Введение в программирование на C++

Артамонов Ю.Н.

Филиал «Котельники» университета «Дубна»

3 октября 2018 г.

Содержание

1 О некоторых отличиях языка C++ от языка C

2 Работа с динамическими объектами

Потоковый ввод/вывод в C++

Язык C++ был разработан Бьёрном Страуструпом как надмножество языка C, это значит, что в основном программисты могут использовать компилятор C++ для компиляции существующих программ на C без каких-либо изменений, а некоторые ведущие компании вообще не предлагают специальных средств для разработки программ на языке C (например, Visual Studio). Первое отличие, которое бросается в глаза - это новые воможности C++ по организации ввода - вывода данных. Вместо уже известных printf, scanf рекомендуется использовать соответственно cout, cin, а также операцию передачи данных в поток $<<$ (произносится «послать в») и операцию чтения данных из потока $>>$ (произносится «взять из»). В качестве примера рассмотрим две реализации одной и той же программы. Для компиляции программы на C++ в linux можно использовать команду:

g++ name.cpp -o nameout

Первая реализация программы примера

```
#include <stdio.h>
int main()
{
  int a;
  printf("Введите целое число:");
  scanf("%d",&a);
  printf("Итак, Вы ввели число %d\n", a);
  return 0;
}
```

Как видно, эта программа на C++ вообще ничем не отличается от аналогичной программы на языке C (разве что раньше мы допускали вольность не указывать тип возвращаемого значения у функции main.

Вторая реализация программы

```
#include <iostream>
using namespace std;
int main()
{
  int a;
  cout<<"Введите целое число:";
  cin >> a;
  cout <<"Итак, Вы ввели число " << a<<endl;
  return 0;
}
```

В данном примере отличий уже много. Рассмотрим каждое из них более детально.

Особенности второй реализации

Во-первых, как видно, вместо библиотеки стандартного ввода/вывода *stdio.h* подключается библиотека потокового ввода/вывода *iostream*. После этого становятся доступны функции cout, cin, endl. Правда для их удобного использования следует указать *пространство имен (nameplace)*, используемых по умолчанию. Для этой цели служит директива препроцессору using namespace std;. Если этого не писать, вызвать функции cout, cin, endl тоже можно, но уже длиннее: std::cout, std::cin, std::endl.

Во-вторых, для вывода на экран используется поток cout, в котором не нужно указывать спецификаторы %d и тому подобные вещи. Это уже много удобнее. Библиотека сама понимает, как выводить переменную соответствующего типа на экран (форматированный вывод с заданной точностью рассмотрим позже). Это относится и к cin - все, что нужно, просто перенаправить вывод из cin в переменную a. При этом следует обратить внимание, амперсанд & перед переменной, получающий ее физический адрес, также не нужен.

Особенности второй реализации (продолжение)

Следует также обратить внимание на использование endl для переноса строки (на самом деле специальная аббревиатура endl отвечает за закрытие буфера, что эквивалентно переносу строки). Конечно, вместо endl можно было использовать старый, добрый ecs-код:

```
cout <<"Итак, Вы ввели число " << a<<''\n'';
```

Можно вводить, выводить сразу несколько переменных разных типов, не заботясь об этом, например, так:

```
std::cin >> a >> b;
std::cout <<"Итак, Вы ввели число " <<  a << ", "<< b << std::endl;
```

В любом случае - это новая более гибкая возможность с сохранением старого функционала.

Особенности объявления переменных

В языке C все объявления должны находиться в начале блока до любых исполняемых операторов. В языке C++ объявления могут размещаться всюду, где может стоять исполняемый оператор, при условии, что они предшествуют использованию того, что объявляется. Например,

```
for (int i=0; i<=9; i++) cout << i;
cout <<endl;
```

При этом область действия локальной переменной в языке C++ начинается с ее объявления и распространяется до закрывающейся правой скобки }. Т.е. в нашем примере по выходу из цикла переменная i уже не видна. И код:

```
for (int i=0; i<=9; i++) cout << i;
cout <<endl;
cout << i;
```

приведет к сообщению об ошибке.

Использование прототипов

Прототипы функций позволяют компилятору C контролировать правильность вызова функций с точки зрения соответствия типов. В ANSI C они не являются обязательными.

В языке C++ для всех функций требуется их прототип!

Хотя функция, определенная в файле до первого обращения к ней, действует как прототип. Например, так можно:

```
#include <iostream>
using namespace std;
int f(int a, int b) {return a+b;}
int main() { cout <<f(6,8);  return 0; }
```

А так нельзя:

```
#include <iostream>
using namespace std;
 int main() { cout <<f(6,8);  return 0; }
int f(int a, int b) {return a+b;}
```

Использование прототипов (продолжение)

Здесь уже нужно использовать прототип

```
#include <iostream>
using namespace std;
int f(int, int);
 int main() { cout <<f(6,8);  return 0; }
int f(int a, int b) {return a+b;}
```

В языке C для определения пустого списка параметров в круглые скобки помещается ключевое слово void. Если в круглых скобках прототипа функции в C ничего не содержится, то для этой функции полностью отключается проверка параметров и не делается никаких предположенний относительно числа этих параметров и их типа. При обращениях к этой функции могут передаваться любые параметры без выдачи компилятором каких-либо ошибок.

В C++ при задании пустого списка параметров в круглых скобках также либо записывается void, либо вообще ничего не записывается. Но это лишь означает, что у функции по-честному нет параметров.

Использование прототипов (продолжение)

Реализация программы в виде набора функций хороша с точки зрения разработки программного обеспечения, но обращения к функциям связаны с накладными расходами времени исполнения (нужно время, чтобы переключиться на функцию, а затем вернуться обратно в место ее вызова). В C++ предусмотрены встроенные функции, позволяющие уменьшить накладные расходы при вызове функций - особенно для функций небольших размеров. Для этого можно использовать модификатор inline, который помещается перед типом возвращаемого функцией значения в определении функции и рекомендует компилятору генерировать в месте вызова функции ее копию кода, с тем, чтобы избежать вызова функции. Компромисс состоит в том, что вместо одной копии функции, в которую передается управление всякий раз при ее вызове, в программу вставляется несколько копий этой функции. Компилятор может игнриировать inline и обычно так и поступает, за исключением очень маленьких функций.

Параметры-ссылки

В языке C при всех обращениях к функциям параметны передаются по значению. Передача параметров по ссылке имитируется в C путем передачи в функцию указателя на объект и последущего доступа к этому объекту путем разыменования указателя в вызываемой функции. В языке C++ в функции можно объявлять параметры -ссылки. Чтобы указать, что параметр передается по ссылке необходимо после типа параметра в прототипе функции поместить символ амперсанда &. Например, можно написать такую программу на C++:

```
  #include <iostream>
using namespace std;
void f(int &);
void f(int & a) {a++;}
int main(){
  int a;
  cin>>a;
  f(a);
  cout<<a<<endl;}
```

Параметры-ссылки

Это было бы эквивалентно следующей программе в C

```
#include <stdio.h>
void f(int *);
void f(int* x) {(*x)++;}
int main()
{
  int a;
  scanf("%d",&a);
  f(&a);
  printf("%d", a);
}
```

Ссылки также могут служить в качестве псевдонимов для других переменных внутри некоторой функции. Например, :

```
int a = 666;
int &b = a;
b+=111;
cout<<a<<endl;
```

Параметры-ссылки

В последнем примере это эквивалентно увеличению переменной a на 111. Переменные-ссылки должны инициализироваться при их объявлении и не могут быть переназначены в качестве псевдонимов других переменных.

Динамическое распределение памяти

Для упрощения управленния динамической памятью в C++ предназначены операции new, delete.
В C для этих целей использовались malloc, free. При этом malloc требовало явного указания количества выделяемых байт с помощью sizeof. В C++ память выделить проще. Оператору new не нужно указывать размер памяти, он сам определит его по типу создаваемого объекта. При этом создаваемый объект можно еще инициализировать:

```
float *a = new float (3.14);
```

В этом примере выделяется память для объекта типа float, на который указывает указатель a.

Динамическое распределение памяти (продолжение)

Можно также создавать динамические массивы:

```
#include <iostream>
using namespace std;
int main()
{
  int *ar;
  ar = new int[3];
  cin >> ar[0]; cin >> ar[1]; cin >> ar[2];
  cout<<ar[0]<<ar[1]<<ar[2];
  delete [] ar;
  return 0;
}
```

Динамическое распределение памяти (продолжение)

В данном примере массив заполняется случайными числами:

```
#include <stdlib.h>
#include<iostream>
using namespace std;
int main()
{
  int n;
  cout<<"Введите размерность массива n =";
  cin>>n;
  int *h;
  h = new int [n];
  srand(time(0));
  for (int i = 0; i<n; i++){
    h[i] = 1+rand()%100;
    cout<<h[i]<<endl;
  }
  delete [] h;
  return 0;
}
```

Задание 1.1

Реализуйте алгоритм нахождения простых чисел с помощью решета Эратосфена.

Пример работы с динамическими двухмерными массивами

```
#include <stdlib.h>
#include<iostream>
using namespace std;
int main(){
  int n,m, **h; cin>>n>>m; srand(time(0));
  h = new int *[n];
  for (int i=0;i<n;i++) h[i] = new int [m];
  for (int i = 0; i<n; i++){
    for (int j=0;j<m;j++){
       h[i][j]=rand()%9;
       cout<<h[i][j];
     }
    cout<<endl;
  }
  for (int i = 0;i<n;i++)
    delete [] h[i];
  delete [] h;
  return 0;
}
```

Аналогичный пример - реализация с помошью функций

```
#include <stdlib.h>
#include<iostream>
using namespace std;
void zap(int, int, int **);
int main(){
  int n,m;
  cin>>n>>m;
  int **h;
  h = new int *[n];
  for (int i=0;i<n;i++) h[i] = new int [m];
  zap(n,m,h); srand(time(0));
  for (int i = 0; i<n; i++){
    for (int j=0;j<m;j++) cout<<h[i][j];
    cout<<endl;}
  for (int i = 0;i<n;i++)
    delete [] h[i];
  delete [] h;
  return 0;}
```

Аналогичный пример - реализация с помошью функций (продолжение)

```
void zap(int n, int m, int **ar)
{
  for (int i = 0; i<n; i++)
    for (int j=0;j<m;j++) ar[i][j]=rand()%9;
}
```

Задание 1.2

Реализуйте алгоритм транспонирования матрицы размером $n \cdot m$

Задание 1.3

Реализуйте алгоритм умножения матриц $n \cdot m$, $m \cdot k$.

Задание 1.4

Реализуйте алгоритм циклического сдвига строк матрицы на 1 элемент вправо.

Например, если матрица была вида: $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$,

то в итоге должна получиться матрица: $\begin{pmatrix} 2 & 3 & 4 & 1 \\ 6 & 7 & 8 & 5 \\ 10 & 11 & 12 & 9 \end{pmatrix}$

Задание 1.5

Задача Иосифа. По кругу располагаются n людоедов. Ведущий считает по кругу, начиная с первого, людоеды съедают каждого m-го собрата. Круг смыкается, счет возобновляется со следующего после съеденного. Так продолжается, пока не останется только один людоед. Найти номер оставшегося в живых людоеда.

Параметры, используемые по умолчанию

Часто при обращении к функциям может передаваться особое значение аргумента. Если такой аргумент при вызове функции опускается, в функцию автоматически передается значение по умолчанию. Аргументы по умолчанию должны быть самыми правыми (последними) аргументами в списке параметров функции. При вызове функции с двумя и более аргументами по умолчанию, если опущенный аргумент не является в списке параметров крайним справа, то все аргументы справа от него также должны быть опущены. Аргументы по умолчанию должны быть заданы при первом обяъвлении функции (обычно в прототипе). Рассмотрим пример сортировки массива с использованием пузырьковой сортировки с параметром направления сортировки по умолчанию по убыванию.

Пример пузырьковой сортировки с параметром по умолчанию

```
#include<iostream>
#include<stdlib.h>
using namespace std;
//Функция реализует пузырьковую сортировку
void sort( int *,int,bool = true);
void sort( int *ar, int size, int order){
  int i,j,c;
  if (order){
      for (int i=0; i<size; i++){
        for (int j=0;j<size-i-1;j++)
           if (ar[j]<ar[j+1]){
              c=ar[j]; ar[j]=ar[j+1];ar[j+1]=c;}}}
  else{
      for (int i=0; i<size; i++){
       for (int j=0;j<size-i-1;j++)
            if (ar[j]>ar[j+1]){
              c=ar[j]; ar[j]=ar[j+1];ar[j+1]=c;}}}}
```

Пример пузырьковой сортировки с параметром по умолчанию (продолжение)

```
//Функция заполняет массив случайными числами из заданного диапазона
void filling(int *, int, int);
void filling(int *ar, int diap, int size){
  srand(time(0));
  for (int i = 0; i<size; i++) ar[i] = 1+rand()%diap;}
//Функция печатает массив
void printing(int *, int);
void printing(int *ar, int size){for (int i = 0; i<size; i++) cout<<
  ar[i]<<endl;}
```

Пример пузырьковой сортировки с параметром по умолчанию (продолжение)

```
int main()
{
  int i,j; bool k;
  cout<<"Введите размер массива: "; cin>> i;
  cout<<"Введите верхнюю границу диапазона для генерации случайных чисел: ";
  cin>>j;
  cout<<"Введите направление сортировки: "; cin>>k;
  int *a; a = new int [i];
  filling(a,j,i); printing(a,i); cout<<endl<<endl;
  sort(a,i); printing(a,i);
  return 0;
}
```

Унарная операция разрешения области действия

В языках C и C++ возможно объявление локальных и глобальных переменных с одним и тем же именем. Однако в языке C пока мы находимся в области действия локальной переменной все ссылки на глобальную заблокированы (значение глобальной переменной недоступно). В языке C++ для этих целей предусмотрена *унарная операция разрешения области действия ::*, она позволяет получить доступ к значению глобальной переменной, если мы находимся в области действия локальной переменной. Однако использовать данный механизм нужно осторожно.

- Во-первых, переменная должна быть именно глобальной, т.е. объявлена вне какой-либо функции (и main в том числе);
- Во-вторых, лучше избегать использования глобальных переменных.

Пример использования операции разрешения области действия

```
#include<iostream>
using namespace std;
float value = 3.14;
int main()
{
  int value = 777;
  cout<<"local value = "<< value<<" global value = "<<::value<<endl;
  for(int value = 1;value<5;value++) cout<<value<<"  "<< :: value<<
  endl;
  return 0;
}

local value = 777 global value = 3.14
1  3.14
2  3.14
3  3.14
4  3.14
```

Перегрузка функций

В языке C объявление двух функций с одним и тем же именем в программе недопустимо. В C++ допускается определять разные функции с одним именем, если такие функции различаются набором параметров (по крайней мере их типами). Такая возможность называется *перегрузкой функций*. При вызове перегруженной функции автоматически подбирается нужная функция, исходя из анализа числа, типа и порядка параметров вызова. Перегрузка функций обычно используется для создания нескольких функций с одним и тем же именем, производящих схожие действия над разными типами данных. Перегруженные функции могут иметь разные типы возвращаемых значений, но при этом они все равно обязаны иметь разные списки параметров.

Пример использования перегруженных функций

```
#include<iostream>
using namespace std;
int square(int a){return a*a;}
double square(double a){return a*a;}

int main()
{
  int x=3;
  double y=3.14;
  cout<<"Целочисленная функция square: "<<square(x)<<endl;
  cout<<"Вещественная функция square: "<<square(y)<<endl;
  return 0;
}
```

Шаблоны функций

Перегруженные функции обычно используют для выполнения сходных операций над различными типами данных. Если операции для всех типов данных идентичны, то же самое можно сделать в более сжатой и удобной форме путем определения *шаблона функции*. Для этого пишется одно определение шаблона функции. Исходя из типа аргументов, задаваемых при вызовах функции, C++ автоматически генерирует объектный код для отдельных функций, обрабатывающих вызовы каждого типа. Все определения шаблонов функций начинаются с ключевого слова template, за которым в угловых скобках (< и >) следует список формальных параметров. Каждому формальному параметру предшествует ключевое слово class. Далее следует определение шаблона, которое не отличается от определения любой функции. Рассмотрим использование шаблона на примере предыдущего примера перегрузки функций.

Шаблоны функций (пример)

```
#include <iostream>
using namespace std;
template <class type1, class type2, class type3>
void summa(type1 a, type2 b, type3 &c)
{c=a+b;}
int main(){
  int a,b, res1;
  float c,d, res2;
  cout<<"int a="; cin >>a;
  cout<<"int b="; cin >>b;
  cout<<"float c="; cin >>c;
  cout<<"float d="; cin >>d;
  summa(a,b, res1);
  summa(c,d, res2);
  cout<<res1<<endl;
  cout<<res2<<endl;
  return 0;}
```

Шаблоны функций (продолжение)

Таким образом, шаблоны функций дают возможность порождать функции, которые выполняют одинаковые операции над различными типами данных, однако сам шаблон функции при этом определяется только один раз.

Еще один прример использования шаблонов

```
#include<iostream>
using namespace std;
template <class type1, class type2>
type1 g(type1 a, type2 b)
{return a+b;}

int main()
{
  int a,b;
  cout<<"a="; cin >> a;
  cout<<"b="; cin >> b;
  cout<<g(a,b)<<endl;
  float c,d;
  cout<<"c="; cin >> c;
  cout<<"d="; cin >> d;
  cout<<g(c,d)<<endl;
  return 0;
}
```

Работа со структурами данных

В языке C и в C++ можно использовать специальные составные типы данных - *структуры*. Структуры это новый тип данных, являющийся по сути контейнером для хранения нескольких других типов данных. Например, определим структуру, задающую некоторые атрибуты сотрудника организации:

```
struct person{
char last_name[20];
int year;
}
```

Работа со структурами данных (пример)

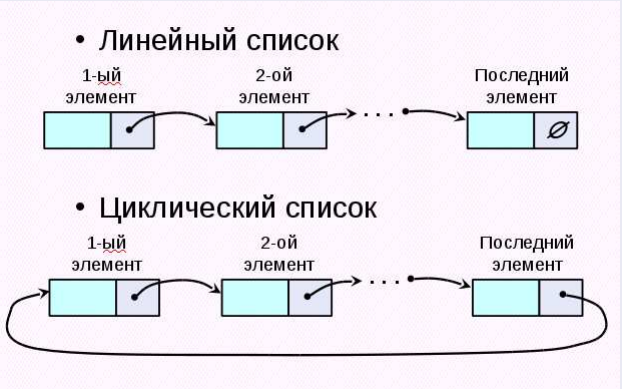
```
#include <iostream>
using namespace std;
struct person{
char last_name[20];
int year;
};
  int main(){
     person man, a, *woman;
     woman = &a;
     cout<<"Фамилия женщины: "; cin>>woman->last_name;
     cout<<"Возраст женщины: "; cin>>woman->year;
     cout <<woman->last_name<<endl;
     cout <<woman->year<<endl;
     cout<<"Фамилия мужчины: "; cin>>man.last_name;
     cout<<"Возраст мужчины: "; cin>>man.year;
     cout <<man.last_name<<endl;
     cout <<man.year<<endl;
     return 0;}
```

Структуры, ссылающиеся на себя

Кроме обычных структур, можно создавать структуры, ссылающиеся на себя.Такие структуры содержат в качестве элемента указатель, который ссылается на структуру того же типа. Например,

```
struct node
{
  int data;
  struct node *next;
}
```

Это позволяет создавать произвольные динамические структуры данных. В качестве примера рассмотрим структуру связанного списка:



Определение связанного списка

Связанный список - это линейный набор ссылающихся на себя структур, называемых узлами, и объединенных указателем-связкой. Доступ к связанному списку обеспечивается указателем на первый узел списка. Доступ к следующим узлам производится через связывающий указатель текущего узла. Указатель последнего узла списка устанавливается в NULL, отмечая конец списка. Каждый узел создается по мере необходимости. Узел может содержать данные любого типа, в том числе и другие структуры. Списки удобны, когда заранее неизвестно, сколько элементов данных будет содержать структура. Связанные списки являются динамическими, поэтому длина списка при необходимости может увеличиваться или уменьшаться. В отличие от массивов, элементы списка располагаются последовательно только логически. Физически в памяти они могут располагаться любым образом.

Пример определения связанного списка

```
#include<iostream>
using namespace std;
struct lst{
  int data;
  struct lst *next;
};

int main()
{
  lst *a,*b,*c, d;

  a = new  lst;
  b = new  lst;
  c = new  lst;

  cout<<"a.data = "; cin>>a->data;
  cout<<"b.data = "; cin>>b->data;
  cout<<"c.data = "; cin>>c->data;
```

Пример определения связанного списка (продолжение)

```
  a->next = b;
  b->next = c;
  c->next = &d;
  d.data= 666;
  d.next = NULL;
  lst *p;
  p=a;
  while (p != NULL)
    {
      cout << p->data<<endl;
      p=p->next;
    }
  delete [] a;
  delete [] b;
  delete [] c;
  return 0;
}
```

Пример определения стека

```
  #include<iostream>
using namespace std;
struct stack{
  int data;
  struct stack *next;
};
int pop(stack * &p)
{
  if (p != NULL)
    {
      int a = p->data;
      p=p->next;
      return a;
    }
  else
    return 0;
}
```

Пример определения стека (продолжение)

```
void push(stack * &p, int element)
{
  stack *current;
  current = new stack;
  current->data = element;
  current->next = p;
  p = current;
}
int main()
{
  stack *my_stack=NULL, *current = NULL;
  int key, a;
```

Пример определения стека (продолжение)

```
  do{
      cout<<"1 – Добавить в стек, 2 – удалить из стека; 3 – выход"; cin>>
  key;
      switch (key){
case 1:
  cout<<"a = "; cin >> a;
  push(my_stack, a);
  break;
case 2:
  cout<<pop(my_stack)<<endl;
  break;
case 3:
  break;
default:
  cout<<"Ошибка ввода"<<endl;}}
  while (key != 3);
```

Пример определения стека (продолжение)

```
    while (my_stack != NULL)
    {
      current = my_stack->next;
      delete [] my_stack;
      my_stack = current;
    }
  return 0;
}
```