深蓝学院
shenlanxueyuan.com

C++ 基础

第 15 章：其它的工具与技术

主讲人 李伟

微软高级工程师

《C++ 模板元编程实战》作者

目录

异常处理

- 用于处理程序在调用过程中的非正常行为
  - 传统的处理方法：传返回值表示函数调用是否正常结束
  - C++ 中的处理方法：通过关键字 try/catch/throw 引入异常处理机制 异常产生和异常处理的位置是不同的
    好处：产生异常的地方和处理异常的地方进行了分离。
- 异常触发时的系统行为——栈展开
  - 抛出异常后续的代码不会被执行
  - 局部对象会按照构造相反的顺序自动销毁
  - 系统尝试匹配相应的 catch 代码段
    - 如果匹配则执行其中的逻辑，之后执行 catch 后续的代码
    - 如果不匹配则继续进行栈展开，直到“跳出” main 函数，触发 terminate 结束运行

```
terminate called after throwing an instance of 'int'
/home/test1/.codelite/tmp/test1/codelite-exec.sh: 行 3:  3581 已放弃
   （核心已转储） ${command}
Hit any key to continue...
```

- 异常对象
  - 系统会使用抛出的异常拷贝初始化一个临时对象，称为异常对象 catch(int e)
  - 异常对象会在栈展开过程中被保留，并最终传递给匹配的 catch 语句

```
void f1() {
  throw 1;//抛出异常
}
void f2() {
  f1();
}
void f3() {
  f2();
}
int main() {
  //在我们希望处理的地方写try
  try {
    f3();
  }
  catch(int e) {
    //处理异常
    std::cout << "exception is carched: " << e;
  }
}
//系统首先会由main建立一个栈帧，在main这个
栈帧上面放f3(),f3()上面会放f2()的栈帧。
我们在f1()抛出这个异常之后，会在main处理异常
。相应的，就会将f1,f2,f3的栈帧抛弃。将程序的
运行环境交还给main()这个函数。
上面抛弃f1,f2,f3的过程被称为栈展开。
```

异常处理——续 1

- try / catch 语句块
  - 一个 try 语句块后面可以跟一到多个 catch 语句块
  - 每个 catch 语句块用于匹配一种类型的异常对象
  - catch 语句块的匹配按照从上到下进行
  - 使用 catch(...) 匹配任意异常
  - 在 catch 中调用 throw 继续抛出相同的异常
- 在一个异常未处理完成时抛出新的异常会导致程序崩溃
  - 不要在析构函数或 operator delete 函数重载版本中抛出异常
  - 通常来说， catch 所接收的异常类型为引用类型

```
try {
  f1();
}
catch(int e) {
  std::cout << "xx";
}
catch(int e) {
  std::cout << "xx2";
}
//这个代码也是合法的代码。不过是从上到下进行匹配。可以说下面的catch语句没有什么用
```

```
throw 314;
try() {
}
catch(,,,) {
  std::cout << "exception";
  throw; //这个throw和前面那个throw不一样。这里表示的是，我要抛出接收到的这个异常（只能写在catch内部）
}
```

如果抛出的是简单的int，double类型。那么可以不用引用

如果不用引用，那么就需要拷贝初始化这个对象。但是如果在拷贝初始化的过程中包含了一个抛出异常的操作。那么就会导致程序崩溃

```
struct Base {};
struct Derive: Base {};

void f1() {
  throw Derive{};
}
int main() {
  try {
    f1();
  }
  catch(Base& e) {
    std::cout << "Base ";
  }
  catch(Derive& e) {
    std::cout << "Derive";
  }
//系统会打印Base.为什么呢？它在throw的
时候构造了一个临时对象，这个异常对象是
Derived,接下来它使用这个异常对象来初始化
catch的对象。接下来他发现它可以初始化Base&
,因此系统就会直接匹配Base&这个函数
//那为什么double不能匹配int的对象呢？
我们在进行异常匹配的时候，系统不会进行这样
的转化，只有下面这几种情况才会进行转化
1.非cosnt -> const
2. 派生类 -> 基类
3. 数组函数 -> 指针
}
//通常来说在匹配异常的时候，
建议大家使用引用的方式
```

异常处理——续 2

主要用途之一就是给构造函数.能够捕获在函数初始化列表中出现的异常

- 异常与构造、析构函数
  - 使用 function-try-block保护初始化逻辑
  - 在构造函数中抛出异常：
    - 已经构造的成员会被销毁，但类本身的析构函数不会被调用
- 描述函数是否会抛出异常
  - 如果函数不会抛出异常，则应表明以为系统提供更多的优化空间
    - C++ 98 的方式： throw() / throw(int, char)
    - C++11 后的改进： noexcept / noexcept(false)

```
void fun() noexcept //函数不会抛出异常
void fun() noexcept(false) //函数可能会抛出异常。相比于c++98,把一些编译器信息去掉了，让运行期处理。
```

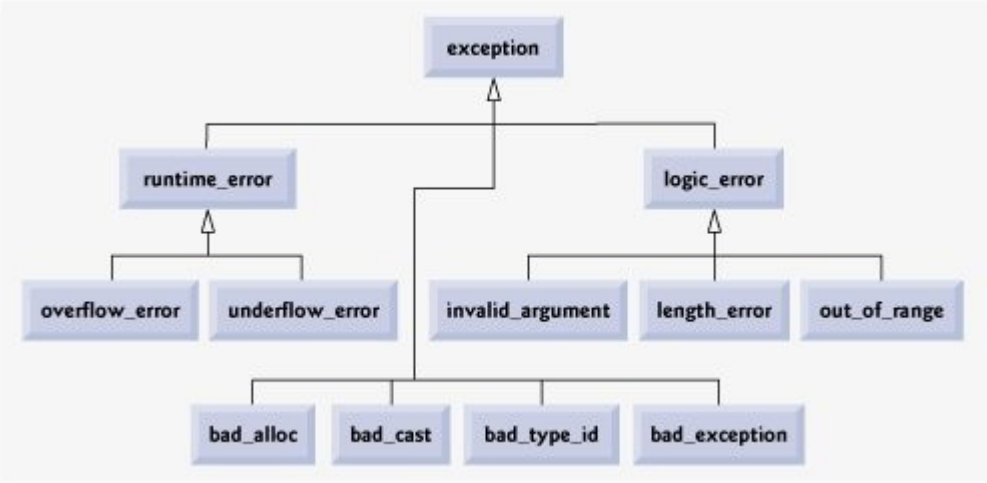
  - noexcept
    - 限定符：接收 false / true 表示是否会抛出异常
    - 操作符：接收一个表达式，根据表达式是否可能抛出异常返回 false/true
    - 在声明了 noexcept 的函数中抛出异常会导致 terminate 被调用，程序终止
    - 不作为函数重载依据，但函数指针、虚拟函数重写时要保持形式兼容

```
struct Str {
  Str() {throw 100;}
}

class Cla {
public:
  //func-try-block
  Cla() try: m_mem() //try应该被写在这里
  {
    //这个函数被调用的时候，类已经被
初始化好了。
    // try {}//我们虽然希望在这里catch,但实际
上我们catch不到任何东西
    // catch(int) {
    //  std::cout << "exption";
    //}
  }
  //catch应该写在这里
  catch(int) {
    std::cout << "exception";
  }
private:
  Str m_mem;
}

int main() {
  try {
    Cla obj;
  }
  catch (int) {
    std::cout << "exception is catched";
  }
}

//输出的时候，两个catch都捕获到了异常。因为c++
规定。他会在class的catch语句中隐式的加一个throw
这就意味着，如果在main函数中没有写catch，那么
编译器会报错。因为class中编译器隐式加的throw没有
被捕获。
```

```
//noexcept
void fun() noexcept(true) {}
int main() {
  std::cout << noexcept(fun());
  这里面的Noexcept不再是一个限定符了，而是
一个操作符。接受一个表达式。在这里，如果这个
表达式会抛出异常，那就会输出false，如果不会
抛出异常，就会返回true.
}

//这个用法可以帮助我们来检测其它函数是否会
抛出异常
void fun2() {}
void fun() noexcept(noexcept(fun2()) {
//外层noexcept是一个限定符，内层noexcept是
一个操作符
// noexcept(noexcept(fun2()) && noexcept(fun3()))
  //...
  fun2();
  //...
}
```

异常处理——续 3

- 标准异常（参考文献）




```
void fun() {
  throw std::runtime_error("Invalid input");
}
int main() {
  try {
    fun();
  }
  catch (std::runtime_error& e) {
    std::cout << e.what();
  }
}
```

- 正确对待异常处理
  - 不要滥用：异常的执行成本非常高
  - 不要不用：对于真正的异常场景，异常处理是相对高效、简洁的处理方式
  - 编写异常安全的代码

```
要尽量避免裸的资源分配
int* ptr = new int[3];
delete []ptr;//裸的资源分配
```

异常处理的逻辑：
1.建立临时异常对象
2. 维护栈展开逻辑
3. 跳跃1-多个函数，到catch

枚举与联合

- 枚举(enum)：一种取值受限的特殊类型
  - 分为无作用域枚举与有作用域枚举（C++11 起）两种
  - 枚举项缺省使用 0 初始化，依次递增，可以使用常量表达式来修改缺省值
  - 可以为枚举指定底层类型，表明了枚举项的尺寸 可以节省空间
  - 无作用域枚举项可隐式转换为整数值；也可用 static_cast 在枚举项与整数值间转换
  - 注意区分枚举的定义与声明

```
enum Color: char //指定底层类型
{
  Red,
  Green
};
```

```
Color x = 100;//编译报错。整数值不能隐式的转换为枚举项
std::cout << Color::Red; //编译报错，有作用域枚举项不可以隐式转换为整数值
std::cout << static_cast<int>(Color::Red); //显示转换可以
```

- 联合（union）：将多个类型合并到一起以节省空间 共享内存
  - 通常与枚举一起使用
  - 匿名联合
  - 在联合中包含非内建类型（C++11 起）

```
union Str {
  int x;
  int y;
};
int main() {
  Str obj;
  obj.x = 100; //将x和y都修改了
  std::cout << obj.y << std::endl; //打印出来是100
}
```

```
//无作用域枚举
enum Color {
  Red,
  Green,
  Yellow
};
int main() {
  Color x = Red;
}
```

```
//有作用域枚举
enum class Color {
  Red,
  Green,
  Yellow
};
int main() {
  Color x = Color::Red;
}
```

```
enum Color : int;//没有int是不合法的。
因为系统不知道enum的大小
enum Color {// 定义
  Red,
  Green
};

//对于有作用域枚举来说，
enum class Color;//声明就是可以通过编译的

//通常来说大家不会用枚举的定义和声明分开。
只有在一些特殊的情况会用。这个知识点可以
暂时不记
```

```
enum Type {
  Char,
  Int,
};
union Str {
  char x;
  int y;
};

struct S {
  Type t;
  Str obj;
};
int main() {
  S s;
  s.t = Char;
  s.obj.x = 'c';
}
//每次使用联合之前都去查一下
type的类型。然后根据type选择
正确的类型

//改进版本
struct S {
  enum Type {
    Char,
    Int,
   };
  union { //匿名联合，把x和y作为
S的数据成员，但是这个x和y共享内存
    char x;
    int y;
  };
  Type t;
};
int main() {
  S s;
  s.t = S::Char;
  s.x = 'c';//可以简化代码编写
}
```

嵌套类与局部类

只有在类内定义成员函数的时候会打破编译从上到下的运行顺序

- 嵌套类：在类中定义的类
  - 嵌套类具有自己的域，与外围类的域形成嵌套关系
    - 嵌套类中的名称查找失败时会在其外围类中继续查找
  - 嵌套类与外围类单独拥有各自的成员

使用场景：我们要定义一个类，这个类需要一些小的数据结构来辅助它做一些工作。(这些数据结构比较简单，只被这个类使用。并且放到外面可能引发名称冲突。)，那么可以尝试使用嵌套类

```
//嵌套类
class Out {
public:
  class In {
  public:
    inline static int val = 3;
  };
};

int main() {
  Out::In::val;
}
```

- 局部类：可以在函数内部定义的类
  - 可以访问外围函数中定义的类型声明、静态对象与枚举
  - 局部类可以定义成员函数，但成员函数的定义必须位于类内部
  - 局部类不能定义静态数据成员

```
局部类
void fun() {
  struct Helper {
    int x;
    int y;
  };
  Helper h;
}
```

嵌套名字空间与匿名名字空间

- 嵌套名字空间
  - 名字空间可以嵌套，嵌套名字空间形成嵌套域
  - 注意同样的名字空间定义可以出现在程序多处，以向同一个名字空间中增加声明或定义
  - C++17 开始可以简化嵌套名字空间的定义

```
namespace Out::In {
  int z;
}
```

- 匿名名字空间
  - 用于构造仅翻译单元可见的对象
  - 可用 static 代替
  - 可作为嵌套名字空间

```
namespace {
  int y;
}

int main() {
}
```

```
namespace MyNs {
  namespace {
    int x;//内部命名嵌套名字空间。只在当前文件可见
  }
}
int main() {
  MyNs::x = 3;
}
```

位域与 volatile 关键字

```
struct A {
  bool b1 : 1; //表示b1只使用1位来保存，一个比特
  bool b2 : 1;
}; (1个字节表示8位)
//因为这两个被放到一个字节了，节省了内存。那么在读取的时候，就需要花费额外的时间
```

- 位域：显示表明对象尺寸（所占位数）
  - 在结构体 / 类中使用
  - 多个位域对象可能会被打包存取
  - 声明了位域的对象无法取地址，因此不能使用指针或非常量引用进行绑定
  - 尺寸通常会小于对象类型所对应的尺寸，否则取值受类型限制

- volatile关键字
  - 表明一个对象的可能会被当前程序以外的逻辑修改
  - 相应对象的读写可能会加重程序负担
  - 注意慎重使用——一些情况下可以用 atomic 代替

深蓝学院
shenlanxueyuan.com

感谢聆听！

Thanks for Listening