Ввод, вывод. Файлы

1 Введение

Итак, программисты C++ знают, что память нужно освобождать. Желательно всегда. И они знают, что если где-то пишется new, обязательно должен быть соответствующий delete. Но ручные манипуляции с памятью могут быть чреваты, например, следующими ошибками:

- утечки памяти;
- разыменовывание нулевого указателя, либо обращение к неициализированной области памяти;
- удаление уже удаленного объекта;

Существует техника управления ресурсами посредством локальных объектов, называемая RAII. То есть, при получении какого-либо ресурса, его инициализируют в конструкторе, а, поработав с ним в функции, корректно освобождают в деструкторе. Ресурсом может быть что угодно, к примеру файл, сетевое соединение, а в нашем случае блок памяти. Вот простейший пример:

Листинг 1: Пример

```
class VideoBuffer {
    int *myPixels;
public:
    VideoBuffer() {
        myPixels = new int[640 * 480];
    }
    void makeFrame() { /* work with rastr */ }
    ~VideoBuffer() {
        delete [] myPixels;
    }
};
int game() {
    VideoBuffer screen;
    screen.makeFrame();
}
```

Это удобно: по выходу из функции нам не нужно заботиться об освобождении буфера, так как для объекта screen вызовется деструктор, который в свою очередь освободит инкапсулированный в себе массив пикселей. Конечно, можно написать и так:

Листинг 2: Пример

```
int game() {
    int *myPixels = new int[640 * 480];
    // some work
    delete [] myPixels;
}
```

В принципе, никакой разницы, но представим себе такой код:

Листинг 3: Пример

```
int game() {
    int *myPixels = new int[640 * 480];
    // some work
    if (someCondition)
        return 1;
    // some work
    if (someCondition)
        return 4;
    // end of work
    delete [] myPixels;
}
```

Придется в каждой ветке выхода из функции писать delete [], либо вызывать какие-либо дополнительные функции деинициализации. А если выделений памяти много, либо они происходят в разных частях функции? Уследить за всем этим будет все сложнее и сложнее. Ок, будем использовать RAII, в конструкторах инициализировать память, в деструкторе освобождать. И пусть поля нашего класса будут указателями на участки динамической памяти:

Листинг 4: Пример

```
class Foo {
    int *data1;
    double *data2;
    char *data3;
public:
    Foo() {
        data1 = new int(5);
        ...
    }
    ~Foo() {
        delete data1;
        ...
    }
}
```

Теперь представьте, что полей не 3, а 30, а значит в деструкторе придется для всех них вызывать delete. А если мы второпях добавим новое поле, но забудем его убить в деструкторе, то последствия будут негативными. В итоге получается класс, нагруженный операциями выделения/освобождения памяти, да еще и непонятно, все ли было правильно удалено.

Поэтому предлагается использовать умные указатели: это объекты, которые хранят указатели на динамически аллоцированные участки памяти произвольного типа. Причем они автоматически очищают память по выходу из области видимости.

Сначала рассмотрим то, как они выглядят в C++, затем перейдем к обзору некоторых распространенных типов умных указателей.

2 Простейший smart pointer

Листинг 5: Пример

```
// our smart pointer class
template <typename T>
class smart_pointer {
    T *m_obj;
public:
    // some object for this class
    smart_pointer(T *obj)
        : m_obj(obj)
```

```
    { }
    // On leaving the field of view, we will destroy this object
    ~smart_pointer() {
        delete m_obj;
    }
    // overdrive operators<
    // Selector. Allows access to data of type T by means of "arrow"
    T* operator->() { return m_obj; }
    // With the help of such an operator, we can dereference the pointer
    //and get a reference to the object that it stores
    T& operator* () { return *m_obj; }
}
int test {
    // We return myClass to the smart pointer
    smart_pointer<MyClass> pMyClass(new MyClass(/*params*/);
    // We address to a method of class MyClass by means of the selector
    pMyClass->something();
    //Assume that for our class there is a function to output it to ostream
    // This function usually gets a reference to an object by one
    //of the parameters,
    //which should be displayed
    std::cout << *pMyClass << std::endl;
    // on the output of the skoupe, the MyClass object will be deleted
}
```

Понятно, что наш смарт пойнтер не лишен недостатков (например, как хранить в нем массив?), но он в полной мере реализует идиому RAII. Он ведет себя так же, как и обычный указатель (благодаря перегруженным операторам), причем нам не нужно заботиться об освобождении памяти: все будет сделано автоматически. По желанию к перегруженным операторам можно добавить const, гарантировав неизменность данных, на которые ссылается указатель. Теперь, если вы поняли, что получаете определенные преимущества, при использовании таких указателей, рассмотрим их конкретные реализации. Если вам не нравится эта идея, то все равно, попробуйте использовать их в какой-нибудь своей маленькой программке. Итак, наши смарт-пойнтеры:

- boost::scope_ptr;
- std::auto_ptr;
- std::tr1::shared_ptr;

3 boost::scoped_ptr

Он находится в библиотеке буст. Реализация простая и понятная, практически идентичная нашей, за несколькими исключениями, одно из них: этот пойнтер не может быть скопирован (то есть у него приватный конструктор копирования и оператор присваивания). Поясню на примере:

Листинг 6: Пример

```
#include <boost/scoped_ptr.hpp>
int test() {
    boost::scoped_ptr<int> p1(new int(6));
    boost::scoped_ptr<int> p2(new int(1));
    p1 = p2; // we cant!
}
```

Оно и понятно, если бы было разрешено присваивание, то и p1 и p2 будут указывать на одну и ту же область памяти. А по выходу из функции оба удалятся. Что будет? Никто не знает. Соответственно, этот пойнтер нельзя передавать и в функции.

Тогда зачем он нужен? Советую применять его как указатель-обертка для каких-либо данных, которые выделяются динамически в начале функции и удаляются в конце, чтобы избавить себя от головной боли по поводу корректной очистки ресурсов.

4 std::auto_ptr

Чуть-чуть улучшенный вариант предыдущего, к тому же он есть в стандартной библиотеке. У него есть оператор присваивания и конструктор-копировщик, но работают они несколько необычно.

Листинг 7: Пример

```
#include <memory>
int test() {
    std::auto_ptr<MyObject> p1(new MyObject);
    std::auto_ptr<MyObject> p2;
    p2 = p1;
}
```

Теперь при присваивании в p2 будет лежать указатель на MyObject (который мы создавали для p1), а в p1 не будет ничего. То есть p1 теперь обнулен. Это так называемая семантика перемещения. Кстати, оператор копирования поступает таким же образом. Зачем это нужно? Ну например у вас есть функция, которая должна создавать какой-то объект:

Листинг 8: Пример

```
    std::auto_ptr<MyObject> giveMeMyObject();
```

Это означает, что функция создает новый объект типа MyObject и отдает его вам в распоряжение. Понятней станет, если эта функция сама является членом класса (допустим Factory): вы уверены, что этот класс (Factory) не хранит в себе еще один указатель на новый объект. Объект ваш и указатель на него один. В силу такой необычной семантики auto_ptr нельзя использовать в контейнерах STL. Но у нас есть shared_ptr.

5 std::shared_ptr

Умный указатель с подсчетом ссылок. Что это значит. Это значит, что где-то есть некая переменная, которая хранит количество указателей, которые ссылаются на объект. Если эта переменная становится равной нулю, то объект уничтожается. Счетчик инкрементируется при каждом вызове либо оператора копирования либо оператора присваивания. Так же у shared_ptr есть оператор приведения к bool, что в итоге дает нам привычный синтаксис указателей, не заботясь об освобождении памяти.

Листинг 9: Пример

```
#include <memory>
#include <iostream>
int test() {
    std::shared_ptr<MyObject> p1(new MyObject);
    std::shared_ptr<MyObject> p2;
    p2 = p1;
    if (p2)
        std::cout << "Hello, world!\n";
}
```

Теперь и p2 и p1 указывают на один объект, а счетчик ссылок равен 2, По выходу из скоупа счетчик обнуляется, и объект уничтожается. Мы можем передавать этот указатель в функцию:

Листинг 10: Пример

```
int test(std::shared_ptr<MyObject> p1) {
    // doing something
}
```

Заметьте, если вы передаете указатель по ссылке, то счетчик не будет увеличен. Вы должны быть уверены, что объект MyObject будет жив, пока будет выполняться функция test.

6 Источники

- https://www.boost.org/doc/libs/1_39_0/libs/smart_ptr/scoped_ptr.htm
- https://habr.com/post/140222/
- https://tproger.ru/problems/write-a-class-for-smart-pointer/

Содержание