**Объекты-функции.**

Это альтернатива указателям на функции, которые слишком неуклюжи, опасны и давно устарели. Как правило. Объект-функция – это обычный объект класса, в котором перегружен оператор вызова функции (), чтобы создать «прокачанный указатель на функцию». Рассмотрим объект-функцию, вычисляющий при каждом вызове следующий элемент общеизвестного ряда Фибоначчи (1, 1, 2, 3, 5, 8, 13 …).

class Fib

{

int m\_a0, m\_a1;

public:

Fib() : m\_a0(0), m\_a1(1) {}

int operator()()

{

int temp = m\_a0;

m\_a0 = m\_a1;

m\_a1 = m\_a0 + temp;

return m\_a0;

}

};

int \_tmain(int argc, \_TCHAR\* argv[])

{

Fib f;

for (int i = 0; i < 10; i++)

cout << f() << ' '; // Первые 10 чисел ряда Фибоначчи,

// f() - выглядит как вызов функции.

// Но на самом деле это вызов оператора для объекта f.

getchar();

return 0;

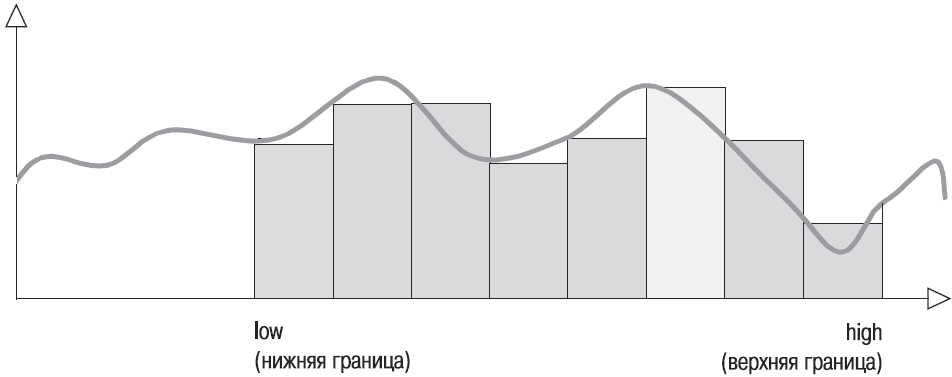
}

Компилятор распознаёт синтаксис f() как вызов функции-члена operator() переменной f. В данном случае преимущество объекта-функции перед функцией или указателем на функцию заключается в том, что состояние, необходимое для вычисления следующего элемента ряда Фибоначчи, хранится в самом объекте f. Функция тоже может иметь состояние, например, благодаря локальным или глобальным статическим переменным. Возможно, состояние в функцию пришлось бы передавать явно. Кроме того, в отличии от функции, использующей статические переменные, можно иметь одновременно несколько объектов f, осуществляющих вычисления независимо друг от друга.

NB: То есть если имеется только функция, сохраняющая своё состояние во внутренних статических переменных, нельзя её использовать для нескольких независимых вычислений. А с помощью объектов-функций можно. Если создать с десяток таких объектов, то каждый из них будет хранить своё состояние, и для каждого имеется перегруженный оператор ().

Вторым достоинством применения объектов-функций является возможность создать эффект указателя на виртуальную функцию, образуя иерархию объекта-функции с помощью виртуального оператора operator().

*Пример*. Вычисление интеграла методом аппроксимации площади под кривой, как показано на следующем рисунке:



Для приближенного вычисления площади под кривой, как суммы площадей прямоугольников функция интегрирования будет итеративно вызывать функцию для значений, находящихся между нижней и верхней границами.

1. Классический метод, с помощью указателя на функцию.

// Функция для значений интеграла

typedef double (\*fv)(double);

// Просто для примера

double y(double x)

{

return x \* x;

}

// Функция интегрирования

double Integral(fv y, double low, double high)

{

const int numSteps = 8;

double step = (high-low)/numSteps;

double area = 0.0;

while(low < high)

{

area += y(low)\*step; // Площадь прямоугольника

low += step;

}

return area;

}

Такой вариант работоспособен, но не гибок, потому что интегрируемая функция y(x) обозначается при помощи указателя на функцию (fv). Альтернатива – создание иерархии объекта-функции. Базовым классом иерархии будет простой интерфейсный класс, объявляющий чистую виртуальную функцию operator().

// Абстрактный класс с виртуальным оператором ()

class Func

{

public:

virtual double operator() (double x) = 0;

// Если имеется хотя бы одна виртуальная функция,

// необходимо объявить деструктор также виртуальным

virtual ~Func() {};

};

double Integral2(Func &y, double low, double high)

{

const int numSteps = 8;

double step = (high-low)/numSteps;

double area = 0.0;

while(low < high)

{

area += y(low)\*step; // Площадь прямоугольника

low += step;

}

return area;

}

// Объект-функция, которую будем интегрировать

// Конструируется на основе имеющейся функции

class fy : public Func

{

double (\*m\_fv)(double);

public:

fy(double (\*y)(double)) : m\_fv(y) {}

virtual ~fy(){}

virtual double operator() (double x)

{

return m\_fv(x);

}

};

int \_tmain(int argc, \_TCHAR\* argv[])

{

// 2. Классическое вычисление интеграла

cout << "area = " << Integral(y, 0, 10) << endl;

// 3. Использование объекта функции

cout << "area = " << Integral2(fy(y), 0.0, 10.0) << endl;

// NB: При передаче параметра создаётся анонимный объект (fy(y))!

getchar();

return 0;

}

Теперь функция Integral2 способна интегрировать объект-функцию любого типа, который является экземпляром класса Func. Примечателен тот факт, что тело функции Integral2 совершенно не изменилось, потому что синтаксис вызова объекта-функции тот же, что и для указателя на функцию.

Функции-члены можно интегрировать, заключая указатель на функцию-член и объект класса в соответствующий интерфейс-обёртку.

NB: таким образом, можно любые функции наделять состоянием. Конструируется объект, в полях которого сохраняется указатель на эту функцию, а также состояние для этой функции. Чтобы вызвать эту функцию, перегружается оператор ().

NB: класс, который реализует operator(), называется функтором.

Также можно интегрировать функции-члены (NB: других классов), заключая указатель на функцию-член и объект класса в соответствующий интерфейс-обёртку.

NB: Допустим имеется некоторый класс C, для которого необходимо интегрировать его функцию-член f. Тогда необходимо конструировать объект-обёртку с учётом знания о классе C. Принцип тот же, что и для обычной функции, но дополнительную гибкость даёт использование шаблона, так как это позволяет создавать обёртки для многих классов, функции-членов которых необходимо интегрировать.

// Пусть имеется некоторый класс, содержащий функцию y(x), которую необходимо интегрировать

class Curve

{

public:

double y(double x) {return x \* x;}

};

// Обертка

template <class C> class Y : public Func

{

// Указатель на функцию член

double (C::\*m\_py)(double);

// Объект необходим для вызова функции члена

C &m\_obj;

public:

Y(double (C::\*py)(double), C &obj) : m\_py(py), m\_obj(obj) {}

double operator()(double x)

{

return (m\_obj.\*m\_py)(x);

}

virtual ~Y(){} // деструктор наследника разрушает только свою часть (на заметку)

};

// 5. Интегрирование функции-члена

cout << "area = " << Integral2(Y<Curve>(&Curve::y, Curve()), 0.0, 10.0) << endl;

**Паттерн «Команда».**

Если объект-функция выступает в качестве функции обратного вызова (callback), то это экземпляр паттерна Команда (Command).

Функции обратного вызова применяются, когда «каркас» знает, когда следует сделать что-то, но не знает, что сделать, в то время, когда «клиент каркаса» знает что делать в случае конкретного события, но не знает когда это делать. Вместе они образуют единое целое. Обратные вызовы – распространённая техника программирования, которая обычно реализуется через простые указатели на функции.

Рассмотрим интерактивный тип Button, отображающий на экране кнопку с надписью, и выполняющий действие при щелчке по ней.

NB: класс знает когда необходимо выполнить действие, но не знает о самом действии ничего. Паттерн «Команда» вполне годится.

class Button

{

void (\*m\_pAction)(void);

const string &m\_label;

public:

Button(const string &label) : m\_pAction(nullptr), m\_label(label) {}

void SetAction(void (\*pAction)(void))

{

m\_pAction = pAction;

}

void OnClick() const

{

if (m\_pAction)

m\_pAction();

}

};

void PlayMusic()

{

cout << "playing the music ... " << endl;

}

// в main.cpp

Button b("play");

b.SetAction(PlayMusic);

// ...

b.OnClick(); // вызывает зарегестрированный callback

Пользователь Button задаёт функцию обратного вызова и затем передаёт управление Button коду каркаса, который может определять момент нажатия кнопки и выполнять действие. Такое разделение обязанностей часто называют принципом Голливуда: «не звоните нам, мы сами вам позвоним». Мы настраиваем кнопку на выполнение определённого действия в случае нажатия. Код каркаса знает, что это действие должно быть инициировано при нажатии кнопки.

Применение простого указателя на функцию в качестве callback имеет строгие ограничения. Функции часто обрабатывают некоторые данные, но с указателем на функцию не ассоциированы никакие данные. Например, функция PlayMusic() не может знать какую песню ей воспроизводить. Как правило, вместо указателя на функцию лучше применять объект-функцию. Объект-функция (или иерархия объектов-функций) в сочетании с принципом Голливуда образует экземпляр паттерна «Команда».

Очевидное преимущество объектно-ориентированного подхода состоит в том, что объект-функция может инкапсулировать данные. Другой положительный момент – объект-функция может обладать динамическим поведением, посредством виртуальных членов. То есть возможна иерархия взаимосвязанных объектов-функций. Доработаем Button в контексте паттерна «Команда».

// Класс, инкапсулирующий действие, и данные, необходимые для его совершения

class Action

{

public:

virtual ~Action() {}; // деструктор должен быть виртуальным, т.к.

// иначе он не вызывается по указателю на базовый класс

virtual void operator()() = 0; // чисто виртуальный оператор

virtual Action \*clone() const = 0; // чисто виртуальный метод

};

// Создадим конкретное действие

class PlayMusic : public Action

{

const string &m\_song;

public:

PlayMusic(const string &song) : Action(), m\_song(song) {}

virtual ~PlayMusic(){}

// необходимо определить оператор ()и метод clone()

virtual void operator()()

{

cout << "playing: " << m\_song.c\_str() << endl;

}

virtual Action \*clone() const

{

return new PlayMusic(m\_song);

}

};

// Класс, с которым работает каркасс

class Button2

{

const string &m\_label;

Action \*m\_pAction; // это и есть команда

public:

Button2(const string& label) : m\_label(label), m\_pAction(nullptr) {}

~Button2()

{

if (m\_pAction)

{

delete m\_pAction;

m\_pAction = nullptr;

}

}

// Установка объекта-функци в качестве команды

void SetAction(const Action \*pAction)

{

Action \*p = pAction->clone(); // вернёт указатель на объект в куче

if (p)

{

if (m\_pAction) // если ранее был создан такой объект, уничтожаем его

delete m\_pAction;

m\_pAction = p; // сохраняем указатель на новый объект

}

}

// Метод, вызываемый каркасом для активаци команды

void OnClick(void)

{

if (m\_pAction)

(\*m\_pAction)();

}

};

// где то в main.cpp

Button2 b2("play 2");

string song("Urban hits");

b2.SetAction(&PlayMusic(song)); // song не должен быть анонимным, иначе он

// уничтожится после данного вызова

b2.OnClick(); // вызов действия (объекта-функции)

Теперь Button может работать с любым объектом-функцией, представляющим собой Action. Данные, необходимые для выполнения действия теперь инкапсулируются в том же объекте, что и само действие.

Наличие иерархии паттерна «Команда» позволило сочетать паттерн «Прототип» с паттерном «Команда» для того, чтобы создавать клонируемые команды.

**Объекты-функции STL.**

STL позволяет в некоторых ситуациях определять альтернативную операцию, подобную «меньше чем». Это имеет место быть, когда невозможно сравнивать объекты как «больше» или «меньше», без специального контекста. Такие операции называются компараторами, потому что в них сравниваются два значения. Создав компаратор для данного класса, его можно использовать для сортировки. При этом в функцию сортировки передаётся указатель на функцию сравнения. Для ускорения вычислений компаратор реализуют как inline, но при передаче его в качестве аргумента в функцию сортировки, он не будет встроен в неё, так как имя функции-компаратора преобразуется в указатель на функцию. В итоге страдает скорость сортировки. Ситуацию можно исправить, применив в качестве компаратора объект-функцию.

// Класс, для которого вводится объект-функция компаратор

class State

{

int m\_nPop;

public:

int population() const

{

return m\_nPop;

}

State(): m\_nPop(0){}

State(int pop): m\_nPop(pop){}

};

// функция сравнения

inline bool popLess(const State &a, const State &b)

{

return a.population() < b.population();

}

// класс объекта-функции компаратора

struct PopLess : public binary\_function<State, State, bool>

{

bool operator() (const State &a, const State &b)

{

return (a.population() < b.population());

}

};

// 8. Применение компаратора

State state\_a = 100, state\_b = 200;

cout << "a < b = " << PopLess()(state\_a, state\_b); // опять же, анонимный объект :)

PopLess – это характерный пример правильно построенного объекта-функции STL. Во-первых, это объект-функция. Он перегружает оператор (), поэтому может быть вызван с помощью обычного синтаксиса вызова функции. Это важно, потому что универсальные алгоритмы STL, такие как sort, написаны таким образом, что для создания их экземпляра может использоваться как указатель на функцию, так и объект-функция, при условии, что к ним применим обычный синтаксис вызова функции.

Во-вторых, он является производным от стандартного базового класса binary\_function. Этот механизм позволяет другим частям реализации STL обращаться к объекту-функции во время компиляции. В данном случае это обеспечивает возможность получать информацию о типе аргумента и возвращаемом типе объекта-функции (хотя в данном примере эта возможность не задействована).

В-третьих, у объекта-функции нет данных членов, нет виртуальных функций, нет прямо объявленных конструкторов или деструктора, а реализация () – встраиваемая. Выступающие в качестве компараторов STL объекты-функции должны быть компактными, простыми и быстрыми.

Теперь можно осуществить сортировку с помощью объекта-функции sort(states, states+50, PopLess()). Положительный эффект применения объекта-функции в качестве компаратора заключается в том, что сравнение будет встроено (inline), тогда как в случае применения указателя на функцию этого сделать нельзя. При создании экземпляра шаблона функции компилятору будет известен тип компаратора (PopLess), что, в свою очередь, даёт ему информацию о будущем вызове PopLess::operator() и обеспечивает возможность встроить эту функцию, используя встраиваемый вложенный вызов popLess.

Ещё один распространённый случай применения объектов-функций STL – использование в качестве предикатов. Предикат – это логическая операция в отношении одного объекта (компаратор можно рассматривать как разновидность бинарного предиката).

*Пример*.

struct IsWarm : public unary\_function<State, bool>

{

bool operator(const State &a) const

{

return a.aveTempF() > 60;

}

};

Рекомендации по разработке предикатов STL аналогичны рекомендациям по компараторам STL, за исключением того, конечно, что предикаты STL представляют собой унарные, а не бинарные функции.