const

作用

1. **修饰变量**：说明该变量不可以被改变。
   - 例：`const int a = 10;` 表示 `a` 是一个常量整型，其值不能被修改。
2. **修饰指针**：分为指向常量的指针（pointer to const）和自身是常量的指针（常量指针，const pointer）。
   - 指向常量的指针：`const int* p = &a;` 表示 `p` 是一个指针，指向一个常量整型，`p` 可以改变指向，但不能通过 `p` 修改其指向的值。
   - 常量指针：`int* const p = &a;` 表示 `p` 是一个常量指针，指向整型，`p` 的指向不能改变，但可以通过 `p` 修改其指向的值。
3. **修饰引用**：指向常量的引用（reference to const），用于形参类型，即避免了拷贝，又避免了函数对值的修改。
   - 例：`void func(const int &ref) {...}` 表示 `ref` 是对常量的引用，函数 `func` 不能通过 `ref` 修改其指向的值。
4. **修饰成员函数**：说明该成员函数内不能修改成员变量。
   - 例：`void MyClass::func() const {...}` 表示这个成员函数不会修改任何成员变量。

const 的指针与引用

- **指针**
  - **指向常量的指针（pointer to const）**：`const int* ptr;`
  - **自身是常量的指针（常量指针，const pointer）**：`int* const ptr;`
- **引用**
  - **指向常量的引用（reference to const）**：`const int& ref;`
  - **没有 const reference**：因为引用只是对象的别名，引用不是对象，不能用 const 修饰。

宏定义 #define 和 const 常量

特性	宏定义 #define	const 常量
定义方式	文本替换	常量声明
处理时机	预处理器处理	编译器处理
类型安全检查	无	有
内存分配	不分配	分配
存储位置	代码段	数据段
可取消性	可通过 `#undef` 取消	不可取消
例子	`#define PI 3.14`	`const double Pi = 3.14;`

static

作用

1. **修饰普通变量**
   - 修改变量的存储区域和生命周期，使变量存储在静态区。
   - 在 `main` 函数运行前分配空间，如果有初始值则用初始值初始化，否则用默认值初始化。
   - 例子：

```
static int count = 10; // 静态变量
```

2. **修饰普通函数**
   - 表明函数的作用范围，仅在定义该函数的文件内可用。
   - 用于多人开发项目中防止命名冲突。
   - 例子：

```
static void helperFunction() { /* ... */ } // 静态函数
```

3. **修饰成员变量**
   - 使所有对象共享同一个变量，且不需要实例化对象即可访问该成员。
   - 例子：

```
class MyClass {
public:
    static int sharedVar; // 静态成员变量
};
```

4. **修饰成员函数**
   - 使得该成员函数可以在不实例化对象的情况下被访问。
   - 静态成员函数内不能访问类的非静态成员。
   - 例子：

```
class MyClass {
public:
    static void staticFunction() { /* ... */ } // 静态成员函数
};
```

this 指针

概念

- `this` 是一个隐含于每个非静态成员函数中的特殊指针。
- 它指向调用该成员函数的对象。

工作机制

- 当对一个对象调用成员函数时，编译器将对象的地址赋给 `this` 指针，然后调用成员函数。
- 每次成员函数存取数据成员时，都隐式使用 `this` 指针。
- 在成员函数被调用时，`this` 作为一个隐含参数自动传递，是指向成员函数所属对象的指针。

this 指针的类型

- 在非 const 成员函数中声明为：`ClassName *const this`。
  - 这意味着不能给 `this` 指针赋值。
- 在 const 成员函数中声明为：`const ClassName* const`。
  - 表明不能修改 `this` 指针指向的对象的数据成员。

this 作为右值

- `this` 是个右值，所以不能取得 `this` 的地址（即不能使用 `&this`）。

使用场景

1. **实现对象的链式引用**
   - 通过返回 `*this`，可以实现方法链式调用。
   - 例子：

```
class MyClass {
public:
    MyClass& setX(int x) {
        this->x = x;
        return *this;
    }
private:
    int x;
};
```

2. **避免对同一对象进行赋值操作**
   - 在赋值操作符中检查自赋值。
   - 例子：

```
MyClass& MyClass::operator=(const MyClass& other) {
    if (this != &other) {
        // 处理赋值
    }
    return *this;
}
```

3. **在实现数据结构时**
   - 如在链表、树等数据结构中的节点操作。
   - 例子：

```
class ListNode {
public:
    ListNode* getNext() const {
        return this->next;
    }
private:
    ListNode* next;
};
```

inline 内联函数

特征

- 相当于把内联函数的内容写在调用内联函数的地方。
- 相当于不用执行进入函数的步骤，直接执行函数体。
- 类似于宏，但比宏**多了类型检查，具有真正的函数特性。**
- 编译器一般不会内联包含循环、递归、switch 等复杂操作的函数。
- 在类声明中定义的函数（除了虚函数）都会自动隐式地当作内联函数。

使用

内联函数的声明与定义

```
// 声明1（加 inline，建议使用）
inline int functionName(int first, int second,...);

// 声明2（不加 inline）
int functionName(int first, int second,...);

// 定义
inline int functionName(int first, int second,...) {
    // 函数体
};

// 类内定义，隐式内联
class A {
    int doA() { return 0; }  // 隐式内联
};

// 类外定义，需要显式内联
class A {
    int doA();
};

inline int A::doA() { return 0; }  // 需要显式内联
```

编译器对 inline 函数的处理步骤

1. 将 inline 函数体复制到调用点处。
2. 为所有 inline 函数中的局部变量分配内存空间。
3. 将 inline 函数的输入参数和返回值映射到调用方法的局部变量空间中。
4. 如果 inline 函数有多个返回点，将其转换为代码块末尾的分支（使用 GOTO）。

优缺点

优点

- 提高程序运行速度，避免了函数调用的开销。
- 相比宏，进行安全检查和自动类型转换。
- 类成员函数自动转换为内联，可以访问类的成员变量。
- 可以进行运行时调试。

缺点

- 代码膨胀，可能增加内存消耗。
- inline 函数改变需要重新编译。
- 是否内联由编译器决定，程序员无法完全控制。

虚函数（virtual）与内联（inline）

- 虚函数可以被声明为内联，但当它表现出多态性时，在运行时通常不能被内联。
- 内联建议在编译期间进行，而虚函数的多态性决定于运行期。
- inline virtual 只能在编译器确切知道调用的对象类型时内联（如直接调用 `Base::who()`）。

```
class Base {
public:
    virtual void who() {
        cout << "I am Base\n";
    }
};

class Derived : public Base {
public:
    void who() override {  // 覆盖基类的虚函数
        cout << "I am Derived\n";
    }
};

int main() {
    Base b;
    Derived d;

    // 直接调用
    b.who();  // 输出 "I am Base"
    d.who();  // 输出 "I am Derived"

    // 多态调用
    Base* ptr = new Derived();
    ptr->who();  // 输出 "I am Derived"
    delete ptr;

    return 0;
}
```

volatile

概念

`volatile` 关键字是一种类型修饰符，用于声明变量。它表示变量的值可能会被一些编译器未知的因素所更改，这些因素包括操作系统、硬件或其他线程等。

用途

- **防止编译器优化**：使用 `volatile` 告诉编译器，它不应该对这种对象进行优化。
- **强制每次从内存读取**：声明为 `volatile` 的变量在每次访问时都必须从内存中直接读取。对于没有被 `volatile` 修饰的变量，编译器可能会出于优化的目的，从 CPU 寄存器中读取其值。

例子

```
volatile int i = 10; // 声明一个 volatile 整型变量
```

在这个例子中，变量 `i` 被声明为 `volatile`。这意味着编译器在每次需要读取 `i` 的值时，都会直接从内存中读取，而不会使用可能存储在寄存器中的旧值。

结合 const 使用

- `const` 可以和 `volatile` 一起使用。这通常用于表示某个对象是只读的，但它的值可能会被外部因素更改。
- 例如，一个只读的状态寄存器可以声明为 `const volatile`。

volatile 指针

- 指针本身可以是 `volatile`，表示指针指向的地址可能会被外部因素更改。
- 也可以指向一个 `volatile` 类型的数据，表示通过该指针访问的数据可能会被外部因素更改。

例子

```
volatile int* p = &i; // 指向 volatile 整型的指针
```

在这个例子中，`p` 是一个指针，指向 `volatile int` 类型的数据。这意味着通过 `p` 对 `i` 的任何读取都将直接从内存中进行，以确保获取最新的值。

assert()

概念

- `assert()` 是一个宏，用于在调试过程中测试特定的条件是否为真。
- 它的原型定义在 `<assert.h>`（用于 C 语言）或 `<cassert>`（用于 C++）头文件中。

用途

- **条件测试**：`assert()` 宏用于验证程序中的假设和条件。如果条件为真（非零），程序继续执行；如果条件为假（零），则程序会显示错误消息并终止执行。
- **调试辅助**：`assert()` 常用于调试目的，帮助开发者快速定位错误。

关闭 assert

- 可以通过定义宏 `NDEBUG` 来关闭 `assert` 功能。这通常用于生产环境中，以避免因断言失败而导致的程序终止。
- `NDEBUG` 宏需要在包含 `<assert.h>` 或 `<cassert>` 之前定义。

使用示例

```
#define NDEBUG          // 定义 NDEBUG，关闭 assert
#include <assert.h>     // 或者 #include <cassert> 在 C++ 中

// ...

assert(p != NULL);      // 由于 NDEBUG 已定义，此 assert 不会执行
```

在这个示例中，由于定义了 `NDEBUG` 宏，因此 `assert(p != NULL)` 不会执行任何检查。如果去掉 `#define NDEBUG` 行，那么在 `p` 为 `NULL` 时，程序会打印错误消息并终止执行。

sizeof()

概念

- `sizeof()` 是一个运算符，用于计算变量或数据类型在内存中占用的空间大小。

使用

1. **数组**：对数组使用 `sizeof()`，可以得到**整个数组所占的空间大小。**
    - 示例：`sizeof(array)`
2. **指针**：对指针使用 `sizeof()`，得到的是**指针本身的大小，而非它指向的数据大小。**
    - 示例：`sizeof(pointer)`

#pragma pack(n)

概念

- `#pragma pack(n)` 是一个编译器指令，用于设置结构体、联合以及类成员变量的内存对齐方式。

使用

- `#pragma pack(push)` 和 `#pragma pack(pop)` 用于保存和恢复当前的对齐状态。
- `#pragma pack(n)` 设置后续声明的结构体、联合或类的成员变量按照 `n` 字节对齐。

示例

```
#pragma pack(push)  // 保存当前对齐状态
#pragma pack(4)     // 设置新的对齐为 4 字节

struct test {
    char m1;
    double m4;
    int m3;
};

#pragma pack(pop)   // 恢复之前的对齐状态
```

在这个示例中，`struct test` 的成员将按照 4 字节对齐方式来排列。这意味着即使 `char` 类型通常只需要 1 字节，但在这个结构体中，它可能会占用 4 字节的空间（具体取决于具体的编译器实现和其他成员的排列）。

位域（Bit Fields）

概念

- 类或结构体可以将其（非静态）数据成员定义为位域，即在一个位域中包含一定数量的二进制位。

特点

- **用途**：当程序需要向其他程序或硬件设备传递二进制数据时，通常会使用位域。
- **内存布局**：位域在内存中的布局是与机器相关的。
- **类型限制**：位域的类型必须是整型或枚举类型。带符号类型中的位域行为会因具体实现而有所不同。
- **取地址限制**：取地址运算符（`&`）不能作用于位域，且任何指针都无法指向类的位域。

示例

```
struct BitField {
    unsigned int mode: 2; // mode 占用 2 位
    // 其他成员
};
```

在这个示例中，`mode` 是一个占用 2 位的位域。这种定义允许 `mode` 存储的值范围限制在 0 到 3 之间。

extern "C"

概念

- `extern "C"` 用于 C++ 代码中，表明被修饰的变量和函数是按照 **C 语言的编译和链接方式处理**。

作用

- 让 C++ 编译器将 `extern "C"` 声明的代码当作 C 语言代码处理，这主要用于确保 C++ 代码能够与 C 语言库链接，**避免因 C++ 的符号修饰（name mangling）导致链接问题。**
- C++ 的符号修饰（name mangling）是一种在编译时期对函数和变量名进行唯一化的过程，以支持诸如函数重载等 C++ 特性。这个过程可能会导致链接时出现问题，特别是当 C++ 代码需要与 C 语言编写的库或函数互操作时。

使用场景

- 当需要在 C++ 中调用在 C 语言中编写的函数时，应该用 `extern "C"` 来声明这些函数。

示例

```
#ifdef __cplusplus
extern "C" {
#endif

void *memset(void *, int, size_t);

#ifdef __cplusplus
}
#endif
```

在这个示例中，`memset` 函数被定义为 `extern "C"`，这意味着即使在 C++ 环境中，它也会按照 C 语言的方式进行编译和链接。这样做确保了 C++ 程序可以与包含 `memset` 的 C 语言库正确链接。

struct 和 typedef struct 在 C 和 C++ 中的差异

C 语言中的使用

在 C 语言中，`struct` 关键字用于定义一个结构体，而 `typedef` 用于为类型创建一个新名称。例如：

```
// C 语言
typedef struct Student {
    int age;
} S;
```

这等价于：

```
// C 语言
struct Student {
    int age;
};

typedef struct Student S;
```

在这两种情况下，`S` 等价于 `struct Student`。由于 C 语言中的标识符名称空间规则，可以定义与 `struct Student` 名称不冲突的函数 `Student()`：

```
void Student() {
    // 函数实现
}
```

C++ 语言中的使用

C++ 改变了符号的查找规则，使得 `struct` 关键字在某些情况下可以省略。

1. **省略 struct 关键字**：
   在 C++ 中，如果在类标识符空间中定义了 `struct Student`，则可以直接使用 `Student` 来声明变量，不必显式写出 `struct`：

   ```
   // C++
   struct Student {
       int age;
   };

   void f(Student me); // 正确，"struct" 关键字可省略
   ```

2. **同名函数和结构体**：
   如果在定义了与结构体 `Student` 同名的函数后，`Student` 只代表该函数，不再代表结构体：

   ```
   // C++
   typedef struct Student {
       int age;
   } S;

   void Student() {}   // 定义后 "Student" 只代表此函数

   int main() {
       Student();      // 调用 Student 函数
       struct Student me; // 或者 "S me";
       return 0;
   }
   ```

   在这种情况下，`Student` 作为函数名将覆盖结构体的名称。要引用结构体，必须使用 `struct Student` 或其别名 `S`。

C++ 中 struct 和 class

区别

1. **默认访问控制**：
   - `struct` 的成员默认是 `public`。
   - `class` 的成员默认是 `private`。
2. **默认继承访问权限**：
   - `struct` 继承默认是 `public`。
   - `class` 继承默认是 `private`。

观点

- `struct` **更适合作为数据结构的实现体，强调数据的公开性。**
- `class` **更适合作为对象的实现体，强调封装和数据隐藏。**

union 联合

特点

- **默认访问控制符**：`union` 的默认访问控制符是 `public`。
- **构造函数和析构函数**：可以包含构造函数和析构函数。
- **成员限制**：不能包含引用类型的成员。
- **继承限制**：不能继承其他类，也不能作为基类。
- **虚函数**：不能含有虚函数。
- **匿名联合**：
  - 在定义所在的作用域内可以直接访问联合成员。
  - 不能包含 `protected` 或 `private` 成员。
- **全局匿名联合**：必须是静态（`static`）的。

使用

联合（`union`）是一种特殊的类，在任何时刻**只有一个数据成员可以有值。**如果给某个成员赋值，其他成员的值变为未定义。

示例：

```
union MyUnion {
    int intVal;
    float floatVal;
    char charVal;
};

MyUnion u;
u.intVal = 5; // 设置 int 成员
u.floatVal = 5.5; // 现在 float 成员是有效的，int 成员变为未定义
```

在这个例子中，联合 `MyUnion` 可以存储一个 `int`、一个 `float` 或一个 `char`，但在同一时间只能存储其中一个。当新的成员被赋值时，之前的成员值变为未定义。

C 实现 C++ 类

封装

在 C 中，可以通过结构体和函数来模拟封装。属性可以作为结构体的成员，方法可以通过函数指针在结构体内实现。

继承

继承可以通过嵌套结构体来模拟。子结构体包含一个父结构体作为其第一个成员，这样它就继承了父结构体的所有属性和方法。

多态

多态可以通过在结构体中使用函数指针来实现。可以根据需要在运行时更改这些函数指针，从而改变对象的行为。

示例

假设有一个基本的“动物”类，然后有一个“猫”类继承自“动物”类。

```c
#include <stdio.h>

// 基类 Animal
typedef struct Animal {
    void (*speak)(struct Animal*);  // 函数指针，用于实现多态
} Animal;

// 一个实现 Animal speak 方法的函数
void animalSpeak(Animal* animal) {
    printf("This animal makes a sound.\n");
}

// 创建 Animal
Animal newAnimal() {
    return (Animal){ animalSpeak };
}

// 子类 Cat 继承自 Animal
typedef struct Cat {
    Animal animal;  // 继承
    // 可以添加猫特有的属性和方法
} Cat;

// Cat 的 speak 方法
void catSpeak(Animal* animal) {
    printf("Meow!\n");
}

// 创建 Cat
Cat newCat() {
    Cat cat;
    cat.animal = newAnimal();
    cat.animal.speak = catSpeak;  // 重写 speak 方法
    return cat;
}

int main() {
    Animal animal = newAnimal();
    animal.speak(&animal);  // 输出: This animal makes a sound.

    Cat cat = newCat();
    cat.animal.speak((Animal*)&cat);  // 输出: Meow!

    return 0;
}
```

在这个例子中：

- `Animal` 结构体代表基类，其中有一个 `speak` 函数指针，用于实现多态。
- `Cat` 结构体继承自 `Animal`，通过在其内部包含一个 `Animal` 实例来实现。
- `Cat` 重写了 `speak` 方法，展现了多态的特性。

- 通过函数 `newAnimal` 和 `newCat` 创建相应的实例，并通过函数指针调用方法，展示了封装的特性。

explicit（显式）关键字

概念

`explicit` 关键字用于修饰类的构造函数和转换函数，以防止 C++ 中的某些隐式类型转换。

使用场景

1. **修饰构造函数**：
   - 当 `explicit` 修饰构造函数时，它阻止了该构造函数参与隐式转换和复制初始化。
   - 这意味着不能隐式地从一个类型转换为该类类型，必须显式地调用构造函数。
2. **修饰转换函数**：
   - 当 `explicit` 修饰类的转换函数时，它阻止了该转换函数在大多数情况下的隐式转换。
   - 但"按语境转换"（contextual conversion）仍然允许，如在布尔表达式中的隐式转换。

示例

explicit 构造函数

```
class MyClass {
public:
    explicit MyClass(int x) {
        // 构造函数实现
    }
};

MyClass obj = 10;  // 错误：不能隐式转换
MyClass obj(10);   // 正确：显式调用构造函数
```

在这个例子中，由于 `MyClass` 的构造函数被 `explicit` 关键字修饰，因此不能隐式地将一个 `int` 类型的值转换为 `MyClass` 类型的对象。

explicit 转换函数

```
class MyClass {
public:
    operator bool() const {
        return true;
    }
    /*
    如果是这样则不可以转换
    explicit operator bool() const {
        return true;
    }
    */
};

MyClass obj;
bool myBool = obj;  // 允许：隐式转换为 bool 类型
```

在这个例子中，尽管 `MyClass` 类型的对象可以隐式转换为 `bool` 类型，但如果转换函数被 `explicit` 修饰，则这种隐式转换将不被允许（除非在特定的语境下，如布尔表达式中）。

结论

使用 `explicit` 关键字可以避免意外的隐式转换，增强代码的安全性和清晰度。它是 C++ 语言中防止类型转换错误的重要特性之一。

friend 友元类和友元函数

特性

- **访问私有成员**：友元函数或友元类可以访问类的私有成员和保护成员。
- **破坏封装性**：虽然提供了某种程度的灵活性，但也破坏了类的封装性和隐藏性。
- **不可传递性**：友元关系**不可传递**。如果类 A 是类 B 的友元，类 B 是类 C 的友元，不意味着类 A 是类 C 的友元。
- **单向性**：友元关系**是单向的**，如果类 A 是类 B 的友元，不意味着类 B 也是类 A 的友元。
- **声明形式和数量无限制**：可以声明任意数量的友元，不受限制。

使用

```
class MyClass {
    friend void friendFunction(MyClass &obj); // 声明友元函数
    friend class FriendClass; // 声明友元类
    // ...
};
```

using 声明和指示

using 声明

- **单个成员引入**：`using` 声明一次只引入命名空间的一个成员，使得程序更加清晰。
- **语法**：`using namespace_name::name;`

构造函数的 using 声明（C++11）

- **继承构造函数**：派生类可以使用基类的构造函数。
- **语法**：

  ```
  class Derived : Base {
  public:
      using Base::Base; // 继承所有基类构造函数
  };
  ```

using 指示

- **整个命名空间引入**：使得某个命名空间中的所有名字都可见，无需前缀限定符。
- **语法**：`using namespace namespace_name;`
- **注意**：尽量少使用 `using` 指示，以避免命名冲突和污染命名空间。

使用建议

- **少用 using 指示**：可能导入不需要的名称，增加冲突风险。

```
using namespace std;
```

- **多用 using 声明**：只导入指定名称，更安全。

```
using std::cin;
using std::cout;
using std::endl;
```

:: 范围解析运算符

范围解析运算符 `::` 用于明确指定一个标识符的作用域。

分类

1. **全局作用域符（`::name`）**：
   - 当 `::` 前没有任何前缀时，它指定名称在全局作用域中查找。
   - 用于访问全局变量，尤其是当全局变量的名称被局部变量隐藏时。
2. **类作用域符（`class::name`）**：
   - 用于指定一个类的成员，无论是静态成员变量、成员函数还是类型定义（如嵌套类）。
   - 在类外定义成员函数时使用。
3. **命名空间作用域符（`namespace::name`）**：
   - 用于指定一个命名空间中的成员。
   - 用于访问命名空间中定义的变量、函数、类型等。

使用示例

全局作用域符

```
int count = 10; // 全局变量

void func() {
    int count = 5; // 局部变量
    ::count = 7;  // 修改全局变量 count
}
```

在这个例子中，`::count` 用于访问并修改全局变量 `count`，而非局部变量。

类作用域符

```cpp
class MyClass {
public:
    static int value;
    void method();
};

int MyClass::value = 5; // 定义类的静态成员

void MyClass::method() {
    // 定义类的成员函数
}
```

在这里，`MyClass::value` 和 `MyClass::method()` 用于定义 `MyClass` 类的静态成员和成员函数。

命名空间作用域符

```cpp
namespace MyNamespace {
    void myFunction() {
        // 函数实现
    }
}

int main() {
    MyNamespace::myFunction(); // 调用命名空间中的函数
}
```

在这个例子中，`MyNamespace::myFunction` 用于访问 `MyNamespace` 命名空间中的 `myFunction` 函数。

enum 枚举类型

枚举类型 `enum` 在 C++ 中用于定义一组命名的整型常量。

限定作用域的枚举类型（C++11）

- **语法**：`enum class EnumName { ... };`
- **特点**：
  - 强类型，不会隐式转换到其他类型。
  - 枚举值的作用域限定于枚举类内部，需要通过枚举类来访问。

示例

```cpp
enum class OpenModes { Input, Output, Append };

OpenModes mode = OpenModes::Input; // 使用枚举类名访问
```

在这个例子中，`OpenModes` 是一个限定作用域的枚举类，其成员 `Input`、`Output` 和 `Append` 必须通过 `OpenModes` 类来访问。

不限定作用域的枚举类型

- **语法**：`enum EnumName { ... };`
- **特点**：
  - 枚举值直接位于枚举类型所在的作用域中，可以直接访问。
  - 可以隐式转换为整数类型。

示例

```
enum Color { Red, Yellow, Green };

Color color = Red; // 直接访问
```

在这个例子中，`Color` 是一个不限定作用域的枚举类型，它的成员 `Red`、`Yellow` 和 `Green` 可以直接在枚举类型的作用域内访问。

匿名枚举

- **语法**：`enum { ... };`
- **特点**：
  - 没有名称，适用于只需要一次的枚举。
  - 其值也位于定义枚举的作用域内。

示例

```
enum { FloatPrec = 6, DoublePrec = 10 };

int precision = FloatPrec; // 直接访问
```

在这个例子中，定义了一个匿名枚举，其中包含两个枚举值 `FloatPrec` 和 `DoublePrec`，它们可以在枚举定义的作用域内直接访问。

decltype

概念

- `decltype` 是一个关键字，用于查询表达式的类型。

语法

- `decltype(expression)`

使用

- **尾置返回类型**：在 C++11 中，可以使用 `decltype` 指定函数的返回类型，尤其是当返回类型依赖于函数的参数时。

示例

```
// 尾置返回类型
template <typename It>
auto fcn(It beg, It end) -> decltype(*beg) {
    // 处理序列
    return *beg;  // 返回序列中一个元素的引用
}

// 结合 typename 和 remove_reference
template <typename It>
auto fcn2(It beg, It end) -> typename
std::remove_reference<decltype(*beg)>::type {
    // 处理序列
    return *beg;  // 返回序列中一个元素的拷贝
}
```

在这些例子中，`decltype` 用于推断函数的返回类型，`fcn` 返回一个元素的引用，而 `fcn2` 返回一个元素的拷贝。

引用

左值引用

- **定义**：常规引用，通常表示对象的身份。
- **语法**：`Type&`

右值引用

- **定义**：绑定到右值（临时对象或将要销毁的对象）的引用，表示对象的值。
- **特点**：
  - 实现转移语义（Move Semantics）和完美转发（Perfect Forwarding）。
  - 提高效率，减少不必要的对象拷贝。
- **语法**：`Type&&`

引用折叠规则

- `X& &`、`X& &&`、`X&& &` 折叠成 `X&`
- `X&& &&` 折叠成 `X&&`

宏

特点

- 宏定义在预处理器中执行，可以实现类似于函数的功能。
- 宏中的“参数”不是真正的函数参数，而是在宏展开时进行的直接文本替换。

注意事项

- 宏定义不是函数，不具有类型安全和作用域等函数特性。
- 使用宏时要注意参数的副作用，因为它们可能被宏体内多次替换和求值。

成员初始化列表

好处

1. **更高效**：使用成员初始化列表可以**减少一次调用默认构造函数**的过程。
2. **场合需求**：
    - **常量成员**：常量只能在初始化时赋值，不能在构造函数体内赋值。
    - **引用类型**：引用必须在定义时初始化，并且不能重新赋值。
    - **无默认构造函数的类类型成员**：使用成员初始化列表可以避免调用默认构造函数进行初始化。

示例

```
class Example {
    const int constMember; // 常量成员
    int& refMember;        // 引用成员
    MyClass obj;           // 无默认构造函数的类类型成员

public:
    Example(int x, int& y, MyClass z) : constMember(x), refMember(y), obj(z) {
        // 初始化列表中初始化成员
    }
};
```

`std::initializer_list` 列表初始化

介绍

- 使用花括号 `{}` 初始化对象，适用于构造函数接受 `std::initializer_list` 参数的情况。

使用

- 示例代码：

```
#include <iostream>
#include <vector>
#include <initializer_list>

template <class T>
struct S {
    std::vector<T> v;
    S(std::initializer_list<T> l) : v(l) {
        std::cout << "constructed with a " << l.size() << "-element list\n";
    }
    void append(std::initializer_list<T> l) {
        v.insert(v.end(), l.begin(), l.end());
    }
    std::pair<const T*, std::size_t> c_arr() const {
        return {&v[0], v.size()};
    }
};
```

```
int main() {
    S<int> s = {1, 2, 3, 4, 5}; // 列表初始化
    s.append({6, 7, 8});          // 函数调用中的列表初始化

    // 输出vector的内容
    for (auto n : s.v)
        std::cout << n << ' ';
    std::cout << '\n';
}
```

- 特点：支持自动类型推导、用于构造函数以及其他函数的参数。

面向对象程序设计 (OOP)

面向对象程序设计是一种以对象为中心的编程范式，它利用“对象”来表示数据和操作数据的方法。这种范式侧重于数据的抽象和封装，以及数据之间的交互。

面向对象的三大特性

1. **封装**：
    - 目的是将数据（属性）和操作数据的方法（行为）捆绑在一起，形成一个独立的对象。
    - 封装有助于隐藏内部实现的细节，只暴露对外的接口。
    - 关键字：`public`，`protected`，`private`。
        - `public` 成员：可被任意实体访问。
        - `protected` 成员：只允许被子类及本类的成员函数访问。
        - `private` 成员：只允许被本类的成员函数、友元类或友元函数访问。
2. **继承**：
    - 允许新创建的类（派生类）继承现有类（基类）的属性和方法。
    - 继承支持代码重用，并可以建立类之间的层次关系。
3. **多态**：
    - 指同一个操作作用于不同的对象时，可以产生不同的效果。
    - 实现方式：
        - **重载多态**（编译期）：函数重载、运算符重载。
        - **子类型多态**（运行期）：通过虚函数实现。
        - **参数多态性**（编译期）：类模板、函数模板。
        - **强制多态**（编译期/运行期）：类型转换。

多态的分类和实现

静态多态（编译期/早绑定）

- **函数重载**：
    - 同一作用域内，函数名相同但参数列表不同的多个函数。

```
class A {
public:
    void doAction(int a);
    void doAction(int a, int b);
};
```

动态多态（运行期/晚绑定）

- **虚函数**：
  - 使用 `virtual` 关键字修饰成员函数，使其成为虚函数。
  - 动态绑定：使用基类引用或指针调用虚函数时，会根据对象的实际类型来调用相应的函数。

注意事项：

- 派生类对象可以赋值给基类指针或引用，反之不可。
- 普通函数（非类成员函数）不能是虚函数。
- **静态函数（`static`）不能是虚函数。**
- 构造函数不能是虚函数。
- 内联函数在表现多态性时，不能作为虚函数。

示例代码：

```
class Shape {                   // 形状类
public:
    virtual double calcArea() {
        // ...
    }
    virtual ~Shape();
};

class Circle : public Shape {   // 圆形类
public:
    virtual double calcArea();
    // ...
};

class Rect : public Shape {     // 矩形类
public:
    virtual double calcArea();
    // ...
};

int main() {
    Shape *shape1 = new Circle(4.0);
    Shape *shape2 = new Rect(5.0, 6.0);
    shape1->calcArea();         // 调用圆形类的方法
    shape2->calcArea();         // 调用矩形类的方法
    delete shape1;
    delete shape2;
    return 0;
}
```

虚函数

纯虚函数

- **定义**：在基类中无法实现的虚函数，仅提供接口，没有具体实现。
- **声明方式**：`virtual ReturnType FunctionName() = 0;`
- **作用**：确保派生类实现该函数，提供统一的接口。
- **特点**：含有纯虚函数的类称为抽象类，这种类不能实例化。

虚函数

- **作用**：允许在派生类中重写函数，实现多态。
- **特点**：
  - 虚函数在类中是实现的，即使是空实现。
  - 虚函数在子类中可以不重写；但纯虚函数必须在子类中实现。
  - 用于“实作继承”，继承接口及其实现。

虚函数和纯虚函数的关系

- **共同点**：都支持多态，通过虚函数表实现动态绑定。
- **区别**：
  - 纯虚函数只是接口的声明，必须在派生类中实现。
  - 虚函数可以有默认实现，子类可以选择性重写。

虚函数指针与虚函数表

- **虚函数指针**：指向虚函数表，用于运行时确定函数地址。
- **虚函数表**：存储类中所有虚函数的地址，位于程序的只读数据段。

虚函数表（Virtual Table，通常称为 V-Table）和虚函数指针是C++实现多态的底层机制。这些概念对于理解面向对象编程中的动态绑定非常关键。

虚函数表 (V-Table)

当类中包含虚函数时，编译器会为该类创建一个虚函数表。这个表是一个静态数组，存储着指向类的虚函数的指针。对于每个有虚函数的类，都有一个对应的虚函数表。如果派生类重写了基类的虚函数，虚函数表中相应的入口会被更新为指向派生类中的函数。

虚函数指针 (V-Pointer)

虚函数指针是对象在内存中的一部分，指向相关联的虚函数表。每个实例化的对象都会有一个虚函数指针，即使是派生类对象也是如此。当调用虚函数时，程序通过虚函数指针来访问虚函数表，再从表中找到对应的函数地址来执行。

例子

假设有一个基类 `Base` 和一个从 `Base` 派生的类 `Derived`，它们都有虚函数 `show`。

```
class Base {
public:
    virtual void show() {
        cout << "Base show" << endl;
    }
};

class Derived : public Base {
public:
    void show() override {
        cout << "Derived show" << endl;
    }
};
```

当创建 `Base` 类型的对象时，该对象的内存布局中会包含一个指向 `Base` 类的虚函数表的虚函数指针。同样，当创建 `Derived` 类型的对象时，该对象的内存布局中也会包含一个虚函数指针，但它指向 `Derived` 类的虚函数表。

如果有如下代码：

```
Base* b = new Derived();
b->show();
```

这里发生的是：

1. `b` 是 `Base` 类型的指针，但指向 `Derived` 类型的对象。
2. 调用 `show` 函数时，程序首先查找 `b` 指向对象的虚函数指针。
3. 虚函数指针指向 `Derived` 的虚函数表。
4. 从虚函数表中找到 `show` 函数的实际地址，这是 `Derived` 类中 `show` 函数的地址。
5. 执行 `Derived` 中的 `show` 函数。

虚继承

- **目的**：解决多继承中的菱形继承问题。
- **实现方式**：通过虚基类指针（`vbptr`）和虚基类表。
- **特点**：
  - 虚基类在子类中只存在一份拷贝。
  - 虚基类指针会被继承，指向虚基类表，记录虚基类与派生类的偏移地址。

虚继承与虚函数的区别

- **共同点**：都使用虚指针和虚表。
- **区别**：
  - 虚继承中，虚基类存在于继承类中，占用存储空间；虚基类表存储的是偏移量。
  - 虚函数不占用存储空间；虚函数表存储的是函数地址。

类模板、成员模板、虚函数

- 类模板中可以使用虚函数。
- 类的成员模板（模板成员函数）不能是虚函数。

类模板中的虚函数

假设有一个类模板 `Base`，它有一个虚函数 `func`。然后，可以定义一个派生自这个类模板的类，并重写这个虚函数。

```
template <typename T>
class Base {
public:
    virtual void func() {
        std::cout << "Base func" << std::endl;
    }
};

template <typename T>
class Derived : public Base<T> {
public:
```

```
    void func() override {
        std::cout << "Derived func" << std::endl;
    }
};
```

在这个例子中，`Base` 是一个类模板，其中包含一个虚函数 `func`。`Derived` 是从 `Base` 派生的类模板，并重写了虚函数 `func`。

类的成员模板不能是虚函数

现在假设有一个普通类（非模板类），但它有一个模板成员函数。这个模板成员函数不能是虚函数。

```
class MyClass {
public:
    template <typename T>
    void func(T value) {
        std::cout << "Value: " << value << std::endl;
    }

    // 不能这样做：virtual template <typename T> void func(T value);
};
```

在这个例子中，`MyClass` 是一个普通类，它有一个模板成员函数 `func`。但是，不能使 `func` 成为虚函数，因为C++不支持模板虚函数。

抽象类、接口类、聚合类

抽象类

- **定义**：含有至少一个纯虚函数的类。
- **特点**：不能直接实例化。
- **用途**：提供一个基础框架，让派生类实现具体功能。

示例：

```
class AbstractClass {
public:
    virtual void pureVirtualFunction() = 0; // 纯虚函数
};
```

接口类

- **定义**：仅含有纯虚函数的抽象类。
- **特点**：仅用于声明接口，没有数据成员和实现。
- **用途**：强制派生类实现特定的函数。

示例：

```
class InterfaceClass {
public:
    virtual void interfaceFunction1() = 0;
    virtual void interfaceFunction2() = 0;
};
```

聚合类

- **定义**：用户可以直接访问其成员，并且具有特殊的初始化语法形式的类。
- **特点**：
  - 所有成员都是 public。
  - 没有定义任何构造函数。
  - 没有类内初始化。
  - 没有基类，也没有 virtual 函数。

示例：

```
struct AggregateClass {
    int a;
    double b;
    // ... 其他 public 成员
};
// 使用聚合初始化
AggregateClass obj = {1, 3.14};
```

在这个例子中，`AggregateClass` 是一个聚合类，因为它的所有成员都是公开的，并且它没有定义构造函数、没有类内初始化、没有基类和虚函数。这使得它可以使用花括号初始化语法进行初始化。

内存分配和管理

malloc、calloc、realloc、alloca

1. **malloc**：
   - 功能：申请指定字节数的内存。
   - 特点：申请到的内存中的初始值不确定。
2. **calloc**：
   - 功能：为指定长度的对象分配能容纳其指定个数的内存。
   - 特点：申请到的内存的每一位（bit）都初始化为 0。
3. **realloc**：
   - 功能：更改以前分配的内存长度（增加或减少）。
   - 特点：当增加长度时，可能需要将以前分配区的内容移到另一个足够大的区域，而新增区域内的初始值不确定。
4. **alloca**：
   - 功能：在栈上申请内存。
   - 特点：程序在出栈时自动释放内存。不具可移植性，不宜使用在需要广泛移植的程序中。C99 中支持变长数组（VLA），可替代 alloca。

malloc、free

- **用途**：用于分配和释放内存。
- **使用示例**：
  - 申请内存，确认是否成功：

    ```
    char *str = (char*) malloc(100);
    assert(str != nullptr);
    ```

- 释放内存后指针置空：

```
free(p);
p = nullptr;
```

new、delete

- **new / new[]**：
  - 功能：**先底层调用 malloc 分配内存**，然后**调用构造函数**创建对象。
  - 特点：自动计算所需字节数。
- **delete/delete[]**：
- 功能：**先调用析构函数清理资源**，然后底层调用 free 释放空间。
- **使用示例**：

```
int main() {
    T* t = new T(); // 先内存分配，再构造函数
    delete t;       // 先析构函数，再内存释放
    return 0;
}
```

定位 new (Placement New)

- **功能**：在预先指定的内存区域创建对象。
- **使用方法**：
  - `new (place_address) type`
  - `new (place_address) type (initializers)`
  - `new (place_address) type [size]`
  - `new (place_address) type [size] { braced initializer list }`
- **特点**：
  - `place_address` 是一个指向预分配内存的指针。
  - `initializers` 提供一个初始值列表。

示例：使用定位 new 创建对象

假设有一个简单的类 `MyClass`，想在预分配的内存区域中创建这个类的对象。

```
class MyClass {
public:
    MyClass() {
        std::cout << "MyClass constructed" << std::endl;
    }
    void show() {
        std::cout << "Method show called" << std::endl;
    }
    ~MyClass() {
        std::cout << "MyClass destructed" << std::endl;
    }
};
```

先分配一块内存，然后在这块内存上使用定位 new 来创建 `MyClass` 的实例。

```cpp
#include <iostream>
#include <new> // 必须包含这个头文件

int main() {
    // 分配足够的内存
    char memory[sizeof(MyClass)];

    // 在预分配的内存上创建对象
    MyClass* myObject = new (memory) MyClass;

    // 使用对象
    myObject->show();

    // 显式调用析构函数
    myObject->~MyClass();

    // 由于使用了 placement new，无需释放内存
    return 0;
}
```

在这个例子中，`memory` 是一个字符数组，分配了足够容纳 `MyClass` 对象的空间。然后，使用定位 new 在这块内存上构造了 `MyClass` 的实例。需要注意的是，由于使用了定位 new，所以**析构函数不会自动被调用，需要显式调用它。**同样，因为内存是手动管理的，所以也不需要使用 `delete` 来释放内存。

delete this 合法吗?

`delete this` 的合法性和注意事项

1. **对象通过 `new` 分配**：`this` 指针必须指向一个通过 `new`（而非 `new[]`、placement new）分配的对象。此外，它不能指向栈上（局部）对象、全局对象或其他对象的一部分。
2. **最后一次使用 `this`**：确保 `delete this` 是成员函数中最后一次使用 `this` 指针的操作。这意味着，执行 `delete this` 后，不应再访问任何成员变量或调用任何成员函数。
3. **调用 `delete this` 后不再使用**：一旦调用了 `delete this`，对象的内存就被释放了，因此再访问该对象是非法的，可能导致未定义行为。
4. **确保无其他引用**：在调用 `delete this` 之前，要确保没有其他地方还保留着对该对象的引用。一旦对象被删除，所有指向它的指针都将变成悬挂指针。

示例

```cpp
class MyClass {
public:
    void doSomething() {
        // Do some operations
        delete this;
        // 不能再访问任何成员，包括成员函数和成员变量
    }
};

int main() {
    MyClass* obj = new MyClass();
    obj->doSomething();
    // 此时 obj 已经不再有效
    return 0;
```

```
}
```

在这个例子中，对象 `obj` 通过 `new` 创建，并在 `doSomething` 成员函数内部使用 `delete this` 删除。一旦 `delete this` 被调用，`obj` 就不再指向一个有效的对象。

结论

虽然在技术上 `delete this` 是合法的，但它需要非常谨慎地使用，因为错误地使用它很容易导致程序错误、内存泄漏或其他未定义行为。通常，它只在特定的设计模式和场景中使用，比如在一些引用计数的实现中。在大多数情况下，应避免使用 `delete this`，并寻找更安全、更清晰的设计和实现方式。

如何定义一个只能在堆上（栈上）生成对象的类?

只能在堆上生成对象的类

方法

- 将析构函数设置为私有。

原因

- 在C++中，栈上对象的生命周期是由编译器自动管理的，这包括了自动调用析构函数来销毁对象。如果析构函数是私有的，编译器将无法在栈上创建该类的对象。

示例

```
class HeapOnly {
public:
    // 提供一个公共的接口来创建对象
    static HeapOnly* createInstance() {
        return new HeapOnly();
    }

private:
    // 析构函数为私有
    ~HeapOnly() {}

    // 类的其他成员
};
```

在这个例子中，由于析构函数是私有的，所以不能在栈上创建 `HeapOnly` 类的对象。但可以通过公共的静态成员函数 `createInstance` 在堆上创建对象。

只能在栈上生成对象的类

方法

- 将 new 和 delete 操作符重载为私有。

原因

- 在堆上创建对象通常涉及使用 new 关键字。通过将 new 和 delete 设置为私有，可以阻止在堆上分配类的实例。

示例

```
class StackOnly {
private:
    // 重载 new 和 delete 为私有
    void* operator new(size_t size) = delete;
    void operator delete(void* pointer) = delete;

public:
    StackOnly() {}
    ~StackOnly() {}

    // 类的其他成员
};
```

在这个例子中，由于 new 和 delete 被设置为私有（或者被删除），因此不能在堆上创建 `StackOnly` 类的对象。但可以在栈上正常创建和销毁对象。

智能指针

头文件

```
#include <memory>
```

智能指针类型

1. **C++ 98**
    - `std::auto_ptr`：早期的智能指针，现已被弃用。
2. **C++ 11**
    - `std::shared_ptr`：实现共享式拥有。多个智能指针可共享同一对象，对象在最后一个引用被销毁时释放。
    - `std::unique_ptr`：实现独占式拥有。保证同一时间只有一个智能指针指向对象，可以转移拥有权。
    - `std::weak_ptr`：实现非拥有性共享。它不会增加对象的引用计数，用于打破环状引用。
    - `std::auto_ptr`（弃用）：由于设计上的瑕疵，如缺乏移动语义，已在C++11中被废弃。

shared_ptr

- 允许多个 `shared_ptr` 实例共享同一个对象的拥有权。
- 采用引用计数机制。
- 提供定制删除器，可解决跨动态链接库的问题，支持自动解除互斥锁。
- 可与 `weak_ptr` 配合使用，防止环状引用。

weak_ptr

- 用于观察但不延长 `shared_ptr` 管理的对象的生命周期。
- 不增加对象的引用计数。
- 在 `shared_ptr` 的最后一个实例被销毁后，`weak_ptr` 会自动变为空。

`std::weak_ptr` 是一种智能指针，它被设计为与 `std::shared_ptr` 协同工作，用于解决可能由 `std::shared_ptr` 引起的循环引用问题。`std::weak_ptr` 持有对对象的非拥有（弱）引用，这意味着它不会增加对象的引用计数。这使得 `std::weak_ptr` 指向的对象可以被正常释放，即使仍有 `weak_ptr` 指向它。

例子：使用 `std::weak_ptr` 解决循环引用

假设有两个类，`A` 和 `B`，每个类中都有一个指向另一个类实例的 `std::shared_ptr`。这种情况下，如果 `A` 的实例拥有一个指向 `B` 的实例的 `std::shared_ptr`，而 `B` 的实例又拥有一个指向 `A` 的实例的 `std::shared_ptr`，就会产生循环引用。这样，即使外部没有指针指向这两个对象，它们也不会被析构，从而导致内存泄漏。

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

class B; // 前向声明

class A {
public:
    std::shared_ptr<B> b_ptr;
    ~A() { std::cout << "A destroyed" << std::endl; }
};

class B {
public:
    std::shared_ptr<A> a_ptr;
    ~B() { std::cout << "B destroyed" << std::endl; }
};

int main() {
    std::shared_ptr<A> a = std::make_shared<A>();
    std::shared_ptr<B> b = std::make_shared<B>();

    a->b_ptr = b;
    b->a_ptr = a;

    return 0; // a 和 b 由于循环引用，不会被销毁
}
```

在上面的例子中，由于循环引用，即使 `main` 函数执行完毕，`A` 和 `B` 的实例也不会被销毁。

要解决这个问题，可以使用 `std::weak_ptr` 来代替其中一个类中的 `std::shared_ptr`。这样，其中一个类对另一个的引用就不会增加引用计数，从而打破循环引用，使得对象能够正确释放。

```
class B;

class A {
public:
    std::shared_ptr<B> b_ptr;
```

```
    ~A() { std::cout << "A destroyed" << std::endl; }
};

class B {
public:
    std::weak_ptr<A> a_ptr; // 使用 weak_ptr
    ~B() { std::cout << "B destroyed" << std::endl; }
};

int main() {
    std::shared_ptr<A> a = std::make_shared<A>();
    std::shared_ptr<B> b = std::make_shared<B>();

    a->b_ptr = b;
    b->a_ptr = a;

    return 0; // 现在 a 和 b 能够被正确销毁
}
```

在这个修改后的例子中，当 `main` 函数执行完毕时，`A` 和 `B` 的实例都能够被正确销毁，因为 `B` 中的 `a_ptr` 是一个 `std::weak_ptr`，它不会增加 `A` 实例的引用计数。

总结

`std::weak_ptr` 是一种特殊的智能指针，用于解决 `std::shared_ptr` 可能引起的循环引用问题，同时提供对对象的安全访问，而不影响对象的生命周期。

unique_ptr

- 提供独占式拥有权。
- 只有一个 `unique_ptr` 可指向一个特定对象。
- 支持移动语义，可以安全地转移拥有权。
- 可以管理数组，并且相比 `auto_ptr` 提供更安全的资源管理。

auto_ptr（弃用）

- 在C++11之前的智能指针，现在不推荐使用。
- 存在的问题：
    - 执行复制操作时会改变源对象的状态（所有权转移）。
    - 缺乏移动语义。
    - 不支持管理数组。

强制类型转换运算符

static_cast

- **用途**：用于非多态类型的转换。
- **特点**：
    - 不执行运行时类型检查，转换安全性不如 `dynamic_cast`。
    - 通常用于转换数值数据类型（如 `float` -> `int`）。
    - 可以在整个类层次结构中移动指针。向上转换（子类转化为父类）是安全的，但向下转换（父类转化为子类）可能不安全。

- **示例**：

```
float f = 3.5;
int i = static_cast<int>(f); // float 转 int
```

dynamic_cast

- **用途**：用于多态类型的转换。
- **特点**：
  - 执行运行时类型检查。
  - 仅适用于指针或引用。
  - 对不明确的指针的转换将失败（返回 `nullptr`），但不引发异常。
  - 可以在整个类层次结构中移动指针，包括向上转换和向下转换。
- **示例**：

```
class Base { virtual void dummy() {} };
class Derived : public Base { int a; };

Base *pb = new Derived;
Derived *pd = dynamic_cast<Derived*>(pb);

if(pd == nullptr) {
    cout << "转换失败" << endl;
} else {
    cout << "转换成功" << endl;
}
```

const_cast

- **用途**：用于删除 `const`、`volatile` 和 `__unaligned` 特性（例如，将 `const int` 类型转换为 `int` 类型）。
- **示例**：

```
const int a = 10;
const int* pa = &a;
int* p = const_cast<int*>(pa);
*p = 20; // 改变了 a 的值
```

reinterpret_cast

- **用途**：用于位的简单重新解释。
- **特点**：
  - 可以将任何指针转换为任何其他指针类型，也允许将任何整数类型转换为任何指针类型以及反向转换。
  - 不安全，应谨慎使用，除非所需转换本身是低级别的。
  - 不能去除 `const`、`volatile` 或 `__unaligned` 特性。
- **示例**：

```
int* p = new int(10);
char* ch = reinterpret_cast<char*>(p); // 将 int* 转换为 char*
```

bad_cast

- **用途**：由于强制转换为引用类型失败，`dynamic_cast` 运算符引发的 `bad_cast` 异常。
- **使用示例**：

```
try {
    Circle& ref_circle = dynamic_cast<Circle&>(ref_shape);
}
catch (bad_cast b) {
    cout << "Caught: " << b.what();
}
```

在这个示例中，尝试将 `Shape` 类型的引用转换为 `Circle` 类型的引用。如果 `ref_shape` 实际上不是 `Circle` 类型的对象，这个转换将失败并引发 `bad_cast` 异常。

运行时类型信息 (RTTI)

dynamic_cast

- **用途**：用于多态类型的转换。
- **特点**：执行运行时类型检查，安全地将指针或引用从基类转换为派生类。
- **示例**：

```
class Base { virtual void dummy() {} };
class Derived: public Base { int a; };

Base *b = new Derived;
Derived *d = dynamic_cast<Derived*>(b);
if (d) {
    std::cout << "转换成功" << std::endl;
} else {
    std::cout << "转换失败" << std::endl;
}
```

typeid

- **用途**：用于在运行时确定对象的类型。
- **特点**：
  - `typeid` 运算符返回一个 `type_info` 对象的引用。
  - 如果通过基类的指针或引用获取派生类的实际类型，基类必须有虚函数。
  - 仅能获取对象的实际类型。
- **示例**：

```
class Base { virtual void dummy() {} };
class Derived : public Base {};

Base* base = new Derived;
if (typeid(*base) == typeid(Derived)) {
    std::cout << "base 指向 Derived 类型" << std::endl;
}
```

type_info

- **用途**：描述编译器生成的类型信息。
- **头文件**：`<typeinfo>`
- **特点**：
  - 存储指向类型名称的指针。
  - 提供方法比较两个类型是否相等或确定它们的排列顺序。
  - 类型的编码规则和排列顺序未指定，可能因程序而异。
- **示例**：

```
type_info const& info = typeid(*base);
std::cout << "类型名称: " << info.name() << std::endl;
```