1. Собственные типы данных и инструкции

1.1. На машине, поддерживающей кодовую таблицу ASCII, напишите программу с использованием цикла **for**, которая печатала бы по очереди все прописные и строчные буквы.

1.2. Переделайте программу

```
#include <iostream>
#include <assert.h>
using namespace std;
int gcd(int m, int n)
{ int r;
while((r=m%n)) m=n,n=r;
return n;
}
int main()
{ int x,y,g;
cout<<"\nПрограмма gcd на C++\n";
do
 { cout<<"\nВведите два целых числа (окончание x=y):";
 cin>>x>>y; assert(x*y!=0);
 cout<<"\nGCD("<<x<< ", "<<y<<") = "<<(g=gcd(x,y))<<endl;
 assert(x%g==0&&y%g==0);
 } while(x!=y);
return 0;
}
```

так, чтобы в ней использовалась **fstream.h**. Программа должна получать аргументы из командной строки: **gcd gcd.dat gcd.ans**

1.3. Проверьте следующие преобразования и попытайтесь определить, что происходит в каждом случае:

```
int i=3,*j=&i;
bool flag=true;
double x=1.5;
```

Используйте старые приведения и запишите каждое из значений как **(int)** и как **(double)**. Посмотрите, меняется ли что-нибудь, если использовать **static_cast<>.**

2. Функции и указатели

2.1. C++ позволяет передавать функции main () аргументы командной строки. Следующий код выводит свои аргументы командной строки:

//Вывод аргументов командной строки начиная с самого правого

```
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{ for (--argc; argc >= 0; --argc)
cout<<argv[argc]<<endl;
return 0;
}
```

Аргументу **argc** передается количество аргументов командной строки. Каждый аргумент является строкой, помещенной в двухмерный массив **argv**. Откомпилируйте программу в исполняемый файл с именем **echojxrg**. Запустите его со следующими аргументами:

```
echo_arg a man a plan a canal panama
```

Модифицируйте так, чтобы она выводила аргументы ; командной строки слева направо и нумеровала их.

2.2. Напишите обращающую строку программу, используя память, распределенную с помощью **new**. В результате строка **s1** должна содержать обращение строки. Используйте **new** для размещения строки **s1** длиной в **strlen(s2)+1** достаточно). Обращение строки реализуйте через функцию

```
char* strrev(char*& s1, const char* s2);
```

2.3. Напишите программу, которая размещает в свободной памяти одномерный массив. Его нижняя и верхняя границы задаются пользователем. Программа должна убедиться, что верхняя граница больше нижней. Если это не так, осуществляется аварийный выход. Используйте пакет **assert.h**:

```
#include <assert.h>
assert(ub-lb>0); //ввод нижней и верхней границ
```

Размер массива будет (верхняя граница–нижняя граница+1) элементов. Дан стандартный массив C++ именно такой длины. Напишите функцию, которая использует данный стандартный массив для инициализации динамического массива. Проверьте её, выведя в красиво отформатированном виде оба массива до и после инициализации.

3. Реализация АТД в базовом языке

3.1.1 Создайте структуру C++ для молочных продуктов (**dairy**), включающую название, вес порции, энергетическую ценность (в калориях), содержание белков, жиров и углеводов. Двадцать пять грамм сыра содержат **375** калорий, **5** грамм белков, **8** грамм жира и **0** грамм углеводов. Покажите, как присвоить эти значения переменным-членам вашей структуры. Напишите функцию, которая по заданным переменной типа **struct dairy** и весе в граммах (размер порции), возвращала бы число калорий в этой порции.

3.1.2. Используя структуру **card**, описанную в теме «Агрегатный тип struct», напишите процедуру сортировки розданных игроку карт. В карточных играх большинство игроков держит свои карты, отсортировав их по достоинству. Эта процедура должна расположить вначале тузы, затем короли, и так далее до двоек. У игрока пять карт.

3.1.3.Используйте структуру **ch_stack**, описанную в теме «Пример: стек» Напишите функцию

```
void reverse(char s1[],char s2[]);
```

Строки **s1** и **s2** должны быть одинакового размера. Строка **s2** должна стать обращённой копией строки **s1**. Внутри **reverse** используйте **ch_stack** для выполнения обращения.

3.1.4. Напишите для типа **ch_stack**;

```
//поместить n символов из s1[] в ch_stack stk
void pushm(int n,const char s1[],ch_stack stk);
//извлечь n символов из stk в s1[]
void popn(int n,char s1[],const ch_stack stk);
```

3.2.1. Добавьте в пакет комплексных чисел (см. тему «Комплексные числа») процедуру вычитания **complex sub(complex& a,complex& b)** проверьте её.

3.2.2. Добавьте в пакет комплексных чисел (см. тему «Комплексные числа») процедуру умножения **complex mul(complex& a,complex& b)** проверьте её.

3.2.3. Добавьте в пакет комплексных чисел (см. тему «Комплексные числа») процедуру деления **complex div(complex& a,complex& b)** проверьте её.

3.3.1. Добавьте процедуру деления, возвращающую **complex**, когда ей передается **double**. Используйте её для программирования **complex div(complex,double)** и **complex div (double, complex)**. Заметьте, как здесь перегружается функция **div()**, что обеспечивает полезную интеграцию при смешивании двух типов **double** и **complex**. Проверить работоспособность функций.

3.3.2. Добавьте процедуру умножения, возвращающую **complex**, когда ей передается **double**. Используйте её для программирования **complex mul(complex,double)** и **complex mul(double, complex)**. Заметьте, как здесь перегружается функция **mul()**, что обеспечивает полезную интеграцию при смешивании двух типов **double** и **complex**. Проверить работоспособность функций.

3.3.3. Добавьте процедуру сложения, возвращающую **complex**, когда ей передается **double**. Используйте её для программирования **complex add(complex,double)** и **complex add(double, complex)**. Заметьте, как здесь перегружается функция **add ()**, что обеспечивает полезную интеграцию при смешивании двух типов **double** и **complex**. Проверить работоспособность функций.

3.4.1. Мы хотим определить структуру **ch_deque** (Double Ended Queue, deque - двусторонняя очередь.) для реализации двусторонней очереди. Двусторонняя очередь допускает помещение в оба конца (**push**) и извлечение с обоих концов (**pop**). Это обычная форма контейнера.

```
struct ch_deque { char s[max_len]; int bottom,top; };
void reset(ch_deque* deq)
{
deq->top=deq->bottom=max_len/2; deq->top--;
}
```

Объявите и реализуйте **push_t()**, **pop_t ()**, **push_b()**, **pop_b()**, **out_stack()**, **top_of()**, **bottom_of()**, **empty()** и **full()**. Функция **push_t()** выполняет помещение наверх (**push on top**). Функция **push_b()** отвечает за помещение вниз (**push on bottom**). Функция **out_stack ()** должна выводить всю очередь сверху донизу. Функции **pop_t ()** и **pop_b()** соответствуют извлечению сверху (pop from top) и извлечению снизу (pop from bottom). Если вершина ниже основания, то это означает пустую (**empty**) очередь. Проверьте каждую функцию.

3.4.2. Расширьте тип данных **ch_deque** добавлением функции **relocate()**. Если **ch_deque** заполнена, то вызывается **relocate()**, и содержимое **ch_deque** перемещается в пустую область памяти, выравненную относительно центра **max_len/2** массива **s**. Вот заголовок объявления этой функции;

```
//Возвращает true, если завершается успешно, false — если нет
bool relocate(ch_deque* deq)
```

3.4.3. Напишите функцию, которая меняет местами содержимое двух строк. Если вы поместите символьную строку в стек, а затем извлечете её, она станет обращённой. Но при обмене строками нам нужен неизменный порядок символов. Используйте двустороннюю очередь **deque** для выполнения такого обмена. Строки будут храниться в двух символьных массивах одинакового размера, но сами строки могут быть разной длины. Вот прототип этой функции:

```
void swap(char sl[],char s2[]);
```

3.4.4. Напишите списочную операцию для подсчета элементов списка. Подразумевается рекурсия до конца списка с выходом после обнаружения указателя **NULL**. Если список оказывается пустым, возвращается значение **0**, в противном случае возвращается число элементов в списке.

```
int count(slist* head);
```

3.4.5. Напишите функцию **lookup()**, которая ищет в списке конкретный элемент **c**. Если элемент найден, возвращается указатель на этот элемент, в противном случае возвращается указатель на **NULL**.

```
slist* lookup(char c,slist* head);
```

4. Сокрытие данных и функции-члены класса

4.1.1 Разработайте класс **person**, содержащий члены для хранения имени, возраста, пола, телефона. Напишите функции-члены, которые могут изменять эти члены данных по отдельности. Напишите функцию-член **person::print()**, которая печатала бы красиво отформатированные данные о человеке.

4.1.2. Используйте структуру **ch_stack**, описанную в теме «Доступ: закрытый и открытый». Напишите функцию

```
void reverse(const char s1[],char s2[]);
```

Строки **s1** и **s2** должны быть одинакового размера. Строка **s2** должна превращаться в обращённую копию строки **s1**. Внутри **reverse** используйте **ch_stack** для выполнения обращения.

4.1.3. Для типа **ch_stack** из темы «Доступ: закрытый и открытый», запишите как функции-члены:

```
//Помещение n символов из s1 в ch_stack
void push(int n,const char s1[]);
//Извлечение n символов из ch_stack в символьную строку
void pop(int n,char s1[]);
```

Подсказка: Не забудьте вставить символ конца строки перед тем, как её выводить.

4.2. Представьте как класс код **deque** (двусторонняя очередь, допускающая вход и выход с обоих концов)

```
class deque
{ public:
   void reset();
  private:
   char c[max_len];
   int bottom,top;
};
void reset(deque* deq)
{
deq->top=deq->bottom=max_len/2;
deq->top--;
}
```

Объявите и реализуйте функции **push_t**, **pop_t**, **push_b**, **pop_b**, **out_stack**, **top_of**, **bottom_of**, **empty** и **full**. Функция **push_t** служит для помещения элемента в начало очереди; функция **push_b** — в конец. Функция **out_stack** должна выводить всю очередь от начала до конца. Функции **pop_t** и **pop_b** отвечают за извлечение элементов из начала и конца очереди соответственно. Ситуация, когда начало очереди опускается ниже конца, означает пустую очередь. Протестируйте каждую функцию.

4.3. Расширьте тип данных **deque**, добавив функцию-член **relocate**. Если очередь заполнена целиком, вызывается функция **relocate**, и содержимое **deque** перемещается в пустую область памяти, выровненную относительно центра **max_len/2** массива **s**. Прототип этой функции должен выглядеть так:

```
//Возвращает true, если завершается успешно,
//false — если нет
bool deque::relocate()
```

4.4.1. Напишите функцию, которая меняет содержимое двух строк. Если вы поместите символьную строку в **ch_stack**, а затем извлечете её, она станет обращённой. Но при обмене строками нам нужен неизменный порядок символов. Используйте **deque** для выполнения такого обмена. Строки будут храниться в символьных массивах одинакового размера, но сами строки могут быть разной длины. Вот прототип этой функции:

```
void swap(char s1[],char s2[]);
```

4.4.2. Допишите функции-члены:

```
complex complex::plus(complex a, complex b);
                    //выполняет бинарное сложение c=plus(a,b)
complex complex::mpy(complex a, complex b)
                    //выполняет бинарное умножение c=mpy(a,b)
```

4.4.3. Перепишите класс **twod** из темы «Контейнеры и доступ к их содержимому» так, чтобы он мог содержать комплексные значения. Проверьте его, инициализовав этот контейнер набором комплексных чисел, и произведите контрольный вывод. Затем выполните транспонирование и выведите результат.

4.4.4. Напишите класс **chess_piece** (шахматная фигура), который описывал бы положение фигуры на шахматной доске. Шахматная доска имеет вертикали от **a** до **h** и горизонтали от **1** до **8**. Вначале белый король стоит на **e1**, а черный — на **e8**. Проверьте ваш класс, введя начальные позиции для всех **32** шахматных фигур и напечатав содержимое доски.

4.4.5. Перепишите вложенный цикл функции **transpose()** из темы «Контейнеры и доступ к их содержимому», заменив где это возможно вызовы функции **twod;:element_lval()** на вызов функции **twod::element_rval()**. Затем перепишите этот цикл, заменив три оператора присваивания на обращение к

```
inline void swap(doubled, double&);
```

Проверьте все три версии.

4.4.6. Напишите функцию-член для умножения матриц, используя переменные **twod**.

```
twod twod::mmpy(const twod& a,const twod& b);
```

5. Создание и уничтожение объектов

5.1.1. Напишите функцию-член для класса **mod_int**:

```
void add(int i) ; //добавление i к v по модулю 60
```

5.1.2. Используйте тип **ch_stack**, из темы «Создание динамического стека», и включите в него конструктор по умолчанию для размещения **ch_stack** из **100** элементов. Напишите программу, которая обменивает содержимым два стека **ch_stack**; используйте массив стеков **ch_stack** для выполнения этой работы. Обмениваемые стеки будут двумя первыми стеками **ch_stack** в массиве. Один из способов может состоять в использовании четырёх **ch_stack** — **st[0]**, **st[1]**, **st[2]** и **st[3]**. Поместите содержимое **st[1]** в **st[2]**, **st[0]** в **st[3]**, **st[3]** в **st[1]**, а **st[2]** в **st[0]**. Реализовав функции печати, которые выводили бы все элементы **ch_stack**, убедитесь, что после обмена содержимым порядок элементов в стеках не изменился. Можно ли решить задачу с помощью только трёх **ch_stack**?

5.1.3. Добавьте к типу **ch_stack** конструктор со следующим прототипом:

```
ch_stack::ch_stack(const char* c);
//инициализация символьным массивом
```

5.2.1. Используйте тип **my_string**, из темы «Пример: динамически размещаемые строки», и напишите следующие функции- члены:

```
//strcmp: отрицательна, если s<s1, равна 0,если s==s1,
     // положительна, если s>s1, здесь s — неявный аргумент
int my_string::strcmp(const my_string& s1)
//strrev обращает my_string
void my_string::strrev();
//перегруженная print для печати первых n символов
void my_string::print(int n);
```

5.2.2. Используя тип **vect** из темы «Класс vect», напишите следующие функции:

```
//складывает значения всех элементов и возвращает их сумму
int vect::sumelem();
//печатает все элементы
void vect::print();
//складывает два вектора в третий v(неявный_аргумент)=v1+v2
void vect::add(const vect& v1,const vect& v2);
//складывает два вектора и возвращает v(неявный_аргумент)+v1
vect vect::add(const vect& v1);
```

5.2.3. Определите класс **multi_v** как:

```
class multi_v
{ public:
multi_v(int i):a(i),b(i),c(i),size(i) {}
void assign(int ind,int i,int j,int k);
void retrieve(int ind,int& i,int& j,int& k) const;
void print(int ind) const;
int ub const { return(size-1); }
private:
vect a,b,c;
int size;
};
```

Напишите и проверьте код для функций-членов **assign()**, **retrieve()** и **print()**. Функция **assign()** должна присваивать **i**, **j** и **k** элементам **a[ind]**, **b[ind]** и **c[ind]**, соответственно. Функция **retrieve()** выполняет обратное по отношению к функции **assign()**. Функция **print()** должна печатать три значения **a[ind]**, **b[ind]** и **c[ind]**.

5.2.4. Используйте тип **slist**, приведённый в теме «Пример: односвязный список», для написания следующих функций-членов:

```
//конструктор slist с инициализатором — массивом символов slist::slist(const char* c);
//length возвращает длину slist
int slist::length();
//возвращает число элементов со значением c
int slist::count_c(char c);
```

Напишите функцию-член **append()**, которая будет добавлять список в конец неявно заданного аргумента-списка, а затем очищать добавленный **slist**, обнуляя голову:

```
void slist::append(slist& e);
```

Напишите функцию-член **copy()**, которая будет копировать список:

```
//неявный аргумент принимает копию e
void slist::copy(const slist& e);
```

Не забудьте уничтожить неявный список перед тем, как делать копию. Вам нужна специальная проверка, чтобы избежать копирования списка в самого себя.

5.2.5. Создайте тип для безопасного трёхмерного массива под названием **v_3_d**.

```
//Реализация безопасного трехмерного массива
class v_3_d
{ public:
   int ub1(),ub2(),ub3();
   v_3_d(int l1,int l2,int l3);
   ~v_3_d();
   int& element(int i,int j,int k) const;
   void print() const;
  private;
   int*** p;
   int si,s2,s3;
};
```

Инициализуйте и выведите трёхмерный массив.

5.2.6. Определить класс C++, который будет похож на множество (set) в Pascal. Приведённое ниже представление является 32-разрядным машинным словом:

```
//Реализация АТД для типа set
const unsigned long int masks[32]=
{
0x80000000, 0x40000000, 0x20000000, 0x10000000,
0x8000000, 0x4000000, 0x2000000, 0x1000000,
0x800000, 0x400000, 0x200000, 0x100000,
0x80000, 0x40000, 0x20000, 0x10000,
0x8000, 0x4000, 0x2000, 0x1000,
0x800, 0x400, 0x200, 0x100,
0x80, 0x40, 0x20, 0x10, 0x8, 0x4, 0x2, 0x1
};
class set
{ public:
   set(unsigned long int i) { t=i; }
   set() { t=0x0; }
   void u_add(int i) { t|=masks[i]; }
   void u_sub(int i) { t&=~masks[i]; }
   bool in(int i) const
    { return bool((t&masks[i])!=0); }
   void pr_mems() const;
   set set_union(const set& v) const
    { return (set(t|v.t)); }
  private:
   unsigned long int t;
};
```

Напишите код **pr_mems** для вывода всех элементов множества. Напишите код для функции-члена **intersection**, возвращающей пересечение множеств.

5.2.7. Дополните пакет обработки полиномов, написав код для процедур **void polynomial::release()** и **void polynomialprint()**, которого нет в тексте (см. раздел 6.9, «Многочлен как связный список», на стр. 174).

Напишите код для процедуры сложения многочленов **void polynomialplus()**.

5.2.8. Усовершенствуйте форму класса **my_string** с подсчётом ссылок, вставив в соответствующие функции-члены проверку утверждения, что **ref_cnt** неотрицателен. Зачем это может понадобиться (см. тему «Строки, использующие семантику ссылок»?

5.2.9. Измените класс **matrix**, чтобы получить конструктор, выполняющий транспонирование (см. тему «Двумерные массивы». Он должен содержать перечислимый тип в качестве второго аргумента, который показывает, какая трансформация должна выполняться с массивом.

```
enum transform {transpose,negative,upper};
matrix::matrix(const matrix& a,transform t)
{
//транспонирование base[i][j]=a.base[j][i]
//смена знака base[ij[j]=-a.base[i][j]
//верхнетреугольная матрица base[i][j]=a.base[i][j]
// i<=j, иначе 0
}
```

6. Ad hoc полиморфизм

6.1.1. Напишите конструктор для класса **rational**, который при заданных двух целых — делимом и частном, использует алгоритм нахождения наибольшего общего делителя и сводит внутреннее представление к наименьшим значениям **a** и **q** (см. тему «Перегрузка и выбор функции»).
Перегрузите операторы равенства и сравнения для класса **rational**. Заметьте, что если два рациональных числа представлены в форме, данной в предыдущем упражнении, они равны тогда и только тогда, когда равны их делимые и частные (см. тему «Перегрузка и выбор функции»).
6.1.2. Напишите функцию, которая складывает вектор **v** с матрицей **m**. Вот прототип, который надо добавить в классы **matrix** и **vect**:

```
friend vect add(const vect& v, matrix& m);
```

Вектор **v** будет поэлементно складываться с каждой строкой матрицы **m** (см. тему «Дружественные функции»).
6.1.3. Определите класс **complex** как

```
class complex { public:
complex(double r) { real = r; imag =0; }
void assign(double r, double i)
{ real = r; imag = i; } void print()
{ cout << real « " + " << imag << "i "; } operator double()
{ return (sqrtfreal * real + imag * imag));} private:
double real, imag;
} ;
```

Дополнить этот класс, перегрузив некоторые операторы. Например, функцию-член **print()** можно заменить, создав дружественную функцию **operators()**:

```
ostream& operator<<(ostream& out,complex x)
{ out<<x.real<<" + "<<x.imag<<"i ";
return out;
}
```

Напишите и проверьте код для оператора унарного минуса. Он должен возвращать **complex**, причём значения действительной и мнимой частей меняют знак.
Для типа **complex** напишите бинарные операторы-функции сложения, умножения и вычитания. Каждая функция должна возвращать значение **complex**. Запишите их как дружественные функции. Почему не следует использовать функции- члены?
Напишите две дружественные функции:

```
friend complex operator*(complex,double);
friend complex operator+(double,complex);
```

6.2.1 Определите новый класс **matrix_iterator** как итерирующий класс, который перебирает по очереди все элементы матрицы (см. тему «Перегрузка оператора () для индексирования»). Используйте его для нахождения наибольшего элемента матрицы.
6.2.2. Переделайте АТД **my_string**, используя перегрузку операторов (см. тему «Пример: динамически размещаемые строки»). Функция-член **assingn** должна быть изменена так, чтобы стать **operator=**. Функция-член **concat** должна стать **operator+**. Кроме того, перегрузите оператор **[]** так, чтобы он возвращал **i**-ый символ из **my_string**. Если такого символа нет, должно возвращаться значение **-1**.
6.2.3. Переделайте АТД списка с помощью перегрузки операторов (см. тему «Пример: односвязный список»). Функция-член **prepend()** должна быть заменена на **operator+()**, а **del()** — на **operator--()**). Кроме того, перегрузите оператор **()** так, чтобы он возвращал **i**-ый элемент списка.
6.2.4. Дополните пакет обработки полиномов функциями **polynomial::release()** и **polynomial::print()**. В качестве примера посмотрите как реализована эта функция в классе **slist** из темы «Пример: односвязный список». Возьмите функцию-член **polynomial::plus()** из темы «Многочлен как связный список» и представьте её в виде кода для перегрузки **operator+**:

```
polynomial operator+(const polynomial&,const polynomial&)
```

Эта функция должна быть дружественной классу **polynomial**.