第19章 特殊工具与技术

19.1 控制内存分配

19.1.1 重载 new 和 delete

1. new一个对象分三个步骤：
   （1） new 表达式调用名为 operator new （或者 operator new[] ）的标准库函数。该函数分配一块足够大的、原始的、未命名的内存空间以便存储特定类型的对象（或者对象的数组）。
   （2）编译器运行相应的构造函数以构造这些对象，并为其传入初始值。
   （3）对象被分配了空间并构造完成，返回一个指向该对象的指针。
2. delete 一个对象分两个步骤：
   （1）调用对象的析构函数。
   （2）编译器调用名为 operator delete （或者 operator delete[] ）的标准库函数释放内存空间。
3. 如果应用程序希望控制内存分配的过程，则它们需要定义自己的 operator new 函数和 operator delete 函数。即使在标准库中已经存在这两个函数的定义，我们仍旧可以定义自己的版本。
4. 当自定义了全局的 operator new 函数和 operator delete 函数后，我们就负担起了控制动态内存分配的职责。这两个函数必须是正确的：因为它们是程序整个处理过程中至关重要的一部分。
5. 当我们将上述两个运算符函数定义成类的成员时，它们是**隐式静态的**，也必须是静态的。而且它们不能操纵类的任何数据成员。
6. 标准库函数 operator new 和 operator delete 的名字容易让人误解。和其他 operator 函数不同（比如 operator= ），这两个函数并没有重载 new 表达式和 delete 表达式。实际上，我们根本无法自定义 new 表达式或 delete 表达式的行为。
7. 我们提供新的 operator new 函数和 operator delete 函数的目的在于改变内存分配的方式，但是不管怎样，我们都不能改变 new 运算符和 delete 运算符的基本含义。

```
void* operator new(size_t size){
    if(void* mem = malloc(size))
        return mem;
    else
        throw bad_alloc();
}

void operator delete(void* mem) noexcept {
    free(mem);
}
```

19.2 运行时类型识别

1. **运行时类型识别(run-time type identification, RTTI)** 的功能由两个运算符实现：
   （1）**type**运算符，用于返回表达式的类型。
   （2）**dynamic_cast** 运算符，用于将基类的指针或引用安全地转换成派生类的指针或引用。
2. 程序员必须清楚地知道转换的目标类型并且必须检查类型转换是否被成功执行。
3. 使用RTTI必须要加倍小心。在可能的情况下，最好定义虚函数而非直接接管类型管理的重任。

19.2.1 `dynamic_cast` 运算符

1. `dynamic_cast` 运算符的使用形式：

   ```
   dynamic_cast<type*>(e)
   dynamic_cast<type&>(e)
   dynamic_cast<type&&>(e)
   ```

2. 我们可以对一个空指针执行 `dynamic_cast` ，结果是所需类型的空指针。
3. 当引用的类型转换失败时，程序抛出一个名为 `std::bad_cast` 的异常：

```
void f(const Base& b)
{
    try {
        const Derived& d = dynamic_cast<const Derived&>(b);
    } catch (bad_cast) {
        // 处理类型转换失败的情况
    }
}
```

19.2.2 `typeid` 运算符

1. `typeid` 表达式的形式是 `typeid(e)` ，其中e可以是任意表达式或类型的名字。 `type` 操作的结果是一个常量对象的引用，该对象的类型是标准库类型 `type_info` 或 `type_info` 的公有派生类型。

```
Derived* dp = new Derived;
Base* bp = dp;  // 两个指针都指向Derived对象;

// 在运行时比较两个对象的类型
if(typeid(*dp) == typeid(*bp)) {
    // dp 和 bp 指向同一类型的对象
}

//检查运行时类型是否是某种指定的类型
if(typeid(*bp) == typeid(Derived)) {
    // bp实际指向Derived对象
}

// 注意，typeid应该作用于对象，因此我们使用*bp而非bp
// 下面的检查永远是失败的：bp的类型是指向Base的指针
if(typeid(bp) == typeid(Derived)) {
    // 此处的代码永远不会执行
}
```

2. 当 `typeid` 作用于指针时（而非指针所指的对象），返回的结果是该指针的静态编译时类型。
3. 如果p是一个空指针，则 `typeid(*p)` 将抛出一个名为 `bad_typeid` 的异常。

19.2.4 `type_info` 类

1. 创建 `type_info` 对象的唯一途径是使用 `typeid` 运算符。
2. 对于某种给定的类型来说， `name` 的返回值因编译器而异并且不一定与在程序中使用的名字一致。对于 `name` 返回值的唯一要求是，类型不同则返回的字符串必须有所区别。
3. 有的编译器提供了额外的成员函数以提供程序中所用类型的额外信息。

19.3 枚举类型

1. **枚举类型（enumeration）** 使我们可以将一组整型常量组织在一起。和类一样，每个枚举类型定义了一种新的类型。枚举属于字面值常量类型。
2. c++包含两种枚举：限定作用域和不限定作用域的。
3. 在限定作用域的枚举类型中，枚举成员的名字遵循常规的作用域准则，并且在枚举类型的作用域外是不可访问的。与之相反，在不限定作用域的枚举类型中，枚举成员的作用域与枚举类型本身的作用域相同。
4. 枚举成员是 `const` ，因此在初始化枚举成员时提供的初始值必须是常量表达式。
5. 想要初始化 `enum` 对象或者为 `enum` 对象赋值，必须使用该类型的一个枚举成员或者该类型的另一个对象。

6. 即使某个整数值恰好与枚举成员的值相等，它也不能作为函数的 enum 实参使用。
7. c++11新标准中，我们可以在 enum 的名字后加上冒号以及我们想在该 enum 中使用的类型。
8. c++11新标准中，我们可以提前声明 enum 。 enum 的前置声明（无论隐式地还是显示地）必须指定其成员的大小。
9. enum 的声明和定义必须匹配，这意味着在该 enum 的所有声明和定义中成员的大小必须一致。而且，我们不能在同一个上下文中先声明一个不限定作用域的 enum 名字，然后再声明一个同名的限定作用域的 enum 。

19.4 类成员指针

1. **类成员指针（pointer to member）** 是指可以指向类的非静态成员的指针。
2. 成员指针的类型囊括了类的类型以及成员的类型。
3. 当初始化一个这样的指针时，我们令其指向类的某个成员，但是不指定该成员所属的对象；直到使用成员指针时，才提供成员所属的对象。

19.4.1 数据成员指针

```
class Screen {
private:
    std::string contents;
}

// pdata可以指向一个常量（非常量）Screen对象的String对象
const string Screen::*pdata;

// 初始化一个成员指针（或给它赋值）时，需指定它所指的成员。
pdata = &Screen::contents;
```

1. 当我们初始化一个成员指针或为成员指针赋值时，该指针并没有指向任何数据。成员指针指定了成员而非该成员所属的对象，只有当解引用成员指针时我们才提供对象的信息。

```
Screen myscreen, *pScreen = &myScreen;


// .*解引用pdata以获得myScreen对象的contents成员
auto s = myScreen.*pdata;

// ->*解引用pdata以获得pScreen所指对象的contents成员
s= pScreen->*pdata;

// 从概念上来说，这些运算符执行两个操作：
// 它们首先解引用成员指针以得到所需的成员
// 然后像成员访问运算符一样，通过对象（.*）或指针（->*）获取成员
```

2. 可以定义一个返回私有成员指针的函数。

```
class Screen {
public:
    // data()是一个静态成员，返回一个成员指针
    static const std::string Screen::*data()
        { return &Screen::contents; }
private:
    std::string contents;
}

// 调用data()
const string Screen::*pdata = Screen::data();

// 获取myScreen对象的contents成员
auto s = myScreen.*pdata;
```

19.4.2 成员函数指针

```
class Screen {
public:
    char get_cursor() const;
    char get(pos ht, pos wd) const;
    // ...
}

// 定义成员函数的指针:
// pmf是一个指针，它可以指向Screen的某个常量成员函数
// 前提是该函数不接受任何实参，并且返回一个char
auto pmf = &Screen::get_cursor;

// 也可以先声明一个指针，令其指向含有两个形参的get()
char (Screen::*pmf2)(Screen::pos, Screen::pos) const;
pmf2 = &Screen::get;

// 调用成员函数指针
Screen myScreen, *pScreen = &myScreen;
char c1 = (pmyScreen->*pmf)();
char c2 = (myScreen.*pmf2)(0, 0);
```

1. 因为函数调用符优先级比较高，所以在声明指向成员函数的指针并使用这样的指针进行函数调用时，括号必不可少：`(C::*p)(parms)` 和 `(obj.*p)(args)` 。
2. 通过使用类型别名，可以令含有成员指针的代码更容易读写。

```
// Action 是函数指针的别名
using Action =
    char (Screen::*)(Screen::pos, Screen::pos) const;

// 让Action函数指针指向Screen的get(pos,pos)
Action get = &Screen::get;

// 可以定义一个函数action
// 将指向成员函数的指针Action作为action函数的形参类型
Screen& action(Screen& , Action = &Screen::get);

Screen myScreen;
// 等价的调用:
action(myScreen);                // 使用默认实参
action(myScreen, get);           // 使用我们定义的函数指针get
action(myScreen, &Screen::get)   // 显示地传入地址
```

3. 对于普通函数指针和指向成员函数的指针来说，一个常见的用法是将其存入一个函数表当中。

19.4.3 将成员函数用作可调用对象

1. 因为成员指针不是可调用对象，所以我们不能直接将一个指向成员函数的指针传递给算法：

```
vector<string> svec;

// 错误，必须使用.*或->*调用成员指针
auto fp = &string::empty;   // fp指向string的empty函数
find_if(svec.begin(), svec.end(), fp);

// 从指向成员函数的指针获取可调用对象的
// 一种方法是使用标准库模板function：
function<bool (const string&)> fcn = &string::empty;
find_if(svec.begin(), svec.end(), fcn);

// 第二中方法是使用mem_fn
find_if(svec.begin(), svec.end(), mem_fn(&string::empty));

// 第三中办法是使用bind
find_if(svec.begin(), svec.end(), bind(&string::empty, _1));
```

19.6 union：一种节约空间的类

1. **联合（union）** 是一种特殊的类。一个 union 可以有多个数据成员，但是在任意时刻只有一个数据成员可以有值。
2. 当我们给 union 的某个成员赋值后，该 union 的其他成员就变成未定义的状态了。
3. union 不能含有引用类型的成员。
4. 由于 union 既不能继承自其他类，也不能作为基类使用，所以在 union 中不能含有虚函数。
5. **匿名union** 不能包含受保护的成员或私有成员，也不能定义成员函数。
6. 当我们将 union 的值改为类类型成员对象的值时，必须运行该类型的构造函数。反之，当我们将类类型成员的值改为一个其他值时，必须运行该类型的析构函数。

19.7 局部类

1. 类可以定义在某个函数的内部，我们称这样的类为**局部类（local class）**。
2. 局部类定义的类型只在定义它的作用域内可见。
3. 局部类的所有成员（包括函数在内）都必须完整定义在类的内部。因此，局部类的作用与嵌套类相比相差很远。

4. 局部类只能访问外层作用域定义的类型名、静态变量以及枚举成员。
5. 如果局部类定义在某个函数内部，则该函数的普通局部便不能被该局部类使用。
6. 可以在局部类的内部再嵌套一个类。此时，嵌套类的定义可以出现在局部类之外。不过，嵌套类必须定义在与局部类相同的作用域内。

19.8 固有的不可移植的特性

1. 为了支持低层编程，c++定义了一些固有的**不可移植（nonportable）的特性。
2. 该特性是指因机器而异的特性，当我们将含有不可移植特性的程序从一台机器转移到另一台机器上时，通常需要重新编写该程序。

19.8.1 位域

1. 类可以将其（非静态）数据成员定义成**位域（bit-filed）**。
2. 位域在内存中的布局是与机器相关的。
3. 位域类型必须是整型或枚举类型。
4. 通常情况下最好将位域设为无符号类型，存储在带符号类型中的位域的行为将因具体实现而定。

```
typedef unsigned int Bit;
class File {
    Bit mode: 2;         // mode占2位
    Bit modified: 1;     // modified占1位
    Bit prot_owner: 3;   // prot_owner占3位
    // ...
}
```

19.8.2 `volatile` 限定符

1. `volatile` 是一种类型限定符，告诉编译器变量可能在程序的直接控制之外发生改变。它起到一种标示作用，令编译器不对代码进行优化操作。

19.8.3 链接指示： `extern "C"`

1. c++使用**链接指示（linkage directive）** 指出任意非c++函数所用的语言。