Объектно-ориентированное программирование с использованием языка C++

ШАГ компьютерная АКАДЕМИЯ

Урок №17

Объектно-ориентированное программирование с использованием языка C++

Содержание

Использование функторов

Как вам известно из прошлых уроков, в библиотеке STL существует понятие функторов. Сейчас мы познакомимся с этим понятием поближе.

Функторы (иначе говоря, функциональные объекты) — это специализированный вид классов, которые включают в себя перегруженный оператор вызова функции. Как правило, функтор можно применить везде, где требуется функция. Проще говоря, когда должна быть вызвана функция, вызывается перегруженный оператор вызова функции.(оператор "круглые скобки").

Основным отличием функтора от обычной функции, является возможность хранения некоторого значения по принципу статических переменных. Принцип работы таков:

Первое обращение к функтору возбуждает вызов конструктора, который инициализирует сам функтор. При использовании обычной функции приходится хитрить, чтобы выполнить начальную инициализацию. Кроме того, некоторые источники говорят о том, что вызов функтора выполняется гораздо быстрее, чем вызов обычной функции с помощью указателя.

Рассмотрим пример, который позволит нам более детально разобраться в назначении функторов. Реализуем таблицу умножения. Требования следующие: первый множитель увеличивается слева направо, второй — сверху вниз. Это классический пример на алгоритм итерации.

Разберем алгоритм выполнения задания:

- Генерация строки с набором частных для одного числа и повторение генерации для нескольких чисел.
  - Инициализация контейнера-списка (или вектора).
  - Заполнение контейнера-списка (или вектора).
  - Вывод значений контейнера-списка (или вектора) на экран.
- Создание функтора, который должен последовательно генерирует число, начиная от заданного, и каждый раз увеличивает его на определенное значение.

```
//библиотека для работы алгоритмов
//мы еще будем рассматривать её более подробно сегодня
#include <algorithm>
#include <iostream>
//библиотека контейнера списка
#include <list>
using namespace std;

/*
Функциональный объект, содержащий два поля:
1. для хранения значения приращения (delta)
2. для текущего значения генерируемого числа (current)
*/
class addNumberFrom
{
    int delta;
        int current;
/*
Конструктор класса инициализирует значение приращения
и текущее значение.
Последнее может быть опущено, и тогда оно будет
считаться равным 0
*/
```



```
public:
    addNumberFrom(int number, int from = 0):delta(number),
                  current(from) {}

/*
Основа функтора — перегруженный оператор вызова
функции — прибавляет значение приращения
к текущему генерируемому числу
*/

    int operator()()
    {
        return current += delta;
    }
};

//Вывод заголовка для таблицы умножения.

void main()
{
    cout<<"TABLE:"<<"\n\n";
    cout << "----------" <<"\n\n";

    for(int i=1; i<=10; i++)
    {
        //Создание контейнера-списка.
        list<int> l(10);

        /*
        Вызов алгоритма generate_n. Естественно,
        сам он не может ничего генерировать,
        однако последовательно перебирает значения,
        диапазон которых задан начальным
        итератором и количеством элементов списка.
        Для записи числа в каждое значение вызывается
        функция, на которую ссылается третий параметр:
        */
```

```
        generate_n(l.begin(), l.size(), addNumberFrom(i));

        /*
        Но мы вместо функции подставляем перегруженный
        оператор вызова функции — объект addNumberFrom.
        Если вызов происходит впервые, то вызывается
        конструктор объекта. Он инициализирует поле
        delta значением переменной i,а поле current -
        значением по умолчанию второго параметра
        конструктора, т. е. 0.

        Таким образом, контейнер-список заполняется
        произведениями числа в переменной i и
        множителями от 1 до 10. В алгоритме generate_n
        используется метод size(), который возвращает
        количество элементов в списке.
        Если имеются начальный и конечные итераторы,
        тогда лучше воспользоваться алгоритмом
        generate.
        */

        //Собственно показ числа из контейнера-списка

        copy(l.begin(), l.end(),
                  ostream_iterator<int>(cout,"\t"));

    }
}
```

Надеемся, пример помог вам понять функторы более глубоко. Теперь отметим, что зачастую различные арифметические и логические операции, а также операции сравнения могут реализовываться стандартными средствами STL, то есть набором уже существующих, встроенных функторов:

1. ***Арифметические функторы:***
   - **plus** сложение **x + y**;
   - **minus** вычитание **x – y**;
   - **multiplies** умножение **x × y**;
   - **divides** деление **x / y**;
   - **modulus** взятие остатка **x % y**;
   - **negate** обращение знака **–x**;
2. ***Функторы сравнения:***
   - **equal_to** равно **x == y**;
   - **not_equal_to** не равно **x != y**;
   - **greater** больше **x > y**;
   - **less** меньше **x < y**;
   - **greater_equal** больше или равно **x => y**;
   - **less_equal** меньше или равно **x <= y**;
3. ***Логические функторы:***
   - **logical_and** логическое "и" **x && y**;
   - **logical_or** логическое "или" **x || y**;
   - **logical_not** логическое "не" **! x.**

Использование предикатов

И снова STL... Еще одно понятие, которое мы с Вами не рассматривали детально — предикаты.

Зачастую, вам могут понадобиться специальные средства, которые принимают решения в зависимости от ситуации и координируют выполнение какой-либо вашей программы. Обычно для этих целей мы используем логические выражения языка программирования. Однако, в STL принято решать данную проблему иным способом. А именно, создавать предикаты.

Предикат — специальная функция, которая возвращает логическое значение (true либо false).

Наверное, вы уже догадались, что создать предикат достаточно легко — нужно просто написать функцию которая возвращает тип bool. Рассмотрим пример. Попробуем создать предикат, определяющий четность числа. Если число окажется четным, предикат будет возвращать true. Используем следующий порядок действий:

- Четность определяется получением остатка от деления на 2.
- Затем, заполняется контейнер-список значениями от 1 до 10. (для этого вызывается метод **push_back()**, который добавляет значение в конец списка)
- Затем значения выводятся на экран.
- Далее осуществляется поиск и выделение всех четных значений.

Примечание: Эту задачу решает алгоритм remove_if. Обратите внимания, что многие алгоритмы, оканчивающиеся на _if, в качестве последнего параметра используют предикаты. Алгоритм remove_if сдвигает все подходящие значения в начало контейнера и возвращает итератор, который указывает на элемент, следующий за удаляемыми значениями. По этой причине, мы будем считать, что начиная с адреса, на который ссылается возвращаемый итератор, и до конца области данных контейнера располагаются неудаленные значения, для которых предикат вернул false. Здесь, в нашем примере, мы используем данную особенность следующим образом: все четные значения выводятся на экран (для этого мы копируем их в поток вывода, передав потоковому итератору пару итераторов, которые описывают область удаленных значений.)

```
#include <algorithm>
#include <iostream>
#include <list>
using namespace std;

//предикат
bool isEven(int num)
{
    return bool(num % 2);
}

void main()
{
    list<int> l;
        list<int>::iterator t;
```

```
    for(int i=1; i<=10; i++)
        l.push_back(i);

    copy(l.begin(), l.end(),
        ostream_iterator<int>(cout," "));
    cout<<"\n\n";
    t=remove_if(l.begin(), l.end(), isEven);
    copy(l.begin(), t, ostream_iterator<int>(cout," "));
}
```


Использование алгоритмов

Как вы уже наслышаны, в библиотеке STL существуют функции, которые выполняют различные стандартные действия. Действия эти, например, — поиск, преобразование, сортировка, копирование и так далее. Эти функции называются алгоритмами. В качестве параметрами для алгоритмов принято использовать итераторы.

Следует отметить, что алгоритму абсолютно всё равно, какого типа итератор, ему передали. Самое главное, чтобы этот итератор попадал в определенную группу. Например, если параметром алгоритма должен быть однонаправленный итератор, то передаваемый итератор должен быть либо однонаправленным, либо двунаправленным, или же итератором произвольного доступа.

Основная информация об алгоритмах

Все алгоритмы делятся на две группы: те, которые изменяют данные, и те, которые их не изменяют.

Каждый алгоритм представляет собой шаблон функции или набор шаблонов функций. То есть, любой алгоритм может работать с абсолютно разными контейнерами.

Алгоритмы, возвращающие итератор, обычно, для сообщения о неудаче используют конец входной последовательности.

Алгоритмы не выполняют проверки диапазона на их входе и выходе.

Если алгоритм возвращает итератор, это будет итератор того же типа, что был на входе в алгоритм.

Алгоритмы определены в заголовочном файле <algorithm>.

Приведем наиболее используемые алгоритмы библиотеки STL.

1. ***Немодифицирующие операции:***
   - **for_earch()** — выполняет операции для каждого элемента последовательности;
   - **find()** — находит первое вхождение значения в последовательность;
   - **find_if()** — находит первое соответствие предикату в последовательности;
   - **count()** — подсчитывает количество вхождений значения в последовательность;
   - **count_if()** — подсчитывает количество выполнений предиката в последовательности;
   - **search()** — находит первое вхождение последовательности как подпоследовательности;
   - **search_n()** — находит в последовательности подпоследовательность, состоящую из n повторений и возвращает её первое вхождение.
2. ***Модифицирующие операции:***
   - **copy()** — копирует последовательность, начиная с первого элемента;
   - **swap()** — меняет местами два элемента;
   - **replace()** — заменяет элементы с указанным значением;
   - **replace_if()** — заменяет элементы при выполнении предиката;

   - **replace_copy()** — копирует последовательность, заменяя элементы с указанным значением;
   - **replace_copy_if()** — копирует последовательность, заменяя элементы при выполнении предиката;
   - **fill()** — заменяет все элементы данным значением;
   - **remove()** — удаляет элементы с данным значением;
   - **remove_if()** — удаляет элементы при выполнении предиката;
   - **remove_copy()** — копирует последовательность, удаляя элементы с указанным значением;
   - **remove_copy_if()** — копирует последовательность, удаляя элементы при выполнении предиката;
   - **reverse()** — меняет порядок следования элементов на обратный;
   - **random_shuffle()** — перемещает элементы согласно случайному равномерному распределению ("тасует" последовательность);
   - **transform()** — выполняет заданную операцию над каждым элементом последовательности;
   - **unique()** — удаляет равные соседние элементы;
   - **unique_copy()** — копирует последовательность, удаляя равные соседние элементы.
3. ***Сортировка:***
   - **sort()** — сортирует последовательность с хорошей средней эффективностью;
   - **partial_sort()** — сортирует часть последовательности;
   - **stable_sort()** — сортирует последовательность, сохраняя порядок следования равных элементов;

- **lower_bound()** — находит первый элемент, меньший чем заданное значение;
- **upper_bound()** — находит первый элемент, больший чем заданное значение;
- **binary_search()** — определяет, есть ли данный элемент в отсортированной последовательности;
- **merge()** — сливает две отсортированные последовательности.

4. ***Работа с множествами:***
   - **includes()** — проверка на вхождение;
   - **set_union()** — объединение множеств;
   - **set_intersection()** — пересечение множеств;
   - **set_difference()** — разность множеств.
5. ***Минимумы и максимумы:***
   - **min()** — меньшее из двух;
   - **max()** — большее из двух;
   - **min_element()** — наименьшее значение в последовательности;
   - **max_element()** — наибольшее значение в последовательности.
6. ***Перестановки:***
   - **next_permutation()** — следующая перестановка в лексикографическом порядке;
   - **prev_permutation()** — предыдущая перестановка в лексикографическом порядке.

Понятие сложность алгоритма

Сегодня мы с вами уже познакомились с понятием алгоритмы. Очень часто появляется необходимость выбора оптимального алгоритма из набора алгоритмов определенного вида. При этом следует учитывать ограничения на размер исходных данных решаемой задачи и параметры операционной системы, частоту процессора, объем памяти.

Для выяснения истины, в теории рассматриваются алгоритмы решения не конкретных, а так называемых, массовых задач. Массовая задача представляется как бесконечная серия индивидуальных задач. Индивидуальная же задача характеризуется объемом входных данных, требуемых для решения этой задачи. При этом, если размер индивидуальной задачи — некое натуральное число n, то сложность алгоритма решения массовой задачи становится функцией от n.

Для начала, рассмотрим алгоритм простого перебора всех двоичных ключей длины n. Поскольку очевидно, что количество таких ключей — 2 в степени n, то в данном алгоритме 2 в степени n шагов. Иначе говоря, сложность этого алгоритма равна 2 в степени n. Такой алгоритм является вариантом простого перебора и называется ***экспоненциальным алгоритмом***.

Рассмотрим теперь алгоритм умножения столбиком двух n-значных чисел. Он состоит из n в степени 2 умножений однозначных чисел, и его сложность, измеренная

количеством таких умножений, естественно равна n в степени 2. Такой алгоритм называют — ***полиномиальный алгоритм***.

Всё вышеописанное является лишь достаточно простой теорией и только даёт нам понять, что правильность это отнюдь не единственное качество, которым должна обладать грамотная программа. Эффективность, характеризующая время выполнения программы для различных входных данных — это еще одно важное преимущество программы.

Нахождение точной зависимости для конкретной программы это очень сложная задача. По этой причине обычно ограничиваются асимптотическими оценками конкретного алгоритма, то есть описанием его примерного поведения при больших значениях основного параметра. Иногда для асимптотических оценок используют традиционное отношение O (O большое) между двумя функциями. Например, функция f(n) = n2/2 + 3n/2 + 1 возрастает приблизительно так же как n2/2 (мы отбрасываем медленно растущее слагаемое 3n/2+1). Константный множитель 1/2 также отбрасываем и получаем асимптотическую оценку для алгоритма, которая обозначается как O(n2).

Здесь мы приведем сравнительную таблицу времен выполнения алгоритмов с различной сложностью и разберемся, почему с увеличением быстродействия компьютеров важность использования быстрых алгоритмов значительно возрасла.

Рассмотрим четыре алгоритма решения одной и той же задачи, имеющие сложности log n,n ,n в степени 2 и 2 в степени n. Предположим, что второй из этих алгоритмов

требует для своего выполнения на неком компьютере при значении опередеенном параметра (например 10 в степени 3) ровно одну минуту времени. В этом случае время выполнения этих четырех алгоритмов на том же компьютере будут выглядеть примерно так:

Сложность алгоритма	n=10	$n=10^3$	$n=10^6$
log n	0.2 сек.	0.6 сек.	1.2 сек.
n	0.6 сек.	1 мин.	16.6 ч.
n^2	6 сек.	16.6 ч.	1902 г.
2^n	1 мин.	10^{295} лет	10^{300000} лет

Итак, подведем итог. С увеличением быстродействия компьютеров возрастают и значения параметров, для которых работа того или иного алгоритма завершается за приемлимое время. Таким образом, увеличивается среднее значение величины, и, следовательно, возрастает величина отношения времен выполнения быстрого и медленного алгоритмов. Чем быстрее компьютер, тем больше относительный проигрыш при использовании плохого алгоритма!