Вычислительная техника и компьютерное моделирование в физике

Лекция 3

Зинчик Александр Адольфович
zinchik_alex@mail.ru

Указатели

- на объект
- на функцию
- на void

Указатель на функцию содержит адрес в сегменте кода, по которому располагается исполняемый код функции.
Формат:

тип (*имя) (список_типов_аргументов);

```
int (*fun) (double, double);
```

Инициализация указателей

1.Присваивание указателю адреса существующего объекта:

- с помощью операции получения адреса:

```
int a = 5;
int* p = &a;
int* p (&a);
```

- с помощью значения другого инициализированного указателя:

```
int* r = p;
```

- с помощью имени массива или функции:

```
int b[10];
int* t = b;
void f(int a ){ /* ... */ }
void (*pf)(int);
pf = f;
```

2. Присваивание указателю адреса области памяти в явном виде:

```
char* vp = (char *)0xB8000000;
```

3.Присваивание пустого значения:

```
int* suxx = NULL;
int* rulez = 0;
```

4. Выделение участка динамической памяти и присваивание ее адреса указателю:

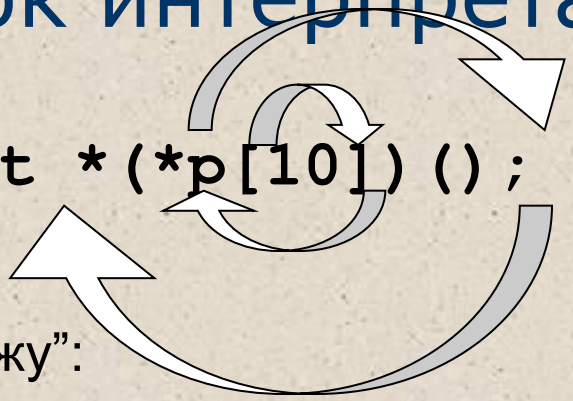
- с помощью операции new:

```
int* n = new int;
int* m = new int (10);
int* q = new int [10];
```

- с помощью функции malloc:

```
int* u = (int *)malloc(sizeof(int));
```

Порядок интерпретации описаний



массив из 10 указателей на функции без параметров, возвращающих указатели на int

“изнутри наружу”:

- если справа от имени имеются квадратные скобки, это массив, если скобки круглые — это функция;
- если слева есть звездочка, это указатель на проинтерпретированную ранее конструкцию;
- если справа встречается закрывающая круглая скобка, необходимо применить приведенные выше правила внутри скобок, а затем переходить наружу;
- в последнюю очередь интерпретируется спецификатор типа.

Освобождение памяти:

delete n; delete [] q; free (u);

Операции с указателями

- разадресация
- присваивание
- сложение с константой
- вычитание
- инкремент (++), декремент (– –)
- сравнение
- приведение типов

При работе с указателями часто используется операция получения адреса (&).

Операция разадресации

```
char a; char * p = new char;
*p = 'Ю'; a = *p;
```

```
#include <stdio.h>
int main(){
   unsigned long int A = 0Xcc77ffaa;
   unsigned int* pint = (unsigned int *) &A;
   unsigned char* pchar = (unsigned char *) &A;
   printf(" | %x | %x |", *pint, *pchar);
}
```

Присваивание указателей

Присваивание без явного приведения типов допускается в двух случаях:

- указателям типа void*;
- если тип указателей справа и слева от операции присваивания один и тот же.

Приведение типов:

(тип) выражение

Арифметические операции с указателями

- **инкремент и декремент**

```
short * p = new short [5];
p++;
long * q = new long [5];
q++;
```

```
*p++ = 10;   //*p = 10; p++;
(*p)++;
```

- **сложение с константой**
- **разность**

Операция получения адреса &

Унарная операция получения адреса & применима к величинам, имеющим имя и размещенным в оперативной памяти. Таким образом, нельзя получить адрес скалярного выражения, неименованной константы или регистровой переменной.

```
int a = 5;
int* p = &a;
```

Ссылки

```
int kol;
int& pal = kol;
 const char& CR = '\n';
```

тип & имя;

- Переменная-ссылка должна явно инициализироваться при ее описании, кроме случаев, когда она:
  - является параметром функции
  - описана как extern
  - ссылается на поле данных класса
- После инициализации ссылке не может быть присвоена другая переменная.
- Тип ссылки должен совпадать с типом величины, на которую она ссылается.
- Не разрешается определять указатели на ссылки, создавать массивы ссылок и ссылки на ссылки.

Классы потоков

- ios — базовый класс
- istream — входные
- ostream — выходные
- iostream — двунаправленные
- ifstream — входные файловые
- ofstream — выходные файловые
- fstream — двунаправленные файловые

Заголовочные файлы

- <ios>
- <istream>
- <ostream>
- <iostream>
- <fstream>
- <iomanip>

Преимущества и недостатки потоков

- Основным преимуществом потоков по сравнению с функциями ввода/вывода, унаследованными из библиотеки C, является контроль типов, а также расширяемость; потоки могут работать с расширенным набором символов wchar_t.
- К недостаткам потоков можно отнести снижение быстродействия программы

Стандартные потоки

Объект	Класс
cin	istream
cout	ostream
cerr	ostream
clog	ostream

операции извлечения из потока >> и помещения в поток << определены путем перегрузки операций сдвига

Операции << и >>

- *Числовые значения* можно вводить в десятичной или шестнадцатеричной системе счисления (с префиксом 0x) со знаком или без знака. Вещественные числа представляются в форме с фиксированной точкой или с порядком.
- При *вводе строк* извлечение происходит до ближайшего пробела
- *Значения указателей* выводятся в шестнадцатеричной системе счисления.
- Под любую величину при выводе отводится столько позиций, сколько требуется для ее представления.

Форматирование данных

В потоковых классах форматирование выполняется с помощью

- флагов
- манипуляторов
- форматирующих методов.

Флаги и форматирующие методы

```
#include <iostream.h>
int main(){
   long a = 1000, b = 077;
   cout.width(7);
 cout.setf(ios::hex | ios::showbase | ios::uppercase);
   cout << a;
   cout.width(7);  cout << b << endl;
   double d = 0.12, c = 1.3e-4; cout.setf(ios::left);
   cout << d << endl;
   cout << c;
   return 0;
}
```

Манипуляторы

```
#include <iostream.h>
#include <iomanip.h>
int main(){
   double d[] = {1.234, -12.34567, 123.456789,
            -1.234, 0.00001};
cout << 13 << hex << ' ' << 13 << oct << ' ' <<
   13 << endl;
cout   << setfill('.') << setprecision(4)
    << setiosflags(ios::showpoint | ios::fixed);
   for (int i = 0; i < 5; i++)
    cout << setw(12) << d[i] << endl;
   return 0;
}
```

```
13 d 15
```

```
......1.2340
....-12.3457
```

Методы обмена с потоками

Программа считывает строки из входного потока в символьный массив.

```
#include "iostream.h"
int main(){
    const int N = 20, Len = 100;
    char str[N][Len];
    int i = 0;
    while (cin.getline(str[i], Len, '\n') && i<N){
     // …
     i++;
    }
    return 0;
}
```

Примеры

```
// Проверить, установлен ли флаг flag:
if(stream_obj.rdstate() & ios::flag)
// Сбросить флаг flag:
stream_obj.clear(rdstate() & ~ios::flag)
// Установить флаг flag:
stream_obj.clear(rdstate() | ios::flag)
// Установить флаг flag и сбросить все
  остальные:
stream_obj.clear( ios::flag)
// Сбросить все флаги:
 stream_obj.clear()
```

Примеры

```
void CheckStatus(ifstream &in){
  int i;
  i = in.rdstate();

  if(i & ios::eofbit)
   cout << "EOF is occured" << endl;
  else if(i & ios::failbit)
   cout << "Not fatal i/o error" << endl;
  else if(i & ios::badbit)
   cout << "Fatal i/o error« << endl;
}
```

Файловые потоки

Использование файлов в программе:

- создание потока;
- открытие потока и связывание его с файлом;
- обмен (ввод/вывод);
- закрытие файла.

Каждый класс файловых потоков содержит **конструкторы**, с помощью которых можно создавать объекты этих классов различными способами.

Конструкторы

- Конструкторы без параметров создают объект соответствующего класса, не связывая его с файлом:
  - ifstream();
  - ofstream();
  - fstream();
- Конструкторы с параметрами создают объект соответствующего класса, открывают файл с указанным именем и связывают файл с объектом:
  - ifstream(const char *name, int mode = ios::in);
  - ofstream(const char *name, int mode = ios::out | ios::trunc);
  - fstream(const char *name, int mode = ios::in | ios::out);

Открытие файла

Открыть файл в программе можно с использованием:

- конструкторов;
- метода open, имеющего такие же параметры, как и в соответствующем конструкторе:

```
ifstream inpf ("input.txt");
if (!inpf){
    cout << "Невозможно открыть файл для чтения";
    return 1; }
ostream f;
f.open("output.txt", ios::out);
if (!f){
    cout << "Невозможно открыть файл для записи";
    return 1; }
```

Пример

(программа выводит на экран содержимое файла):

```
#include <fstream.h>
int main(){
    char text[81], buf[81];
    cout << "Введите имя файла:";
    cin >> text;
    ifstream f(text, ios::in | ios::nocreate);
    if (!f){
     cout << "Ошибка открытия файла"; return 1;
    }
    while (!f.eof()){
     f.getline(buf, 81);
     cout << buf << endl;
    }
    return 0;
}
```

Пример

- *Функция — это именованная последовательность описаний и операторов, выполняющая какое-либо законченное действие*. Функция может принимать параметры и возвращать значение.

- Любая программа на C++ состоит из функций, одна из которых должна иметь имя main (с нее начинается выполнение программы). Функция начинает выполняться в момент *вызова*. Любая функция должна быть *объявлена* и *определена*. Объявление функции должно находиться в тексте раньше ее вызова для того, чтобы компилятор мог осуществить проверку правильности вызова.

- *Объявление функции* задает ее имя, тип возвращаемого значения и список передаваемых параметров. *Определение функции* содержит, кроме объявления, *тело* функции, представляющее собой последовательность операторов и описаний в фигурных скобках:

- **[класс] тип имя ([список_параметров]) [throw (исключения)]**
- **{ тело функции }**

- С помощью необязательного модификатора **класс** можно явно задать область видимости функции, используя ключевые слова **extern** и **static**:
- **extern** — глобальная видимость во всех модулях программы (по умолчанию);
- **static** — видимость только в пределах модуля, в котором определена функция.
- Тип возвращаемого функцией значения может быть любым, кроме массива и функции (но может быть указателем на массив или функцию). Если функция не должна возвращать значение, указывается тип void.
- Список параметров определяет величины, которые требуется передать в функцию при ее вызове. Элементы списка параметров разделяются запятыми. (в объявлении имена можно опускать).

- *В определении, в объявлении и при вызове одной и той же функции типы и порядок следования параметров должны совпадать*.
- На имена параметров ограничений по соответствию не накладывается, поскольку функцию можно вызывать с различными аргументами, а в прототипах имена компилятором игнорируются (они служат только для улучшения читаемости программы).

- Тип возвращаемого значения и типы параметров совместно определяют *тип функции.*
- Для вызова функции в простейшем случае нужно указать ее имя, за которым в круглых скобках через запятую перечисляются имена передаваемых аргументов.
- Вызов функции может находиться в любом месте программы, где по синтаксису допустимо выражение того типа, который формирует функция.
- Если тип возвращаемого функцией значения не void, она может входить в состав выражений или, в частном случае, располагаться в правой части оператора присваивания.

Пример функции, возвращающей сумму двух целых величин:

```
#include <stdio.h>
int sum(int a, int b);// объявление функции
int main(){
   int a = 2, b = 3, c, d;
   c = sum(a, b);      //вызов функции
       scanf("%d", d);
   printf("%d", sum(c, d));     // вызов функции
   return 0;
}
int sum(int a, int b){     // определение функции
   return (a + b);
}
```

- Все величины, описанные внутри функции, а также ее параметры, являются локальными. Областью их действия является функция. При вызове функции, как и при входе в любой блок, в стеке выделяется память под локальные автоматические переменные

- Кроме того, в стеке сохраняется содержимое регистров процессора на момент, предшествующий вызову функции, и адрес возврата из функции для того, чтобы при выходе из нее можно было продолжить выполнение вызывающей функции.

- При выходе из функции соответствующий участок стека освобождается, поэтому значения локальных переменных между вызовами одной и той же функции не сохраняются.
- Если этого требуется избежать, при объявлении локальных переменных используется модификатор static:

```
#include <iostream.h>
void f(int a){
   int m = 0;
   cout<<endl<<"\tm\tp "<<endl;
   while (a--){
     static int n = 0;
     int p = 0;
     cout<<n++<<'\t'<<m++<<'t'<<p++;
     cout<<endl;
    }
}
int main(){
f(3); f(2);
return 0;}
```

- Статическая переменная n размещается в сегменте данных и инициализируется один раз при первом выполнении оператора, содержащего ее определение.
- Автоматическая переменная m инициализируется при каждом входе в функцию.
- Автоматическая переменная p инициализируется при каждом входе в блок цикла. Программа выведет на экран:

```
n  m  p
0  0  0
1  1  0
2  2  0
n  m  p
3  0  0
4  1  0
```

- При совместной работе функции должны обмениваться информацией. Это можно осуществить с помощью глобальных переменных, через параметры и через возвращаемое функцией значение.

- Глобальные переменные видны во всех функциях, где не описаны локальные переменные с теми же именами, поэтому использовать их для передачи данных между функциями очень легко. Тем не менее это не рекомендуется, поскольку затрудняет отладку программы и препятствует помещению функций в библиотеки общего пользования.

- Нужно стремиться к тому, чтобы функции были максимально независимы, а их интерфейс полностью определялся прототипом функции.

- Механизм возврата из функции в вызвавшую ее функцию реализуется оператором

- **return [выражение];**

- Функция может содержать несколько операторов return (это определяется потребностями алгоритма).
- Если функция описана как void, выражение не указывается. Оператор return можно опускать для функции типа void, если возврат из нее происходит перед закрывающей фигурной скобкой, и для функции main.
- Выражение, указанное после return, неявно преобразуется к типу возвращаемого функцией значения и передается в точку вызова функции.

Примеры:

- int f1(){return 1;} //правильно
- void f2(){return 1;} //неправильно, //f2 не должна возвращать значение
- double f3{return 1;}//правильно, 1 //преобразуется к типу double

- Нельзя возвращать из функции указатель на локальную переменную, поскольку память, выделенная локальным переменным при входе в функцию, освобождается после возврата из нее.

- Пример:

```
int* f(){
   int a = 5;
   return &a;        // ошибка
}
```

- Механизм параметров является основным способом обмена информацией между вызываемой и вызывающей функциями. Параметры, перечисленные в заголовке описания функции, называются *формальными*, а записанные в операторе вызова функции — *фактическими*.

- При вызове функции в первую очередь вычисляются выражения, стоящие на месте фактических параметров; затем в стеке выделяется память под формальные параметры функции в соответствии с их типом, и каждому из них присваивается значение соответствующего фактического параметра. При этом проверяется соответствие типов и при необходимости выполняются их преобразования. При несоответствии типов выдается диагностическое сообщение.

- Существует два способа передачи параметров в функцию: по значению и по адресу.

- ***При передаче по значению*** в стек заносятся копии значений фактических параметров, и операторы функции работают с этими копиями. Доступа к исходным значениям параметров у функции нет, а, следовательно, нет и возможности их изменить.

- ***При передаче по адресу*** в стек заносятся копии адресов параметров, а функция осуществляет доступ к ячейкам памяти по этим адресам и может изменить исходные значения параметров:

```
#include <stdio.h>
void f(int i, int* j, int& k);
int main(){
   int i = 1, j = 2, k = 3;
   printf("i\tj\tk\n“);
   printf(“%d\t%d\t%d\n”, i, j, k);
   f(i, &j, k);
   printf(“%d\t%d\t%d\n”, i, j, k);
   return 0;
}
void f(int i, int* j, int& k){
   i++; (*j)++; k++;
}
```

- Результат работы программы:

```
i  j  k
1  2  3
1  3  4
```

- Первый параметр (i) передается по значению
- Второй параметр (j) передается по адресу с помощью указателя.
- Третий параметр (k) передается по адресу с помощью ссылки.

- *При передаче по ссылке* в функцию передается адрес указанного при вызове параметра, а внутри функции все обращения к параметру неявно разыменовываются.
- Поэтому использование ссылок вместо указателей улучшает читаемость программы, избавляя от необходимости применять операции получения адреса и разыменования. Использование ссылок вместо передачи по значению более эффективно, поскольку не требует копирования параметров.
- Если требуется запретить изменение параметра внутри функции, используется модификатор const:

- int f(const char*);
- char* t(char* a, const int* b);

- Рекомендуется указывать const перед всеми параметрами, изменение которых в функции не предусмотрено. Это облегчает отладку больших программ, так как по заголовку функции можно сделать вывод о том, какие величины в ней изменяются, а какие нет.
- Таким образом, исходные данные, которые не должны изменяться в функции, предпочтительнее передавать ей с помощью константных ссылок.
- По умолчанию параметры любого типа, кроме массива и функции (например, вещественного, структурного, перечисление, объединение, указатель), передаются в функцию по значению.

Передача массивов в качестве параметров

- При использовании в качестве параметра *массива* в функцию передается указатель на его первый элемент, иными словами, массив всегда передается по адресу.
- При этом информация о количестве элементов массива теряется, и следует передавать его размерность через отдельный параметр (в случае массива символов, то есть строки, ее фактическую длину можно определить по положению нуль-символа):

```
#include <stdio.h>
int sum(const int* mas, const int n);
int const n = 10;
int main(){
   int marks[n] = {3, 4, 5, 4, 4};
   printf("Сумма элементов массива: “);
   sum(marks, n);
   return 0;
}
```

```
int sum(const int* mas, const int n){
// варианты:   int sum(int mas[], int n)
// или      int sum(int mas[n], int n)
// (величина n должна быть константой)
   int s = 0;
   for (int i = 0; i<n; i++) s += mas[i];
   return s;
}
```

Передача имен функций в качестве параметров

- Функцию можно вызвать через указатель на нее. Для этого объявляется указатель соответствующего типа и ему с помощью операции взятия адреса присваивается адрес функции:

```
void f(int a ){ /*  */ }  // определение функции
void (*pf)(int);          // указатель на функцию
pf = &f;
// указателю присваивается адрес функции
// (можно написать pf = f;)
pf(10);// функция f вызывается через указатель pf
    // (можно написать (*pf)(10) )
```

Параметры со значениями по умолчанию

- Чтобы упростить вызов функции, в ее заголовке можно указать значения параметров по умолчанию.
- Эти параметры должны быть последними в списке и могут опускаться при вызове функции. Если при вызове параметр опущен, должны быть опущены и все параметры, стоящие за ним.
- В качестве значений параметров по умолчанию могут использоваться константы, глобальные переменные и выражения:

```
int f(int a, int b = 0);
void f1(int, int = 100, char* = 0);

f(100); f(a, 1);
  // варианты вызова функции f
f1(a); f1(a, 10);
 f1(a, 10, "Vasia");
  // варианты вызова функции f1
f1(a,,"Vasia")          // неверно!
```

Рекурсивные функции

- Рекурсивной называется функция, которая вызывает саму себя. Такая рекурсия называется *прямой*.
- Существует еще *косвенная* рекурсия, когда две или более функций вызывают друг друга. Если функция вызывает себя, в стеке создается копия значений ее параметров, как и при вызове обычной функции, после чего управление передается первому исполняемому оператору функции.
- При повторном вызове этот процесс повторяется.

- Для завершения вычислений каждая рекурсивная функция должна содержать хотя бы одну нерекурсивную ветвь алгоритма, заканчивающуюся оператором возврата.

- При завершении функции соответствующая часть стека освобождается, и управление передается вызывающей функции, выполнение которой продолжается с точки, следующей за рекурсивным вызовом.

- Классическим примером рекурсивной функции вычисление факториала. Для того чтобы получить значение факториала числа n, требуется умножить на n факториал числа (n-1).

- Известно также, что 0!=1 и 1!=1.

```
long fact(long n)
{
  if (n==0 || n==1) return 1;
  return (n * fact(n - 1));
}
```

- Рекурсивные функции чаще всего применяют для компактной реализации рекурсивных алгоритмов, а также для работы со структурами данных, описанными рекурсивно, например, с двоичными деревьями.
- Любую рекурсивную функцию можно реализовать без применения рекурсии, для этого программист должен обеспечить хранение всех необходимых данных самостоятельно.
- Достоинством рекурсии является компактная запись, а недостатками — расход времени и памяти на повторные вызовы функции и передачу ей копий параметров, и, главное, опасность переполнения стека.

Заключение

- Спасибо за внимание!
- Вопросы???