
{
	void* ptr = malloc(sz);
```

```
	return ptr;
}
void operator delete[](void* ptr)
{
	free(ptr);
}

int main()
{
	//operator new
	Test* pt = new Test(10);//1 2   //操作符new
	delete pt;              //1 2
	//Test *pt = (Test*)operator new(sizeof(Test)); //申请空间malloc

	Test* pt1 = new Test[10];//40
	return 0;
}
```

定位new

```
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
using namespace std;
//定位new
void* operator new(size_t sz, int* ptr, int pos)   //操作符new
{
	return (ptr + pos);
}
int main()
{
	int ar[10];
	new(ar, 0) int(1); //定位new
	new(ar, 3) int(100);
	new(ar, 5) int(200);

	int* p = new int;
	return 0;
}
```



实现类的对象只能在堆上开辟， 禁止拷贝构造和对象赋值

```
class Test
{
	friend Test* CreateTestObject(int data);
public:
	static Test* CreateObject(int data)
	{
		return new Test(data);
	}
private:
	Test(int d = 0)
	{
		m_data = d;
	}
protected: //继承中才体现价值
	Test(const Test &);// = delete;  //拷贝构造方法受保护
	Test& operator=(const Test &);  //赋值运算符重载方法受保护
private:
```

```
    int m_data;
};

Test* CreateTestObject(int data)
{
    return new Test(data);
}

int main()
{
    //Test t(1);  //禁止在栈上开辟对象
    //Test t1(t);  //禁止拷贝构造
    //Test t2;   //禁止在栈上开辟对象
    //t2 = t;  //禁止赋值语句
    Test *pt1 = Test::CreateObject(1);
    Test* pt2 = CreateTestObject(1);
    //*pt1 = *pt2;  //禁止赋值语句  Test::operator =”: 无法访问 protected 成员
    return 0;
```

只允许对象在堆上开辟一般将构造方法设置成静态方法（用类名加作用域访问符调用）

```
public:
    static Test* CreateObject(int data)
    {
        return new Test(data);
    }
```

或者将构造方法设置成友元函数（相当于一个可以访问私有数据成员的全局函数， 优点是即不用对象去驱动也不需要作用域访问限定）

```
friend Test* CreateTestObject(int data);

Test* CreateTestObject(int data)
{
    return new Test(data);
}
```

作用域访问符protected:的价值在继承中才体现

对于成员方法protected和privated是一样的

成员方法限制成protected或者privated时，由于方法不可能被调用，所以只需要声明就可以了，不需要实现方法：

在C++98标准中：

```
protected: //继承中才体现价值
        Test(const Test &);// = delete;  //拷贝构造方法受保护
        Test& operator=(const Test &);  //赋值运算符重载方法受保护
```

在C++11标准中可以写成：

```
protected: //继承中才体现价值
        Test(const Test &) = delete;  //拷贝构造方法受保护
    Test& operator=(const Test &) = delete;  //赋值运算符重载方法受保护
```

set_new_handler

```
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
using namespace std;

void Out_Of_Memory()
{
        cout << "new:Out Of Memory." << endl;
        //执行内存回收
        exit(1);
}

int main()
{
        set_new_handler(Out_Of_Memory);
        int* p = new int[536870911];//  new
        //int *p = new int[5];
        delete[]p;
        return 0;
}
//set_new_handler 的功能：
//0 尽可能 满足 需求
//1 申请成功 直接返回
//2 申请不成功，设置了set_new_handler方法，有可能成功返回
//3 内存确实不足，抛出异常
```

函数模板-->泛型

```
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
using namespace std;
template<class Type1, class Type2>
Type1 Max(Type1 a, Type2 b)
{
	return a > b ? a : b;
}

int main()
{
	cout << Max(1, 1.2) << endl;  //char
	return 0;
}
```

用不同类型的参数使用函数模板时，称为函数模板的实例化。模板参数实例化分为：隐式实例化和显式实例化

```
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
using namespace std;
template<typename Type1, typename Type2>
Type1 Max(Type1 a, Type2 b)
{
	return a > b ? a : b;
}
void main()
{
	cout<<Max('A', 'B')<<endl;  //char
	cout<<Max(10,20)<<endl;     //int
	cout<<Max(12.34,23.45)<<endl; //double
	cout<<Max((double)1,2.3)<<endl;
	cout<<Max(1,(int)2.3)<<endl;
	cout<<Max<int>(1, 2)<<endl;  //显式实例化
}
```

模板函数的特化

效率不高， 所以需要特化

```
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
using namespace std;
template<typename Type1, typename Type2>
```

```
Type1 Max(Type1 a, Type2 b)
{
	return a > b ? a : b;
}
double Max(int a, double b)   //模板函数的特化
{
	return a > b ? a : b;
}

void main()
{
	cout<<Max(1, 1.2)<<endl;
	cout<<Max<double>(1, 1.2) << endl;
	cout << Max(1, 1.2) << endl;
}
```

类模板

```
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
using namespace std;
template<typename Type>
class Stack
{
public:
	Stack(size_t sz = STACK_DEFAULT_SIZE)
	{
		capacity = sz > STACK_DEFAULT_SIZE ? sz : STACK_DEFAULT_SIZE;
		base = new Type[capacity];
		top = 0;
	}
	~Stack()
	{
		delete[]base;
		capacity = top = 0;
	}
public:
	void push(const Type& x)
	{base[top] = x;}
	void pop()
	{top--;}
	Type Top()const
```

```
        {return base[top];}
        Type& Top()
        {return& base[top];}
public:
        bool empty()const
        {return top == 0;}
        bool full()const
        {return top == capacity;}
private:
        enum { STACK_DEFAULT_SIZE = 20 };
        Type* base;
        size_t capacity;
        size_t top;
};

int main()
{
        Stack<int> Int_st;
        return 0;
}
```