

1. Улучшения языка C

Замена для препроцессора

Квалификатор `const` позволяет создавать типизированные константы, с ограниченной областью видимости в отличие от констант, определенных `#define`. Задать значение константе можно только при создании.

```
const int size=100;  
int a[size];  
const double pi=3.14159265358979323846;
```

Также этот квалификатор используется для указания, что параметр не меняется внутри функции.

```
char *strcpy(char *dst, const char *src);
```

Функция, определенная с помощью спецификатора `inline` становится встраиваемой, т.е. её код подставляется в точку вызова. При этом устраняются все недостатки, свойственные макроопределениям. Время работы программы уменьшается за счет операций сохранения/восстановления регистров и передачи управления, но размер

программы увеличивается. Спецификатор `inline` носит рекомендательный характер, для больших функций компилятор генерирует обычный вызов. Тело встраиваемой функции определяется в заголовочном файле.

```
inline int max(int x, int y)
{ return x>y?x:y;
}
```

Макроопределение `#define max(x,y) ((x)>(y)?(x):(y))` работает для любых типов данных. Чтобы функция `max` работала так же, её нужно определить как шаблон:

```
template <typename T>
inline T max(T x, T y)
{ return x>y?x:y;
}
```

Ссылки

В языке C реализована только передача параметров по значению. Если потребуется изменение переменной внутри функции, то нужно передать функции адрес переменной, применив операцию `&`, а внутри функции выполнять операцию разыменования для параметра при всех обращениях к этой переменной. В C++ появилась замена для указателей – ссылка, которую можно рассматривать как константный указатель, который всегда разыменован.

Ссылка должна инициализироваться при создании некоторым объектом, адрес которого будет храниться в ссылке. Перенаправить ссылку на другой объект нельзя. Изменение ссылки приводит к изменению объекта.

```
int a=5;  
int &b=a;  
b=6; // a=6
```

Чаще всего ссылки используются в качестве параметров функций.

```
void swap(int &x, int &y)
{ int t;
  t=x;
  x=y;
  y=t;
}
int main()
{ int a=1, b=2;
  swap(a,b);
  // a=2, b=1
}
```

Функция может возвращать ссылку. Результат такой функции можно использовать в качестве левого операнда присваивания. Можно возвращать только ссылку на аргумент функции (или его часть), переданный по ссылке, или на статический объект.

new и delete

Функции `malloc` и `calloc` не обеспечивают соответствия между размером выделенной памяти и типом данных.

```
double *p; // до модификации int *p;
```

```
...
```

```
p=calloc(n,sizeof(int)); // забыл исправить тип, но  
компилятор не заметит этой ошибки
```

Результат `calloc` имеет тип `void *`, который в С может быть преобразован к указателю любого типа без явного вызова операции преобразования, но даже использование явного преобразования результата не гарантирует, что в аргументах не будет ошибки:

```
p=(double *)calloc(n,sizeof(int));
```

Кроме того, многие программисты не заботятся о проверке результата функции на 0, считая, что памяти много.

Для устранения недостатков функций `malloc`, `calloc` и `free` в С++ были введены операции `new` и `delete`.

Операция `new` возвращает *типизированный* указатель, а после выделения памяти вызывается конструктор, который инициализирует память. Если память не может быть выделена, возникает исключительная ситуация `bad_alloc`, т.е. проверять возвращаемое значение на 0 нет необходимости. Допускаются следующие формы для операции `new`:

- `new` **тип**
- `new` **тип** ()
- `new` **тип** [**выражение**]
- `new` **тип** [**выражение**] ()
- `new` **тип** (**список_выражений**)

3-я и 4-я форма используется для создания массива объектов, остальные — для создания одиночных объектов. Различие между 1-й и 2-й, 3-й и 4-й формами проявляется только для стандартных типов данных и классов без конструкторов. В 1-м и 3-м случаях состояние объектов остается неопределенным, во 2-м и 4-м — инициализируется 0. Для классов с конструкторами в первых четырех формах вызывается конструктор по умолчанию. 5-я форма предназначена для инициализации созданного объекта с помощью конструктора с параметрами, а для стандартных типов данных можно указать только одно выражение, значение которого

используется для инициализации созданного объекта. В VC5.5 4-я форма не поддерживается.

Операция `delete` освобождает память, предварительно вызвав деструктор для объектов. Существуют две формы операции `delete`:

- `delete` **указатель**
- `delete[]` **указатель**

1-я форма используется для уничтожения одиночного объекта, 2-я форма – массива объектов.

Примеры:

```
int *p;  
p=new int(10);  
...  
delete p;  
p=new int[10];  
...  
delete[] p;
```

Нельзя смешивать формы `new` и `delete`. Нельзя выделять память с помощью функций `malloc` и `calloc`, а освобождать с помощью `delete`, или выделять с помощью `new`, а освобождать с помощью `free` даже для стандартных типов данных, не требующих вызова конструкторов и деструкторов, так как эти операции могут сохранять дополнительную информацию о выделенной памяти или использовать другую реализацию кучи.

Функции

Функциям, выполняющим одинаковые действия с данными различных типов, в С++ можно дать одинаковые имена. Компилятор сможет определить, какую функцию из набора вызвать, по типу аргументов.

```
void print(int x)
{ printf("%d", x);
}
void print(char x)
{ printf("%c", x);
}
int main()
{
    print(10);
    print('A');
}
```

Чтобы на этапе компоновки программы можно было установить соответствие между функциями и их вызовами в имя функции на языке С++ включаются типы ее параметров. Для функций на языке С типы параметров не важны, поэтому перед объявлением функции из библиотек на языке С нужно

написать `extern "C".`

Обычно один и тот же заголовочный файл используется для программ на C и C++, поэтому в него добавляет следующие строки:

```
#ifdef __cplusplus
extern "C" {
#endif
// объявления функций на C
#ifdef __cplusplus
}
#endif
```

Вместо определения набора функций, отличающихся только количеством параметров, можно указать при объявлении функции в заголовочном файле значения по умолчанию для некоторых параметров. Эти параметры должны быть последними в списке. Если при вызове параметр пропущен, то должны быть пропущены и все параметры за ним. В качестве значений по умолчанию можно использовать константы и глобальные переменные.

```
void fun(int a, int n=100, double eps=1e-9);
...
f(1,200,1e-6);
```

```
f(1, 200); // f(1, 200, 1e-9);  
f(1); // f(1, 100, 1e-9);
```

Пространства имен

Разные разработчики библиотек могут определить глобальные объекты с одинаковыми именами или функции с одинаковыми именами и параметрами. Чтобы избежать конфликта имен при использовании этих библиотек в одной программе, в C++ появилась возможность разделения глобального пространства имен. Каждый разработчик должен определить свое пространство имен:

```
namespace ИМЯ {  
    объявления и определения  
}
```

Описания, относящиеся к одному пространству имен, могут быть разделены на несколько модулей. Можно определять вложенные пространства имен.

Обращаться к объектам и функциям из того же пространства имен можно напрямую. При использовании этих объектов и функций вне этого пространства, нужно либо обращаться указывая их полное имя:

имя_пространства_имен :: имя_функции (аргументы) , либо сделать их доступными для прямого использования с помощью оператора

`using имя_пространства_имен :: имя_функции ;`

или

`using namespace имя_пространства_имен ;`

По умолчанию все глобальные объекты и функции определяются в неименованном пространстве имен. Можно обращаться к глобальному объекту следующим образом:

```
int x=1;  
int main()  
{ int x=2;  
  cout << x << "\n"; // 2  
  cout << ::x << "\n"; // 1  
}
```

Все объекты и функции стандартной библиотеки C++ STL определены в

пространстве имен `std`. Поэтому необходимо в программе указывать оператор `using`:

```
#include <iostream>
using namespace std;
int main()
{ cout<<"Hello, world!\n";
}
```

Существуют также варианты заголовочных файлов для стандартной библиотеки, в которых все функции объявлены в пространстве имен `std`. Например, `<cstdio>` содержит объявления из `<stdio.h>` (для получения имени C++ версии из имени файла заголовочного файла C нужно убрать `".h"` и добавить букву `"c"`).

Операции преобразования

Нельзя преобразовать `void *` к указателю любого типа без явного вызова операции преобразования.

```
int *p1=(int *)calloc(100,sizeof(int)); // старая форма
int *p2=reinterpret_cast<int *>(calloc(100,sizeof(int))); //
новая форма
```

Операцию преобразования **(тип) выражение** можно записывать в форме **тип (выражение)**, если имя типа состоит из одной лексемы.

Любое преобразование является источником потенциальных ошибок, поэтому в C++ были введены четыре специализированных операции преобразования. При использовании этих операций компилятор сможет выполнить проверку на допустимость преобразования на этапе компиляции или при выполнении. Вызовы этих операций легче находить в тексте программы, чем вызовы в форме, унаследованной от C, что упрощает модификацию программы.

`const_cast<тип> (выражение)`

Служит для добавления или удаления модификатора `const` или `volatile` (антоним `register`). Так как добавление `const` не требует явного вызова операции преобразования, то вызывать эту операцию необходимо только для удаления `const`.

`dynamic_cast<тип> (выражение)`

Служит для преобразования указателя или ссылки на объект одного класса в указатель или ссылку соответственно на объект другого класса в пределах одной иерархии классов. Оба класса должны иметь хотя бы один виртуальный метод, например, деструктор. Если преобразование недопустимо, то в случае преобразования указателей операция возвращает нулевой указатель, а в случае ссылок – исключение `bad_cast`. Для повышающего преобразования (к указателю или ссылке на объект базового класса) явного вызова операции преобразования не требуется, поэтому эта операция используется для понижающего (к указателю или ссылке на объект производного класса) или перекрёстного преобразования.

`static_cast<тип> (выражение)`

Служит для вызова преобразований для стандартных типов данных (например, вещественное в целое и обратно) и преобразований, определенных программистом. Также можно использовать для повышающих и понижающих преобразований в пределах одной иерархии классов *без проверки*, при этом не требуется наличие виртуальных методов.

`reinterpret_cast<тип> (выражение)`

Служит для преобразований не связанных между собой типов, например, указатель в целое и обратно, указатель `void *` к любому указателю. При переносе программы на компьютер с процессором другой архитектуры или разрядностью все операции преобразования этого вида должны тщательно проверяться.

Другие изменения

Появился тип `bool` и зарезервированные слова `true` и `false`. Любое ненулевое значение преобразуется в `true`, а нулевое – в `false`.

Объекты можно объявлять в любой точке программы, в том числе в инициализирующей части цикла `for` и в условии `if` и `while`. Можно инициализировать объекты любым выражением, не только константным, в том числе и статические объекты.

Операции `typeid` в качестве аргумента можно указать любое выражение или тип (как операции `sizeof`). Операция возвращает информацию о типе выражения в виде объекта класса `type_info`. Если определить тип выражения невозможно, операция порождает исключение `bad_typeid`. Метод `name()` класса `type_info` возвращает строку `char *` с именем типа, а операции `==` и `!=` позволяют сравнить два типа.

Примечание: Некоторые из указанных возможностей C++ были введены в новый стандарт языка C 1999 года. Полной поддержки стандарта C99 пока нет ни в одном из распространенных компиляторов.

2. Классы

Основные определения

Программа с точки зрения ООП является совокупностью взаимодействующих между собой объектов. **Объект** представляет собой конкретный опознаваемый предмет, единицу или сущность (реальную или абстрактную), имеющую четко определенное функциональное назначение в данной предметной области.

Каждый объект обладает состоянием, поведением и идентичностью и является экземпляром некоторого класса. **Класс** – это множество объектов, имеющих схожую структуру и поведение.

Класс является абстрактным типом данных, определяемым пользователем, и представляет собой описание реального объекта в виде данных и операций для работы с ними.

Данные-элементы класса называются **полями**, в них хранится текущее состояние объекта. Операции могут быть реализованы как **методы** (функции-элементы) или обычные функции. Операции могут быть следующих видов:

- *конструктор* служит для инициализации объекта;
- *деструктор* необходим для освобождения используемых объектом ресурсов;
- *селектор* позволяет считать состояние объекта, не изменяя его;
- *модификатор* используется для изменения состояния объекта;
- *итератор* позволяет получить доступ к частям объекта в некотором порядке.

Определение класса. Спецификаторы доступа

Синтаксис:

```
class имя_класса {  
    элементы_класса  
};
```

или

```
struct имя_класса {  
    элементы_класса  
};
```

С помощью спецификаторов доступа можно управлять видимостью элементов класса. Спецификатор `private` (закрытый) означает видимость элементов только для методов и друзей класса. Спецификатор `protected` (защищенный) означает видимость элементов только для методов и друзей класса и его наследников (наследники могут видеть эти элементы только в объектах собственного типа). Спецификатор `public` (открытый) означает видимость элементов из любого кода. Действие любого спецификатора распространяется до следующего спецификатора или до конца объявления класса. По умолчанию для элементов класса, объявленного с помощью `struct`, установлен доступ `public`, а для класса, объявленного с помощью `class`, — `private`.

Рекомендуется все поля в `class` делать закрытыми или защищенными элементами, а в `struct` — открытыми.

Пример объявления класса:

```
const int Stack_size=100; // размер стека
class Stack {
    int s[Stack_size]; // поля в закрытом разделе
    int t;
public:
    void clear(); // методы в открытом
    bool isEmpty();
    bool isFull();
    void pop();
    int top();
    void push(int);
};
```

Определение и вызов методов. Указатель **this**

Объявление класса обычно помещается в заголовочном файле (.h), а сами методы определяются в файле реализации (.cpp). Определения простых методов можно поместить в объявление класса, в этом случае они будут считаться определенными со спецификатором `inline`. При определении методов вне интерфейса класса непосредственно перед именем метода необходимо указать имя класса и символы `::`:

Пример объявления класса:

```
const int Stack_size=100; // размер стека
class Stack {
    int s[Stack_size];
    int t;
public:
    void clear() { t=-1; } // встраиваемые методы
    bool isEmpty() { return t==-1; }
    bool isFull() { return t==Stack_size-1; }
    void pop();
    int top();
    void push(int);
```

```
};  
class StackEmpty {}; // классы для информирования об ошибках  
class StackFull {};
```

Реализация методов класса:

```
void Stack::pop()  
{ if(!isEmpty())  
    --t;  
}  
int Stack::top()  
{ if(isEmpty())  
    throw StackEmpty(); // сообщаем об ошибке  
    return s[t];  
}  
void Stack::push(int v)  
{ if(isFull())  
    throw StackFull(); // сообщаем об ошибке  
    s[++t]=v;  
}
```

При вызове метода сначала указывают имя объекта, затем после точки указывают имя метода и список аргументов в скобках. При использовании указателя на объект, вместо точки используют операцию ->

Пример вызова методов:

```
int main()
{
    Stack s1;
    s1.clear();
    s1.push(10);
    cout<<s1.top()<<"\n";
    s1.pop();
    ...
    Stack *s2;
    s2=new Stack;
    s2->clear();
    s2->push(10);
    cout<<s2->top()<<"\n";
    s2->pop();
    ...
}
```


Адрес объекта, к которому применяется метод, передается с помощью скрытого параметра и доступен внутри метода под именем `this`. С помощью указателя `this` можно также обратиться к полю при совпадении имени поля и имени параметра метода. Другой вариант решения этой проблемы – написать имя класса и `::` перед именем поля.

```
struct time {  
    int h,m;  
    void set(int h, int m)  
    {  
        this->h=h;  
        time::m=m;  
    }  
};
```

Конструктор

Определение: Конструктор – это специальный метод, имя которого совпадает с именем класса. Конструктор не имеет возвращаемого значения, даже `void`. Конструктор не может быть объявлен как `const`, `static` или `virtual`. Класс может иметь несколько конструкторов с разными сигнатурами.

Синтаксис: **имя_класса** (**тип_параметра1**, **тип_параметра2**, ...) ;

Задача: Инициализировать **все** поля.

Назначение: Вызывается компилятором автоматически при создании объектов:

- для локальных объектов – при выполнении оператора, в котором они объявлены;
- для глобальных статических объектов – перед вызовом функции `main` (порядок вызовов **не определен**), для собственных статических значений функции – при первом выполнении этой функции;
- для объектов, создаваемых в динамической памяти – при выполнении операции `new`.

Замечания:

- Перед выполнением тела конструктора, компилятором автоматически вызываются конструкторы для базовых классов, затем конструкторы для полей в порядке, указанном при объявлении класса.
- Можно указать аргументы для вызова конструкторов базовых классов и полей, используя список инициализации, который записывается после : между заголовком и телом конструктора. В списке инициализации через запятую пишутся имена базовых классов или полей, в скобках указываются аргументы для конструкторов. Для базовых классов и полей, не указанных в списке инициализации вызываются конструкторы по умолчанию. Базовые классы и поля в списке инициализации рекомендуется перечислять в порядке, указанном при объявлении класса. Можно указать их в другом порядке, но ошибки использования неинициализированных полей станут менее заметными, так как вызовы конструкторов всё равно будут происходить в порядке, заданном при объявлении класса.
- Конструктор не может быть вызван как метод, он вызывается только при создании объекта:
- Выражение **имя_класса(аргументы)** создает временный объект,

уничтожаемый после выполнения оператора, в котором используется это выражение (скобки обязательны, даже если список аргументов пустой).

- Выражение `new имя_класса(аргументы)` создает объект в динамической памяти (если список аргументов пустой, скобки можно не писать).
- Оператор объявления `имя_класса имя_объекта(аргументы);` создает новый объект в статической памяти или на стеке (если список аргументов пустой, скобки не пишутся).

Примеры:

```
class Stack {  
    int size, t;  
    int *s;  
public:  
    Stack(int) ;  
    bool isEmpty() { return t==-1; }  
    bool isFull() { return t==size-1; }  
    void pop();  
    int top();  
    void push(int) ;  
};
```

```
Stack::Stack(int size)
{
    Stack::size=size;
    s=new int [size];
    t=-1;
}
void fun(const Stack &);
int main()
{
    Stack s1(100); // создается объект на стеке
    s1.push(10);
    ...
    fun(Stack(100)); // создается временный объект и
передается по ссылке в функцию
    Stack *s2;
    s2=new Stack(100); // объект создается в динамической
памяти
    s2->push(10);
    ...
}
```

Аналогичный синтаксис можно использовать для создания объектов стандартных типов:

```
int a(100); // вместо int a=100;  
int *b;  
b=new int(100); // выделить динамически память для int  
                // и инициализировать её числом 100
```

Пример определения конструктора со списком инициализации

```
Stack::Stack(int size):  
    size(size), // имя_поля(выражение) - нет коллизии имен  
    s(new int [size]), t(-1)  
{}
```

Деструктор

Определение: Деструктор – это специальный метод без параметров, имя которого состоит из символа ~ и имени класса. Деструктор не имеет возвращаемого значения, даже `void`. Деструктор не может быть объявлен как `const` или `static`, но может быть `virtual`.

Синтаксис: `~имя_класса () ;`

Задача: Освободить используемые объектом ресурсы, обычно это динамически выделяемая память, реже каналы ввода-вывода, связанные с файлами, устройствами и т.п.

Назначение: Вызывается компилятором автоматически при уничтожении объектов:

- для локальных объектов – при выходе из блока, в котором они объявлены;
- для статических объектов – после завершения функции `main`;
- для объектов, созданных в динамической памяти – при выполнении операции `delete`.

Замечания:

- После выполнения тела деструктора, компилятором автоматически вызываются деструкторы для полей, а затем базовых классов в порядке, обратном указанному при объявлении класса.
- Если деструктор явно не определен, компилятор сам создает деструктор с пустым телом: `~имя_класса () { }`
- В случае использования классом динамической памяти необходимо определять деструктор самостоятельно.
- Если класс будет использован как базовый в иерархии классов, рекомендуется сделать его деструктор виртуальным. Тогда при выполнении операции `delete` будет вызываться правильный деструктор, независимо от типа указателя.
- Не рекомендуется вызывать деструктор явно, единственным исключением является случай повторной инициализации объекта (пример в перегрузке `new`)

Пример:

```
class Stack {  
    int size,t;  
    int *s;  
public:  
    ~Stack();  
    ...  
};  
Stack::~~Stack()  
{  
    delete[] s;  
}
```

Конструктор по умолчанию

Определение: Конструктор по умолчанию – это конструктор, который можно вызвать без аргументов. Конструктор может иметь параметры, но все они должны иметь значения по умолчанию.

Синтаксис:

имя_класса () ;

**имя_класса (тип_параметра1 =значение_по_умолчанию, тип_параметра2
=значение_по_умолчанию, ...) ;**

Задача: Такая же, как у любого конструктора – инициализировать все поля.

Назначение: Без этого конструктора невозможно создать массив объектов.

Замечания:

- Если в классе не определен **ни один** конструктор, компилятор создает сам конструктор по умолчанию. В этом автоматически созданном конструкторе вызываются конструкторы по умолчанию для всех базовых классов и полей.
- Если у какого-то поля или базового класса нет конструктора по умолчанию, компилятор при попытке самостоятельно создать конструктор по умолчанию выдаст сообщение об ошибке.

Примеры:

```
class String {  
    int len;  
    char * str;  
public:  
    String(const char *s=""); // конструктор по умолчанию  
};  
String::String(const char *s):len(strlen(s)),str(new  
char[len+1])  
{  
    strcpy(str,s);  
}  
int main()  
{  
    String a; // создание строки, используя конструктор по  
умолчанию  
    String b[100]; // создание массива строк  
    String c(); // это не определение объекта, а объявление  
функции без параметров!!!  
}
```

Конструктор копий

Определение: Конструктор копий – это конструктор, у которого один аргумент – ссылка на константный объект определяемого класса.

Синтаксис: **имя_класса** (const **имя_класса**&) ;

Задача: Создание точной копии объекта, переданного в качестве аргумента.

Назначение: Конструктор вызывается компилятором при передаче аргументов в функцию по значению или возврате значения определяемого класса из функции. Можно также использовать явно при создании нового объекта, имеющего состояние одинаковое с другим объектом.

Замечания:

- Если конструктор копий не определяется программистом, компилятор создает его сам. В этом автоматически созданном конструкторе вызываются конструкторы копий для всех базовых классов и полей.
- В большинстве случаев нет необходимости определять конструктор копий самостоятельно.
- Но если класс содержит указатели на динамически выделяемую память, то при создании копии произойдет дублирование идентичности этих значений. В результате изменение одного объекта приведёт к изменению другого объекта. Поэтому в случае использования указателей необходимо определять конструктор копий самостоятельно.
- Если создание копий не требуется, рекомендуется объявить конструктор копий в разделе `private`, не определяя его реализацию.

Примеры:

```
class Stack {
    int size,t;
    int *s;
public:
    Stack(const Stack &); // конструктор копий
    ...
};

Stack::Stack(const Stack &a) :
    size(a.size),t(a.t),s(new double[size])
{
    for(int i=0; i<=t; ++i)
        s[i]=a.s[i];
}

Stack reverse(Stack x)
    // параметр x функции reverse передается по значению
{ Stack z(100);
    while(!x.isEmpty())
    { z.push(x.top());
      x.pop();
    }
}
```

```
    return z; // при возврате сначала вызывается конструктор копий,  
              // чтобы создать копию z, а затем деструктор для z  
}  
int main()  
{  
    Stack a(100);  
    ...  
    Stack b(a); // b - это копия a  
    ...  
    a=reverse(b);  
        // перед вызовом fun будет вызван конструктор копий создания копии b  
        // также компилятор выделит память для возвращаемого значения  
        // для инициализации этой памяти при завершении fun будет  
        // вызван конструктор копий  
}
```


Конструктор-преобразователь

Определение: Конструктор-преобразователь – это конструктор, который можно вызвать с одним аргументом. Конструктор может иметь более одного параметра, но все они должны иметь значения по умолчанию.

Синтаксис:

имя_класса (**другой_тип**) ;

имя_класса (const **другой_тип**&) ;

Задача: Инициализировать все поля, используя для установки начального состояния значение другого типа.

Назначение: Используется для преобразования в определяемый тип значения другого типа. Компилятор может вызывать его самостоятельно, если аргумент функции или операции имеет тип **другой_тип**, а параметр – тип **имя_класса**.

Замечания:

- Если конструктор с одним аргументом не должен использоваться как конструктор-преобразователь, перед ним нужно написать `explicit` (явный).

Примеры:

```
class String {  
public:  
    String(const char *s=""); // конструктор-преобразователь  
    ...  
};  
void fun(const String &s);  
int main()  
{  
    String a("ABC"); // создание строки  
    char s[]="ABCDE";  
    fun(s);  
    // перед вызовом функции будет вызван конструктор-преобразователь,  
    // временный объект будет уничтожен после выполнения оператора  
    a=String(s); // явный вызов конструктора
```

```

    a=(String)s; // устаревшая форма операции преобразования
    a=static_cast<String>(s); // новая форма
}

class Stack {
public:
    explicit Stack(int); // конструктор, но не преобразователь
    ...
};
void fun(Stack x);
int main()
{
    Stack a(100);
    ...
    fun(100); // ошибка, нет преобразователя int -> Stack
    fun(Stack(100)); // ОК
    a=Stack(100); // ОК, явный вызов конструктора
    a=(Stack)100; // ошибка, нет преобразователя
    a=static_cast<Stack>(100); // ошибка, нет преобразователя
}

```

Специальные элементы класса

Поля, значения которых не меняются на протяжении всего времени существования объекта, рекомендуется описывать со спецификатором `const`. Инициализировать такие поля нужно в списке инициализации конструктора.

Методы, не изменяющие состояния объекта (селекторы), рекомендуется описывать как константные. Спецификатор `const` указывается в заголовке метода после списка параметров. Только константные методы можно применять к константным объектам. Напротив, к неконстантным объектам можно применять как обычные методы, так и константные. Можно определить одновременно константный и обычный методы с одинаковым списком параметров, тогда для константных объектов будет вызываться константный метод, для неконстантных – обычный.

```
class Vector {  
    const int size;  
    double *const data;  
public:
```

```

Vector(int size):size(size),data(new double[size]) {}
~Vector() { delete []data; }
Vector(const Vector &):
int length() const { return size; }
double get(int i) const;
double &get(int i);
};
Vector::Vector(const Vector &v):
    size(v.size),data(new double[size])
{ for(int i=0; i<size; ++i)
    data[i]=v.data[i];
}
class BadIndex {};
double Vector::get(int i) const
{ if(i<0 || i>=size) throw BadIndex();
  return data[i];
}
double & Vector::get(int i)
{ if(i<0 || i>=size) throw BadIndex();
  return data[i];
}

```

Поля, описанные с помощью спецификатора `static`, являются общими для всех объектов класса. Фактически это глобальные переменные, видимостью которых можно управлять с помощью спецификаторов доступа. В файле реализации необходимо разместить поле в статической памяти и указать её начальное значение. Память, занимаемая статическим полем, не учитывается при определении размера объект с помощью операции `sizeof`.

Статические методы могут обращаться только к статическим полям и вызывать другие статические методы. Этим методам не передается указатель `this`. Вызывать эти методы можно либо обычным образом, либо через имя класса и `::`

```
// объявление
class One { // класс, допускающий создание только одного экземпляра
    int a;
    static One *o; // указатель на единственный экземпляр
    One(int x):a(x) {} // ограничиваем доступ к конструктору
    One(const One&); // запрещаем создание копий
    One& operator=(const One&); // и операцию присваивания
public:
    static One *instance(); // получение экземпляра класса
    ~One() { o=NULL; } // разрешаем уничтожение
    int get() { return a; } // прочие методы
};

// реализация
One * One::o=NULL;
One * One::instance()
{ if(o==NULL) //если нет экземпляра
    o=new One(10); // создать его
    return o;
}

// использование
int main()
```

```
{  
    cout<<One::instance()->get()<<"\n";  
    delete One::instance();  
}
```

При использовании одновременно спецификаторов `const` и `static` получается аналог глобальной константы, видимостью которой можно управлять. Значение такой константы нужно непосредственно указывать при объявлении.

```
class Stack {  
    static const int size=100;  
    int s[size];  
    int t;  
public:  
    void clear() { t=-1; }  
    bool isEmpty() const { return t== -1; }  
    bool isFull() const { return t==size-1; }  
    void pop();  
    int top() const;  
    void push(int);  
};
```


Друзья класса

Вызов метода похож на вызов обычной функции, за исключением того, что объект, к которому применяется метод, передается как скрытый аргумент. Получение длины строки можно реализовать как метод:

```
class String {
    int len;
    char * str;
public:
    int length() const { return len; }
    ...
};
int main()
{ String s;
    ...
    cout<<s.length()<<"\n";
}
```

или использовать обычную функцию, которой строка передается в качестве параметра. Тогда эта функция для получения доступа к закрытым элементам класса должна быть объявлена внутри класса со спецификатором `friend`

```

class String {
    int len;
    char * str;
public:
    friend int length(const String &);
    ...
};
int length(const String & s)
{ return s.len; // необходим доступ к закрытому элементу
}
int main()
{ String s;
    ...
    cout<<length(s)<<"\n";
}

```

Как реализовать операцию – через метод или дружественную функцию – зависит только от предпочтений программиста, но чаще через обычные функции реализуются такие операции, которым необходимы два или более объекта определяемого типа.

Функция может быть дружественной сразу нескольким классам, можно объявить дружественным метод другого класса. Если многие методы другого класса должны иметь доступ к закрытым элементам, то можно объявить дружественным весь класс: `friend class имя_класса;`

Спецификаторы доступа не оказывают влияния на друзей класса. Друзей класса можно описывать в любом разделе класса.

Рекомендации по проектированию

Все поля и вспомогательные методы в `class` следует делать закрытыми или защищенными элементами. Для доступа к полям пользователям класса могут быть предоставлены тривиальные селекторы и модификаторы. Для запрета чтения или изменения некоторого поля соответствующий селектор или модификатор не определяется.

```
class A {  
    int value;  
public:  
    int getValue() const { return value; } // тривиальный  
селектор  
    void setValue(int newValue) { value=newValue; } //  
тривиальный модификатор  
};
```

Друзьями класса следует делать только операции этого класса. Прочие функции и классы должны использовать открытый интерфейс. Более того, некоторые операции могут быть реализованы через обычные функции, не являющиеся друзьями класса, если открытого интерфейса достаточно для их

реализации.

Методы и дружественные функции должны непосредственно обращаться к полям для получения их значений, а не через тривиальные селекторы. Более того, если какая-то операция класса, реализованная через обычную функцию, использует тривиальные селекторы для получения значений полей, необходимо сделать ее дружественной и обращаться к полям непосредственно.

Необходимо минимизировать интерфейс класса. В набор операций необходимо включать те операции, которые нельзя эффективно реализовать через другие методы и функции.

Методы, не изменяющие состояния объекта, должны быть объявлены как константные.

Методы, изменяющие состояние объекта, не должны возвращать информацию о состоянии объекта (текущем или предыдущем).

Взаимосвязанные поля (например, день и месяц в дате) должны

устанавливаться вызовом одного метода.

Если параметр не изменяется в теле функции, а его размер больше или равен 16 байт или он содержит указатель на динамически выделенную память, то его необходимо передавать как ссылку на константный объект. Иначе можно передавать по значению.

Для классов с динамическим выделением памяти обязательно определяйте деструктор, конструктор копий и операцию присваивания или запретите их автоматическое создание компилятором.

Объявляйте деструкторы виртуальными в классах, являющихся базовыми в иерархии.

В объявлении класса определяйте только методы, состоящие из одного оператора присваивания или `return`.

3. Перегрузка функций и операций

Правила связывания

Если требуется реализация одной и той же операции для данных различных типов, то можно определить нескольких функций с одним и тем же именем. Такие функции называют перегруженными. Перегруженные функции должны быть объявлены в одной области видимости (блоке, классе или пространстве имен) и иметь различные сигнатуры (список параметров функции в скобках). Сигнатуры должны отличаться количеством параметров или их типами. Возвращаемое значение в сигнатуру не входит.

Связывание – это процесс выбора экземпляра функции из набора в одной области видимости, который соответствует набору аргументов при ее вызове. Различают раннее (статическое, во время компиляции) и позднее (динамическое, во время выполнения) связывание. В C++ используется раннее связывание, но механизм виртуальных методов можно рассматривать как позднее связывание по первому, скрытому аргументу.

Выбор функции при связывании происходит на основе следующих приоритетов:

- Точное соответствие.
- Повышение точности: `char`, `short`, `wchar_t`, `enum`, `bool` → `int`; `float` → `double`.
- Стандартные преобразования: любой числовой тип в любой числовой тип (близость типов во внимание не принимается); любой указатель в `void *`; константа 0 к любому числовому типу или указателю; указатель или ссылка на производный класс к указателю или ссылке на базовый класс.
- Преобразования через конструкторы-преобразователи или операции преобразования, определенные программистом.
- Соответствие . . . (специальное обозначение для любого количества параметров, см. `printf`).

Выбирается функция или метод с большим точным числом соответствий, при одинаковом количестве точных соответствий – рассматривается количество повышений точности, затем учитывается число стандартных преобразований и т.д. Если две или более функции из набора получают одинаковый приоритет, то компилятор выводит сообщение об ошибке.


```
void f(double);           // 1
void f(const char *);    // 2
void f(...);             // 3
int main()
{
    int a[10];
    float b;
    f(b); // 1, повышение точности
    f('A'); // 1, стандартное преобразование
    f("A"); // 2, точное соответствие
    f(0); // ошибка, соответствует функциям 1 и 2
    f(1); // 1, стандартное преобразование
    f(a); // 3, соответствие ...
}
```

Правила перегрузки операций

- Нельзя определить новый лексический символ для операции.
- Нельзя изменить приоритет, арифметичность (т.е. количество аргументов) и ассоциативность операций.
- Нельзя перегрузить операцию для стандартных типов данных.
- Нельзя перегружать операции `.` `::` `.*` `?:` `sizeof` `typeid`.
- Операции `=` `[]` `()` `->` можно перегружать только как методы.
- Не рекомендуется перегружать операции `,` `||` `&&`.
- Поведение перегруженных операций должно соответствовать поведению этих операций для стандартных типов данных: операция `+` не должна изменять своих аргументов, операция `=` должна возвращать ссылку на левый аргумент и т.п.

Формат перегрузки унарной операции @ как метода:

тип_результата operator@()

или

тип_результата operator@() const

Формат перегрузки унарной операции @ как функции:

тип_результата operator@(имя_класса &)

или

тип_результата operator@(const имя_класса &)

Исключение: В случае постфиксных ++ и -- добавляется фиктивный параметр типа int.

как метод: **имя_класса** operator@(int)

как функция: **имя_класса** operator@(имя_класса & , int)

Формат перегрузки бинарной операции @ как метода:

тип_результата `operator@` (**тип_параметра2**)

или

тип_результата `operator@` (**тип_параметра2**) `const`

Левый аргумент операции передается как `*this` и может быть только определяемого типа. Конструкторы-преобразователи к левому аргументу не применяются.

Формат перегрузки бинарной операции @ как функции:

тип_результата `operator@` (**тип_параметра1**, **тип_параметра2**)

где по крайней мере один из параметров должен относиться к определяемому типу. Если левый аргумент может иметь тип, отличный от определяемого, необходимо определять операцию как функцию.

Замечания:

- При перегрузке операции как функции, эту функцию нужно делать дружественной классу только в том случае, если для её эффективной реализации необходим доступ к закрытым элементам класса, иначе лучше определять обычную функцию (см. тему ["Рекомендации по проектированию"](#))
- Если операция присваивания не определяется программистом, компилятор создает её сам. В этом автоматически созданном методе вызываются операции присваивания для всех базовых классов и полей.
- В большинстве случаев нет необходимости определять операцию присваивания самостоятельно.
- Но если класс содержит указатели на динамически выделяемую память, то при присваивании произойдет дублирование идентичности этих значений. В результате изменение одного объекта приведёт к изменению другого объекта. Поэтому в случае использования указателей необходимо определять операцию присваивания самостоятельно.
- Если присваивание объектов не требуется, рекомендуется объявить операцию присваивания в разделе `private`, не определяя её реализацию.
- Кроме присваивания компилятор сам определяет операцию взятия адреса.

Примеры перегрузки операций

Перегрузка операции присваивания:

а) для случая, когда перевыделения памяти не происходит

```
class BadSize {}; // класс для информирования об ошибке
class Vector {
    const int size;
    double *const data;
public:
    Vector &operator=(const Vector &) ;
    ...
};
Vector & Vector::operator=(const Vector &v)
{ if(size!=v.size)
    throw BadSize();
  for(int i=0;i<size;++i)
    data[i]=v.data[i];
  return *this;
}
```

б) для случая, когда память перевыделяется

```
class String {
    int len;
    char * str;
public:
    String &operator=(const String &);
    ...
};

String & String::operator=(const String &s)
{ String t(s); // создание копии
  std::swap(len,t.len); // обмен состояний объекта t и *this
  std::swap(str,t.str);
  return *this; // объект t со старым состоянием *this уничтожается
}
```

Определение постфиксного ++ через префиксный:

```
class A {  
    ...  
public:  
    A& operator++ ();  
};  
A operator++ (A &x, int)  
{ A t(x);  
  ++x;  
  return t;  
}
```

Определение + через += :

```
class A {  
public:  
    A& operator+=(const A&) ;  
};  
A operator+(A x, const A &y) // первый параметр передается  
по значению  
{ return x+=y;  
}
```


Определение != через == :

```
class A {  
public:  
    friend bool operator==(const A&, const A&);  
};  
bool operator!=(const A &x, const A &y)  
{ return !(x==y); }
```

Определение > >= <= == через < :

```
class A {  
public:  
    friend bool operator<(const A&, const A&);  
};  
bool operator>(const A &x, const A &y)  
{ return y<x; }  
bool operator>=(const A &x, const A &y)  
{ return !(x<y); }  
bool operator<=(const A &x, const A &y)  
{ return !(y<x); }  
bool operator==(const A &x, const A &y)  
{ return !(x<y || y<x); }
```

Перегрузка операции индексирования:

```
class BadIndex {}; // класс для информирования об ошибке
class Vector {
    const int size;
    double *const data;
public:
    double &operator[](int); // для неконстантных объектов
    double operator[](int) const; // для константных объектов
    ...
};

double & Vector::operator[](int i)
{ if(i<0 || i>=size)
    throw BadIndex();
  return data[i];
}

double Vector::operator[](int i) const
{ if(i<0 || i>=size)
    throw BadIndex();
  return data[i];
}
```

```
int main()
{ Vector a(10); // вектор из 10 элементов
  a[4]=a[5]*2; // изменяем элемент вектора
}
```

Перегрузка &, -> и *:

```
class A;
class Ptr { // умный указатель
public:
  A *operator->();
  A &operator*();
  ...
};
class A {
public:
  Ptr operator&(); // создание умного указателя
  ...
};
```

Перегрузка операции () для создания функционалов:

```
struct str_less { // функционал для сравнения двух строк
    bool operator()(const char *a, const char *b)
    { return strcmp(a,b)<0; }
};
int main()
{ char *s[100];
  sort(s,s+100,str_less()); // использование функционала
}
```

Перегрузка операции () для индексации по двум или более ключам:

```
class Matrix {
public:
    Matrix(int n, int m);
    double &operator()(int i, int j);
    double operator()(int i, int j) const;
};
int main()
{ Matrix a(10,10); // матрица 10x10
  a(4,5)=1; // изменяем элемент матрицы }

```

Перегрузка операций << и >> ввода-вывода:

```
class Point {  
    double x, y;  
public:  
    friend istream &operator>>(istream &, Point&);  
    friend ostream &operator<<(ostream &, const Point&);  
};  
istream &operator>>(istream &s, Point &p)  
{ char c;  
    return s>>c>>p.x>>c>>p.y>>c; // можно проверить, какие  
    СИМВОЛЫ до, между и после чисел  
}  
ostream &operator<<(ostream &s, const Point &p);  
{ return s<<" ("<<p.x<<" "<<p.y<<" " "; // \n не выводится!  
}
```

Формат ввода должен точно соответствовать формату вывода, не должно быть подсказок на ввод.

Операция преобразования

Операция преобразования определяется как метод с заголовком вида:

```
operator другой_тип ();
```

и используется при необходимости для преобразования из определяемого типа в **другой_тип**.

Не следует определять одновременно операцию преобразования и обратный к ней конструктор-преобразователь.

```
class String {  
    int len;  
    char * str;  
public:  
    String(const char *s=""); // конструктор-преобразователь  
    String& operator=(const String&); // операция присваивания  
    operator const char *() const { return str; } // операция  
преобразования  
    friend bool operator==(const String &, const String &);  
    ...  
};
```

```

int main()
{
    int x;
    String fmt, txt("ABC");
    fmt="%d\n"; // Ok, неявное преобразование const char * ->
String
    printf(fmt,x); // Ok, неявное преобразование String ->
const char *
    if(fmt==txt) // Ok, преобразований не нужно
        ...
    if(fmt=="ABC") // Ошибка, какое преобразование выполнять?
        // можно сравнить два String, а можно два char *
        ...
}

```

В таких случаях не определяют операцию преобразования, вместо неё делают обычный метод:

```

class String {
    int len;
    char * str;
public:

```

```

String(const char *s=""); // конструктор-преобразователь
String& operator=(const String&); // операция присваивания
const char * c_str() const { return str; } // метод вместо
операции преобразования
friend bool operator==(const String &, const String &);
...
};
int main()
{
    int x;
    String fmt, txt("ABC");
    fmt="%d\n"; // Ok, неявное преобразование const char * ->
String
    printf(fmt.c_str(), x); // Ok, нужно явно вызывать метод
    if(fmt==txt) // Ok, преобразований не нужно
        ...
    if(fmt=="ABC") // Ok, неявное преобразование const char * -> String
        ...
}

```


Перегрузка new и delete

Реализовать выделение участков памяти одинакового размера из некоторого пула можно более эффективно, чем выделение участков памяти произвольного размера. Кроме того, при работе с участками памяти одинакового размера не возникает проблема фрагментации, когда свободная память делится на множество участков небольшого размера.

Чтобы повысить эффективность программ, использующих динамические структуры данных, можно создать собственные варианты операций `new` и `delete` для выделения памяти под объекты некоторых классов, например, элементов списка. Эти операции определяются как статические методы класса, отдельно для одиночных объектов и массивов:

```
static void *operator new(size_t, доп_параметры) ;  
static void *operator new[] (size_t, доп_параметры) ;  
static void operator delete(void *, size_t);  
static void operator delete[] (void *, size_t);
```

Значения для дополнительных параметров можно указать при вызове `new` следующим образом:

```
new (доп_аргументы) тип (список_выражений) ;
```

Можно также перегрузить глобальную операцию `new`, указав дополнительные параметры. Глобальная операция `new` уже перегружена с дополнительным параметром типа `void *` (память не выделяется, а возвращается указанный адрес, используется для инициализации с помощью конструктора произвольного участка памяти) и `nothrow_t` (в случае ошибки не возникает исключительной ситуации `bad_alloc`, операция возвращает 0).

Пример повторной инициализации объекта с помощью перегруженной формы оператора `new` и явного вызова деструктора:

```
Vector a(100);  
... // действия с вектором размером 100  
a.~Vector();  
new(&a) Vector(200);  
... // действия с вектором размером 200
```

4. Шаблоны

Шаблоны функций

С помощью шаблонов функций можно определить алгоритм, который может быть применен к данным различных типов. Компилятор генерирует код при первом вызове по заданному шаблону. Шаблонная функция определяется как обычная функция, но её определение должно быть в заголовочном файле и перед заголовком функции необходимо написать:

```
template <параметры_шаблона>
```

Параметрами шаблона чаще всего являются типы обрабатываемых функцией данных (перед именем такого параметра нужно написать `typename` или `class`), но допустимо также передавать константы и объекты, как и обычной функции. Для параметров шаблона можно указать значения по умолчанию.

Пример шаблона функции:

```
template <typename T>
inline T maximum(T a, T b)
{
    return a < b ? b : a;
}
```

Если параметры шаблона относятся только к типам параметров функции, то её можно вызывать обычным образом, компилятор определит аргументы шаблона самостоятельно. В остальных случаях их нужно указать явно после имени функции в <>:

```
int a,b,c;
```

```
double d,e;
```

```
c=max(a,b); // оба аргумента функции имеют тип int, значит  
аргумент шаблона - тип int
```

```
e=max<double>(d,a); // тип аргумента шаблона нельзя  
определить по типу аргументов функции,  
// указываем явно тип double
```

Как и обычные функции, шаблоны функций можно перегружать. Из набора шаблонных функций вызывается функция с наиболее специализированными параметрами, точно соответствующая аргументам при вызове.

Если определены одновременно обычные и шаблонные функции с одинаковым именем и при вызове не указаны аргументы для шаблона в <>, то сначала компилятор выполняет поиск обычной функции с точным соответствием типов параметров, затем шаблона функции с точным соответствием типов параметров, а затем пытается добиться соответствия

аргументов с параметрами обычной функции с помощью операций преобразования.

Шаблоны классов

Перед шаблоном класса необходимо написать:

`template <параметры_шаблона>`

Методы шаблонного класса должны быть определены в заголовочном файле либо в объявлении класса (тогда они являются встраиваемыми), либо как шаблонные функции.

Пример шаблона класса:

```
template <typename T=int, int size=100>
class Stack {
    T s[size];
    int t;
public:
    Stack():t(-1) {} // встраиваемые методы
    bool isEmpty() { return t==-1; }
    bool isFull() { return t==size-1; }
    void pop();
    const T & top();
    void push(const T &);
};
class StackEmpty {}; // классы для информирования об ошибках
class StackFull {};
template <typename T, int size>
void Stack<T,size>::pop()
{ if(!isEmpty())
    --t;
}
```

```

template <typename T, int size>
const T & Stack<T,size>::top()
{ if(isEmpty())
    throw StackEmpty();
    return s[t];
}
template <typename T, int size>
void Stack<T,size>::push(const T& v)
{ if(isFull())
    throw StackFull(); // сообщаем об ошибке
    s[++t]=v;
}

```

При создании объектов шаблонного класса после имени класса в <> указываем аргументы шаблона:

```

Stack<int,200> a; // стек целых чисел размером 200
Stack<double> b;  // стек вещественных чисел размером 100
Stack<> c;        // стек целых чисел размером 100

```

Специализация

Можно специализировать (изменять реализацию) для определенных типов данных шаблон функции, отдельные методы шаблона класса или шаблон класса целиком.

При специализации функций и отдельных методов допускается только *полная специализация*. Перед определением специализации пишется `template <>`, а после имени функции или класса добавляется список аргументов основного шаблона в `<>`. Аналогичной функциональности можно добиться, если определить обычную функцию или перегрузить шаблон функции. Разница в том, что генерация кода для обычной функции будет происходить при компиляции её определения (т.е. всегда), а для шаблона и специализации – при первом использовании (инстанцировании). Разница между специализацией и перегрузкой шаблона чисто концептуальная.

Пример специализации шаблона функции:

```
template<>  
inline const char * maximum<const char *>(const char *a,  
const char * b)  
{ return strcmp(a,b)>0?a:b; }
```


Специализация шаблона класса определяется как шаблон класса, но нельзя указывать значения по умолчанию для параметров шаблона, а после имени класса указывается список аргументов основного шаблона в <>, который позволяет компилятору установить связь между параметрами специализации и основного шаблона. При специализации шаблона класса можно полностью изменить весь интерфейс класса. Допустима частичная специализация шаблона класса.

Пример специализации шаблона класса:

```
template <int size> // частичная специализация
class Stack<char *,size> {
    char *s[size];
    int t;
public:
    Stack():t(-1) {} // встраиваемые методы
    bool isEmpty() { return t==-1; }
    bool isFull() { return t==size-1; }
    ~Stack() { while(!isEmpty()) pop(); }
    void pop();
```

```

    const char * top();
    void push(const char *);
};

template <int size>
void Stack<char *,size>::pop()
{ if(!isEmpty())
    delete[] s[t--];
}

template <int size>
const char * Stack<char *,size>::top()
{ if(isEmpty())
    throw StackEmpty();
    return s[t];
}

template <int size>
void Stack<char *,size>::push(const char *v)
{ if(isFull())
    throw StackFull(); // сообщаем об ошибке
    s[++t]=new char[strlen(v)+1];
    strcpy(s[t],v);
}

```

Инстанцирование

Инстанцирование шаблона – это генерация кода функции или класса по шаблону для конкретных параметров. Различают неявное инстанцирование, которое происходит при вызове функции или создании объекта класса, и явное инстанцирование с помощью резервированного слова `template`. Инстанцирование можно делать только в точке программы, где доступна реализация шаблона функции или методов шаблонного класса.

Реализация шаблонов должна быть определена в заголовочном файле, чтобы компилятор смог выполнить инстанцирование шаблона для любых аргументов, но можно скрыть реализацию, используя явное инстанцирование. В заголовочном файле при этом остаётся только объявление шаблонов, а реализация помещается в отдельный файл. Для всех возможных вариантов применения шаблонов в этом модуле выполняется явное инстанцирование. Плюсы: компиляция выполняется быстрее, сохраняется know-how, минус: использование шаблонов с непредусмотренными аргументами приведёт к ошибке при компоновке программы.

Заголовочный файл my.h:

```
template <typename T> T sqr(T) ;  
template <typename T> class X  
{  
    T x;  
public:  
    X(T x) :x(x) {}  
    T get() const;  
    void set(T) ;  
};
```

Реализация модуля my.cpp:

```
#include "my.h"  
template <typename T> T sqr(T x)  
{ return x*x;  
}  
template <typename T> T X<T>::get()  
{ return x;  
}  
template <typename T> void X<T>::set(T x)  
{ this->x=x; }
```

```
template double sqr(double); // явное инстанцирование шаблона функции
template int sqr(int);        // для двух типов аргументов
template class X<int>;        // и класса для типа int
```

Применение:

```
#include <iostream>
#include "my.h"
using namespace std;
int main()
{
    cout<<sqr(10)<<"\n";    // ОК
    cout<<sqr(1.2)<<"\n";  // ОК
    cout<<sqr(10L)<<"\n";  // Ошибка, нет sqr(long)

    X<int> x1(10);
    cout<<x1.get()<<"\n";  // ОК
    X<double> x1(1.2);
    cout<<x1.get()<<"\n";  // Ошибка, нет X<double>::get()
    return 0;
}
```

5. Наследование

Отношение наследования между классами

Механизм наследования позволяет создавать новые классы на основе существующих путем расширения или изменения их структуры и поведения. Наследование не следует путать с агрегацией (отношением часть-целое) и использованием:

- водитель → автомобиль: водитель сам никуда не может поехать, нет связей между их свойствами, но только водитель может выполнять действия над автомобилем, следовательно, это отношение использования;
- двигатель → автомобиль: двигатель самостоятельно никуда не может поехать (нет колес), характеристики двигателя являются частью свойств автомобиля (максимальная скорость, мощность, тип топлива, его расход и т.п.), следовательно, это отношение агрегации;
- грузовик → автомобиль: грузовик имеет такие же свойства и поведение, как автомобиль, добавляется новый элемент – кузов и возможность перевозить грузы, следовательно, это отношение наследования.

После имени *производного класса* можно написать символ : и указать список *базовых классов* через запятую. Перед именем базового класса можно указать спецификатор доступа для наследуемых элементов. По умолчанию для базовых классов в классах, объявленных с помощью `struct`, устанавливается спецификатор доступа `public`, а в классах, объявленных с помощью `class`, – `private`.

Можно изменить уровень доступа к отдельным наследуемым элементам класса (кроме закрытых элементов базового класса, доступа к которым из производного класса нет), используя `using`.

```
class A {  
public:  
    void f();  
    void g();  
};  
class B : protected A {  
public:  
    using A::g; // g – public, f – protected  
};
```

С помощью `using` можно в производном класса добавить метод к набору методов с тем же именем в базовом классе:

```
class A {
public:
    void f();
};
class B1 : public A {
public:
    void f(int);
};
class B2 : public A {
public:
    void f(int);
    using A::f;
};
int main()
{ B1 b1;
  b1.f(); // Ошибка, метод перекрыт
  B2 b2;
  b2.f(); // Ok, метод перегружен }
```


Виртуальные методы и абстрактные классы

Полиморфизм обеспечивает взаимозаменяемость объектов с одинаковым интерфейсом. В C++ полиморфизм возможен только для объектов, относящихся к одной иерархии, и реализуется с помощью виртуальных методов. Те методы, которые могут быть переопределены в производных классах, должны быть объявлены в базовом классе с помощью спецификатора `virtual`. При переопределении метод должен иметь то же имя и набор параметров. Спецификатор `virtual` при переопределении метода в производных классах можно не указывать. Допускается изменять тип возвращаемого результата – вместо ссылки или указателя на базовый класс вернуть ссылку или указатель на производный класс. Переопределить виртуальный метод можно в сколь угодно дальнем потомке и сколько угодно раз. Если метод в каком-то классе этой иерархии не переопределяется, то используется определение из базового класса этого класса.

Статический метод нельзя объявить виртуальным.

При вызове виртуального метода вызывается его реализация из реального класса объекта, хотя объект может быть представлен ссылкой или указателем на базовый класс. Если необходимо вызвать реализацию

виртуального метода из базового класса, то перед именем метода нужно написать имя базового класса и `::`.

Если определить виртуальный метод в базовом классе нельзя, то после заголовка вместо тела метода нужно указать `=0`. Такой метод называется *чисто виртуальным*, а класс, содержащий хотя бы один такой метод, – *абстрактным*. Нельзя создавать объекты абстрактных классов, но можно работать со ссылками или указателями на абстрактные классы. В производном классе все чисто виртуальные методы должны быть определены, чтобы класс перестал быть абстрактным.

```
class Shape {  
public:  
    virtual double square()=0; // чисто виртуальный метод  
};  
class Rectangle: public Shape  
{ double w,h;  
public:  
    Rectangle(double w, double h):w(w),h(h){}  
    double square() { return w*h; } // переопределение  
    виртуального метода
```

```

};
class Circle: public Shape
{ double r;
public:
    Circle(double r):r(r){}
    double square() { return r*r*acos(0.)*2; } //
переопределение виртуального метода
};
int main()
{ Shape *s=new Circle(10);
  cout<<(s->square())<<"\n"; // вызов виртуального метода
}

```

Чисто виртуальному методу в абстрактном классе можно дать определение, которое можно вызвать по имени класса.

```

class Figure {
    int color, x, y;
public:
    Figure(int color, int x, int y):color(color),x(x),y(y) {}
    virtual void draw()=0; // чисто виртуальный метод
};

```

```

void Figure::draw() // определение чисто виртуального метода
{ setcolor(color);
  moveto(x,y); }
class Rectangle: public Figure
{ int w,h;
public:
  Rectangle(int color, int x, int y, int w, int h)
    :Figure(color,x,y),w(w),h(h){}
  void draw(); // переопределение виртуального метода
};
void Rectangle::draw()
{ Figure::draw(); // вызов реализации метода из базового класса
  linerel(w,0);
  linerel(0,h);
  linerel(-w,0);
  linerel(0,-h);
}
int main()
{ Figure *s=new Rectangle(RED,10,20,100,50);
  ...
  s->draw(); // вызов виртуального метода }

```

Множественное наследование

При множественном наследовании класс наследует структуру и поведение от нескольких классов.

При множественном наследовании могут появиться следующие проблемы:

1. Наследование от общего предка. В этом случае используют виртуальные классы.

```
class A {};  
class B: virtual public A {};  
class C: virtual public A {};  
class D: public B, public C {};
```

При использовании виртуальных классов конструкторы выполняются в специальном порядке: сначала конструкторы всех виртуальных классов, а потом конструкторы базовых классов и полей в обычном порядке и тело конструктора. Для деструкторов порядок строго противоположный.

2. Методы с одинаковой сигнатурой в разных базовых классах. В этом случае используют явное указание имени класса перед именем метода при вызове.

```
class A {  
public:  
    void f();  
};  
class B {  
public:  
    void f();  
};  
class C: public A, public B {};  
int main()  
{ C c1;  
    c1.A::f(); // метод из класса A  
    c1.B::f(); // метод из класса B  
}
```

3. Переопределение виртуальных методов с одинаковой сигнатурой в разных базовых классах. В этом случае вводят промежуточные классы.

```
class A {
public:
    virtual void f(); };
class B {
public:
    virtual void f(); };
class A1: public A {
public:
    virtual void fA() { A::f(); }
    void f() { fA(); } };
class B1: public B {
public:
    virtual void fB() { B::f(); }
    void f() { fB(); } };
class C: public A, public B {
public:
    void fA();
    void fB(); };
```

6. Исключительные ситуации

Назначение. Стандартные исключения

В ходе выполнения функции может произойти *исключительная ситуация*, когда из-за обнаруженной ошибки продолжить нормальное выполнение алгоритма невозможно. Выполнение этого алгоритма должно завершиться, затем функция может:

- Выдать сообщение об ошибке и аварийно завершить выполнение программы. Недостатки: снижается надежность программы, введенная пользователем информация теряется, нет возможности предпринять альтернативные действия, например, сохранить информацию на другом диске.
- Вернуть специальное значение. Например, функция `malloc` при ошибке выделения памяти возвращает `NULL`. Недостатки: область значений функции может содержать все значения типа результата функции, проверка результата каждого вызова в сложном выражении невозможна, а в более простых приводит к существенному увеличению размера программы, поэтому часто проверка не выполняется.
- Установить состояние ошибки в специальной глобальной переменной. Например, математические функции записывают код ошибки в `errno`.

Недостатки: проверка этой переменной после каждого оператора с вызовом функции приводит к существенному увеличению размера программы, поэтому часто проверка не выполняется.

- Вызвать специальную функцию обработки ошибки. Например, некоторые реализации математических функций при ошибке вызывают функцию `matherr`. Недостатки: при отсутствии исключений эта функция должна выполнить одно из трех рассмотренных ранее действий.

Механизм исключений позволяет передать управление и всю необходимую информацию в ту часть программы, которая сможет обработать данную ошибку, а затем продолжить нормальное выполнение программы. Размер программы при этом не увеличивается.

Все стандартные исключения являются производными от класса `exception`.

К стандартным исключениям относятся:

- `bad_alloc`, генерируемое операцией `new`;
- `bad_cast`, генерируемое операцией `dynamic_cast`;
- `bad_typeid`, генерируемое операцией `typeid`;
- `bad_exception`, генерируемое для обработки исключения, непредусмотренном в заголовке функции;
- исключения, порождаемые классами и алгоритмами STL.

Порождение и перехват

Для порождения исключений используется оператор

`throw` **выражение** ;

Рекомендуется указывать в качестве аргумента объекты специальных классов, а не стандартных типов (`int`) или классов общего назначения (`vector`). Классы исключений рекомендуется выстраивать в иерархии.

Операторы, которые могут генерировать исключения, записываются внутри блока `try`, после которого идет набор обработчиков исключений `catch`. У каждого обработчика указывается тип исключений, который он может перехватывать, и опционально имя, с помощью которого можно обращаться к созданному оператором `throw` объекту внутри обработчика.

Перехват исключения рекомендуется делать по ссылке, чтобы избежать *срезки* объекта при преобразовании к базовому классу.

```
try
{ // операторы
}
catch(bad_alloc &)
{ // обработка ошибки выделения памяти
}
catch(io_error & e)
{ e.stream.clear(); // обработка ошибки ввода-вывода
}
catch(exception &)
{ // обработка других стандартных исключений
}
```

Порядок обработки исключений следующий:

- Создается копия аргумента `throw` в статической памяти.
- Происходит поиск подходящего обработчика. В процессе поиска выполняется *раскрутка* стека, т.е. выход из функций, где нет подходящих обработчиков и вызов деструкторов для локальных объектов.
- Если подходящий обработчик найден, управление передается ему, иначе вызывается функция `terminate`, которая по умолчанию аварийно

завершает программу, но можно задать другое действие с помощью функции `set_terminate`

- После выполнения обработчика копия аргумента `throw` в статической памяти уничтожается и управление передается на оператор, следующий за последним из обработчиков того набора, где был найден обработчик.

Если обработчик не смог полностью обработать исключение, то можно повторно сгенерировать исключение внутри обработчика с помощью оператора

`throw;`

Проверка на соответствие типа объекта и обработчика выполняется в порядке перечисления обработчиков, поэтому обработчики для исключений производных классов должны быть указаны до обработчиков базовых классов. Самым последним обработчиком в наборе может быть указан `catch (. . .)`, который перехватывает все исключения. Такой обработчик можно использовать для корректного освобождения ресурсов:

```

int g(int n)
{
    int * a;
    a=new int[n];
    try
    { // ...
    }
    catch(...)
    { delete[] a;
      throw;
    }
    delete[] a;
}

```

Спецификация исключений в заголовке функции

Возврат исключения – это альтернативный вариант возврата результата из функции, область значения функции при этом расширяется новым типом. Тип результата указывается в заголовке функции, аналогично должна быть возможность указать тип альтернативного результата. Для этого в заголовке функции после списка параметров указывается резервированное слово

`throw` со списком типов исключений в скобках. По умолчанию функция может вернуть исключение любого типа.

```
void f(); // исключение любого типа
```

```
void f() throw(); // никаких исключений
```

```
void f() throw(bad_alloc, io_error); // только исключения  
bad_alloc, io_error и производных от них классов
```

Если функция попытается вернуть непредусмотренное исключение, то вызывается функция `unexpected`, которая по умолчанию пытается создать исключение `bad_exception`. Можно задать другое действие с помощью функции `set_unexpected`. Если `bad_exception` нет в списке допустимых исключений, вызывается функция `terminate`.

Рекомендуется, чтобы деструктор не возвращал исключений, так как деструкторы вызываются при раскрутке стека и возникновение исключений в этот момент приведет к вызову `terminate` и аварийному завершению программы.

7. STL

Общие сведения

Standard Template Library – это набор классов и алгоритмов, входящих в стандарт языка C++. Готовые структуры данных и алгоритмы из библиотеки позволяют ускорить написание программ, что очень важно в условиях соревнований.

Вспомогательные компоненты

В заголовочном файле `<utility>` определены шаблоны функций для `operator!=` из `operator==` и `operator>`, `<=`, `>=` из `operator<` (для использования шаблонов написать `using namespace std::rel_ops;`).

Также `<utility>` включает шаблон класса для разнородных пар значений `pair<T1, T2>`. Для этого типа определены операции `==` и `<`. Для создания пары используется функция `make_pair(5, 3.1415926)`.

Функциональные объекты – это объекты, для которых определён `operator()`. Они важны для эффективного использования библиотеки. В местах, где ожидается передача указателя на функцию алгоритмическому шаблону, интерфейс установлен на приём объекта с определённым

`operator()`. Это не только заставляет алгоритмические шаблоны работать с указателями на функции, но также позволяет им работать с произвольными функциональными объектами. Использование функциональных объектов вместе с шаблонами функций увеличивает выразительную мощность библиотеки также, как делает результирующий код более эффективным. Например, если мы хотим поэлементно сложить два вектора `a` и `b`, содержащие `double`, и поместить результат в `a`, мы можем сделать это так:

```
transform(a.begin(), a.end(), b.begin(), a.begin(),  
plus<double>());
```

В заголовочном файле `<functional>` определены функционалы `plus`, `minus`, `times`, `divides`, `modulus`, `negate`, `equal_to`, `not_equal_to`, `less`, `greater`, `less_equal`, `greater_equal`, `logical_or`, `logical_and`, `logical_not`. Пользователь может производить свои функционалы, наследуя от `unary_function<TArg, TRez>` и `binary_function<TArg1, TArg2, TRez>` и определяя `operator()`. Можно превращать бинарные функционалы в унарные с помощью связывателей `bind1st` и `bind2nd`, заменяющих соответствующий аргумент на константу: `bind2nd(less<int>(), 40)` (значение меньше 40).

Итераторы

У всех контейнерных классов определены итераторы (аналог указателя), позволяющие получить доступ к отдельным элементам: `КЛАСС::iterator`, `КЛАСС::const_iterator`, `КЛАСС::reverse_iterator`, `КЛАСС::const_reverse_iterator` (`const` если не хотим изменять значения, `reverse` если от конца к началу)

Начальный итератор контейнера: `v.begin()` `v.rbegin()`

Конечный итератор: `v.end()` `v.rend()`

К итераторам применимы операции `++` и `--`, а иногда `+$n`, `-$n`

```
vector<double> v(100);
```

```
for (vector<double>::iterator iv=v.begin(); iv!=v.end(); ++iv)  
    *iv=0.5;
```

Для сравнения:

```
double v[100];
```

```
for (double *iv=v; iv!=v+100; ++iv)  
    *iv=0.5;
```

Алгоритмы

Все шаблонные алгоритмы работают не только со структурами данных в библиотеке, но также и с встроенными структурами данных C++. Например, все алгоритмы работают с обычными указателями.

Обработка скалярных значений

<code>max(a,b)</code>	максимальное значение
<code>min(a,b)</code>	минимальное значение
<code>swap(a,b)</code>	обмен

Обработка последовательностей

<code>max_element(first,last)</code>	значение максимального элемента набора
<code>min_element(first,last)</code>	значение минимального элемента набора
<code>find(first,last,value)</code>	поиск первого элемента, равного value
<code>find_if(first,last,fun)</code>	поиск первого элемента, для которого <code>fun(x)==true</code>
<code>count(first,last,value)</code>	подсчет количества элементов, равных value
<code>count_if(first,last,fun)</code>	подсчет количества элементов, для которых <code>fun(x)==true</code>
<code>search(first1,last1,first2,last2)</code>	поиск вхождения подпоследовательности, заданной итераторами (first2,last2)

<code>search_n(first1, last1, n, value)</code>	поиск подпоследовательности из n значений value
<code>copy(first, last, where)</code>	копирование последовательности в where
<code>transform(first, last, where, fun)</code>	преобразование элементов последовательности с помощью функционального объекта (функции) fun
<code>transform(first1, last1, first2, where, fun)</code>	преобразование элементов двух последовательностей, например, суммирование двух векторов a и b: <pre>vector<int> a(n), b(n), c(n); transform(a.begin(), a.end(), b.begin(), b.end(), c.begin(), plus<int>())</pre>
<code>fill(first, last, value)</code>	заполнение значением value
<code>fill_n(where, n, value)</code>	записать n значений value
<code>generate(first, last, fun)</code>	заполнение результатом функционального объекта (функции) fun()
<code>generate_n(where, n, fun)</code>	записать n результатов функционального объекта (функции) fun()
<code>random_shuffle(first, last)</code>	перемешивание

```
remove(first, last,  
value)
```

```
remove_if(first, last,  
fun)
```

```
remove_copy(first, last,  
where, value)
```

```
remove_copy_if(first,  
last, where, fun)
```

```
replace(first, last,  
oldvalue, newvalue)
```

```
replace_if(first, last,  
fun, newvalue)
```

```
replace_copy(first,  
last, where, oldvalue,  
newvalue)
```

удаление элементов, равных value,
возвращается итератор на конец новой
последовательности

например, удаление всех 0:

```
a.erase(a.remove(a.begin(),  
a.end(), 0), a.end())
```

удаление элементов, для которых fun(x)==true

копирование элементов, не равных value

копирование элементов, для которых
fun(x)==false

замена элементов, равных oldvalue на
newvalue

замена элементов, для которых fun(x)==true
на newvalue

копирование с заменой

```
replace_copy_if(first,  
last, where, fun,  
newvalue)
```

копирование с заменой

```
reverse(first, last)  
reverse_copy(first,  
last, where)
```

изменение порядка элементов на обратный

копирование в обратном порядке

```
rotate(first, middle, last)
```

циклический сдвиг, элемент *middle становится первым

```
rotate_copy(first,  
middle, last, where)
```

копирование с циклическим сдвигом

```
partition(first, last,  
fun)
```

перемещение элементов, для которых fun(x)==true, в начало последовательности

```
stable_partition(first,  
last, fun)
```

перемещение элементов, для которых fun(x)==true, в начало последовательности с сохранением порядка в исходной последовательности

```
next_permutation(first,  
last)
```

следующая перестановка, возвращается false, если следующей перестановки не существует

```
prev_permutation(first,  
last)
```

предыдущая перестановка

```
accumulate(first, last,  
init, fun=plus())
```

подсчет суммы элементов,
можно вместо сложения указать другую
бинарную функцию

*Сортировка и обработка упорядоченных последовательностей, множеств
(упорядоченных последовательностей без повторений)*

```
sort(first, last)
```

сортировка последовательности

```
stable_sort(first, last)
```

сортировка с сохранением порядка
элементов с равным ключом

```
unique(first, last)
```

удалить повторения элементов,
возвращается итератор на конец новой
последовательности

```
unique_copy(first, last,  
where)
```

копирование без повторений

```
merge(first1, last1,  
first2, last2, where)
```

слияние

```
binary_search(first, last,  
value)
```

проверка наличия значения value

```
lower_bound(first, last,  
value)
```

поиск первой позиции для вставки
значения

```
upper_bound(first, last,  
value)
```

поиск последней позиции для вставки
значения

например, подсчет количества элементов,
равных v:

```
int k= upper_bound(a.begin(), a.end(), v) -  
        lower_bound(a.begin(), a.end(), v);
```

```
includes(first1, last1,  
first2, last2)
```

проверка, что множество (first2, last2)
является подмножеством множества
(first1, last1)

```
set_intersection(first1,  
last1, first2, last2, where)
```

пересечение множеств

```
set_difference(first1,  
last1, first2, last2, where)
```

разность множеств

```
set_symmetric_difference(  
first1, last1,  
first2, last2, where)
```

симметрическая разность множеств

```
set_union(first1, last1,  
first2, last2, where)
```

объединение множеств

Работа с кучей

<code>make_heap(first, last)</code>	создать кучу из элементов последовательности
<code>push_heap(first, last)</code>	добавить значение <code>*(last-1)</code> к куче <code>(first, last-1)</code> , после выполнения куча станет <code>(first, last)</code>
<code>pop_heap(first, last)</code>	удалить наибольшее значение из кучи <code>(first, last)</code> и поместить его в <code>*(last-1)</code> , после выполнения куча станет <code>(first, last-1)</code>
<code>sort_heap(first, last)</code>	превратить кучу в упорядоченную последовательность

Для аргумента `where` можно использовать либо итератор (указатель) на начало контейнера достаточного размера, либо `back_inserter(s)`, который возвращает добавляющий итератор для контейнера `s`.

Класс vector

`vector` – вид последовательности, которая поддерживает итераторы произвольного доступа, он поддерживает операции вставки и удаления в конце с постоянным временем; вставка и удаление в середине занимают линейное время.

Создание:

```
vector<int> a; // 0 элементов
vector<double> b(10); //10 элементов
vector<double> c(10,0.0); //10 элементов, начальное значение 0
```

Доступ к элементам:

```
b[0]=c[4]; // без проверки
b.at(0)=c.at(4); // с проверкой выхода за границы
c.back()=b.front(); // последний и первый элементы (эмуляция стека)
```

Текущий размер: `c.size()`

При необходимости можно изменять размер:

```
a.resize(100);
```

```
b.resize(200,2.0); // всем новым элементам присвоить 2.0  
a.clear(); // опять 0 элементов
```

Добавлять новое значение в конец: `c.push_back(1.5);`

Удалять последнее значение: `c.pop_back();`

Вставка в середину:

```
c.insert(c.begin()+pos,1.5);
```

```
c.insert(c.begin()+pos,5,1.5); // 5 значений
```

```
c.insert(c.begin()+pos,b.begin(),b.end()); // из другого вектора
```

Удаление из середины:

```
c.erase(c.begin()+pos);
```

```
c.erase(c.begin()+pos,c.end()); // все элементы с pos до конца
```

Для повышения эффективности операций изменения размера можно зарезервировать память: `c.reserve(100);`

Работают операции сравнения (в лексикографическом порядке) и присваивание.

Замечание: `vector<bool>` хранит биты упаковано, поэтому нельзя получить адрес элемента

Класс string

Создание:

```
string a; string b("строка");
```

Длина строки: `b.length()` или `b.size()`

Сцепление:

```
a="текст"; a=a+b; a+=b; a+='!';
```

Обращение к символам: `a[i]=b[j]; a.at(i)='A';`

Подстрока: `a.substr(pos,n)` – длиной `n` символов, начиная с `pos`,

`a.substr(pos)` – до конца строки

Строка Си: `a.c_str()`

Вставка: `a.insert(pos,b);`

Удаление: `a.erase(pos,n)` – `n` символов, `a.erase(pos)` – до конца строки, `a.clear()` – всей строки

Замена: `a.replace(pos,n,b);`

Поиск:

```
size_type i=a.find(b); // первого вхождения подстроки  
i=a.find('A'); // или символа  
i=a.rfind(b); // последнего вхождения
```

```
i=a.rfind('A');
```

Вторым аргументом функции можно указать начальную позицию поиска.

Ввод строки:

```
getline(cin,s);  
getline(cin,s,'\n');
```

Ассоциативные контейнеры

`set` – это ассоциативный контейнер, который поддерживает уникальные ключи (не содержит ключи с одинаковыми значениями) и обеспечивает быстрый поиск ключей

```
set<int> s;  
s.insert(5); // добавить значение  
s.erase(5); // удалить значение  
s.erase(s.begin()); // удалить первый  
if(s.find(5)!=s.end()) // значение найдено
```

`multiset` допускает множественные копии того же самого значения ключа, для нахождения границ можно использовать `s.upper_bound(5)` и `s.lower_bound(5)`

`map` – ассоциативный контейнер, который поддерживает уникальные ключи и обеспечивает быстрый поиск значений другого типа `T`, связанных с ключами. Итератор в случае `map` является «указателем» на `pair<const Ключ,Значение>`

```
map<int,string> m;  
m[5]="текст"; // доступ по ключу  
s.insert(make_pair(5,string(«текст»))); // добавить значение  
s.erase(5); // удалить значение  
s.erase(s.begin()); // удалить первый  
if(m.find(5)!=m.end()) // значение найдено
```

`multimap` допускает множественные копии того же самого значения ключа, для нахождения границ можно использовать `m.upper_bound(5)` и `m.lower_bound(5)`

Прочие классы

```
bitset<100> b; // вектор из 100 битов
```

Допустимы все поразрядные операции `&` `|` `<<` `>>` `^` `~`
Быстрый подсчет единичных битов: `b.count()`

Проверка битов: `b.any()` – хотя бы один бит равен 1, `b.none()` – все биты равны 0, `b.test(i)` – состояние i-го бита

Установка битов: `b.set()` – все биты в 1, `b.set(i)` – i-й бит в 1, `b.reset()` – все биты в 0, `b.reset(i)` – i-й бит в 0, `b.set(i, val)` – i-й бит в `val`

Операция индексации `b[i]` позволяет проверить и установить i-й бит.

```
deque<int> d; // двухсторонняя очередь
list<int> l; // список
stack<double> s; // стек
queue<double> p; // очередь
priority_queue<double> p; // очередь с приоритетами
valarray<double> v(100); // числовой массив, определены
операции + - * / и т.д.
```

Поточные классы

Поточные классы `istream` и `ostream` получают instantiation классов-шаблонов для типа `char`:

```
typedef basic_ostream<char> ostream; // обычные символы  
typedef basic_ostream<wchar_t> wostream; // широкие символы  
(например, Unicode)
```

Для неформатированного ввода и вывода удобно использовать объекты `cin` и `cout`, но при необходимости управлять форматированием проще воспользоваться `printf` и `scanf`.

Вывод с помощью `printf` (для сравнения)

```
// вывод целого числа в шестнадцатичном виде с ведущими нулями  
int x=123;  
printf("%08X\n", x);  
// вывод вещественного числа в поле шириной 10 с 2 десятичными знаками  
double f=12.3;  
printf("%10.2lf\n", f);
```

Управление форматированием с помощью методов потоковых классов

Метод	Назначение
<i>Управление форматированием</i>	
<code>setf(flag)</code>	установка негрупповых флагов форматирования
<code>setf(flag, group)</code>	установка флага из группы
<code>unsetf(flag)</code>	сброс негрупповых флагов
<code>fmtflags flags()</code>	текущие флаги
<code>width(int)</code>	установка минимальной ширины поля, после вывода значения ширина поля сбрасывается на 0
<code>int width()</code>	текущая ширина поля
<code>precision(int)</code>	установка максимального количества десятичных знаков для вывода вещественных чисел, по умолчанию – 6
<code>int precision()</code>	текущая точность
<code>fill(char)</code>	установка символа заполнения по умолчанию – ' '
<code>int fill()</code>	текущий символ заполнения

Проверка и установка состояния

<code>bool good()</code>	true, если ошибок не было
<code>bool eof()</code>	true, при выполнении последнего ввода был обнаружен конец файла
<code>bool fail()</code>	true, при выполнении последнего ввода была ошибка ввода-вывода
<code>clear()</code>	очистить состояние ошибки для продолжения ввода
<code>clear(ios::failbit)</code>	установить состояние ошибки

Посимвольный ввод-вывод

<code>put(char)</code>	Вывод символа
<code>write(char* buf, int n)</code>	Вывод n символов
<code>flush()</code>	записать информацию из буфера на диск
<code>get(char&)</code>	Ввод символа
<code>int get()</code>	
<code>read(char* buf, int n)</code>	Ввод n символов
<code>int peek()</code>	подсмотреть следующий символ, для конца файла возвращается EOF
<code>putback(char)</code>	вернуть символ в поток

Позиционирование в файле

<code>pos_type tellg()</code>	получить текущую позицию чтения
<code>seekg(pos_type)</code>	установить позицию чтения
<code>seekg(off_type, seek_dir)</code>	
<code>pos_type tellp()</code>	получить текущую позицию записи
<code>seekp(pos_type)</code>	установить позицию записи
<code>seekp(off_type, seek_dir)</code>	

Открытие и закрытие файлов (классы `ifstream`, `ofstream`, `fstream`)

<code>open(char *name)</code>	открыть файл, где mode комбинируется из
<code>open(char *name, mode)</code>	флагов <code>in</code> , <code>out</code> , <code>app</code> , <code>ate</code> , <code>trunc</code> , <code>binary</code>
<code>close()</code>	закрыть файл

Группы и флаги определены в базовом классе `ios`

Флаг	Значение
------	----------

Группа `basefield` – система счисления для целых чисел

<code>dec</code>	десятичная
<code>oct</code>	восьмеричная
<code>hex</code>	шестнадцатеричная

Группа floatfield – формат представления для вещественных чисел

fixed с фиксированной точкой

scientific в экспоненциальной форме

Группа adjustfield – выравнивание в поле вывода

right вправо

left влево

internal знак числа – слева, значение – справа

Без группы

showbase выводить 0x перед шестнадцатеричными и 0 перед
восьмеричными числами

showpoint всегда выводить . и дробную часть

uppercase выводить шестнадцатеричные цифры и E в
вещественных числах в верхнем регистре

showpos выводить знак числа для положительных чисел

boolalpha выводить значения bool как true и false

skipws игнорирование пробелов при вводе символов с помощью
операции >>

```
// вывод целого числа в шестнадцатичном виде с ведущими нулями
int x=123;
cout.setf(ios::hex,ios::basefield);
cout.setf(ios::uppercase);
cout.width(8);
cout.fill('0');
cout<<x<<"\n";
cout.fill(' ');
// вывод вещественного числа в поле шириной 10 с 2 десятичными знаками
double f=12.3;
cout.setf(ios::fixed,ios::floatfield);
cout.setf(ios::showpoint);
cout.width(10);
cout.precision(2);
cout<<f<<"\n";
```

Управление форматированием с помощью манипуляторов из <iomanip>

Манипулятор	Назначение
<code>setw(w)</code>	установка ширины поля вывода
<code>setprecision(p)</code>	установка количества десятичных знаков для вывода вещественных чисел
<code>setfill(c)</code>	установка символа заполнения
<code>flush</code>	записать информацию из буфера на диск
<code>endl</code>	вывести переход на новую строку, а затем выполнить <code>flush</code>
<code>dec oct hex</code>	выбор системы счисления для целых чисел
<code>right left internal</code>	выравнивание в поле вывода
<code>fixed scientific</code>	формат представления для вещественных чисел
<code>showbase noshowbase</code>	установка и сброс соответствующих флагов форматирования
<code>showpoint noshowpoint</code>	
<code>showpos noshowpos</code>	
...	

Полезный совет

Использование `endl` существенно замедляет работу программы, так как для каждой строки выполняется принудительный вывод буфера в файл (это полезно только для ведения лога действий программы). Поэтому вместо `endl` лучше использовать `'\n'` или `"\n"`.

```
// вывод целого числа в шестнадцатичном виде с ведущими  
нулями
```

```
int x=123;
```

```
cout<<hex<<uppercase<<setw(8)<<setfill('0')<<x<<setfill(''  
'<<"\n";
```

```
// вывод вещественного числа в поле шириной 10 с 2  
десятичными знаками
```

```
double f=12.3;
```

```
cout<<fixed<<showpoint<<setw(10)<<setprecision(2)<<f<<"\n";
```

Управление форматированием с помощью класса `format` из библиотеки `boost`

```
// вывод целого числа в шестнадцатичном виде с ведущими нулями
int x=123;
cout<<format("%08X\n")%x;
// вывод вещественного числа в поле шириной 10 с 2 десятичными знаками
double f=12.3;
cout<<format("%10.2f\n")%f;
```

Управление форматированием с помощью собственных манипуляторов

```
// Пример манипулятора для вещественных чисел
struct fmt_f {
    int w,p;
    fmt_f(int w=0, int p=6):w(w),p(p){}
};
ostream& operator<<(ostream& s, fmt_f f)
{ s.width(f.w);
  s.fill(' ');
  s.setf(ios::fixed,ios::floatfield);
  s.precision(f.p);
```

```
    if (f.p>0)
        s.setf(ios::showpoint);
    else
        s.unsetf(ios::showpoint);
    return s;
}
// вывод вещественного числа
double f=12.3;
// printf("%10.2f\n", f);
cout<<fmt_f(10,2)<<f<<"\n";
// printf("%10.0f\n", f);
cout<<fmt_f(10,0)<<f<<"\n";
// printf("%10f\n", f);
cout<<fmt_f(10)<<f<<"\n";
// printf("%f\n", f);
cout<<fmt_f()<<f<<"\n";
```


Примеры решений задач

Бендер-парламентер (map, vector, partial_sort)

```
#include <map>
#include <algorithm>
#include <functional>
#include <string>
#include <cstdio>
#include <cctype>
#include <vector>
using namespace std;

typedef map<string,int> map1;
typedef map1::value_type mval1;
typedef const mval1 *pval1;
struct word_less: public binary_function<pval1, pval1, bool>
{
    bool operator()(const pval1 v1, const pval1 v2)
    {
        if (v1->second<v2->second) return false;
        if (v1->second>v2->second) return true;
        if (v1->first.length()>v2->first.length()) return true;
    }
}
```

```

        if (v1->first.length() < v2->first.length()) return false;
        return v1->first < v2->first;
    }
};

struct copy_val: public unary_function<mval1, pval1> {
    pval1 operator() (const mval1 &v1)
    { return &v1;
    }
};

```

```

vector<pval1> words;
map<int, pval1> m1;

```

```

int main()
{ int i, ch;
  string s;
  while ((ch=getchar()) != EOF)
  { if (isalpha(ch))
      s+=tolower(ch);
    else
    { if (s.length() > 0)

```

```

        ++m1[s];
        s.clear();
    }
}
transform(m1.begin(), m1.end(), back_inserter(words), copy_val());
partial_sort(words.begin(), words.begin()
+10, words.end(), word_less());
for (i=0; i<10; i++)
    printf("%s\n", words[i]->first.c_str());
return 0;
}

```

Collection of Postage Stamps (bitset)

```

#include <iostream>
#include <bitset>
using namespace std;
bitset<1000001> b;
int main()
{ int i, k, v;

```

```

cin>>k;
b.reset();
b.set(0);
for(i=0;i<k;i++)
{  cin>>v;
   b|=b<<v;
}
cout<<b.count()<<endl;
return 0;
}

```

[Cymma](#)(vector, sort, upper_bound)

```

#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    int n,k,s,a,i,maxlen,item,p,newlen;
    cin>>n>>k;

```

```

vector<pair<int,int> > sum(n+1);
sum[0]=make_pair(0,0);
s=0;
for(i=1;i<=n;++i)
{ cin>>a;
  s+=a;
  sum[i]=make_pair(s,i);
}
sort(sum.begin(),sum.end());
item=maxlen=0;
for(i=0;i<=n;i++)
{ if(i>0 && sum[i].first==sum[i-1].first) continue;
  p=upper_bound(sum.begin(),sum.end(),make_pair(sum[i].fir
st+k,n+1))-sum.begin();
  if(p>0 && sum[p-1].first==sum[i].first+k &&
    (newlen=sum[p-1].second-sum[i].second)>0)
  { if(maxlen<newlen)
    { maxlen=newlen;
      item=sum[i].second+1;
    }
    else if(maxlen==newlen)

```

```
        item=min(item,sum[i].second+1);
    }
}
cout<<item<<" "<<maxlen<<endl;
return 0;
}
```