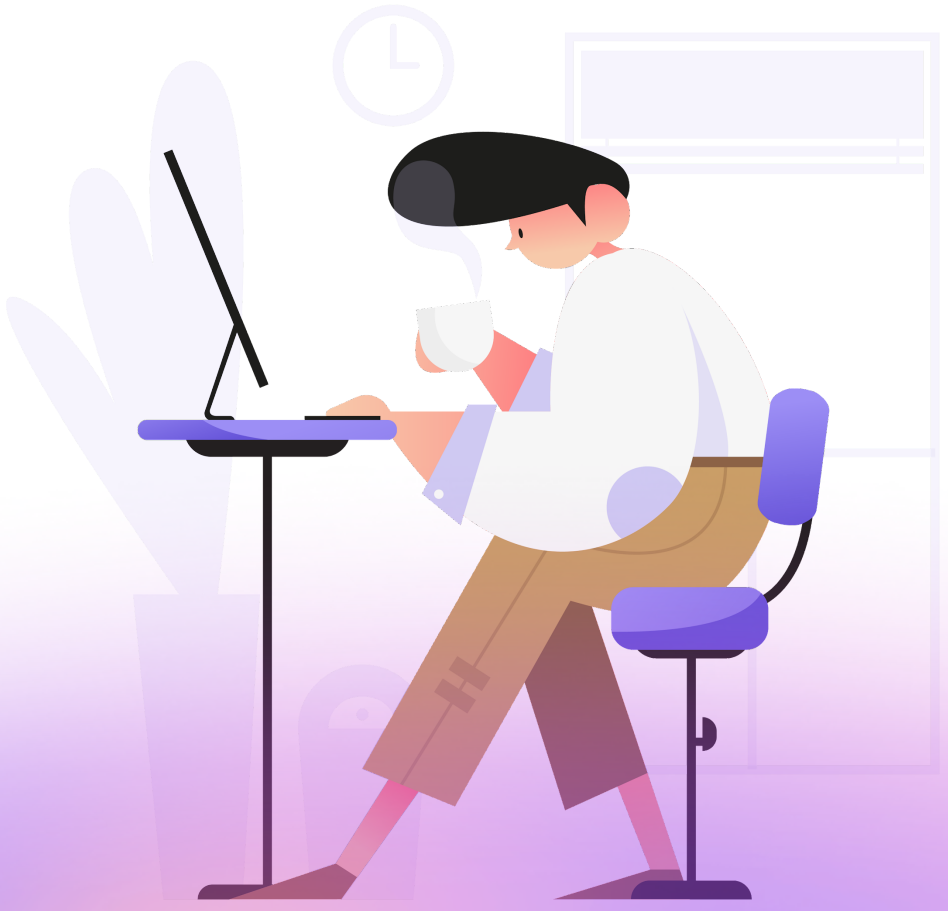
GeekBrains

Основы C++

Функции



На этом уроке

1. Узнаем всё, или почти всё о функциях. Аргументы, параметры, возвратные значения.
2. Научимся описывать указатели на функции и функции обратного вызова.
3. Рассмотрим inline функции и механизм перегрузки функций.
4. Изучим, зачем нужны пространства имён

Оглавление

Функции

На вводном занятии рассматривались принципы организации вычислительного процесса на языке Си, а именно, модульность программы и организации модулей в виде большого количества относительно небольших по размеру функций. Функции вызываются для выполнения необходимых преобразований над данными, или для организации вычислительного процесса (имеется в виду как сам вычислительный процесс, так и прерывания его, ввод-вывод информации, взаимодействие с оператором и т.п.).

Включение функций в структуры наряду с данными привело к появлению в языке понятия “объект” и языка C++, а сами функции выросли в методы классов, но это будет рассмотрено позднее, на

занятиях по объектно-ориентированному программированию (ООП). На этом занятии рассмотрим функции в их классическом понимании.
Итак, чтобы использовать функцию в программе, ее надо:

- объявить для компилятора (подобно объявлению переменной);
- определить (описать);
- вызвать на исполнение (поговорим о “прямом” и “обратном” вызове функций);
- завершить ее исполнение (либо приостановить исполнение переходом к исполнению вложенной функции или переходом к обработке асинхронного внешнего, обычно аппаратного, прерывания).

Объявление функции.

Объявление функции представляет собой строку с именем функции, завершенную символом “`;`”. После `;` в строке могут быть комментарии, оформленные в установленном порядке. В целом такая строка называется **прототипом функции**. Объявление функции используются транслятором для проверки правильности вызова функции. Имена аргументов в объявлении функции могут быть опущены, достаточно указать через запятую только типы аргументов. Допускается также список аргументов при объявлении функции завершать запятой с многоточием `(, ...)`. Это означает, что число передаваемых функции параметров переменно, но не меньше, чем следует идентификаторов до многоточия

Примечание:
Многие компиляторы принимают многоточие `(,...)` в списке параметров при объявлении функции, но только в одном случае: если список параметров **завершается** запятой с последующим многоточием, вот так, например:
`[тип_функции] имя_функции ([тип аргумент_1], [тип аргумент_2], ...);`
Такая запись означает, что функция может вызываться с переменным числом аргументов, но не менее, чем число явно перечисленных, в нашем примере двух. В случае вызова такой функции программист должен сам контролировать реальное количество аргументов, находящихся в стеке при вызове функции, и выбирать из стека аргументы сверх явно перечисленных. Поскольку такое объявление и вызов функции относятся к «экзотическим» приемам программирования, и не сильно распространены на практике, не будем больше заострять внимание на теме вызова функций с переменным числом `(,...)` аргументов. С такими аргументами принято работать с помощью предопределённого набора макросов:

`va_start`	Даёт доступ к аргументам (макрос)
`va_arg`	Ссылается на следующий аргумент (макрос)
`va_copy`	Создаёт копию аргументов (макрос, добавлен в C++11)
`va_end`	Указывает на окончание аргументов (макрос)
`va_list`	Содержит информацию об аргументах, необходимую для работы va_start, va_arg, va_end, and va_copy (typedef)

Простейший пример подсчёта среднего арифметического переданных целочисленных аргументов

```
#include <stdarg.h>
double average(int count, ...) {
```

```
    va_list ap;
    int j;
    double sum = 0;
    va_start(ap, count); // Требуется чтобы получить адрес первого
    for (j = 0; j < count; j++) {
        sum += va_arg(ap, double); // Увеличивает ap до следующего аргумента.
    }
    va_end(ap);
    return sum / count;
}
```

В C++ вызов функций с переменным числом аргументов легко достигается при помощи перегрузки функции, пример которой будет показан на этом занятии при обсуждении перегрузки функции.

Функции, также как операнды и выражения, имеет каждая свой тип. Тип функции является типом возвращаемого в результате ее работы значения и описания принимаемых в функцию параметров. Это те же самые типы переменных, которые мы рассматривали на соответствующем занятии. Плюс один тип дополнительно: `void`.
Тип `void` присваивается функции, которая не должна возвращать после своей работы никаких значений. В функции типа `void` оператор завершения функции `return` должен не иметь при себе возвращаемого функцией выражения, или может вообще отсутствовать. Кратко об этом говорилось на первом занятии. Опишем эту механику подробнее:

```
void print_msg(void); // Это пример объявление функции.
```

Важное замечание. При определении функции `print_msg` мы записали строку прототип так: `void print_msg(void);` но могли бы написать и так: `void print_msg();` как определена в большинстве примеров функция `main()`. Обе записи при определении были бы идентичны. Но при объявлении функции в ANSI C и C99 это существенно разные записи. Разница в следующем: `void print_msg(void);` подразумевает, что никаких аргументов у функции далее, при определении, быть не может. `void print_msg();` - далее, при определении, у функции может быть любое число аргументов любого типа, как их может и не быть вообще. Из этого замечания становится понятно, как объявлена функция `main`. На самом деле у нее могут быть аргументы, но мы определили ее, как определили, и это корректно.
Все вышеизложенное про объявление функций относится к классическому C. До сих пор мы особенно не касались различий между языками Си и C++. Разве что форматы оформления комментариев показали без упора на то, что формат «// комментарий» был введен в синтаксис выражений только начиная с языка C++. Но есть и другие, более значительные отличия, которые относятся к объявлению и использованию функций и переменных, ранее имевшемся в Си. Пришла пора перечислить основные из них:

1. В Си локальные переменные могут быть объявлены только в начале блока, перед первой инструкцией “действия”. В C++ такие переменные могут объявляться в любом месте программы, и что особенно привлекательно - рядом с местом их первого использования. Например, в классическом C, до C99 оператор:
   `for (int i = 0; i < 16; i++) { тело оператора };`
   вызовет ошибку, поскольку переменная i объявлена в первом выражении самого оператора, т.е. внутри самой инструкции ‘действия’. А в C++ такой синтаксис допустим, и даже приветствуется.
2. В C++ определен **тип данных** `bool`, которого не было в классическом C. Единственными значениями этого типа данных могут быть только уже знакомые нам `true` и `false`. Также, как и в C, `false` соответствует `0`, а `true` - все ненулевые значения типа `int`, но использование

`true` и `false` в качестве значений типа данных `bool` значительно усиливает контроль типов и дает возможность различать данные булева и целого типов.

3. В Си два объявления функции (с указанием и без указания `void` параметра) как мы говорили выше, не эквивалентны. В C++ наоборот, эти выражения эквивалентны, и означают отсутствие в командной строке аргументов.
4. В Си, если тип возвращаемого функцией значения при объявлении функции явно не указан, то (по умолчанию) функция возвращает значение типа `int`. Если оператора `return значение;` в теле такой функции нет, то функция возвращает неопределенное значение типа `int`. В языке C++ указание типа возвращаемого функцией значения **обязательно**.
5. В C++ появилось понятие «встраиваемых» (in-line) функций.
6. В C++ появились понятия «виртуальной функции» (virtual function), «родовых» (шаблонных, template) функций и т.п.

Определение функции

Определение функции – это оформленный в соответствии с определенными правилами набор операторов и операций языка Си, рассмотренных на прошлых занятиях. Вот стандартный для языка Си формат определения функции:

```
[тип_функции] имя_функции ([тип аргумент_1], ... [тип аргумент_n])
{
[тип объявление_1;]
[объявление_2;]
....
[объявление_m;]
[оператор_1;]
[оператор_2;]
[операция_1;]
[оператор_3;]
....
[оператор_k;]
[операция_l;]
[return[выражение];]
}
```

Квадратными скобками `[]` выделены необязательные элементы определения функции, многоточиями `...` показана множественность элементов. Как видим, минимально необходимый набор элементов определения функции:

```
имя_функции() { }
```

Выше показан пример определения “пустой”, т.е. не выполняющей никаких действий, функции (для языка C). Трудно придумать, зачем такое определение может понадобиться, однако оно возможно. Определение функции может размещаться в одном или нескольких файлах. Заканчивается определение функции закрывающей фигурной скобкой `}`. Оператор `return[выражение];` может встречаться в определении функции и более одного раза, и даже с разными `[выражениями]`, если функция завершается из оператора ветвления, все выражения должны быть одного типа, а именно `тип_функции`. Если оператор `return` отсутствует, то завершает выполнение функции операция `}` (закрывающая скобка).

По умолчанию (если в первой строке определения опущен `тип_функции`), то функция языка C имеет тип `int`. Напомним, в C++ такое определение недопустимо. Имя функции должно быть уникальным и соответствовать тем же правилам, что и имена переменных, рассмотренные на прошлых занятиях. **Одна функция, и только одна, должна иметь имя** `main`**.** В круглых скобках, вслед за именем функции, перечисляются через запятую передаваемые в функцию переменные вместе с указанием их типов. В данном контексте они называются **параметрами функции**. Если список параметров пуст, то функция называется непараметризованной, и при определении ее имя выглядит так: `имя_функции()`, но при объявлении то же имя должно выглядеть так: `имя_функции (void);`

Вслед за строкой с именем следует **тело функции**, обязательно заключенное в фигурные скобки `{}`, даже если это тело пустое, т.е. не содержит ни одной строки кода. Тело функции состоит из объявления внутренних (локальных) переменных и последовательности операторов и операций, рассмотренных нами на предыдущих занятиях, а также необязательный для void-функции оператор `return[выражение];`
Объявленные внутри функции переменные называются локальными, потому что имеют область видимости только в пределах этой функции. Слово локальная область видимости означает, что память под них выделяется только в момент вызова функции на исполнение, и освобождается эта память сразу после завершения исполнения функции. Передать значения этих переменных можно через параметры в функции, вызываемые только изнутри тела данной функции, либо вернуть значения этих переменных после завершения функции оператором `return`. Обратиться к локальным переменным данной функции из других функций напрямую невозможно. Компилятор допускает объявление переменных одного типа и с одинаковым именем в двух и более разных функциях, но это будут совершенно разные переменные, никак не связанные друг с другом (принцип **“наложения имен”**). Здесь мы снова подошли к понятиям области видимости, классов памяти и пространств имен в языке Си. По ходу изложения далее поговорим о них подробнее, пока же вернемся к определению функций. Вот пример простейшей Си программы с использованием функций:

```
#include <stdio.h>
void print_msg(void) {
    printf("Hi everyone !!!\n");
}

main() {
    print_msg();
}
```

Это пример программы, состоящей из трех функций. Одна из функций называется `main()`. Именно по этому имени ее запускает на исполнение операционная система (ОС), когда вы набираете в командной строке имя исполняемого файла своей программы, либо кликнете на этом имени левой клавишей мышки, если Ваша ОС допускает такой вызов программы на исполнение.
Второй функцией является непараметризованная `void print_msg(void)`, она вызывается из `main()`.
Третьей функцией является библиотечная `printf(...)`. Эта функция вызывается из `print_msg()` и выводит в терминал строку, `“Hi everyone !!!\n”`, где служебный символ `“\n”` означает “перевод строки”. Перечень и содержимое обязательно поставляемых с компиляторами языка Си библиотек стандартизован (о стандартах языка мы говорили на вводном занятии). Здесь лишь заметим, что `printf()` включена в стандартную библиотеку `stdio`, заголовочный файл которой и подключен к нашей программе строкой `#include<stdio.h>`. Без подключения этой библиотеки компилятор выдаст сообщение об ошибке.
Это полностью работоспособный пример программы, однако попробуйте поменять местами функции `main()` и `void print_msg(void)`, и компилятор выдаст сообщение об ошибке, поскольку **любая функция до ее использования должна быть объявлена для компилятора**. В языке Си допускается не объявлять функцию, только если она определена до момента ее использования. Именно это мы и применили в примере, однако при наличии в модуле большого числа функций, которые взаимно вызывают друг друга, это не удобно, поскольку программисту придется следить кроме как за логикой программы, еще и за порядком расположения объявлений функций в модулях. Проще в одном месте, перед первым определением первой функции, объявить все планируемые к использованию функции, а затем определять их в произвольном порядке по мере необходимости.

Вызов функции на исполнение.

На предыдущих занятиях мы говорили о том, что язык Си позволяет задавать область видимости каждого элемента данных и область действия каждой функции. Сейчас снова вернемся к таблице классов памяти, которая связывает между собой понятия типов переменных, областей их действия и времени жизни (для функции это сама функция, точнее её имя, аргументы функции, возвращаемые значения, а также локальные и глобальные переменные). Да, в языке Си сама функция

рассматривается как своего рода переменная, к которой можно обращаться не только по имени (т.е. адресу в памяти), а и с использованием указателей, о чем подробнее поговорим на этом занятии.

Класс памяти	Тип переменной	Область действия	Время жизни
`auto (register)`	внутренняя	Функция или блок кода	Функция или блок кода
`extern`	внешняя	Остаток файла, другие файлы с объявлением `extern`	Программа
`static`	внешняя	Остаток файла (только внутри файла)	Программа
	внутренняя	Функция или блок кода	
`(опущен)`	внешняя	Как `extern`	Как `extern`
	внутренняя	Как `auto`	Как `auto`

В языке Си атрибуты `extern` и `static` могут быть связаны не только с объектами данных, но и с любыми другими объектами, в том числе и с функциями. Все функции по умолчанию трактуются как внешние (`extern`). Как и в случае данных, это означает, что областью действия функции является вся программа. Указание перед именем функции и спецификацией ее типа служебного слова `static` определяет функцию статической. Это определение сужает область действия функции на оставшуюся часть модуля, в котором она определена. Определение функций статическими, где это возможно, относится к хорошему стилю разработки программ.

Передача параметров при вызове функции на исполнение

Формальные **параметры** – это переменные, которые принимают значения, передаваемые функции при вызове, в соответствии с порядком следования их имен в списке аргументов функции.
Класс хранения таких объектов, как аргументы функций, которые помещаются в сегмент стека при вызове функции, является автоматическим (`auto`), т.е. область памяти для хранения этих элементов данных выделяется автоматически при вызове функции и также автоматически освобождается, когда исполнение этой функции завершается. Область действия аргументов ограничена текущей функцией.
Временем жизни аргументов является продолжительность исполнения функции. Как только функция завершит работу, их значения будут утеряны. Когда функция будет вызвана в следующий раз, место в стеке будет выделено заново и в него будет скопировано новое значение из вызывающей функции.
Аргументы функций - не единственный пример объектов с автоматическим классом хранения при вызове функции. Автоматические объекты, наряду с регистровыми и внутренними статическими, могут быть определены внутри любого блока операторов языка Си. Такие объекты данных запоминаются в сегменте стека. Следовательно, они создаются при входе в блок и уничтожаются при выходе из него. Тем самым, их время жизни совпадает с временем исполнения блока.
Существует два способа передачи аргументов в функцию: по значению (иногда говорят копированием) и по ссылке (на самом деле передача по ссылке делится на передачу по ссылке и по указателю, но смысл при этих двух подходах остаётся один).
При передаче аргументов по значению в стек записываются копии передаваемых аргументов, и функция работает с этими копиями в стеке. Значит функция не может изменить сами значения аргументов в переменных вызывающей функции. Вот пример передачи аргументов в функцию по значению:

```
void swap(int a, int b) {
  int t;
  t = a;
```

```
  a = b;
  b = t;
  printf("swap: a = %d, b = %d;\n", a, b);
}

main() {
  int x = 27, y = 32;
  printf("Перед обменом: x = %d; y = %d;\n", x, y);
  swap(x, y);
  printf("После обмена: x = %d; y = %d;\n", x, y);
}
```

Результатом выполнения будет вывод на дисплей сообщений:

```
Перед обменом: x = 27; y = 32;
swap: a = 32, b = 27;
После обмена: x = 27; y = 32;
```

При передаче аргументов по ссылке в функцию передается не значение аргумента, а ссылка на адрес размещения аргумента в памяти. Этот адрес записывается в соответствующий указатель. Имея этот адрес, функция может изменить содержимое памяти, т.е. реальное значение аргумента. Вот тот же пример, но передача аргументов производится по ссылке:

```
void swap(int *a, int *b) // Параметрами функции являются указатели на целое
{
  int t;
  t = *a; // Локальной переменной t присваивается значения, взятое из ячейки
памяти, на которую указывает первый аргумент функции
  *a = *b; // Содержимое ячейки памяти, на которую указывает второй аргумент
функции, присваивается ячейке памяти, на которую указывает первый аргумент
  *b = t; // Ячейке памяти, на которую указывает второй аргумент функции,
присваивается значение локальной переменной t
  printf("swap: a = %d, b = %d;\n", *a, *b);
}

main() {
  int x = 27, y = 32;
  printf("Перед обменом: x = %d; y = %d;\n", x, y);
  swap(&x, &y);
  printf("После обмена: x = %d; y = %d;\n", x, y);
}
```

Результатом выполнения будет вывод на дисплей сообщений:

```
Перед обменом: x = 27; y = 32;
swap: a = 32, b = 27;
После обмена: x = 32; y= 27;
```

И, чтобы закрыть тему основ передачи аргументов по ссылке, собственно передача по ссылке. Здесь параметр функции - это сразу ссылка на переменную (то есть её адрес) без промежуточного сохранения этого адреса в указатель. Параметр в такую функцию передаётся “как есть”. Явным ограничением на передачу параметров по ссылке (и по указателю) является использование в качестве

аргументов только lvalue, то есть нельзя будет передать в функцию числовой литерал (например, `swap(x, 32);`)

```
void swap(int &a, int &b) {
   int t;
   t = a;
   a = b;
   b = t;
   printf("swap: a = %d, b = %d;\n", a, b);
}

int main() {
   int x = 27, y = 32;
   printf("Перед обменом: x = %d; y = %d;\n", x, y);
   swap(x, y);
   printf("После обмена: x = %d; y = %d;\n", x, y);
}
```

```
Перед обменом: x = 27; y = 32;
swap: a = 32, b = 27;
После обмена: x = 32; y = 27;
```

Возврат значений из функции

Мы уже обсудили, что типом функции, указываемым перед ее именем, является на самом деле тип возвращаемого функцией значения. Функция может возвращать значения любого типа, кроме массива или другой функции. Но функция может возвращать указатель на любой тип, включая массив или функцию.

Указатели на функции.

Мы уже говорили выше о том, что в определении функции имя функции без аргументов является в языке Си указателем на эту функцию. Используя указатель на функцию удобно решать два типа задач:

- присвоив указатель функции, использовать его в дальнейшем как косвенную ссылку для вызова функции на исполнение;
- передавать функцию в качестве аргумента для других функций, передавая присвоенный функции указатель (функции, способные работать, в том числе по указателю, из других функций называются функциями обратного вызова, callbacks).

На первый взгляд использование косвенной ссылки для вызова функции может привести к увеличению трудоемкости написания программы, но на самом деле этот прием существенно уменьшает размер программы в случае вызова функции из библиотеки схожих функций, т.е. функций, имеющих тот же список параметров и возвращающих тот же тип данных.

Далее следует пример использования указателя на функцию для вызова функции `quad`, которая рассчитывает значения квадратного трехчлена для десяти аргументов `x` в заданном диапазоне от `first` до `last` с шагом `delta`. Функция ниже написана на языке C, для C++ тут не хватает явного указания типа `double` при объявлении указателя (`double (*fx)(double);`) . Рассчитанные значения выводятся на экран. Обратите внимание на формат объявления указателя на функцию `quad` и комментарии к этой строке, а также на то как инициализирован указатель на функцию.

```
#include <stdio.h>
int main() {
    double x;
```

```
    const double first = 0.0;
    const double last = 10.0;
    const double delta = 1.0;
    double (*fx)(); // Объявление указателя на функцию. Скобки вокруг (*fx)
обязательны, т.к. приоритет операции * ниже приоритета следующих скобок ().
    double quad(double); // Объявление прототипа функции.

    fx = quad; // Инициализация указателя на функцию.
    x = first;
    while (x <= last) {
        printf("f (%1f = %1f) \n", x, (*fx)(x));
        x += delta;
    }
}

double quad(double x) {
      double  a = 1, b = -3, c = 5;
      return   a * x * x + b * x + c;
}
```

Вызов этого кода даст в результате следующий вывод

```
f (0.000000 = 5.000000)
f (1.000000 = 3.000000)
f (2.000000 = 3.000000)
f (3.000000 = 5.000000)
f (4.000000 = 9.000000)
f (5.000000 = 15.000000)
f (6.000000 = 23.000000)
f (7.000000 = 33.000000)
f (8.000000 = 45.000000)
f (9.000000 = 59.000000)
f (10.000000 = 75.000000)
```

Вот тот - же пример, но с использованием массива указателей на функции. В массиве объявлены три указателя, один - для вызова определенной пользователем функции `quad`, два других - для вызова стандартных функций `sqrt` и `log` из библиотеки `math.h(cmath)`:

```
#define MAX 3
#define <math.h>

int main() {
   int i;
   double x;
   const double first = 0.0;
   const double last = 10.0;
   const double delta = 1.0;

   double (*fx[MAX])(double); // Объявление массива указателей на функции.
   double quad(double); // Объявление прототипа определенной пользователем
функции.

   fx[0] = quad; // Инициализация массива указателей на функции.
```

```
    fx[1] = sqrt;
    fx[2] = log;

    for(i = 0; i < MAX; i++) {
        x = first;
        while (x <= last) {
            printf("f (%1f = %1f)\n", x, (*fx[i])(x));
            x += delta;
        }
    }
}

double quad(double x) {
    double a = 1, b = -3, c = 5;
    return a * x * x + b * x + c;
}
```

Рекурсивные функции

Функция, вызывающая саму себя прямо или косвенно, называется рекурсивной. Очевидно, что рекурсивные функции должны обладать двумя свойствами:

- в теле функции должна содержаться проверка граничного условия, гарантирующего прекращение рекурсивного вызова;
- сама функция должна изменять значения аргументов, с которыми она вызывается рекурсивно, для того, чтобы избежать бесконечной рекурсии.

Компилятор не ограничивает число рекурсивных вызовов функции. При каждом вызове новые ячейки памяти класса `auto` (или даже `register`) выделяются для параметров функции и локальных переменных, так что их значения в предшествующих, незавершенных вызовах, остаются недоступными и не портятся (если, конечно, не используется косвенная адресация). Однако размер стека не бесконечен, поэтому слишком большая глубина рекурсии (слишком большое число вложенных вызовов) может привести к переполнению стека.

Для переменных, объявленных на внутреннем уровне с классом памяти `static` или `extern`, новые ячейки памяти не выделяются при каждом рекурсивном вызове. Выделенная им память сохраняется до завершения выполнения программы.

Обычно в качестве примера рекурсивной функции приводится пример вычисления факториала числа, мы приведем внешне похожий пример подсчета суммы целых чисел от 1 до 5:

```
main() {
  int sum(int); // Объявление рекурсивной функции
  int res;
  res = sum(5); // Вызов рекурсивной функции
  printf("Итого: sum = %d\n", res);
}

int sum(int n) // Реализация рекурсивной функции
{
  if(n > 1)
    return (n + sum(n - 1));
  else
    return 1;
}
```

При выполнении программы:

- вызывается sum(5) возвращает значение 5 + sum(4)

- вызывается sum(4) возвращает значение 4 + sum(3)
- вызывается sum(3) возвращает значение 3 + sum(2)
- вызывается sum(2) возвращает значение 2 + sum(1)
- вызывается sum(1) возвращает значение 1

Итого: sum(= 5 + 4 + 3 + 2 + 1) = 15

Inline-функции.

В C++ можно задать функцию, которая на самом деле не вызывается на исполнение, а ее тело встраивается в программу во всех местах ее вызова. В этом `inline`-функция похожа на макрос языка Си, но только похожа. Основных отличий два:

- запрос на встраивание функции в программу (**спецификатор** функции `inline`) является именно запросом, а не командой для компилятора. Т.е., если компилятор по какой-либо причине не в состоянии выполнить встраивание функции в программу, то `inline`-функция будет компилироваться и вызываться на исполнение как обычная функция, а запрос `inline` будет проигнорирован;
- при написании параметризованного макроса реальные аргументы (те, которые могут встретится в функции, вызывающей макроподстановку) замещают в тексте подстановки формальные параметры (те, которые указаны в круглых скобках при определении макроподстановки). Есть, конечно, меры для борьбы с такой путаницей в аргументах макроподстановок (о них мы поговорим подробнее на занятии, посвященном препроцессингу и макроподстановкам), но `inline`-функции в принципе исключают такую опасность.

В отличие от обычных функций `inline`-функции, в случае их успешного встраивания в программу по месту вызова, становятся никак не связанными с механизмами вызова функции, передачи параметров через стек и возврата функцией своего значения. А это означает значительную экономию времени и числа используемых машинных команд, особенно для коротких функций.

По вышеперечисленным причинам программисты и предпочитают в последнее время использовать `inline`-функции вместо макросов. Но есть и пара негативных моментов, касающихся `inline`-функций:

- частое использование `inline`-функций вместо обычных вызывает значительное увеличение размера исполняемого файла, впрочем этот недостаток присущ и макрорасширениям;
- `inline`-функция должна быть полностью определена до ее первого вызова

```
inline int sign(int x)  {
    return (x < 0 ? -1 : x ? 1 : 0);  // функция возвращает -1 если x<0; 1 если x>0; и 0 если x=0; по месту своего встраивания в программу
}
```

В классическом C тоже самое достигается при помощи параметризованной макроподстановки:

```
#define sign(x) ((x) < 0 ? -1 : (x) ? 1 : 0)
```

но о макроподстановках и о формате их строки поговорим на отдельном занятии, посвященном специально этой теме.

Перегрузка функций.

Перегрузка функций (function overloading) является одним из ключевых моментов языка C++, и формирует то ядро, вокруг которого развивается вся среда программирования C++. Если две или более функций в C++ носят одно и то же имя, то говорят, что они перегружены. При этом перегруженные функции должны отличаться либо типом, либо числом своих аргументов, либо и тем, и другим одновременно. Перегрузить функции в C++ легко: достаточно объявить и определить все возможные варианты. В старых компиляторах для этой цели еще применялось **ключевое слово overload**, но поскольку сейчас оно повсеместно вышло из употребления и не поддерживается большинством современных компиляторов, больше не будем о нем упоминать, просто его можно

встретить в программах на C++, написанных в середине 80-х годов прошлого века, да формат его использования есть в книгах по C/C++ Герберта Шилдта (Herbert Schildt). Перегрузка функции date для вывода в терминал даты в виде строки, либо в виде трех целых:

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

void data(const char *date); // дата в виде строки
void data(int month, int day, int year); // дата в виде чисел

int main() {
    date("8/23/99");
    date(8, 23, 99);
    return 0;
}

// Выдача даты из строки:
void data(const char *date) {
    cout << "Дата: " << date << "\n";
}

// Выдача даты из трех целых:
void data(int month, int day, int year) {
    cout << "Дата: " << month << "/";
    cout << day << "/" << year << "\n";
}
```

На компилятор возлагается задача выбора соответствующей конкретной версии перегруженной функции, а значит и метода обработки данных. В зависимости либо от типа данных, либо от их количества. В отличие от типа и количества аргументов, разные типы возвращаемых данных не могут служить достаточным отличием для перегрузки функций. Так в следующем примере:

```
int f1(int a);
double f1(int a);
...
f1(10);
...
```

у компилятора нет способа выяснить, какую версию перегруженной функции вызвать, что и приведет к сообщению об ошибке и прекращению компиляции.

Аргументы по умолчанию.

В языке C++ добавлена возможность указать компилятору при объявлении функции какое значение придать при вызове функции тому параметру, который не указан явно при вызове функции. Такой параметр называется аргументом по умолчанию(default argument). Вот пример такого присвоения:

```
void f(int a=0, int b=1); // объявление функции с аргументами по умолчанию
```

Далее все вызовы функции f() будут правильными:

```
f();      // a=0, b=1 по умолчанию
f(5);     // a=5, что указано явно при вызове, b=1 по умолчанию
f(5, 10); // a=5, b=10, что указано явно при вызове
```

Данный синтаксис сильно напоминает инициализацию переменных, передаваемых в функцию, и является, на самом деле, скрытой формой перегрузки функций.
Из приведенного примера понятно, что нельзя передать по умолчанию первый аргумент, и при этом явно передать второй аргумент. Возможна передача в функцию аргументов по умолчанию наряду с аргументами, передаваемыми обычным путем, но тогда аргументы по умолчанию должны быть последними в списке передаваемых аргументов (это поведение немного напоминает поведение аргумента переменной длины).
Аргументы по умолчанию можно задать либо при объявлении функции, либо при ее определении, только если такое определение предшествует первому вызову функции, но не одновременно в объявлении и определении. Аргументы по умолчанию должны быть либо константами, либо глобальными переменными. Они не могут быть локальными переменными, либо другими параметрами.

Указатели на перегруженные функции.

Мы уже обсуждали указатели на обычные функции в C. При этом указатель на функцию в C имеет тип просто абстрактного указателя `*f()`, поскольку имеется только одна функция, на которую он может указывать. Аналогично в C++ можно присвоить адрес функции указателю, и получить доступ к функции через этот указатель. Но в C++ ситуация осложняется тем, что функция может быть перегружена, и определить, на какой из вариантов ссылается указатель - на первый взгляд не такая простая задача. Выход из затруднения состоит в способе объявления указателя на перегруженную функцию в C++. А именно: объявления указателей на функцию должны соответствовать объявлениям перегружаемой функции.
Сказанное выше должно стать более понятным после рассмотрения следующего примера. В примере объявляется перегруженная функция `space()`. Первая ее версия выводит на экран некоторое число пробелов, заданных в переменной `count`. Вторая версия выводит на экран некоторое число иных символов, вид которых задан в переменной `ch`. В функции `main()` объявляются два указателя на эти функции, `void (*fp1)(int)` как указатель на функцию с одним параметром, и `void (*fp2)(int, char);` как указатель на функцию с двумя параметрами.

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

void space(int count) {
    for(; count; count--) cout << ' ';
}

void space(int count, char ch) {
    for(; count; count--) cout << ch;
}

int main() {
    void (*fp1)(int); // Объявление указателя на функцию с одним параметром
    void (*fp2)(int, char); // Указатель на функцию с двумя параметрами

    fp1 = space; // Получение адреса функции space(int)
    fp2 = space; // Получение адреса функции space(int, char)

    fp1(22); // Выводит 22 пробела
    cout << " | \n";

    fp2(10, 'x'); // Выводит 10 символов x
    cout << " | \n";
```

```
    return 0;
}
```

Как видно из примера, компилятор способен определить, какой из указателей будет ссылаться на какую из перегруженных функций.

Библиотеки функций.

В отличие от языков программирования – предшественников, язык Си изначально создавался как язык, максимально абстрагированный от архитектуры вычислительной платформы, от операционной системы, и даже от предназначения для решения задач какого-либо конкретного направления (как, например, Фортран предназначен для вычислений в сфере инженерии и науки, Cobol – для экономических расчётов, Clipper был “заточен” под базы данных, различные Ассемблеры – под аппаратно – зависимое программирование и т.д.). Основными механизмами, обеспечивающими такую гибкость языка, стали модульность программы и библиотеки функций. Т.е. для взаимодействия с операционной системой, или для математических вычислений, специалистами в данной отрасли разрабатываются модули – функции из элементарных операций и операторов, имеющихся в языке Си. Эти функции собираются и хранятся в библиотеках, но вовсе не являются необходимой частью языка. В зависимости от решаемой задачи, необходимый набор библиотек подключается к программе на этапе компиляции, а о существовании библиотек по другим отраслям знаний конкретный программист может иметь и весьма смутное представление.

Этот принцип настолько глубоко укоренен в идеологии языка Си, что даже такие важнейшие с точки зрения языка операции, как ввод (с клавиатуры) – вывод (на терминал) или работа с накопителями информации вынесены в библиотеки. Эти библиотеки получили общее название “**стандартных библиотек языка Си**”. Условно функции стандартных библиотек можно разделить на две категории:

- функции, которые имеются в библиотеке любой системы программирования на Си от любого разработчика, и эти функции могут быть использованы на любой аппаратной платформе и под любой операционной системой;
- функции, уникальные для конкретной системы программирования на Си конкретного разработчика, или обеспечивают доступ к уникальным возможностям конкретной операционной системы или конкретной аппаратной платформы.

Далее мы будем говорить о библиотеках первой из условных категорий, в которых стандартизованы как минимально необходимый набор функций, так и имена функций, и имена библиотек. Язык Си развивается, и, например, с появлением C++ все больше программистов предпочитают для ввода – вывода информации использовать соответственно операторы `>>` и `<<` (которые мы знаем как операции побитового сдвига соответственно вправо и влево, в C++ они играют ту же роль, но могут быть и “перегружены”). Так, например, инструкция:

```
cout << "Эта строка выводится на экран.\n";
```

может быть и удобнее, чем уже привычный вызов функции:

```
printf("Эта строка выводится на экран.\n");
```

дело вкуса, но функции `printf()` и `scanf()` по прежнему доступны в C++. И дело не только во вкусах. Существует большое количество ранее разработанных программ и библиотек, которые востребованы до сих пор. Для совместимости с этим наследием, а также для предотвращения конфликтов имен в библиотеках независимых между собой разработчиков компиляторов (так называемая “обратная совместимость”) и существует стандартная библиотека Си.

В стандартных библиотеках накоплено очень много функций, подробное их описание занимает сотни, если не тысячи страниц документации. Доступны и исходные коды этих функций. Мы попытаемся обратить внимание слушателей на наиболее востребованные из таких функций (ввод-вывод, работа с файлами, анализ символов, преобразование данных, обработка строк, математические функции, и

т.п.). К библиотечным функциям динамического управления памятью мы вернемся на отдельном занятии, посвященном этой теме.

Пространства имен.

Понятие пространства имен (`namespace`) появилось только в диалекте C++ языка C, и означает оно некоторую объявляемую область программы, выделение которой необходимо для того, чтобы избежать конфликтов имен идентификаторов. Точнее говоря, сама сущность пространства имен была и в Си, т.е. до появления C++, но там существовало всего лишь одно глобальное пространство имен, которое никак нигде не объявлялось, поскольку было единственным. А уже в глобальном пространстве существовали локальные области видимости имен (переменных и функций), о которых мы говорили выше.
Если вы откроете файл - исходник любой программы на языке Си старого стиля, то почти наверняка первой строкой (после комментариев о назначении программы и авторских правах) там будет что-то вроде:

```
#include <iostream.h>
```

т.е. подключение к программе библиотеки iostream, объявления переменных и функций которой находятся в заголовочном файле iostream.h, или какой-либо другой библиотеки, например stdio, через ее заголовочный файл stdio.h. В новом стиле языка C++ вместо этого будет 2 строки:

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

которые означают ровно тоже, что и одна строка старого стиля. Но, как всегда, обращайте внимание на синтаксис. Во-первых, в угловых скобках `<iostream>` вместо `<iostream.h>`. Это означает, что для подключения к программе библиотек нового стиля достаточно указать только имя библиотеки. Это имя теперь является всего лишь абстракцией, идентификатором, необходимым для того, чтобы объявления соответствующих прототипов и определений соответствовали ранее принятым стандартам языка Си. Кроме того, к именам многих стандартных библиотек в C++ добавляется символ ‘c’ впереди (без пробела). Так, например,
`<cmath>` соответствует `<math.h>`
`<cstring>` соответствует `<string.h>`
`<cstdio>` соответствует `<stdio.h>`
и т.д. Сделаны такие добавления были на ранних этапах развития C++, с целью отличать подключение библиотек Си от подключения аналогичных библиотек C++. Кстати, до сих пор практически все компиляторы C++ допускают одновременное использование библиотек старого и нового стилей, хотя, конечно, старый стиль вытесняется с каждым следующим стандартом всё больше. Однако, несмотря на то, что проходят годы, библиотеки старого стиля не планируют полностью выводить из обихода.

Строка `using namespace std;` означает как раз задание нового пространства имен (`namespace`) с именем `std` (обратите внимание на точку с запятой в конце строки). Содержание заголовков нового стиля помещается в пространство имен `std`, а само пространство имен `std` помещается в глобальное пространство имен. Таким образом обеспечивается совместимость заголовков старого и нового стилей.

Стало быть, если так вышло, что Вы работаете с каким-либо старым компилятором, и он не воспринимает инструкцию `namespace`, просто удалите строку `using namespace std;` и используйте заголовки старого стиля в глобальном пространстве имен. Правда при этом, если в Вашей программе используются стандартные библиотеки от разных разработчиков, есть шанс столкнуться с конфликтами имен между этими библиотеками. Поэтому будем считать это плохой затеей, лучше обновить компилятор.

Итак, путем объявления именованных областей при помощи инструкции `namespace` программист имеет возможность разделить глобальное пространство имен, в том числе и выделить

пространство имен (по существу - область видимости) для своих целей. Стандартная форма использования инструкции:

```
namespace имя_пространства  {
    объявления;
}
```

Вот пример объявления пространства имен MyNameSpace (обращайте внимание на синтаксис):

```
namespace MyNameSpace  {
    int i, j;
    void myfunc(int k)  {
      cout << k;
    }
    void seti(int x)  {
        i = x;
    }
    int geti()  {
        return i;
    }
}
```

К идентификаторам, объявленным в пространстве имен, в том же пространстве имен можно обращаться напрямую, как показано в примере выше. Но чтобы обратиться, допустим, к переменной `j` в этом примере, из той части программы, которая не входит в пространство имен `MyNameSpace`, перед именем члена пространства имен, к которому идет обращение, необходимо указать **оператор разрешения области видимости** `::` (двойное двоеточие):

```
MyNameSpace::j = 5;
```

Конечно, использовать оператор расширения области видимости один-два раза не представляет большого труда. Но если обращений к членам пространства имен много, как например обращений к библиотеке, часто предпочитают использовать инструкцию `using`, которую мы показывали выше. У этой инструкции две основные формы применения:

```
using namespace имя_пространства;
using имя_пространства::член;
```

При использовании первой формы все члены, определенные в указанном пространстве имен, становятся доступными в текущем (откуда идет обращение) пространстве имен, и появляется возможность работать с ними напрямую, без необходимости указывать для каждого члена оператор разрешения области видимости. При использовании второй формы инструкции using видимым становится только указанный в инструкции член пространства имен, и только с ним в дальнейшем можно работать напрямую, без применения при каждом обращении оператора разрешения области видимости.

В C++ можно объявлять более одного пространства имен с одним и тем же именем. Применяется такой прием чаще всего для разделения одного пространства имен на несколько файлов (по умолчанию `namespace` действует от своего объявления и до объявления другого пространства имен в том же файле, или до окончания файла, если другие пространства имен в нем не назначаются). Но можно и в пределах одного файла несколько раз объявлять пространство имен с одним и тем же именем. В таком случае это будет одно и то же пространство имен, поделенное на несколько частей. Одно пространство имен может быть вложено в другое. Но пространство имен не

может быть вложено в какую-либо другую область видимости, например, пространство имен нельзя объявлять внутри функции.

В C++ можно объявить т.н. **безымянное пространство имен** (`unnamed namespace`). Синтаксис такого объявления:

```
namespace  {
    объявления;
}
```

Идентификаторы, объявленные в `unnamed namespace` являются уникальными только в пределах того файла, в котором `unnamed namespace` объявлено.

Некоторые итоги раздела про пространства имен:

- наибольшую выгоду от применения пространства имен в C++ получили стандартные библиотеки. Теперь достаточно объявить их в **пространстве имен std** и использовать библиотеки со схожими именами идентификаторов в других пространствах имен;
- пространства имен можно объявлять только внутри глобального пространства имен, или внутри другого объявленного пространства имен, но не внутри функций;
- в небольших и средних по объему программах нет смысла объявлять пользовательские пространства имен, только **std** для библиотек, поскольку и так легко отследить все имена переменных, функций и классов;
- по своему действию пространства имен ограничивают области видимости подобно идентификатору `static`, о котором мы говорили на занятии по переменным, и который применим и к функциям.

Практическое задание

1. Задать целочисленный массив, состоящий из элементов 0 и 1. Например: [1, 1, 0, 0, 1, 0, 1, 1, 0, 0]. Написать функцию, заменяющую в принятом массиве 0 на 1, 1 на 0 (** без применения `if-else`, `switch`, `() ? :`);
2. Задать пустой целочисленный массив размером 8. Написать функцию, которая с помощью цикла заполнит его значениями 1 4 7 10 13 16 19 22;
3. * Написать функцию, в которую передается не пустой одномерный целочисленный массив, функция должна вернуть истину если в массиве есть место, в котором сумма левой и правой части массива равны. Примеры: checkBalance([1, 1, 1, || 2, 1]) → true, checkBalance ([2, 1, 1, 2, 1]) → false, checkBalance ([10, || 1, 2, 3, 4]) → true. Абстрактная граница показана символами ||, эти символы в массив не входят.
4. * Написать функцию, которой на вход подаётся одномерный массив и число n (может быть положительным, или отрицательным), при этом функция должна циклически сместить все элементы массива на n позиций.
5. ** Написать функцию из первого задания так, чтобы она работала с аргументом переменной длины.
6. ** Написать все функции в отдельных файлах в одном пространстве имён, вызвать их на исполнение в основном файле программы используя указатели на функции.

Используемые источники

1. *Брайан Керниган, Деннис Ритчи.* Язык программирования C. — Москва: Вильямс, 2015. — 304 с. — ISBN 978-5-8459-1975-5.

2. *Stroustrup, Bjarne* The C++ Programming Language (Fourth Edition)