Запрос ресурсов путём инициализации

По материалам:

Бьерн Страуструп. Язык программирования C++. Второе дополненное издание. п. 9.4

Если в некоторой функции потребуются определённые ресурсы, например, нужно открыть файл, отвести блок памяти в области свободной памяти, установить монопольные права доступа и т.д., для дальнейшей работы системы обычно бывает крайне важно, чтобы ресурсы были освобождены надлежащим образом. Обычно такой "надлежащий способ" реализует функция, в которой происходит запрос ресурсов и освобождение их перед выходом. Например:

```
void use_file(const char* fn)
{
  FILE* f = fopen(fn,"w");

  // работаем с f

  fclose(f);
}
```

Все это выглядит вполне нормально до тех пор, пока вы не поймёте, что при любой ошибке, происшедшей после вызова fopen() и до вызова fclose(), возникнет особая ситуация, в результате которой мы выйдем из use_file(), не обращаясь к fclose().

Можно дать примитивное решение:

```
void use_file(const char* fn)
{
  FILE* f = fopen(fn,"w");
  try {
      // работаем с f
  }
  catch (...) {
      fclose(f);
      throw;
  }
  fclose(f);
}
```

Вся часть функции, работающая с файлом f, помещена в проверяемый блок, в котором перехватываются все особые ситуации, закрывается файл и особая ситуация запускается повторно. Недостаток этого решения в его многословности, громоздкости и потенциальной расточительности. К тому же всякое многословное и громоздкое решение чревато ошибками, хотя бы в силу усталости программиста. К счастью, есть более приемлемое решение. В общем виде проблему можно сформулировать так:

```
void acquire()
{
  // запрос ресурса 1
  // ...
  // запрос ресурса n

  // использование ресурсов

  // освобождение ресурса n
  // ...
  // освобождение ресурса 1
}
```

Как правило бывает важно, чтобы ресурсы освобождались в обратном по сравнению с запросами порядке. Это очень сильно напоминает порядок работы с локальными объектами, создаваемыми конструкторами и уничтожаемыми деструкторами. Поэтому мы можем решить проблему запроса и освобождения ресурсов, если будем использовать подходящие объекты классов с конструкторами и деструкторами. Например, можно определить класс FilePtr, который выступает как тип FILE* :

```
class FilePtr {
   FILE* p;
public:
   FilePtr(const char* n, const char* a)
     { p = fopen(n,a); }
   FilePtr(FILE* pp) { p = pp; }
   ~FilePtr() { fclose(p); }

   operator FILE*() { return p; }
};
```

Построить объект FilePtr можно либо, имея объект типа FILE*, либо, получив нужные для fopen() параметры. В любом случае этот объект будет уничтожен при выходе из его области видимости, и его деструктор закроет файл. Теперь наш пример сжимается до такой функции:

```
void use_file(const char* fn)
{
  FilePtr f(fn,"w");
  // работаем с f
}
```

Деструктор будет вызываться независимо от того, закончилась ли функция нормально, или произошёл запуск особой ситуации.

Описанный способ управления ресурсами обычно называют "запрос ресурсов путём инициализации". Это универсальный прием, рассчитанный на свойства конструкторов и деструкторов и их взаимодействие с механизмом особых ситуаций. Объект не считается построенным, пока не завершил выполнение его конструктор. Только после этого возможна раскрутка стека, сопровождающая вызов деструктора объекта. Объект, состоящий из вложенных объектов, построен в той степени, в какой построены вложенные объекты. Хорошо написанный конструктор должен гарантировать, что объект построен полностью и правильно. Если ему не удаётся сделать это, он должен, насколько это возможно, восстановить состояние системы, которое было до начала построения. Для простых конструкторов было бы идеально всегда удовлетворять хотя бы одному условию - правильности или законченности объектов, и никогда не оставлять объект в "наполовину построенном" состоянии. Этого можно добиться, если применять при построении членов способ "запроса ресурсов путём инициализации".

Рассмотрим класс X, конструктору которого требуется два ресурса: файл x и замок y (т.е. монопольные права доступа к чему-либо). Эти запросы могут быть отклонены и привести к запуску особой ситуации. Чтобы не усложнять работу программиста, можно потребовать, чтобы конструктор класса X никогда не завершался тем, что запрос на файл удовлетворён, а на замок нет. Для представления двух видов ресурсов мы будем использовать объекты двух классов

FilePtr и LockPtr (естественно, было бы достаточно одного класса, если x и y ресурсы одного вида). Запрос ресурса выглядит как инициализация представляющего ресурс объекта:

```
class X {
  FilePtr aa;
  LockPtr bb;
  // ...
  X(const char* x, const char* y)
     : aa(x),      // запрос `x'
     bb(y)         // запрос `y'
  { }
  // ...
};
```

Теперь, как это было для случая локальных объектов, всю служебную работу, связанную с ресурсами, можно возложить на реализацию. Пользователь не обязан следить за ходом такой работой. Например, если после построения aa и до построения bb возникнет особая ситуация, то будет вызван только деструктор aa, но не bb. Это означает, что если строго придерживаться этой простой схемы запроса ресурсов, то все будет в порядке. Еще более важно то, что создателю конструктора не нужно самому писать обработчики особых ситуаций.

Для требований выделить блок в свободной памяти характерен самый произвольный порядок запроса ресурсов. Примеры таких запросов уже неоднократно встречались в этой книге:

```
class X {
  int* p;
  // ...
public:
  X(int s) { p = new int[s]; init(); }
  ~X() { delete[] p; }
  // ...
};
```

Это типичный пример использования свободной памяти, но в совокупности с особыми ситуациями он может привести к

исчерпанию памяти. Действительно, если в init() запущена особая ситуация, то отведённая память не будет освобождена. Деструктор не будет вызываться, поскольку построение объекта не было завершено. Есть более надёжный вариант этого примера:

```
template<class T> class MemPtr {
public:
   T* p;
   MemPtr(size_t s) { p = new T[s]; }
   ~MemPtr() { delete[] p; }
   operator T*() { return p; }
}

class X {
   MemPtr<int> cp;
   // ...
public:
   X(int s):cp(s) { init(); }
   // ...
};
```

Теперь уничтожение массива, на который указывает p, происходит неявно в MemPtr. Если init() запустит особую ситуацию, отведённая память будет освобождена при неявном вызове деструктора для полностью построенного вложенного объекта cp. Отметим также, что стандартная стратегия выделения памяти в C++ гарантирует, что если функции operator new() не удалось выделить память для объекта, то конструктор для него никогда не будет вызываться. Это означает, что пользователю не надо опасаться, что конструктор или деструктор может быть вызван для несуществующего объекта.

Не все программы должны быть устойчивы ко всем видам ошибок. Не все ресурсы являются настолько критичными, чтобы оправдать попытки защитить их с помощью описанного способа "запроса ресурсов путём инициализации". Есть множество программ, которые просто читают входные данные и выполняются до конца. Для них самой подходящей реакцией на динамическую ошибку будет просто прекращение счета (после выдачи соответствующего сообщения). Освобождение всех затребованных ресурсов возлагается на систему, а пользователь должен произвести повторный запуск программы

с более подходящими входными данными. Наша схема предназначена для задач, в которых такая примитивная реакция на динамическую ошибку неприемлема. Например, разработчик библиотеки обычно не в праве делать допущения о том, насколько устойчива к ошибкам, должна быть программа, работающая с библиотекой. Поэтому он должен учитывать все динамические ошибки и освобождать все ресурсы до возврата из библиотечной функции в пользовательскую программу. Метод "запроса ресурсов путём инициализации" в совокупности с особыми ситуациями, сигнализирующими об ошибке, может пригодиться при создании многих библиотек.