

Программирование на языке C++

Зариковская Наталья Вячеславовна

Лекция 8

Динамическое распределение памяти

- При решении значительного числа задач, которые приходится решать в настоящее время, заранее трудно предположить, сколько оперативной памяти будет необходимо для сохранения данных и будут ли они нужны вообще.
- Это приводит к необходимости использования динамической памяти.
- Выделение памяти во время выполнения программы называется динамическим распределением памяти. Использование динамической памяти при решении различных задач позволяет производить выделение и освобождение памяти по мере необходимости.
- Примерами таких объектов являются узлы деревьев или элементы списка, которые входят в структуры данных, размер которых на этапе трансляции неизвестен.
- В языке C++ версий 3.11 и ниже не было средств для работы со свободной памятью, для этого использовались функции (они доступны и в C++) из стандартных библиотек.

Динамическое распределение памяти

- Ниже приводятся наиболее часто используемые, для различных моделей памяти, функции управления динамической памятью. В скобках указываются соответствующие заголовочные файлы, которые необходимо объявлять при их использовании.

- `alloca (malloc.h);` `farcoreleft (alloc.h);` `free (alloc.h,stdlib.h);`
- `allocmem (dos.h);` `farfree (alloc.h);` `heapcheck (alloc.h);`
- `bios_memsizes (bios.h);` `farheapcheck (alloc.h);` `heapcheckfree (alloc.h);`
- `brc (alloc.h);` `farheapcheckfree (alloc.h);` `heapchecknode (alloc.h);`
- `calloc (alloc.h,stdlib.h);` `farheapchecknode (alloc.h);` `heapwalk (alloc.h);`
- `coreleft (alloc.h,stdlib.h);` `farheapfillfree (alloc.h);` `malloc (alloc.h, stdlib.h);`
- `_dos_allocmem (dos.h);` `farheapwalk (alloc.h);` `realloc (alloc.h, stdlib.h);`
- `_dos_setbloc (dos.h);` `farmalloc (alloc.h);` `sbrk (alloc.h);`
- `farcalloc (alloc.h);` `farrealloc (alloc.h);` `setblock (dos.h).`
-

Динамическое распределение памяти

- Для отведения памяти используются функции `malloc()` и `calloc()`.
- `#include <stdlib.h>` or `#include <alloc.h>`
- `void *malloc(size_t size);`
- `#include <stdlib.h>`
- `void *calloc(size_t nitems, size_t size);`

- Функция `malloc()` принимает один параметр - размер выделяемого блока памяти в байтах и возвращает указатель на выделенный блок памяти. При невозможности выделить память возвращается значение `null`. Тип указателя `void*`, поэтому перед его использованием нужно явное приведение типа.

- Функция `calloc()` принимает два параметра - число элементов и размер элемента и инициализирует выделенную память нулями. Возвращает она тоже `void**`, поэтому перед его использованием также нужно явное приведение типа.

-
- `int pi = (int)malloc(sizeof(int));` //память для одного элемента типа int
- `int pia =(int)malloc(size*sizeof(int));` //для массива с элементами int размером size
- `int pia2 =(int)calloc(size,sizeof(int));` // то же самое с инициализацией нулями
-

Динамическое распределение памяти

- Для освобождения памяти, отведенной по `malloc()` или `calloc()` используется функция `free()`. Эта функция имеет вид:
- `#include <stdlib.h>`
- `void free(void *block);`
-
- У нее один параметр - указатель на память, которую нужно освободить. Он может быть любого типа.
- При выделении памяти для массива следует описать соответствующий указатель и присвоить ему значение при помощи функции выделения памяти.
- Например, при выделении памяти для одномерного массива `arr[20]` можно воспользоваться следующими операторами
- `float *arr;`
- `arr=(float*)(calloc(20,sizeof(float)));`

Динамическое распределение памяти

- Рассмотрим примеры использования функций выделения и освобождения памяти на примерах работы с векторами (строками).
- `#include <stdio.h>`
- `#include <string.h>`
- `#include <alloc.h>`
- `#include <process.h>`
- `int main(void)`
- `{ char *str;`
- `/* выделение памяти для строки */`
- `if ((str = (char *) malloc(15)) == NULL)`
- `{printf("память для строки не может быть выделена \n");`
- `exit(1); /* */`
- `}`
- `/* копирование строки "Добрый день!" */`
- `strcpy(str, "Добрый день!");`
- `/* вывод строки на экран дисплея */`
- `printf("Введена строка %s\n", str);`
- `/* освобождение памяти */`
- `free(str);`
- `return 0;`
- `}`

Динамическое распределение памяти

- Несколько сложнее выполняется выделение и освобождение памяти для векторов указателей на векторы (двумерные массивы). Для этого вначале необходимо распределить память для вектора указателей на векторы, а затем распределить память для вектора. Например, для выделения памяти для вектора указателей на векторы `arr[n][m]` можно воспользоваться следующими операторами
- `void main()`
- `{float **arr;`
- `int n,m,i;`
- `cin>>n>>m;`
- `arr=(float**)(calloc(m,sizeof(float*)));`
- `for(i=0;i<m;i++)`
- `arr[i]=(float*)(calloc(n,sizeof(float)));`
- `// далее выполняется какая-либо обработка`
- `/* освобождение памяти */`
- `for(i=0;i<m;i++)`
- `free(arr[i]);`
- `free(arr);`
- `}`

Динамическое распределение памяти

- В языке C++ использование функций `malloc()` или `calloc()` не имеет смысла. Операции `new` и `delete` выполняют динамическое распределение и отмену распределения памяти с более высоким приоритетом, нежели функции `malloc` и `free`. Свободная память в программах на языке C++ выделяется с помощью операции `new`, которая применяется к спецификации типа (абстрактному имени типа).
- `void* operator new(size_t);`
- `void operator delete(void*);`
- `size_t` - беззнаковый целочисленный тип, определенный в `<stddef.h>`.

- В этом случае выделится память, достаточная для размещения объекта такого типа и в результате будет возвращен указатель на выделенную память, например:
- `int *pi = new int;`
- Здесь выделена память для объекта типа `int`. Тип возвращаемого значения «указатель на `int`». Пустой указатель означает неудачное завершение операции. Этот случай возникает, когда недостаточный объем или слишком большая фрагментация распределяемой области памяти. Следует, однако, отметить, что в отличие от функции `malloc` оператор `new` не очищает выделенную память и содержит "мусор".
- Операция `new` удобнее тем, что она в качестве параметра получает тип создаваемого объекта, а не его размер и возвращает указатель на заданный тип, не требуя приведения типа.
- C++ допускает явную инициализацию выделяемой памяти для объекта любого типа с простым именем
- `int *pi = new int(100); // *pi == 100` – значение, записанное в
- `// выделенную динамическую память.`

Динамическое распределение памяти

- Массив выделяется в свободной памяти при помощи следующей за спецификацией типа размерности, которая заключена в квадратные скобки. При создании с помощью `new` многомерных массивов следует указывать все размерности массива. **Например,**
- `matr = new int[10][10][10]; // допустимо`
- `matr1 = new int[10][][10]; // нельзя`
-
- Размерность может быть выражением произвольной сложности. Операция `new` возвращает указатель на первый элемент массива. Например:
- `int i = 200;`
- `// ps указывает на массив из 400 элементов типа char.`
- `char *ps = new char[i*2];`
-
- Выделение и освобождение памяти для векторов указателей на векторы (двумерные массивы) с использованием оператора `new` более понятна, чем рассмотренный выше пример. Например:
- `// выделение памяти для вектора указателей (строк)`
- `a= new int *[nn];`
- `for (int j = 0; j < nn; j++)`
- `a[j] = new int[mm]; //выделение памяти для вектора (строк)`

Динамическое распределение памяти

- Корректная работа с указателями заключается в том, чтобы отвести память в динамической области и установить указатель на эту память. Следует помнить, что если эту память потом не освободить явно, то она не освободится и после окончания программы (это называется «мусором»).
- К памяти, отведенной в динамической области, нет иного доступа, кроме как через указатель, который ее адресует. Поэтому если этому указателю будет присвоен какой-либо другой адрес памяти, то та память, на которую он указывал, будет для программы потеряна и никогда не освободится:
- //отводим память в динамической области
- `int *ia = new int[100];`
- `int *ia2 = new int[100]; // еще отводим память`
- `ia = ia2; // ia указывает на ту же память, что и ia2, а память,`
- `//на которую указывал ia до присваивания недоступна,`
- `// и становится «мусором»`
-

Динамическое распределение памяти

- Память, выделенная с помощью операции `new`, будет занята до тех пор, пока программист явно ее не освободит. Для явного освобождения этой памяти используется операция `delete`, которая применяется к указателю, адресуемому динамический объект. Например:
- `int *pi = new int; // память отведена`
- `delete pi; // память освобождена`
- Здесь память, занимаемая `*pi`, снова возвращается в свободную память и впоследствии снова может быть выделена с помощью `new`.
- Для освобождения памяти отведённых массиву необходимо вставлять пару пустых квадратных скобок между `delete` и указателем :
- `int *parr = new int[100];`
- `delete [] parr;`
- Для освобождения памяти, отведённых массиву объектов, не являющихся классами, можно использовать оператор `delete` и без квадратных скобок:
- `delete parr; // аналогично delete [] parr;`
- Операция `delete` должна применяться только к памяти, выделенной с помощью операции `new`. Применение этого оператора к памяти, выделенной при помощи другого оператора, приведёт к ошибке. Однако, применение `delete` к нулевому указателю не считается ошибкой и просто игнорируется. Особенно опасно повторное применение `delete` к одному и тому же указателю.

Динамическое распределение памяти

- Одна из наиболее распространенных ошибок при использовании динамической памяти возникает тогда, когда два указателя адресуют одну и ту же область памяти, и после применения `delete` к одному из них применение `delete` к другому ведет к повторному освобождению уже освобожденной памяти и зависанию программы :
- `int *pi = new int[100];`
- `int *pi2 = pi; // pi и pi2 указывают на одну и ту же память`
- `delete pi; // освободить эту память`
- `// ...`
- `delete pi2; // зависание - повторное освобождение памяти`
-
- Сложность отладки состоит в том, что между двумя освобождениями может быть много операторов, и они могут быть в разных функциях и в разных файлах.
- К указателю на константный объект операцию `delete` применять нельзя, т.к. она все же меняет значение объекта:
-
- `const int *pi = new int(1024);`
- `delete pi; // ошибка компиляции`
-

Динамическое распределение памяти

- Свободная память может исчерпаться по ходу выполнения программы. По умолчанию операция `new` возвращает `0`, если нет достаточного количества свободной памяти. Поэтому необходимо учитывать, что `new` может вернуть `0`:
- `int *ia = new int[size];`
- `if (ia)`
- `{// что-то делаем с ia`
- `}`
- `else`
- `{// ошибка - не хватило памяти`
- `}`
- Для единообразной обработки подобных ошибок можно воспользоваться механизмом особых ситуаций или изменить стандартную обработку ситуации нехватки памяти. Для этого нужно установить свой обработчик такой ситуации.
-

Динамическое распределение памяти

- Программист может расположить объект в свободной памяти по определенному адресу. Для этого вызывается операция `new` в следующем виде:
- **`new (адрес-расположения) тип;`**
- Здесь адрес-расположения должен быть указателем. Для того, чтобы использовать этот вариант операции `new`, должен быть подключен заголовочный файл `new.h`. Эта возможность позволяет программисту предварительно выделять память, которая позже будет содержать объекты, созданные с помощью этой формы операции `new`, что может оказаться более надежным способом их размещения. Например:
- `int size = 10000;`
- `// здесь память для buf есть`
- `char *buf = new char[sizeof(int)*size]; // отводим память`
- `// ...`
- `// а здесь этой памяти уже могло бы не быть`
- `int *pi = new (buf) int[size];`
- `// не отводим память, а размещаем объекты типа int в buf в количестве size`
-

Краткие выводы из содержания лекции :

- 1) для размещения объектов в свободной памяти используется операция `new`, а для освобождения - операция `delete`.
- 2) если `new` не хватает памяти для размещения объекта, то она по умолчанию возвратит `0`.
- 3) если применить `delete` к нулевому указателю, то это действие проигнорируется и ошибки не будет.
- 4) если применить `delete` к указателю на память, которая уже была освобождена ранее, то программа, скорее всего, зависнет.
- 5) для работы со строками символов в C++ используются указатели вида `char *`.
- 6) если нужно скопировать одну строку в другую, не присваивайте их, а используйте `strcpy()`; если нужно сравнить две строки, используйте `strcmp()`. Основная работа со строками делается функциями обработки строк.
- Для их использования подключайте `string.h`
-

Организация взаимодействия функций в программе

- Как было определено выше, элементарной единицей программы на языке С есть функция. Функция предназначена для решения определённой задачи различной степени сложности. Функции, предназначенные для решения сложных задач, могут в свою очередь содержать обращения к произвольному числу функций, предназначенных для решения менее сложных задач. Таким образом, решение задачи на языке С++ предполагает её функциональную декомпозицию, когда функции более высокого уровня обеспечивают данными и воспринимают результат функций более низкого уровня.
- При вызове функции на время её работы выделяется память в рабочем стеке программы. Эта память автоматически освобождается по завершению работы функции. Каждому формальному параметру, согласно его спецификации выделяется память в том же рабочем стеке. При вызове функции, как отмечалось выше, путём указания имени функции, за которым в скобках через запятую указываются фактические параметры, производится инициализация формальных параметров значениями фактических параметров.
- Используя функции, следует различать три понятия - определение функции (описание действий, выполняемых функцией – исходный код), объявление функции (задание формы обращения к функции - прототип) и вызов функции.

Организация взаимодействия функций в программе

- Синтаксис C++ предусматривает, чтобы функция была либо определена, либо объявлена до её вызова. Объявление функции (задание прототипа функции) содержит имя функции, тип возвращаемого результата, список формальных параметров с указанием их типа или указание только типов формальных параметров.
- Список формальных параметров может заканчиваться запятой “, ” или запятой с многоточием “, ... ”, это означает, что число аргументов функции переменное. При этом предполагается, что функция имеет, по крайней мере, столько обязательных аргументов, сколько формальных параметров задано перед последней запятой в списке параметров. Такой функции может быть передано большее число аргументов, но над доп. аргументами не проводится контроль типов.
- В C++ определено несколько способов передачи параметров функции и получения результатов вычисления функции, вызывающей средой. Существует четыре базовых способа передачи параметров функции: вызов функции с передачей значений; вызов функции с передачей адресов переменных; вызов функции с использованием механизма ссылок при передаче параметров; посредством глобальных параметров. Но не каждый из этих способов обеспечивает возврат изменённых параметров в основную функцию (программу).

Вызов функции с передачей значений

- Этот способ передачи параметров обеспечивает передачу копий переменных в стек, организуемый при активизации функции. Таким образом, функция не получает доступа к фактическим параметрам вызова, а лишь к собственным локальным копиям фактических параметров. Их изменение при вычислении функции не приводят к изменению фактических параметров. При этом обеспечивается защита самих переменных от их изменения в функции. По окончании работы функции производится освобождение стека, и локальные значения теряются.
- **Пример:**
- `#include<iostream.h>`
- `int sum(int,int); // прототип функции`
- `void mane(void)`
- `{int a,b,c;`
- `cin >> a >> b; //вызов функции и передача параметров значений a и b`
- `c=sum(a,b);`
- `cout << c << endl;`
- `}`
- `int sum(int d, int l)`
- `{int f; // d и l это локальные копии фактических параметров a и b`
- `f=d+l;`
- `d=l=0; //изменения d и l не приводят к изменению фактиче- //ских параметров a и b`
- `return f ; // результат передаётся в точку вызова`
- `}`
- Следует отметить, что использование оператора `return` в приведенном выше примере обязательно, иначе функция была бы бессмысленна, поскольку результат её работы был бы потерян.

Вызов функции с передачей адресов (параметры – указатели)

- Этот способ передачи параметров обеспечивает передачу в стек адресов передаваемых данных, что позволяет функции работать непосредственно с данными. Формальные параметры имеют тип указатель. Таким образом, в функцию необходимо передавать адрес того объекта, который требуется изменить, а внутри функции разыменовывать параметр - указатель. Пример:
- `#include<iostream.h>`
- `sum(int,int,int*);`
- `void main()`
- `{int a,b,c=0;`
- `cin>>a>>b;`
- `sum(a,b,&c); // передаётся адрес -&c`
- `cout<<c<<endl;`
- `}`
- `void sum(intd,intl,int*f)`
- `{*f=d+l;`
- `}`
-

Вызов функций с использованием механизма ссылок

- Вызов функций с использованием механизма ссылок при передаче параметров обеспечивает доступ к передаваемым параметрам посредством определения их альтернативного имени. Для этого каждый формальный параметр, который необходимо изменить, описывают с типом ссылка. Поскольку ссылка является синонимом имени объекта, передаваемого в функцию, то все действия, которые производятся над ссылкой, являются действиями над самим объектом, а не с его локальной копией. В этом случае объект в стек не копируется, копируется только ссылка. **Например:**
- `#include<iostream.h>`
- `void sum(int,int&,int&);`
- `void main()`
- `{int a,b,c=0;`
- `cin >> a >> b;`
- `sum(a,b,c);`
- `cout << c << endl;`
- `}`
- `void sum(int d,int &l,int &f)`
- `{f=d+l; // имеем дело со ссылками l и f, т.е. действия производятся над переменными b и c`
- `}`
- Если внутри функции значение параметра следует сохранить, то параметр необходимо описывать как ссылку на константный объект, например
- `sum(int d, const int &l,int &f)`
- `{`
- `// изменение l приведет к ошибке компиляции`
- `}`
- Ссылки могут быть использованы в качестве результатов функции.

Вызов функции с передачей данных посредством глобальных параметров

- Этот способ передачи исходных данных в вызываемую функцию и возвращения результата вычислений путём использования глобальных параметров. Он основан на том факте, что глобальные параметры доступны любой функции в области её существования. Например
- `#include <iostream.h>`
- `int a,b,c;`
- `sum();`
- `main()`
- `{cin >> a >> b;`
- `sum();`
- `cout<<c<<endl;`
- `}`
- `sum()`
- `{c=a+b; //a,b,c- глобальные переменные`
- `}`
-

Вызов функции с передачей данных посредством глобальных параметров

- //В матрице d[5,5] заменить нулями все отрицательные //элементы, которые находятся на главной диагонали и над нею. Найти максимальный элемент среди элементов, которые расположены ниже побочной диагонали.

//В матрице d[5,5] заменить нулями все отрицательные //элементы, которые находятся на главной диагонали и над // нею. Найти максимальный элемент среди элементов, // которые расположены ниже побочной диагонали.

```
#include<stdio.h>
#include<iostream.h>
int d[4][4],b;
void nol ();
void poisk();
void main ()
{cout<<"введите значение элементов матрицы 5*5";
for(int i=0; i<5; i++){
for(int k=0; k<5; k++){
cin>>d[i][k];}}
nol();
cin>>b;
poisk();
}
```

```
void nol ()
{ int i,k;
i=0;
for(;i<5;i++){
k=i;
for(;k<5;k++)
if( d[i][k]<0)
d[i][k]=0;};
for(i=0; i<5; i++){
for(k=0;k<5;k++){
cout<<" "<<d[i][k];
cout<<endl; //обеспечивает переход на следующую строку
}
} }void poisk( )
{
int k,i,max;
{
max=d[0][0];
for(i=4; i>=0; i--)
{
for(k=0;k<i;k++)
if(max< d[i][k]) max=d[i][k];
}
printf("max=%d\n",max);
cin>>b;
}}
```

Вызов функции с передачей аргументов по умолчанию

- В языке C++ начиная с версии 3.11 и выше, определена возможность передачи значений аргументов функции по умолчанию. Этот способ передачи значений параметров используется в том случае, когда необходимо обеспечить передачу только части значений параметров, а не всех.
- Объявление значений параметров функции по умолчанию производится путём задания значений аргументов в прототипе функции. Эти задания производятся посредством оператора присваивания. При вызове функции те параметры, которые не указаны, принимают значения по умолчанию. Если параметры указаны, то значения по умолчанию игнорируются.
- **Например**, вычисление квадратной функции могло бы быть таким:

```
#include<iostream.h>
float ur(float x,float a=0.,float b=0.,float c=0.);
int main()
{ float a=1.,b=2.,c=3.,x=0.5,y;
  y=ur(x,a,b,c);
  cout<<"введены все аргументы"<<"\n";
  cout<<"y="<<y<<"\n";
  y=ur(x,a,b);
  cout<<"введены x,a и b"<<"\n";
  cout<<"y="<<y<<"\n";
  y=ur(x);
  cout<<"введено x"<<"\n";
  cout<<"y="<<y<<"\n";
  cin>>a;
}
float ur(float x,float a,float b,float c)
{return a*x*x+b*x+c;
}
```

Вызов функции с передачей аргументов по умолчанию

- На экране дисплея мы получим следующие результаты работы вышеприведенной программы.
- Введены все аргументы
- $y=4.25$
- введены x, a и b
- $y=1.25$
- введено x
- $y=0.$

Использование векторов в качестве аргументов функции

- При передаче массива в функцию C++ не делает копии данных (нет передачи по значению), а передаёт только его адрес (вызов по ссылке). Поэтому в качестве аргумента при вызове функции необходимо передать адрес начала массива. Учитывая, что имя массива является указателем константой, то при вызове функции в качестве фактического параметра достаточно указать только имя вектора. Реализация функции, в этом случае, в качестве формальных параметров должна содержать либо полное описание
 - **имя_функции(имя_вектора[размерность]);**
 - либо частичное
 - **имя_функции(имя_вектора[]);**
- Например:
 - `f (int b[],int n);` // прототип функции с частичным описанием вектора
 - `void main()`
 - `{int a[5];`
 - `f(a);` // вызов функции f где в качестве фактического
 - `//параметра передано имя вектора ' a'`
 - `}`
 - // реализация функции с частичным описанием вектора
 - `f (int b[], int n)`
 - `{}`
 -

Использование векторов в качестве аргументов функции

- Допустимо также использование в качестве формальных параметров указателей. В этом случае заголовок реализации функции, выше описанной задачи будет иметь следующий вид `f(int *b,int n)`.
- `#include<iostream.h>`
- `const int n=5;`
- `f(int *b,int n);// или f(int *,int);`
- `void main()`
- `{int a[n];`
- `cout<<"введите 5 чисел";`
- `for(int i=0;i<n;i++)`
- `cin>>a[i];`
- `int s=f(a,n);//вызов ф - ции f, в качестве фактического`
- `//параметра передано имя вектора 'a'`
- `cout<<" ответ "<<s;`
- `}`
- `f (int *b, int n)`
- `{int s=0;`
- `for(int i=0;i<n;i++)`
- `s+=b[i];`
- `return s;`
- `}`

Использование векторов в качестве аргументов функции

```
#include <iostream.h>
const int n=2;
void proiz(int A[ ][n],int B[ ][n],int C[ ][n],int n);
void main()
    {
int matraA[n][n],matraB[n][n],matraC[n][n],dop;
for(int i=0;i<n;i++)
for(int j=0;j<n;+j)
{cout<<"введите["<<i<<","<<j<<"]
элемент"<<"\n";
cin>>*(*(matraA+i)+j);}
cout<<"\n";
for(int i=0;i<n;i++)
for(int j=0;j<n;+j)
{cout<<"введите["<<i<<","<<j<<"]
элемент"<<"\n";
cin>>*(*(matraB+i)+j);}
cout<<"\n";
proiz( matraA,matraB,matraC,n);
cout<<"произведение матриц"<<"\n";
```

```
for(int i=0;i<n;i++)
{for(int j=0;j<n;j++)
cout<< "["<<i<<","<<j<<"]
"<<*(*(matraC+i)+j)<<" ";
cout<<"\n";}
cin>>dop;
}
void proiz(int A[ ][n],int B[ ][n],int C[ ][n],int n)
{ int s;
for(int k=0;k<n;k++)
{for(int i=0;i<n;i++)
{s=0;
for(int j=0;j<n;j++)
s+=A[k][j]*B[j][i];
C[k][i]=s;
} }
}
```

Функции с произвольным числом параметров

- При разработке программного обеспечения иногда трудно предусмотреть количество параметров, передаваемых функции. Для решения этой задачи в языке C, как отмечалось выше, предусмотрена возможность организации функции с произвольным числом параметров. В этом случае внутри списка параметров можно указать многоточие («...»), это означает, что далее могут следовать ещё какие-то параметры. В этом случае проверка соответствия типов формальных и фактических параметров не производится.
- В языке C допустимы две формы записи:
- `Имя_fun(список-параметров,...);`
- //запятую после последнего параметра можно не ставить
- `Имя_fun(...);`
- Примером функции с переменным числом параметров с использованием многоточия является функция `printf()` из стандартной библиотеки `stdio` - ввода-вывода языка C
- `int printf(const char * ...);`
- Тем самым устанавливается, что в вызове `printf()` должен быть по крайней мере один параметр типа `const char *` (символьная строка), а остальные могут быть, а могут и не быть.
- При вызове функции с переменным числом параметров следует обратить внимание, что следующие два описания не эквивалентны :
- `void имя_f();`
- `void имя_f(...);`
- В первом случае `имя_f()` объявлена как функция без параметров. Во втором случае `имя_f()` объявлена как функция с нулем или более параметров.

Вызов функции посредством указателя на функцию

- Как отмечалось выше, функция может быть вызвана не классическим способом - посредством указателя на функцию. Указатель на функцию содержит адрес первого байта выполняемого кода функции.
- Указатель на функцию должен быть объявлен и инициализирован по определённым правилам.
- Указатель на функцию объявляется следующим образом:
- **[тип] (*и_указ_функции) (сп. форм. параметров| void);**
- где [тип] - тип возвращаемого результата работы функции; (*и_указ_функции)- указатель на функцию; сп. форм. параметров - определение формальных параметров с указанием их типа.
- При этом скобки здесь обязательны. Дело в том, что операция вызова функции (круглые скобки в ее объявлении) имеет более высокий приоритет, чем операция разыменования.
- Поэтому, если мы напишем:
- **[тип] *и_указ_функции (сп. форм. параметров| void);**
- то у нас получится объявление функции, возвращающей указатель на заданный тип (звездочка относится к типу результата).
- **Например:**
- `char * strcpy(char*S1, const char*S2);`
- //функция возвращает указатель на символьный литерал S1 в который копируется строка S2
- Инициализация функции выполняется обычным образом
- **и_указ_функции= имя_функции; ,**
- где имя_функции - имя некоторой функции с идентичными указателю на функцию формальными параметрами.

Вызов функции посредством указателя на функцию

- В качестве примера ниже приводится простейшая программа вычисления суммы и произведения элементов заданной матрицы с использованием указателя на функцию.
- `#include<iostream.h>`
- `const int n=3;`
- `int sum(int A[][n],int); //правильно A[n][n] ,A[][n]`
- `int pr(int A[n][n],int n);`
- `void main()`
- `{int ii,A[n][n],y,proiz;`
- `int (*p)(int [][n],int);`
- `p=sum; // Инициализация функции sum`
- `for (int i=0;i<n;i++)`
- `for (int j=0;j<n;j++)`
- `{cout<<"введите очередной ["<<i<<","<<j<<"]элемент\n";`
- `cin>>A[i][j];}`
- `cout<<"\n";`
- `y= p(A,n); //вызов функции sum`
- `cout<<"y="<<y<<"\n";`
- `p=pr; // Инициализация функции pr`
- `proiz= p(A,n); // вызов функции pr`
- `cout<<"proiz="<<proiz<<"\n";`
- `}`

Вызов функции посредством указателя на функцию

- `int sum(int A[n][n],int n) // A[][]-ошибка и правильно A[][n] и A[n][n]`
- `{ int s=0;`
- `for(int i=0;i<n;i++)`
- `for(int j=0;j<n;j++)`
- `s+=*(A+i+j);`
- `return s;`
- `}`
- `int pr(int A[n][n],int n)`
- `{int p=1;`
- `for(int i=0;i<n;i++)`
- `for(int j=0;j<n;j++)`
- `p*=*(A+i+j);`
- `return p;`
- `getchar();}`

- Синтаксис языка C++ допускает использование векторов указателей на функции. Объявление векторов и их инициализация для выше приведенного примера будет иметь следующий вид:
- `int (*p[2])(int [][n],int)={sum,pr};// Инициализация функции sum и функции pr`

Вызов функции посредством указателя на функцию

- При использовании вектора указателей на функции функция main имела-бы следующий вид:
- `void main()`
- `{int ii,A[n][n],y,proiz;`
- `int (*p[2])(int [][n],int)={sum,pr};// Инициализация функции sum и функции pr`
- `for (int i=0;i<n;i++)`
- `for (int j=0;j<n;j++)`
- `{cout<<"введите очередной ["<<i<<","<<j<<"]элемент\n";`
- `cin>>A[i][j];}`
- `cout<<\n';`
- `y= p[0](A,n); //вызов функции sum`
- `proiz= p[1](A,n); // вызов функции pr`
- `cout<<"y="<<y<<\n';`
- `cout<<"proiz="<<proiz<<\n';`
- `getchar();`
- `}`

Вызов функции посредством указателя на функцию

- Использование указателей на функцию нашло широкое применение, когда необходимо передать функцию как параметр другой функции. Например:
- `#include<iostream.h>`
- `#include<stdio.h>`
- `const int n=3;`
- `int A[n][n],y,k=2,l;`
- `int sum(int A[][n],int);`
- `int pr(int A[n][n],int n);`
- `// Указатель на функцию - аргумент функции`
- `int ob(int (*p)(int A[][n],int));`
- `void main()`
- `{int (*p)(int [][n],int);`
- `for (int i=0;i<n;i++)`
- `for (int j=0;j<n;j++)`
- `{cout<<"введите очередной ["<<i<<","<<j<<"]элемент\n";`
- `cin>>A[i][j];}`
- `cout<<"\n";`
- `y=ob(sum);`
- `cout<<"y="<<y<<"\n";`
- `y=ob(pr);`
- `cout<<"pr="<<y<<"\n";`
- `getchar();`
- `}`

Вызов функции посредством указателя на функцию

- `int ob(int (*p)(int A[][n],int n))`
- `{ int y;`
- `y= p(A,n);`
- `return y;`
- `}`
-
- `int sum(int A[n][n],int n)`
- `{int s=0;`
- `for(int i=0;i<n;i++)`
- `for(int j=0;j<n;j++)`
- `s+=*(A+i)+j);`
- `return s;`
- `}`
- `int pr(int A[n][n],int n)`
- `{int p=1;`
- `for(int i=0;i<n;i++)`
- `for(int j=0;j<n;j++)`
- `p*=*(A+i)+j);`
- `return p;`
- `}`

Вызов функции посредством указателя на функцию

- Язык допускает так же использование вектора указателей на функции в качестве аргумента функции. Это позволяет организовывать коммутаторные функции. Например, для нашего примера, это могло-бы выглядеть так:

```
#include<iostream.h>
#include<stdio.h>
const int n=3;
int A[n][n],y,k,l,kk;
int sum(int A[ ][n],int );
int pr(int A[n][n],int n);
int ob(int (*p[ ])(int A[ ][n],int ),int);
void main()
{
int (*p[ ])(int [ ][n],int )={sum,pr};
for (int i=0;i<n;i++)
for (int j=0;j<n;j++)
{cout<<"введите очередной ["<<i<<","<<j<<"]элемент\n";
cin>>A[i][j];}
cout<<\n';
cout<<"введите k \n";
cin>>kk;
y=ob(p,kk);
cout<<"y="<<y<<\n';
cout<<"введите k \n";
cin>>kk;
y=ob(p,kk);
cout<<"pr="<<y<<\n';
getchar();
}
```

```
int ob(int (*p[2])(int A[ ][n],int n) ,int kk)
{ int y;
y= p[kk](A,n);
return y;
}

int sum(int A[n][n],int n)
{int s=0;
for(int i=0;i<n;i++)
for(int j=0;j<n;j++)
s+=*(A+i+j);
return s;
}

int pr(int A[n][n],int n)
{int p=1;
for(int i=0;i<n;i++)
for(int j=0;j<n;j++)
p*=*(A+i+j);
return p;
}
```

Перегружаемые функции

- В C++ допускается использование двух и более функций с одним и тем же именем, если эти функции отличаются друг от друга числом или типом используемых параметров. Если такие функции объявлены внутри одной и той же области видимости, то они называются перегружаемыми. Компилятор автоматически определяет вызов необходимой функции путём поиска соответствий количества и типа формальных параметров соответствующем фактическом вызове функции. Например:
- `int sum(int a, int b)`
- `{return (a+b);}`
- `double sum(double a, double b)`
- `{return(a+b);}`
- `double sum(double a, double b, double c)`
- `{return(a+b+c);}`
-
- Приведенные выше функции отличаются друг от друга следующим образом: первая от второй типом формальных параметров и типом возвращаемого результата; первая от третьей количеством и типом формальных параметров и типом возвращаемого результата; вторая от третьей количеством формальных параметров.
- При использовании перегружаемых функций следует придерживаться следующего правила- функции должны отличаться по количеству параметров или по типу этих параметров.
- Следует обратить внимание на недопустимость организации перегружаемых функций отличающихся только типом возвращаемого результата или когда параметры идентичны (например: `int` и `const int &`; `int` и `int &`).

Шаблонные функции

- При решении значительного числа задач часто приходится иметь дело с функциями, у которых алгоритм решения задачи одинаков и отличны только данные. Например, алгоритмы решения задач поиска максимума, минимума, сортировки и ряд других. Использование перегружаемых функций, с учётом строгой типизации языка C++, потребует многочисленное переопределение этих функций для каждого поддерживаемого типа данных, даже притом, что коды каждой версии практически одинаковы.
- Решение этой задачи состоит в использовании шаблонов функции. Шаблон функции представляет собой некоторую обобщённую функцию для семейства связанных перегружаемых функций, предназначенный для решения конкретной задачи. Определение функции обычно производится в заголовочном файле, и имеют следующий вид:
 - **template <class type>**
 - **тип myfunc(type param1, type param2)**
 - **{**
 - **/// операторы тела функции**
 - **}**
- где: `template<class type >` зарезервированное строковое выражение, указывающее компилятору, что `type` есть неопределённый пользователем идентификатор типа; `тип` - тип шаблонной функции; `myfunc` – произвольный идентификатор шаблонной функции; `(type param)`- формальные параметры, хотя бы один из которых должен иметь либо тип `type`, либо указатель на переменную типа `type` (`type *`) или ссылку на переменную типа `type` (`type ¶m`); "операторы тела функции" - схема реальных операторов, которые будут сгенерированы компилятором в подходящую функцию в соответствии с типом данных прототипа функции (если функция определена в заголовочном файле) или реально используемым при вызове.

Шаблонные функции

- Таким образом, тип данных шаблонной функции играет роль дополнительного параметра.
- В шаблоне функции может быть определено несколько меток-заполнителей типов данных, а также использованы параметры предопределённых типов. Например:
- **template <class type1, class type2>**
- **void myfunc(type1 a, type2 b, int c)**
- **{**
- **//”операторы тела функции”**
- **}**
-
- При использовании в программе шаблонной функции компилятор генерирует подходящую функцию в соответствии с типом данных, реально используемым при вызове.

Шаблонные функции

Например:

```
// Пример родовой функции или шаблона
// сортировки методом Шелла
```

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

```
#include <stdlib.h>
```

```
#include <stdio.h>
```

```
#include <time.h>
```

```
const int n=10;
```

```
// Это шаблон функции сортировки методом
// Шелла
```

```
template <class T> void chelsort(T arr[],int col)
```

```
//сортировка методом Шелла
```

```
{int h[] = {9,5,3,1};
```

```
for(int i=0,deapazon=h[0]; i<4; deapazon=h[++i])
```

```
{ for(int i=deapazon; i < col; i++)
```

```
for(int j=i-deapazon;j>=0&&arr[j]>arr[j+deapazon];
```

```
j-=deapazon)
```

```
{ T temp = arr[j];
```

```
arr[j] = arr[j+deapazon];
```

```
arr[j+deapazon] = temp;
```

```
}
```

```
}
```

```
}
```

```
int main(void)
```

```
{ int arr[n], i,col=n;
```

```
float arr1[n];
```

```
randomize();
```

```
// формирование массива целых чисел
```

```
for(i=0; i<n; i++)
```

```
arr[i]= rand() % 100;
```

```
chelsort(arr,col);
```

```
for(i=0; i<n; i++)
```

```
cout<<arr[i]<<"\n";
```

```
// формирование массива вещественных
чисел
```

```
for(i=0; i<n; i++)
```

```
arr1[i]= rand()/ 100.;
```

```
chelsort(arr1,col);
```

```
for(i=0; i<n; i++)
```

```
cout<<arr1[i]<<"\n";
```

```
getchar();
```

```
return 0;
```

```
}
```


Шаблонные функции

- При необходимости можно переопределить генерацию шаблонной функции для конкретного типа или конкретной реализации с помощью нешаблонной функции. Например:
- // Переопределение родовой функции-шаблона сортировки методом Шелла

```
#include <iostream.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
const int n=10;
// Это шаблон функции сортировки методом
Шелла
template <class T> void chelsort(T arr[],int col)
//сортировка методом Шелла
{int h[] = {9,5,3,1};
.....
}
// Здесь переопределяется родовая версия
функции chelsort
void chelsort(float arr[],int col)
{ int h[] = {13,9,5,3,1};//изменения
.....
}
```

```
int main(void)
{ int arr[n], i,col=n;
  float arr1[n];
  randomize();
  // формирование массива целых чисел
  .....
  // формирование массива вещественных
чисел
  .....
  chelsort(arr1,col); // вызов
перегруженной функции chelsort
  .....
  getchar();
  return 0;
}
```

Если шаблонные функции определены в заголовочном файле, то в программе достаточно указать только прототипы этих функций. Компилятор ищет шаблон функции, совпадающий по типу возвращаемого значения, количеству формальных параметров и типу тех формальных параметров, которые определены.

Функции inline

- В языке C++ нашли широкое применение встраиваемые функции (inline). Эти функции встраиваются, в местах вызова этих функций, в рабочий код программы на этапе компиляции, что обеспечивает экономию времени затрачиваемого на вызов функции.
- Определение встраиваемых функций отличается от описанных выше только наличием зарезервированного слова «inline» перед идентификатором типа функции, при её описании. В C++5 допускается использование встраиваемых функций без использования зарезервированного слова «inline», когда эта функция описывается непосредственно в заголовочном файле при описании класса.
- Следует знать, что ни все функции определённые как встраиваемые компилятор действительно определит как встраиваемые. Считается, что целесообразно делать функцию встроенной только в том случае, когда объем её кода меньше, чем размер кода, который потребуется для вызова ее извне. Пример может быть таким.

-

Функции inline

- `#include<iostream.h>`
- `inline float ur(float x,float a=0.,float b=0.,float c=0.);`
- `int main()`
- `{ float a=1.,b=2.,c=3.,x=0.5,y;`
- `y=ur(x,a,b,c);`
- `cout<<"введены все аргументы"<<"\n";`
- `cout<<y<<"\n";`
- `y=ur(x,a,b);`
- `cout<<"введены x,a и b"<<"\n";`
- `cout<<y<<"\n";`
- `y=ur(x);`
- `cout<<"введен x"<<"\n";`
- `cout<<y<<"\n";`
- `cin>>a;`
- `return 0;`
- `}`
- `inline float ur(float x,float a,float b,float c)`
- `{`
- `return a*x*x+b*x+c; }`
-

Краткие выводы из содержания лекции :

- 1) различают определение и объявление функции. Определение функции - это ее заголовок и тело.
- 2) перед вызовом функция должна быть объявлена. Объявление функции - это ее прототип, включающий имя, типы параметров и тип возврата.
- 3) объявления функций, вызываемых в нескольких файлах, нужно вынести в заголовочный файл, а потом подключить его во все эти файлы.
- 4) если типы фактических параметров функции и типы ее параметров, указанные в ее объявлении (прототипе) не совпадают и нет неявного преобразования то будет ошибка компиляции.