Функции - часть типа данных

Основная идея объектно-ориентированного языка - объекты (переменные пользовательских типов) сами знают, как нужно работать со своими данными. То есть функции для обработки данных того или иного типа должны быть связаны с самими данными - быть частью объектов. Такие функции, принадлежащие типу данных, в объектно-ориентированных языках принято называть методами объекта.

В C++ сам тип данных, который может включать и данные, и методы (функции), называется классом (class), а переменная такого типа - объектом, или, чаще, представителем (instance) класса. При этом в C++ есть все средства для того, чтобы сделать классы очень похожими на встроенные типы - позволяющими инициализировать своих представителей, использовать их в выражениях, обеспечить возможности ввода-вывода и обработки ошибок и так далее.

Конструкторы и деструкторы

Важный этап в жизни переменных - их создание и уничтожение. Например, когда создается статическая переменная встроенного типа, можно рассчитывать, что она будет инициализирована нулем. Ее можно также явно инициализировать при создании другим значением. Соответственно, в какой-то момент работы программы приходит время уничтожить переменную и освободить отведенную под нее память.
Чтобы представитель класса вел себя подобным образом, в классе определяют специальные функции - конструкторы, которые и занимаются инициализацией. Когда переменная должна быть уничтожена, вызывается другая функция - деструктор, которая «аккуратно» выполняет все завершающие операции (например, закрывает вспомогательные файлы).

Динамическая память - C++-стиль

Поскольку в классе есть возможности для аккуратной инициализации и уничтожения переменных, нужно позаботиться, чтобы эти возможности работали и при динамическом создании переменных. В C-стиле, основанном на функциях *malloc()* и *free()*, такого добиться трудновато. Поэтому в C++ использован другой подход - в язык добавлены операторы *new* и *delete*. Делают они примерно то же, что *malloc* и *free*, но при этом знают о существовании конструкторов и деструкторов.

Перегрузка операторов и функций

Для того чтобы переменные пользовательского типа можно было использовать в выражениях наравне со встроенными типами, C++ позволяет определять для них функции, которые вызываются при обработке операторов. Например, определив в соответствующем классе подходящую функцию, можно определить оператор умножения для матриц или комплексных чисел. Это действие носит называние перегрузка операторов (operator overloading).

Часто одной функции на оператор оказывается недостаточно. Например, ту же матрицу можно умножить на другую матрицу, а можно на константу. Ясно, что список параметров у этих двух функций будет разным. В C++ позволительно определять несколько функций с одним и тем же именем - лишь бы эти функции различались по типу и количеству параметров. Это и само по себе весьма удобное нововведение, но, главное, такая возможность позволяет задавать в классах наборы функций для перегружаемых операторов.

Ссылки (references)

В C++ для пользовательских типов оператор вызывает соответствующую функцию класса. Ключевое преимущество передачи параметров по ссылке состоит в том, что функции возможно передать переменную, функция изменит ее, и (при необходимости) вернет результат по ссылке. При этом в обоих случаях имеем дело не с копией с таким же значением, как у оригинала, а с самой оригинальной переменной - именно это и требуется в выражениях вроде *i=++k*.

Исключения (exceptions)

При создании C++ пришлось подумать и об обработке ошибок. Использовать в объектном коде С-стиль, проверяя, какое значение вернула функция не всегда возможно. А о схожести поведения встроенных и пользовательских типов и вовсе пришлось бы забыть. Поэтому в C++ предусмотрен другой принцип работы с ошибками, более гибкий и универсальный. И средства для этого внесены в сам синтаксис языка.

Когда какая-то функция в C++ обнаруживает ошибку, она генерирует так называемое исключение (exception), причем вместе с этим исключением она может передавать на верхний уровень практически любую информацию о подробностях возникшей ошибки. И при этом генерация исключения никак не связана с возвращаемым значением функции (тем, которое ставят в операторе return). Если на верхних уровнях никто не позаботился об обработке ошибок такого типа, то программа аварийно завершится. Если же обработка для них предусмотрена, то соответствующий уровень, получив информацию, переданную с исключением, может попытаться исправить ситуацию и повторить вызов функции

Наследование и полиморфизм

В C++ производный класс может наследовать не от одного, а от нескольких базовых классов, а те в свою очередь, тоже могут быть чьими-то наследниками, так что наследование и полиморфизм открывает поистине безграничные возможности для творчества. Впрочем, они же и приводят к нечитабельности объектного кода - попробуй проследить через десяток предков, что сделает какой-нибудь оператор.

Новый стиль ввода-вывода

В C++ ввод-вывод реализован не как в C, то есть не на основе функций (не переписывать же printf() всякий раз, когда очередному классу потребуются возможности ввода-вывода). Эта проблема в C++ решена следующим образом - никаких специальных операторов для ввода-вывода язык не предусматривает, он просто использует уже имеющиеся операторы << и >> (битового сдвига). Оказывается, что таких средств, как классы, перегрузка операторов, наследование вполне достаточно, чтобы сделать из них операторы ввода-вывода как для встроенных, так и для пользовательских типов данных и при этом сохранить их первоначальное назначение.

В итоге:

Класс в C++ включает в себя не только данные, но и методы (функции) для работы с этими данными.

Конструкторы и деструкторы позволяют правильно создавать и уничтожать представителей классов.

Благодаря новым операторам new и delete, конструкторы и деструкторы вызываются даже при работе с динамической памятью.

Перегрузка функций и операторов и добавление в язык ссылок дают возможность использовать в выражениях объекты наравне с переменными встроенных типов.

Наследование позволяет расширять функциональность уже имеющихся базовых классов, создавая на их основе производные классы. Полиморфизм позволяет при необходимости работать с производными классами так же, как с базовыми.

Используя перечисленные выше средства, C++ предоставляет программисту новый стиль ввода вывода, который позволяет работать как с переменными встроенных типов, так и с объектами классов.

C++ как "улучшенный" C

Новый вариант комментариев

В обычном C комментарии ограничены с обеих сторон специальными комбинациями символов

/* комментарий в стиле C */

и могут занимать несколько строк. В C++ можно также поставить подряд два символа //, при этом все после этих символов до конца строки будет также расцениваться, как комментарий // Пример комментария в C++

i++; // инкрементируем переменную i

C++ как "улучшенный" C

Классические C-комментарий не могут вкладываться друг в друга. Например, в программе для "штатных" комментариев можно использовать новый стиль, а при отладке комментировать целые куски кода, не рискуя получить ошибку из-за вложенных комментариев:

```
/* Temporarily commented for debugging

// Function for finding substring
char *substring(char *str)
{
  if (str == 0) // Check argument against 0
  {
    return 0; // invalid argument
  }

  ...

END Temporarily commented for debugging */
```

C++ как "улучшенный" C

Неименованные параметры

Случается, в вызове функции стоит большее число параметров, чем на самом деле функция использует (причины могут быть разные, но речь сейчас не о них). В этом случае можно не указывать имя ненужного параметра при определении функции. Для сравнения:

```
int f(int x, int y, int z) {
  return x+z;
}
```

и

```
int f(int x, int, int z) {
  return x+z;
}
```

В последнем варианте тому, кто смотрит код, сразу ясно - о втором параметре функции не забыли, его не используют намеренно.

C++ как "улучшенный" C

Новый синтаксис приведения типов и инициализации
В C++ добавлен новый синтаксис для инициализации переменных и приведения типов. В следующем примере присутствуют оба варианта синтаксиса:

```
int i=1; // Old style initialization
int j(1); // New style initialization
...

i = (int) 1.5/j; // Old style typecast
i = int(1.5/j); // New style typecast
```

Новый синтаксис и для инициализации, и для приведения типов напоминает вызов функции.

C++ как "улучшенный" C

Доступ к замаскированной глобальной переменной
В C параметр или локальная переменная функции маскируют одноименную глобальную переменную, и у программиста нет возможности ее использовать. В C++ в подобной ситуации функция также будет использовать локальную переменную или параметр, но при необходимости вы сможете добраться и до глобальной, для этого перед ее именем надо поставить *::* (два символа двоеточия):

```
int i; // global scope variable

void f() {
  int i;
  i = 0; // Local i used

  ::i=0 // Global i used, would be impossible in C

}
```

Сдвоенный символ двоеточия - это так называемый оператор разрешения области видимости (scope resolution operator) и используется он не только для доступа к глобальным переменным.

C++ как "улучшенный" C

Создание переменных

Еще одно улучшение - в C++ переменные можно создавать не только на глобальном уровне либо в начале блока до первого исполняемого оператора, но и после исполняемых операторов, и даже в заголовке цикла for. В приведенном примере

```
main(int argc, char **argv) {

  if (argc==0)
    return 0;

  char *p; // Error in C, OK in C++

  for (int i=0; i<argc; i++) { // Error in C, OK in C++
    p = argv[i];
  }

  return 0;
}
```

C++ как "улучшенный" C

Обе локальные переменные создаются против правил C (эти строки отмечены комментариями), однако C++ такое допускает. Во втором случае, когда переменная i создается прямо в заголовке цикла for, может возникнуть вопрос - а какая у нее область видимости, к какому блоку она относится. При стандартизации языка на эту тему долго спорили, в конце концов сошлись на том, что такая переменная видна только в шапке и теле цикла, но не видна в блоке, в котором этот цикл стоит.

C++ как "улучшенный" C

Встроенные (inline) функции и макросы

Макросы (макроопределения) с параметрами. В C возможно директивой препроцессора
#define sqr(x) ((x)*(x))
подставить выражение *((i)*(i))* вместо *sqr(i)* везде в тексте программы (аргумент при использовании макроопределения может быть любой). Это очень похоже на вызов функции, но замена происходит именно текстовая. По этой причине макросы работают быстрее функций (чем и привлекательны). И по этой же причине служат источником неприятных ошибок - написав *sqr(i++)*, переменная инкрементируется не один, а два раза.

C++ как "улучшенный" C

C++ предлагает достойную и безопасную замену макросам - встроенные функции. В нем можно перед определением функции добавить ключевое слово *inline*

```
inline int sqr(int i) { return i*i; }
```

и процессор будет встраивать код функции прямо в места ее вызовов. И при этом позаботится, чтобы вызов *sqr(i++)* сработал правильно, как и для "полновесной" функции. Так что этот вариант позволяет генерировать очень эффективный код, и при этом избавляет от неприятностей, характерных для макросов.

Ключевое слово *inline* носит для транслятора рекомендательный характер - слишком сложные функции он по прежнему будет делать полновесными. Так что inline-функции желательно делать короткими - в идеале однострочными.

Встраиваемые функции особенно ценны в объектном коде, для которого характерно изобилие вызовов коротких функций - там они могут кардинально повысить эффективность программы.

Пространства имен

Пространства имен помогают избегать конфликтов имен (функций, переменных и так далее). Если попытаться сравнить *random()* из стандартной библиотеки со своим генератором случайных чисел, то в программе на C придется изобретать для своего генератора другое имя. В C++ возможно поместить свою функцию в пространство имен, как бы расширить ее имя

```
namespace my_funcs {
  long random() { ... };
};
```

и после этого использовать следующим образом: long l;

```
l = my_funcs::random();
```

Здесь *::* (двойное двоеточие) - оператор разрешения области видимости (scope resolution operator).

Аналогичным способом застраховались от конфликта имен и разработчики стандартной библиотеки - только пространство имен у них называется *std*. Так что теперь вы можете пользоваться обеими функциями:

```
l = std::random(); // из stdlib
l = my_funcs::random(); // my own function
```

Указывать для каждой функции пространство имен довольно утомительно, так что вы с помощью директивы *using* можете сказать, какой именно функцией (или набором функций) хотите пользоваться: using

Можно в директиве *using* поставить не имя функции, а все пространство имен, при этом все имена без *::* будут ссылаться либо на функции в текущем файле (если они не вынесены, как *my_funcs::random*, в отдельное пространство имен), либо на пространство, указанное в директиве *using*:

```
using namespace std;
l = random(); // std::random()
l = abs(); // std::abs()

l = my_funcs::random(); // own function
```

Как пользоваться стандартными файлами заголовков в C++?
До того, как был принят стандарт, довольно долго в C++ не было пространств имен, а стандартные файлы заголовков так же, как и в C, заканчивались суффиксом .h - например <iostream.h>. С принятием стандарта все имена библиотеки вынесли в пространсво std, однако к тому времени на C++ было написано много программ и библиотек. И для того, чтобы сохранилась совместимость с ними, файлы заголовков пришлось исполнить в двух вариантах - в старом, с суффиксом .h, и в новом - вообще без суффикса. При этом к заголовкам, пришедшим из C, прибавили спереди букву c, например <stdio.h> превратился в <cstdio>. А у заголовков, которых в C не было, просто убрали суффикс - <iostream.h> стал называться <iostream>.

C++-стиль ввода-вывода.

Посмотрим, как выглядит ввод-вывод в C++. рассмотрим программу:

```
// file c++io.cc
#include <string>
#include <iostream>

using namespace std;

main() {
  string name;
  int age;

  cout << "Enter your name and age" << endl;

  cin >> name >> age ;

  cout << "Hello " << name << ", your age is " << age << endl;
```

Сначала в программу включаются два стандартных файла заголовков

```
#include <string>
#include <iostream>
```

для работы со строками и с вводом-выводом в C++ стиле. Затем директивой `using namespace std;` указываем, что собираемся работать с именами из стандартной библиотеки. Затем идет функция *main()*:

```
main() {
  string name;
  int age;
  ...
```

В ней мы объявляем две переменные - *name* и *age*. Тип *string* это - не встроенный тип, а пользовательский (C++ класс). Именно ради него включен стандартный заголовок <string>.

Дальше идет собственно C++-вывод. Инструкция cout << "Enter your name and age" << endl; печатает приглашение, а затем переходит на новую строку (*endl* - сокращение от end of line - делает то же, что "\n" в форматной строке *printf*). Вывод попадает в выходной поток *cout* - это аналог *stdout* в C. Оператор сдвига << здесь служит для целей вывода, так сказать, сдвигает печатаемые значения в поток.

Похожим образом выглядит и чтение данных из потока, только вместо *cout* стоит *cin* (аналог *stdin*), и используется оператор >> сдвига вправо - данные сдвигаются из потока в программу:
cin >> name >> age ;
и затем в *cout* выводится приветствие и печатаются значения введенных переменных: cout << "Hello " << name << ", your age is " << age << endl;

Собрав и запустив такую программу, можно убедиться, что она работает корректно:

```
~/c++course/praktikum> c++ c++io.cc -o c++io
~/c++course/praktikum> ./c++io
Enter your name and age
Anonymous 24
Hello Anonymous, your age is 24
~/c++course/praktikum>
```

Стандартные потоки C++:

cin - стандартный поток ввода. Аналог *stdin* в C. Буфуризованный.
cout - стандартный поток вывода. Аналог *stdout* в C. Буферизованный.
cerr - стандартный поток сообщений об ошибках. Аналог *stderr* в C. Небуферизованный.
clog - Буферизованный вариант cerr (это не самостоятельный поток, он связан с тем же устройством или файлом, что и *cerr*). Предназначен для вывода больших диагностических сообщений, поскольку за счет буферизации работает быстрее.
Как и в C, стандартные потоки открываются автоматически при старте программы, и так же автоматически закрываются при успешном завершении. Разумеется, как и в C, программа может при необходимости использовать и другие потоки, но их придется открывать и закрывать

Перегрузка функций. Прототипы и сигнатуры.
Перегрузка функций - механизм, который позволяет писать функции с одним и тем же именем, но с разными параметрами, а транслятору, соответственно, догадываться, когда какую функцию надо вызывать.

Сейчас, мы, разумеется, будем работать только с глобальными функциями (то есть, с обычными функциями в понимании C), но механизм этот в первую очередь предусмотрен для методов класса - тех функций, которые наравне с данными включаются в пользовательские типы данных.

Для того чтобы понять, как работает перегрузка функций, необходимо нужно освоить два новых термина. Делать это будем на примере функции *strcpy()* из стандартной библиотеки C (заголовок <string.h>). Причем само определение функции, ее тело, нас сейчас не интересует, обсуждать мы будем только объявление функции. Итак, согласно странице описания (*man strcpy*) функция объявлена следующим образом

```
char *strcpy(char *dst, const char *src);
```

Такое объявление - тип функции (*char **), ее имя (*strcpy*), количество и типы параметров (*char *dst, const char *src*) - принято называть прототипом функции. Имена параметров в прототипе нужны не транслятору, а человеку для улучшения читабельности программ. Так что прототип функции *strcpy* можно записать еще короче: char *strcpy(char *, const char *);

Итак, прототип функции включает в себя тип самой функции, ее имя, количество и типы ее параметров. Именно прототип указывается при объявлении функции.

Сигнатура функции - это прототип за вычетом типа функции, то есть strcpy(char *, const char *);

В С вычислим наибольшее из двух значений с помощью троичного оператора *(?:)*. *max()* - вполне естественное название для подобной функции. Но сам оператор умеет работать с разными типами данных, например, с *int* и *double*. В С, чтобы использовать одно и то же имя для разных типов, применим макроподстановку с параметрами:

```
#define max(x,y) ( (x)>(y) ? (x) : (y) )
```

C++ позволяет поступить проще

```
int max(int x, int y) { return x>y ? x : y ; }

double max(double x, double y) { return x>y ? x : y ; }
```

Таким образом, написаны две функции max под разные наборы параметров. Это и есть перегрузка (overloading) функций.
Теперь, когда транслятор встретит в программе вызовы *max()*:

```
int a;
a = max(1, 3);

double d;
d = max(2.5, -1.0);
```

он по типу используемых параметров подберет и вызовет подходящий вариант функции, в первом случае *max(int,int)*, во втором - *max(double,double)*.

Можно написать и функцию, которая определяет максимальное из трех значений double max(double x, double y, double z) {
  return max(max(x,y), z);
}
double d = max(1.0, 2.0, 3.0);

Такой код тоже будет работать, поскольку транслятор подбирает подходящую функцию не только по типам, но и по количеству параметров. Вот на что транслятор при таком подборе не смотрит, так это на тип самой функции. Написать double max(int x,int y) { ... };
int max(int x,int y) { ... }; он не позволит - даст сообщение об ошибке, поскольку вызвав функцию, возможно использовать возвращаемое значение, а можно его проигнорировать // No info about type of return value
max(1.0, 2.0);

В этом случае у транслятора просто нет возможности узнать, какая функция имелась в виду. Теперь должно быть ясно, зачем наряду с прототипом функции понадобилась и сигнатура - именно ее транслятор использует, когда отыскивает среди перегруженных функций подходящую. И поэтому C++ требует, чтобы сигнатуры всех функций были разными. О сигнатуре функции можно думать, как о неком аналоге имени функции в обычном C. Собственно, за кулисами дело так и обстоит - свои внутренние ("для служебного пользования") имена функций транслятор строит именно на основе сигнатур, и называется это явление"name mangling".

Писать перегруженные функции для всех возможных комбинаций типов параметров - занятие довольно утомительное. Например, для разных сочетаний float и double даже функцию max() с двумя аргументами пришлось бы исполнить в четырех вариантах:

```
float max(float, float) { ... };
double max(float, double) { ... };
double max(double, float) { ... };
double max(double, double) { ... };
```

К счастью, C++ не требует точного соответствия сигнатуры при вызове сигнатуре одной из имеющихся функций. Если точно соответствующей перегруженной функции нет, он в состоянии подобрать из набора наиболее подходящую. Наиболее понятная аналогия из C - арифметические выражения, в которых можно перемешивать переменные разных типов // '9' will be used as (double)'9' double x = '9'* 0.5; Так и с перегруженными функциями - для работы с четырьмя приведенными комбинациями float и double достаточно одной-единственной функции - *max(double,double)*.

Как бороться с конфликтами имен перегруженных функций. Если, имея в распоряжении функции *max(int,int)* и *max(double, double)*, написать

```
int i = max(2.5, 1);
```

то транслятор «окажется» в затруднительное положении - точная сигнатура вызова - *max(double,int)*, а для нее одинаково хороши обе имеющиеся в наборе функции (у обеих - точное совпадение типа одного из параметров, и совместимый тип другого параметра). Транслятор сообщит об этом двусмысленном вызове, и даже укажет возможных кандидатов:

```
ovld.cc:5: error: call of overloaded `max(double, int)' is
        ambiguous
ovld.cc:1: error: candidates are: int max(int, int)
ovld.cc:2: error: double max(double, double)
```

В подобной ситуации проще всего сделать одну из сигнатур более предпочтительной с помощью явного приведения типа прямо при вызове:

```
// will call max(double, double)
int i = max(2.5, double(1));
```

и после такой подсказки транслятор легко отыщет нужную функцию.

Шаблоны (templates)

Шаблоны могут использоваться для создания не только функций, но и пользовательских типов данных. Но сейчас будем говорить только о шаблонах функций.

Рассмотрим пример функции *max*. При этом функции с двумя аргументами отличались лишь используемым типом данных:

```
int max(int x, int y) { return x>y ? x : y ; }
double max(double x, double y) { return x>y ? x : y ; }
```

C++ позволяет в подобных случаях избежать рутинной работы. Для автоматизации процесса надо написать шаблон: template<class T>

```
  T max(T x, T y) { return x>y ? x : y; }
```

(здесь <class T> - не C++-класс, а так называемый параметр шаблона).

Сам по себе такой шаблон еще ни к чему не обязывает транслятор. Но если в программе появится вызов *max()* с двумя аргументами одного типа, то транслятор создаст для него по шаблону соответствующую функцию (если он это не сделал раньше). Так, для двух первых вызовов

```
max('c','d'); // instantiate max(char,char)
max(1.0,2.0); // instantiate max(double,double)
max('c','d'); // max(char,char) already exists
```

транслятор создаст функции *max(char,char)* и *max(double,double)*, заменив в теле шаблона тип-параметр T реальным типом, использованным в выводе. К моменту третьего вызова у него уже будет создана функция *max(char,char)*, которую он и использует. Такое поведение немного похоже на макроподстановки, но только текстовая замена формального параметра шаблона на реальный тип происходит не в месте, где max() используется, а в теле автоматически созданной по заданному образцу функции.

Шаблону можно указать не один параметр-тип, а несколько. Например, такой шаблон будет создавать функции, находящие аргумент с типом большего размера, и возвращающие этот размер в байтах:

```
template<class T1, class T2>
   int maxsize(T1 a, T2 b) {
     int size_a = sizeof(a);
     int size_b = sizeof(b);
     return size_a>size_b ? size_a : size_b;
   }
```

Подобная автоматизация выглядит очень привлекательно, и она в самом деле может заметно облегчить работу. Например, часть стандартной библиотеки C++ - STL (standard templates library) - это как раз набор хорошо продуманных и готовых к использованию шаблонов (правда, там шаблоны не для функций, а для типов данных). Но в работе с шаблонами есть и своя специфика, и при неумелом использовании они способны принести больше хлопот, чем пользы. Два наиболее частых источника разочарований:
Текстовая замена формального типа на фактический в теле шаблона порой приводит к весьма причудливым конструкциям, которые могут давать ошибки при трансляции, или работать совсем не так, как замышлялось разработчиком шаблона. В общем, ситуация очень похожа на проблемы с макроподстановками в C.
Имея в распоряжении шаблон функции, транслятор, естественно, будет создавать по нему тела функций, точно соответствующие сигнатуре вызова. Даже в тех случаях, когда вполне можно было обойтись одной функцией с сигнатурой, пусть не точно совпадающей, но совместимой сразу с несколькими вызовами. Так что неаккуратное использование шаблонов временами приводит к тому, что программа "распухает" - код автоматически сгенерированных функций занимает необоснованно много места в памяти.

Ссылки

Передача параметров по ссылке лишь побочный эффект появления нового типа данных - ссылок. Так что и знакомиться мы сейчас будем в первую очередь с этим новым типом данных и с его особенностями. А параметры функций будем использовать как "наглядное пособие".
Само создание ссылки похоже на создание обычной переменной - тип, имя, инициализатор. Но справа от типа надо поставить символ &, и обязательно явно инициализировать ссылку - связать с тем объектом, на который она должна ссылаться:

```
int i; // целая переменная
int& ref = i;
```

После этого можно пользоваться ссылкой ref так же, как самой переменной i - при этом будет происходить обращение не к ссылке, а к самой переменной:

```
i = 8; // "прямой" доступ к i
ref = 8; // то же действие, но доступ к i через ссылку
```

Иногда говорят, что, создавая ссылку, мы создаем еще одно имя для имеющейся переменной.

Из приведенного примера ясно, что создавать и использовать ссылки не сложнее, чем обычные переменные. Однако при работе со ссылками всегда надо держать в уме две вещи:

Ссылка никогда не может существовать сама по себе - она обязательно связана с какой то переменной. У нее даже нет своего адреса - если вы попробуете взять адрес ссылки, то получите адрес связанной с ней переменной.

Все, что можно сделать с самой ссылкой - это создать ее. После создания повлиять на ее дальнейшую судьбу никак нельзя, поскольку работа будет осуществляться не со ссылкой, а с самой переменной. Ссылку нельзя уничтожить, она может "умереть" только своей смертью, например, когда программа выйдет из локального блока, где ссылка объявлена. Не удастся и перенаправить ее на другую переменную. И так далее.

Как выглядит работа со ссылками в функциях?

Пусть в рамках C написана функция, меняющая значения двух аргументов, использующая указатели:

```
// swap() function definition, C-style (pointers)
void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}

// swap() function usage, C-style
int i = 1, j = 2;
swap(&i,&j); // Now i==2 and j==1
```

Со ссылками все это записывается проще:

```
// swap() function definition, C++-style (references)
void swap(int& a, int& b) {
  int temp = a;
  a = b;   b = temp;
}

// swap() function usage, C++-style
int i = 1, j = 2;
swap(i,j); // Now i==2 and j==1
```

Точно так же можно использовать ссылку и для типа функции (то есть, для типа возвращаемого ею значения). Например, можно вернуть ссылку на переменную, заданную аргументом:

```
int& ret_arg(int &arg) {
  return arg;
}
```

Такая функция вернет не значение аргумента, а именно ссылку на сам аргумент. Возможно, вы еще не почувствовали разницы, но понять ее поможет вот такое использование только что определенной функции:

```
int i;
ret_arg(i) = 2; // Now i==2
```

Выглядит странновато - мы присваиваем что-то вызову функции. Но это и есть самое ценное. Функции, возвращающие ссылки - это левые значения (lvalues), то, что можно ставить слева от оператора присваивания.
Теперь можно приоткрыть завесу над тайной ввода-вывода в C++-стиле. Когда вы пишете
cout << i ;
программа вызывает функцию, соответствующую оператору << (это называется перегрузкой операторов, operator overloading). Причем в качестве одного из аргументов используется *ссылка* на поток cout. И сама функция тоже возвращает ссылку на этот поток. Так что значение выражения
(cout << i)
это тот же cout, и мы можем таким же оператором вывести в него что-нибудь еще:
(cout << i) << j;
А поскольку вызов, соответствующий второму оператору <<, тоже вернет ссылку на поток, мы можем продолжать
((cout << i) << j) << k;
и так далее в том же духе.

Заканчивая разговор о ссылках, следует сказать об одном «популярном» заблуждении. Многие (особенно начинающие) думают, что одно из преимуществ ссылок - их связь с реальным объектом. Дескать, когда вы в C работаете с указателями, вы должны для подстраховки смотреть, указывают ли они куда-нибудь (проверять их на *NULL*). А вот ссылка всегда смотрит на переменную, так что подобных неприятностей с ней быть не может. Так то оно так, но только до известного предела. Дело в том, что переменная может «умереть» раньше ссылки, как например, в таком фрагменте программы.

```
// Allocate variable of type double on heap
double *p = (double*)malloc(sizeof(double));

// Create reference to variable just allocated
double& ref = *p;

ref = 1.0; // same as *p = 1.0

// delete allocated variable
free(p);

// ref is still alive, but its variable already dead,
// so next line will cause trouble
ref = 1.5; // same as *p=1.5, but *p already destroyed
```

Исключения (exceptions)

Исключение - C++-механизм для обработки ошибок.

Когда какая-нибудь функция обнаруживает ошибку, она сообщает об этом вызывающему коду, посылая исключение. Делается это с помощью оператора *throw*. Наша функция будет вычислять факториал, посылая исключение при слишком большом значении аргумента

```
unsigned factorial(unsigned val) {
  if (val > 22)
    throw "Argument too large";
  return (val == 0 ? 1 : val*factorial(val-1));
}
```

В третьей строке стоит ключевое слово *throw*, а затем тот объект, который содержит информацию об ошибке - в нашем случае, это строка, но с тем же успехом можно послать и переменную или структуру. Причем этот объект несет двоякую информацию - тип исключения (*const char**, текстовая строка), и описание нашего конкретного случая (само содержимое этой строки). Послав исключение, функция больше не заботится о его судьбе, да она и не может этого сделать, поскольку оператор *throw* приводит к немедленному прекращению ее работы.

Дальнейшие события происходят в вызвавшем функцию коде, и развиваться они могут по двум сценариям.
Первый сценарий, когда никаких проверок в вызывающем коде не делается:

```
main() {
  unsigned result;
  result = factorial(25);
  cout << result << endl;
  cout << "Goodbye" << endl;
}
```

В этом случае исключение, дойдя до функции *main()*, приведет к аварийному завершению программы

Во втором сценарии вызывающий код обрабатывает исключения, заключая код, который может их генерировать, в try-блок и назначая обработчик для нужного типа исключения с помощью catch-блока:

```
main() {
  unsigned result;

  try {
    result = factorial(25);
    cout << result << endl;
  }
  catch (const char *msg) {
    cerr << " Factorial raised exception " << msg << endl;
  }

  cout << "Goodbye" << endl;

}
```

Работает подобная конструкция следующим образом - если исключения не возникает, то catch-блок не получает управления. Так что после вызова *factorial()* печатается результат и сразу после этого сообщение *Goodbye*.

Когда же *factorial()* взводит исключение, то остальные операторы блока try не выполняются (в нашем примере не выполняется строка с печатью результата), и просматриваются catch-блоки, относящиеся к данному блоку try. Если удается отыскать подходящий по типу исключения, то его код и выполняется, причем в параметр "шапки" блока засылается тот объект, который пришел с исключением. Мы в нашем примере получаем строку с описанием ошибки в параметре msg блока catch, которую и печатаем.

```
~> ./exceptions
Factorial raised exception Argument too large
Goodbye
~>
```

Что случается, если подходящего блока catch нет? В этом случае try-catch не перехватывает исключение и реализуется не второй, а первый сценарий.

```
#include <iostream>

using namespace std;

unsigned factorial(unsigned val) {
  if (val > 22)
    throw "Argument too large";
  return (val == 0 ? 1 : val*factorial(val-1));
}

main() {
  unsigned n;
  while (cin >> n)
    try {
      cout << "factorial(" << n << ")="<<factorial(n) << endl;
    }
    catch (char *msg) {
      cerr << "exception " << msg << endl;
    }
}
```

Естественно, код в try-блоке может взводить исключения не только одного типа, соответственно и catch-блоков в этой конструкции может быть несколько. При генерации исключения программа будет просматривать их один за другим, в том порядке, как они написаны, и выполнит код первого же подходящего по типу catch-блока: try {

```
  ...
}
catch (const char *msg) {
  cerr << "const char * exception handled" << endl;
}
catch (int error_number) {
  cerr << "int exception handled" << endl;

}
catch(...) {
  cerr << "Handler for any type of exception called" << endl;
}
```

Обратим внимание на последний catch-блок - он перехватывает исключения любого типа. И поэтому, кстати, стоит последним – если поставить его первым, то все остальные catch-блоки никогда не вызовутся.
Что именно принято делать в блоках catch? Ответ зависит от того, чего вы хотите добиться и какой стратегией обработки ошибок пользуетесь. Можно подправить какие-то параметры и снова попытаться выполнить блок try. Можно напечатать сообщение об ошибке. А иногда достаточно вызвать в catch-блоке *exit()*, чтобы программа завершилась не аварийно, без дампа памяти.
И последнее об исключениях. Взведенное исключение несет в себе двоякую информацию - сам тип объекта-исключения, и его значение, содержимое. Однако, если само значение не требуется, можно вполне ограничиться только первой половиной - типом. Рассмотрим такой пример:

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

struct my_exception {};

main() {
  try {  throw 1;  }
  catch (int) {  cerr << "integer exception caught"<<endl;  }

  try {  struct my_exception m;  }
```

В первом try-блоке значение (1) передается, но при обработке не используется. А вторая пара try-catch даже использует в качестве исключения структуру my_exception, у которой нет никакого содержимого - только тип.