Peter the Great
Saint-Petersburg Polytechnic University

Наука Программирования

Занятие №3
«Приведение типов. Механизм исключений. Задача «разбор командной строки»

Осенний семестр 2017
Преподаватель: асс. каф. Чуканов В.С

18.09.17

Содержание

- Исключения
- Исключения в конструкторах и деструкторах
- Операторы приведения типов
- Постановка задачи: разбор аргументов командной строки
- Описание базового класса переменной
- Описание произвольной переменной
- Менеджер переменных

Исключения

- Механизм исключений
  - Регламентирует редко случающиеся ситуации, влекущие радикальные изменения в текущем поведении программы либо прекращение ее работы
- Код, генерирующий исключения
  - Помещается в блок try { }
  - Исключения создаются вызовом throw [exception_object];
- Код, обрабатывающий исключения
  - Помещается в блок catch {}

Объекты-исключения

- Семантика
  - Данные об ошибке
- std::exception
  - Базовый класс исключения в стандартной библиотеке
  - Все исключения стд. библиотеки наследованы от std::exception
- Объекты-исключения
  - В качестве объекта-исключения может выступать любой класс
  - Классы исключений могут быть организованы в иерархию

```
try
{
    m_task_desc->memory.h_pinned_memmgr.reset(new mem_ops::memory_manager_pinned);
}
catch (const std::exception &e)
{
    LOG_STREAM << "[ERROR] Pinned memory exception occured: ";
    LOG_STREAM << e.what();
    LOG_STREAM << "Pinned memory manager will be reset to NULL. \n";
    m_task_desc->memory.h_pinned_memmgr.reset();
}
```

Обработка исключений

- catch блоки обрабатываются в порядке объявления
- catch (…) { }
  - Ловит любое исключение
  - Должен быть объявлен последним
- catch (BaseClass &b)
  - Должен быть объявлен после всех наследников

Обработка исключений

- Обработка исключения = развертка стека вызовов
- При возникновении исключения поиск обработчика заканчивается во внешнем блоке try .. catch, в который «обернут» main (winmain)
  - Попадание в этот блок ведет к вызову функции terminate()
- Возникновение исключения во время развертки стека = вызов terminate()

Исключения в конструкторах и деструкторах

- Исключение в конструкторе
  - Объект не является созданным, и тело деструктора не будет вызвано
  - Деструкторы предков и полей вызываются в стандартном порядке
- Исключение в деструкторе
  - При возникновении на этапе развертки стека вызовов приведет к вызову terminate()
- Следует избегать ОБА ВАРИАНТА использования исключений

Операторы приведения типов

- type_to_cast a = static_cast <type_to_cast> (b);
  - Приведение объекта b к типу type_to_cast
  - Проверка уровня компиляции
- type_to_cast *a = dynamic_cast <type_to_cast*> (b);
  - Используется для приведения типов с проверкой в run-time (RTTI, run-time type info)
  - При неудаче вернет NULL
- type_to_cast &a = dynamic_cast <type_to_cast&> (b);
  - В случае ошибки порождает исключение std::bad_cast
- const_cast
  - Снимает модификаторы const и volatile
- reinterpret_cast
  - Максимально небезопасное приведение типов
  - Конвертация указателя в int, любого типа в любой другой

Ассоциативный контейнер map

- std::map
- Отсортированная структура данных, состоящая из пар ключ-значение
  - Реализует красно-черное дерево
- Тип «ключа» должен иметь оператор сравнения
- Операции (ключ = std::string)
  - Объявление std::map<std::string, int> mymap;
  - Добавление mymap[“firstVal”] = 10;
  - Поиск
    - *std*::*map*<*std*::*string*, int>::*iterator* – тип итератора по контейнеру
    - *std*::*map*<*std*::*string*, int>::*iterator it = mymap.find(“firstVal”);*
    - *it == mymap.end()* – верно в случае отсутствия элемента в контейнере

Разбор аргументов командной строки

- Параметры ком. строки
  - Пара «имя» «значение»
  - Значение может быть произвольного типа
  - Значение считывается из строки
- Задача: разработать класс для обработки командной строки
  - Регистрация имени новой переменной
  - Установка значения по умолчанию для переменной
  - Возврат значения по имени переменной
  - Заполнение значений переменных по массиву argV и кол-ву аргументов argC

Класс переменной

- Пара «имя» - «значение»
- Переменная = значение
- Класс переменной
  - Унифицированный интерфейс для хранения значения любого типа
  - Интерфейс для извлечения значения любого типа из строки

Базовый класс переменной

- Любой унифицированный интерфейс = базовый класс

```
class variable_base
{
public:
    /*!
     * \brief A method for extracting a value from string.
     * \param var_string String with value to extract
     * \return true if parsing was successful
     */
    virtual bool set_value(const std::string& var_string) = 0;
};
```

Класс переменной: реализация

- Потомок = шаблон
- Для каждого фиксированного типа реализуется set_value()
- Хранение любых наследников возможно по указателю на базовый класс

```
template <typename T>
class variable: public variable_base
{
public:
    variable(const T& init_value)
    {
        m_value = init_value;
    }

    bool set_value(const std::string& var_string);

    T get_value() const
    {
        return m_value;
    }

private:
    T m_value;
};
```

Менеджер переменных

- Поля

```
//! Map of registered variables
std::map<std::string, variable_base*> m_vars;
//! Map iterator type
typedef std::map<std::string, variable_base*>::iterator
                                                variable_iterator;
//! Pair type
typedef std::pair<std::string, variable_base*> variable_pair;
```

Регистрация новой переменной

- Тип неизвестен = шаблонный метод
- Тип специализации класса-переменной = типу специализации метода регистрации переменной

```
template <typename T>
void register_variable(const std::string& var_name, const T& val)
{
    variable_pair p( var_name, new variable<T>(val) );
    m_vars.insert( p );
}
```

Получение значения переменной

- Для указания значения переменной необходимо указать тип
  - Метод - шаблонный
- Алгоритм
  - Поиск по имени переменной
  - Тип специализации метода = тип специализации наследника variable
  - Приведение к типу наследника
    - dynamic_cast

Получение значения переменной: реализация

```
template <typename T>
bool get_variable(const std::string& var_name, T* var_value)
{
    variable_iterator var_iter = m_vars.find(var_name);
    if (var_iter == m_vars.end())
        return false;
    variable<T> *var_ptr = dynamic_cast< variable<T>* >(var_iter->
second);
    if (var_ptr == NULL)
      return false;
    *var_value = var_ptr->get_value();
    return true;
}
```

Закрытый метод set_value

- Поиск переменной и установка значения

```
bool variable_mgr::set_value(const std::string& var_name, const
std::string& var_string)
{
    variable_iterator var_iter = m_vars.find(var_name);
    if (var_iter == m_vars.end())
        return false;
    return var_iter->second->set_value(var_string);
}
```

Парсинг командной строки

```
void variable_mgr::run_parser(int argc, char* argv[])
{
    int var_id = 1;
    while ( var_id < argc - 1 )
    {
        if (!set_value(argv[var_id], argv[var_id + 1]))
        {
            std::string msg = "ERROR: Either unidentified option or
missing parameter: ";
            msg += argv[var_id];
            throw err::parse_exception(msg);
        }
        var_id += 2;
    }
}
```

Указание реализаций set_value

- Компилятору необходимо указать реализации для всех используемых в коде специализаций variable

```
template <>
bool variable<float>::set_value(const std::string& var_string)
{
    m_value = static_cast<float> (strtod( var_string.c_str(),
NULL));
    return true;
}

template <>
bool variable<std::string>::set_value(const std::string&
var_string)
{
    m_value = var_string;
    return true;
}
```

Заключение

- Исключения
  - Механизм детектирования и обработки ошибок
  - Необходимо избегать возникновения исключений в к-торах и д-торах
- Приведение типов
  - static_cast – проверка только на уровне компиляции (ближе к C-style)
  - dynamic_cast – проверка в run-time
- Составление контейнеров объектов разных типов
  - Требует дополнительной иерархии
  - Наследник = шаблон
  - Хранение по указателю на базовый класс
  - Может требовать dynamic_cast