Тема 2. Умные указатели

В чем проблема

Одной из неприятных проблем при работе с языком C++ являются ошибки памяти при использовании указателей. Эти ошибки обычно обусловлены следующими причинами:

— обращение по неинициализированному или неверно инициализированному указателю;

— обращение по указателю, который уже не соответствует выделенной ранее памяти;

— повторное освобождение уже ранее освобожденной памяти;

— память по указателю не освобождается.

В результате этих ошибок программа может либо выдать отказ segmentation fault при попытке использовать недоступные для вас участки памяти, либо работать с неправильными данными, если указатель показывает на вашу память, но не туда, куда требуется (например, разрушить стек системный вызовов), либо нерационально использовать память в случае несвоевременного ее обсвобождения.

Все эти ошибки так или иначе связаны с выделением памяти в одном месте программы и последущей передачей этих указателей для работы в другие места программы. В такой ситуации весьма сложно бывает отслеживать когда именно и где такую память надо освобождать и сохраняет ли указатель свою корректность.

В силу важности и общности пролемы контроля за состонием памяти, да и любых других ресурсов, например, файлов, сетевых соединений и т.п., C++ вводит концепцию RAII (Resource Acquisition Is Initialization), которая представляет собой идею связывания жизненного цикла ресурса с жизненным циклом объекта в C++. А именно, ресурсы должны захватываться/выделяться при создании и освобождаются при уничтожении объектов.

Как ее решать

В простейшем случае класс в своем конструкторе захватывает память или, скажем, открывает файл, а в деструкторе ее освобождает (закрывает). Тогда доступ к ресурсу (памяти, файлу) будет корректен, пока объект существует в своей области видимости, а затем будет корректно завершен с уничтожением объекта (причем как нормальным, так и аварийным в результате, например, исключения).

Вот простой пример, класс-обертка вокруг указателя на файл

```
class FILEPTR {
    FILE *f;
public:
    FILEPTR(FILE *_f = nullptr) { f = _f; }
    FILE * operator=(FILE *_f) { if (f) fclose(f); f = _f; return f; }
    FILE * operator*() { return f; }
    operator FILE*() { return f; }
    operator bool() { return (f != nullptr); }
    ~FILEPTR() { if (f) fclose(f); }
};
```

С таким “указателем” можно безбоязненно писать код типа

```
FILPTR f = fopen(.....);
if (!f) return;
......
if (....) return;
```

```
if (....) return;
if (....) return;
......
```

не заботясь о том, что надо бы закрывать открытый файл перед каждой попыткой выхода по return при выполнении той или иной из указанных проверок. И в конце кода тоже нет необходимости закрывать файл, так как он автоматически закроется при уничтожении переменной f при выходе из области ее видимости.

Идея, воплощенная в предыдущем примере, была доведена до обособленной стандартной реализации в языке C++11 в виде так называемых умных указателей (smart pointers). Эти объекты ведут себя так же как и обычные указатели, но освобождают ресурс (вызывают delete на себя) при выходе из зоны видимости. Однако указатели можно присваивать друг другу, возвращать через return и т.д. и возникает вопрос, а в какой момент уничтожение переменной-указателя должно приводит к освобождению ресурса, поскольку мы можем иметь несколько указателей, полученных друг их друга присваиванием.

Язык C++11 вводит три типа умных указателей, как раз исходя из того какая дисциплина определяет момент освобождения ресурса.

```
template <class T>
 unique_ptr<T>  - монопольное владение ресурсом
 shared_ptr<T>  - совместное владение ресурсом
 weak_ptr<T>    - обертка вокруг shared_ptr для специальных задач
```

Рассмотрим эти указатели по порядку.

unique_ptr

Этот указатель инициализируется в своем конструкторе и далее запрещает как другие присваивания в себя, так и присваивание своего значения другим указателям. При выходе из зоны его видимости вызываептся delete для хранящегося в нем обычного указателя на выделенный ресурс. Таким образом, мы создаем этот указатель, пользуемся им монопольно, и ресурс, стоящий за ним освобождается автоматически. Так как мы этот указатель не можем менять, то ресурс не может быть утрачен в результате неверного присваивания.

Для создания такого указателя используется либо его конструктор, либо специальный шаблонный макрос `make_unique<>`.

```
unique_ptr<int> a(new int(10));            // *a есть 10
unique_ptr<int> b = make_unique<int>(20);  // *b есть 20
int x = *a; // можно
*b = 123;   // можно
a = b;      // нельзя
b = &x;     // нельзя
```

Но что интересно, поскольку unique_ptr используется для уникального доступа, то мы имеем право передать это доступ другому, используя семантику перемещения

```
a = move(b);  // a и b  поменяются местами
```

С использованием такой техники можно передавать `unique_ptr` параметром в функцию по значению.

Вот пример, иллюстрирующий некоторые способы использования `unique_ptr`. Умные указатели — это классы, имеющие свои функции-методы. В частности, метод `get()` для `unique_ptr` возвращает обычный константный указатель на исходный ресурс. Здесь он используется, чтобы увидеть “истинный” адрес ресурса.

```
#include<iostream>
#include<memory>      // для использования умных указателей
#include<vector>

using namespace std;

// класс, чтобы понять работу конструкторов-деструкторов
struct XX {
   int *px;
   XX(int v)  { px = new int(v); cout << "XX("<<v<<")\n"; }
   ~XX()  { cout << "~XX("<<*px<<")\n"; delete px; }
};

// функции просто печатают истинный адрес
// и значение, которое по этому адресу хранится

void f(unique_ptr<int> p)  // передача в функцию по значению
{
    cout << "f(): ";
    if (p.get()) cout << p.get() << " " << *p;
    cout << endl;
}
void g(unique_ptr<int> &p)  // передача в функцию по ссылке
{
    cout << "g(): ";
    if (p.get()) cout << p.get() << " " << *p;
    cout << endl;
}
void h(unique_ptr<int> &&p)  // передача в функцию по rvalue
{
    cout << "h(): ";
    if (p.get()) cout << p.get() << " " << *p;
    cout << endl;
}

// макрос печати для сокращения записи
#define print(s,p)   cout << s << p.get() << " " << (p?*p:0) << endl
#define printXX(s,p)   cout << s << p.get() << " " << (p? *(p->px):0) << endl

int main()
{
    // создаем через конструктор
    unique_ptr<int> p1(new int(10));
    print("p1 ",p1);

    // создаем через make_unique
    //unique_ptr<int> p2 = make_unique<int>(20);  // можно так
    auto p2 = make_unique<int>(20);               // а так проще
    print("p2 ",p2);

    //unique_ptr<int> p1 = p2;   - так запрещено

    // присваиваем через перемещение
    cout << "\n p1 = move(p2); -----------\n";
```

```
    p1 = move(p2);
    print("p1 ",p1);
    print("p2 ",p2);

    // то же самое с классами
    unique_ptr<XX> x1(new XX(100));
    unique_ptr<XX> x2(new XX(200));
    printXX("x1 ",x1);
    printXX("x2 ",x2);

    cout << "\n p1 = move(p2); -----------\n";
    x1 = move(x2);
    printXX("x1 ",x1);     // обратите внимание на деструктор
    printXX("x2 ",x2);

    // еще создание по существующему указателю
    cout << "\n from ordinary ptr -----------\n";
    int *q3 = new int(30);  // обычный указатель
    unique_ptr<int> p3 = make_unique<int>(*q3);
    unique_ptr<int> p4(q3);
    print("p3 ",p3);
    print("p4 ",p4);
    cout << "q3 " << q3 << " " << *q3 << endl;

    // передача в функцию
    cout << "\n function call -----------\n";
    unique_ptr<int> q1 = make_unique<int>(123);
    //f(q1);  - нельзя
    g(q1);
    print("q1 ",q1);
    f(move(q1));
    print("q1 ",q1);
    h(move(p1));
    print("p1 ",p1);

    // создание указателя на массив (delete [])
    unique_ptr<int[]> p5 = make_unique<int[]>(10);
    // auto p5 = ...

    return 0;
}
```

shared_ptr

Этот указатель дает возможность его копировать (присваивать другим) и использовать эти копии для работы в разных местах программы. Не вдаваясь в тонкости, можно говорить, что такой указатель дополнительно хранит количество ссылок на объект (количество копий указателя), которое увеличивается при каждом новом копировании/присваивании и уменьшается при уничтожении какой-либо переменной-копии. При достижении этим счетчиком нулевого значения, уже удаляется сам исходный объект (освобождается ресурс).

Создается `shared_ptr` точно так же как и `unique_ptr`, т.е. через конструктор или с помощью макроса `make_shared`.

Данный указатель также имеет несколько собственных методов, которые мы рас-

смотрим далее, а пока приведем более-менее нетривиальный содержательный пример использования такого указателя.

Рассмотрим стек на базе списка. У нас была подобная реализация, которая заключалась в поддержке однонаправленного списка узлов, ссылающихся последовательно друг на друга. Вершина стека соответствовала крайнему узле списка. Узлы добавлялись и удалялись с края списка, реализуя операции добавления и уделения элементов в стеке. Ссылки между узлами ранее были обычными указателями, а теперь мы их сделаем разделяемыми указателями, поскольку их придется переприсваивать в процессе работы со стеком. За счет использования умных указателей мы теперь сможем не заботиться об освобождении памяти под узлы, поскольку она должна освобождаться автоматически.

```
#include<iostream>
#include<memory>
#include<vector>
using namespace std;

class stack{
   struct node { // узел списка (с печатью что происходит)
       int value;
       shared_ptr<node> next;  // ссылка на следующий
       node(int x, shared_ptr<node> n) {
           value = x; next = n;
           cout << "node " << this << endl;
       }
       ~node() { cout << "~node " << this << endl; }
   };
   shared_ptr<node> top;  // указатель на вершину стека

public:
   stack() : top() { /*top = nullptr;*/ }

   void push(int x)  { top = make_shared<node>(x, top); }
   bool pop (int &x) {
      if (!top) return false;
      shared_ptr<node> p = top;
      x = top->value;
      top = top->next;
      return true;
   }
   friend ostream & operator<<(ostream &os, const stack &s);
};
// полная распечатка стека
ostream & operator<<(ostream &os, const stack &s)
{
    os << "stack\n";
    for (auto p = s.top; p; p = p->next) os << " " << p->value;
    os << "\n--------\n";
    return os;
}
// класс для демонстрации других свойств
struct sp {
   int x;
   sp(int xx) { x = xx; cout << "sp " << this << endl; }
   ~sp() {cout << "~sp " << this << endl; }
```

```
};
// функция, которая что-то делает с указателем
shared_ptr<sp> f(shared_ptr<sp> q) {
   (q->x)++;
   return q;
}

int main()
{
    int x;
    stack s;
    s.push(1);
    s.push(2);
    s.push(3);
    s.push(4);
    cout << s;
    s.pop(x);
    s.pop(x);
    cout << s;

    // создание и присваивание с функцией
    cout << "\ncreate and assign ------------\n";
    shared_ptr<sp> p1(new sp(10));
    shared_ptr<sp> p2;
    p2 = f(p1);

    // массив указателей
    cout << "\narray of pointers ------------\n";
    vector<shared_ptr<sp>> v(10);
    for (int i=0; i<5; i++) {  // меньше чем длина !!!
       v[i] = make_shared<sp>(i);
    }
    return 0;
}
```

Запустив эту программу, можно увидеть в какой момент вызваются конструкторы и деструкторы соответствующих объектов, которые управляются через умные указатели.

weak_ptr

С разделяемым указателем `shared_ptr` возникает одна серьезная проблема обработки циклических зависимостей. Классический пример здесь — два класса, которые содержат внутри ссылки друг на друга в виде `shared_ptr`. В этом случае каждый класс не может быть уничтожен при выходе из области видимости потому, что другой класс содержит ссылку на него через разделяемый указатель. Вот простейший вариант этого примера. Чтобы увидеть, что дектрукторы не срабатывают, мы их определилия явно с выводом соответствующего сообщения.

```
struct A {
   shared_ptr<A> other;
   ~A() { cout << "~A()\n"; }
};
int main()
{
```

```
    shared_ptr<A> px(new A);
    shared_ptr<A> py(new A);
    px->other = py;   // shared = shared - увеличение количества ссылок
    py->other = px;
    return 0;
}
```

Теперь заменим `shared_ptr` на `weak_ptr` во взаимных ссылках.

```
struct A {
   weak_ptr<A> other;
   ~A() { cout << "~A()\n"; }
};
int main()
{
    shared_ptr<A> px(new A);
    shared_ptr<A> py(new A);
    px->other = py;   // weak = shared - увеличесния количества ссылок нет
    py->other = px;
    return 0;
}
```

Вот в этом случае оба деструктора отрабатывают нормально.

Итак, `weak_ptr` — это просто обертка над `shared_ptr`, которая при появлении не увеличивает количество ссылок, хранящихся в исходном `shared_ptr`. Соответственно уничтожение `weak_ptr` не уменьшает количество ссылок на исходный объект из `shared_ptr`. У этих типов указателей есть метод `use_count()` который возвращает количество ссылок на исходный объект. Если в предыдущий код вставить строчки типа

```
    cout << px.use_count() << endl;
    cout << py.use_count() << endl;
```

то можно увидеть как меняется количество ссылок в первом и втором вариантах.

Но тут нас поджидает другая проблема. Так как `weak_ptr` существует независимо от `shared_ptr`, то вполне может случиться ситуация, когда мы вышли из области видимости `shared_ptr` (и он уничтожил свой исходный объект), но все еще находимся в области видимости `weak_ptr`, который в этом случае уже непонятно чему соответствует.

```
    weak_ptr<int> p;
    {
       shared_ptr<int> q(new int(100));
       p = q;
    }
    // здесь q уже нет, а p еще есть!
```

Эта проблема решается так. Во-первых, `weak_ptr` не позволяет напрямую обращаться к исходному объекту. То есть у него нет операторов `*` и `->`, а доступ к этому объекту можно получить только через `shared_ptr`, который возвращается методом `lock()`. Если соответствующий `shared_ptr` уже уничтожен то метод `lock()` вернет nullptr со всеми вытекающими последствиями. Кроме этого `weak_ptr` позволяет заренее проверить, что происходит с соответствующим `shared_ptr` с помощью методов `expired()` и уже упомянутого `use_count()`. Вот пример, иллюстрирующий применение этих методов.

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

using namespace std;

struct Link {
    int value;
    //shared_ptr<Link> other;
    weak_ptr<Link> other;
    Link(int v) { value = v; }
    ~Link() { cout << "~Link\n"; }
};
ostream & operator<<(ostream & os, const Link &s) {
//    os << "Link " << &s << " -> " << s.other.get() << " v=" << s.value << endl;
    os << "Link " << &s << " -> " << s.other.lock().get() << " v=" << s.value << endl
    return os;
}

int main()
{
    int x;
    shared_ptr<Link> p1(new Link(100));
    shared_ptr<Link> p2(new Link(200));
    cout << *p1;
    cout << *p2;
    p1->other = p2;
    p2->other = p1;
    cout << *p1;
    cout << *p2;
    cout << p1.use_count() << endl;
    cout << p2.use_count() << endl;

    weak_ptr<int> wp;
    {
         auto sp = make_shared<int>(123);
         wp = sp;
         cout << wp.lock().get();
         cout << " " << wp.use_count();
         cout << " " << *(wp.lock()) << endl;
    }
    cout << " " << wp.use_count() << endl;
    if (wp.expired()) cout << "expired\n";
    if (wp.lock() == nullptr) cout << "sp nullptr\n";

    return 0;
}
```