

Программирование.

Практика: Калитаев Александр Николаевич /

Лекции: /Торчинский Вадим Ефимович

Языки программирования: С (структурно-процедурный) / С++ (Объектно-ориентированный)

Литература: Павловская Т. А. «Программирование на языке высокого уровня. С++»

Язык С был создан в середине 70-х профессором Керниганом Риччи. В середине 80-х профессор Страуструп расширил язык и создал С++.

Алфавит языка С.

Элементы алфавита делятся на 4 группы:

- Символы, используемые для образования ключевых слов и идентификаторов – символы латинского алфавита, арабских цифр и нижнего подчёркивание.
- Знаки операций и спецсимволы: + - ? () и тд
- Управляющие и разделительные символы: [Space], [Tab], “\n”, [Enter]
- Управляющие последовательности: спецсимвольные комбинации, используемые в функции ввода-вывода.

Язык С – регистрозависимый; прописные и строчные символы – различные.

Последовательность разделительных символов компилятором рассматривается как 1 символ.

Управляющая последовательность начинается с символа \ (обратный слэш), за которым следует комбинация латинских букв и символов.

Примеры управляющих последовательностей:

\t – табуляция; \r – [enter]; \o – символ с кодом 0; \" – кавычка; \n – перевод строки; \' – апостроф; \\ - обратный слэш.

С помощью управляющей последовательности можно закодировать любой символ, указав его восьмеричный или шестнадцатеричный код.

Например, \r = \015 = \xD

Константы

В языке C есть 4 типа констант: целые (integer), вещественные (float), символьные (char) и строковые (string).

1. Целая константа записывается в 10-ричной, 8-ричной и 16-ричной форме. $16_{10} = 020 = 0x10$
2. Вещественные значения в памяти хранятся в формате с плавающей точкой. $0.xxxxx * 10^{yyy}$, где xxxxx – мантисса числа, а yyy – порядок (сдвиг плавающей точки). $0.0012 = 0.12E-2 = 0.12 * 10^{-2}$.
3. Символьные константы – символ, заключённый в апострофы. Вместо символа может быть управляющая последовательность.
4. Строковые константы – последовательность символов, заключённая в кавычках. В памяти строковая константа занимает n+1 байт, где n – количество символов в строке. Причина – ограничитель строки, символ с кодом 0 – «терминатор».

Комментарии

Комментарии – набор символов, игнорируемый компилятором. В C есть 2 вида комментариев: однострочные и произвольные. Однострочные комментарии непарные и начинаются с «//». Произвольные комментарии парные, начинаются с «/*» и заканчиваются «*/».

Переменные

Переменная – именованная область памяти с возможностью изменения хранящейся там информации. Объявление переменных:

[Класс памяти] Идентификатор (инициатор).

[] можно не учитывать

Класс памяти определяет время жизни и область видимости переменной. Инициатор задаёт начальное значение переменной.

Классы памяти: 1) auto -> 2) register -> 3) static -> 4) extern

Перед типом можно указать ключевое слово const. В результате объявляется не модифицируемая переменная.

const double pi = 3.14

Типы данных

1. Целые типы делятся на 2 класса: signed и unsigned. Диапазоны: $-2^{l-1} \dots 2^{l-1} - 1$ и $0 \dots 2^l - 1$, где l – количество памяти, отводимой на переменную в битах.

Size, Bytes	Signed	Unsigned
1	Char	Unsigned char
2	Short // __int16	Unsigned short
4	Long // __int32	Unsigned long
8	Long long // __int64	Unsigned long long

Кроме того, целые типы – int и unsigned int. Размер типа int равен машинному слову – разрядности системы, для которой компилируется программа.

Положительные значения хранятся в прямом коде, отрицательные – в дополнительном коде.

Дополнительный код получается через обратный код и добавление единицы.

2. Вещественные числа – числа с плавающей точкой.

a. *float*: 4 байта, мантисса – 7~8 цифр, порядок - ~38

b. *double*: 8 байт, мантисса – 11~12 цифр, порядок - ~308

c. *long double*: 10 байт, мантисса – 15~16 цифр, порядок - >1000

Вещественные числа хранятся следующим образом: старший бит – знак; мантисса и порядок.

3. Указатели – адреса памяти, где расположены данные для объявления указателя, используется модификатор типа *.

int *P; //пример указателя

P содержит адрес памяти, по которому размещены данные типа int. Вне зависимости от типа данных, на которые ссылается указатель, размер переменной указателя одинаков и обычно составляет 4 байта. Размер указателя 4 байта позволяет адресовать 4 Гб ОЗУ. Для работы с указателями часто используют следующие операции:

- Взятие адреса – знак & (амперсант). Аргументом является ID переменной. Результат: адрес памяти, по которому расположена данная переменная.
- Разадресация – знак *. Позволяет получить данные по указанному адресу (аргумент – адрес). Операция леводопустимая – данные доступны для чтения и записи.

Перечисляемые типы

Используется для задания символических имен, значения этого типа. Это помогает написанию и отладке программы. Объявление и перечисление имеет следующий формат: enum [тип] {список_перечисления} [переменные]

Имя пользовательского типа

Список перечисления имеет несколько конструкций вида:

Идентификатор [= константное выражение]

Идентификатор – значение перечисляемого типа

Константное выражение определяет целое число, которое соответствует указанному типу идентификатора. По умолчанию первый идентификатор имеет значение 0, а следующие – на единицу больше предыдущего.

```
enum Color {Red, Green, Blue} c;
```

```
c = Red;
```

```
enum Color c2; // ЯП Си
```

```
Color c2; // ЯП Си++
```

```
enum T {T1, T2 = 3, T3, T4 = 1, T5...};
```

Тип void.

Тип void – неопределённый тип данных. Используется для указания того факта, что функция не возвращает значений. Указатель на void является не типизированным, т. е. указателем на произвольный тип данных.

Операции.

По количеству операндов операции в Си делятся на 3 группы. Первая – унарные функции, второй – бинарные функции, третья – тернарная операция с тремя операндами.

Унарные операции.

1. Взятие адреса, разадресация (&, *)
2. Унарный минус (- [число]) // a = -a;
3. Унарный плюс (никогда не используется)
4. ! – логическое отрицание. Аргумент – целочисленное выражение. Если его значение равно нулю, то результат – единица. Иначе – ноль. !1 -> 0; !0 -> 1
5. ~ - Побитовое логическое отрицание (дополнение).
~00010111 -> 11101000
unsigned char c;
c = ...;
c = c + ~c; // 255
6. sizeof. Аргументом операции может быть произвольное выражение либо имя типа. Результат – размер типа в байтах.

```
double d;
```

```
sizeof(d); >> 8
```

```
sizeof(short); >> 2
```

7. Инкремент (++) и декремент (--). Аргументом инкремента и декремента выступает идентификатор переменной. Инкремент увеличивает значение переменной на 1, а декремент уменьшает. Различают префиксное и постфиксное использование этих операций. В первом случае возвращается изменённое значение переменной, а во втором – старое.

```
int a = 1;
```

```
/int a = 1, b\
```

```
a++; // 2
```

```
b = ++a;
```

```
b = a++
```

```
a -> 2, b -> 2
```

```
a -> 2, b -> 1
```

Бинарные операции.

Бинарные операции делятся на несколько групп:

1. Мультипликативные (операции умножения): *, /, %. У умножения и деления операнды могут быть целые и вещественные; если оба операнда – целые, то результат – также целый. Если хотя бы один – вещественный, то и результат – вещественный. Для остатка от деления аргументы должны быть только целочисленные.

```
14 / 5 -> 2
```

```
14 % 5 -> 4
```

```
14 / 5. -> 2.8
```

2. Аддитивные операции (операции сложения): +, -. Правила аналогичны умножению и делению.

3. Операции сдвига: << (сдвиг влево) и >> (сдвиг вправо). Оба аргумента – целочисленные; второй аргумент показывает, на сколько позиций сдвигаются биты первого аргумента.

5 << 1; // 10 5 >> 1; // 2

4. Операции отношения (арабское сравнение): >, >=, <, <=, ==, !=. Операции сравнения возвращают либо 0, либо 1.

5. Поразрядные или побитовые операции: & (битовое И), | (битовое ИЛИ), ^ (побитовое исключающее ИЛИ).

12 & 10 -> 8 (1100 & 1010 = 1000)

12 | 10 -> 14 (1100 | 1010 = 1110)

12 ^ 10 -> 6 (1100 ^ 1010 = 0110)

6. Логические операции: &&, || (логич. И, ИЛИ). Результат операции – либо 0, либо 1.

7. Операции присваивания: [выр1=выр2]; выражения должны быть совместимыми по типу, выр1 должно быть леводопустимым. Результат операции – значение присваиваемого выражения. Кроме простого присваивания есть составное присваивание: *=, /=, %=, -=, +=, <<=, >>=, &=, |=, ^=.

8. Последовательное вычисление. Обозначается запятой, используется там, где по синтаксису возможно единственное выражение, а требуется поместить несколько выражений.

Тернарные операции.

1. Тернарная операция: выр1?выр2:выр3

2. Выражение 1 – целочисленное, выражения 2 и 3 – произвольные.

3. Если значение [выр1] не равно 0, то вычисляется [выр2], иначе – [выр3]

min=a<b?a:b;

Преобразование типов. 21.9.22

При вычислении выражений происходит автоматическое преобразование типом операторов. Такое преобразование называют неявным. При неявном преобразовании последовательно применяются следующие правила:

- Float преобразуется в double;
- Если один оператор – Long double, то второй преобразуется к этому типу;
- Если один тип – double, то второй тоже преобразуется в double;
- Char и short преобразуются к int;
- Unsigned char и unsigned short преобразуются к unsigned int;
- Если один оператор – unsigned long, то и второй преобразуется к unsigned long;
- Если один оператор – long, то и второй преобразуется к long;
- Если один операнд – unsigned int, то и второй преобразуется к этому же типу.

```
float a;  
int b;  
a / b:    float/int -> double/int -> double/double
```

Неявное преобразование происходит в случае присваивания, если мы присваиваем вещественное в целое. При этом происходит отбрасывание дробной части, и компилятор выдаёт предупреждение (warning).

```
int a = 8.6; // a -> 8
```

Программист может самостоятельно осуществлять приведение типов, то есть использовать явное преобразование. Для этого перед преобразуемым выражением указывается имя типа в круглых скобках.

```
(float)(14/5); -> 2.  
(float) 14/5; -> 2.8
```

Структура программы состоит из группы функций, которые могут вызывать друг друга. Кроме того, вне функций могут располагаться объявления переменных и типов. Программа может состоять из нескольких файлов, каждый из которых транслируется отдельно, а потом происходит сборка в один исполняемый файл. Выполнение программ всегда начинается с функции с названием main().

Операторы языка.

Все операторы, кроме составных, заканчиваются точкой с запятой (;).

Оператор выражения.

Любое выражение, заканчивающееся точкой с запятой (;) является оператором. Обычно выполнение этого оператора имеет побочный эффект, заключающийся в изменении значений переменных, вводе-выводе и тому подобных.

```
a = sin(t) + y; // изменение значения переменной
z ++; // изменение значения переменной
printf("%d", i); // нет результата, синтаксически верно
z+1; //
```

Составной оператор.

Представляет собой несколько операторов и объявлений, заключенных в фигурные скобки.

```
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

Составной оператор используется там, где синтаксически требуется оператор, например, тело цикла.

Ввод-вывод.

Функции ввода-вывода не являются составной частью языка С. (Язык С задумывался как универсальный язык, для любых компьютеров и операционных систем.) Функции ввода-вывода располагаются в стандартных библиотеках, которые могут быть специфичны для конкретной архитектуры вычислительной системы. Рассмотрим стандартную библиотеку ввода-вывода stdio.h:

Подключается так: `#include <stdio.h>`

Для вывода в библиотеке определена функция printf, которая имеет следующий формат:

```
printf(control, arg1, arg2, ...);
```

Функция осуществляет вывод значения аргументов под управлением строки control.

Управляющая строка содержит два типа объектов:

- Обычные символы, которые выводятся без изменений;
- Спецификации преобразования, каждый из которых выполняет вывод очередного аргумента. Каждая спецификация начинается с символа % и заканчивается символом преобразования. Между ними могут находиться: **1.** Знак минус, указывающий на выравнивание по левому краю; **2.** Число, задающее ширину поля вывода; **3.** Точка и число, задающее количество цифр дробной части.

Будем использовать два символа спецификации: **d** – для целых типов данных, **f** – для *float*, **lf** – для *double*, **s** – для строк, **c** – для вывода типа **char** в виде символа.

```
int a = 38; double p = 3.14159;
```

```
printf("Results is: \n");
```

Начиная с этой лекции запрещается использовать **cin/cout**.

```
printf("a = %d\n", a); // связка %d вычисляет значение выражения a и выводит его
```

```
printf("a = %d, p = %lf\n", a, p); << a = 38, p = 3.14159 [enter]
```

```
int a = 38; double p = 3.14159; int b = 48;
```

```
printf("%d %c\n", b, b); // 48 0 (48 – код символа 0)
```

```
printf("a=%-4d, p=%7.3lf\n", a, p); // a=38, pi= 3.142;
```

По умолчанию, заполняющим символом является пробел. Его можно заменить на ноль, указав 0 перед шириной поля.

```
int day = 7, month=4, year=2022;
```

```
printf("%02d.%02d.%d\n", day, month, year);
```

```
// << 07.04.2022
```

Для ввода используется функция ***scanf(control, arg1, arg2, ...)***. Аргументами функции *scanf* являются адреса переменных, значения которых вводятся с клавиатуры.

scanf("%d%d", &a, &b); // операции взятия адресов переменных

Можно использовать любой разделитель или их комбинацию, запятая не является разделителем.

Оператор if.

If (выражение)

оператор1;

[else оператор2;]

int t=2; b=7; r = 3;

if (t>b)

if (b<r) r=b;

else r=t // else относится к ближайшему if

printf("%d", r); // 3

int t=2; b=7; r=3;

if(t>b)

{if (b<r) r=b;}

else r=t;

printf("%d", r); // 2

int t=2; b=7; r=3;

if(t=b) r = 11;

else r =12;

printf("%d", r); // 11, t ->7

int t=2; b=7; r=3;

if(t==b) r = 11;

else r =12;

printf("%d", r); // 12

Дано 3 числа, представляющие собой день/месяц/год. Определить, возможна ли указанная дата.

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
int main() {
    int day, month, year, flag;
    printf("Input DD MM YYYY ");
    scanf("%d %d %d", &day, &month, &year);
    if(year < 1 || month < 1 || month > 12 || day < 1 || day > 31) flag = 0;
    else {
        int dayInMonth;
        if(month == 4 || month == 6 || month == 9 || month == 11)
            dayInMonth = 30;
        else if (month == 2) /* високосным считается год номер которого делится на 400, либо на делится на 4 и не
делится на 100 */
            if (year % 400 == 0 || year % 4 == 0 && year % 100 != 0)
                dayInMonth = 29;
            else dayInMonth = 28;
        else dayInMonth = 31;
        flag = day <= dayInMonth;
        if (flag) printf("Possible date\n");
        else printf("Impossible date\n");
        return 0;
    }
}
```

Оператор switch (как и оператор if) является оператором условного перехода. Если if делит алгоритм на две ветки, то switch – на произвольное число ветвей.

```
Switch (выражение)
{ [объявления]
  [case константное выр1] : [список операторов]
  [case константное выр2] : [список операторов]
  .....
  [default: [список операторов]]
}
```

Выражение в круглых скобках должно быть целочисленным. Значение этого выражения является ключом выбора. Значение ключа последовательно сравнивается со значениями константных выражений. Если значения совпадают, то управление передаётся на соответствующий список операторов. Если совпадения не найдено, выполняются операторы, следующие за меткой default. Если нет метки default, происходит выход из цикла.

Следует отметить, что case и default существуют только при начальной проверке. Все операторы от начального до конца тела switch выполняются независимо от ключевых слов.

```
int i = 2;
switch (i){
    case 1: i += 2;
    case 2: i *= 3;
    case 0: i /= 2;
    case 4: i -= 5;
}
printf("%d\n", i); \\ -2 т.к. выполнились все операторы, начиная с ключевого case 2
```

Для того, чтобы прервать тело switch, используют оператор break.

```
int i = 2;
switch (i){
```

```

        case 1: i += 2; break;
        case 2: i *= 3; break;
        case 0: i /= 2; break;
        case 4: i -= 5;
    }
    printf("%d\n", i); \\ 6

```

Дано целое неотрицательное K. Вывести значение K и слово «рубля», «рубль» или «рублей» в соответствии с правилами русского языка.

```

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
int main() {
    int K;
    printf("Input summa ");
    scanf("%d", &K);
    if (K % 100 > 10 && K % 100 < 15) printf("%d rubley\n", K);
    else {
        int n = K % 10;
        switch (n) {
            case 1: printf("%d rubl\n", K); break;
            case 2:
            case 3:
            case 4: printf("%d rublya\n", K); break;
            default: printf("%d rubley\n", K); break;
        }
    }
    return 0;
}

```

Цикл с предпроверкой (предусловием) реализуется с помощью оператора while, который имеет следующий формат:

```

while (условие)
    {тело цикла}

```

```

int sum = 0, i = 1;
while (i <= 100) {
    sum += i;
    i++;
}

```

Цикл с постусловием реализуется с помощью оператора do while:

```

int sum = 0, i = 1;
do {
    sum += i;
    i++;
}
while (i <= 100);

```

Оператор for. 28.9.22

В C оператор for реализует наиболее общий оператор цикла.

```
for ([выр1]; [выр2]; [выр3])  
    [тело цикла]
```

Выражение 1 обычно используется для установки начального значения переменных, управляющих циклом.

Выражение 2 определяет условие выполнения цикла.

Выражение 3 определяет изменение переменных после каждого выполнения тела цикла.

Схема вычисления:

1. вычисляется выражение 1
2. вычисляется выражение 2
3. если значение выражения 2 не равно 0
 - a. выполняется тело цикла
 - b. вычисляется выражение 3
 - c. переход к п.2
4. иначе – цикл заканчивает работу и управление передаётся следующему оператору.

Схема выполнения for через while:

```
выр1;  
while (выр2) {  
    тело цикла;  
    выр3;  
}
```

Пример (сумма чисел от 1 до 100):

```
int s = 0;  
for (int i = 1; i <= 100, i++)  
    s += i;
```

В чашке Петри находится колония из B бактерий.

В какой-то момент времени туда попадает V вирусов ($V < B$).

Каждый час вирус съедает 1 бактерию, после чего количество и тех, и других удваивается.

Определить, за какое время бактерий не останется.

```
#define _CRT_SECURE_NO_WARNINGS
```

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

```
int main() {  
    long b, v;  
    unsigned short t = 0;  
    do {  
        printf("Input bacteria and viruses ");  
        scanf("%ld%ld", &b, &v);  
    } while (v >= b);  
    for (; b > 0; t++) { // абсурдный вариант через последовательные вычисления: for(; b > 0; b -= v, b *= 2, v *= 2,  
t++);  
        b -= v; // на языке C это будет так:  
        b *= 2; // b <= 1;  
        v *= 2; // v <= 1;  
        t++;  
        printf("There are %d bacterias left and %d viruses left\n", b, v);  
    }  
    printf("Time consumed: %d", t);  
    return 0;  
}
```

Цикл с выходом из середины.

Данный тип циклов можно моделировать на основе любого оператора цикла. Например, на основе оператора while.

```

while (1) {
    тело_цикла1;
    if (y) break; // y – условие выхода из цикла
    тело_цикла2;
}

```

Предположим, необходимо найти произведение неизвестного заранее количества сомножителей. Признаком окончания ввода будем считать число 0.

Оператор `continue`, как и оператор `break`, прерывает выполнение тела цикла. Но, в отличие от `break`, цикл не заканчивает работу, а управление передаётся либо на вычисление условия продолжения цикла (`while`, `do while`), либо на выражение 3 (`for`).

Массивы.

Массив – группа однотипных элементов. В отличие от рассмотренных ранее простых типов, массив относится к структурированным. Из объявления массива компилятор должен получить информацию о типе и количестве элементов.

‘Тип’ ‘идентификатор’ ‘[константное выражение]’ // в данном случае [] обязательны.

Тип элементов массива, имя массива, количество значений в массиве. При объявлении массив может быть инициализирован.

```

int a[4];
float b[5] = {3.1, 7.2, 1.8, 9.4, 4.0}

```

если инициаторов меньше, чем элементов, то остальные автоматически обнуляются.

В идентификаторе массива хранится его адрес. В памяти массив типа `double` из 5 элементов занимает 24 байта, а не 20 байт, т.к. дополнительно хранится указатель на адрес.

Для доступа к элементам массива используется индексное выражение:

```

выр1[выр2]

```

Тип индексного выражения соответствует типу элементов массива. Обычно, выражение 1 – указатель, а выражение 2 является целочисленным. Указатель определяет адрес массива, а целое число – номер элемента. **! В С нумерация начинается с нуля !**

То есть доступ к массиву есть как по чтению, так и по записи.

На самом деле, нет жесткого требования, чтобы `выр1` являлось казателем, а `выр2` – целочисленным.


По синтаксису

При вычислении адресного выражения не производится проверка, является ли данная память частью массива.

В си не определены двумерные и многомерные массивы, но нет запрета на то, чтобы элементом массива был другой массив.

```
int a[2][3] = {
    {7, 4, 8};
    {1, 0, 9};
}
// представление памяти:
int a[n]; // первый алгоритм – ввод с клавиатуры
for (i = 0; i < n; i++) {
    printf("a[%d] =", i);
    scanf("%d", &a[i])

str[] = "ABC";
```



Заполнение массива случайными числами. Псевдослучайные числа получаются на основе предыдущих. Для генерации используют функцию `rand()` из библиотеки `<stdlib.h>`. `rand` возвращает случайное число от 0 до `rand_max`. Если не предпринять никаких действий, то база генерации будет одной и той же, и `rand()` будет возвращать одну и ту же последовательность. Для изменения базы генерации используют функцию `srand()`. Эта функция должна использоваться только один раз.

Функция `time` с аргументом 0 возвращает число секунд, прошедших с 1 января 1970 года.

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
using namespace std;

int main() {
    int n, a[10], max, min;
    // какой-нибудь код
    srand((unsigned int)time(0));
    for (int i = 0; i < n; i++) a[i] = rand() % (max - min + 1) + min;
}

(double)rand() / RAND_MAX * (max - min) + min
```

Обнуление массива.

Обнуление через инициализацию `// int a[n] = {0};`

Если требуется повторное обнуление, то такой цикл писать неэффективно:

```
for (int i = 0; i < n; i++) a[i] = 0;
```

`memset()`. В библиотеке `mem.h` или `memory.h` данная функция заполняет область памяти указанным байтом. Функция имеет три аргумента: адрес области, байт заполнения, размер области. `memset(a, 0, sizeof(a));`

Вывод массива в столбик:

```
for (int i = 0; i < n; i++) printf("%d\n", a[i]);
```

Вывод массива в строку:

```
for (int i = 0; i < n; i++) printf("%d ", a[i]);
printf("\n");
```

Поиск минимального (максимального) элемента:

1. Поиск по значению:

```

int min = a[0];
for (int i = 0; i < n; i++)
    if (a[i] < min) min = a[i];
printf("%d\n", min);

```

2. Поиск по индексу:

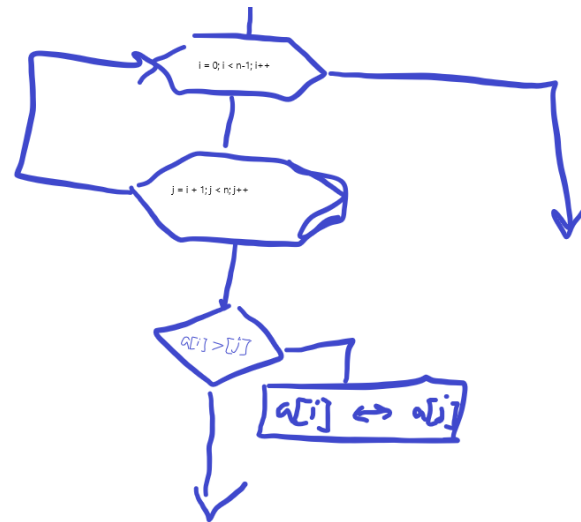
```

int minimind = 0;
for (int i = 0; i < n; i++)
    if (a[i] < a[minimind])
        minimind = i;
printf("%d\n", a[minimind]);

```

Алгоритм, эффективный с точки зрения простоты написания.

Сортировка пузырьком: элементы просматриваются парами от первого до предпоследнего и, при необходимости, эти элементы меняются местами. Блок-схема.



```

int main() {
    int c, n;
    int a[10];
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (a[i] > a[j]) {
                c = a[i];
                a[i] = a[j];
                a[j] = c;
            }
            else {
            }
        }
    }
}

int a[m][n];
for (int i = 0; i < m; i++){
    for (int j = 0; j < n; j++) {
        printf("a[%d][%d]= ", i, j);
        scanf("%d", &a[i][j]);
    }
}

```

Вывод двухмерного массива в виде прямоугольной матрицы

```

for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        printf("%7d", a[i][j]);
    }
    printf("\n")
}

```

Двумерные массивы.

Олимпиада по программированию проводится в n турах. Необходимо определить победителей каждого тура и победителей в общем зачёте.

Участник \ Тур	I	II	Итог
Иванов	25	45	70
Петрова	35	30	65
Сидоров	35	35	70
Максимум	35	45	70

В C++ нет типа данных для двух и более мерных массивов, но нет запрета на помещение одномерного массива в массив. Так и получается двумерный массив.

Массивы: 1. Фамилии; 2. Результаты;

```
#include <stdio.h>
using namespace std;
int main(int argc, char* argv[]) {
    const int n = 3; //число участников
    const int m = 2; //число туров
    char name[n][20]; //фамилия участника из 19 символов
    int res[n][m + 1]; //результаты
    int max[m + 1]; //лучшие результаты
    int i, j;
    //ввод данных
    for (i = 0; i < n; i++) {
        printf("Input a surname of %d-th participant: ", i + 1);
        scanf("%s", &name[i]);
        for (j = 0; j < m; j++) {
            printf("Input uts result in %d-st round: ", m + 1);
            scanf("%d", &res[i][j]);
        }
    }
    //подсчёт суммарного результата
    for (i = 0; i < n; i++) {
        res[i][m] = 0;
        for (j = 0; j < m; j++) res[i][m] += res[i][j];
    }
    //поиск максимальных результатов
    for (j = 0; j < m + 1; j++) {
        max[j] = res[0][j];
        for (i = 1; i < n; i++) {
            if (res[i][j] > max[j]) max[j] = res[i][j];
        }
    }
    //вывод победителей в турах
    int t;
    for (j = 0; j < m; j++) {
        printf("Winners in %d-st round with result %d\n", j + 1, max[j]);
        t = 0;
        for (i = 0; i < n; i++) {
            if (res[i][j] == max[j]) printf("%d. %s\n", ++t, name[i]);
        }
    }
    //вывод победителя в общем зачёте
    t = 0;
    printf("Winner in general offset with result %d\n", max[m]);
    for (i = 0; i < n; i++) {
        if (res[i][m] == max[m]) printf("%d. %s\n", ++t, name[i]);
    }
    //getchar ();
    //getchar ();
    return 0;
}
```


Структуры.

Структура – составной объект, в который входят компоненты произвольной формы.

Структура определяется следующим образом:

```
struct [тип]
    {список_определений} // одна или несколько записей вида тип идентификаторы;
    [переменные];

    struct Person {
        char name[24];
        enum { male, female }, sex;
        int age;
    } p;
```

... называются членами структуры или полями. Переменная структурного типа занимает в памяти столько байт, сколько все её компоненты в сумме, причём, следует учитывать выравнивание по границе слова. Доступ к элементам структуры осуществляется двумя способами.

1 – известно имя переменной. В этом случае используется оператор точка.

```
strcpy(p.name, "Bill");
```

2 – известен адрес структуры. Используется оператор «->».

```
Person *p2 = 8p;
printf("%s\n", p2->name);
```

Операция является леводопустимой.

Для структур, в отличие от массивов, разрешено присваивание.

```
Person r;

strcpy(r.name, p.name);
r.sex = p.sex;
r.age = p.age;

r = p;
```

Для распределения стипендии между студентами: получившие 4 и 5 получают базовую стипендию, отличники – повышенную на 25%, остальные – стипендию в размере 0 рублей. Считать что максимально в сессии 5 экзаменов.

```
#include <stdio.h>
int main() {
    const int n = 5; //any
    struct stud {
        char name[32];
        char group[16];
        int ball[5];
        int countEx;
        float grant;
    };
    stud s[n];
    int i, j;
    for (i = 0; i < n; i++) {
        printf("Enter a name of %d-st student: ", i + 1);
        scanf("%s", s[i].name);
        printf("Enter its group: ");
        scanf("%s", s[i].group);
        printf("Enter quantity of examinations: ");
        scanf("%d", &s[i].countEx);
        for (j = 0; j < s[i].countEx; j++) {
            printf("Enter its ball for %d-st exam: ", j + 1);
            scanf("%d", &s[i].ball[j]);
        }
    }
}
```

Объединения.

Объединение подобно структурам, однако в каждый момент времени может быть использован только один элемент объединения. Инициализация объединения идентична структуре, только вместо слова `struct` используется слово `union`.

```
Union T {  
    Double d;  
    Float f;  
    Char c[12];  
}
```

Элементы типа `union` занимает в памяти объём, равный объёму максимального поля.

Объединение используется в целях экономии памяти, преобразовании типов и доступа к отдельным байтам.

```
Union u{  
    Float x;  
    Int y;  
    scanf ("%f", &u.x);  
}
```

Структуры и объединения. Битовые поля.

09.11.22

Может быть битовое поле, обеспечивающее доступ к отдельным битам в байте

```
struct {  
    unsigned 'идент1': 'длина поля';  
    unsigned 'идент2': 'длина поля';  
} 'идент';
```

Ограничения:

- ⇒ нельзя создавать массивы битовых полей;
- ⇒ нельзя к битовому полю применять операцию взятия адреса.

```
struct {  
    unsigned a1 : 2;  
    unsigned a2 : 5;  
    unsigned a3 : 1;  
} b;
```

Пример: необходимо определить, как в памяти хранятся целые числа: от старшего байта к младшему, или от младшего – к старшему.

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

```
int main() {  
    struct {  
        unsigned t1 : 2;  
        unsigned t2 : 2;  
        unsigned t3 : 2;  
        unsigned t4 : 2;  
    } bitstr;
```

```
int main() {  
    unsigned char a = 27; // 00011011  
    *((unsigned char*)&bitstr) = a;  
    printf("f1 = %d f2 = %d f3 = %d f4 = %d\n", bitstr.t1, bitstr.t2, bitstr.t3, bitstr.t4);  
    return 0;  
}
```

f1 = 3 f2 = 2 f3 = 1 f4 = 0 => в памяти целое число хранится от младшего бита к старшему

Определение переменных и типов.

Тип объявляемой переменной зависит от того, какое ключевое слово было использовано в качестве идентификатора типа и использовали ли при описании модификаторы типа. В C определены следующие модификаторы:

- ⇒ * – модификатор указателя
- ⇒ [] – модификатор массива
- ⇒ () – модификатор функции

В C одновременно можно использовать более одного модификатора, что даёт возможность создавать сложные типы данных. Необходимо запомнить, что некоторые комбинации модификаторов недопустимы:

- ⇒ Элементами массива не могут быть функции.
- ⇒ Функции не могут возвращать массивы и функции.

При интерпретации сложных описаний используется правило «изнутри – наружу»:

1. Начать с идентификатора и посмотреть вправо, есть ли квадратные/круглые скобки;
2. Если они есть, то проинтерпретировать эту часть описания и посмотреть налево в поисках «*»;
3. Если на любой стадии встретится закрывающая круглая скобка, то в начале нужно применить все правила внутри круглых скобок, а затем продолжить интерпретацию;
4. Интерпретировать спецификатор типа.

`int *(*comp[10]) ();` - массив из 10 указателей на функцию без аргументов, возвращающую указатель на тип `int`

элемент `comp` – идентификатор (1). `[10]` – спецификатор (2), `*` - указатель (3), `()` – функция без аргументов (4), `*` - указатель (5), `int` – тип (6).

`Char *(*(*var()))[10];` - переменная `var`, являющаяся указателем на функцию без аргументов возвращающую указатель на массив из 10 указателей на тип `char`.

В C имеется возможность объявлять пользовательские типы данных. Это возможно не только при объявлении перечисления, структуры или объединения. Более общий вариант – использовать ключевое слово `typedef`. При использовании `typedef` идентификатор, стоящий на месте описываемого объекта, является новым пользовательским типом, а не переменной.

```
typedef int* (*T) ();  
T comp[10];
```

`typedef` – универсальный способ определения типа.

Функции.

В отличие от других языков программирования, в **C** нет деления подпрограмм на процедуры и функции, здесь вся программа состоит только из функций.

Функция – совокупность объявлений и операторов, предназначенная для решения определённых подзадач.

Каждая функция имеет имя, которое используется для её объявления, определения и вызова. При вызове функции ей, с помощью аргументов (формальных параметров), передаются некоторые значения (фактические параметры). Функция может возвращать некоторое единственное значение. Это значение и есть результат выполнения функции, которые подставляются в точку вызова. Допускается использование функций, не возвращающих значения. С использованием функций в **C** связано три понятия:

- ⇒ Определение функции;
- ⇒ Объявление функции;
- ⇒ Вызов функции.

Определение функции

Определение функции задаёт тип возвращаемого значения, имя функции, описание формальных параметров, а также объявление переменных и операторы, образующие тело функции. Пример:

```
int randvalue(int min, int max) {  
    int r = rand() % (max - min + 1) + min;  
    return r;  
}
```

Тип возвращаемого значения - **int**, имя функции - **randvalue**, описание формальных параметров - (**int min**, **int max**).

Оператор **return** прекращает выполнение тела функции и в качестве возвращаемого значения передаёт значение своего аргумента.

Если необходимо создать функцию, не возвращающую значений, то в качестве типа функции указывают **void**, а оператор **return** используют без аргументов.

Объявление функции

В **C** нет требования, чтобы определение функции предшествовало её вызову. Для того, чтобы компилятор мог проверить соответствие фактических и формальных параметров в случае, если вызов функции предшествует её определению, перед вызовом следует поместить объявление (прототип) функции. Объявление функции аналогично её определению, за исключением тела функции (оно отсутствует). Кроме того, можно опустить имена формальных параметров.

Сколько необходимо купить карточек лотереи, где нужно угадать m чисел из n возможных, чтобы получить максимальный выигрыш из возможных.

```
#include <stdio.h>  
int main() {  
    setlocale(LC_ALL, "Russian");  
    int m, n, count;  
    scanf("%d%d", &m, &n);  
    double f(int); // тк факториал -  
    count = (int)(f(n) / f(m) / f(n - m));  
    printf("%d\n", count);  
    return 0;  
}  
  
double f(int x) {  
    double p = 1.;  
    for (int i = 2; i <= x; i++)  
        p *= i;  
    return p;  
}
```

Вызов функции

(самая важная тема в семестре, на экзамене многие задают вопросы)

(лир. отступление) Рассмотрим, каким образом распределяется память, выделенная операционной системой для работающей программы.

Память делится на 4 сегмента:

- ⇒ Code segment (сегмент кода) – содержит программные инструкции в машинном коде.
- ⇒ Data segment (сегмент данных) – содержит программные объекты с глобальным временем жизни.
- ⇒ Stack segment (сегмент стека) – содержит программные объекты с локальным временем жизни.
- ⇒ Heap segment (сегмент кучи) – содержит программные объекты, временем жизни которых управляет программист.

Функции используют память из сегмента стека. Данная память организовывается дисциплиной обслуживания LiFo (Last in - First out = последний пришёл – первый вышел). До создания слова стек, для данных последовательностей использовался термин «магазины» (по аналогии с оружием)

Некоторая область стека отводится функции и остаётся связанной с ней до окончания работы, после чего может быть занята другой функцией. В стеке выделяется место для формальных параметров и локальных переменных функции. При вызове функции, память, отведённая под формальные параметры, инициализируется значениями фактических параметров. Стандартным способом передачи аргументов является копирование их значений, т.е. передача по значению. Функция получает копии этих значений, поэтому изменение формальных параметров в вызываемой функции никак не скажется на значениях фактических параметров в вызывающей функции.

```
#include <stdio.h>
void swap(int a, int b) {
    int tmp = a;
    b = a;
    a = tmp;
    printf("2.a = %d b = %d\n", a, b);
}

int main() {
    int a = 1, b = 2;
    printf("1.a = %d b = %d\n", a, b);
    swap(a, b);
    printf("3.a = %d b = %d\n", a, b);
    return 0;
}
```

Для того, чтобы вызываемая функция могла изменять переменные, вызывающие функции, необходимо передавать туда не значения этих переменных, а их адреса.

```
void swap(int *a, int *b) {
    int tmp = *a;
    *b = *a;
    *a = tmp;
    printf("".a = %d b = %d\n", *a, *b);
}
```

Ссылки

В C++ имеется модифицированная форма указателей, называемая ссылкой (reference). Ссылка – это указатель, который автоматически разадресовывается при любом обращении к нему, поэтому ссылке нельзя присвоить значение, её можно только инициализировать. Инициализирующим выражением должен быть идентификатор переменной. Используется модификатор типа «амперсант» – &

```

int main() {
    int a = 1;
    int &r = a;
    printf("%d %d\n", a, r); // 1 1
    a = 2;
    printf("%d %d\n", a, r); // 2 2
    r = 3;
    printf("%d %d\n", a, r); // 3 3
}

```

Вопрос: зачем нужна ссылка, если с тем же успехом можно обращаться к самой переменной?

Ответ: одной из областей применения ссылок является использование в формальных параметрах функций. В этом случае происходит передача по адресу (значению адреса) без использования операции взятия адреса и разадресации.

```

void swap(int& a, int& b) {
    int tmp = a;
    b = a;
    a = tmp;
    printf("2.a=%d b=%d\n", a, b);
}

```

Другой способ использования ссылок – использование их при объявлении возвращаемого функцией типа. Это позволяет сделать вызов функции леводопустимым выражением

```

#include <stdio.h>
#include <assert.h>
const int arrSize = 8;
int& refItem(int i) {
    static int arr[arrSize];
    assert(i >= 0 && i < arrSize);
    return arr[i];
}
int main() {
    for (int i = 0; i < arrSize; i++) {
        refItem(i) = 1 << i;
        printf("arr[%02d]=%d\n", refItem(i));
    }
    return 0;
}

```

“assert” останавливает работу программы в случае значения аргумента, равного нулю.

Объявление массива arr с классом памяти static обеспечивает ему глобальное время жизни.

Цель этой функции следующая: осуществление проверки индексации массива.

Передача массива в функцию

Объявление формального параметра может содержать инициатор, то есть выражение, которое должно обеспечить присвоение параметру начального значения. Данный инициатор – не обязательно константное выражение, так как инициализация происходит не на стадии компиляции, а в ходе выполнения программы. Инициализация происходит только в том случае, если отсутствует соответствующий фактический параметр.

В **C++** действуют следующие синтаксические ограничения:

- ⇒ Инициализация производится, начиная с правого формального параметра
- ⇒ Список инициализированных параметров не допускает пропусков (неинициализированные параметры не могут чередоваться с инициализированными)

⇒
`void f(int, int);`

`void f(int, int = 1);`

`void f(int = 1, int);`

`void f(int = 1, int = 1);`

Прототипы функции могут находиться в разных областях видимости и содержать различные инициаторы

```
void f1(int a) {  
    int f(int, int = 1);  
    printf("%d\n", f(a));  
}
```

```
void f2(int a) {  
    int f(int, int = 2);  
    printf("%d\n", f(a));  
}
```

```
int main() {  
    f1(3);  
    f2(3);  
    return 0;  
}  
  
int f(int a, int b) {  
    return a * b;  
}
```

Функции с переменным числом параметров

При вызове функций с переменным числом параметров используется любое требуемое число аргументов. В определении и прототипе такой функции переменное число параметров задаётся

многоточием в конце обязательных формальных параметров. Вопрос – приходилось ли вам использовать функции с переменным числом параметров? Пример – printf, scanf.

```
int printf(const char*, ...);
```

Все аргументы, используемые при вызове функции, размещаются в стеке. Программист отвечает за определение количества этих параметров и за правильность выборки их из стека. Для обеспечения удобного доступа к необязательным параметрам можно использовать три макроса:

Va_start, va_arg и va_end, состоящие в библиотеке <stdarg.h>.

Макрос va_start предназначен для установки аргумента на начало списка необязательных параметров и имеет вид следующей функции:

```
void va_start(arg_ptr, right_param);
```

right_param – последний обязательный параметр функции; arg_ptr должен быть объявлен следующим образом: va_list arg_ptr;

Макрос va_arg обеспечивает доступ к очередному параметру и имеет следующий вид:

```
Type_arg va_arg(arg_ptr, type);
```

Va_arg извлекает значение типа type по адресу ... и увеличивает arg_ptr на величину type

Va_end обнуляет arg_ptr

```
Void va_end(arg_ptr);
```

Пример:

// для вычисления среднего арифметического произвольной последовательности неотрицательных целых чисел.
Концом последовательности будем считать -1.

```
#include <stdarg.h>
```

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

```
int main() {
```

```
    float avg(int, ...);
```

```
    printf("%f\n", avg(2, 3, 4, -1));
```

```
    printf("%f\n", avg(5, 6, 7, 8, 7, -1));
```

```
    return 0;
```

```
}
```

```
float avg(int x, ...) {
```

```
    int i = 0, j = 1, sum = 0;
```

```
    va_list arg;
```

```
    va_start(arg, x);
```

```
    if (x != -1) sum = x;
```

```
    else return 0;
```

```
    while ((i = va_arg(arg, int)) != -1)
```

```

    { sum += i; j++; }

    va_end(arg);

    return (float)sum / j;
}

```

Напишем функцию avg без использования макроса.

```

float avg(int x, ...) {
    if (x == -1) return 0;

    int sum = x, i = 1;

    int* arg = &x;

    arg++;

    while (*arg != -1) {
        sum += *arg;

        arg++;

        j++;
    }

    return (float)sum / j;
}

```

Передача параметров функции main()

Функция main() может быть определена с параметрами, которые могут быть переданы из внешнего окружения, например, из командной строки. Во внешнем окружении все данные представлены в виде строк. Для передачи параметров командной строки в функцию main() используются два аргумента. int main(int argc, char* argv[]);

Общепринятые, но необязательные имена этих параметров – argc и argv. Argc содержит число слов в командной строке, включая имя программы, а argv – сами эти слова.

|| D:\>cprog working "C program" 1 || где cprog – имя файла;

Функция main может содержать и третий параметр char* argp[], которые служат для передачи в программу параметров или переменных операционной системы.

Дело в том что у операционной системы есть переменная с именем path, которая хранит пути до всех файлов. Аргументы argc, argv и argp представлены в памяти следующим образом:

Argc[4]

Atgv[] -> [] -> [D:\cprog.exe\0]

[] -> [working\0]

[] -> [C program\0]

[] -> [1\0]

[NULL]

```
Argp[] -> [] -> [path=C:\;D:\10]
```

```
[] -> [include=D:\include\0]
```

```
[NULL]
```

Пример печати параметров командной строки операционной системы, передаваемых в программу

Динамические объекты отличаются от статических тем, что их временем жизни управляет программист. Память под динамические объекты выделяется в сегменте кучи (heap). Выделение и освобождение памяти может осуществляться двумя способами:

- ⇒ Первый способ – выделение памяти в соответствии с типом указателя. Данный способ появился в C++. Для выделения памяти используются операторы new, который состоит из ключевого слова new, за которым следует спецификатор типа [new int]. Выполнение данного оператора приведёт к выделению памяти в куче под один объект типа int. Особенностью работы с кучей является то, что размещённые в ней объекты не имеют имени. Поэтому оператор new возвращает не сам объект, а указатель на него [int *p = new int;]. Вторая особенность оператора new – память автоматически не инициализируется. Можно указать инициатор в круглых скобках (а со стандарта 14 – и в фигурных) [int* p = new int(0);]. В случае, если места в куче недостаточно, то генерируется исключение с типом «bad_alloc» (записывайте эту тарабарщину; рассказывать, что это я пока не буду). Для освобождения памяти используется оператор delete. Аргументом оператора является адрес памяти, выделенный с помощью оператора delete p;. Вопрос: следует ли проверять аргумент delete на значение “0” [if (p) delete p;]? Ответ: нет, оператор delete сам выполняет эту проверку. Ещё одно замечание: следует понимать разницу между временем жизни переменной p и динамического объекта, который этот указатель адресует. p живёт в соответствии с правилами создания и уничтожения статических объектов, а временем жизни динамического объекта управляет программист.

```
int main() { ...
    int i;
    char str[] = "ABCD";
    int* pi = &i;
    short* ps = 0;
    double* p1 = new double(33.);
    delete str; // неправильно, указатель str "живёт" в стеке, а не в куче
    delete pi; // аналогично, pi - адрес сегмента стека
    delete ps; // delete не имеет смысла, т.к. значение = 0, тем не менее, он безопасен
    delete p1; // хороший delete.
    ...
}
```

- ⇒ Рассмотрим основные ошибки, связанные с динамическим выделением памяти:
 - Не освобождение выделенной памяти (утечка памяти). Симптомы – программа через определённое время перестаёт работать. Пример ошибки (самая распространённая ошибка):

```
int n; // некоторое n
int* p;
for(int i = 0; i < n; i + 1) {
    p = new int;
    ...}
delete p;
```

- Повторное применение delete к одной и той же области памяти.

```
int* p = new int;
-----;
int* q = p;
-----;
delete p;
-----;
delete q;
```

- Изменение объекта после его удаления.

```
int* p = new int;
delete p;
*p = 7;
```

- ⇒ Второй способ – выделение памяти под не типизированные указатели. Данный механизм появился в C. Оператор new выделяет область памяти размером, соответствующим заданному типу. Программист имеет возможность выделить область памяти

произвольного размера. Для этого используются функции семейства `alloc`. Для их использования подключаем либо библиотеку `<alloc.h>`, либо `<stdlib.h>`. Первая функция, которую мы рассмотрим – функция `malloc`. Имеет следующий прототип: `[void* malloc(unsigned int size);]`. Выделяет блок памяти размером `size` байтов. В случае успеха возвращается указатель на выделенный блок, иначе – 0. Если `size = 0`, возвращается 0. Обратите внимание, все функции семейства `alloc` возвращают указатель на `void` – не типизированные данные. Пример:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main() {
    char* str;
    if (!(str = (char*)malloc(10))) {
        printf("Not enough memory\n");
        exit(1);
    }
    strcpy(str, "Hello");
    printf("string is %s\n", str);
    free(str);
    return 0;
}
```

Особенность, как и все функции семейства `alloc`, мы должны явно приводить к необходимому типу. К тому же, `malloc` не инициализирует выделенную память.

Функция `calloc`.

Прототип:

`void* calloc(unsigned int nitems, unsigned int size);`

`Calloc` выделяет блок памяти размером `n items`, умноженное на `size`. Память инициализируется нулями.

```
int main() {
    str = (char*)calloc(10, sizeof(char));
}
```

Третья функция семейства `alloc` – `realloc`;

прототип имеет следующий вид: `void* realloc(void* block, unsigned int size);`

`realloc` позволяет изменить размер крайнего выделенного блока памяти.

Block содержит адрес памяти, выделенный функциями семейства `alloc`. Если блок равен нулю, `realloc` работает как `malloc`. Size – новый размер блока. Если `size` равен нулю, то память освобождается. `Realloc` возвращает адрес нового блока, который может отличаться от старого. Содержимое старого блока копируется в новый.

```
int main() {
    char* str;
    str = (char*)malloc(10 * sizeof(char));
    strcpy(str, "Hello");
    printf("string is %s\nAddress is %p\n", str, str);
    str = (char*)realloc(str, 20 * sizeof(char));
    strcat(str, ", world!");
    printf("string is %s\nAddress is %p\n", str, str);
}
```

Последнее про функции семейства. Для освобождения памяти используется функция `free()`;

`void free(void* block);`

Здесь `block` – адрес памяти, выделенный семейством `alloc`.

Используя динамическую память, необходимо внимательно следить за её очисткой.

Динамические одномерные массивы можно создавать не только с помощью функций семейства `alloc`, но и с помощью оператора `new`. Количество элементов массива указывается в квадратных скобках после спецификатора типа.

__ Пример __

```
int n;  
  
scanf("%i", &n);  
  
int* arr = new int[n];  
  
delete[] arr;
```

__ Конец примера __

Для освобождения памяти из-под массива используется форма оператора `delete` с пустыми операторными скобками.

Динамические двумерные массивы.

Для создания двумерного массива необходимо в начале распределить память на массив указателей на одномерные массивы, а затем распределить память на эти одномерные массивы. Пусть необходимо составить двумерный массив из N строк и M столбцов.

__ Пример __

```
int main() {  
  
    int m, n;  
  
    scanf("%i%i", &m, &n);  
  
    // способ для C  
    double** a = (double**)calloc(m, sizeof(double*));  
  
    for (int i = 0; i < m; i++)  
        a[i] = (double*)calloc(n, sizeof(double));  
  
    // способ для C++  
    double** a = new double* [m];  
  
    for (int i = 0; i < m; i++)  
        a[i] = new double[n];  
  
}
```

__ Конец примера __

Массив типа указатель на указатель на `double`, т.к. элементами являются указатели на `double`.

__ Рисуем память кучи (heap) __

__ Конец рисунка памяти __

Освобождение памяти ведётся в обратном порядке.

__ Пример __

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



```
        free(a[i]); // delete a[i];

    free(a); // delete[] a;

__ Конец примера __
```

Выделим память под трёхмерный массив $N \times M \times L$, а также очистим его.

__ Пример __

```
int main() {

    int m, n, l;

    scanf("%i%i%i", &n, &m, &l);

    long*** a = new long** [m];

    for (int i = 0; i < m; i++) {

        a[i] = new long* [n];

        for (int j = 0; j < n; j++)

            a[i][j] = new long[l];

    }

    for (int i = 0; i < m; i++) {

        for (int j = 0; j < n; j++)

            delete[] a[i][j];

        delete[] a[i];

    }

    delete[] a;

}
```

__ Конец примера __

Рассмотрим пример, в котором память для массива распределяется вызываемой функцией, а сам массив используется вызывающей функцией.

__ Пример для C __

// Стилль C

```
void vvod(long** arr, int n) {

    *arr = (long*)calloc(n, sizeof(long));

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

        (*arr)[i] = i;

}
```

```
int main() {

    int n;

    scanf("%i", &n);

    long* b;
```

```

        vvod(&b, n);

        // некоторый код

        free(b);

        // некоторый код
    }

__конец примера__

__Пример для C++__

// Стил ь C++
void vvod(long* &arr, int n) {
    arr = new long[n];

    for (int i = 0; i < n; i++)
        arr[i] = i;
}

```

```

int main() {
    int n;

    scanf("%i", &n);

    long* b;

    vvod(b, n);

    // некоторый код

    delete[] b;

    // некоторый код
}

__конец примера для C++__

```

arr – ссылка на указатель на long.

Отметим то обстоятельство, что указатель на массив необязательно должен показывать на начальный элемент. Он может быть сдвинут так, что индекс начального элемента будет отличен от нуля.

__пример__

```

int main() {
    int* a = new int[3];

    a[0] = 1;

    a--;

    printf("%i\n", a[1]); // 1
}

```

```

        a += 2;

        printf("%i", a[-1]); // 1

        a--;

        delete[] a;
    }

```

__конец массива__

Массив с базой индексации равным единице

__пример__

```

int main() {

    int n, m;

    float** b = new float* [b];

    for (int i = 0; i < m; i++) {

        b[i] = new float[n];

        b[i]--;

    }

    b--;

    for (int i = 0; i < m; i++)

        delete[] ++b[i];

    delete[] ++b;

}

```

__конец примера__

Рассмотрим динамическое выделение памяти. Для массива указателей на функции, имеющих один параметр типа double и возвращающий параметр типа double.

__пример__

```

#define _CRT_SECURE_NO_WARNINGS

#include <bits/stdc++.h>

// <stdio.h>; <stdlib.h>; <math.h>

int main() {

    typedef double(*pfun)(double);

    pfun* masfun;

    double x = 0.5, y;

    masfun = (pfun*)calloc(3, sizeof(pfun));

    masfun[0] = cos;

    masfun[1] = sin;

    masfun[2] = tan;

    for (int i = 0; i < 3; i++) {

```

```
y = masfun[i](x);  
printf("x = %lf y = %lf\n", x, y);  
}  
free(masfun);  
return 0;  
}  
__ конец примера __
```

Важно правильно управлять памятью.

Директивы препроцессора.

Директивы препроцессора представляют собой инструкции, записанные в тексте С-программы и выполняемые до трансляции программы. Программа, обрабатывающая эти инструкции, называется препроцессором. Препроцессор изменяет текст программы. Все директивы препроцессора начинаются с символа # (октогорт). После директив препроцессора «;» не ставится.

Директива #include включает в текст программы содержимое указанного файла. Она имеет две синтаксические формы: #include "Имя файла" // #include <Имя файла>

Имя файла должно соответствовать соглашениям ОС и может содержать путь к файлу.

Если имя файла заключено в угловые скобки, то считается, что нужен стандартный заголовочный файл, и компилятор его ищет в предопределённом месте (зависит от ЯП, ОС и др.).

Если имя в кавычках, то заголовочный файл пользовательский и поиск начинается с текущего каталога. Директива *include* может быть вложенной, т.е. включаемый файл может также содержать директивы *include*. Следующая директива - **#define**

Директива **#define**.

Данная директива служит для замены часто использующихся констант, ключевых слов, операторов или выражений некоторыми идентификаторами. Идентификаторы, заменяющие константы – именованные константы. Идентификаторы, заменяющие фрагменты программ – макроопределениями. Макроопределения могут иметь аргументы. Директива `define` имеет две формы:

```
#define идент текст
```

```
#define идент(парам) текст
```

Руководствуясь данной директивой препроцессор заменяет все последующие вхождения идентификатора на текст. Пример:

```
#define WIDTH 80
```

```
#define LENGTH (WIDTH + 10)
```

```
__конец примера__
```

Препроцессор заменит все последующие вхождения `WIDTH` на `80`, а все последующие вхождения `LENGTH` на `(80 + 10)`.

Макроопределения также могут быть с параметрами.

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

```
__конец примера__
```

Чтобы отменить результат `define`, следует использовать директиву `#undef`.

Условные директивы препроцессора.

Заголовочный файл, подключаемый директивой `#include` также в свою очередь может содержать директивы `#include`. Поэтому возникает проблема повторного включения файлов, поэтому во избежание этой проблемы используют условные директивы препроцессора.

```
#ifndef идентификатор
```

```
#define идентификатор
```

```
...
```

```
#endif идентификатор
```

В `define` принято использовать верхний регистр

Если идентификатор не определён, не определён, то первый `define` определяет этот идентификатор. При попытке повторно определить файл, то он не сработает. Также есть условные директивы `#ifdef`; `#if`; `#else`; `#elif`.

С помощью препроцессора создаются языки программирования, которые могут переводиться препроцессором на Си.

Последняя тема данного семестра – **линейно односвязный список**.

Типы определены следующим образом.

```
struct List_Item {
```

```

int key;

List_Item *next;

};

```

Структура содержит некий ключ и адрес следующей аналогичной структуры.

Сделаем многофайловый проект (для этого есть окно менеджера проекта).

Для начала – заголовочный файл

```

//list.h

#ifndef _LIST
#define _LIST

struct List_Item{

    int key;

    List_Item* next;

};

typedef List_Item* List;

void PrintList(List); // Отладочная функция для печати списка. Аргумент - адрес начала списка.

List InsertBegin(List, int); // Вставка элемента в начало списка. Аргументы - адрес начала списка, ключ вставляемого
элемента. Возвращает адрес нового начала списка.

List FindItem(List, int); // Поиск элемента. Аргументы - адрес начала списка, искомый ключ. Возвращает адрес
найденного элемента, либо 0.

void InsertAfter(List, int); // Вставка элемента в список после указанного. Аргументы - адрес элемента, после
которого осуществляется вставка, ключ вставляемого элемента.

void DeleteAfter(List); // Удаление элемента после указанного. Аргумент - адрес указанного элемента.

void FreeList(List); // Освобождение памяти из под списка. Аргумент - адрес начала списка.

#endif

```

___конец заголовочного файла___

рисунок

Далее – файл с кодом:

```

// list.cpp

#include "list.h"

#include <stdio.h>

#include <stdlib.h>

void PrintList(List p) {

    while (p) {

```

```

        printf("%d ", p->key);

        p = p->next;
    }

    printf("\n");
}

```

```

List InsertBegin(List p, int x) {
    List q = new List_Item;

    q->key = x;
    q->next = p;
    return q;
}

```

```

List FindItem(List p, int x) {
    while (p && p->key != x) // p - проверка на пустой список; p->key!=x - проверка на соответствие элементов.
        p = p->next;

    return p;
}

```

```

void InsertAfter(List p, int x) {
    List q = new List_Item;

    q->key = x;
    q->next = p->next;
    p->next = q;
}

```

```

void DeleteAfter(List p) {
    List q = p->next;

    p->next = q->next;
    delete q;
}

```

___конец файла с кодом___

Задание на 2 семестр по этому коду:

1. Набрать (собрать многофайловый проект с кодом)

2. Отладить
3. В некоторых функциях выявить слабинку и исправить.
4. Добавить функцию удаления элемента по ключу.

Оглавление

Объектно-ориентированный подход к программированию.	2
Конструкторы	5
Разрушение объекта	7
Доступ к членам класса	8
Друзья класса	9
Статические члены класса.	12
Наследование	13
Перегрузка операций.	15
Перегрузка операции преобразования типа.	17
Переопределение ввода-вывода.	18
Манипуляторы.	19
Шаблоны класса.	25
Обработка исключительных ситуаций.	28
RTTI .	31
Функции-члены с модификаторами const и volatile.	33
Объявление mutable.	34
Вложенные классы.	35
Библиотека STL – Standard Template Library.	37
Преобразование типов в стиле C++	37
Библиотека Boost.	38
Стандарт C++ 11.	40
Умные указатели.	42
Пространства имён (namespace).	44
Определение члена пространства имён.	

Ошибка! Закладка не определена.

Объектно-ориентированный подход к программированию.

Предназначение понятия класса состоит в том, чтобы предоставить программисту инструмент для создания новых типов, столь же удобных в обращении, что и встроенные типы. В идеале пользовательский тип ничем не должен отличаться от встроенных типов. В определении нового типа основная идея – отделить несущественные подробности реализации (например, формат хранения данных) от тех качеств, которые необходимы для его правильного использования (список функций для управления данными). То есть, работа с данными и внутренними подпрограммами должна осуществляться через специальный интерфейс (канализироваться). Пример: определим понятие «дата» (календарная) и функции для работы с ней.

```
struct date {
    int day, month, year;
};
void set_date(date*, int, int, int);
void print(date);
```

В этом случае никакой связи между данными и функциями для их обработки нет. Такую связь можно установить, объявив (определив) функции внутри структуры (возможность появилась в C++).

```
struct date {
    int day, month, year;
    void set(int, int, int);
    void print();
};
void set_date(date*, int, int, int);
void print(date);
```

Уже не нужно передавать, для какой структуры нужно выполнять функции.

Для определения таких функций следует использовать квалификацию именем структурного типа.

```
void date::set(int d, int m, int y) {
    day = d;
    month = m;
    year = y;
}
```

Это называется «инструкцией по видимости», то есть откуда брать функции.

Для обращения к таким функциям используют тот же синтаксис, что и для обращения к полям структуры.

```
date d;
d.set(10, 2, 2023);
```

Внутри функции set можно добавить код, осуществляющий проверку корректности переданных данных. Но это не имеет смысла, так как никто не запрещает напрямую обратиться к полям. Такое ограничение можно наложить, используя вместо структуры – class.

```
class date {
    int day, month, year;
public:
    void set(int, int, int);
    void print();
    // -----
};
```

Метка public делит тело класса на две части: первая часть является закрытой, имена из этой части могут быть использованы только внутри функций, объявленных в классе. Вторая часть – открытая и представляет собой интерфейс для манипулирования данными. Имена из этой части доступны во всей программе. Тогда теперь присваивание вида d.day = 78; будет недоступно.

Определения:

Переменные, объявленные в классе называются **членами класса**, или **полями**.

Функции, объявленные (определённые) в классе называются функциями-членами класса (function-member), или методами.

Переменные, типом которых является класс, называются объектами класса, экземплярами класса, или инстанциями класса. [Класс – тип данных, переменная – объект данного типа.]

Механизм ограничения доступа к полям и методам с выделением интерфейсной части для манипуляции объектом называется инкапсуляцией.

Вопрос:

Каким образом функция-член класса, которая хранится в единственном экземпляре в сегменте кода узнаёт, для какого объекта она была вызвана?

Проблема: Объекты хранят только поля, а не данные.

Ответ: дело в том, что у методов класса есть скрытый параметр – указатель на объект, для которого вызывается эта функция. Этот указатель имеет имя `this`, и для класса `x` имеет следующий тип: `X* this`. На `this` можно ссылаться в теле функции, например:

```
void date::set(int d, int m, int y) {
    this->day = d;
    this->month = m;
    this->year = y;
}
```

В большинстве случаев использование `this` излишне.

Второй случай - когда используются адреса объектов при работе функции (`this` необходим).

Например – функция, которая вставляет элемент в односвязный список следом за текущим.

```
class dlink {
    dlink* pre, * suc;
public:
    void append(dlink*);
    // -----
}

void dlink::append(dlink* p) {
    p->suc = suc;
    p->pre = this;
    suc->pre = p;
    suc = p;
}
```

Inline.

При программировании с использованием класса часто используются множества небольших функций. Это снижает эффективность из-за стоимости вызова функций. Чтобы справиться с этой проблемой был разработан аппарат `inline`-функций. Функция, определённая (а не просто объявленная) в теле класса считается `inline`-функцией. Это означает, что вызовов таких функций не происходит, их тело подставляется в точку вызова.

```
class date {
    int day, month, year;
public:
    void set(int d, int m, int y) {
        day = d; month = m; year = y;
    }
};
```

Функцию как инлайн можно определить и вне тела класса:

```
inline void date::print(){
    //-----
}
```

Не всегда написанная инлайн функция будет инлайн, например – в рекурсивной функции.

Использование функций типа `set()` для инициализации объекта чревато ошибкой. Программист может забыть это сделать, либо сделать это повторно. В C++ есть механизм, позволяющий программисту описать функцию, явно предназначенную для инициализации объекта. Такая функция называется конструктором. В C++ такая конструкция распознаётся по тому, что имеет то же имя, что и сам класс. Для конструктора не указывается тип возвращаемого значения. Класс содержит конструктор по умолчанию, который не имеет параметров и не выполняет никаких действий. Программист имеет возможность заменить его своим. Класс может содержать

несколько конструкторов, различающихся параметрами. Если мы объявили свой конструктор, то конструктор по умолчанию не работает. Напишем два конструктора:

```
class date {
    int day, month, year;
    int isCorrect(int, int, int);
public:
    date() {
        day = 1; month = 1; year = 1900;
    }
    date(int, int, int);
    // -----
};

date::date(int d, int m, int y) {
    if (isCorrect(d, m, y)) {
        day = d; month = m; year = y;
    }
    else {
        day = 1; month = 1; year = 1900;
    }
}
```

Как работать с объектами? Теперь мы можем объявлять объекты либо с тремя параметрами, либо без параметров.

```
date d1(10, 2, 2023).d2, d3(15, 15, 15);
d1.print(); //10.02.2023
d2.print(); // 01.01.1900
d3.print(); // 01.01.1900
```

Ну и последнее. Иногда, чтобы не размножать конструкторы, используют значения по умолчанию.

```
date(int = 1, int = 1, int = 1900);
```

Иногда такой способ заменяет несколько конструкторов.

Конструкторы

17.02.23

Объект может строиться не только на основе какой-либо переданной информации, но и как копия однотипного объекта.

```
date d1(17, 2, 2023);
date d2 = d1;
```

Копирование одного объекта в другой выполняется путём последовательного копирования каждого члена класса и осуществляется конструктором копирования по умолчанию. Если такое поведение не устраивает, программист должен переопределить конструктор копирования. Конструктор копирования имеет единственный параметр – константную ссылку на однотипный объект. Например, для класса **X** конструктор копирования будет выглядеть таким образом: **X(const X&)**; Сейчас мы рассмотрим пример, в котором вы увидите что конструктор копирования иногда нужен.

```
#include <stdio.h>
using namespace std;

class A {
public:
    int* a, N;
    A(int);
    void print();
};
A::A(int n) {
    N = n;
    a = new int[n];
    for (int i = 0; i < N; i++)
        a[i] = i;
}
void A::print() {
    for (int i = 0; i < N; i++)
        printf("%d ", a[i]);
    printf("\n");
}

class B {
public:
    int* a, N;
    B(int);
    B(const B&);
    void print();
};
B::B(int n) {
    N = n;
    a = new int[n];
    for (int i = 0; i < N; i++)
        a[i] = i;
}
void B::print() {
    for (int i = 0; i < N; i++)
        printf("%d ", a[i]);
    printf("\n");
}
B::B(const B & v) {
    N = v.N;
    a = new int[N];
    for (int i = 0; i < N; i++)
        a[i] = v.a[i];
}

int main() {
    A a1(3);
    A a2 = a1;
```

```
    a2.a[1] = 8;
    printf("Object a1 - ");
    a1.print();
    printf("Object a2 - ");
    a2.print();
    B b1(3);
    B b2 = b1;
    b2.a[1] = 8;
    printf("Object b1 - ");
    b1.print();
    printf("Object b2 - ");
    b2.print();
    return 0;
}
// Вывод:
// Object a1 - 0    8    2
// Object a2 - 0    8    2
// Object b1 - 0    1    2
// Object b2 - 0    8    2
```

Разрушение объекта

Для некоторых классов требуется при разрушении объекта освобождают захваченные ресурсы, то есть, выполнить действия, обратные конструктору. Существует возможность определить функцию, которая автоматически вызывается при разрушении объекта. Такая функция называется деструктором. Для класса X деструктор выглядит как ~X. Деструктор не имеет параметров. Для деструктора не указывается возвращаемое значение.

```
~B() {  
    delete[] a;  
}
```

Доступ к членам класса

Соккрытие информации – это формальный механизм, ограничивающий доступ к внутреннему представлению класса. В теле класса может быть объявлено 3 вида идентификаторов: члены класса (поля), функции-члены класса (методы) и вложенные типы. Доступ к этим именам контролируется с помощью секций тела класса, помеченных ключевыми словами *public*, *private* и *protected* (спецификаторы доступа). Секция *public* называется открытой, имена из этой секции доступны с любого места программы. Секция *private* является закрытой. Имена из этой секции доступны только в функциях-членах класса (методах) и в дружественных функциях. Секция *protected* называется защищённой. Имена доступны из функций-членов класса (методов), из дружественных функций, и из функций-членов производных классов. Замечания по синтаксису: в классе может быть несколько одноимённых секций; каждая секция продолжается либо до метки следующей секции, либо до закрывающейся фигурной скобки “}”; по умолчанию секция, следующая за открывающейся фигурной скобкой, считается *private* (в то время как в структурах секция по умолчанию – *public*);

Друзья класса

Пример: предположим, определены два класса – *vector* и *matrix*. Для простоты допустим, что вектор состоит из четырёх элементов, а матрица – из четырёх векторов. Доступ к элементам матрицы и вектора осуществляется через метод *elem()*, где осуществляется проверка допустимости индекса. Необходимо написать функцию умножения матрицы на вектор.

```
vector_multiply(const matrix& m, const vector& v) {  
    vector r;  
    for (int i = 0; i < 4; i++) {  
        r.elem(i) = 0;  
        for (j = 0; j < 4; j++)  
            r.elem(i) += m.elem(i, j) * v.elem(j);  
    }  
    return r;  
}
```

Метод *elem()* вызывается 52 раза. Это неэффективно, так как в данном алгоритме нет смысла в проверке индексов. Если сделаем *multiply* членом класса *vector*, то сможем непосредственно обращаться к элементам вектора. Если сделаем *elem()* членом класса *matrix*, то появится возможность непосредственно обращаться к элементам матрицы. Но одновременно членом двух классов функция быть не может. Механизм дружественных функций позволяет открыть для функции доступ к закрытой части класса. Других особенностей дружественная функция не имеет. Она не получает указатель *this*.

```
class vector {  
    float v[u];  
    //-----  
    friend vector_multiply(matrix&, vector&);  
};  
class matrix {  
    vector v[u];  
    //-----  
    friend vector_multiply(matrix&, vector&);  
};
```

```
vector_multiply(const matrix& m, const vector& v) {  
    vector r;  
    for (int i = 0; i < 4; i++) {  
        r.v[i] = 0;  
        for (j = 0; j < 4; j++)  
        }  
        r.v[i] += m.v[i].v[j] * v.v[j];  
    }  
    return r;  
}
```

Замечания по поводу...

Пример: необходимо реализовать битовое множество целых однобайтовых беззнаковых (*unsigned*). Битовое множество не содержит элементов, а содержит информацию об их наличии или отсутствии. Битовое множество является стандартным типом языка Pascal.

```
//pascalset.h  
#ifndef _PASCALSET  
#define _PASCALSET  
#include <memory.h>  
  
class PasSet {  
    unsigned char data[32]; // хранит информацию о наличии/отсутствии элементов множества.  
    unsigned char getMask(unsigned char x) {  
        return 1 << (x % 8);  
    }  
};
```

```

    }
    unsigned char getNumByte(unsigned char x) {
        return x / 8;
    }
public:
    PasSet() { memset(data, 0, sizeof(data)); }
    int inSet(unsigned char); // проверка вхождения элемента во множество.
    void include(unsigned char); // включение элемента во множество.
    void exclude(unsigned char); // исключение элемента из множества.
    void print(); // отладочная функция.};
#endif // !_PASCALSET

```

Конец .h-файла

```

// pascalset.cpp
#include <stdio.h>
#include "pascalset.h"
int PasSet::inSet(unsigned char x) {
    unsigned char nbyte = getNumByte(x);
    unsigned char mask = getMask(x);
    return data[nbyte] & mask;
}

void PasSet::include(unsigned char x) {
    unsigned char nbyte = getNumByte(x);
    unsigned char mask = getMask(x);
    data[nbyte] |= mask;
}

void PasSet::exclude(unsigned char x) {
    unsigned char nbyte = getNumByte(x);
    unsigned char mask = getMask(x);
    data[nbyte] &= ~mask;
}

void PasSet::print() {
    for (int i = 0; i < 256; i++)
        if (inSet((unsigned char)i))
            printf("%d ", i);
    printf("\n");
}

```

Конец .cpp-файла

```

// mainset.cpp
#include <stdio.h>
#include "pascalset.h"
using namespace std;
int main() {
    PasSet s;
    s.include(8); s.include(3); s.include(5);
    s.print(); // 3 5 8
    s.exclude(3);
    s.print(); // 5 8
    if (s.inSet(5))
        printf("5 is in set\n");
    else printf("5 isn't in set\n");
    // 5 is in set
    return 0;
}

```

Конец main.cpp-файла

Статические члены класса.

В каждом объекте находится своя копия членов класса (полей). Однако, иногда возникает необходимость в том, чтобы все однотипные объекты могли совместно использовать некоторые данные. Например, в классе – односвязный список, все элементы списка должны знать адрес начала списка. Описание поля `begin` как статик означает, что он будет в единственном экземпляре и не копироваться в каждый объект класса `List`.

```
class List {  
    static List* begin;  
    //-----  
public:  
    //-----  
};
```

Инициализируются статические члены класса в глобальной области видимости ниже класса.

```
List* List::begin = 0;
```

При работе со статическими членами класса можно использовать обычный синтаксис

```
p->begin;
```

Но так делать не следует, и при доступе следует использовать квалификацию

```
List::begin
```

Наследование.

Наследование – один из основополагающих принципов (парадигм) ООП. Под наследованием понимают возможность определения производных типов на основе ранее объявленных. Следует отличать наследование от другого вида иерархии – композиции (встраивания). Встраивание предполагает возможность объявления в классе отдельных полей на основе ранее объявленных классов. В случае наследования класс создаётся на основе ранее объявленного класса, он наследует и, возможно, модифицирует его данные и функции. Получившийся класс может служить базовым для новых производных классов. В С++ количество непосредственных предков производного класса не ограничено. Если родительский класс один, то говорят об одиночном, или простом наследовании. Если более одного – то о множественном наследовании. Наследование в С++ реализовано таким образом, что наследуемые компоненты не перемещаются в производный класс, а остаются в базовом классе. В результате объект производного класса содержит подобъект базового класса.

```
class A {  
public:  
    A() { printf("A\n"); }  
    ~A() { printf("~A\n"); }  
    int x0;  
    int f0() { return 1; }  
};
```

```
class B : public A {  
public:  
    B() { printf("B\n"); }  
    ~B() { printf("~B\n"); }  
    int x1, x2, xx;  
    int f1() { return 100; }  
    int f2() { return 200; }  
};
```

```
class C : public B {  
public:  
    C() { printf("C\n"); }  
    ~C() { printf("~C\n"); }  
    int x1, x2, x3;  
    int f1() { return 1000; }  
    int f3() { return 3000; }  
};
```

Приведён пример одиночного наследства. Класс Б является наследником класса А, класс С является наследником класса Б.

Слово public перед именем базового класса означает, что открытые поля родительского класса в производном также будут открыты.

В Си++ различают непосредственные и косвенные базовые классы. Для класса С непосредственным базовым классом является класс Б, а косвенным – класс А.

Объявление приводит к созданию объекта сложной структуры. Объект класса С включает в себя подобъекты класса Б и А. В классе С приняли участие три конструктора. Порядок их вызова регламентирован в направлении от базовых классов к производным. Порядок вызовов деструкторов противоположен. Рассмотрим структуру объекта класса С.

Класс может использовать без изменения то, что наследовано, или переопределить.

На основании класса paset создадим однобайтовое множество беззнаковых, создадим класс paset. Для этого унаследуем класс и изменим метод print.

```
class PasSetChar : public PasSet {  
public:  
    void print();
```

```
};  
void PasSetChar::print() {  
    for (int i = 0; i < 256; i++)  
        if (inSet((unsigned char)i))  
            printf("%c ", (char)i);  
    printf("\n");  
}
```

Перегрузка операций.

Программист может определять смысл операций при применении к объектам определенного класса. Кроме стандартных операций можно переопределять операции индексирования [], операцию вызова функции (), new и delete[]. Кроме того, можно переопределять операцию преобразования типа между пользовательским типом и стандартным типом данных. Рассмотрим пример.

```
class complex {
    double re, im;
public:
    complex(double r = 0, double i = 0)
    {re = r; im = i;}
    friend complex operator +(complex, complex);
    friend complex operator *(complex, complex);
};

complex operator +(complex a, complex b) {
    complex c;
    c.re = a.re + b.re;
    c.im = a.im + b.im;
    return c;
}

complex operator *(complex a, complex b) {
    complex c;
    c.re = a.re * b.re - a.im * b.im;
    c.im = a.re * b.im + a.im * b.re;
    return c;
}
```

Функции с именами оператор + и оператор * задают смысл операции сложения и умножения для комплексных чисел. Приоритет операций сохраняется.

```
int main() {
    complex a(1., 2.), b(2., 1.);
    complex c = a + b; // или же complex c = operator+(a, b);
    complex d = b * (a + c);
}
```

Существует возможность переопределять функции не только с помощью дружественных функций, но и с помощью функций-членов класса. В этом случае количество аргументов на единицу меньше, т.к. первым аргументом операции выступает текущий объект.

```
complex operator+ (complex b) {
    complex c;
    c.re = re + b.re;
    c.im = im + b.im;
    return c;
}
```

Ещё один тренировочный пример

```
class x{
    friend x operator - (x); // унарный минус
    friend x operator - (x, x); // бинарный минус, вычитание
    friend x operator - (); // ошибка, нет аргумента
    friend x operator - (x, x, x); // ошибка, тернарного минуса не существует
    x* operator &(); // операция взятия адреса
    x operator &(x); // битовое "и"
    x operator &(x, x); // ошибка, тернарного амперсанта нет
}
```

```
};
```

Добавим в класс PasSet оператор выделения множества.

```
class PasSet {  
    unsigned char data[32]; // хранит информацию о наличии/отсутствии элементов множества.  
    friend PasSet operator +(const PasSet&, const PasSet&);  
};  
  
PasSet operator + (const PasSet& a, const PasSet& b) {  
    PasSet c;  
    for (int i = 0; i < 32; i++)  
        c.data[i] = a.data[i] | b.data[i];  
    return c;  
}
```


Перегрузка операции преобразования типа.

`X::operator T();`

Функция-член класса, где T — имя типа определяет преобразование из X в T. Например, класс `tiny` — целые числа в диапазоне от 0 до 63.

```
class tiny {
    char v;
    int assign(int i) { return v = (i & ~63) ? 0 : i; }
public:
    tiny(int i) { assign(i); }
    tiny(tiny& i) { v = i.v; }
    int operator=(tiny& i) {
        return v = i.v;
    }
    int operator=(int i) {
        return assign(i);
    }
    operator int() { return v; }
};

int main() {
    tiny c1 = 2;
    tiny c2 = 62;
    tiny c3 = c2 - c1; // 60
    tiny c4 = c3;      // 60
    int i = c1 + c2;   // 64
    c1 = c2 + 2 * c1;  // 0
    c2 = c1 = c1 - i;  // 0
}
```

Вместо переопределения всех операций достаточно переопределить операцию приведения к типу.

Переопределение ввода-вывода.

В Си++ предоставляет группу классов, объектов и функций для ввода-вывода библиотеки `<iostream>`. При потоковом вводе-выводе синтаксис более простой и интуитивный, но основное преимущество заключается в том, что механизм потоков можно использовать как для стандартных, так и для пользовательских типов данных. { например, как вывести из предыдущего примера объект `c1`? Ведь нельзя для `printf` придумать спецификатор вывода, но если мы будем использовать механизм потоков для вывода, то это можно сделать.

В библиотеке определены 4 потоковых объекта: `[cin]`, `[cout]`, `[cerr]` (стандартный вывод ошибок) и `[clog]` – буферизованная версия `[cerr]`. Существует возможность перенаправлять потоки на другие устройства и файлы. Для работы с потоками переопределены операции битового сдвига. `[<<]` – записать в поток, `[>>]` – прочитать в потоке. Объект `[cin]` принадлежит классу `{istream}`, остальные объекты – классу `{ostream}`. Рассмотрим пример.

```
int a, b, c;
cout << "Input A and B "; // поток вывода, по умолчанию связан с экраном.
cin >> a >> b; // поток ввода, по умолчанию связан с клавиатурой.
c = a + b;
cout << "Sum = " << c << '\n';
```

Как и в функции `printf()` при выводе можно осуществлять форматирование.

```
int i = 87;
cout.width(7);
cout.fill('*');
cout << i << '\n'; // [*****87]
cout << i << '\n'; // [87]
```

```
double d = 3.14159;
cout.precision(4);
cout.width(8);
cout << d << '\n'; // [ 3.142]
```

Манипуляторы.

Существует альтернативный механизм форматирования – манипуляторы. Манипуляторы – специальные инструкции, включаемые в цепочку потоковых операций.

Например, вывести d из прошлого примера можно так:

```
cout << setw(8) << setprecision(4) << d << endl;
```

Различают параметризованные и не параметризованные манипуляторы. Для использования параметризованных манипуляторов нужно подключить библиотеку `<iomanip>`.

Ввод-вывод двоичных данных.

Для вывода используются следующие методы:

```
ostream& ostream::put(char); // помещает в поток 1 байт
```

```
ostream& ostream::write(const char* buff, int n); // помещает в поток указанное количество байт из буфера
```

Для ввода используются две версии `get()`:

```
istream& istream::get(char&); // считывает из потока 1 байт
```

```
istream& istream::get(char* buff, int max, int term = '\n'); // считывает из потока байты и помещает их в буфер до тех пор, пока либо не будет символ-ограничитель(терминатор), либо пока не будет прочитано max байт
```

Ввод-вывод списка.

Для работы с файлами удобно использовать библиотеку `<fstream>` и определены классы `ifstream` и `ofstream` для записи файлов.

Необходимо написать утилиту, которая осуществляет побайтовое копирование файлов.

Консольные утилиты бывают с интерфейсом командной строки. Как в `main` получить параметры командной строки?

Утилита копирования:

```
// dcopy
#include <iostream>
#include <fstream>
#include <process.h>
using namespace std;
int main(int argc, char argv[]) {
    char ch;
    if (argc != 3) {
        cerr << "dcopy file1 file 2\n";
        wxexit(-1);
    }
    ifstream source;
    ofstream dest;
    source.open(argv[1]);
    if (!source) {
        cerr << "Can't open source file\n";
        exit(-1);
    }
    dest.open(argv[2]);
    if (!dest) {
        cerr << "Can't open destination file\n";
        exit(-1);
    }
    while (dest && source.get(ch))
        dest.put(ch);
    if (dest)
        cout << "dcopy completed\n";
    else cerr << "copy error\n";
    source.close();
    dest.close();
    return 0;
}
```

Ввод-вывод с диска.

Для работы с файлами удобно использовать библиотеку <fstream> в ней определены классы ifstream, ofstream для записи файла наследован от класса ifstream.

Рассмотрим пример, необходимо написать утилиту которая осуществляет побайтовое копирование файла.

```
//dcopy
#include <iostream>
#include <fstream>
#include <process.h>
int main(int argc, char* argv[]) {
    char ch;
    if(argc != 3) {
        cerr << "dcopy file1 file2\n";
        exit(-1);
    }
    ifstream source;
    ofstream dest;
    source.open(argv[1]);
    if(!source) {
        cerr << "cant open source file\n";
        exit(-1);
    }
    dest.open(argv[2]);
    if (!dest) {
        cerr << "cant open destination file\n";
        exit(-1);
    }
    while(dest && source.get(ch))
        dest.put(ch);
    if(dest)
        cout << "dcopy completed\n";
    else cerr << "copy error\n";
    source.close();
    dest.close();
    return 0;
}
```

Пример: ввод, вывод с диска массива структур.

```
#include <iostream>
#include <fstream>
#include <string.h>
using namespace std;

int main()
{
    struct emp/ {
        char name[100];
        int dept;
    };
    emp/ e[2];
    strcpy(e[0].name, "aaaa");
    e[0].dept = 1;
    strcpy(e[1].name, "bbbbbbb");
    e[1].dept = 2;
    ifstream source;
    ofstream dest;
    char filename[] = "s.dat";
    dest.open(filename);
    for(int i = 0; i < 2; i++)
```

```

        dest << e[i].name << " " << e[i].dept << endl;
    dest.close();
    source.open(filename);
    for(int i = 0; i < r; i++)
    {
        source >> e[i].name >> e[i].dept;
        cout << e[i].name << " " << e[i].dept << endl;
    }
    source.close();
    return 0;
}
/*
sdat
Aaaa 1
Bbbbbbb 2
*/

```

Если поле name может содержать пробелы, то такой способ ввода/вывода непригоден. Следует использовать разделитель отличный от "пробела", например коммерческая "@". Запись в файл будет выглядеть следующим образом.

```

//sdat
//aaaa@1
//bbbbbbb@2

for(int i = 0; i < 2; i++)
    dest << e[i].name << '@' << e[i].dept << endl;

```

Чтение будет следующим:

```

char buf[1];
for(int i = 0; i < 2; i++)
{
    source.get(e[i].name, 100, '@');
    source.get(); // сбрасывает ограничитель.
    source.get(buf, 16, '\n');
    source.get();
    e[i].dept = atoi(buf); //Atoi анализирует строчку и забирает число.
    cout << e[i].name << " " << e[i].dept << endl;
}

```

Ввод / вывод для пользовательских типов данных:

```

class emp/ {
    char name[100];
    int dept;
    float sales;
public:
    // -----
};

```

Необходимо определить операцию потокового вывода. Сделаем это через дружественную функцию.

```

class emp/ {
    char name[100];
    int dept;
    float sales;
public:
    // -----

```

```

        friend ostream& operator<<(ostream& , Emp/ &);
};

ostream& operator<<(ostream& str, Emp/ & e) {
    str << e.name < " : Отдел " << setw(4) << e.dept << tab << "з/п " << e.sales << "р.\n ";
    return str;
}

```

При переопределении должна возвращаться ссылка на потовый объект для того чтобы можно было продолжить цепочку операций ввода/вывода.

Пример.

```

emp e1("Иванов", 127, 1234), e2("Сидоров", 46, 4321);
cout << e1 << e2;

```

Иванов: отдел 127 з/п 1234 р.

Сидоров: отдел 46 з/п 4321 р.

Манипулятор tab определен следующим образом:

```

ostream& tab b(ostream& str) {
    return str << '\t';
}

```

Шаблоны функции

Предположим у нас есть функция преобразующая алгоритм сравнения двух величин.

```

int min(int a, int b) {
    return a < b ? a : b;
}

```

Хотелось бы иметь аналогичную функцию для других типов данных. Можно попробовать решить её средствами препроцессора #define.

```

#define MIN(a,b) ((a) < (b) ? (a) : (b))

```

Но это не всегда возможно.

C++ предоставляет еще одно средство для решения этой задачи - шаблон функции. Шаблон функции позволяет определить семейство функции. Это семейство характеризуется общим алгоритмом, применяемым к разным типам данных. Определим шаблон функции для класса функции.

```

template <class T> //заголовок шаблона
T min(Ta, Tb) {
    return a < b ? a : b;
}

```

Объявление шаблона функции начинается ключевым словом `template`, за которым следует заключение в угловые скобки список параметров шаблона.

Каждый параметр шаблона состоит из слова `class` либо слова `typename`, за которым следует идентификатор.

Все идентификаторы из списка параметров шаблона обязаны входить в список параметров функции и играть там роль спецификации типа.

Кроме шаблонных параметров в функции могут быть и обычные.

На основе анализа вызова шаблонов функции транслятор самостоятельно генерирует семейство функции с нужными параметрами.

Пример

```
min(1, 2)
min(1., 2.)
```

Транслятор сгенерирует 2 варианта функции `min`: первый вариант с типами данных `int`, второй с типами `double`.

Пример

Определим шаблон функции `neq()` для определения неравенства данных различных типов

Код

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

template <class Type>
int neq<Type, Type>;
class ComplexType {
    double real, imag;
public:
    ComplexType(double re = 0., double im = 0.) {
        real = re; imag = im;
    }
    int operator != (ComplexType& c) {
        return !(real == c.real && imag == c.imag);
    }
};

int main() {
    int l = 1, j = 2;
    float k = 1, l = 2;
    ComplexType C1(1., 1.), C2(2., 2.);
    cout << "neq() for int:" << neq(l, j) << endl;
    cout << "neq() for float:" << neq(k, l) << endl;
    cout << "neq() for ComplexType:" << neq(C1, C2) << endl;
    return 0;
}

template <class Type>
int neq<Type a, Type b> {
    return a != b;
}
```

Пример. Этот пример подтверждает обязательность включения всех параметров шаблона, список параметров определяемой функции. Независимо от того какая роль предназначается шаблонному параметру его присутствие в списке параметров обязательно.

Пример говорит о том что мы хотим объявить шаблон и тело функции

```
#include <iostream>
#include <typeinfo>
using namespace std;
template <class YYY, class ZZZ>
YYY tf(ZZZ, YYY, int);
int main() {
    cout << tf((int)0, '1', 2) << endl;
    cout << tf((float)0, "This is the string...", 3) << endl;
    return 0;
}
```

```
template <class YYY, class ZZZ>
YYY tf(ZZZ, YYY y, int x) {
    ZZZ z;
    for (int i = 0; i < x; i++)
        cout << "tf() for " << typeid(z).name() << endl;
    return y;
}
```

Давайте разбираться. Функция tf у нас используется до определения, поэтому нужен прототип функции.

Смотрим на определение. Первый параметр идёт без имени, нам не нужно его значение, только тип. С помощью typeid() выводится тип переменной z.

Шаблоны класса.

Шаблон класса – предписание для создания класса, в котором один или несколько типов параметризованы. Например, необходимо создать класс вектор, который может содержать произвольные типы данных.

```
template <class Type>
class Vectir {
    Type* arr;
    int size;
public:
    vector(int);
    ~vector();
    Type& operator[](int);
    //-----
};
```

Ну, идея такая же как и с функциями. Чтобы не создавать массив целых и т.д., мы создаём шаблон, по которому создаётся семейство классов. При объявлении объекта в угловых скобках указывается требуемый тип данных.

```
array <int>a;
array <double>b;
array <complexType> c;
```

Встретив эти три объявления объектов генератор сгенерирует три класса.

Пример: реализуем шаблонный класс stack на основе динамического массива. Стек – последовательность дисциплины обслуживания – LIFO.

Пример заголовочного файла

```
// <stack.h>
#ifndef _CSTACK
#define _CSTACK
#include <iostream>
#include <stdlib.h>
using namespace std;

template <class Type>
class CStack {
    Type* data; // адрес динамического массива, указатель на динамический массив
    int top; // индекс вершины стека
    int size; // размер динамического массива
public:
    CStack(int);
    ~CStack() { free(data); }
    void push(Type); // добавление элемента в стек
    int pop(Type&); // выталкивание элемента из стека, метод возвращает успех/неуспех, поэтому он типа int,
    а сам элемент возвращается через параметр
    int isempty() { return !top; } // проверка заполненности стека
    void print();
};

template <class Type>
CStack<Type>::CStack(int sz) {
    data = (Type*)calloc(sz, sizeof(Type));
    top = 0;
    size = sz;
}

template <class Type>
void CStack<Type>::push(Type item) {
```

```

        if (top == size) {
            size += 8;
            data = (Type*)realloc(data, size * sizeof(Type));
        }
        data[top++] = item;
    }
}
template <class Type>
int CStack<Type>::pop(Type& item) {
    if (isempty()) return 0;
    item = data[--top];
    return 1;
}
template <class Type>
void CStack<Type>::print() {
    for (int i = top - 1; i >= 0; i--)
        cout << data[i] << ' ';
    cout << endl;
}
#endif

```

Пример тестирующей программы

```

#include <stdio.h>
#include <math.h>
#include "cstack.h"
int main() {
    printf("Int\n");
    CStack<int> ti(2);
    for (int i = 0; i < 4; i++)
        ti.push(i);
    ti.print();

    int k;
    for (int i = 0; i < 7; i++)
        if (ti.pop(k)) printf("%d\n", k);
        else printf("Empty\n");
    printf("Double\n");

    CStack<double> td(2);
    for (int i = 10; i < 14; i++)
        td.push(sqrt((double)i));
    td.print();
    double fl;
    for (int i = 0; i < 7; i++)
        if (td.pop(fl)) printf("%lf\n", fl);
        else printf("Empty\n");
    return 0;
}

```

Конец примеров. Вывод программы:

```

Int
3 2 1 0
3
2
1
0
Empty
Empty
Empty

```

```

Double
3.60555 3.4641 3.31662 3.16228
3.605551
3.464102
3.316625

```

3.162278

Empty

Empty

Empty

Обработка исключительных ситуаций.

Существует категория ошибок, которые не способны выявить препроцессоры, трансляторы и программы сборки, например, Runtime errors, появляющиеся в ходе выполнения программы. В терминах ООП такие ошибки называют исключительными ситуациями.

Исключительная ситуация – событие, прерывающее нормальный ход выполнения программы.

Различают синхронные и асинхронные исключительные ситуации.

Синхронная исключительная ситуация возникает непосредственно в ходе выполнения программы, причём её причина заключается непосредственно в действиях, выполняемых самой программой. Среди таких: операции деления на ноль, некорректная работа с адресами и т.п. Асинхронные исключительные ситуации непосредственно не связаны с выполнением программы. Их причинами могут быть аппаратные прерывания. Реакцией на исключительные ситуации являются исключения (exceptions).

Исключительная ситуация не всегда неожиданна. Часто при разработке алгоритма закладывается ряд реакций на исключительные ситуации. Например, пользователь должен ввести число, а он вводит текст. Рассмотрим пример: функция, проверяющая корректность индекса при работе с массивом. Данная функция возвращает единицу, в случае корректного индекса и 0 – в случае некорректного индекса. Существует ряд проблем, связанных с данной реализацией программного кода: чем больше видов возвращаемых значений, тем менее наглядным становится код функции и анализ возвращаемых значений. Кроме того, существуют функции, для которых тип возвращаемого значения не указывается: конструкторы, деструкторы и операции приведения типа. В Си++ имеются встроенные средства для генерации и обработки исключительных ситуаций. С помощью этих средств активизируется механизм, позволяющий двум несвязанным (независимо разработанным) фрагментам программы обмениваться информацией об исключительных ситуациях. Когда встречаются исключительные ситуации, та часть программы, которая её обнаружила, может сгенерировать (возбудить/выбросить) исключение. Генерация исключения обеспечивается операцией throw [выражение]. Параметром функции является произвольное выражение. Оператор операции исключения может находиться в любом месте программы. Его месторасположение называют точкой генерации.

```
throw 1; throw "ABC"; throw 7; throw NULL; throw &x;
```

Выполнение необработанного генератора throw приводит к завершению выполнения программы. Обычно оператор throw используется в сочетании с try-блоком (блоком обработки/испытания). Их взаимодействие обеспечивается через стек вызова. Поэтому точка генерации исключения должна находиться в теле функции, непосредственно или косвенно вызываемой из множества операторов try-блока. Try-блок имеет следующий формат:

```
Try {
```

```
    Оператор 1;
```

```
    Оператор 2;
```

```
    -----  
    Оператор N;
```

```
}
```

Список_реакций.

Список реакций состоит из одного или нескольких catch-предложений, например:

```
Catch(тип1 идент) {
```

```
    Операторы
```

```
}
```

```
Catch(тип 2 идент) {
```

```
    Операторы
```

```
}
```

```
-----  
Catch(тип X идент) {
```

```
    Операторы
```

```
}
```

При возникновении исключительной операции выполнение операторов try-блока прекращается. Среди catch-предложений ищется то, тип которого совпадает с типом выражения в операторе throw, идентификатор получает значение этого выражения и выполняются соответствующие операторы, после чего управление передаётся на следующий за try-блоком оператор.

Если будет оператор throw внутри try-блока,
то на экзамене больше «3» не будет,
а на работе вас расстреляют.

Пример. Вернёмся к классу CStack. Метод .pop() возвращал значение 0 в случае попытки выталкивания из пустого стека. В случае успеха значение возвращалось через параметр функции. Более естественен способ, когда функция при успехе возвращает значение, а при попытке выталкивания из пустого стека возбуждается исключение.

```
class StackException {
public:
    char mes[256];
    StackException(char* m) {
        strcpy(mes, m);
    }
};

template <class Type>
class CStack {
    //-----
    Type pop();
    //-----
};

template <class Type>
class CStack<Type>::pop() {
    if (isempty())
        throw StackException("Stack is empty.");
    return data[--top];
}

#endif

//cstackmain.cpp
#include <stdio.h>
#include <math.h>
#include <iostream>
//#include "cstack.h"
int main() {
    CStack cin->stack(10) :
        int a;
    for (int i = 0; i < 5; i++)
        stack.push(i);
    try {
        for (int i = 0; i < 9; i++) {
            a = stack.pop();
            cout << a << endl;
        }
    }
    catch (StackException e) {
        cout << e.mes << endl;
    }
    return 0;
}
```

Что происходит? В стек записали 5 элементов, пытаемся выпихнуть 9. Вопрос: почему сообщение о пустом стеке вывели только 1 раз? Ответ: try-блок прервался.

Catch-предложения вида catch (...) перехватывают все ранее не перехваченные исключительные ситуации.

По этой теме мы прошли практически всё, чтобы вы могли решать задачи на классы.

RTTI .

Аббревиатура RTTI обозначает Real Time Type Identification – определение типа во время выполнения программы. Для работы с этим механизмом необходимо подключить библиотеку `<typeinfo>`. В библиотеке определена операция `typeid`, у которой может быть два вида аргументов: либо имя типа, либо выражение. Операция возвращает константную ссылку на объект типа (класса) `type_info`. В классе определены операции сравнения на равенство и неравенство (`==`; `!=`), а также методы (`.name()`; `.before()`); `.name()` возвращает указатель на строку с именем типа, `.before()` возвращает истину, если класс объекта является базовым по отношению к классу аргумента.

```
#include <typeinfo>
#include <iostream>
using namespace std;
template <class T1, class T2>
void t(T1 p1, T2 p2) {
    cout << "p1 = " << p1 << " Type - " << typeid(p1).name() << endl;
    cout << "p2 = " << p2 << " Type - " << typeid(p2).name() << endl;
    if (typeid(p1) == typeid(p2))
        cout << "Types are equal\n";
    else cout << "Types aren't equal\n"
}

int main() {
    t(6, 6.2);
    f(6, 6);
    return 0;
}

/*
Вывод:
p1 = 6   Type - int
p2 = 6.2 Type - double
Types aren't equal
p1 = 6   Type - int
p2 = 6   Type - int
Types are equal
*/
```

Пример 2.

Создадим функции для нахождения минимального и максимального значения заданного целого типа.

```
#include <iostream>
#include <typeinfo>
#include <string.h>
using namespace std;
template <class T>
T getMaxValue(T c) {
    if (strstr(typeid(c).name(), "unsigned"))
        return ~(T)0;
    return ~((T)1 << sizeof(T) * 8 - 1);
}
template <class T>
T getMinValue(T c) {
    if (strstr(typeid(c).name(), "unsigned"))
        return 0;
    return ((T)1 << (sizeof(T) * 8 - 1));
}
int main() {
    unsigned long long c = 0;
    long long d = 0;
```

```
cout << "Type is " << typeid(c).name() << "; max = " << getMaxValue(c) << "; min = " << getMinValue(c) << endl;  
cout << "Type is " << typeid(d).name() << "; max = " << getMaxValue(d) << "; min = " << getMinValue(d) << endl;  
return 0;  
}
```

Первое: у нас есть два типа целых: знаковое и беззнаковое. Первый if проверяет тип на беззнаковость. Если беззнаковое – то максимумом будет число где все единички, а у знакового – ноль и все единицы. Минимум беззнакового – 0, минимум знакового – единица и все нули.

Функции-члены с модификаторами `const` и `volatile`.

Любая попытка модифицировать константные объекты из программы приводит к ошибке компиляции. Однако, инстанция (объект) класса, как правило, не модифицируется программой напрямую. Вместо этого вызывается тот или иной публичный метод. Чтобы не было покушений на константность объекта, компилятор должен различать безопасные (не изменяющие объект) и небезопасные (которые пытаются это сделать) функции-члены. Программист может указать безопасные методы, добавив после круглых скобок спецификатор `const`.

Например:

```
class x {
public:
    char get() const;
    void set(char);
};

char x::get() const { /* == */ }
```

Если объект объявлен со спецификатором `const`, то для него могут быть вызваны методы только с этим спецификатором.

```
const X x;
char c = x.get(); // успех
x.set('1'); // ошибка
```

Наличие спецификатора `const` не предотвращает все возможные изменения. Если класс содержит указатели, то адресуемые ими объекты могут быть изменены константной функцией-членом.

```
void Text::bad(const string& p) const {
    _text = p.c_str(); // error
    for (int i = 0; i < p.size(); i++)
        _text[i] = p[i]; // успех
}
```

Константную функцию можно перегружать не константной функцией с тем же списком параметров.

```
class x {
public:
    char get() const;
    char get();
    void set(char);
};

int main() {
    const X cx;
    X x;
    char c = cx.get();
    char c2 = x.get();
}
```

Хотя конструкторы и деструкторы не являются константными функциями-членами, они вызываются для константных объектов. Объект становится константным после вызова конструктора, и перестаёт быть таким перед вызовом деструктора. Функцию-член класса также можно объявить со спецификатором `volatile`. Объект объявляется с этим спецификатором, если его значение изменяется способом, который не обнаруживается компилятором. Например, данные объекта связаны с портом ввода-вывода. Для таких объектов вызываются только функции-члены со спецификатором `volatile`, конструктор и деструктор.

Объявление mutable.

Чтобы разрешить модификацию члена класса, принадлежащего константному объекту, можно объявить его изменчивым. Данное поле можно обновлять в том числе и функцией-членом со спецификатором `const`.

```
class x {
    mutable int a;
    int b;
public:
    void set(int x) const {
        a = x; // успех
        b = x; // ошибка
    }
};
```

```
class X {
    static int t;
public:
    int get() { return t; }
    void set(int iner) { t += iner; }
};

int X::t = 0;
```

Методы `get()` и `set()` обращаются к глобальному статическому члену класса `t`.

```
X x1, x2;
x1.set(1);
x2.set(1);
cout << x1.get() << " " << x2.get() << endl; // 2 2
```

Совершенно неважно, для какого объекта вызываются `set()` и `get()`, поэтому лучше объявить такие функции-члены как статические.

```
class X {
    static int t;
public:
    static int get() { return t; }
    static void set(int iner) { t += iner; }
};
```

При определении вне тела класса статического метода спецификатор `static` не указывается. Для статических методов недопустимы спецификаторы `const` и `volatile`. В статический метод указатель `this` не передаётся, поэтому непосредственное обращение к нестатическому члену класса запрещено.

```
X x1, x2;
X::set(2);
cout << x1.get() << " " << x2.get() << endl; // 2 2
```

Не стоит путать статические типы в C++ и в Java.

Вложенные классы.

Класс, объявленный внутри другого класса называют вложенным. Он является сочленом объемлющего класса и его объявление может находиться в любой из секций тела класса.

```
class List {
public:
    class ListItem {
        friend class List;
        ListItem(int val = 0);
        ListItem* next;
        int value;
        // ==
    };
private:
    ListItem* list;
    ListItem* at_end;
    // ==
};
```

У объемлющего класса нет прав доступа к закрытым членам вложенного класса, поэтому класс List был сделан дружественным. Аналогично, вложенный класс не может иметь доступ к закрытой части объемлющего класса. Тип `ListItem` объявлен в открытой части класса `List`, поэтому может быть использован в любом месте кода программы. Для этого потребуется квалификация: `List::ListItem::`, поэтому `ListItem` лучше объявить в закрытой части класса `List`.

```
class List {
public:
    /*
    ==
    */
private:
    class ListItem {
    public:
        ListItem(int val = 0);
        ListItem* next;
        int value;
        // ==
    };
    ListItem* list;
    ListItem* at_end;
    // ==
};
```

Данный конструктор также определяется с помощью квалификации.

```
List::ListItem::ListItem(int val) {
    value = val;
    next = 0;
}
```

Вложенный класс можно определять и вне тела объемлющего.

```
class List {
public:
    /*
    ==
    */
private:
    class ListItem;
    ListItem* list;
    ListItem* at_end;
```

```
        // ==  
};  
  
class List::ListItem {  
    public:  
        ListItem(int val = 0);  
        ListItem* next;  
        int value;  
        // ==  
}  
};
```

И последнее запишем: пока компилятор не увидел определение вложенного класса, разрешается объявлять лишь указатели и ссылки на него.

Библиотека STL – Standard Template Library.

Преобразование типов в стиле C++

В **C** явное преобразование типов осуществляется путём указания имени типа в круглых скобках перед преобразуемым выражением [double d = (double) 1013;]. В **C++** добавлены дополнительные операторы приведения типа. В результате, во-первых, появились дополнительные возможности, во-вторых, появился контроль над преобразованием. Добавлено 4 оператора преобразования:

- `const_cast<тип>(объект)` – используется для явного переопределения модификаторов `const` или `volatile`. То есть, данный оператор может снять данный модификатор.

```
void test_pow(const int* v) {  
    int* temp;  
    temp = const_cast<int*>(v);  
    *temp = *v * *v;  
}
```

```
int main() {  
    int x = 10;  
    test_pow(&x);  
    cout << x << endl; // 100  
    return 0;  
}
```

- `dynamic_cast<тип>(объект)` – проверяет законность выполнения заданной операции приведения типа. Если такую операцию выполнить нельзя, то выражение устанавливается равным нулю. Данный оператор в основном используется для полиморфных типов.

Примера не будет.

- `reinterpret_cast<тип>(объект)` – приводит один тип в совершенно другой, например, целый в указатель, или наоборот.

```
int main() {  
    int x;  
    char str[] = "This is the string!";  
    cout << str << endl; // This is the string  
    x = reinterpret_cast<int>(str);  
    cout << x << endl; // 4286208  
    return 0;  
}
```

- `static_cast<тип>(объект)` – выполняет не полиморфное преобразование типов и является аналогом преобразования в **C**.

```
double d = static_cast<double>(10)13;
```

Библиотека Boost.

Boost – собрание библиотек, расширяющих C++. Свободно распространяются вместе с исходным кодом. Проект был создан после принятия стандарта C++, когда многие были недовольны не включением в стандарт некоторых библиотек. Часть библиотек из проекта Boost являются кандидатами на включение в очередные стандарты C++.

Библиотеки Boost охватывают следующее: алгоритмы, обход ошибок компилятора, несоответствующих стандарту, многопоточное программирование, контейнеры, структуры данных, функциональные объекты, обобщённое программирование, графы, ввод-вывод, итераторы, математические и числовые операторы, работа с памятью, синтаксический и лексический разбор, мета-программирование, обработка строк и текста, и др.

Рассмотрим умножение матрицы на вектор.

```
#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/matrix.hpp>
#include <boost/numeric/ublas/io.hpp>
using namespace boost::numeric::ublas;
int main() {
    vector<double> x(2);
    matrix<double> A(2, 2);
    A(0, 0) = 0;    A(0, 1) = 1;
    A(1, 0) = 2;    A(1, 1) = 3;
    vector<double> y = prod(A, x);
    std::cout << y << std::endl;
    return 0;
}
```

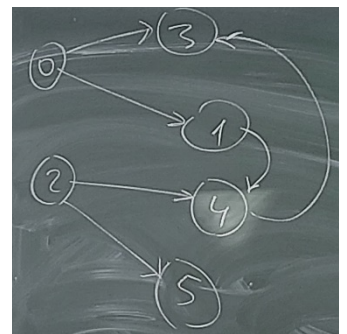
Boost предоставляет различные генераторы псевдослучайных чисел, для каждого из которых можно задать конкретное распределение.

```
#include <boost/random.hpp>
#include <ctime>
using namespace boost;
using namespace std;

mt19937 rnd; // rnd - генератор случайных чисел, их есть несколько типов
double SampleNormal(double mean, double sigma) { // mean - математическое ожидание; sigma -
    normal_distribution<double> norm_dist(mean, sigma);
    variate_generator<mt19937, normal_distribution<double>> normal_sample2(rnd, norm_dist); // привязка
    // распределения к генератору
    return normal_sample2();
}

int main() {
    rnd.seed(unsigned int) time(0));
    for (int i = 0; i < 10; i++)
        cout << SampleNormal(0, 1) << endl;
    return 0;
}
```

Пример 3. Алгоритмы на графах. Boost предоставляет реализацию концепции графа в виде нескольких представлений графа и большого числа алгоритмов. Приведём пример выполнения топологической сортировки. Бывают случаи, когда отношения упорядоченности заданы не для всех элементов множества, а только для некоторых пар, например – университетская программа. Для некоторых дисциплин необходимо, чтобы определённые дисциплины были уже вычитаны, а некоторые дисциплины не зависят друг от друга. Например, если возьмём всю программу 1 курса, то для программирования необходимы некоторые главы информатики и математики. Топологическая сортировка означает чтение дисциплин в таком порядке, чтобы все необходимые дисциплины были прочитаны



к началу чтения каждой дисциплины. Множества, где установлен необходимый порядок, можно отобразить в виде графа:

```
#include <iostream>
#include <list>
#include <algorithm>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/topological_sort.hpp>
#include <iterator>
#include <utility>
using namespace boost;
using namespace std;

int main() {
    typedef adjacency_list<vecS, vecS, directedS, property<vertex_color_t, default_color_type>> Graph;
    // создаётся тип графа на основе типов инценденции

    typedef graph_traits<Graph>::vertex_descriptor vertex;
    // тип вершин

    typedef vector<vertex> container;
    // контейнер для вершин

    typedef pair<size_t, size_t> Pair;
    // тип для дуг графа

    Pair edges[6]{ Pair(0, 1), Pair(2, 4), Pair(2,5), Pair(0, 3), Pair(1, 4), Pair(4,3) };

    Graph G(edges, edges + 6, 6);

    property_map< Graph, vertex_index_t>::type id = get(vertex_index, G);
    // словарь для быстрого поиска вершин

    container c;
    // контейнер для хранения результата - отсортированных вершин.

    topological_sort(G, back_inserter(c));
    cout << "TopSort result : ";
    for (container::reverse_iterator ii = c.rbegin(); ii != c.rend(); ii++)
        cout << id[*ii] << " ";
    cout << endl;
    return 0;
}

/*
TopSort result: 2 5 0 1 4 3
*/
```

Стандарт C++ 11.

В 21 веке вышло несколько стандартов C++, они все добавляют новые вещи, один из самых полных – 11 года. Общий принцип: есть программисты, принимающие новые стандарты с удовольствием. Появляются новые возможности, но код становится менее читаемым.

auto.

Для переменной можно указать тип auto. В этом случае тип переменной определяется компилятором по типу инициатора.

```
auto a = 3.14; // double
auto b = 1; // int
auto& c = b; // int&
auto g = new auto(123); // int*
const auto h = 1; // const int
auto l = 1, m = true, n = 1.161; // error
auto o; // error
vector<int> v; //
auto it = v.begin(); // std::vector<int>::iterator
```

decltype.

Оператор, возвращающий переданный ему тип переданного ему итератора.

```
template <class X, class Y>
auto add(X x, Y y) -> decltype(x + y) {
    return x + y;
}
```

Lambda-выражения.

Lambda – безымянные объекты функций, способные захватывать переменные в области видимости. У них есть свой список захвата – **capture list**. Список захвата пишется в квадратных скобках. [] – ничего не захватывает; [=] – захват локальных объектов по значению; [&] – захват локальных объектов по ссылке; [a, &b] – захват объекта a – по значению, b – по ссылке.

```
int x = 1;
auto getX = [=] { return x; }; // захват переменной по значению
getX(); // 1
auto addX = [=](int y) { return x + y; }; // на самом деле x при этом не меняется, но функция передаёт 2;
addX(1); // 2
auto getXRet = [&]() -> int& { return x; }; // захват по ссылке и изменение переменной
getXRet() = 7; // x == 7
auto f1 = [&x] { x = 3; }; // можно было оставить void, переменная меняется по ссылке
f1(); // x == 3
```

Псевдонимы шаблонов.

```
template <typename T>
using Vec = std::vector<T>;
Vec<int> v; // std::vector<int>
```

Const expr.

Выражения, вычисляемые во время компиляции.

```
constexpr int sqr(int x) { return x * x; }
// -----
int a = sqr(2); // аргумент - только константа / переменная с модификатором const
```

Преобразование конструкторов.

Преобразует фигурные скобки в аргументы конструктора

```
X x(0); => X x{ 0 };
```

Делегирование конструкторов.

Конструкторы могут вызывать другие конструкторы в том же классе с помощью списка инициализаторов.

```
class Foo {
    int foo;
public:
    Foo(int f) : foo(f) { }
    Foo() : Foo(0) { }
};
// -----
Foo foo{};
```

Такая пара конструкторов работает как с параметром, так и без него. Достаточно часто вызывают свои конструкторы, а ещё чаще – конструкторы родительских классов (при наличии иерархии).

Удалённые функции.

Позволяет запретить определённые операции с объектами, например – предотвратим копирование объекта.

```
class A {
    int x;
public:
    A(int x) : X(x) { }
    A(const A&) = delete;
    A operator = (const A&) = delete;
};

A x(123);
A y = x; // ошибка, конструктор копирования удалён
A z(321);
z = x; // ошибка, оператор "=" удалён.
```

Std::array.

Контейнер, построенный поверх Си-подобных массивов.

```
#include <array>
std::array<int, 3> a = { 2, 1, 3 };
std::sort(v.begin(), v.end());
```

Цикл по коллекциям.

Позволяют удобно перебирать элементы контейнера. Следует обратить внимание на наличие ссылки. Без неё изменение не произойдёт. Такие циклы называются «For Each» – «Для каждого».

```
std::array<int, 5> a{ 1, 2, 3, 4, 5 };
for (int& x : a) x *= 2; // {2, 4, 6, 8, 10}
```

Std::chrono.

Библиотека **chrono** содержит набор служебных функций и типов, которые имеют дело с длительностью, временем и временными точками. Например, замерим длительность выполнения некоторого участка кода.

```
std::chrono::time_point<std::chrono::steady_clock>start, end;
start = std::chrono::steady_clock::now();
// участок кода
end = std::chrono::steady_clock::now();
std::chrono::duration<double>elapsed_second = end - start;
std::cout << elapsed_second.count();
```

Std::to_string.

Переводит в строку

```
std::to_string(1.2) -> "1.2"
```

Умные указатели.

`std::unique_ptr`

`std::shared_ptr`

`std::weak_ptr`

Объект класса `Unique_ptr` является перемещаемым, но не копируемым. После перемещения (ровно как и при создании конструктором по умолчанию) `unique_ptr` содержит `nullptr` – пустой указатель.

```
#include <memory>
using namespace std;
//-----
class Foo {
public:
    void bar() { cout << "Bar\n"; }
    ~Foo() { cout << "~Foo\n"; }
};

int main(){           /* какой-то блок кода */

    std::unique_ptr<Foo> p1{ new Foo{} }; // p1 владеет объектом класса Foo
    p1->bar();
    std::unique_ptr<Foo> p2{ std::move(p1) }; // теперь p2 владеет объектом класса Foo
    p2->bar();
    if (p1) p1->bar(); // false - p1 является nullptr
    p1 = std::move(p2); // p1 - собственник Foo, p2 - уничтожается
    p1->bar();
    return 0;
}
// экземпляр Foo автоматически уничтожается когда p1 выходит из области видимости.
```

Второй тип указателей. `Shared_ptr` выступает в противовес `unique_ptr` и является разделяемым. Это означает что указатель может быть скопирован, и каждая его копия может обращаться к одному и тому же указателю. При разрушении объекта `shared_ptr`, ресурс, хранящийся в нём будет освобождён тогда и только тогда, когда не существует других объектов типа `shared_ptr`, ссылающихся на этот ресурс.

```
shared_ptr<Foo> p1{ new Foo };
{
    auto p2 = p1;
    cout << p1.use_count() << " " << p2.use_count() << endl;
    // use_count показывает количество ссылок на объект
}
cout << p1.use_count() << endl;
return 0;
```

`std::weak_ptr` не является самостоятельным умным указателем, а является лишь дополнением к указателю **`shared_ptr`**. **`Weak_ptr`** конструируется из существующего **`weak_ptr`** и его единственной целью является предоставление пользователю информации об ассоциированных с ним `shared_ptr`-указателях.

```
/* какой-то блок кода */

shared_ptr<Foo> p1{ new Foo };
weak_ptr<Foo> p2(p1);
cout << p2.expired() /* проверяет, обнулён ли счётчик ссылок */ <<
    " " << p2.use_count() << endl; // 0 1
auto p3 = p2.lock();
cout << p2.expired() << " " << p2.use_count() << endl; // 0 2
p1.reset();
p3.reset(); // ~Foo
```

```
    cout << p2.expired() << " " << p2.use_count() << endl;  
    // 1 0  
    return 0;  
}
```

Пространства имён (namespace).

Определение пространства имён.

По умолчанию, любой объект/функция/тип/шаблон, объявленный в глобальной области видимости, также называемой областью видимости глобального пространства имён вводит понятие глобальной сущности. Каждая такая сущность должна иметь уникальное имя, например, функция и объект не могут быть одноимёнными, даже если они объявлены в разных файлах.

```
#include <iostream>
```

```
class cplusplus_primer_matrix { /* ... */ };
void inverse(cplusplus_primer_matrix&);
namespace cplusplus_primer {
    class matrix { /* ... */ };
    // ...
    void inverse(matrix&);
}
```

С точки зрения пользователя, мы можем произвести декомпозицию – разбиение пространства имён на блоки, например, интерфейсный блок и блок реализации. Каждое пространство имён имеет свои области видимости.

В разных пользовательских пространствах могут встречаться члены с одинаковыми именами.

Если мы хотим обратиться к члену пространства имён, то необходима конструкция `имя_пространства::имя_члена` (`namespace_name::member_name`). Но такое обращение не всегда удобно. Поэтому существуют механизмы, позволяющие облегчить использование пространств имён в программах. Это: псевдонимы пространств имён (`typedef`), использование `using`-объявлений и `using`-директив.

Определение пространства имён.

```
namespace cplusplus_primer {
    class matrix { /* ... */ };
    const double pi = 3.1416;
}

namespace cplusplus_primer {
    void inverse(matrix&);
    matrix operator+(const matrix& m1, const matrix& m2)
    { /* ... */ };
}
```

Определение пространства имён не обязательно должно быть непрерывным, т.е. можно выполнить декомпозицию определения пространства имён на несколько определений.

Возможность декомпозиции определения пространства имён является целесообразной в большей степени при организации исходного кода пользовательской библиотеки.

```
// primer.h
namespace cplusplus_primer {
    class matrix { /* ... */ };
    const double pi = 3.1416;
    matrix operator+(const matrix& m1, const matrix& m2);
    void inverse(matrix&);
}
```

```
// primer.cpp
#include "primer.h"

namespace cplusplus_primer {
    void inverse(matrix& m) { /* ... */ }
```

```

        matrix operator+(const matrix& m1, const matrix& m2) { /* ... */ }
    }

// main.cpp
#include "primer.h"

void func(cplusplus_primer::matrix& m) {
    cplusplus_primer::inverse(m);
}

int main() { /* ... */ }

```

Вложенные пространства имён.

```

namespace cplusplus_primer {
    // первое вложенное пространство имён
    // матричная часть библиотеки
    namespace MatrixLib {
        class matrix { /* ... */ };
        const double pi = 3.1416;
        matrix operator+(const matrix& m1, const matrix& m2);
        void inverse(matrix&);
        //-----
    }
    // второе вложенное пространство имён
    // зоологическая (каво?) часть библиотеки
    namespace AnimalLib {
        class ZooAnimal { /* ... */ };
        class Bear : public ZooAnimal { /* ... */ }; // медведь, не пиво
        class Raccon: public Bear { /* ... */ }; // енот, но это не точно
        //-----
    }
}

```

Пространство имён `cplusplus_primer` содержит в себе два пространства – `MatrixLib` и `AnimalLib`. С точки зрения удобства и читабельности

```
cplusplus_primer::MatrixLib::matrix;
```

Безымянные пространства имён.

Иногда возникает необходимость сделать так, чтобы некая глобальная сущность была видима только на небольшом участке программы, например, у нас есть библиотека алгоритмов сортировки, и все эти алгоритмы используют функцию swap для обмена местами элементов массива.

```
// sortlib.cpp

void swap(Item&, Item&);
void sortInsert(Item*, int);
void sortHeap(Item*, int);
void sortBubble(Item*, int);
```

Для того, чтобы функцию swap убрать из глобальной области видимости, можно использовать безымянное пространство имён.

```
namespace{
    void swap(Item& a, Item& b) {
        Item tmp = a;
        a = b;
        b = tmp;
    }
}
```

В результате имя swap может использоваться только внутри файла `sortlib.cpp` без использования квалификации. В других файлах данный swap не виден и могут использоваться одноимённые сущности. Для swap мы можем обойтись средствами C – static.

Использование членов пространства имён.

Использование квалификации при каждом обращении к членам пространства имён достаточно применительно. Существуют механизмы, облегчающие использование членов пространств имён: псевдонимы пространств имён, using-объявления и using-директивы

Псевдонимы пространства имён.

Псевдоним пространства имён используется для задания короткого имени пространства.

```
namespace mlib = cplusplus_primer::MatrixLib;
void f(mlib::matrix m) {}
```

Синтаксически допускается использование одного псевдонима для нескольких пространств имён.

Using-объявления.

Using-объявления позволяют обращаться к пространствам имён без квалификации. Using-объявление вводит пространство в ту область, где оно использовано.

```
void f(matrix m){ ... }
```

Имя, вводимое в using-объявлении аналогично введению любой другой сущности.

- 1 – оно должно быть уникально в данной области видимости;
- 2 – оно скрывает одноимённую сущность из внешней области;
- 3 – оно скрывается объявлением одноимённой сущности во вложенной области.

```
namespace blip {
    int bi = 16, bj = 15, bk = 23;
}
int bj = 0;
void f() {
```

```
using blip::bi;
++bi; // 17
using blip::bj;
++bj; // 16
int bk;
using blip::bk; // ошибка. двойная инициализация переменной
}
int wrongInit = bi; // ошибка. bi вне области видимости
```

Using-директива.

В старых версиях C++ пространств имён ещё не существовало. Для подключения новых версий библиотек к старым программам был создан механизм, который вводит все члены пространств имён в указанную область видимости. Этот механизм называется using-директивой. По итогу, она просто уничтожает пространства имён. Все компоненты стандартных библиотек C++ находятся в пространстве имён std.