Косвенная адресация и ссылки

Косвенной адресацией, называется способ хранения данных, при котором в переменной записано не конкретное значение, а информация о его местоположении. В C++ механизмы косвенной адресации присутствуют в трех основных типах объектов: указатели, ссылки, итераторы.

Работа с указателями в заданиях этого курса не предусмотрена, поэтому они будут рассмотрены позже.

Ссылка в C++ — это тип данных, переменная которого хранит адрес другой переменной и инициализируется при объявлении. После инициализации всякие операции над ссылочной переменной выполняются над объектом, на который она ссылается. Переменные ссылочного типа объявляются при помощи описателя ссылок `&`. Рассмотрим пример.

```
int x=10;
int &y=x;
y++;
cout<<x<<" "<<y<<endl; //Вывод 11 11
x=20-x;
cout<<x<<" "<<y<<endl; //Вывод 9 9
x+=y;
cout<<x<<" "<<y<<endl; //Вывод 18 18
```

Таким образом, ссылка может рассматриваться как псевдоним для существующей переменной.

В большинстве случаев ссылки используются при передаче параметров в функции. Классический пример — это функция `swap`, выполняющая обмен значений двух переменных.

```
void swap(int &x,int &y) {
  int t=x;
  x=y;
  y=t;
}
int main() {
  int a=1,b=2;
  swap(a,b);
  cout<<a<<" "<<b;//Вывод 2 1
}
```

В переменные `x` и `y` заносятся адреса переменных `a` и `b`, в результате все присваивания в теле функции `swap` происходят с ними. Такой способ передачи параметров в функции называется "по ссылке".

Если из объявления функции `swap` убрать описатели ссылок `&`, то параметры будут передаваться по значению. Тогда, в результате присваиваний изменятся значения переменных `x` и `y`, но это никак не повлияет на `a` и `b`.

Таким образом, ссылки используются для того, чтобы позволить исполняемой функции изменить значение переменной вызывающей функции. Но это не единственное их применение. Важно понимать, что ссылка занимает несколько байт и ее передача в функцию происходит значительно быстрее чем передача объектов большого размера, например, векторов. Поэтому, передавать такие объекты в функции практически всегда следует по ссылке. Для того, чтобы показать, что объект передаваемый в функцию является неизменяемым, можно использовать ключевое слово `const`. Например, правильная декларация функции, принимающей на вход вектор и не изменяющей его, должна выглядеть так.

```
void any_function(const vector<int> &V)
```

Итераторы

Последовательность элементов называется итерируемой, если каждый элемент в ней может быть найден или построен по значению или местоположению предыдущего элемента. Из определения следует, что это понятие итерируемой последовательности является очень общим. В него входят, например, натуральные числа, данные записанные в файл, элементы массива и так далее. Фактически, всякая последовательность элеменов в программировании является итерируемой.

Итератором называется объект, косвенно адресующий один элемент итерируемой последовательности, и позволяющий перемещаться по ней. Тем самым, алгоритмы, использующие итераторы, способны работать с различными видами последовательностей, что позволяет существенно расширить диапазон их применения.

Чаще всего итераторы в C++ применяются для работы с содержимым контейнеров. Далее все примеры будут использовать контейнеры типа `vector`. Для того, чтобы объявить итератор на элемент контейнера требуется воспользоваться вложенным типом `iterator`, например,

```
vector<double>::iterator it;
```

здесь объявлен итератор на элемент вектора вещественных чисел.

Каждый контейнер содержит две функции для получения итераторов: `begin()` и `end()`, — возвращающие итераторы на начало и конец последовательности. Важно помнить, что концом любой последовательности является фиктивный элемент, доступ к которому запрещен. Таким образом, диапазон `[begin(),end())` является полуоткрытым. Все алгоритмы стандартной библиотеки C++ унифицированы под работу с полуоткрытыми диапазонами.

Если итератор может быть проинициализован при создании, то для его объявления можно использовать ключевое слово `auto`. Например, цикл по всем элементам контейнера `V` через итератор может быть организован следующим образом.

```
for (auto it=V.begin();it!=V.end();++it)
```

Ключевое слово `auto` в C++ указывает, что тип переменной следует определить автоматически по типу выражения, расположенного справа от знака равенства.

Все итераторы поддерживают четыре основных операции:

- `*` — доступ к адресуемому элементу;
- `++` — смещение к следующему элементу итерируемой последовательности;
- `==` — сравнение итераторов, возвращающее истину при условии что оба итератора указывают на один и тот же элемент итерируемой последовательности;
- `!=` — сравнение итераторов, обратное к операции `==`.

Например, цикл вывода всех элементов произвольного контейнера `V` может выглядеть так.

```
for (auto it=V.begin();it!=V.end();++it)
  cout<<*it<<endl;
```

Имеется возможность перебирать элементы в обратном порядке. Для этого можно использовать итераторы, возвращаемые функциями `rbegin()` и `rend()`. Например, для вывода элементов вектора в обратном порядке можно использовать такой цикл.

```
for (auto it=V.rbegin();it!=V.rend();++it)
  cout<<*it<<endl;
```

В связи с тем, что циклы по всем элементам контейнера очень распространены, для них существует более компактная форма записи, называемая цикл по коллекции.

```
for (auto x:V)
  cout<<x<<endl;
```

Здесь в `x` последовательно заносятся все элементы коллекции `V`. Переменная в таком цикле может иметь ссылочный тип. В этом случае можно будет изменять элемент контейнера, при условии, что он допускает эту возможность. Например, стандартный цикл ввода последовательности целых чисел может выглядеть так.

```
int n;
cin>>n;
vector<int> V(n);
for (auto &x:V)
  cin>>x;
```

Стандартная библиотека C++ определяет пять классов итераторов, из которых выжнейшими являются два: двунаправленные итераторы и итераторы произвольного доступа. Они различаются перечнем дополнительных операций. Двунаправленные итераторы помимо базовых, реализуют операцию `--` для перемещения по последовательности в обратном направлении. Почти все контейнеры C++ предоставляют именно двунаправленные итераторы.

Наиболее мощным классом итераторов являются итераторы произвольного доступа. Они дополнительно реализуют следующие операции:

- `it[n]` — индексация позволяет получить `n`-тый элемент относительно текущего;
- `it␣+␣n` — сложение с целым числом позволяет сместить итератор на заданное количество элементов;
- `it1-it2` — разность итераторов позволяет узнать количество элементов между ними.
- `it1<it2` — операции сравнения позволяют определить взаимное положение итераторов (слева, справа).

В векторах реализованы именно итераторы произвольного доступа.

Структуры

Структуры относятся к пользовательским типам данных. Это означает, что описание таких типов присутствует в программе или в ее библиотеках. Переменная структурного типа позволяет хранить в себе несколько значений различных типов, называемых полями. Перечень и количество полей заносится в описание структурного типа данных. В следующем примере определен структурный тип `point` с тремя полями.

```
struct point {
  double x;
  double y;
  int color;
};
```

Теперь можно определять переменные типа `point`, причем в каждой из переменных будет три поля с конкретными значениями. Для доступа к полям используется специальная операция `.`, как показано в примере.

```
point a,b,y;
a.x=10.0;
a.y=-5.5;
a.color=255;
b={-0.5,0.5,0};
```

```
y=a;
cout<<y.x<<" "<<y.y<<" "<<y.color;
```

Присвоить значение переменной можно списком инициализации в фигурных скобках. Также для переменных структурного типа автоматически реализуется операция присваивания. Не следует путать имя переменной и название поля. В приведенном примере имя поля `y` никак не конфликтует таким же с именем переменной.

Не существует никаких ограничений на работу с пользовательскими типами. Без ограничений они могут быть элементами контейнеров, например, вектора. Все алгоритмы работают с ними также как и с базовыми типами.

Вместе с тем для работы базовых алгоритмов с пользовательскими типами должны быть определены требуемые операции. Например, для работы алгоритма сортировки необходима операция сравнения. Для этого язык C++ поддерживает простой механизм перегрузки операторов. Требуется просто определить функцию с определенным именем и набором параметров. Рассмотрим пример перегрузки операции сравнения для типа `point`.

```
bool operator<(const point &a,const point &b) {
    if (a.color<b.color)
        return true;
    else if (a.color>b.color)
        return false;
    else
        return a.x*a.x+a.y*a.y<b.x*b.x+b.y*b.y;
}
```

Функция имеет имя `operator<` и определяет соответствующую операцию. Результатом сравнения должно быть логическое значение `bool`, а аргументами — две точки. Аргументы передаются по ссылке, так как это является более эффективным. Ключевое слово `const` указывает, что аргументы не изменяются в ходе работы функции. С одной стороны это может предотвратить некоторые ошибки программиста, с другой стороны делает функцию более общей, так как теперь аргументами функции могут быть также и неизменяемые, и временные объекты.

Реализация функции может быть произвольной, однако, следует понимать, что алгоритмы, использующие операцию сравнения, делают определенные допущения о ее работе, а именно, предполагают, что она удовлетворяет аксиомам линейного порядка.

- Для любых a и b операция $a < b$ однозначно определена.
- $a < b$ ложно тогда и только тогда, когда $b \leq a$ — истинно.
- $a \leq a$.
- $a \leq b$ и $b \leq a$ тогда и только тогда, когда $a = b$.
- Если $a \leq b$ и $b \leq c$ то $a \leq c$.

Легко проверить, что приведенный выше вариант операции сравнения удовлетворяет аксиомам линейного порядка. Рассмотрим еще один пример.

```
bool operator<(const point &a,const point &b) {
    if (a.color<b.color)
        return true;
    else if (a.x<b.x)
        return true;
    else
        return false;
}
```

Можно заметить, что для `p={0.5,0.5,255}` и $q = 1.0, 1.0, 127$ при заданной операции сравнения выполняется одновременно $p < q$ и $q < p$, следовательно, данная операция не удовлетворяет аксиомам линейного порядка, и, использование такого метода сравнения, например, с функцией сортировки приведет к непредсказуемой работе алгоритма.

Пары и кортежи

Для упрощения программ существует два стандартных вида структур: пара и кортеж. Пара (`pair`) это шаблонный класс с двумя параметрами: первый и второй элемент пары. Их требуется указать при объявлении переменных.

```
pair<int,double> a;
pair<long long,int> b;
```

Имена полей пары являются стандартными: `first` и `second`. Создать пару можно списком инициализации в фигурных скобочках или при помощи функции `make_pair`.

```
pair<int,double> a=make_pair(10,3.0);
pair<long long,int> b={100LL,0};
a.first=18;
b.second+=10;
```

Как правило, пары используется совместно с контейнерами. Наприиер можно создать вектор из 100 пар, проинициализировав его парой `{0,1}`;

```
vector<pair<int,int>> V(100,{0,1});
```

Для пар определены операции лексикографического сравнения. Это означает что результат сравнения определяется по первым элементам двух пар, а в случае их равентсва — по вторым.

Еще одной стандартной структурой является кортеж. Кортеж позволяет объединять несколько полей разных типов. Интерфейс создания кортежа похож на интерфейс пары, например

```
tuple<int,double,int,string> a,b,c;
a={10,2.5,18,"hello"};
b=make_tuple(18,3.7,8,"world");
c=a;
```

Также как и для пар, для кортежей определена операция лексикографического сравнения.

Основным отличием является интерфейс доступа к полям. Все поля пронумерованы, начиная с нуля. Доступ к полям осуществляется через шаблошшую функцию `verb<N>(x)`, где `N` — порядковый номер поля. Например,

```
get<0>(a)=get<2>(b)+10;
cout<<get<3>(c);
cin>>get<1>(b);
```

Для работы с кортежами требуется подключить библиотеку `tuple`.

Алгоритмы стандартной библиотеки C++

Все алгоритмы стандартной библиотеки C++ собраны в библиотеке `algorithm`. Все алгоритмы написаны при помощи итераторов, которые используются для задания диапазонов обрабатываемых элементов. Все диапазоны являются полуоткрытыми, то есть при указании диапазона `[b,e)` итератор `b` указывает на первый элемент обрабатываемой последовательности, а `e` — на элемент следующий за последним обрабатываемым элементом.

Линейный поиск

Линейный поиск — это алгоритм последовательного поиска нужного элемента в заданном диапазоне. Функция имеет три обязательных параметра: итераторы, указывающие диапазон поиска и искомое значение. Функция возвращает итератор на первый найденный элемент в случае успеха и итератор конца диапазона в случае неудачи. Далее пример использования функции для поиска позиции числа в векторе.

```
vector<int> V={8,12,0,-3,5,0,9,7,0,11};
int c=7;
auto it=find(V.begin(),V.end(),c);
if (it==V.end())
  cout<<"v not contain "<<c<<endl;
else
  cout<<"v contain "<<c<<" in position "<<it-V.begin()<<endl;
```

Разность итераторов равна количеству элементов между итераторами, таким образом, разность найденного итератора и `V.begin()` позволяет узнать порядковый номер найденного элемента.

Для поиска всех элементов `find` можно использовать в цикле.

```
vector<int> V={8,12,0,-3,5,0,9,7,0,11};
int c=0;
auto it=find(V.begin(),V.end(),c);
while (it!=V.end()) {
  cout<<"v contain "<<c<<" in position "<<it-V.begin()<<endl;
  ++it;
  it=V.find(it,V.end(),c);
}
```

В этом примере после каждого успешного поиска итератор смещается на один элемент вправо, и следующий поиск начинается с этой позиции.

Сортировка

Наиболее используемым алгоритмом стандартной библиотеки является функция сортировки. Два ее обязательных параметра это итераторы, указывающие диапазон сортируемых элементов. Как правило, сортировка применяется к вектору значений, а диапазоном являются все элементы вектора. Тогда вызов функции для вектора `V` может выглядеть так.

```
sort(V.begin(),V.end());
```

По умолчанию сортировка выполняется по возрастанию. Указать другой порядок можно несколькими способами. Если вектор содержит элементы произвольного пользовательского типа, то можно определить для него оператор сравнения `<`, как было показано выше. Кроме того, третьим параметром функции можно указать компаратор — специальный объект для сравнения двух элементов одного типа. Компаратором может быть произвольная функция с двумя параметрами, возвращающая логическое значение. Компаратор должен возращать истину, если после сортировки первый параметр должен быть левее второго.

Если `T` — это тип элемента то заголовок функции компаратора может выглядеть так.

```
bool cmp(const T &x, const T &y)
```

Другим вариантом для компаратора является лямбда-функция — анонимная функция, определяемая в месте использования. Описателем лямбда-функции являются квадратные скобки, как показано в примере.

```
vector<int> V={-5,-1,20,3,-8};
sort(V.begin(),V.end(),[](const int x,const int y) {
      if (x>=0 && y<0) return true;
      else if (x<0 && y>=0) return false;
      else return x<y;
      });
```

Здесь два первых условия в сравнении определяют, что все положительные числа будут размещены раньше чем отрицательные, а числа одного знака ддолжны быть упорядочены по возрастанию.

Если в компаратор требуется передать некоторый параметр для рассчетов, то его можно указать в квадратных скобках.

```
vector<int> V={10, 4, 3, 9, 11, 7};
int m=3;
stable_sort(V.begin(),V.end(),[m](const int x,const int y) {
       return x/m<y/m;
      });
```

В этом примере элементы, числа будут опорядочены по возрастанию, однако равными будут считаться те, результаты деления которых на `m` совпадут. При этом функция `stable_sort` сохраняет взаимное расположение равных элементов, поэтому после сортировки последовательность примет вид `4␣3␣7␣10␣9␣11`

Наконец, для простого изменения последовательности сортировки можно использовать реверсивные итераторы или функциональный объект `greater`.

```
vector<int> V={10, 4, 3, 9, 11, 7};
sort(V.rbegin(),V.rend());
sort(V.begin(),V.end(),greater<int>());
```

Нижняя и верхняя грань

Пусть задан диапазон упорядоченных по возрастанию элементов $a_0, a_1, a_2, \ldots a_{n-1}$ и некоторое число w. Нижней гранью называется такая позиция l, что $a_l \geq w$. Верхней гранью называется такое число u, что $a_u > w$. Если ни одно число из диапазона не удовлетворяет заданным условиям, то нижней (верхней) гранью считается позиция n. Для диапазона элементов, хранящихся в векторе или близких к нему структурах данных эти алгоритмы поиска нижней и верхней грани работают за время $O(\log n)$, а для всех остальных структур данных — за $O(n)$. Функции, реализующие эти алгоритмы, называются `lower_bound` и `upper_bound` и имеют три обязательных параметра: два итератора, указывающие диапазон поиска, и значение нижней (верхней) грани. Четвертым необязательным параметром является бинарный предикат сравнения элементов, такой же, как и для алгоритма сортировки.

Алгоритмы нижней и верхней грани можно использовать для определения количества элементов, лежащих в заданных диапазонах значений.

```
vector<int> V={-5,-4,-3,-2,-1,0,1,2,3,4,5};
int a=-3,b=4;
//Количество элементов в диапазоне (a;b);
cout<<lower_bound(b)-upper_bound(a)<<endl;
//Количество элементов в диапазоне [a;b);
cout<<lower_bound(b)-lower_bound(a)<<endl;
//Количество элементов в диапазоне (a;b];
cout<<upper_bound(b)-upper_bound(a)<<endl;
//Количество элементов в диапазоне [a;b];
cout<<upper_bound(b)-lower_bound(a)<<endl;
```

Самостоятельно ответьте на вопрос, почему подсчет количества элементов для разных видов диапазонов производится, как показано в примере.

Поиск уникальных элементов

Этот алгоритм убирает повторяющиеся элементы из упорядоченной последовательности, смещая уникальные значения влево так, чтобы они образовывали непрерывную последовательность. Функция `unique` имеет два обязательных параметра: итераторы, указывающие исходный диапазон. Функция возвращает итератор на конец области из уникальных элементов. При этом элементы, которые находятся справа от этого итератора, остаются в памяти но заполнены неопределенными значениями, поэтому их можно удалить, уменьшив размер вектора.

```
vector<int> V={-3,5,0,7,-2,0,5,-2};
sort(V.begin(),V.end())
auto it=unique(V.begin(),V.end());
V.resize(it-V.begin());
```

В этом примере вектор сначала сортируется, поскольку `unique` работает только с упорядоченными диапазонами, далее выполняется поиск уникальных элементов, после чего удаляются неопределенные значения, расположенные справа от уникальных.

Операции над диапазонами

К упорядоченным диапазонам уникальных элементов можно применять теоретико-множественные операции, такие как объединение (`set_union`), пересечение (`set_intersection`), разность (`set_difference`), симметрическая разность (`set_symmetric_difference`).

Каждая из функций, реализующих эти алгоритмы, содержит пять обязательных параметров: два итератора, задающие диапазон элементов первого множества; два итератора, задающие диапазон элементов второго множества; итератор, задающий начало диапазона для результата операции. Все функции возвращают итератор на конец результирующего диапазона. Очень важно понимать, что место в памяти для записи результата должно быть выделено заранее. Это создает определенные сложности, связанные с тем, что количество элементов в ответе неизвестно до выполнения операции. Поэтому, при использовании таких функций может оказаться полезным специальный класс объектов, называемых адаптерами итераторов. Адаптеры предоставляют интерфейс итератора для манипуляций над объектами, которые сами не являются итераторами. При работе с весторами используется адаптер `back_inserter(V)`, выполняющий при получении нового элемента операцию вставки в хвост вектора `V`. Рассмотрим два примера.

```
vector<int> A={11,18,23,31,40,55,59};
vector<int> B={3,11,17,23,25,29,31,37};
vector<int> C;
C.resize(A.size()+B.size());
auto it=set_union(A.begin(),A.end(),B.begin(),B.end(),C.begin());
C.resize(it-C.begin());
```

В этом примере мы хотим найти объединение элементов, лежащих в векторах `A` и `B`. При использовании соответствующей функции `set_union` мы должны быть уверены, что память для результата выделена до начала операции в полном объеме. Мощность объединения двух множеств не превосходит суммы мощностей, поэтому изначально вектору `C` задается максимально возможный размер. После выполнения операции размер вектора уменьшается, чтобы отбросить неиспользуемые элементы. Минусом этого способа является то, что уменьшение размера вектора не приводит к уменьшению размера памяти, используемой им. Особенно важным это может оказаться для функций нахождения пересечения

и разности множеств, поскольку мощность результата может существенно различаться в зависимости от входных данных.

```
vector<int> A={11,18,23,31,40,55,59};
vector<int> B={3,11,17,23,25,29,31,37};
vector<int> C;
set_intersection(A.begin(),A.end(),B.begin(),B.end(),back_inserter(C));
```

В этом примере для добавления используется адаптер итератора. Не требуется предварительно выделять память, поскольку теперь это делается автоматически при вставке элемента.

Циклический сдвиг

Алгоритм циклического сдвига позволяет позволяет циклически сместить элементы некоторого диапазона. Функция, реализующая этот алгоритм, имеет три параметра: итератор, указывающий на начало дипазона, итератор, указывающий на элемент, который должен стать первым после сдвига, и итератор, указывающий на конец диапазона. Рассмотрим примеры.

```
vector<int> A={1,2,3,4,5,6,7,8,9};
rotate(A.begin()+2,A.begin()+4,A.begin()+8);
```

В первом примере циклический сдвиг применяется к числам `3,4,5,6,7,8`. В результате сдвига `5` окажется на первом месте и диапазон примет вид `5␣6␣7␣8␣3␣4`. А весь вектор будет содержать значения `1␣2␣5␣6␣7␣8␣3␣4␣9`.

```
string s="qwertyuiop";
rotate(s.begin(),s.begin()+6,s.end());
```

В результате будет получена строка «`uiopqwerty`».

Перестановки

Алгоритмы `next_permutation` и `prev_permutation` так переставляют элементы исходного диапазона, чтобы получить последовательность лексикографически большую (меньшую) исходной, причем полученная перестановка будет непосредственно следовать (предшествовать) предыдущей в лексикографическом порядке.

Функции имеют два параметра: итераторы, указывающие на границы диапазона. Функция возвращает логическое значение, равное истине, если следующая большая (меньшая) перестановка успешно найдена. В протином случае функция возвращает ложь и находит лексикографически минимальную перестановку. Например, следующая программа

```
string s="aabb";
cout<<s<<endl;
while (next_permutation(s.begin(),s.end()))
  cout<<s<<endl;
```

выведет последовательно строки `aabb`, `abab`, `abba`, `baab`, `baba`, `bbaa`.