c++本身的并发库怎么包装不同系统的接口，

模板元编程看理解

stl看看也最好了

南山mtk

零散小知识

- C++11中，原子类型被视为一种库特性，而非语言特性，因其可以通过库的方式来实现
- 复制及再抛出异常（page6）
- 并行动态初始化和析构（page6）
- c++11中允许对非静态成员变量使用sizeof操作
- 模板的默认参数应该从右至左指定（page66）
- 模板的显式实例化定义、外部模板声明和使用与全局变量的定义、外部申明和使用类似（page68）
- 局部和匿名类型可以作为模板实参（page72）
- 创建只在栈上分配内存的类：用`=delete`修饰`operator new/delete`（page251）

  创建只在堆上分配内存的类：私有化析构函数
- 常量成员函数中，可以修改类型为引用的成员变量，因为修改的不是引用本身，而是引用的值。

变长参数的宏定义以及__VA_ARGS__

page40

```
void log_print(const char *, const char *, int, const char *, ...);
#define WRITE_LOG(LOGTEXT, ...) do{ \
    log_print(__FILE__, __FUNCTION__, __LINE__, LOGTEXT, ##__VA_ARGS__);\
}while(0);

void log_print(const char *, const char *, int, const char *, ...){
    // 函数申明
}
// 应用
```

#error，static_assert和assert

用于排除逻辑上不可能存在的状态

- **#error**
  - 语法格式：`#error error-message`
  - 预编译期断言，编译程序时，只要遇到`#error`就会生成一个编译错误提示消息，并停止编译。
- **assert**
  - 语法格式：`assert( expression );`
  - 运行期断言，计算表达式，如果结果为`false`，则打印诊断消息并中止程序。

- 由于NDEBUG宏的存在，`assert()`在release模式下被禁用，因此它仅在debug模式下显示错误消息，如想在release模式下输出，**应使用`#undef NDEBUG`，同时该预编译指令必须放在`#include<cassert>`前面。**

```
// 禁用assert()的简要实现
#ifdef NDEBUG
#define assert(expr)            (static_cast<void> (0))
#else
......
#endif
// 显然，在定义了NDEBUG宏后，assert()被展开为一条无意义的语句，然后被编译器优化掉
```

所以对于对应用程序正确运行至关重要的检查（如检查`dynamic_cast()`的返回值）时，为了确保它们在debug模式下也会执行，应使用 if 语句

- **static_assert**
  - 语法格式：`static_assert(constant-expression, string-literal);`（注意只能是**常量表达式**和**常量字符串**）
  - 编译期断言， 如果指定的常数表达式为`false`，则编译器显示指定的消息，并且编译失败

```
// static_assert()简要实现
#define assert_static(e) \
    do { \
        enum { assert_static__ = 1 / (e) }; \
    } while(0)
```

异常

异常用于处理逻辑上可能发生的错误

异常处理工作原理

每当您使用 throw 引发异常时，编译器都将查找能够处理该异常的 catch(Type)。异常处理逻辑首先检查引发异常的代码是否包含在 try 块中，如果是，则查找可处理这种异常的 catch(Type)。如果 throw 语句不在 try 块内，或者没有与引发的异常兼容的 catch( )，异常处理逻辑将继续在调用函数中寻找。 因此，异常处理逻辑沿调用栈向上逐个地在调用函数中寻找，直到找到可处理异常的 catch(Type)。在 退栈过程的每一步中，都将销毁当前函数的局部变量，因此这些局部变量的销毁顺序与创建顺序相反。（chapter28.4）

std::exception 类

捕获 std::bad_alloc 时，实际上是捕获 new 引发的 std::bad_alloc 对象。std::bad_alloc 继承了 C++标准类 std::exception，而 std::exception 是在头文件中声明的。
下述重要异常类都是从 std::exception 派生而来的。

- bad_alloc：使用 new 请求内存失败时引发。
- bad_cast：试图使用 dynamic_cast 转换错误类型（没有继承关系的类型）时引发。
- ios_base::failure：由 iostream 库中的函数和方法引发。

std::exception 类是异常基类，它定义了虚方法 what( )；这个方法很有用且非常重要，详细地描述 了导致异常的原因。在程序清单 28.2 中，第 18 行的 `exp.what( )` 提供了信息 bad array new length，让用户知道什么地方出了问题。由于 std::exception 是众多异常类型的基类，因此可使用 `catch(const exception&)` 捕获所有将 `std::exception` 作为基类的异常：

```
void SomeFunc()
{
   try
   {
      // code made exception safe
   }
   catch (const std::exception& exp) // catch bad_alloc, bad_cast, etc    {
      cout << "Exception encountered: " << exp.what() << endl;    }
}
```

从 std::exception 派生出自定义异常类

可以引发所需的任何异常。然而，让自定义异常继承 std::exception 的好处在于，现有的异常处理程序 catch(const std::exception&)不但能捕获 bad_alloc、bad_cast 等异常，还能捕获自定义异常，因为它 们的基类都是 exception。

```
#include <exception>
#include <iostream>
#include <string>
using namespace std;

class CustomException : public std::exception
{
    string reason;
public:
    // constructor, needs reason
    CustomException(const char* why) :reason(why) {}

    // 重写父类what()虚函数
    // 注意noexcept关键字，可避免在what()引发异常
    virtual const char* what() const noexcept
    {
        return reason.c_str();
    }
};

double Divide(double dividend, double divisor)
{
    if (divisor == 0)
        throw CustomException("CustomException: Dividing by 0 is a crime");

    return (dividend / divisor);
}

int main()
{
    cout << "Enter dividend: ";
    double dividend = 0;
    cin >> dividend;
```

```
    cout << "Enter divisor: ";
    double divisor = 0;
    cin >> divisor;
    try
    {
        cout << "Result is: " << Divide(dividend, divisor);
    }
    catch (exception& exp)// catch CustomException, bad_alloc, etc
    {
        cout << exp.what() << endl; 474
            cout << "Sorry, can't continue!" << endl;
    }

    return 0;
}
```

noexcept修饰符

noexcept 表示其修饰的函数不会抛出异常。在 C++11 中如果 noexcept 修饰的函数抛出了异常，编译器会直接调用 std::terminate () 函数来终止程序的运行。这比基于异常机制的 throw 在效率上会高一些。

语法：

```
void excpt_func() noexcept;
void excpt_func() noexcept(constant_expression);// 只能是常量表达式
```

常量表达式的结果会被转换成一个 bool 类型的值。该值为 true，表示函数不会抛出异常；反之，则有可能抛出异常。这里，不带常量表达式的 noexcept 相当于声明了 noexcept (true)，即不会抛出异常。

```
#include <iostream>
#include <exception>
using namespace std;

void Throw() { throw 1; }                       // 该函数唯一作用是抛出异常
void NoBlockThrow() { Throw(); }        // 该函数会让throw()中抛出的异常继续抛出
void BlockThrow() noexcept {Throw(); }  // 该函数不允许让throw()抛出的异常继续抛出，而是调用terminate()中断程序

int main() {
    try {
        Throw();
    }
    catch (...) {
        cout << "Found throw." << endl; // Found throw.
    }
    try {
        NoBlockThrow();
    }
    catch (...) {
        cout << "Throw is not blocked." << endl; // Throw is not blocked.
    }
    try {
        BlockThrow();                   // terminate() called after throwing an instance of 'int'
```

```
    }
    catch (...) {
        cout << "Found throw 1." << endl;
    }
    // 编译选项: g++ -std=c++11 2-6-1.cpp
}
```

noexcept操作符

noexcept作为操作符时，通常用于模板

语法：

```
template <class T>
    void fun() noexcept(noexcept(T())) {}
```

这里，fun 函数是否是一个 noexcept 的函数，将由 T() 表达式是否会抛出异常所决定。这里的第二个 noexcept 就是一个 noexcept 操作符。当其参数是一个有可能抛出异常的表达式的时候，其返回值为 false，反之为 true。

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

bool judge(int i) noexcept
{
    bool bj = static_cast<bool>(i);
    return true;
}

template<typename Type>
void mswap(Type& x, Type& y) noexcept(noexcept(judge(x)))    //此处先判断judge(x))是否被noexcept修饰，如果是，则mswap也被noexcept修饰
{
    throw 1;
    cout << "throw" << endl;
}

int main() {
    int x = 0;
    x++;
    int y = 4;

    try {
        mswap(x, y);
    }
    catch (...) {
        cout << "Found throw." << endl; // Found throw.
    }
    // 编译选项: g++ -std=c++11 2-6-1.cpp
}
```

应用场景：

- 移动构造函数（move constructor）

- 析构函数（destructor）。这里提一句，在新版本的编译器中，析构函数是默认加上关键字noexcept的。下面代码可以检测编译器是否给析构函数加上关键字noexcept。

```
    struct X
    {
        ~X() { };
    };

    int main()
    {
        X x;
        // explicitly marked as noexcept(true)
        static_assert(noexcept(x.~X()), "Ouch!");
    }
```

C++11初始化列表

C++11列表初始化

提示

C++11 引入了固定宽度的整型，让您能够以位为单位指定整数的宽度。这些类型为 int8_t 和 unit8_t，分别用于存储 8 位的有符号和无符号整数。您还可能使用 16 位、32 位和 64 位的整型，它们为 int16_t、uint16_t、int32_t、uint32_t、int64_t 和 uint64_t。要使用这些类型，必须包含头文件<cstdint>。

3.3.1 使用列表初始化避免缩窄转换错误

使用取值范围较大的变量来初始化取值范围较小的变量时，将面临出现缩窄转换错误的风险，因为编译器必须将大得多的值存储到容量没那么大的变量中，下面是一个这样的示例：

```
int largeNum = 5000000;
short smallNum = largeNum; // compiles OK, yet narrowing error
```

缩窄转换并非只能在整型之间进行，但如果使用 double 值来初始化 float 变量、使用 int 值来初始化 float 或 double 变量，或者使用 float 值来初始化 int 变量，可能导致缩窄转换错误。有些编译器可能发出警告，但这种警告并不会导致程序无法通过编译。在这种情况下，程序可能在运行阶段出现 bug，但这种 bug 并非每次运行时都会出现。

为避免这种问题，C++11 引入了列表初始化来禁止缩窄。要使用这种功能，可将用于初始化的变量或值放在大括号（{}）内。列表初始化的语法如下：

```
int largeNum = 5000000;
short anotherNum{ largeNum }; // error! Amend types
int anotherNum{ largeNum }; // OK!
float someFloat{ largeNum }; // error! An int may be narrowed
float someFloat{ 5000000 }; // OK! 5000000 can be accomodated
```

这种功能的作用虽然不明显，但可避免在执行阶段对数据进行缩窄转换导致的 bug：这种 bug 是不合理的初始化导致的，难以发现。

[列表初始化及decltype_列表初始化类中的结构体-CSDN博客]

[【C++从练气到飞升】24--C++11：列表初始化 | 声明 | STL的升级-CSDN博客]

C++98中的{}

仅允许使用花括号 { } 对**数组**或者**结构体元素**进行统一的列表初始化

```
struct Point
{
```

```
    int _x;
    int _y;
};

int main()
{
    int arrya1[] = { 1, 2, 3, 4 };//列表初始化，初始化数组
    int array2[5] = { 0 };//列表初始化，初始化数组
    Point p = { 1, 2 };//列表初始化，初始化结构体元素
    Point array3[] = { {1, 2}, {3, 4}, {5, 6} };//列表初始化，初始化结构体数组
    return 0;
}
```

C++11中的{}

用于内置类型初始化

使用初始化列表时，可以添加等号（=），也可以不添加，其中new表达式中不可带等号

```
int main()
{
    //C++11 中列表初始化应用在内置类型上
    int x1 = 10;
    int x2 = { 20 };
    int x3{ 30 };//不带等号

    //C++11 中列表初始化也可以适用于 new 表达式中
    int* p1 = new int[5]{100, 112, 113};
    return 0;
}
```

用于用户自定义的类型初始化

创建对象时也可以使用列表初始化的方式来调用构造函数初始化

> 如果在 Date 类的构造函数加上 explicit，那么 Date d3 = { 2022, 1, 3 }; 就会出现编译报错，因为这条语句本质上是因为**多参数的构造函数支持隐式类型转换**。Date d2{ 2022, 1, 2 }; 没事，仍然可以正常运行。

```
class Date
{
public:
    Date(int year, int month, int day) : _year(year), _month(month),
_day(day)
    {
        cout << "Date(int year, int month, int day)" << endl;
    }
private:
    int _year;
    int _month;
    int _day;
};

int main()
{
```

```
    Date d1(2022, 1, 1); // old style

    // C++11支持的列表初始化，这里会调用有参构造函数初始化
    Date d2{ 2022, 1, 2 };
    Date d3 = { 2022, 1, 3 };//本质上是多参数的构造函数支持隐式类型转换，注意此处由于是在初始化阶段，所以调用的是有参构造函数，不是operate=()
    const Date& d1 = { 2003, 10, 18 };//(正确)
    return 0;
}
```

typeid(变量名).name()

C++11 标准中，`typeid(变量名).name()`函数返回的是一个 const char*，指向一个以 null 终止的字符串，表示类型的类型名称。

initializer_list

一般作为容器中构造函数的参数

```
int main()
{
    auto il = { 1, 2, 3 };
    cout << typeid(il).name() << endl;
    return 0;
}
// 输出结果：
// class std::initializer_list<int>
```

用于容器初始化以及赋值

新增 initializer_list 作为参数的构造函数和operator=，创建对象时使用 initializer_list 来调用构造函数初始化

> 需要注意，下面代码中创建 Date 类型的对象 d1，本质上是隐式类型的转换。但是**容器存储的数据个数可多可少，并不确定，所以提供了形参为 initializer_list 的构造函数**，这样就方便我们将任意数量的元素存到容器中。而 Date 作为一个日期类对象，它的三个成员变量是固定的，所以不需要提供形参为 initializer_list 的构造函数。因此**自定义类的列表初始化本质上是隐式类型转换，而容器类的列表初始化本质上是使用 initializer_list 作为形参的构造函数**。

```
int main()
{
    //调用vector中形参为 initializer_list 的构造函数
    vector<int> v = { 1,2,3,4 };
    // 这里先隐式转换，调用有参构造创建一个pair对象，再调用形参为 initializer_list 的构造函数
    map<string, string> dict = { {"sort", "排序"}, {"insert", "插入"} };
    //调用vector中形参为 initializer_list 的赋值运算符重载函数
    v = { 10, 20, 30 };

    Date d1 = { 2003, 4, 5 };//这里是直接调用3个参数的构造函数 --- 隐式类型转换

    return 0;
}
```

成员变量的就地初始化

C++11 快速初始化成员变量 - kaizenly - 博客园 (cnblogs.com)

> - C++11中，允许就地地使用 `=` 或者 `{}` 对**非静态成员变量**进行列表初始化。
> - 对于**非常量的静态成员变量**而言，需类内申明，类外初始化。（保证编译时，类静态成员的定义最后只存在于一个目标文件中）
> - 对于**常量的静态成员变量**而言，需就地初始化。
> - 初始化列表的效果总是优先于就地初始化。

```
struct C
{
    C(int i, int j) :c(i), d(j){};

    int c;
    int d;
};

struct init
{
    // 对成员变量就地使用列表初始化
    int a = 1;
    string b{ "hello" };
    C c{ 1, 3 };
};
```

define/const？？？

？？？#define 宏定义看这一篇文章就够了-CSDN博客

define是在编译的**预处理阶段**起作用；而const是在**编译、运行**的时候起作用

宏不检查类型，一般来说加上一个大括号；const会检查数据类型

const/static？？？

static

C/C++之static函数与普通函数

- static修饰普通全局变量/函数
  隐藏。所有不加static的全局变量和函数具有全局可见性，可以在其他文件中使用，**加了之后只能在该文件所在的编译模块（即申明该函数/变量的文件）中使用**
- static修饰普通局部变量

  改变了局部变量的存储位置，从原来的栈中存放改为静态存储区。但是作用域仍为局部作用域，局部静态变量在离开作用域之后，并没有被销毁，而是仍然驻留在内存当中，直到程序结束，只不过我们不能再对他进行访问。
- static修饰类的成员变量/成员函数
  static成员变量：只与类关联，不与类的对象关联。定义时要分配空间，可以被非static成员函数任意访问。
  static成员函数：不具有this指针，无法访问类对象的普通成员变量和普通成员函数； 不能被声明为const、虚函数和volatile；可以被非static成员函数任意访问

const

C语言中const关键字的用法-CSDN博客？？？

- 不考虑类的情况
  const常量在定义时必须初始化，之后无法更改，成为以一个**编译期常量**；const形参可以接收const和非const类型的实参
- 考虑类的情况
  const成员变量：只能通过构造函数初始化列表进行初始化，并且必须有构造函数；

  const成员函数：const对象只能调用const成员函数；且只有mutable修饰的数据的值可以改变

继承构造函数

page75

> - 为了避免二义性，尽量不要在构造函数中使用默认参数，且继承构造函数不会继承默认参数
> - `using Base::Base`是**隐式声明继承的**。即假设一个继承构造函数不被相关的代码使用，编译器不会为之产生真正的函数代码，这样比透传基类各种构造函数更加节省目标代码空间。但此时派生类无法对类自身定义的新的类成员进行初始化，通过**就地初始化**解决。
> - 基类的构造函数被声明为**私有**或者派生类是从基类**虚继承**的，那么就不能在派生类中声明继承构造函数。
> - 一旦使用了继承构造函数，编译器就不会为派生类生成默认构造函数。
> - 派生类继承了多个基类的时候。多个基类中的部分构造函数可能导致派生类中的继承构造函数的函数名、参数相同，解决办法就是**显式的继承类的冲突构造函数**。

```
class Base {
public:
    Base(int x) {
        // 基类构造函数
    }
};
class Derived : public Base {
public:
    using Base::Base; // 继承基类的所有构造函数，也就是继承构造函数
};
int main() {
    Derived d(10); // 直接使用继承来的基类构造函数
    return 0;
}
```

```
struct A
{
    A(int) {}
};
struct B
{
    B(int) {}
};
struct C :A, B
{
    using A::A;
    using B::B;
```

```
    C(int i) :A(i), B(i) {}      // 显式的继承类的冲突构造函数解决冲突
};

int main() {
    C c(1);
}
```

`Derived`类通过`using Base::Base;`语句继承了`Base`类的所有构造函数。因此，在创建`Derived`类的实例时，我们可以直接使用`Base`类的构造函数来初始化`Derived`对象。

委派构造函数

C++11之委派构造函数

page80

使用方法

> - 用于减少构造函数中的重复代码
> - 概念：**将构造的任务分派给一个目标构造函数来完成。**（eg1）
> - **委派构造函数**：初始化列表中调用"基准版本"的构造函数就是委派构造函数。
>   **目标构造函数**：被调用"基准版本"构造函数就是目标构造函数。
> - 委派构造函数不能使用 初始化列表 初始化成员变量（eg1）
> - 目标构造函数的执行总是**先于**委派构造函数。因此需要避免目标构造函数和委托构造函数体中初始化同样的成员。（eg1）

```
// eg1
class Info
{
public:
    Info() :Info(1, 'a') {}              // 委派构造函数，使用目标构造函数完成委派构造函数的初始化

    Info(int i) : Info(i, 'a') { type = i; }         // 可能出错，委派构造函数中应避免对同样的成员变量进行多次初始化
    Info(char ch) : Info(1, ch) {}                   // 正确，委派构造函数

    Info(int i, char ch, int j)                      // 错误，委托构造函数中不能使用初始化列表
        : Info(i, ch)
        , typej(j)
    {}

private:
    Info(int i, char ch) :type(i), name(ch) { type += 1 }   // 目标构造函数
    int type{ 1 };
    char name{ 'a' };
    int typej{ 0 };
};
int main()
{
    Info info(5);
```

```
    cout << info.type << endl;  // 此时输出的值为5：首先会调用目标构造函数，此时type=6；然后调用委托构造函数体，此时type=5，显然这并不符合我们的预期
    return 0;
}
```

链式委托

`Info()` 委派 `Info(int)` 进行构造工作，然后 `Info(int)` 委派 `Info(int, char)`。这就是**链状委派。**

```
// 链状委派构造函数
class Info
{
public:
    Info() :Info(1) {}
    Info(int i) : Info(i, 'a') {}   // Info(int i) 既是目标构造函数，也是委派构造函数

private:
    Info(int i, char ch) :type(i), name(ch) {}

public:
    int type{ 1 };
    char name{ 'a' };
};
```

委托环

链状委派没有任何问题，但是形成环形链状委派也叫**委派环**（delegation cycle）就会导致**构造死循环**

```
// 委派环
class Info
{
public:
    // 编译错误
    Info(int i) : Info('c') {}
    Info(char ch) : Info(2) {}

private:
    int type{ 1 };
    char name{ 'a' };
};

```

使用场景

构造模板函数

使用构造模板函数作为目标函数，分别使用vector，deque来初始化list容器

```
class Test
{
private:
    template<class T>
    Test(T first, T last) :m_list(first, last) {}

public:
    Test(vector<int>& v) :Test(v.begin(), v.end()) {}
    Test(deque<int>& d) : Test(d.begin(), d.end()) {}

private:
    list<int> m_list;
};
```

异常处理

[构造函数与函数try块 构造函数里用trycatch]

```
class Test
{
public:
    Test(double d)
        try : Test(1, d)            // 函数try块不仅可以用于初始化列表，还可以用于委派构造函数
    {
        cout << "Run the body" << endl;
    }
    catch (...)
    {
        cout << "caught exception." << endl;
    }

private:
    Test(int i, double d)
    {
        cout << "going to throw" << endl;
        throw 0;
    }
    int type;
    double data;
};

int main()
{
    Test info(5);
    return 0;
}

```

右值引用，移动语义，完美转发

左值引用和右值引用？？？

左值与右值这两个概念是从 C 中传承而来的，左值指既能够出现在等号左边，也能出现在等号右边的变量；右值则是只能出现在等号右边的变量。

使用场景：**当函数返回了一个函数内部定义的临时变量时，可以用右值引用接收**

- **左值引用只能绑定左值**

```
int a1 = 10;
int& a2 = 10;// 编译错误：非常量左值引用
int& a2 = a1;//编译正确，左值引用可以接受左值
```

- **常量左值引用既可以接收左值，也可以接收右值**

```
    int& a = 7;//编译报错：非常量左值引用
    const int& a = 7;//编译正确，常量左值引用可以接收右值
    const int& b = a;//编译正确，常量左值引用可以接收左值
    //常量左值引用可以绑定右值是一条非常棒的特性，但是也存在一个缺点----常量性，一旦使用常量左值引用，表示我们无法在函数内修改该对象的内容（强制类型转换除外），所以需要另一个特性帮助我们完成工作，即右值引用。
```

- **右值引用只能绑定右值，或者通过std::move()绑定左值**，能够有效**延长右值的生命周期，减少对象复制，提升程序性能**；在语法方面，右值引用在类型后加&&

  右值引用既可以作为左值，也可以作为右值，当右值引用有名字的时候为左值：`int&& a = 3;`此时a为左值；右值引用没名字的时候为右值：`move()`的返回值为右值引用，此时右值引用为右值，因为该返回值只能被右值引用接收；

```
    int i = 0;
    int& j = i;      //左值引用
    int&& k = 11;    //右值引用
    //上述代码中，k是一个右值引用，如果用k引用i，会引起编译错误，右值引用的特点是可以延长右值的生命周期，对于11，理解可能不是很深，请看下面代码
```

```
class X{
public:
    X(){}
    X(const X& x){}
    ~X(){}
    void show(){cout<<"show"<<endl;}
};
X make_x(){
    X x1;
    return x1;
}
int main(){
#ifdef 0
    //对于该段代码，一共发生了一次构造，两次拷贝构造
    //第一次是调用构造从函数中创建x1临时变量，第二次是拷贝构造出一个临时值作为make_x返回值，第三次是拷贝构造出x2
    X x2(make_x());
```

```
    x2.show();                    //此处是通过make_x()的返回值又构建了一个x2，调用了x2的拷贝构造函数
#endif
    //使用右值引用，只需一次构造，一次拷贝构造即可实现同样功能
    //第一次是构造x1，第二次是构造x2
    //这里就很好体现出“延长生命周期，减少对象复制”的特点
    X&& x2 = make_x();
    x2.show();                    // 此时x2就是make_x()的返回值，x2延长了返回值的生命周期。
}
```

> !!!!注意，上面的代码在关闭了RVO后`X x2(make_x());`依然只有两次构造，再好好看看rvo

std::move

[一文读懂C++右值引用和std::move - 知乎 (zhihu.com)]

> - std::move就是将左值转为右值。这样就可以重载到移动构造函数了，移动构造函数将指针赋值一下就好了，不用深拷贝了，提高性能
> - move本质上就是一个`static_cast<X&&>`，强制转换为右值
> - 使用场景：**可移动对象在<需要拷贝且被拷贝者之后不再被需要>的场景，建议使用`std::move`触发移动语义，进行指针的拷贝，提升性能。**
> - 移动语义一定会修改临时变量的值，所以不能用`const Array&&`接收

```
class Array {
public:
    ......

    // 优雅
    Array(Array&& temp_array) {
        data_ = temp_array.data_;
        size_ = temp_array.size_;
        // 为防止temp_array析构时delete data，提前置空其data_
        temp_array.data_ = nullptr;
    }

public:
    int *data_;
    int size_;
};
```

如何使用：

```
// 例1：Array用法
int main(){
    Array a;

    // 做一些操作
    .....

    // 左值a，用std::move转化为右值
    Array b(std::move(a));
}
```

- 注意：`std::move`本身移动不了什么，唯一的功能是把左值强制转换成右值，实现等同于一个类型转换：`static_cast<T&&>(lvalue)`，所以单纯的`std::move(xxx)`不会有性能提升，**通常使用`std::move`结合类的构造函数实现移动语义或者通过右值引用延长返回值生命周期。**

```
int &&ref_a = 5;
ref_a = 6;

// 等同于以下代码：

int temp = 5;
int &&ref_a = std::move(temp);
ref_a = 6;
```

向成员传递的移动语义

> - 写了移动构造函数之后一定要写一个对应的拷贝构造函数（eg1）

```
// eg1
class HugeMem
{
public:
    HugeMem(int size) {
        sz = size;
        c = new int[sz];
    }
    ~HugeMem() {
        delete[] c;
    }
    HugeMem(HugeMem&& hm) {
        sz = hm.sz;
        c = hm.c;
        hm.c = nullptr;
    }

    int* c;
    int sz;
};
class Moveable
{
public:
    Moveable() : i(new int(3)), h(1024) { }
    ~Moveable() { delete i; }
    Moveable(Moveable&& m) noexcept // 移动构造函数应声明为noexcept，因为如果在m.i = nullptr之前抛出异常，会导致在调用函数中的m.i依然可以访问
        : i(m.i)
        , h(move(m.h))  // 移动语义向成员变量的传递
    {
        m.i = nullptr;
    }

    int* i;
    HugeMem h;
};

```

```cpp
Moveable GetTemp() {
    Moveable tmp = Moveable();
    return tmp;
}

int main() {
    Moveable a(GetTemp());
    return 0;
}
```

首先，`GetTemp()`返回一个类型为Moveable的临时变量（右值），随后调用Moveable的移动构造函数（注意在移动构造函数中，该返回值被一个右值引用接收，转换成了左值）。此时，m为左值，m.i，m.h也都为左值。

然后，在移动构造函数的初始化列表中，完成了对i的赋值，同时调用了HugeMem中的移动构造函数完成了对成员变量h的赋值；在函数体中，通过`m.i = nullptr`完成对地址的置空，防止重复释放（HugeMem的初始化同理），不过这里可以讲两个比较有趣的点

- 如果我们把`h(move(m.h))`改成`h(m.h)`，会发现编译报错，这是因为`h(m.h)`会调用拷贝构造函数，而在重载了移动构造函数之后，拷贝构造函数被隐藏，因此报错，所以**移动构造和拷贝构造必须同时提供**，只声明其中一种的话，类都仅能实现一种语义（诸如`unique_ptr`例外）。
- 如果我们对Moveable中的每一个构造函数与析构函数中都加上cout语句的话，会发现在整个程序中，只创建了一个Moveable的对象，这可能是因为编译器的返回值优化，g++/clang++中使用`"-fno-elide-constructors"`关闭返回值优化即可（page103）。

返回值优化？？？

C++的那些事——返回值优化 - 知乎 (zhihu.com)

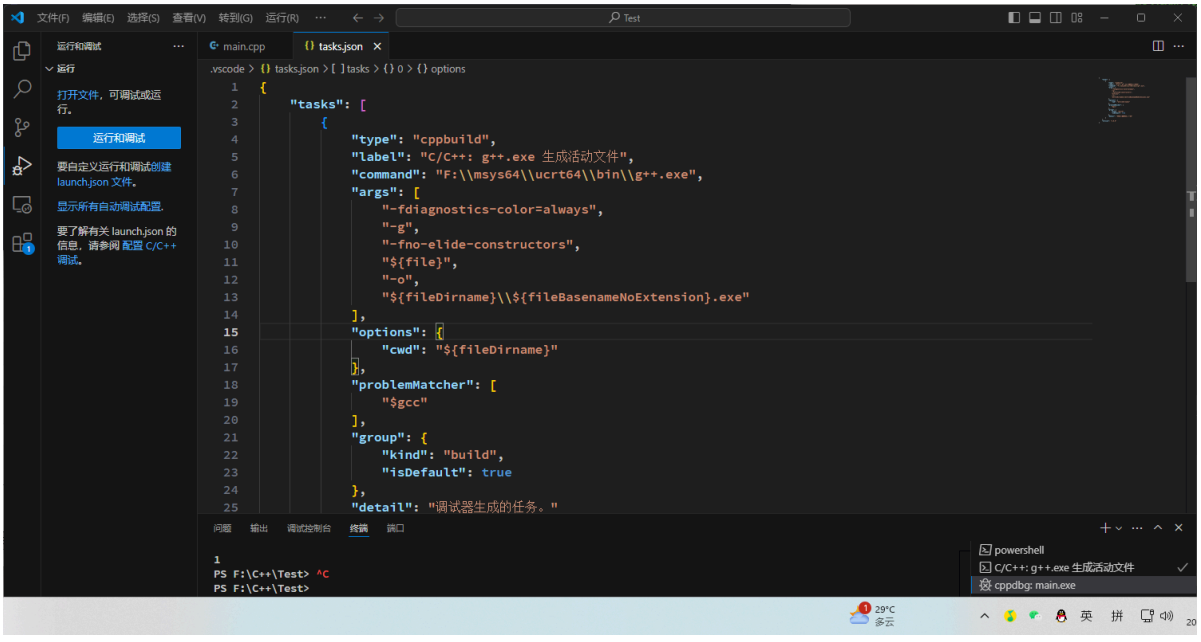
返回值优化在编译的哪个阶段完成？（我感觉是在编译期，可以自己在vscode里试试，自己在每个阶段都用用这个关闭选项，哪个阶段能正常运行就用哪个）

在编译期的原因（Linux编程.docx中有编译阶段讲解）：

预处理：展开头文件，删注释，不太可能

汇编：将汇编代码转换成机器码，不可能

链接：将exe链接lib等库文件，不可能



移动语义实现高性能置换

```
template <class T>
void swap(T& a, T& b)
{
    T tmp(move(a));
    a = move(b);
    b = move(tmp);
}
```

- 如果T是可以移动的，那么移动构造和移动赋值将会被用于这个置换。整个过程，代码都只会按照移动语义进行指针交换，不会有资源的释 放与申请。
- 而如果T不可移动却是可拷贝的，那么拷贝语义会被用来进行置换。这与普通的置换语句是相同的

move_if_noexcept

该函数用于替代`move()`。该函数在类的移动构造函数没有noexcept 关键字修饰时返回一个左值引用从而使变量可以使用拷贝语义，而在类的移动构造函数有noexcept 关键字时，返回一个右值引 用，从而使变量可以使用移动语义。

```
int main() {
    Moveable a;
    Moveable b(move_if_noexcept(a));

    // 编译选项: g++ -std=c++11 2-6-1.cpp
}
```

万能引用？？？

引用折叠

page103

```
typedef const int T;
typedef T& TR;
TR& v = 1;          // 该申明再C++98中导致编译错误
```

其中`TR& v=1`这样的表达式会被编译器认为是不合法的表达式，而在C++11中，一 旦出现了这样的表达式，就会发生引用折叠，即将复杂的未知表达式折叠为已知的简单表达式，具体如下图。

表 3-2　C++11 中的引用折叠规则

TR 的类型定义	声明 v 的类型	v 的实际类型
T&	TR	A&
T&	TR&	A&
T&	TR&&	A&
T&&	TR	A&&
T&&	TR&	A&
T&&	TR&&	A&&

如果定义中出现了左值引用，引用折叠总是优先将其折叠为左值引用。而模板对类型的推导规则就比较简单，当转发函数的实参是类型的一个左值引用，那么模板参数被推导为X&类型，而转发函数的实参是类型X的一个右值引用的话，那 么模板的参数被推导为X&& 类型。结合以上的引用折叠规则，就能确定出参数的实际类型。 进一步，我们可以把转发函数写成如下形式

```cpp
template <typename T>
void IamForwording(T&& t){
    IrunCodeActuallyy(static_cast<T&&>(t));
}
```

- 对于左值引用而言：调用了`IrunCodeActuallyy()`中参数为左值引用的重载函数。
- 对于右值引用而言：当它使用右值引用表达式引用的时候，**该右值引用是个左值**，如果不用`static_cast`的话，那么调用的就是参数为左值引用的重载函数。如果我们想在函数调用中继续传递右值，就需要使用`move`来进行左右值的转换。而`move`通常就是一个`static_cast`，最终调用了`IrunCodeActuallyy()`中参数为右值引用的重载函数。

完美转发？？？

不过在C++11中，用于完美转发的函数却不再叫作`move`，而是另 外一个名字`forward`。所以我们可以把转发函数写成这样:

```cpp
template <typename T>
void IamForwording(T&& t){
    IrunCodeActuallyy(forward(t));  // move 和 forward 在实际实现上差别并不大。
}
```

实际应用：

- 用于包装函数：

```cpp
#include <iostream>
using namespace std;

template <typename T, typename U>
void PerfectForward(T&& t, U& Func)
{
    cout << t << "\tforwarded..." << endl;
    Func(forward<T>(t));
}
void RunCode(int& x) {}
void RunHome(int&& y) {}
void RunComp(double& z) {}

int main() {
    PerfectForward(1, RunCode);
    PerfectForward(2, RunHome);
    PerfectForward(2.3, RunComp);

    return 1;
}
```

- 实现make_pair，make_unique

运算符重载

[c++基础梳理（九）：运算符重载 - 知乎 (zhihu.com)](#)

[运算符重载详解 - 上](#)

[【C++ | 重载运算符】一文弄懂C++运算符重载，怎样声明、定义运算符，重载为友元函数_c++ 重载-CSDN博客](#)

- 根据返回的内容判断返回值用引用/非引用

  > 返回的内容是重载函数中定义的临时变量，则返回非引用
  >
  > 返回的内容是对象本身（this指针）
  >
  > - 返回值为引用：使用链式编程。eg：++(++OBJ_NAME)；OBJ_NAME = OBJ_1 = OBJ_2
  > - 返回值为非引用：不使用链式编程

- 后置--/++的伪参数能否用其他类型?

  > 不行，只能用int

- 二元操作符的参数传递理解

  > 两个参数，一个参数随对象通过this指针进行传递

- 运算符重载的两种方法

  > 使用成员函数重载
  >
  > 使用全局函数搭配friend关键字（访问类中的私有成员）进行重载

运算符格式：

```
返回值类型 operator 运算符(形参列表)     // 一般运算符重载都是搭配着类来使用，eg：++OBJECT_NAME
{
    ...
}
```

operate<<:

> 流对象入参不能复制，只能引用
>
> 重载时只能声明为友元函数，因为要实现cout在左侧

```
class Person {
private:
    std::string name;
    int age;
public:
    Person(string name, int age) : name(name), age(age) {}
    // 声明友元函数，为实现cout在左侧，左移运算符重载只能使用友元函数实现
    friend ostream& operator<<(ostream& os, const Person& person);
};
// 流对象中不能复制，只能引用
ostream& operator<<(ostream& os, const Person& person) {
    os << "Name: " << person.name << ", Age: " << person.age;
```

```
    return os;
}
```

operate++（前置/后置）：

> 可以用右值引用接收后置++的返回值

```
#include<iostream>
#include<string>
using namespace std;
class myinteger {
    friend ostream& operator<<(ostream& cout, const myinteger&& myint);
    friend ostream& operator<<(ostream& cout, const myinteger& myint);
private:
    int mynum;
public:
    myinteger() : mynum(0){}
    // 前置++重载,
    // 前置++返回值有两种形式：①void（无法使用链式编程） ②引用
    myinteger& operator++() {
        mynum++;
        return *this;//返回自身用this
    }
    //后置++重载,
    //占位参数只能是int，不能是其他类型
    myinteger operator++(int) {//返回值只能是非引用，因为temp为临时对象，不过在外面可以通过右值引用来延长temp生命周期
        myinteger temp = *this;
        mynum++;
        return temp;
    }
};

//对于使用了后置++的类来说，传参时需要用右值引用来接收temp的临时对象
ostream& operator<<(ostream& cout, const myinteger&& myint) {
    cout<< myint.mynum;
    return cout;
}

// 对于非临时对象而言，直接使用左值引用即可
ostream& operator<<(ostream& cout, const myinteger& myint) {
    cout << myint.mynum;
    return cout;
}

int main() {
    myinteger myint;
    // 前置递增
    cout << ++(++myint) << endl;
    cout << myint << endl;
    // 后置递增
    cout << myint++ << endl;        // 调用的右值引用的那个重载函数
    cout << myint << endl;
    system("pause");
    return 0;
```

```
}
```

operate=:

为了实现链式编程，返回值应该使用引用

```
class myinteger {
private:
    int mynum;
public:
    myinteger() : mynum(0){}

    myinteger& operator=(const myinteger& myint) {
        if (this != &myint)// 如果两个对象的地址不相同那么就可以进行赋值
        {
            this->mynum = myint.mynum;
        }
        return *this;
    }
};
```

operate():

C++重载运算：函数调用运算符_重载函数调用运算符-CSDN博客

> 如果类重载了函数调用运算符，则我们可以像使用函数一样使用该类的对象。因为这样的类同时也能存储状态，所以与普通函数想比它们更加灵活。

类型转换运算符重载

- 普通类型 <==> 普通类型

  1. 标准数据类型之间会进行隐式类型安全转换
  2. 转换规则如下：

- 普通类型 ==> 类类型

  转换构造函数（不是拷贝构造函数，是只有一个参数的有参构造函数）

```
#include <iostream>
#include <string>

using namespace std;

class Test
{
    int mValue;
public:
    Test() : mValue(0) {}
    explicit Test(int i){       // 有参构造函数
        mValue = i;
    }
```

```
    Test operator + (const Test& p){
        Test tmpret(mValue + p.mValue);
        return tmpret;
    }

    int value(){
        return mValue;
    }
};

int main()
{
    Test t1;
    t1 = 10;  //Error 在 不使用explicit && 有参构造中有且仅有一个参数时，t1=10正确
    t1 = static_cast<Test>(10);    // t = Test(5);

    Test r;
    r = t1 + static_cast<Test>(5);   // r = t1 + Test(5);使用了 有参构造 和operate+重载
    cout << r.value() << endl;

    return 0;
}
```

- 类类型 ==> 普通类型

  函数原型：`operate Type();`

> 1. 类型转换运算符与一元有参构造函数具有同等的地位
> 2. 使得编译器有能力将对象转化为基本数据类型
> 3. 编译器能够隐式的使用类型转换函数

```
#include <iostream>
#include <string>

using namespace std;

class Test
{
    int mValue;
public:
    Test(int i = 0) : mValue(i) { }
    int value() { return mValue; }
    operator int()
    {
        return mValue;
    }
};

int main()
{
    Test t(100);
    int i = t;          // 隐式调用了operate int()

```

```
    cout << "t.value() = " << t.value() << endl;
    cout << "i = " << i << endl;

    return 0;
}
```

- 类类型 <==> 类类型

> 需要用explicit修饰一元有参构造
>
> 不过更好的办法是像qt那样将类类型转换函数定义成公共成员函数（str.toInt(); str.toDouble;）

```
#include <iostream>
#include <string>

using namespace std;

class Value
{
public:
    Value(){}
    Value(Test& t){
        cout << "一元有参构造" << endl;
    }
};

class Test
{
    int mValue;
public:
    Test(int i = 0) : mValue(1) {}
    int value(){ return mValue; }
    operator Value()        // 类型转换运算符重载
    {
        Value ret;
        cout << "operator Value()" << endl;
        return ret;
    }

};

int main()
{
    Test t(100);
    Value v = t;    // 一元有参构造函数 和 类型转换运算符 发生冲突，编译器不知道应该调用哪个函数。因此发生了错误。
                    // 解决办法：将 一元有参构造 申明为explicit，这样隐式转换时会调用类型转换运算符
    return 0;
}
```

operator*和operator->及智能指针类的实现

C++知识点43——解引用运算符和箭头运算符的重载及智能指针类的实现_重载解引用运算符-CSDN博客

operate[]:

C++重载[]（下标运算符）详解-CSDN博客

> operate[ ]必须以成员函数的形式进行重载，格式如下
>
> ```
> 返回值类型& operator[ ] (参数);     // 表示对应元素可读可写
> const 返回值类型 & operator[ ] (参数) const;    // 表示对应元素只读，用于提供给常对象使用
> 实际开发中两种形式都应该提供
> ```

重载下标运算符“[ ]”时，认为它是一个双目运算符，例如 X[Y] 可以看成：

```
[ ]-----双目运算符;
X-----左操作数;
Y-----右操作数。
也可以看成 X.operate[](Y);
```

```
#include <iostream>
using namespace std;
class Array {
public:
    Array(int length = 0) : m_length(length) {
        if (length == 0) {
            m_p = NULL;
        }
        else {
            m_p = new int[length];
        }
    }
    ~Array(){ delete[] m_p; }
public:
    int& operator[](int i) {                // 重载operate[]
        return m_p[i];
    }
    const int& operator[](int i) const{     // 重载只读operate[]
        return m_p[i];
    }
public:
    int length() const { return m_length; }
private:
    int m_length;  //数组长度
    int* m_p;  //指向数组内存的指针
};

int main()
{
    Array objarr(1);
    cout << objarr[0] << endl;  // 调用operate[]

    const Array constobjarr(1);
```

```
    cout << constobjarr[0] << endl; //  调用只读operate[]
    return 0;
}
```

operate"" YourLiteral用户自定义字面量/自定义后缀操作符

C++11 用户自定义字面量-CSDN博客

函数原型：`ReturnType operator"" YourLiteral(ParmType value)`

> - 通过实现一个后缀操作符，将 申明了该后缀标识的字面量 转化为 需要的类型（一般时将 const char* 或 unsigned long long 转换成对应类的对象）。
> - 合法的参数列表：
>
> ```
> const char *
> unsigned long long
> long double
> const char *, size_t
> const wchar_t *, size_t
> const char16_t *, size_t
> const char32_t *, size_t
> ```
>
> - 显然需要定义成全局函数

```
#include<iostream>
#include<string>
using namespace std;
struct RGBA {
    uint8_t r, g, b, a;
    RGBA(uint8_t r, uint8_t g, uint8_t b, uint8_t a) :r(r), g(g), b(b), a(a)
{}
};
// 自定义后缀操作符，将字符串常量转换成RGBA类对象，其中size表示字符串长度
RGBA operator"" _RGBA(const char* str, size_t size) {
    const char* r = nullptr;
    const char* g = nullptr;
    const char* b = nullptr;
    const char* a = nullptr;
    for (const char* p = str; p != str + size; ++p) {
        if (*p == 'r') r = p + 1;
        if (*p == 'g') g = p + 1;
        if (*p == 'b') b = p + 1;
        if (*p == 'a') a = p + 1;
    }
    if (r == nullptr || g == nullptr || b == nullptr) throw;
    if (a == nullptr) {
        return RGBA(atoi(r), atoi(g), atoi(b), 0);
    }
    else {
        return RGBA(atoi(r), atoi(g), atoi(b), atoi(a));
    }
}

// 输出运算符重载
ostream& operator<<(ostream& os, const RGBA& color) {
```

```
    return os << "r=" << (int)color.r << " g=" << (int)color.g << " b=" << (int)color.b << " a=" << (int)color.a << endl;
}

int main() {
    //自定义字面量来表示RGBA对象
    cout << "r255 g255 b255 a40"_RGBA << endl;// 这里先调用operate"" _RGBA，后调用operate<<

    system("pause");
    return 0;
}
```

C++类型转换？？？

C++类型转换：隐式转换和显式转换_c++隐式转换-CSDN博客

???C++隐式转换

隐式转换

- 基本类型之间会隐式转换
- nullptr可以转换为任意类型指针
- 任意类型指针可以转换为void指针
- 子类指针可以转换为父类指针
- 类的隐式转换（最容易产生风险）
    - 单参数构造函数：允许隐式转换特定类型来初始化对象。
    - 赋值操作符：允许从特定类型的赋值进行隐式转换。
    - 类型转换操作符：允许隐式转换到特定类型
    - 初始化列表：多参数的构造函数支持隐式类型转换来初始化对象。

> 对于`Test aa3 = 3`而言，是一个隐式类型转换。首先会调用`Test(int a)`来将3构造成一个Test类型的临时变量，然后再调用拷贝构造函数来构造aa3。**编译器遇到有参构造+拷贝构造->优化为直接调用有参构造。**

```
class Test {
public:
    Test(int a):m_val(a) {}
    bool isSame(Test other)
    {
        return m_val == other.m_val;
    }
private:
        int m_val;
};

int main(void) {

    Test a(10);
    if (a.isSame(10)) //该语句将返回true
    {
        cout << "隐式转换" << endl;
```

```
    }
    return 0;
}
```

本来用于两个Test对象的比较，竟然和int类型相等了。这里就是由于发生了隐式转换，实际比较的是一个临时的Test对象。这个在程序中是绝对不能允许的。

explicit关键字

> - 用于禁止在创建对象时发生隐式转换
> - **只适用于单参的构造函数，不能用于多参构造，拷贝构造等其他构造函数或普通函数。（当多参构造函数只剩一个参数没有定义默认参数时也可以用）**

```cpp
#include <iostream>
using namespace std;

class A {
public:
    A(int x) : m_x(x) {}
private:
    int m_x;
};

class B {
public:
    explicit B(int x) : m_x(x) {}
private:
    int m_x;
};

int main() {
    A a1 = 1;// 可以隐式转换

    B b1 = 2;// 错误：只能显式转换
    B b2 = static_cast<B>(3)// 正确：显式转换
    return 0;
}
```

C++类型转换运算符

C++强制类型转换

使用方法：`new_type result = cast_operator<new_type> (expression);`

static_cast

> - 编译时检查，确保指针被转换为相关类型
> - 用途：
>   - 用于基类和子类之间指针或引用的转换（向上转换安全，向下转换不安全，因为没有动态类型检查）
>   - 用于基本数据类型之间的转换（不保证安全性），以及左值引用转换为右值引用
> - static_cast不能转换掉表达式的const、volatile、或者__unaligned属性。

基本数据类型转换：

```cpp
char a = 'a';
int b = static_cast<char>(a);//正确，将char型数据转换成int型数据

double *c = new double;
void *d = static_cast<void*>(c);//正确，将double指针转换成void指针

int e = 10;
const int f = static_cast<const int>(e);//正确，将int型数据转换成const int型数据

const int g = 20;
int *h = static_cast<int*>(&g);//编译错误，static_cast不能转换掉g的const属性
```

类的向上和向下转换：

```cpp
if(Derived *dp = static_cast<Derived *>(bp)){//下行转换是不安全的
  //使用dp指向的Derived对象
}
else{
  //使用bp指向的Base对象
}

if(Base*bp = static_cast<Derived *>(dp)){//上行转换是安全的
  //使用bp指向的Derived对象
}
else{
  //使用dp指向的Base对象
}
```

dynamic_cast

C++强制类型转换操作符 dynamic_cast - 狂奔~ - 博客园 (cnblogs.com)

语法：

```cpp
dynamic_cast<type*>(e);             // 指针转换失败时，返回nullptr
dynamic_cast<type&>(e);            // 引用转换失败时，会抛出std::bad_cast异常
dynamic_cast<type&&>(e);         // 只能转换成指针/引用
```

> - 运行时检查，所以前提是子类重写父类虚函数，比static_cast更安全
> - 用途：
>
>   用于基类与子类之间的向下转换（因为转换开销比static_cast大）
> - 当 基类指针所指对象为基类类型 时，向下转换失败（和动态多态的使用及条件很像）

```cpp
#include<iostream>
using namespace std;

class Base
{
public:
    Base() {};
    virtual void Show() { cout << "This is Base calss"; }
};
class Derived :public Base
```

```
{
public:
    Derived() {};
    void Show() override { cout << "This is Derived class"; }
};
int main()
{
    // 基类指针指向子类对象，向下转换成功
    Base* base = new Derived();
    if (Derived* der = dynamic_cast<Derived*>(base))
    {
        cout << "success" << endl;
    }

    // 基类指针指向基类对象，向下转换失败
    Base* base1 = new Base();
    Derived* d = dynamic_cast<Derived*>(base1);
    if(nullptr == d)
    {
        cout << "fail" << endl;
    }

    system("pause");
    return 1;
}
```

const_cast

C++标准转换运算符const_cast - Ider - 博客园 (cnblogs.com)

语法：

```
const_cast<type*>(e);
const_cast<type&>(e);
const_cast<type&&>(e);// 只能转换成指针/引用
```

> - 用途：
>
>   用来移除变量的const或volatile限定符。
> - 使用const_cast后修改result并不会改变expression的值（其中对result的修改是未定义行为，由编译器决定如何处理）但是两者地址是一样的

```
int main()
{
    const int a = 10;
    int* b = const_cast<int*>(&a);
    *b = 7;                          // 未定义行为，由编译器决定如何处理
    cout << b << " " << &a << endl; // 两者地址一样
    cout << *b << " " << a << endl; // *b=7，a=10

    return 0;
}
```

reinterpret_cast

C++标准转换运算符之 reinterpret_cast_reinterpret cast-CSDN博客

C++标准转换运算符reinterpret_cast - Ider - 博客园 (cnblogs.com)

> - 用途：
>
> 用来处理无关类型之间的转换，强制编译器接受 static_cast 不允许的类型转换；它会产生一个新的值，这个值会有与原始参数（expressoin）有完全相同的比特位。
> - 使用场景（指针 <==>整数，类A指针<==>类B指针）：
> - 从指针类型到一个足够大的整数类型
> - 从整数类型或者枚举类型到指针类型
> - 从一个指向函数的指针到另一个不同类型的指向函数的指针
> - 从一个指向对象的指针到另一个不同类型的指向对象的指针
> - 从一个指向类函数成员的指针到另一个指向不同类型的函数成员的指针
> - 从一个指向类数据成员的指针到另一个指向不同类型的数据成员的指针
> - 转换后的类型值 需要转换回 原始类型，这样才是正确使用reinterpret_cast方式。

```
// reinterpret_cast不会对转换做检查，只会指示编译器将 表达式 当成新类型。
#include <iostream>
using namespace std;
class A {
public:
    int m_a;
};

class B {
public:
    int m_b;
};

class C : public A, public B {};

int main()
{
    C c;
    B* br = reinterpret_cast<B*>(&c);
    B* bs = static_cast <B*>(&c);
    cout << &c << " " << br << " " << bs << endl;

    return 0;
}

```

一般用在哈希函数中

C++11初始化列表

C++11列表初始化

列表初始化及decltype_列表初始化类中的结构体-CSDN博客

【C++从练气到飞升】24--C++11：列表初始化 | 声明 | STL的升级-CSDN博客

C++98中的{}

仅允许使用花括号 { } 对**数组**或者**结构体元素**进行统一的列表初始化

```
struct Point
{
    int _x;
    int _y;
};

int main()
{
    int arrya1[] = { 1, 2, 3, 4 };//列表初始化，初始化数组
    int array2[5] = { 0 };//列表初始化，初始化数组
    Point p = { 1, 2 };//列表初始化，初始化结构体元素
    Point array3[] = { {1, 2}, {3, 4}, {5, 6} };//列表初始化，初始化结构体数组
    return 0;
}
```

C++11中的{}

用于内置类型初始化

使用初始化列表时，可以添加等号（=），也可以不添加，其中new表达式中不可带等号

```
int main()
{
    //C++11 中列表初始化应用在内置类型上
    int x1 = 10;
    int x2 = { 20 };
    int x3{ 30 };//不带等号

    //C++11 中列表初始化也可以适用于 new 表达式中
    int* p1 = new int[5]{100, 112, 113};
    return 0;
}
```

用于用户自定义的类型初始化

创建对象时也可以使用列表初始化的方式来调用构造函数初始化

> 如果在 Date 类的构造函数加上 explicit，那么 Date d3 = { 2022, 1, 3 }; 就会出现编译报错，因为这条语句本质上是因为**多参数的构造函数支持隐式类型转换**。Date d2{ 2022, 1, 2 }; 没事，仍然可以正常运行。

```

```

```cpp
class Date
{
public:
    Date(int year, int month, int day) : _year(year), _month(month), _day(day)
    {
        cout << "Date(int year, int month, int day)" << endl;
    }
private:
    int _year;
    int _month;
    int _day;
};

int main()
{
    Date d1(2022, 1, 1); // old style

    // C++11支持的列表初始化，这里会调用构造函数初始化
    Date d2{ 2022, 1, 2 };
    Date d3 = { 2022, 1, 3 };//本质上是多参数的构造函数支持隐式类型转换，注意此处由于是在初始化阶段，所以调用的是有参构造函数，不是operate=()
    const Date& d1 = { 2003, 10, 18 };//(正确)
    return 0;
}
```

但是如果稍加更改

```cpp
// 添加了一个initializer_list作为参数的构造函数
#include <iostream>
#include <initializer_list>
using namespace std;

class Date
{
public:
    Date(int year, int month, int day) : _year(year), _month(month), _day(day)
    {
        cout << "Date(int year, int month, int day)" << endl;
    }

    explicit Date(initializer_list<int> il)
    {
        cout << "Date(initializer_list<int> il)" << endl;
    }

private:
    int _year;
    int _month;
    int _day;
};

int main()
{
```

```
    Date d2{ 2022, 1, 2 };       // 编译通过，调用了一元有参构造函数
                                 // 我理解成Date d2({ 2022, 1, 2 })
    Date d1 = { 2022, 1, 2 };    // 编译错误：复制列表初始化不能使用标记为“显式”的构造函数
                                 // 也就是说，此处也调用了一元有参构造，而不是三元有参构造
    auto e = { 2022, 1, 2 };
    Date d3 = static_cast<Date>(e); // 编译通过，调用了一元有参构造函数
    Date d4(2022, 1, 2 );           // 编译通过，调用三元有参构造
    return 0;
}
```

如上：**在添加了一个`initializer_list`作为参数的构造函数之后，使用列表初始化时就只会调用这个构造函数了，只能使用传统方式调用三元有参构造**

typeid(变量名).name()

C++11 标准中，`typeid(变量名).name()`函数返回的是一个 const char*，指向一个以 null 终止的字符串，表示类型的类型名称。

initializer_list

一般作为容器中构造函数的参数

```
int main()
{
    auto il = { 1, 2, 3 };
    cout << typeid(il).name() << endl;
    return 0;
}
// 输出结果：
// class std::initializer_list<int>
```

用于容器初始化以及赋值

新增 initializer_list 作为参数的构造函数和operator=，创建对象时使用 initializer_list 来调用构造函数初始化

> 需要注意，下面代码中创建 Date 类型的对象 d1，本质上是隐式类型的转换。但是**容器存储的数据个数可多可少，并不确定，所以提供了形参为 initializer_list 的构造函数**，这样就方便我们将任意数量的元素存到容器中。而 Date 作为一个日期类对象，它的三个成员变量是固定的，所以不需要提供形参为 initializer_list 的构造函数。因此**自定义类的列表初始化本质上是隐式类型转换，而容器类的列表初始化本质上是使用 initializer_list 作为形参的构造函数。**

```cpp
int main()
{
    //调用vector中形参为 initializer_list 的构造函数
    vector<int> v = { 1,2,3,4 };
    // 这里先隐式转换，调用有参构造创建一个pair对象，再调用形参为 initializer_list 的构造函数
    map<string, string> dict = { {"sort", "排序"}, {"insert", "插入"} };
    //调用vector中形参为 initializer_list 的赋值运算符重载函数
    v = { 10, 20, 30 };

    Date d1 = { 2003, 4, 5 };//这里是直接调用3个参数的构造函数 --- 隐式类型转换

    return 0;
}
```

防止类型收窄

类型收窄一般是指一些可以使得数据变化或者精度丢失的隐式类型转换。可能导致类型收窄的典型情况如下:

- 从浮点数隐式地转化为整型数。比如:int a=1.2，这里a实际保存的值为整数1，可以视为类型收窄。
- 从高精度的浮点数转为低精度的浮点数，比如从 long double 隐式地转化为double，或 从double转为float。如果这些转换导致精度降低，都可以视为类型收窄。
- 从整型（或者非强类型的枚举）转化为浮点型，如果整型数大到浮点数无法精确表示，则也可以视为类型收窄。
- 从整型（或者非强类型的枚举）转化为较低长度的整型，比如:unsigned char=1024， 1024明显不能被一般长度为8位的unsigned char所容纳，所以也可以视为类型收窄。

C++11中，使用初始化列表进行初始化的数据编译器是会检查其是否发生类型收窄的

```cpp
int main()
{
    const int x = 1024;
    const int y = 10;

    char a = x;                     // 收窄，但可以通过编译
    char* b = new char(1024);       // 收窄，但可以通过编译

    char c = { x };                 // 收窄，无法通过编译
    char d = { y };                 // 可以通过编译
    unsigned char e{ -1 };          // 收窄，无法通过编译

    float f{ 7 };                   // 可以通过编译
    int g{ 2.0f };                  // 收窄，无法通过编译
    float* h = new float{ 1e48 };   // 收窄，无法通过编译
    float i = 1.21;                 // 可以通过编译

    return 0;
}
```

空基类优化???（可以编在简历里）

C++惯用法之空基类优化-CSDN博客？？？

> 当空类作为基类时，只要两个相同类型的子对象偏移量不同，就不需要为其分配任何空间

非空基类在子类中的内存布局

```
struct A {
    int av;
};

struct B : A{
    int vvv;
    A vv;
};

int main()
{
    B b;
    cout << hex << &b << endl;      // 00000037EEB3FCA8：子类对象地址
    cout << hex << &b.vvv << endl;  // 00000037EEB3FCAC：int变量地址
    cout << hex << &b.vv << endl;   // 00000037EEB3FCB0：基类子对象地址

    return 0;
}
```

显然，int变量地址与子类对象地址不一样，中间隔了四个字节，这个四个字节其实是继承的基类对象的地址。也就是说：**继承关系时，基类和派生类地址相同。**

空基类在子类中的内存布局

```
struct A { };   // 空基类

struct B : A{
    int vvv;
    A vv;
};

int main()
{
    B b;
    cout << hex << &b << endl;      // 000000847333F578：子类对象地址
    cout << hex << &b.vvv << endl;  // 000000847333F578：int变量地址
    cout << hex << &b.vv << endl;   // 000000847333F57C：基类子对象地址

    return 0;
}
```

显然，此时会发现int变量地址与子类对象地址一致，因为此时基类对象为空基类，**当空类作为基类时，只要两个相同类型的子对象偏移量不同，就不需要为其分配任何空间**。这个就叫做**空基类优化技术。**

取消空基类优化的情况

此段代码在msvc和g++中表现不一样，msvc好像对于下面两种情况依然用了空基类优化

```
// eg1
struct A { };   // 空基类

struct B : A {
    A vv;
    int vvv;
};

int main()
{
    B b;
    cout << hex << &b << endl;      // 0x5ffe98：子类对象地址
    cout << hex << &b.vv << endl;   // 0x5ffe99：基类子对象地址（msvc中两者地址相同）
    cout << hex << &b.vvv << endl;  // 0x5ffe9c：int变量地址

    return 0;
}
```

显然，初始化B的时候，由于基类对象地址与基类子对象地址会重合，所以取消了空基类优化。

```
// eg2
#include <iostream>
class EmptyClass{ };
class EmptyFoo : public EmptyClass{ };// 如果取消继承EmptyClass，则最终结果为：1, 1, 1
class NoEmpty :public EmptyClass,  public EmptyFoo{ };
int main(){
    std::cout << sizeof(EmptyClass) << std::endl; //输出：1
    std::cout << sizeof(EmptyFoo) << std::endl; //输出：1
    std::cout << sizeof(NoEmpty) << std::endl; //输出：2（msvc中为1）
}
```

显然，EmptyClass地址和EmptyFoo地址重合，所以取消空基类优化（两者属于继承关系，显然为同一类型）

应用分析

- 使用空基类优化减少占用的内存

```
struct A { };   // 空基类

struct B {
    A vv;
    int vvv;
};

struct B1 : private A {
    int vvv;
};
```

```cpp

int main()
{
    cout << sizeof(B) << endl;  // 8：A为基类子对象，不会用到空基类优化
    cout << sizeof(B1) << endl; // 4：A为空基类，运用了空基类优化

    return 0;
}
```

- std::tuple

POD类型？？？

【C/C++ POD 类型】深度解析C++中的POD类型？？？

page116

> 如果一个类型既是Trivial类型又是Standard layout类型，那么它就是POD类型。

POD（Plain Old Data，平凡旧数据）是C++中的一个概念，它指的是一种可以**通过简单内存复制进行复制和传输的数据类型**。POD类型的对象可以通过`memcpy`或其他等价的操作进行复制，而且它们的内存布局是完全透明和可预测的。

在C++中，POD类型可以分为两类：trivial（平凡）类型和standard layout（标准布局）类型。

平凡类型

平凡类/结构体定义如下：

- 拥有平凡的构造/析构函数

  不申明**默认构造/析构函数**，**拷贝构造函数**，**移动构造函数**，或使用`=default`显式申明默认版本的构造/析构函数时，该构造/析构函数为"平凡化的"

  由上，**只声明了有参构造函数也算“平凡化的”**

- 拥有平凡的拷贝，移动赋值运算符

  同上

- 不能包含虚函数与虚基类

```cpp
struct trival{
    int x;
    trival()=default;
};
struct notrival{
    int y;
    trival();
}
// 可以使用 is_trival<T>::value 来判断
```

标准布局类型

标准布局的类/结构体定义如下：

- 所有非静态成员拥有相同的访问权限
- 存在继承关系时

    - 非静态成员不能同时出现在子类和基类之间
    - 多重继承时，非静态成员不能同时出现在多个基类之间
- 类中第一个非静态成员类型与其基类不同（空基类优化相关知识）
- 没有虚函数和虚基类
- 所有非静态数据成员均符合标准布局类型，其基类也符合标准布局。

可以使用`is_standard_layout<T>::value`来判断

非受限联合体

【C++11新特性：13】—— C++11非受限联合体

page124

定义

- 联合体是一个结构体
- 它的所有成员相对于基地址的偏移量都为0
- 此结构空间要大到足够容纳最"宽"的成员；
- 其内存对齐方式要适合其中所有的成员；
- **任何非引用类型都可以成为联合体的数据成员**，这种联合体即非受限联合体（C++11）
- 静态成员变量不能作为联合体的数据成员（静态成员函数可以）（C++11）

语法

```
// 非受限联合体，需要创建对象后才能使用
union u {
    int a;
    double b;
};
// 非受限的匿名联合体，只能创建在类中，或者用static关键字创建在全局中，不用创建对象也能使用
union {
    int a;
    double b;
};
```

注意事项

如果非受限联合体内有一个非 POD 的成员，而该成员拥有自定义的构造函数，那么这个非受限联合体的默认构造函数将被编译器删除；其他的特殊成员函数，例如默认拷贝构造函数、拷贝赋值操作符以及析构函数等，也将被删除，此时无法创建非受限联合体的对象。

所以如果要在非受限联合体中使用非 POD 的成员，应使用placement new，并自己为非受限联合体定义构造函数。代码如下：

```cpp
#include <string>
using namespace std;
union U {
    string s;
    int n;
public:
    U() { new(&s) string; }
    ~U() { s.~string(); }
};
int main() {
    U u;
    return 0;
}
```

构造时，采用 placement new 将 s 构造在其地址 &s 上，这里 placement new 的唯一作用只是调用了一下 string 类的构造函数。注意，在析构时还需要调用 string 类的析构函数。

placement new

语法：

```cpp
new (place) Type(args...);
// place是一个指针，指向预分配的内存地址
// Type是要创建的对象类型
// args...是传递给构造函数的参数列表
```

`placement new`利用已经申请好的内存来生成对象，它不再为对象分配新的内存，而是将对象数据放在 address 指定的内存中。在本例中，`placement new`使用的是 s 的内存空间。

枚举式类

当非受限的匿名联合体运用于类的声明时，这样的类被称为“枚举式类”：

```cpp
class Student {
public:
    Student(bool g, int a) : gender(g), age(a) {}
    bool gender;
    int age;
};
class Singer {
public:
    enum Type { STUDENT, NATIVE };
    Singer(bool g, int a) : s(g, a) { t = STUDENT; }
    Singer(int i) : id(i) { t = NATIVE; }
    ~Singer() {}
private:
    Type t;
    union {     // 显然如果在非受限的匿名联合体中使用非POD成员，则不用自定义构造/析构函数
        Student s;
        int id;
    };
};

```

```cpp
int main() {
    Singer (true, 13);
    Singer(310217);

    return 0;
}
// 上面的代码中使用了一个匿名非受限联合体，它作为类 Singer 的“变长成员”来使用，这样的变长成员给类的编写带来了更大的灵活性。
```

内联名字空间

> - 内联名字空间可以和内联函数进行类比，两者都很相似，可以理解成**在父命名空间中展开了inline关键字修饰的子命名空间**。此时只要声明父命名空间即可使用inline关键字修饰的子命名空间中的成员。
> - 内联名字空间通常用来实现 ”根据编译器支持的C++版本调用不同代码“ 等类似用途。

原理：

C++内联命名空间：原理剖析与实战应用

语法：

```cpp
// 命名空间
namespace np{
    // ...
}
// 内联命名空间
inline namespace ilnp{
    // ...
}
```

例子：

```cpp
namespace NameA {
    inline namespace NameB {                                // 声明为inline
        struct Example {};
    }

    inline namespace NameC {                                // 声明为inline
        template<typename T> class Class {};
        Example exC;                                        // 不用打开名字空间NameB也能使用，显然此时会发现inline已经破坏了封装性
    }
}

namespace NameA {
    template<> class Class<Example> {};                     // 使用内联名字空间还能在不打开名字空间的情况下，进行模板特化（不用inline修饰则会编译失败）
}

using namespace NameA;
int main()
{
    Example ex;                                             // 不用打开名字空间NameB也能使用
```

```
    Class<Example> classEx;
    return 0;
}
```

实际用途：

```
// 根据编译器支持的C++版本调用不同代码
namespace NameA {
#if __cplusplus == 201103L
    inline
#endif //  __cplusplus
    namespace cpp11 {
        struct Example {};
    }

#if __cplusplus < 201103L
    inline
#endif //  __cplusplus
    namespace cppold {
        struct Example {};
    }
}

using namespace NameA;
int main()
{
    Example ex;                         // 根据编译器的C++版本默认选择inline版本
    cppold::Example ex;
    cpp11::Example ex;
    return 0;
}
```

参数关联名称查找（ADL）：

[c++ - What are the pitfalls of ADL? - Stack Overflow]

该特性允许编译器在命名空间内找不到函数名称的时候，在参数的命名空间内查找函数名字：

```
namespace ns_adl{
    struct A{};
    void ADLFunc(A a){} // ADLFunc定义在namespace ns_adl中
}
int main(){
    ns_adl::A a;
    ADLFunc(a); // ADLFunc无需声明命名空间
}
```

一般化的SFINEA规则？？？

page137

[《深入理解C++11》笔记-模板的别名、模板的SFINEA规则_c++ sfinea-CSDN博客]

[C++模板进阶指南：SFINAE - 知乎 (zhihu.com)]

auto

> - auto占位符在**编译期**起作用（不合标准的无法通过编译），auto修饰的变量必须初始化
> - **形式如auto&&，表示万能引用**，即既可以为左值引用，也可以为右值引用
> - 用 auto 来声明多个变量类型时，**只有第一个变量用于 auto 的类型推导**，然后推导出来的数据类型被作用于其他的变量。所以不允许这些变量的类型不同。
> - 声明为auto的变量**无法带走原始表达式的cv限制符**（即volatile和const属性），但auto&可以
> - 推导地址时，使用auto* 和auto没啥区别；但如果想声明引用，必须用auto&
> - auto从表达式推导出类型后，进行类型定义时，允许一些冗余的符号。比如`cv 限制符`，`&`，`*`，如果推导出的类型已经有了这些属性，冗余的符号则会被忽略。
> - 在结构体/类中，**非静态auto类型不能成员变量**
> - auto不能作为函数参数，不能作为模板实例化类型

```
int main() {
    int x = 1;
    const int xx = 1;

    auto x1 = &x;
    auto* x2 = &x;           // x1和x2的类型都一样

    auto& y1 = x, y2 = 1.3f; // auto被推导为int，所以y2的类型也为int，与右值类型不匹配，报错
    auto& z = xx;            // xx为const，所以z也为const

    auto&& i = move(x);      // i推导为右值引用
    auto&& i1 = xxx;         // i1推导为左值引用

    return 0;
}
```

auto的const和volatile规则：

```
   int num = 1;
   const auto a = num;             // const int类型
   const auto& b = num;            // const int&类型
   volatile auto* c = &num;        // volatile int*类型

   auto d = a;                     // int类型，auto无法带走const属性
   auto& e = a;                    // const int类型，auto引用可以
   auto f = c;                     // int*类型，auto无法带走volatile属性
   auto& g = c;                    // volatile int*类型，auto引用可以
```

auto不能使用的情况：

```
void func(auto x) {}            // auto不能作为函数参数

struct stu {
    auto var = 10;              // 非静态auto类型不能成员变量
    static const auto var = 10; // 编译通过
```

```
};

int main()
{
    char x[3] = { 0 };
    auto y = x;
    auto z[3] = x;              // auto不能作为数组类型

    std::vector<auto> v = { 1 };    // auto不能作为模板实例化类型

    return 0;
}
```

RTTI？？？

变长函数模板？？？

page158

decltype

[《深入理解C++11》笔记-decltype]

[C++11特性：decltype关键字 - melonstreet - 博客园 (cnblogs.com)]

[C++ decltype用法详解-CSDN博客]

> - decltype是一个类型推导运算符，在编译期起作用，decltype修饰的变量不用初始化
> - 当推导的类型包括`cv 限制符`，`&`，同时还额外声明了这些属性时，冗余的符号则会被忽略（冗余的`*`不可被忽略）
> - 推导地址时，使用`decltype(var_name)*`和`decltype(var_name)`有区别；
> - decltype可以带走cv限制符，但如果对象声明中有cv限制符时，该对象的cv 限制符不会出现在其成员的类型推导结果中。

语法：`decltype(expr);`

decltype推导四规则

- 如果type是一个没有带括号的标记符表达式(可以理解为变量名称)或类成员访问表达式，那么decltype(type)就是type对应的类型。此外，如果type是一个重载的函数，则会导致编译失败。
- 否则，假设type的类型为T，且是一个将亡值，那么decltype(type)为T&&。
- 否则，如果type是一个左值，那么decltype(type)为T&。
- 否则，decltype(type)为T。

```
struct S { double d; } s;
int func(int);
int func(double);
int&& rvalue();
const bool isvalue(int);

```

```cpp
int main()
{
    int i = 0;
    int arr[5] = { 0 };
    int* ptr = arr;

    decltype(arr) var1;   // int[5]，不带括号的标记符表达式
    decltype(ptr) var2;   // int*，不带括号的标记符表达式
    decltype(s.d) var3;   // double，不带括号的成员访问表达式
    decltype(func) var4;  // 编译失败，是重载的函数，否则var4被定义为函数类型

    decltype(rvalue()) var5 = 1; // int&&，将亡值

    // 左值推导为类型的引用
    decltype((i)) var6 = i;      // int&带括号的左值
    decltype(++i) var7= i;       // int&,++i返回i的左值
    decltype(arr[3]) var8 = i;  // int&,[]返回i的左值
    decltype(*ptr) var9 = i;     // int&,*返回i的左值
    decltype("lvalue") var10 = "lvalue";  // const char(&)[], 字符串字面常量为左值

    // 以上都不是,推导为本类型
    decltype(1) var11;    // int，处理字符串字面常量以外的值为右值
    decltype(i++) var14;  // int，  i++返回右值
    decltype((isvalue(1))) var15; // const bool,圆括号可以忽略，有问题！！！！去vscode上试试

    return 0;
}
```

cv限制符的继承与冗余的符号

上一篇中说到auto不能带走cv限制符(`const`、`volatile`)，`decltype`是能够带走的。不过，如果对象定义中有`const`或`volatile`限制符，使用`decltype`推导时，其成员不会继承cv限制符。

```cpp
struct st{
    int i = 0;
};

int main()
{
    int i = 0;
    int& j = i;
    decltype(i)& var1 = i;      // int& 类型
    decltype(j)& var2 = i;      // int& 类型，推导出的类型已经有引用，冗余的引用会被忽略

    int* ptr = &i;
    //decltype(ptr)* var3 = &i;     // 编译失败，int**裂隙不能指向int*类型
    decltype(ptr)* var4 = &ptr;     // int**类型，推导出的类型是指针，不会被忽略

    const int k = 0;
    decltype(k) var5 = 1;           // const int类型
    const decltype(k) var6 = 1;     // const int类型,推导出的类型已经有const，冗余的const会被忽略
```

```
    const st cst;
    decltype(cst.i) var7;           // int类型，decltype推导时，成员不继承对象的cv限制符

    return 0;
}
```

decltype推导的类型有时候会忽略一些冗余的符号，包括const、volatile、引用符号&。

数组指针

数组指针基础

数组指针进阶

> - 对于数组arr[5]而言，arr表示数组首个元素地址，&arr表示数组地址，两者值相同，意义不同

- 定义：

  数组指针即定义一个指向数组的指针变量，形如：

```
int(*p)[n]; // n为要定义的个数。
// 变量p的类型是int(*)[5]
```

- 使用typedef简化：

```
typedef int(*pArr)[4]; // pArr是一个指针类型，该指针指向一个可容纳4个int数据的数组
pArr pa;    // 相当于 int(*pa)[4]
```

- 注意事项：
  - 记得加*

```
typedef int(PointArr1)[4];  // 这个不是数组指针，相当于直接定义了一个数组，没有实际意义
PointArr1 pa1 = { 1,2,3,4 };
cout << pa1[1] << endl;
```

  - **&arr表示 整个数组 的首地址，该地址解引用后表示数组中第0个元素的首地址（虽然两者值一样）**

    arr表示这个数组的 第0个元素 的首地址，两者加1时的逻辑不同

```
typedef int(*PointArr)[4];  // PointArr是一个指向数组的指针，该数组可以存放4个int型数据。
int arr[4];
PointArr pa = &arr; // &arr表示 整个数组 的首地址，进行+1操作时，加的是sizeof(arr)，也就是16字节
int* paa = arr;     // arr表示这个数组的 第0个元素 的首地址，&arr和arr值一样，但进行+1操作时，加的是sizeof(int)
(*pa)[1] = 1;       //pa表示整个数组的首地址，解引用后(即*pa)表示数组中第0个元素的首地址
cout << *((*pa)+1) << endl; // 相当于(*pa)[1]
cout << *(paa+1) << endl;   // 相当于paa[1]
```

- 一维数组指针接收二维数组

```
int a[][4] = { {1,2,3,4},{5,6,7,8} };
PointArr p1 = a;                // a表示二维数组第0行数据（一维数组）的地址，与 PointArr p1 = &a[0] 等效
cout << *(*(p1+1)+1) << endl;   // *(p1+1)单独使用时表示的是第 1 行数据，因为使用整行数据没有实际的含义，所以放在表达式中会被转换为第 1 行数据的首地址
                                // *(*(p1+1)+1)表示的是第 1 行第一个元素，是int型数据，有实际意义，可以直接访问
```

- 定义二维数组指针没意义

```
typedef int(*PointArr1)[2][4];      // 编译报错：二维数组指针定义没意义
PointArr1 p11 = {*a[0], *a[1]};
cout << p11[0][1] << endl;
```

函数指针

[函数指针基础]

[指针，函数，数组]

[函数指针进阶]

> - **函数指针首先是个指针**，它指向的值是个函数；**指针函数首先是个函数**，它的返回值是个指针。根据这点去理解其他类似于"指针数组"，"数组指针"的概念。
> - 阅读函数指针时，从内往外看；简化函数指针的时候，从外往内看
> - 注意当函数指针作为函数返回值时，函数声明与函数指针的区别
> - 对于函数`void func();`而言，func和&func都表示函数地址，两者值相同（数组中两者意思不同）
> - `typedef int (*FUNC)(int, int);`表示申明了一个函数指针类型
>
>   `typedef int (FUNC)(int, int);`表示申明了一个函数类型，使用该类型创建的变量只准定义，不准赋值
>
>   `int (FUNC)(int, int);`表示一个函数声明，只能定义，不能创建变量。

- 定义：

  函数指针即定义一个指向函数的指针变量，形如：

```
int (*p)(int, int);
// 这个函数的类型是有两个整型参数，返回值是个整型。对应的函数指针类型：
// int (*) (int, int);
```

其中，`int (*p)(int,int)`可以理解成`int (*)(int,int) p`

- 注意事项：

```
typedef int (*FUNC)(int, int);      //定义一个FUNC代表 int(*)(int, int) 类型
// 也许你会想，定义了函数指针之后，赋值是地址传递，那能不能用值传递呢？答案是不能，原因如下：
// 首先你要明白，之所以对*FUNC要加括号，是因为优先级：()>[]> *，如果不加括号就变成了int* FUNC(int, int);是个指针函数
// 所以对于int (FUNC)(int, int);而言，括号加不加都一样，在去掉括号后，就变成了一个函数类型的申明，使用示例如下：
class A {
public:
    typedef int (FUNC)(int, int);
    FUNC fu;                // 此时FUNC是个函数类型，声明出fu后，fu相当于是个函数声明
};
void A::fu() { }
```

- 函数指针作为函数的返回值

  语法：

```
int (*AFunction(char *ch, int(*p)(int,int)))(int, int);
// 表示AFunction(char *ch,int (*p)(int,int))的返回值为：
// 类型为int (*)(int, int)的函数
```

  - **其中，`int (*AFunction(char *ch,int (*p)(int,int)))(int, int);`可以理解成：**

    `int (*)(int, int) AFunction(char *ch, int (*)(int,int) p);`

  - **该语句是一个函数声明，表示声明了一个叫AFunction的函数，而非函数指针。因为根据优先级关系，AFunction先和右边的括号结合，再和左边的*号结合，所以AFunction不是指针，是它的返回值是指针**

    如果想声明一个函数指针，应该写成这样：

    `int (*(*AFunction)(char *ch, int(*p)(int,int)))(int, int);`

- 使用typedef 简化指针定义

  简化函数指针的时候，从外往内看

```
typedef int (*Pointer)(int,int);    //Pointer等价于类型 int (*)(int,int)，int (*)(int,int)是类型名，Pointer是别名
Pointer p = add; //但是这里由于C语言语法的关系，我们不能写成 int (*)(int,int) Pointer 这样的形式
```

通过`typedef`就能将函数指针的定义统一成“**类型 变量**”的形式，我们再看一个复杂例子：

```
typedef int (*FUNC)(int, int);      //定义一个FUNC代表 int(*)(int, int) 类型
```

```
typedef FUNC(*pfuncAFun)(const char*, FUNC);//定义一个FUNC代表 int(*)(int, int) 类型

int add(int a, int b){
    return a + b;
}
// int (*AFunction(const char* ch, int(*p)(int, int)))(int, int)
FUNC AFunction(const char * ch, FUNC p) //这实际上就是 “类型名称 对象名称”，相比于上面未简化的函数表达式清晰不少
{
    if (ch == "add"){
        return p;
    }
    else
        return NULL;
}

int main() {
//  int (*(*p)(const char*, int(*)(int, int)))(int, int) = AFunction;//注意要用(*p)表示指针
    pfuncAFun p = AFunction; // 相比上面复杂的类型声明，此处简单不少
//  pfuncAFun p = &AFunction; 注意AFunction和&AFunction都表示函数地址，两者值相同

    FUNC p1 = p("add", add);
//  FUNC p1 = (*p)("add", add); 正常调用时，不论是否对p解引用，都可以正常运行该函数。

    return 1;
}
```

- 练习题：

```
(*(void(*)())0)();
// void(*)()：函数指针类型，我们定义为X
// (void(*)())0：即(X)0，将0强制转换为void(*)()类型的函数指针，将该指针定义为Y
// (*(void(*)())0)()：即(*Y)()，表示解引用调用0地址处的函数
```

```
void (*signal(int, void(*)(int)X))(int);
// 1.signal是一个函数声明
// 2.signal 函数有两个参数，第一个参数的类型是int，第二个参数的类型是 void(*)(int) 函数指针类型
// 3.该函数指针指向的函数有一个int类型的参数，返回类型是void
// 4.signal函数的返回类型也是void(*)(int) 函数指针类型，该函数指针指向的函数有一个int类型的参数，返回类型是void
```

```
int (*(*pf())())();
//  分析的时候由内向外
//  int (*(*X())())();
//  int (*Y)();
//  简化的时候由外向内
//  typedef int(*PF_I)();
//  typedef PF_I(*PF_PFI)();
//  PF_PFI pf() { return nullptr; }
```

函数指针数组

- 定义：

  函数指针数组指的是数组里存放的是函数指针：

```
int (FuncArr[5])(int,int);
// 声明了一个叫FuncArr的数组，该数组能存5个函数指针
// 其中函数指针的格式为：返回值为int，参数为int,int
```

函数指针数组的指针

- 定义：

  指向函数指针数组的指针是一个指针，指针指向一个数组，数组的元素都是函数指针。

```
int ((*pFuncArr)[5])(int,int);
// 声明了一个叫pFuncArr的指针，该指针指向一个数组
// 该数组能存5个函数指针
// 其中函数指针的格式为：返回值为int，参数为int,int
```

- 练习题：

  简化`int (*(*pFuncArr)[5])(int,int)`

```
#define OVER0(N) ((N) == 0 ? 1 : (N))
typedef int(*PFUNC)(int, int);
typedef PFUNC(*PFUNCARR)[OVER0(5)];

int (*(*pFuncArr)[5])(int,int);
PFUNCARR pFuncArrtest = pFuncArr;    // 可以正常赋值，说明简化成功

//int (*pFuncArr[5])(int,int);
//PFUNCARR pFuncArrtest = &pFuncArr;// 此时注意pFuncArr是个数组而非数组指针，所以需要取址
```

C++11中的函数指针

函数指针进阶

> - 使用auto进行推断时，auto* 和auto没有区别，都会推断为函数指针
> - 使用函数类型作为函数形参时，调用自动转换为函数指针
> - 类函数指针赋值要使用 **&**
> - 使用 **.*** (实例对象)或者 **->*** （实例对象指针）调用类成员函数指针所指向的函数

- typedef与decltype组合简化函数类型/函数指针类型的声明

```
void test() { }
int main() {
    typedef decltype(test)* pfu2;   // ①记得加*，否则变成了函数类型声明；②decltype无法推导重载函数
    pfu2 pfuu1 = test;

    return 1;
}
```

- 使用auto定义函数类型/函数指针类型

```
void test() { }
// 推导地址时，使用auto* 和auto没啥区别
auto pf = add;//pf为指向add的指针
auto *pf = add;//pf为指向add的指针
```

- 函数指针作为形参

```
typedef decltype(test) add2;
typedef decltype(test)* PF2;//推导地址时，使用decltype()* 和decltype()有区别；

void fuc2 (add2 add);//函数类型形参，调用自动转换为函数指针
void fuc2 (PF2 add);//函数指针类型形参，传入对应函数(指针)即可
```

- 使用decltype优化返回值为函数指针的函数声明

```
decltype(testa)* fuc2(int a);//明确知道返回哪个函数，可用decltype关键字推断其函数类型，
```

- 使用追踪返回类型优化返回值为函数指针的函数

```
auto fuc2(int a) -> decltype(testa)* //fuc2返回函数指针为void(*)()
{
    return testa;
}
```

- 通过函数指针调用类成员函数

  [类成员函数指针]

  - 类函数指针赋值要使用`&`，访问要用`*`解引用（与普通函数指针不同）
  - 使用`.*`(实例对象)或者`->*`（实例对象指针）调用**非静态**类成员函数指针所指向的函数（静态成员函数不用）
  - 取类成员函数地址时：**①记得加&；②记得加上范围限定符**
  - 声明指向非静态类成员函数的指针时，记得加范围限定符，静态的话则不用
  - 对类成员函数取址时：
    - 为**非静态成员函数**时，只能通过联合体进行取址，获得的是该函数在内存中的实际地址
    - 为**虚函数**时，只能通过联合体进行取址，其地址在编译时期是未知的，获得的是一个索引值

- 为**静态成员函数时**，可以直接访问其地址

- 普通函数指针调用类成员函数

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

template<typename src_type>
void* union_cast(src_type src)
{
    union {
        src_type src;
        void* dst;
    }u;
    u.src = src;
    return u.dst;
}

class A
{
public:
    void test(void) { }

    virtual void virtest() { }

    static void stattest() { }
};
void test() {}

int main() {
//  声明指向非静态类成员函数的指针时：①记得加范围限定符
    void(A::*pf)();      // 声明了一个void(A::*)()类型的变量
    pf = &A::test;
    pf = &A::virtest;

//  声明指向静态类成员函数的指针时：不用加范围限定符
    void(*statpf)();
    statpf = &A::stattest;  //初始化的时候和普通成员函数一样。

//  使用.*调用指针指向的函数
    A a;
    (a.*pf)();
    statpf();//静态成员函数可以直接访问

//  如果想访问类成员函数的地址，需要用到我的union_cast函数
    void* pf1 = union_cast(&A::test);   //输出为实际地址
    void* pf2 = union_cast(&A::virtest);//g++输出为0x1，msvc输出为实际地址

    return 1;
}

```

- 类成员函数指针调用类成员函数

  除了函数调用有区别，其余都一样

```cpp
#include <iostream>
using namespace std;

template<typename src_type>
void* union_cast(src_type src)
{
    union {
        src_type src;
        void* dst;
    }u;
    u.src = src;
    return u.dst;
}

class A
{
public:
    void (A::* p1)();   // 此时p1是个成员变量
    void (*p2)();

    void test() { }     // test是个函数地址
    virtual void virtest() { }
    static void stattest() { }

    A() {
        //取类成员函数地址时：①记得加&；②记得加上范围限定符
        p1 = &A::test;      // 将函数地址赋给成员变量
        p1 = &A::virtest;
        p2 = &A::stattest;
    }
};

int main() {
    A a;
//  p1为成员变量，需要用实例化对象进行访问，所以a.p1表示访问该成员变量（即函数地址）
//  由于a.p1指向的是非静态成员函数，所以要通过实例化对象用 .* 进行访问
    (a.*a.p1)();

//  静态成员函数不用传this，所以较为简单
    (*a.p2)();

//  如果想访问类成员函数的地址，需要用到我的union_cast函数
    void* pf1 = union_cast(a.p1);
    void* pf2 = union_cast(&A::virtest);

    return 1;
}
```

回调函数

普通函数的函数回调

```
#include <iostream>        // 包含头文件。
using namespace std;       // 指定缺省的命名空间。

void 个性化表白函数()
{
    // 个性化表白的代码。
}

void 表白神器(个性化表白函数指针p)
{
    // 表白之前的准备工作。

    个性化表白函数指针p();

    // 表白之后的收尾工作。
}

int main()
{
    表白神器(个性化表白函数名);
}
```

理解：

现在我要开发一个方便大伙表白的函数，这个函数名叫做表白神器，在这个函数里干三件事。1. 表白之前的场地布置等准备活动。2. 表白时的动作以及我对女生说的话。3. 表白之后的场地打扫等收尾活动。在这三件事情中，第1，3条都是可以确定的事情，只有第2件事情不确定，因为每个人的做法都不一样，这个时候就可以用到函数指针，那么用户在调用表白函数时只需要传入自己准备好的个性化表白函数指针，即可有一个完整的表白流程。这种函数调用的方法就叫函数的回调，其中个性化表白函数就叫回调函数。

对于回调函数需要注意两点：（在和普通函数调用对比之后，我注意到两点不同的地方）

1. 在表白神器中不能直接调用个性化表白函数，因为不同的用户有不同的个性化表白函数，我根本不知道这个表白神器具体被谁调用。
2. 采用函数指针传参的形式，使不同用户在调用同一个表白神器时，可以根据自己的情况调用不同的个性化表白函数，使函数整体的灵活性更强。
3. 很明显，通过以上2点，对于我的需求而言，我只能使用回调函数来实现

回调函数的传参方式：

1. 由调用者函数提供实参

```cpp
#include<iostream>
#include <string>

using namespace std;

void SayHello(int num, string str)
{
    cout << "Number: " << num << " say: " << str << endl;
}

void show(void (*pFunc)(int, string))
{
    cout << "准备sayhello" << endl;

    int num = 0;
    string str = "hello world"; //先在函数内部定义好pFunc的参数
    pFunc(num, str);            //此时, pFunc的参数由 调用pFunc的函数 传入

    cout << "说完了" << endl;

}
```

2. 由外界提供实参

```cpp
using namespace std;

void SayHello(int num, string str)
{
    cout << "Number: " << num << " say: " << str << endl;
}

void show(void (*pFunc)(int, string), int num, string str)
{
    cout << "准备sayhello" << endl;

    pFunc(num, str);            //此时, pFunc的参数由 调用pFunc的函数的外部 传入

    cout << "说完了" << endl;

}

int main()
{
    int num = 1;
    string str = "hello!";      //在show的外部定义好回调函数的参数

    show(SayHello, num, str);   //传参

    system("pause");
    return 0;

}
```

类成员函数的函数回调

关于C++ 回调函数(callback) 精简且实用c++*回调函数*zhoupian的博客-CSDN博客

静态函数

```cpp
#include<iostream>
#include <string>
#include <functional>

using namespace std;

class Son
{
public:
    static void SayHello();
};
void Son::SayHello()
{
    cout << "I'm son" << endl;
}

void DadFunc(void (*pcallback)());  //在DadFunc中采取函数的回调

int main()
{
    DadFunc(Son::SayHello);          //当回调函数为类的 静态成员函数 时，指明作用域即可

    system("pause");
    return 0;
}
```

非静态函数

std::function：

function<>的实例对象封装各种可调用的实体，包括函数指针，lambda表达式，类的成员函数，普通函数

C++11中的std::function_std::function 大小_大猪的博客-CSDN博客

封装类的非静态成员函数：

```cpp
class Son
{
public:
    void SayHello(string str)
    {
        cout << str << endl;
    }
};

//void DadFunc(void (*pcallback)(string str));  //在DadFunc中采取函数的回调

int main()
{
    Son son;

    /*
    在C++中，成员函数指针与成员函数关联，其中第一个参数始终是类的实例（即this指针），
    后面的参数才是函数的正常参数。因此如果想调用Son的普通成员函数，
    在定义时第一个参数一定要写上Son，这样我在调用Son的SayHello时，
    才能把*this给传进去
    */
    function<void(Son, string)> Callback;    //指明我要调用Son类中的某个特定函数；声明一个function的实例，用于调用特定对象的函数
    Callback = &Son::SayHello;               //指明该函数为SayHello；注意给function的实例赋值时，一定要加上&，因为此处为对类的成员函数进行封装，而非函数指针
    Callback(son, "hello");                  //指明具体对象以及参数；调用son的SayHello

    system("pause");
    return 0;
}
```

封装函数指针：

```cpp
class Son
{
public:
    void SayHello(string str)
    {
        cout << str << endl;
    }
};

//void DadFunc(void (*pcallback)(string str));  //在DadFunc中采取函数的回调

int main()
{
    Son son;

    void (Son::*pfunc)(string);             //调用类非静态成员函数时，要指明函数指针的作用域
    pfunc = &Son::SayHello;                 //函数指针在赋值的时候，右值一定要加上&表示取地址

    function<void(Son, string)> Callback;
    Callback = pfunc;                       //此时function<>类封装的是函数指针
    Callback(son, "hello");

    system("pause");
    return 0;
}
```

std::bind

bind用来绑定函数的某些参数的，在函数的回调之中，可以用来将 类的非静态成员函数 中的第一个参数与具体对象绑定，但注意对于参数列表中的剩余参数，必须显示指明或者用placeholders::_1 / _2 / _3……指定。他的返回值是一个可调用的实体，可以选择用function<>接受

C++11中的std::bind 大猪的博客-CSDN博客

使用bind配合function实现函数的回调（耦合度低）

```cpp
void DadFunc(function<void(string)> funcCallback);  //使用function<>来接受bind返回的可调用实体

int main()
{
    Son son;

    /*
    bind将SayHello与 son对象以及string给绑定起来了（这些都属于SayHello的参数），
    绑定之后这些参数的值是不可以改变的
    用placeholders::_1占位符的话就需要进行再次赋值
    */
    DadFunc(bind(&Son::SayHello, &son, "I'm son!"));

    system("pause");
    return 0;
}

void DadFunc(function<void(string)> funcCallback)
{
    cout << "this is dad" << endl;

    funcCallback("111");                        //如果在main中bind时使用placeholders::_1的话，显示的就是"111"

    cout << "dad is out" << endl;

}
```

使用function<>实现函数的回调（耦合度高）

```cpp
//如果不用bind的话，function的Son由外界提供实参，
//string由调用者函数提供实参（即在函数内部赋值）
void DadFunc(function<void(Son, string)> funcCallback, Son son);

int main()
{
    Son son;

    DadFunc(&Son::SayHello, son);

    system("pause");
    return 0;
}

void DadFunc(function<void(Son, string)> funcCallback, Son son)
{
    cout << "this is dad" << endl;

    funcCallback(son, "111");                    //显然这样不能完全把调用者函数和回调函数进行解耦

    cout << "dad is out" << endl;

}
```

使用追踪返回类型的函数

page164

追踪返回类型

我们先来看一个问题：如果一个函数模板的返回值类型需要依赖于入参的类型，应该怎么写这个模板函数？在上一篇中介绍了decltype的用法，也许可以这样写：

```cpp
template<typename T>
decltype(2 * a) doubleValue(T& a) { return 2 * a; }  // 用decltype推导返回类型
```

但是对于编译器来说，是从左到右进行编译的，decltype在进行推导时并不知道a的类型，所以这种写法是编译不过的。为了解决这个问题，于是引入了追踪返回类型：

```cpp
template<typename T>
auto doubleValue(T& a) -> decltype(2 * a)
{
    return 2 * a;
}
```

基于范围的for循环

C++ 基于范围的for循环

将该特性用于自定义数据结构上的条件：

- 对于用户自定义的类，能对此自定义数据结构类型调用 `begin` 和 `end` 方法，无论是成员函数或者独立函数都可以，要能返回迭代器类型。
- 返回的迭代器类型必须支持 `operator*` 方法，`operator!=` 方法和前缀形式的 `operator++` 方法，同样无论是成员函数或独立函数都可以。
- 先前置申明自定义类，然后实现迭代器类，再实现自定义类，最后实现迭代器类中的 `operator*` 方法

```cpp
// a simple iterator sample.

#include <iostream>
```

```
4
5  using namespace std;
6
7  // forward-declaration to allow use in Iter
8  class IntVector;
9
10 class Iter
11 {
12 private:
13     int _pos;
14     const IntVector *_p_vec;
15
16 public:
17     Iter(const IntVector* p_vec, int pos) : _pos(pos), _p_vec(p_vec){}
18
19     // these three methods form the basis of an iterator for use with a
   range-based for loop
20     bool operator!=(const Iter& other) const
21     {
22         return _pos != other._pos;
23     }
24
25     // this method must be defined after the definition of IntVector since
   it needs to use it
26     int operator*() const;
27     const Iter& operator++()
28     {
29         ++_pos;
30         // although not strictly necessary for a range-based for loop
31         // following the normal convention of returning a value from
32         // operator++ is a good idea.
33         return *this;
34     }
35 };
36
37 class IntVector
38 {
39 private:
40     int _data[100];
41
42 public:
43     IntVector(){}
44
45     int get(int col) const
46     {
47         return _data[col];
48     }
49
50     Iter begin() const
51     {
52         return Iter(this, 0);
53     }
54
55     Iter end() const
56     {
57         return Iter(this, 100);
```

```
    }

    void set(int index, int val)
    {
        _data[index] = val;
    }
};

int Iter::operator*() const
{
    return _p_vec->get(_pos);
}

// sample usage of the range-based for loop on IntVector
int main()
{
    IntVector v;

    for(int i = 0; i < 100; i++)
    {
        v.set(i,i);
    }
    for( int i : v)
    {
        cout << i << endl;
    }
}
```

强类型枚举

C enum(枚举) 基础

《深入理解C++11》强类型枚举

> - **强作用域**，必须定义枚举类型才能使用。
> - **转换限制**，不能隐式转换为整形。
> - **底层类型**，默认底层类型为int，能够显式指定底层类型（必须是整形）。
> - **属于POD类型。**

- 语法：

```
enum class Type : char{              // 定义枚举类型并指定底层类型
    Low,
    Medium,
    High
}type;                  // 定义枚举变量
```

- 例子：

```
enum class Level {
    One,Two,Three,Four,Five
};

enum class Type {
```

```
    Low,Medium,High
};

class Password {
public:
    Password(Level l, Type t) :level(l), type(t) {}
    Level level;
    Type type;
};

int main()
{
    Password pwd(One, Low);          // 编译失败，必须使用枚举类型名字
    Password pwd(Level::One, Type::Low);

    if (pwd.level < Type::Medium){}    // 编译失败，必须使用同一枚举类型进行比较

    if (pwd.level < Level::Three){}

    if (pwd.type > 1){}                // 编译失败，无法隐式转换为整形

    if ((int)pwd.type > 1){}

    return 0;
}
```

指定基本类型之后，各个编译器之间的实现就会相同，便于代码移植

- C++11对原有枚举类型的拓展（了解即可）
  - 原有枚举类型的底层类型默认时，仍由编译器来具体指定实现，但也可以显式指定

```
enum Type : char
{
    c1, c2, c3, c4
};
```

  - 枚举成员的名字除了自动输出到父作用域，也可以再枚举类型定义的作用域内有效

```
enum Type { c1, c2, c3};
Type t1 = c1;
Type t2 = Type::c2;     // 两者都是合法的使用形式
```

智能指针

原理：RAII：利用局部对象自动销毁的特性管理堆内存中资源释放

[c++经验之谈一：RAII原理介绍 - 知乎 (zhihu.com)]

（用于在避免悬空指针，内存泄漏，属于模板类的应用）

智能指针介绍

1. auto_ptr

智能指针 `auto_=tr` 由C++98引入，定义在头文件 `<memory>` 中，在C++11中已经被弃用了，因为它不够安全，而且可以被 `unique_ptr` 代替。那它为什么会被 `unique_ptr` 代替呢？先看下面这段代码：

```cpp
#include <iostream>
#include <string>
#include <memory>
using namespace std;

int main() {
    auto_ptr<string> p1(new string("hello world."));
    auto_ptr<string> p2;
    p2 = p1;                //p2接管p1的所有权
    cout << *p2<< endl;     //正确，输出：hello world.
    //cout << *p1 << endl;  //程序运行到这里会报错

    //system("pause");
    return 0;
}
```

C++

可见，`auto_ptr` 智能指针并不够安全，于是有了它的替代方案：即 `unique_ptr` 指针。

按理来说当p2=p1时，所有权已经转移到p2身上了，但我在cout<<*p1;时，程序不会报错，只有在运行时才会报错，所以auto_ptr被弃用

2. unique_ptr

`unique_ptr` 同 `auto_ptr` 一样也是采用所有权模式，即同一时间只能有一个智能指针可以指向这个对象，但之所以说**使用 `unique_ptr` 智能指针更加安全**，是因为它相比于 `auto_ptr` 而言**禁止了拷贝操作**，`unique_ptr` 采用了移动赋值 `std::move()` 函数来进行控制权的转移。

即用unique_ptr不允许两者之间的赋值操作，只允许通过move()转移所有权

auto_ptr 与 unique_ptr智能指针的内存管理陷阱

```cpp
auto_ptr<string> p1;
string *str = new string("智能指针的内存管理陷阱");
p1.reset(str);  // p1托管str指针
{
    auto_ptr<string> p2;
    p2.reset(str);  // p2接管str指针时，会先取消p1的托管，然后再对str的托管
}

// 此时p1已经没有托管内容指针了，为NULL，在使用它就会内存报错!
cout << "str: " << *p1 << endl;
```

[shared_ptr在多线程下的安全性问题]

[智能指针的原理及实现]

[万字长文全面详解现代C++智能指针：原理、应用和陷阱 - 知乎 (zhihu.com)]

智能指针实现

引用计数变化的几种情况

1. 新创建对象时 = 1
2. 拷贝赋值时 +
3. 拷贝构造时 +
4. weak_ptr的lock()函数 +

5. 指针离开作用域时 -（引用计数为0时，释放已分配的堆内存）

```cpp
/* ---------- shared point ---------- */
template<typename T>
class mshared_ptr {
private:
    int* ref_count;
    T* data_ptr;

public:
    // 无参构造函数
    mshared_ptr() {
        this->data_ptr = nullptr;
        ref_count = nullptr;
    }
    // 有参构造函数
    mshared_ptr(T* data_ptr) {
        this->data_ptr = data_ptr;
        ref_count = new int(1);
    }

    ~mshared_ptr() {
        --(*ref_count);
        if (*ref_count == 0)
        {
            delete data_ptr;
            delete ref_count;
        }
    }

    // 拷贝构造，不用像operator=一样考虑那么多，因为这个是构造函数，本来就没值
    mshared_ptr(mshared_ptr<T>& msh_ptr) {
        data_ptr = msh_ptr.data_ptr;
        ref_count = msh_ptr.ref_count;
        ++(*ref_count);
    }

    // operate=重载，此处需要防止自我复制，且原引用计数要--
    T& operator= (mshared_ptr<T>& msh_ptr) {
    if(this==&other)
        return *this;

    //新指针引用计数要++
    ++*other._refCount;

    //原指针引用计数要--，如果为0，则释放空间
    if (--*_refCount == 0) {
        delete _ptr;
        delete _refCount;
    }

    //重新进行指向
    _ptr = other._ptr;
    _refCount = other._refCount;
    return *this;
```

```
    }

    T* operator-> () {
        return data_ptr;
    }

    T& operator* () {
        return (*data_ptr);
    }
};
```

```
/* ---------- weak point ---------- */
template<typename T>
class WeakPtr
{
public:
    WeakPtr() {};

    WeakPtr(const SharedPtr<T> &p) : ptr(p.get())
    {}

    ~WeakPtr()
    {}

    WeakPtr<T>& operator=(const WeakPtr &p)
    {
        ptr = p.ptr;
        return *this;
    }

    // weak_ptr中禁止重载*和->运算符，只能通过lock()获取对应的shared_ptr实例对data_ptr进行访问
    /*T& operator*(){ return *ptr; }
    T* operator->(){ return ptr; }*/

    operator bool()
    {
        return ptr != nullptr;
    }

private:
    // weak point 只引用，不计数
    T *ptr;

};
```

shared_ptr中的循环引用

[智能指针中的循环引用与weak_ptr的应用]

```
class B;
class A {
public:
    // weak_ptr<B> pb;
```

```
    shared_ptr<B> pb;
    ~A() { cout << "destructor A func" << endl; }
};
class B {
public:
    // weak_ptr<A> pa;
    shared_ptr<A> pa;
    ~B() { cout << "destructor B func" << endl; }
};
int main() {
    shared_ptr<A> a(new A());
    shared_ptr<B> b(new B());
    a->pb = b;
    b->pa = a;
    cout << "a use count:" << a.use_count() << endl;
    return 0;
}
// 运行结果:
// a use count:2
```

class A和class B的对象各自被两个智能指针管理，此时的内存布局是一共有4块内存，其中一块属于A，一块属于B，每块内存都被两个指针指向。

在这种情况下，在main函数结束的时候，根据类的析构顺序，智能指针a和b的析构函数先被调用，对应的两个引用计数同时-1，但根据代码实现和栈/堆内存的释放知识可知，在智能指针的析构函数被调用后，只会释放其成员的栈空间（对于智能指针来说是ref_count和data_ptr的地址）。对于堆内存，需要手动调用delete进行释放，但问题是只有引用计数为0时才会调用delete释放，此时引用计数为1，不会进行释放，而在a，b的析构函数调用完后，程序便结束了，于是留下了4块内存。

解决方法很简单，把class A或者class B中的shared_ptr改成weak_ptr即可，由于weak_ptr不会增加shared_ptr的引用计数，所以A object和B object中有一个的引用计数为1，在pa和pb析构时，会正确地释放掉内存

4.weak_ptr

`weak_ptr` 弱指针是一种不控制对象生命周期的智能指针，它指向一个 `share_ptr` 管理的对象，进行该对象的内存管理的是那个强引用的 `share_ptr` ，也就是说 `weak_ptr` 不会修改引用计数，只是提供了一种访问其管理对象的手段，这也是它称为弱指针的原因所在。

此外，`weak_ptr` 和 `share_ptr` 之间可以相互转化，`share_ptr` 可以直接赋值给 `weak_ptr` ，而 `weak_ptr` 可以通过调用 `lock` 成员函数来获得 `share_ptr` 。

weak_ptr属于弱指针，相对的，shared_ptr属于强指针

既然weak_ptr并不改变其所共享的shared_ptr实例的引用计数，那就可能存在weak_ptr指向的对象被释放掉这种情况。这时，就不能使用weak_ptr直接访问对象。那么如何判断weak_ptr指向对象是否存在呢？C++中提供了lock函数来实现该功能。**如果对象存在，lock()函数返回一个指向共享对象的shared_ptr(引用计数会增1)，否则返回一个空shared_ptr。weak_ptr还提供了expired()函数来判断所指对象是否已经被销毁。**

由于weak_ptr并没有重载operator ->和operator *操作符，因此不可直接通过weak_ptr使用对象，同时也没有提供get函数直接获取裸指针。典型的用法是调用其lock函数来获得shared_ptr示例，进而访问原始对象。

shared_ptr的线程安全问题

> - shared_ptr的引用计数读写都是线程安全的（都是原子操作）
> - 修改shared_ptr指向是线程不安全的（看例1）
> - 对shared_ptr中的data_ptr进行操作是线程不安全的（看例2）

例1： C++ 智能指针线程安全的问题-CSDN博客
此处调用了`operator=`的重载，通过上面的实现可知，原引用计数是需要--的，如果线程A在拷贝构造到一半时轮到线程B，线程B此时调用`operator=`，执行完后转回线程A，此时线程A中的data_ptr对应的值可能已经被释放了

例2： 线程A和线程B访问一个共享的对象，如果线程B在调用该共享对象的成员方法时时间片结束而轮到线程A，线程A析构完该对象后时间片结束轮到线程B，此时线程B再去访问该对象，就会发生错误。此可以通过shared_ptr和weak_ptr来解决共享对象的线程安全问题。

```
#include <iostream>
#include <memory>
#include <thread>

class Test {
  public:
    Test(int id) : m_id(id) {}
    void showID() {
      std::cout << m_id << std::endl;
    }
  private:
    int m_id;
};

// 传入shared_ptr参数时，会隐式转换成weak_ptr
void thread2(std::weak_ptr<Test> t) {
  std::this_thread::sleep_for(std::chrono::seconds(2));
  std::shared_ptr<Test> sp = t.lock();  //通过lock()获取对应的shared_ptr，非nulptr时表示未释放
  if(sp)
    sp->showID();                       // 打印结果：2
}

int main()
{
  std::shared_ptr<Test> sp = std::make_shared<Test>(2);
  std::thread t2(thread2, sp);
  t2.join();

  return 0;
}

```

如果想访问对象的方法，先通过t的**lock方法进行提升操作，把weak_ptr提升为shared_ptr强智能指针。**提升过程中，是通过检测它所观察的强智能指针保存的Test对象的引用计数，来判定Test对象是否存活。**ps如果为nullptr，说明Test对象已经析构，不能再访问；如果ps!=nullptr，则可以正常访问Test对象的方法。**

如果设置t2为分离线程t2.detach()，让main主线程结束，sp智能指针析构，进而把Test对象析构，此时showID方法已经不会被调用，因为在thread2方法中，t提升到sp时，lock方法判定Test对象已经析构，提升失败！

补充：智能指针在观察者模式中的应用

在多数实现中，观察者通常都在另一个独立的线程中，这就涉及到在多线程环境中，共享对象的线程安全问题(解决方法就是使用上文的智能指针)。这是因为在找到监听者并让它处理事件时，其实在多线程环境中，肯定不明确此时监听者对象是否还存活，或是已经在其它线程中被析构了，此时再去通知这样的监听者，肯定是有问题的。

也就是说，**当观察者运行在独立的线程中时，在通知监听者处理该事件时，应该先判断监听者对象是否存活，如果监听者对象已经析构，那么不用通知，并且需要从map表中删除这样的监听者对象。**其中的主要代码为：

```
// 存储监听者注册的感兴趣的事件
unordered_map<int, list<weak_ptr<Listener>>> listenerMap;

// 观察者观察到事件发生，转发到对该事件感兴趣的监听者
void dispatchMessage(int msgid) {
  auto it = listenerMap.find(msgid);
  if (it != listenerMap.end()) {
    for (auto it1 = it->second.begin(); it1 != it->second.end(); ++it1) {
      shared_ptr<Listener> ps = it1->lock();            // 智能指针的提升操作，用来判断监听者对象是否存活
      if (ps != nullptr) {                              // 监听者对象如果存活，才通知处理事件
        ps->handleMessage(msgid);
      } else {
        it1 = it->second.erase(it1);                    // 监听者对象已经析构，从map中删除这样的监听者对象
      }
    }
  }
}
```

weak_ptr

应用场景

一切应该不具有对象所有权，又想安全访问对象的情况。

lock()实现

C++内存管理：shared_ptr/weak_ptr源码（长文预警） - 知乎 (zhihu.com)

该函数线程安全，在该函数中通过while循环的自旋锁不断判断引用计数是否为0（为0表示对应的shared_ptr中的data_ptr已经被释放），然后还要在lock前判断是否有别的线程更改了这个引用计数，在没有更改的时候+1。从判断条件到引用计数+1这段代码通过CAS实现原子操作。

深入理解 C++ weak_ptr | 编程指北 (csguide.cn)

C++11中的智能指针shared_ptr、weak_ptr源码解析 - tomato-haha - 博客园 (cnblogs.com)

enable_shared_from_this？？？

常量表达式？？？

[《深入理解C++11》笔记-常量表达式 c++11常量-CSDN博客]

[编译期常量？？？]

[字面值常量类（常量构造函数）]

> - const描述的是"运行时常量"（修饰普通变量时是编译时常量），constexpr描述的是"编译时常量"
> - 对于constexpr修饰的变量而言，如果没有被调用，则编译器可以不为其生成数据，而仅将其当作**编译期常量**
> - 尽量不要用编译期常量，如果要用，则编译期的浮点常量表达式精度>=运行期浮点常量表达式精度
> - constexpr修饰自定义类型声明时，需要先定义**常量构造函数**
> - constexpr修饰的函数也可以用于非常量表达式中的类型构造。因为如果表达式不满足常量性，那么constexpr关键字将会被忽略。

- constexpr修饰的普通函数
  - 定义：
    - 函数只能包含return语句。
    - 函数必须有返回值。
    - 在使用前必须已经定义。
    - return返回语句中不能使用非常量表达式的函数、全局数据，且必须是一个常量表达式。
  - 例子：

```
const int g2 = 5;
constexpr int func2(int g){
    return g;
}
int main(){
    int a[func2(g2)];// 编译通过
}
```

- constexpr修饰的表达式值

```
const int i = 1;
constexpr int j = 1;
```

  i作为全局变量时，编译器一定为i产生数据;

  对于j而言，如果没有被显式调用，则编译器可以不为其生成数据，而仅将其当作**编译期常量**

  编译期常量：枚举值，右值，constexpr/const修饰的变量。

- constexpr修饰的构造函数

  默认只有内置类型才能修饰为常量表达式值，自定义类型如果要成为常量表达式值，必须定义**常量构造函数**

- 定义：
  - 函数体必须为空。
  - 初始化列表只能由常量表达式来赋值。
- 例子：

```
class Example {
public:
    Example() {}
    constexpr Example(int i) : data(i) {}// 常量构造函数
private:
    int data;
};
int main(){
//  constexpr Example ex;           编译失败
    constexpr Example ex1(1);   // 编译通过
}
```

- constexpr的应用
  - 在模板函数中的应用

    如果模板函数被实例化后不满足常量性，那么constexpr关键字将会被忽略。

    constexpr不可以作用于虚函数，因为虚函数是运行时的行为

```
class Example{
private:
    int data;
};

template<typename T>
constexpr T func(T t)
{
    return t;
}

int main()
{
    constexpr int i = func(1);      // 通过编译，模板函数实例化后满足常量性

    Example ex;
    Example ex1 = func(ex);
    constexpr Example ex2 = func(ex);  // 无法通过编译，因为Example没有常量性，模板的constexpr被忽略

    return 0;
}
```

  - 把计算步骤提前到编译阶段

```
constexpr size_t c_fib(size_t n) noexcept
{
    return n == 0 ? 0 : (n == 1 ? 1 : c_fib(n - 1) + c_fib(n - 2));
}
```

```cpp

int main(void)
{
    constexpr size_t n = 25;
    clock_t startTime, endTime;

    startTime = clock();
    constexpr size_t r2 = c_fib(n);
    endTime = clock();
    std::cout << r2 << ", c_fib: " << endTime - startTime << "ms" << std::endl;                // 1ms

    return 0;
}
```

C语言中的变长参数

[视频-C语言变长实参]

[C语言变长参数函数原理-CSDN博客]

[va_list、va_start和va_end使用 - 知乎 (zhihu.com)]

> - args每次加一时要么加4字节，要么加8字节
> - 在c++的va_start中，会对固参进行类型推导，推导出的类型不能为

- stdarg.h头文件内变参宏定义如下：

```cpp
typedef char * va_list;

// 把 n 圆整 到 sizeof(int) 的倍数，
#define _INTSIZEOF(n)    ( (sizeof(n)+sizeof(int)-1) & ~(sizeof(int)-1) )

// 初始化 ap 指针，使其指向第一个可变参数（在c++中还会对v进行类型推导并进行编译阶段检查，不能为引用）
#define va_start(ap,v)    ( ap = (va_list)&v + _INTSIZEOF(v) )

// 该宏返回当前变参值,并使 ap 指向列表中的下个变参
#define va_arg(ap, type)    ( *(type *)((ap += _INTSIZEOF(type)) - _INTSIZEOF(type)) )

// /将指针 ap 置为无效，结束变参的获取
#define va_end(ap)    ( ap = (va_list)0 )
```

  - **_INTSIZEOF(n)**

    _INTSIZEOF 宏考虑到某些系统需要内存地址对齐。从宏名看应按照 sizeof(int) 即栈粒度对齐，参数在内存中的地址均为 sizeof(int) = 4 的倍数。

    例如，**若 1≤sizeof(n)≤4，则 _INTSIZEOF(n) = 4；若 5≤sizeof(n)≤8，则_INTSIZEOF(n)=8。**

  - **va_start(ap, v)**

    va_start 宏首先获取固定参数在栈中的内存地址，加上该固参所占内存大小后赋值给ap，使 ap 指向固定参数的下一个参数，此时**ap 指向第0个变参。**

  - **va_arg(ap, type)**

这个宏取得 type 类型的可变参数值。首先 ap += _INTSIZEOF(type)，即 ap 跳过当前可变参数而指向下个变参的地址；然后 ap - _INTSIZEOF(type) 得到当前变参的内存地址，类型转换后解引用，最后返回当前变参值，结果是**va_arg返回第n个变参，ap指向第n+1个变参。**

- **va_end(ap)**

  va_end 宏使 ap 不再指向有效的内存地址。该宏的某些实现定义为 ((void*)0)，编译时不会为其产生代码，调用与否并无区别。但某些实现中 va_end 宏用于在函数返回前完成一些必要的清理工作：如 va_start 宏可能以某种方式修改栈，导致返回操作无法完成，va_end 宏可将有关修改复原；又如 va_start 宏可能为参数列表动态分配内存以便于遍历，va_end 宏可释放此内存。因此，从使用 va_start 宏的函数中退出之前，必须调用一次 va_end 宏。

- 使用示例：

```
#include<stdarg.h>
void m_printf1(int cnt, ...){
//  args其实就是char*
    va_list args;
//  初始化args，使args指向首个可变参数（注意args加的不是sizeof(Format)）
    va_start(args, cnt);
    for (int i = 0; i < cnt; i++)
    {
//      va_arg返回的是第n个可变参数，但args指向的是第n+1个可变参数
        cout << va_arg(args, int) << ' ';// 注意下面用的是vfprintf
    }
//  将args置为无效的内存地址
    va_end(args);
}

int m_printf2(const char* const Format, ...) {
    int res;
    va_list args;
    va_start(args, Format);
//  将Format和args组合输出进stdout里
    res = vfprintf(stdout, Format, args);
    va_end(args);
    return res;
}
```

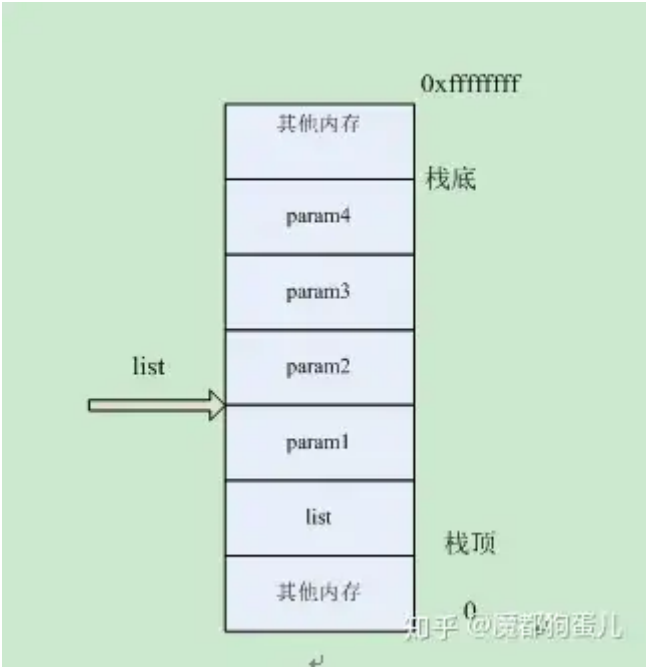
- 原理：

为了理解这些宏的作用，我们必须先搞清楚：C语言中函数参数的内存布局。首先，**函数参数是存储在栈中的，函数参数从右往左依次入栈。**

```
void test(char *para1, char *param2, char *param3, char *param4) {
       va_list list;
       ......
       return;
 }
```

栈由高地址往低地址生长,调用test函数时，其参数入栈情况如下：

当调用va_start(list,param1) 时：list指针指向情况对应下图：



- 手动实现printf

```
#include "stdio.h"
#include "stdlib.h"
void myprintf(char* fmt, ...)         //一个简单的类似于printf的实现，//参数必须都是int 类型
{
    char* pArg=NULL;                 //等价于原来的va_list
    char c;

    pArg = (char*) &fmt;            //注意不要写成p = fmt !!因为这里要对//参数取址，而不是取值
    pArg += sizeof(fmt);           //等价于原来的va_start

    do
    {
        c =*fmt;
        if (c != '%')
        {
            putchar(c);             //照原样输出字符
```

```
        }
        else
        {
           //按格式字符输出数据
           switch(*++fmt)
           {
            case'd':
                printf("%d",*((int*)pArg));
                break;
            case'x':
                printf("%#x",*((int*)pArg));
                break;
            default:
                break;
            }
            pArg += sizeof(int);                //等价于原来的va_arg
        }
        ++fmt;
    }while (*fmt != '/0');
    pArg = NULL;                                //等价于va_end
    return;
}
int main(int argc, char* argv[])
{
    int i = 1234;
    int j = 5678;

    myprintf("the first test:i=%d/n",i,j);
    myprintf("the secend test:i=%d; %x;j=%d;/n",i,0xabcd,j);
    system("pause");
    return 0;
}
```

模板的特化？？？

C++模板的特化和偏特化_模板偏特化-CSDN博客

C++中的变长模板？？？

page201

《深入理解C++11》笔记-变长模板_变长模板类-CSDN博客

C++ 变长模板 template - 明明1109 - 博客园 (cnblogs.com)

使用C++11变长参数模板 处理任意长度、类型之参数实例_c++ 变长模板参数-CSDN博客

原子类型与原子操作？？？

page214

线程局部存储？？？

page232

快速退出？？？

page234

指针空值nullptr？？？

page238

> - `nullptr`和`(void*)0`的区别
>
> `nullptr`是一个**编译期常量**，被`#define`定义为0，它的名字是一个编译时期的关键字，可以被编译器识别，**nullptr到任何指针的转换都是隐式的**
>
> `(void*)0`是一个**强制转换表达式**，返回值也是一个`(void*)`指针类型，该类型必须经过**类型转换后才可以进行赋值操作**

默认函数控制？？？

page245

《深入理解C++11》笔记-默认函数的控制

lambda 对类的捕获（this捕获及局部成员捕获）

iniline？？？

【C++】C++中内联函数详解（搞清内联的本质及用法）-CSDN博客

lambda函数

page252

> - lambda等价于`ret-type operator()(param-list) const{}`的仿函数，mutable可以取消const
> - lambda默认内联（类中定义的函数也是默认内联的）
> - lambda函数可以由auto进行推断，推断结果为对应函数类型
> - 块作用域中的lambda函数仅能捕捉父作用域中的自动变量，捕捉任何非此作用域或者是非自动变量（如静态变量等） 都会导致编译器报错（尤其是在类中，**只能通过捕获this指针来访问类的成员变量**）。
> - 按引用捕获局部变量时，注意局部变量的生存周期

基本用法

- 语法

```
[capture](parameters) mutable -> return-type{statement}
```

  - `[capture]`，捕捉列表。捕捉列表总是在lambda函数的开始，[]是lambda函数的引出符。编译器根据该引出符确定接下来的代码是不是lambda函数。捕捉列表能够捕捉上下文中的变量以供lambda函数使用。下文会介绍捕捉的方法。

- `(parameters)`，参数列表。和普通函数的参数列表一致，如果不需要参数可以连()一起省略。
- `mutable`，一个修饰符。默认情况下，lambda函数总是一个const函数，mutable可以取消其常量性。在使用mutable时，参数列表不可省略，即使参数为空。
- `->return-type`，返回类型。用追踪返回类型声明函数的返回值。不需要返回值时，可以连同符号->一起省略。另外，在返回类型明确的情况下，也可以省略，让编译器推导出返回类型。
- `statement`，函数体。与普通函数一样，但是可以使用捕捉列表中的变量。

- 各种形式

  在lambda函数定义中，参数列表和返回类型都是可选的部分，而捕捉列表和函数体有可能为空，极端情况下最简单的lambda函数就是`[]{};`，当然它什么都没有做。下面罗列一下lambda函数的各种形式：

```
int main(){
    int a = 1, b = 2;

    [] {};                                  // 最简单
    [=] {return a + b; };                   // 省略了参数列表和返回值类型
    auto func1 = [&](int c) {b = a + c; };    // 省略了返回值类型
    auto func2 = [=, &b](int c)->int { return b += a + c; };  // 完整的lambda函数
}
```

- 捕捉列表
  - [var]表示以值传递的方式来捕捉变量var。
  - [=]表示以值传递的方式捕捉所有父作用域的变量，包括this指针。
  - [&var]表示引用传递捕捉变量var。
  - [&]表示引用传递捕捉所有父作用域的变量，包括this指针。
  - [this]表示值传递方式捕捉当前的this指针。
  - 以上的方式可以组合使用，但是不允许对同一个变量以同一方式重复捕捉。

```
[=,&a,&b];  // 按引用捕获a，b，按值捕获其他变量
[&,a,this]; // 注意a不能是成员变量，只能是局部变量。
[=,a];      // 编译报错，重复捕获
```

  - 在块作用域(可以理解为在{}以内的代码)以外的lambda函数捕捉列表必须为空，这种lambda和普通函数除了语法上不同以外，和普通函数差别不大。在块作用域内的lambda函数只能捕捉父作用域中的自动变量，不能捕捉非此作用域的变量或者非自动变量(如静态变量等)。

lambda的基础使用

lambda相当于局部函数，可以就地使用，也可以通过变量来接受

```
// auto被推断为函数类型，x为函数名
auto x = [=](){return 1+1;};
x();    // 使用该lambda函数

// auto被推断为int（lambda的返回值类型）
auto x1 = [=](){return 1+1;}(); // lambda函数就地使用
```