Объектно-ориентированное и системное программирование

КОНСПЕКТ ЛЕКЦИЙ

электронная версия от 28.04.16

Язык си++

Предисловие

Это попытка изложить основные особенности одного из сложнейших и мощнейших языков программирования в небольшом курсе. Практически все возможности си++ так или иначе рассмотрены. Из фундаментальных тем лишь некоторые возможности по стандартам от 2011 и 2014 годов не вошли в рамки курса. Кроме того, приводятся отличия си от си++. Изложение материала не претендует на всеохватывающую полноту — цель курса изложить максимум практических возможностей, избегая архаизмов и дублирующих более длинных конструкций. В следствие того, что до знакомства си++ читатели могли познакомиться с основами таких широко распространенных в сфере образования языков как паскаль и бэйсик, иногда будут приводиться соответствующие иллюстрационные ссылки.

Это пособие — результат многолетней работы со студентами кафедры «Моделирование систем и информационные технологии» Российского государственного технологического университета имени К. Э. Циолковского. Автор пользуется случаем выразить благодарность тем из них, кто своей активной работой так или иначе способствовал появлению этого учебного пособия.

Материал курса предназначен для слушателей, имеющих предварительное знакомство с объектно-ориентированным программированием, например, языком паскаль. Из-за жёстких временных рамок часть материалов излагается “с опережением”, т. е. приводятся примеры, теоретические объяснения к частям которых можно найти только в следующих разделах — общий объем материалов совсем невелик и автор рассчитывает на 2–3 кратное прочтение.

Все конструкции си++ приводятся на жёлтом фоне. Синим цветом выделяются объекты файловой системы и тексты для связанных с си++ программных средств.

Общая структура программы

Текст программы — это последовательность строк, состоящих из символов. Символы делятся на значащие и незначащие. К первым относятся 53 буквы, 52 английские (строчные и заглавные различаются!) и подчёркивание, десятичные цифры и 29 специальных знаков: `! "#%&'()*+,-./:;<=>?[\]^{|}~`. Ко вторым — все прочие, например, буквы алфавита русского языка. Лексемы, минимальные единицы языка со смыслом, делятся на типичные для большинства языков программирования категории: служебные слова, идентификаторы, знаки операций, разделители и литералы.

Служебные слова [всего — 63]

`asm`	`auto`	`bool`	`break`
`case`	`catch`	`char`	`class`
`const`	`const_cast`	`continue`	`default`
`delete`	`do`	`double`	`dynamic_cast`
`false`	`else`	`enum`	`explicit`
`extern`	`export`	`float`	`for`
`friend`	`goto`	`if`	`inline`
`int`	`long`	`mutable`	`namespace`
`new`	`operator`	`private`	`protected`
`public`	`register`	`return`	`reinterpret_cast`
`short`	`signed`	`sizeof`	`static`
`static_cast`	`struct`	`switch`	`template`
`this`	`throw`	`true`	`try`
`typedef`	`typeid`	`typename`	`union`
`unsigned`	`using`	`virtual`	`void`
`volatile`	`wchar_t`	`while`	

Лексемы из специальных знаков [всего — 50]

`!`	`!=`	`#`	`##`	`%`	`%=`	`&`	`&&`
`&=`	`(`	`)`	`*`	`*=`	`+`	`++`	`+=`
`,`	`-`	`--`	`-=`	`->`	`->*`	`.`	`.*`
`...`	`/`	`/=`	`:`	`::`	`<`	`<<`	`<<=`
`<=`	`=`	`==`	`>`	`>=`	`>>`	`>>=`	`?`
`[`	`]`	`^`	`^=`	`{`	`|`	`|=`	`||`
`}`	`~`						

Разделители — это пробелы, концы строк, табуляции или комментарии.

Идентификаторы или имена программных объектов должны состоять только из букв и цифр и начинаться с буквы. Все идентификаторы должны быть объявлены до их использования — исключением являются только метки для `goto`. Поэтому практически все программы начинаются с включения в свой текст дополнительной информации: либо в виде файлов-заголовков с расширением, как правило, `h` (от слова header), `hpp` или `hxx`, либо в виде стандартизированных заголовков — эта информация состоит из объявлений имён функций, типов и т. п.

Выполнение любой программы начинается с функции по имени `main`. Пример программы, печатающей фразу “Hello, World!”.

```
#include <iostream>
using namespace std;
main() {
  cout << "Hello, World!" << endl;
}
```

Листинг hello.cpp

⟨`iostream`⟩ — это название стандартного заголовка, содержащего описание имён `cout` и `endl`. Угловые скобки, в которые заключено имя, означают, что затребован стандартный ресурс системы программирования: либо файл, расположенный в подкаталоге `include` каталога этой системы, либо стандартный заголовок. Если заключать имя не в угловые скобки, а в кавычки, то сначала ищется файл в текущем (рабочем) каталоге. Под именем файла для команды `#include` понимается расширенное имя файла, т. е. собственно имя и путь доступа к нему. Например, строка `#include` ⟨`sys/stat.h`⟩ включает в текст программы файл `stat.h` из подкаталога `include/sys` каталога системы программирования, а строка `#include "headers/timer.h"` включает в текст программы файл `timer.h` из подкаталога `headers` текущего каталога.

Идентификаторы стандартной библиотеки си++ локализованы в пространстве имён (namespace)[1] `std` (standard — стандартное), а директива `using namespace std;` подключает это пространство имён к текущему[2].

Имя `cout` (channel out) — это объектная переменная-поток вывода — все, что заносится в неё операцией `<<` , печатается на экране. Имя `endl` (end line) означает переход на следующую строку.

Программа компилируется в три этапа (первые два проходят обычно параллельно, конвейером):

1) преобразование текста программы препроцессором;
2) собственно компиляция в машинный код — получение объектного модуля;
3) компоновка — связывание модуля с указанными и стандартными ресурсами для получения исполнимой программы.

Препроцессор позволяет использовать макросы, константы, условную компиляцию и некоторые другие возможности. Команды препроцессора — это строки программы, начинающиеся с символа #.

После обработки препроцессором текст программы преобразуется в последовательность деклараций.

Комментарием считается текст, заключённый между знаками `/*` и `*/` или знаками `//` и конец строки. Комментарии допустимы между любыми двумя лексемами, т. е. там же, где допустимы пробелы или концы строк.

В тексте программы можно выделить область для глобальных объектов — это текст программы вне фигурных скобок. Локальная область — это участок программы в фигурных скобках — в неё могут вкладываться другие локальные области. Переменные, определённые в локальной области, недоступны в объемлющих её областях. Пример — напечатается 1 и 2.

```
#include <iostream>
using namespace std;
int i = 1;
main() {
  int j = 2; {
  int k = 3;
  }
  cout << i << endl << j << endl;
  //cout << k << endl; //ошибка: k не существует
}
```

Листинг globloc.cpp

Кроме переменных и функций можно описывать типы данных. Описания типов и переменных можно чередовать с операторами. Операторы языка можно использовать только в теле функций — в глобальной части программы их использовать нельзя. Не допускается определять функцию в теле другой функции, т. е. вложенные функции запрещены. Операторы должны заканчиваться символом `;`.

Будем называть *операциями* функциональные отношения языка. Аргументы операций — это *операнды*. Выражения составляются согласно классической инфиксной математической нотации. Каждое выражение имеет значение, соответствующие некоторому типу. Выражение, заканчивающееся `;`, является *оператором*. Операторы не имеют значений. Пример синтаксически правильного, но бессодержательного оператора — `1+1;`.

Иерархия типов

I. Скалярные: целые, перечисления и вещественные;

[1] Пространства имён — это расширение языка, введение которого сделало си++ не совместимым со своими прежними вариантами.

[2] Эффект похож на оператор WITH паскаля или бэйсика.

II. Составные: массивы, комбинированные (структуры, объединения и классы), функции;
III. Указатели.

Структуры, объединения, классы и перечисления называются также *типами, определяемыми пользователем*. Для этих типов можно определить набор операций.

Ко всем типам данных применимы две унарные операции:

- `&` — получение адреса: если `v` — имя переменной типа `T`, то `&v` — её адрес типа `T*`, т. е. тип адреса — это указатель;
- `sizeof` — операция получения размера области памяти для данных заданного типа: если `u` — переменная некоторого типа или тип, то `sizeof(u)` — это размер памяти в байтах, необходимой для хранения любого объекта этого же типа.

Ко всем типам применима также бинарная операция следования (`,`). Если `e1` и `e2` — выражения, то запись `e1, e2` означает, что надо вычислить сначала `e1`, а затем `e2`, т. е. результатом операции будет `e2`.

Ко всем скалярным типам и типам-указателям применимы шесть стандартных операций сравнения (`==`, `!=`, `<`, `>`, `<=`, `>=`).

Операция присваивания (`=`) применима ко всем типам, кроме массивов и функций. Для начинающих программистов характерна *необнаруживаемая компилятором* ошибка писать `if (a = b)` вместо `if (a == b)`[3].

Целые типы

Определяются служебными словами `bool`, `char`, `short`, `int`, `long`, `wchar_t`. Верно, что `sizeof(char)` ⩽ `sizeof(short)` ⩽ `sizeof(int)` ⩽ `sizeof(long)` ⩽ `sizeof(long long)`[4], `sizeof(bool)` ⩽ `sizeof(long)`, `sizeof(char)` ⩽ `sizeof(wchar_t)` ⩽ `sizeof(long)`.

Логические значения *истина* и *ложь* имеют соответственно числа 1 и 0, которым соответствуют служебные слова `true` и `false`. Любое ненулевое значение считается истинной.

Перед этими словами, кроме `bool`, можно ставить уточняющие служебные слова `signed` (знаковый) или `unsigned` (беззнаковый). По умолчанию прибавляется слово `signed`[5]. Слова `short` и `long` можно рассматривать как прилагательные к слову `int`. Например, записи `signed long`, `long`, `long int` и `signed long int` эквивалентны.

Для IBM PC совместимых компьютеров установились следующие соотношения: `sizeof(char)` = 1, `sizeof(short)` = 2, `sizeof(int)` = `sizeof(long)`[6] = `sizeof(wchar_t)` = 4, `sizeof(long long)` = 8.

К данным целого типа применимы две унарные операции (`+` и `-`) и следующие бинарные:

`+` — сложение, например, `1 + 3` будет 4;

`-` — вычитание, например, `3 - 1` будет 2;

`*` — умножение, например, `2*2` будет 4;

`/` — деление, например, `7/3` будет 2, т. е. остаток отбрасывается;

`%` — остаток от деления, например, `6%2` будет 0, а `7%3` будет 1;

`<<` (сдвиг влево) — сдвиг двоичного представления числа заданное число раз влево, например, `1<<2` будет 4, а `3<<3` будет 24;

`>>` (сдвиг вправо) — сдвиг двоичного представления числа заданное число раз вправо, например, `1>>2` будет 0, а `61>>1` будет 30;

`&` — поразрядное логическое *И*, например, `61&35` будет 33;

`|` — поразрядное логическое *ИЛИ*, например, `61 | 35` будет 63;

`^` — поразрядное логическое *исключающее ИЛИ*, например, `61^35` будет 30;

`~` — поразрядное логическое отрицание, результат этой операции зависит от того, к какому типу данных она применяется, например, `(unsigned char) ~1` будет 254;

`&&` — логическое *И*, например, `61&&35` будет 1, а `55&&0` будет 0;

`||` — логическое *ИЛИ*, например, `61 || 35` и `55 || 0` будут равны 1;

`!` — логическое *НЕ*, например, `!61` будет 0, а `!0` будет 1.

Присваивание является операцией. Значение этой операции — это значение левого операнда после присваивания. Левым операндом в операции присваивания может быть только величина, которая имеет адрес и значение которой можно изменять. К данным целого типа возможно применять следующие операции присваивания:

`=` — простое присваивание, например, значением выражения `v1 = v2 = 3 + 5` будет число 8, кроме того, переменным `v1` и `v2` будет присвоено значение 8;

`+=` — сложение и присваивание (увеличение на заданное значение), например, `v += 2` эквивалентно записи `v = v + 2`;

---

[3] Использование двойного знака равенства для сравнения вызвано тем, что сравнения встречаются в среднем до трёх раз реже присваиваний, а авторы си стремились создать максимально лаконичный язык.

[4] Этот тип отсутствует в стандарте си++ 2003 года, но описан в стандарте си 1999 года и поддерживается, в частности, GNU си++.

[5] Для `char` это определяется настройкой компилятора.

[6] 8 — в 64-разрядных средах.

`-=` — вычитание и присваивание (уменьшение на заданное значение), например, `v -= 2` эквивалентно записи `v = v - 2`;

`*=` — умножение и присваивание (увеличение в заданное число раз), например, `v *= 2` эквивалентно записи `v = v*2`;

`/=` — деление и присваивание (уменьшение в заданное число раз), например, `v /= 2` эквивалентно записи `v = v1/2`;

`%=` — остаток от деления и присваивание, например, `v %= 2` эквивалентно записи `v = v%2`;

`<<=` — сдвиг двоичного представления числа заданное число раз влево и присваивание, например, `v <<= 2` эквивалентно записи `v = v<<2`;

`>>=` — сдвиг двоичного представления числа заданное число раз вправо и присваивание, например, `v >>= 2` эквивалентно записи `v = v>>2`;

`&=` — поразрядное логическое *И* и присваивание, например, `v & = 2` эквивалентно записи `v = v&2`;

`|=` — поразрядное логическое *ИЛИ* и присваивание, например, `v |= 2` эквивалентно записи `v = v | 2`;

`^=` — поразрядное логическое *исключающее ИЛИ* и присваивание, например, `v ^= 2` эквивалентно записи `v = v ^ 2`;

`++` — увеличение на 1 (*инкремент*), например, `++v` (*префиксный инкремент*) значает то же, что и `v += 1`, а `v++` (*постфиксный инкремент*) эквивалентно записи `t = v, v += 1, t`, т. е. значением выражения `v++` является значение `v` до увеличения;

`--` — уменьшение на 1 (*декремент*), например, `--v` означает то же, что и `v = v - 1`, а `v--` эквивалентно записи `t = v, v = v - 1, t`, т. е. значением выражения `v--` является значение `v` до уменьшения;

Тернарной операцией является `?:` — условная операция, например, если `e1`, `e2` и `e3` — выражения, то запись `e1 ? e2 : e3` означает вычислить `e1`, если оно истинно, то значение операции — это `e2`, иначе — `e3`. Выражения `e2` и `e3` могут быть любого типа.

Константы-литералы целого типа можно записывать в виде 10-х, 8-х или 16-х чисел. 16-е числа должны начинаться с `0x` или `0X`, 8-е — с 0. Например, `0xffff` = `0XFFFF` = `0xFFFF` = `0XffFf` = `0177777` = 65535. Кроме того, константы (литералы) типа `char` можно записывать как символы в апострофах: `'D'` = 68, `'@'` = 64, `'0'` = 48. Символ обратная косая черта (backslash, `\`) является специальным, предназначенным для записи невидимых, управляющих символов. Его можно использовать в двух формах:

1) после него следуют три цифры 8-го числа или две цифры 16-го с префиксом `x` или `X`, например, `'D'` = `'\104'` = `'\X44'` = 68, `'\x0A'` = `'\012'` — код перехода на следующую строку;

2) после него следует символ из определённого в стандарте набора. Отдельные, чаще всего используемые символы из этого набора:

`'\n'` = `'\x0A'` = `'\012'` — код перехода на следующую строку;

`'\\'` — сам символ \;

`'\''` — апостроф (одинарная кавычка);

`'\"'` — (двойная) кавычка;

`'\0'` = 0;

`'\t'` = `'\x09'` — знак табуляции.

Для указания на беззнаковость в литералах можно использовать суффикс `U` или `u`, а для указания на `long` (`long long`) можно использовать — `L` или `l` (`LL` или `ll`), например, `11U`, `12L`.

Тип `wchar_t` введён для поддержки 16/32-разрядных кодировок, в частности, Unicode. Для этого типа существуют аналоги функций для `char`. Литералы этого типа должны начинаться с `L` или `l`.

Перечисления

Тип перечисления похож на модификацию целого типа, хотя, строго говоря, целым не является. Описание перечисления — это описание множества констант типа `int`.

Значение каждой константы, которой явно не присваивается целая величина, равно значению предыдущей константы плюс 1. Нумерация начинается с нуля.

Переменной типа-перечисления можно присваивать константы из определения её типа.

Пример — печать 4-х строк: Su, 2, We, 5.

```
#include <iostream>
using namespace std;
enum Week {Su, Mo, Tu, We, Th, Fr, Sa};
ostream& operator<<(ostream& out, const Week& v) {
  const char *names[] =
    {"Su", "Mo", "Tu", "We", "Th", "Fr", "Sa"};
  return out << names[v];
}
```

```
enum RGB {Red = -1, Green, Blue = 2};
main() {
  Week Day = We; //3
  int temp = Tu + Green + Day; //5
  cout << Su << endl << Blue << endl;
  cout << Day << endl << temp << endl;
}
```

Листинг enum.cpp

Вещественный тип

Определяется служебными словами `float` и `double`. Верно, что `sizeof(float)` $\leqslant$ `sizeof(double)` $\leqslant$ `sizeof(long double)`.

Для IBM PC совместимых компьютеров установились соотношения: `sizeof(float)` = 4, `sizeof(double)` = 8, `sizeof(long double)` = 10.

К данным вещественного типа применимы операции, что и к данным целого типа кроме поразрядных и `%`, `%=`, `++`, `--`. Операции `/` и `/=` делят без отбрасывания остатка.

Константы-литералы вещественного типа задаются мантиссой и, если необходимо, порядком. Порядок (целое число) нужно указывать после буквы `e` или `E`. Если в записи числа отсутствует порядок, то необходимо ставить 10-ю точку в мантиссе. Например, `.5` = `0.5` = `5E-1`, `-4.` = `-4.0` = `-4e0` = `-40e-1` = `-.4E1`.

Суффиксы `l`, `L`, `f` и `F` позволяют точно специфицировать типы `long double` и `float` соответственно, например, `1e1l`.

Типы для указателей

Указатель описывается именем и типом объекта, на который он может указывать, например, декларация `int *p1, *p2;` описывает два указателя `p1` и `p2` на объекты целого типа. К указателям можно применять операцию обращения по адресу, `*`, которую иногда называют *разыменование*.

Есть только одна константа типа указатель, 0, пустой указатель. Тип указателей можно преобразовывать в целый и обратно.

К указателям можно применять операции `+` и `-`, второй аргумент `+` должен быть целым, а второй аргумент `-` либо целым, либо указателем. Если `p`, `q` — указатели на данные типа `T`, а `i` — целое, то `p ± i` — это указатель на `i`-е последовательное значение типа `T`, начиная от `p`, т. е. новый адрес получается из старого добавлением/вычитанием `i*sizeof(T)`, а `p - q` — это расстояние, целое число — количество элементов типа `T` между `p` и `q`.

Пример — напечатает 78.

```
#include <iostream>
using namespace std;
main() {
  int i1 = 7, i2 = 8, *p = &i1, distance;
  distance = &i2 - p;
  cout << *p << *(p + distance) << endl;
}
```

Листинг pointers.cpp

Массивы

Массив описывается типом, количеством элементов и идентификатором, например, декларация `float a[7];` описывает массив `a` из 7 вещественных чисел, а декларация `int b[7][5];` описывает массив `b` из 7 массивов из 5 целых чисел, т. е. матрицу 7×5. В некоторых случаях в декларации массива можно опускать количество элементов. Доступ к элементам массива осуществляется при помощи *операции* выделения элемента, `[]`. Следует отличать квадратные скобки в операции выделения элемента и в декларациях массивов, где они не являются операцией. Нумерация элементов массива начинается с нуля. Других операций для массивов нет.

Пример — расчёт определителя матрицы $A = \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$.

```
#include <iostream>
using namespace std;
main() {
  int A[2][2];
  A[0][0] = 0;
  A[0][1] = 1;
  A[1][0] = 2;
  A[1][1] = 3;
```

```
  cout << "det(A)="
    << A[0][0]*A[1][1]-A[0][1]*A[1][0] << endl;
} //-2
```

Листинг array.cpp

Тип массив тесно связан с указателями, например, в предыдущей программе запись `A[1][1]` можно заменить на `*(*(A+1)+1)`, запись `A[0][0]` — на `**A`, `A[0][1]` — на `*(*A+1)`, `A[1][0]` — на `**(A+1)`. Таким образом, здесь `A` ведет себя как константный указатель на последовательность, компоненты которой — это пары целых чисел.

Рассмотрим связь указателей и массивов более подробно на примерах. Сначала рассмотрим описание трёх переменных

`int *A, B[7], C[7];`,

где `A` — это указатель на целое число (тип — `int*`), `B` и `C` — массивы из 7 целых чисел (тип — `int[7]`). `B` или `C` во многих случаях можно рассматривать как указатели-константы — практическая разница между массивами и указателями обнаруживается только в операциях `sizeof`, `typeid` и `&`.

`A` можно присвоить значение `B` или `C`. После присваивания `A = C;` `A` становится почти синонимом `C`.

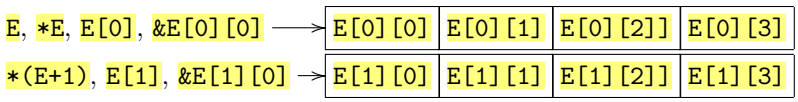
Все указатели совместимы и их можно, как минимум, сравнивать друг с другом. Очевидно, что результат сравнения двух массивов, например, `B` и `C`, будет всегда ложью.

Операция выделения элемента массива может применяться к любому указателю — она лишь удобная форма записи операции обращения по адресу. В общем случае, для любого указателя или массива `X` верно, что `X[0]` = `*X` и `X[n]` = `*(X+n)`.

Рассмотрим теперь описание следующих четырёх переменных

`int **D, E[2][4], *F[4], (*G)[4];`,

где `D` — это указатель на указатель на целое число (тип `int**`). `E` — это массив из двух массивов, состоящих из 4 целых чисел каждый. Можно сказать, что `E` задает матрицу из двух строк и 4-х столбцов (тип `int[2][4]`). `F` — это массив из 4-х указателей на целые числа (тип `int *[4]`). `G` — это указатель на массив из 4-х целых чисел (тип `int (*)[4]`). `D` и `F` не совместимы с `E` и `G` по присваиванию.



Применение операции обращения по адресу к `D`, `E`, `F` и `G` даст следующие результаты: `*D` и `*F` — это указатели на целое число, `*E` — указатель на нулевую строку массива `E`, `*G` — это массив из 4-х целых чисел. Верны, в частности, соотношения `D[n][m]` = `*(*(D + n) + m)`, `E[0][5]` = `E[1][1]`, `E[n][m]` = `*(*(E + n) + m)`. Возможны присваивания `G = E;`, `G = E + 1;` и `D = F;`. Например, верно, что `E[1][2]` = `(E[1] + 1)[1]` = `(E + 1)[0][2]`.

Работа с 3-мерными массивами протекает по обозначенной схеме. Например, переменные `H` и `I`, описанные в декларации

`int H[2][5][4], (*I)[5][4];`

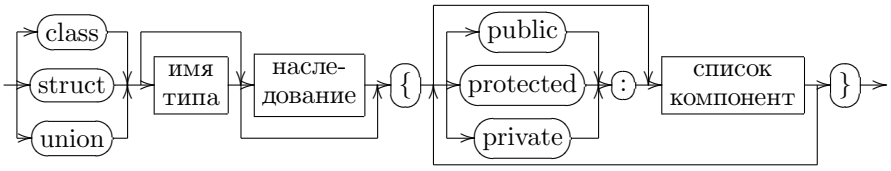
имеют соответственно тип массив размерности $2 \times 5 \times 4$ (`int[2][5][4]`) и тип указатель на массив размерности 5×4 (`int (*)[5][4]`). Можно `I = H;` и нельзя `H = I;`.

Комбинированные типы

Описываются при помощи служебных слов `struct` (записи или структуры), `union` (объединения) или `class` (классы). Структуры и объединения часто также называют классами. Переменные этих типов часто называют *записями*[7]. Например, декларация

`struct Person {char age, name[32];};`

описывает тип `Person` с полями `name` и `age`. В общем случае описанию комбинированного типа соответствует следующая синтаксическая диаграмма:



Наследоваться могут только структуры и классы, но не объединения.

К данным комбинированного типа могут применяться следующие операции: `.` (выделение элемента структуры), `.*` (доступ к указателю на член класса) и `=` (присваивание).

К данным типа указатель на класс могут применяться операции `->` (выделение элемента указываемой структуры) и `->*` (доступ к указателю на член класса указываемого класса). Операция `->` — это сокращение, например, если `p` — это указатель на тип-структуру `Person`, то `p->name` эвивалентно `(*p).name`.

[7] Структуры и, в меньшей степени, классы и объединения соответствуют строкам таблиц реляционных БД.

С классами можно использовать операцию `::` (спецификация области видимости) — для ссылки на тип.

Классы и перечисления с разными именами и идентичными описаниями считаются различными.

В объединениях все поля начинаются с одного адреса в памяти[8]. Например,

```
#include <iostream>
using namespace std;
union {
    int a;
    int b;
    float c;
} t;
main () {
    t.a = 5;
    t.b = 7;
    cout << t.a << t.b << endl; //77
    cout << t.c << endl; // некое вещ. число
}
```

Листинг union.cpp

Декларации

Делятся на *объявления* и *определения*. Объявления вводят имена для используемых в программе объектов: шаблонов, типов, функций, переменных. Определения полностью определяют программный объект, в частности, выделяют память для переменных и функций и связывают её с объявленным именем. Можно говорить и об определении типа или шаблона, если соответствующая декларация полностью описывает тип или шаблон. Присутствие определения до использования имени необязательно.

Объявления часто равнозначны определениям. Любое определение является объявлением. Каждый программный объект должен иметь равно одно определение и может иметь сколько угодно объявлений.

Декларации могут начинаться служебными словами: `typedef`, `extern` (только объявления), `register`, `auto` , `static`, `friend`, `virtual`, `inline`, `mutable`. Слова `register`, `extern`, `auto` и `static` называются *спецификаторами класса памяти*. Слова `register` и `auto` можно использовать только в локальной части программы: для переменных локальной части `auto` используется по умолчанию. Переменные с классом памяти `register` не имеют адреса. Служебные слова `friend`, `virtual` и `inline` можно использовать только с функциями — их называют *спецификаторами функций*. Слово `friend` используют только с функциями-нечленами, а `virtual` — только с функциями-членами того класса, в котором они объявлены. Служебное слово `typedef` используется для определения типа. Слова `virtual`, `mutable`, `friend` используются только внутри классов.

Далее может следовать спецификатор константности `const` или изменчивости `volatile`. Далее сам тип: целый, вещественный, определяемый пользователем, пустой (`void`). Тип `void` означает отсутствие типа и применяется, как правило, с функциями и указателями. Объектов типа `void` существовать не может.

Тип конкретного объекта уточняется значками `()`, `[]`, `*`, `&`. Пара `()` специфицирует функцию, `[]` — массив, значок `*` — указатель, а значок `&` применяется с объектом, который нужно связать с заданной областью памяти. Объекты, специфицированные значком `&`, называют *ссылками*. Этот значок всегда должен стоять ближайшим слева к объекту. Использование ссылок позволяет, в частности, вводить синонимы переменных. Вместе со значком `*` можно использовать квалификаторы `const` и `volatile`.

Результат функции не может иметь тип массив или функция. В объявлении списка параметров функции сами параметры можно *не указывать* — нужно указывать только их типы.

В одной декларации может быть сколько угодно объявлений функций и переменных. Декларируемые объекты отделяются друг от друга запятыми.

Глобальным и статическим переменным по умолчанию приписываются нулевые значения. В описаниях переменных целого, вещественного и указательного типов, а также массивов и структур из элементов этих типов можно использовать инициализацию, т. е. задание этим переменным после знака `=` предопределённого значения. В случае массива или класса список разделённых запятыми значений заключается в фигурные скобки. Константы и ссылки инициализировать необходимо. Объекты классов с явным конструктором, виртуальными функциями, имеющие класс-предок или с защищёнными нестатическими данными нельзя инициализировать таким способом.

Примеры деклараций.

1) Определение типа `CPLint` равнозначного `long * const` — константные указатели на длинное целое.

```
typedef long * const CPLint;
```

2) Определение `i` — указателя на целую константу, объявление `g` — функции с одним целым аргументом, возвращающей целую константу, определение `j` — целой константы 2.

```
const int *i, g(int), j = 2;
```

[8] Это функциональный эквивалент вариантной части записи паскаля.

3) Определение ii — целой переменной, определение массива a из 2 указателей на целые числа, 1-й элемент a — это указатель на ii, 2-й — 0, объявление f — функции с целым аргументом, возвращающей целое число, определение b — указателя на массив из 2-х целых.

```
int ii, *a[2] = {&ii}, f(int), (*b)[2];
```

Таким образом, тип читается сначала направо от декларируемого объекта, а затем налево. Скобки позволяют устанавливать нужный приоритет.

4) Объявление pl — константного указателя на длинное целое.

```
extern CPLint pl;
```

5) Объявление класса Point — типа для точки на экране дисплея, переменной point этого класса и указателя pp на объекты этого же класса.

```
class Point {
  static int n; //количество точек
  double x, y;  //координаты точки
public:
  Point(double x0 = 0, double y0 = 0): x(x0), y(y0) {n++;}
    //создание точки
  virtual ~Point() {n--;}    //уничтожение точки
  double get_x();       //получение координаты x, объявление
  double get_y() {return y;} //для y - определение
  virtual void move(double, double);
  friend bool collision(Point*, Point*);
};
inline double Point::get_x() {return x;}
  //получение координаты x, определение
void Point::move(double dx, double dy) {
   x += dx;
   y += dy;
} //перемещение точки
bool collision(Point *o1, Point *o2) {
  return o1->x == o2->x && o1->y == o2->y;  //проверка на
}    //совпадение двух точек, заданных указателями на них
int Point::n; //определение n
Point point, *pp;
```

Точка point будет иметь координаты (0,0), а поле n установится в 1.

6) Определение n — целой переменной и инициализация её числом 7, определение указателя на целую переменную p и инициализация его адресом n, определение ссылки k — синонима n.

```
int n = 7, *p = &n, &k = n;
```

7) Определение функции f, меняющей знак своего целого аргумента, — соответствует объявлению из пункта 3.

```
int f(int k) {return -k;}
```

8) Определение структуры Date для представления дат. Определения нескольких переменных-дат

```
struct Date {
  int day, month, year;
} VEDay = {7, 5, 1945}, XDay, VernalEquinox = {21, 3};
```

9) Определения переменных: массивов a1 из трёх символов '1', 'a' и '\0' и a2 из трёх символов '9', 'x' и '\0', символа x и символа c с инициализирующим значением 'c'.

```
char a1[3] = {'1', 'a'}, a2[] = "9x", x, c = 'c';
```

Если в определении массива отсутствует число, задающее количество элементов в последней размерности, то это число определяется по инициализатору, т. е. в этом случае инициализатор необходим.

10) Определение словаря из 4 записей.

```
struct DictElem {const char *word; unsigned count;}
  dictionary[] =
    {{"pea", 7}, {"peach", 4}, {"pear", 1}, {"plum"}};
```

Операторы

Составной оператор или *блок* — особенный — после него знак `;` не ставится. Это заключение в фигурные скобки других операторов, деклараций и блоков.

Другие операторы — это оператор *выражения*, *перехода*, *условный*, *цикла с предусловием*, *цикла с постусловием*, *цикла с параметром*, *завершения*, *продолжения*, *переключения*, *возврата*. Перед каждым оператором можно поставить идентификатор-метку, которой можно воспользоваться в операторе перехода. После метки, перед помечаемым оператором или другой меткой необходимо поставить `:`.

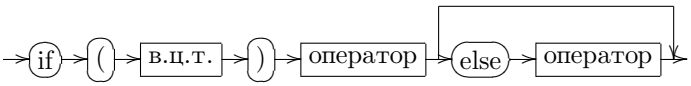
Оператор выражения — это выражение, за которым следует точка с запятой. Примеры.

```
a == 7 ? b = 7 : a += b -= 8;
a = b = 5;
; // пустое выражение
```

Оператор безусловного перехода — это последовательность из `goto`, метки и `;`. Выполнение этого оператора заключается в передаче управления на оператор, помеченный указанной в нем меткой. Нельзя переходить из одной функции в другую, а также нельзя перескочить через декларацию. Использование оператора `goto`, кроме весьма редких ситуаций[9], считается нежелательным. Вместо него лучше использовать операторы условный, циклов, переключения, завершения, продолжения и т. п.

```
main () {L:goto L;} //эта программа никогда не остановится
```

Условный оператор описывается следующим синтаксисом.



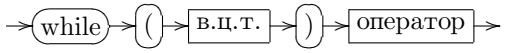
Выражение целого типа — это условие. Если в условный оператор вкладывается условный оператор и т.д., то `else` всегда относятся к ближайшему слева свободному `if`. Примеры.

```
if (5) if (0); else if (4) v=1; else v=2; //v=1
if (5) //это предыдущий оператор, записанный структурно
  if (0);
  else if (4)
    v=1;
  else
    v=2;
```

Декларации внутри блока можно рассматривать и как операторы. Использование декларации в составе любого оператора, кроме составного, автоматически порождает вокруг себя локальную область. Примеры, показывающие, как интерпретируется декларация, вложенная в оператор.

```
if (1) int i;   //означает то же, что и
if (1) {int i;}
if (int i = 5) i = 3; //означает то же, что и
{int i = 5; if (i) i = 3;}
```

Оператор цикла с предусловием (цикл `while`) имеет следующий синтаксис.

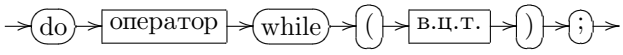


Этот оператор можно выразить через операторы `if` и `goto`, например, оператор `while (i < 5) cout << ++i;` эквивалентен
`L: if (i < 5) {cout << ++i; goto L;}`.

Пример — печать чисел от 1 до 9.

```
int i = 1;
while (i < 10)
    cout << i++;
```

Оператор цикла с постусловием (цикл `do`) имеет следующий синтаксис.



Этот оператор также можно выразить через операторы `if` и `goto`, например, оператор `do cout << ++i; while (i < 5` эквивалентен
`L: cout << ++i; L: if (i < 5) goto L;`.

Таким образом, оператор в цикле `do` выполняется хотя бы один раз[10].

Пример — напечатает 10.

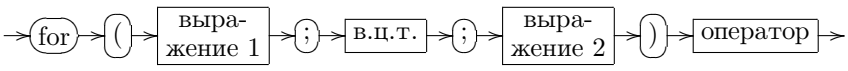
```
int i = 10;
do
    cout << i++;
```

[9] Например, для выхода из вложенных циклов и переключений.

[10] Создатель си++, Бьярне Страуструп, считает это недостатком.

```
while (i < 10);
```

Оператор цикла с параметром (цикл `for`) определяется следующим синтаксисом.



Оператор `for` тоже можно свести к `if` и `goto` или к `while`, например, оператор `for (int i = 1; i < 10; i++) cout <` эквивалентен `{int i = 1; L: if (i < 10) {cout << i; i++; goto L;}}`, т. е. это вариант оператора цикла с предусловием

Особой формой оператора цикла с параметром является бесконечный цикл вида `for (;;) оператор`.

Пример — распечатка чисел от 1 до 9 — функциональный эквивалент примера с `while`.

```
for (int i = 1; i < 10; i++)
    cout << i;
```

Оператор завершения — это `break;`. Его можно использовать только в операторах цикла и в операторе переключения. Он передает управление за пределы содержащего его оператора.

Пример — поиск числа 5 в векторе `B` из 10 компонент, возврат значением переменной `i` индекса компоненты `B`, равной 5.

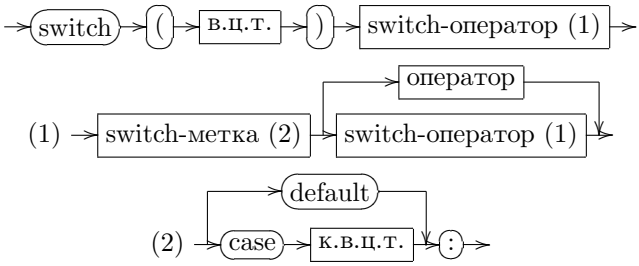
```
for (i = 0; i < 10; i++)
   if (B[i] == 5)
     break;
```

Оператор продолжения — это `continue;`. Его можно использовать только в операторах цикла. Он передает управление на следующий виток цикла.

Пример — заполнение нулевых компонент вектора `B` из 10 компонент единицами.

```
for (int i = 0; i < 10; i++) {
   if (B[i])
     continue;
   B[i] = 1;
}
```

Оператор переключения[11] с синтаксисом по следующим трём диаграммам — синтаксически самый сложный в си++.



Switch-оператор отличается от обычного оператора только тем, что его кроме обычной метки для `goto` можно ещё пометить switch-меткой — оператор переключения сводится к созданию локальной области для специальных меток. Все switch-метки в одном операторе-переключателе должны быть различны. Допускается только одна метка `default`.

Выполнение этого оператора заключается в вычислении значения в.ц.т. и в переходе на равное ему константное в.ц.т.[12] после `case` или, если такого к.в.ц.т. нет, то в переходе на специальную switch-метку `default`, или, если такой метки нет, то переход к следующему оператору. В операторе переключения характерно использовать оператор `break`.

Пример — должен напечатать 0-5-1-1-641-5-641-780-80--.

```
#include <iostream>
using namespace std;
main() {
   for (int i = 0; i <= 9; i++) {
      switch (i) {
         case 1:
         case 5:
            cout << 5;
            break;
         case 6:
         case 4:
            cout << 64;
```

[11] Этот оператор похож на оператор оператор бэйсика ON GOTO.

[12] В нем можно использовать только величины, значения которых известны в момент компиляции.

```
         case 3:
         default:
            cout << 1;
            break;
         case 9:
            if (0)
         case 7:
              cout << 7;
            else
              break;
         case 8:
            cout << 8;
         case 0:
            cout << 0;
      }
      cout << '-';
   }
   cout << endl;
}
```

Листинг switch.cpp

Оператор возврата — это служебное слово `return`, после которого для функций с типом, отличным от `void`, должно следовать выражение.

Использование этого оператора необходимо для осмысленного использования функций, возвращающих значение, т. е. имеющих тип отличный от `void`. Выполнение оператора возврата заключается в присвоении вычисляемой в выражении величины значению функции и выходе из функции. В функциях с типом `void` `return` может отсутствовать — здесь он нужен только для немедленного выхода из функции.

Пример — печатает 64.

```
#include <iostream>
using namespace std;
long power(unsigned x, char p) {
   switch (p) {
      case 0: return 1;
      case 1: return x;
      default: return (p%2 ? x : 1)*power(x*x, p/2);
   }
}
main() {
   cout << power(4,3) << endl;
}
```

Листинг return.cpp

Особый случай — это функция `main`, которая возвращает своё значение операционной системе. Её тип — `int`. Если тип указать явно, то будет необходимо использовать `return`. Если тип не указать, то она возвращает 0.

Спецификаторы деклараций

`typedef` (type definition — определение типа)

Декларация, начинающаяся с этого слова, — это определение имени типа. Использование `typedef` может повысить наглядность программ и сделать их более мобильными. Типы, соответствующие разным именам, но имеющие одинаковые typedef-описания считаются идентичными.

```
#include <iostream>
using namespace std;
typedef int array[12]; //описание типа array
typedef int*(*fp)(array); //тип fp - это указатель на функцию
         //с аргументом типа array, возвращающую указатель на
                                        //целое число типа int
array A;     //переменная типа array
inline int* func5(array A) {return &A[5];}
int* func2(array A) {return &A[2];}
fp func(char i) {
```

```
   switch (i) {
     case 0:
       return &func2; //значок & необязателен
     default:
       return func5;
   }
}
main() {
   for (int i = 0; i < 12; i++)
      A[i] = i*i;
   cout << "A[2]=" << *(*func(0))(A) << ' ';
   cout << "A[5]=" << *(*func(1))(A) << endl;
} //A[2]=4 A[5]=25
```

Листинг typedef.cpp

`static` (статический)

Этот спецификатор имеет разные назначения для переменных-членов классов, функций членов и нечленов классов, локальных и глобальных переменных. Программные объекты, описанные со словом `static`, называются статическими.

Если `static` применяется для переменной-нечлена класса, то он означает, что эта переменная существуют все время, пока выполняется программа. Если применить `static` к переменной, определённой в локальной области, то значение этой переменной будет храниться постоянно, не теряясь при выходе из этой локальной области. Последнее не означает, что к этой переменной можно получить доступ из области, содержащей данную: статические переменные недоступны за пределами своей локальной области. Параметры функции не могут быть статическими.

Пример — напечатается 17.

```
#include <iostream>
using namespace std;
int* xsum(int A[], int sA) {
   //возвращает указатель на сумму элементов
         //массива A, sizeof(A)=sizeof(int*)
   static int d;
   d=0;
   for (int i = 0; i < sA; i++)
      d += A[i];
   return &d;
}
main() {
   int a[5] = {2,3,5,7,11};
   cout << *xsum(a, 4) << endl;
}
```

Листинг static.cpp

Если описать глобальную переменную как статическую, то она станет недоступной для других модулей — это один из способов защиты абстракции. Аналогичным образом, применение `static` при объявлении функции-нечлена класса означает, что определение функции следует искать только в модуле, в котором есть это объявление.

Применение `static` к переменной-компоненте класса означает, что эта переменная является единственной для всех экземпляров данного класса. Например, если создано 10 экземпляров какого-либо класса, то для статических переменных этих 10 экземпляров будет выделено памяти столько же сколько и для одного экземпляра. Можно также говорить, что статические переменные-компоненты класса относятся не к экземплярам класса, как обычные переменные-члены, но к классу в целом. Использование статических переменных-компонент очень похоже на использование функций-членов.

Пример — напечатается 596799.

```
#include <iostream>
using namespace std;
struct X {
   static int a;
   int b;
} z1, z2;
int X::a = 5;
```

```
main() {
   cout << X::a;
   z1.b = 6;
   z2.b = 7;
   z1.a = 8;
   z2.a = 9;
   cout << X::a << z1.b << z2.b << z1.a << z2.a << endl;
```

Листинг class-static.cpp

Использование `static` при объявлении функции-члена класса означает, что эта функция может манипулировать только статическими данными класса.

`extern` (external — внешний)

Этот спецификатор используется при объявлении функций и переменных, не являющихся членами классов.

Если переменная объявлена с этим спецификатором, то это означает, что память под эту переменную выделять не надо, так как она выделена в другом месте. Таким образом, описание переменной с `extern` — это объявление имени глобальной переменной, связь которого с соответствующим местом памяти может происходить на этапе компоновки.

С объявлениями функций спецификатор `extern` используется по умолчанию. Он означает, что определение функции нужно искать во всех возможных модулях, а не только в текущем. Связь имени с кодом функции может происходить на этапе компоновки. Параметры функции не могут быть внешними.

`auto` (автоматический)

Этот спецификатор по умолчанию добавляется ко всем локальным переменным, не описанным явно как статические. Переменная с этим спецификатором называется автоматической — она существуют только то время, пока выполняются операторы, принадлежащие той же области, что и она сама. Память для автоматических переменных выделяется всякий раз, когда управление переходит в область, их содержащую; при передаче управления из этой области выделенная для них память освобождается. В примере, где описывалась статическая переменная `d`, эта переменная не может быть автоматической, так как в этом случае её адрес будет бессмысленным после выхода из содержащей её функции. Параметры функций — всегда автоматические. Служебное слово `auto` является избыточным и практически никогда не используется.

`register` (регистр)

Этот спецификатор уточняет способ размещения автоматических переменных: он *рекомендует* поместить их в самую быструю память компьютера, регистры. Регистры не имеют адреса. Количество регистров в компьютере ограничено (в IBM PC совместимых компьютерах есть только 14/28/120 байт регистровой памяти или 7/15 16/32/64-разрядных регистров) и часть из них используется для системных нужд, поэтому в регистры можно поместить лишь ограниченное количество переменных некоторых типов, размер (`sizeof`) которых не более 2/4/8 байт. Параметры функций можно описывать как регистры.

`inline` (в строку)

Этот спецификатор применяется только с функциями как членами, так и нечленами классов. Он *рекомендует* транслятору по особому организовывать обращение к функции. Такие функции должны определяться в каждом модуле (единице трансляции). Обычные функции состоят из кода, который храниться в памяти в единственном числе. Вызов обычной функции — это передача управления по адресу этого кода: использование имени обычной функции — это использование адреса её кода. Вызов inline-функции — это подстановка кода в точку вызова, т. е. код inline-функции может встретиться в программе много раз. Вследствие того, что для вызова inline-функций не требуется дополнительных действий, необходимых для вызова обычных функций, он происходит быстрее. Использование служебного слова `inline` оправдано только для очень маленьких и быстровыполняемых функций, так как время вызова функции не зависит от её размера и поэтому, чем больше время выполнения кода функции, тем меньше выигрыш от использования подстановки. Кроме того, использование подстановок может привести к неоправданному росту размера кода программы. Способ каждого вызова inline-функции определяется транслятором: он может вызвать такую функцию как подстановкой, так и как обычную функцию. Функции-члены класса, определённые в классе, считаются inline-функциями по умолчанию.

`mutable` (изменчивый)

Используется с членами классов, отменяет `const`, в частности, делает ненужным `const_cast` для таких членов.

Пример.

```
struct A {
   mutable int b;
   int c;
} b = {3, 4};
const A a = b;
main () {
   a.b = 7;
   //a.c = 8; //ошибка
```

```
    const_cast<int&>(a.c) = 8;
}
```

Листинг mutable.cpp

`const` (constant — константа)

Этот спецификатор, который наряду с `volatile` называют также квалификатором, позволяет описать участок памяти как неизменяемый. Такой участок требует задания своего содержимого в определении.

Массивы и функции часто ведут себя как константные указатели. Результат применения операции `&` к имени массива или функции — это значение самого указателя на массив или функцию, т. е. к ним применять эту операцию бессмысленно.

Пример.

```
#include <iostream>
using namespace std;
int A[] = {1,2,3}, *b, *const D = A;
int sqr(int a) {return a*a;}
main() {
    if ((int*)&A == A) cout << '1'; //&A имеет тип int(*)[3]
    if (&b == (int**)b) cout << '2';
    if ((int*)&D == D) cout << '3'; //отличие D от A
    if (&sqr == sqr) cout << '4';
    cout << endl;
} //14
```

Листинг const-addr.cpp

Использование `const` с массивом означает запрет на изменение всех компонент массива.

Спецификатор `const` можно использовать с формальным параметром функции. В частности, этот спецификатор предназначен для защиты от изменения фактических параметров, передаваемых по ссылке.

Пример.

```
#include <iostream>
using namespace std;
void ShowMessage(const char *msg1, char *msg2) {
    //msg1[0] = '!'; //так нельзя из-за const
    cout << msg1;
    msg2[0] = 'X';
    cout << msg2;
}
char msg[] = "ABCD ";
main() {
    msg[1] = '-';
    ShowMessage(msg, msg);
    cout << msg << endl;
} //A-CD X-CD X-CD
```

Листинг const-ptr.cpp

Этот спецификатор можно использовать с функциями-членами классов, после каждого объявления и определения формальных параметров. Такое его использование с функцией означает, что эта функция не может изменить значения компонент класса.

Пример.

```
class Date {
    int d, m, y;
public:
    int get_day() const;
    int get_month() const {return m;}
    int get_year() const {return y;}
    void set_date(int, int, int);
};
inline int Date::get_day() const {
    return d;
}
```

С функциями, тип результата которых не ссылка, использование `const` для указания на константность возвращаемого значения бессмысленно.

`volatile` (изменчивый)

Этот спецификатор позволяет описать участок памяти, как изменяющийся самопроизвольно. Это подавляет возможность временного хранения данных этого участка в более быстрой памяти и предотвращает связанные с этим ошибки.

Например, если внешняя переменная `timer` хранит текущее значение таймера компьютера, то её можно описать следующим образом:
`extern volatile unsigned long timer;`.

Спецификаторы адреса — ссылки

В декларациях можно использовать знак `&`, который здесь означает, что декларируемая величина будет связана с адресом величины, указанной в инициализирующем выражении. Знак `&` позволяет одним сущностям отождествляться с другими. Описание такого отождествления называется описанием ссылки, хотя это и вносит некоторую путаницу[13].

Пример, сопоставляющий переменную, ссылку и указатель на неё.

```
#include <iostream>
using namespace std;
int i = 5;
int &j = i;   //j отождествляется с i
    //j - ссылка, а i - нет, но между ними в их дальнейшем
    //использовании разницы нет
int *p = &i;  //p - указатель (pointer) на i
main() {
   cout << i << ',' << j << ',' << *p << endl; //5,5,5
   j = 7;
   cout << i << ',' << j << ',' << *p << endl; //7,7,7
   *p = 11;
   cout << i << ',' << j << ',' << *p << endl; //11,11,11
   p = &j; //p теперь указывает на j
   *p = 4;
   cout << i << ',' << j << ',' << *p << endl; //4,4,4
}
```

Листинг ref-ptr.cpp

Все переменные в этом примере можно было описать в одной декларации `int i = 57, &j = i, *p = &i;`.

Константы-ссылки можно косвенно изменять. Пример.

```
#include <iostream>
using namespace std;
int a = 12;
const int &b = a;
main() {
   cout << b << ' ';
   //b = 11; //так нельзя
   a = 11;
   cout << b << endl;
} //12 11
```

Листинг const-ref.cpp

Знак `&` можно использовать с типами формальных параметров функций: параметры, описанные с ним, передаются по ссылке.

Пример — в нем `i1` передается по значению, а `i2` — по ссылке.

```
#include <iostream>
using namespace std;
void test(int i1, int& i2) {
   i1 = 7;
   i2 = 8;
```

[13] В других языках, в частности, ява, си#, перл, питон, рубин, пиэйчпи, — ссылки отдельным типом не являются.

```
}
main() {
  int i1 = 4, i2 = 5;
  test(i1, i2);
  cout << i1 << ',' << i2 << endl;
} //4,8
```

Листинг param-ref.cpp

Функция может возвращать значение-ссылку и, следовательно, быть допустимой в левой части операции присваивания. Пример.

```
#include <iostream>
using namespace std;
int t[12];
int& ptr(char n) {
  //тут можно добавить контроль за допустимостью индекса
  return t[n];
}
main() {
   ptr(6) = 12; //аналог t[6] = 12;
   ptr(6)--;
   cout << t[6] << endl;
} //11
```

Листинг ref-func.cpp

Если в этом примере объявить значение `ptr` с `const`, то присваивание и декремент к `ptr(6)` будут неприменимы.

С каждой ссылкой ассоциирован указатель и при обращении к ссылке происходит обращение через этот указатель. Размер памяти, реально занимаемой ссылкой, — это размер указателя, хотя `sizeof` выдает размер объекта, указываемого этим указателем. Например, "тип" ссылки на целое отличается от целого типа только способом доступа к данным на машинном уровне. При оптимизации ссылки в некоторых случаях могут быть реализованы без скрытого указателя, прямым отождествлением.

Пример — размер (`sizeof`) структуры `X` будет `sizeof(char)` + `sizeof(char*)`.

```
struct X {
   char &i, j;
   X(): i(j) {}
};
```

Если поменять местами `i` и `j`, то размер может измениться из-за возможного выравнивания адреса для указателя.

Недопустимы массивы ссылок.

Предопределённые параметры функции и перегрузка

Можно задавать предопределённые значения параметров функции и вызывать её без них. В списке параметров после параметра с предопределённым значением не может следовать параметр без предопределённого значения.

```
#include <iostream>
using namespace std;
int f1(int i1, int i2 = 3, int i3 = 77) {
   return i1 + i2 + i3;
}
//int f2(int a = 2, int b) {return a*b;} //ошибка!
 main() {
   cout << f1(1) << ',' << f1(1, 2) << ','
      << f1(1, 2, 3) << endl;
} //81,80,6
```

Листинг default-params.cpp

Параметры без предопределённых значений можно доопределять, но параметрам с предопределёнными значениями менять эти значения нельзя.

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

```
int f1(int i, int j = 6) {return i*j;}
void out1() {cout << f1(2) << ',';}
int f1(int n = 5, int);
void out2() {cout << f1() << ',';}
main() {
   out1();
   out2();
   cout << f1(4, 8) << endl;
} //12,30,32
```

Листинг default-params2.cpp

В отличие от многих других языков программирования (и даже си) вызов функции производится не только по имени, но и по типам аргументов, что позволяет иметь несколько различных функций, имеющих одно и то же имя. Эта возможность называется *перегрузкой* функций (операций).

```
#include <iostream>
using namespace std;
void func(char c) {cout << c;}
void func(char* s) {cout << s;}
//void func(char &c) {cout << c;} //нельзя, т.к. эту функцию
     //невозможно отличить при вызове от func(char)
void func(int i) {cout << i;}
void func(void) {cout << endl;}
main() {
   char c = ':', s[] = "ABC:";
   int i = 17;
   func(c);
   func(s);
   func(i);
   func();
} //:ABC:17
```

Листинг overload.cpp

Математические функции и время

Заголовком `<cmath>` вводятся, в частности, следующие математические функции: `sqrt(x)`, `pow(x, y)`, `sin(x)`, `cos(x)`, `tan(x)`, `sinh(x)`, `cosh(x)`, `tanh(x)`, `asin(x)`, `acos(x)`, `atan(x)`, `exp(x)`, `log(x)`, `log10(x)`, `floor(x)`, `ceil(x)`, `rint(x)` — соответственно $\sqrt{x}$, x^y, $\sin x$, $\cos x$, $\operatorname{tg} x$, $\operatorname{sh} x$, $\operatorname{ch} x$, $\operatorname{th} x$, $\arcsin x$, $\arccos x$, $\operatorname{arctan} x$, e^x, $\ln x$, $\log_{10} x$, округления в меньшую сторону, в большую сторону и к ближайшему целому.

У всех вышеперечисленных функций и `fabs` аргументы и результат имеют тип `double`. Если добавить букву f к их именам, то получатся имена функций с аргументом и результатом типа `float`, а если l, то — `long double`, например, `sinf`.

В заголовке `<cstdlib>` также объявлены несколько математических функций.

`fabs(x)`, `int abs(int x)` — модуль `x`, $|x|$. Для `abs` можно использовать префиксы l — `long` и ll — `long long`.

`int rand()` — случайное число от 0 до `RAND_MAX` - 1.

`void srand(int i = 0)` — инициализация генератора случайных чисел; генераторы, запущенные с одинаковым значением `i`, производят идентичные последовательности.

В заголовке `<ctime>` объявлено средства доступа к аппаратуре часов реального времени — функции `time` (единственный параметр обычно устанавливают в 0), возвращающей количество секунд с начала 1970 года.

Преобразование типов

Неявное преобразование обычно используется при работе с числовыми типами. Все числовые данные полностью совместимы. По умолчанию все целые числа преобразуются в `int`, а вещественные в `double`. Если операнд выражения имеет тип, больший по размеру, чем `int` или `double`, то результат выражения приводится к этому большему типу. При необходимости целые преобразуются в вещественные.

Для преобразования различных типов указателей друг в друга, в целые числа и наоборот нужно использовать явное преобразование типа. Его можно использовать также для преобразования любых скалярных типов друг в друга. Оно может быть в трёх формах:

1) описание типа в круглых скобках предшествует выражению, тип которого надо преобразовать, например, `(int) 2.4` = 2, `(char*) "abc"`;
2) имя типа используется как имя функции, например, `double(4)` = `4.`, `int('@')` = 64, `char(65)` = `'A'`;

3) используя служебные слова, заканчивающиеся на `_cast` — это позволяет разделить преобразования на категории и при компиляции контролировать соответствие данного преобразования выбранной категории.

Инициализация переменных

Данные числовых типов инициализируются выражениями, а указатели инициализируются адресами соответствующих им по типу объектов. Массивы инициализируются перечислением значений своих компонент, начиная с первого, в фигурных скобках. Если в таком перечислении значений меньше, чем компонент массива, то оставшиеся компоненты инициализируются нулем. Массивы символов или строки можно инициализировать перечислением значений в кавычках. Если в определении массива опустить число его компонент, то это число считается равным количеству элементов в перечислении. В инициализаторах многомерных массивов можно опускать внутренние фигурные скобки.

В результате следующих инициализаций

```
int xa[3][3] = {{88,22},{44}}, (*pa)[3] = xa + 1,
  *p = (int*)xa; //pa - указатель на строки из xa,
    //p - указатель на элементы xa
int a[4] = {72, 74, 85};
int b[] = {72, 74, 85}; //b - массив из 3-х элементов
char s1[4] = {48,'1',50}, s2[] = "\06012";
int m1[2][3] = {{1,2,3}, {4,5,6}};
int m2[2][3] = {{1,2}, {5,6}};
int m4[2][3] = {1, 2, 3, 4, 5, 6};
```

получаем `xa` $= \begin{bmatrix} 88 & 22 & 0 \\ 44 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$, `pa` $= \begin{bmatrix} 44 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$, `a` $= [72 \quad 74 \quad 85 \quad 0]$, `p` $= [88 \quad 22 \quad 0 \quad 44 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0]$, `b` $= [72 \quad 74 \quad 85]$, `s1` $=$ `s2` $=$ `"012"`, `m1` $=$ `m4` $= \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$, `m2` $= \begin{bmatrix} 1 & 2 & 0 \\ 4 & 5 & 0 \end{bmatrix}$.

Структуры инициализируются перечислением значений своих полей, начиная с первого, в фигурных скобках. Если в таком перечислении значений меньше, чем полей в структуре, то оставшиеся поля инициализируются нулем.

Пример — будет напечатано 6 строк: 167 F, 180 M, 161, 147 F, 0, 00.

```
#include <iostream>
using namespace std;
struct Person {
   int height;
   char gender;
};
Person x = {170,'M'},
family[5] = {{167,'F'},{180,'M'},{161},{147,'F'}};
   //family - массив из 5 компонент
main() {
   for (int i = 0; i < 5; ++i)
      cout << family[i].height << ' ' << family[i].gender
         << endl;
   cout << int(family[2].gender) << int(family[4].gender)
      << endl;
}
```

Листинг data-ini.cpp

Видимость имён

Имя, описанное в некоторой области, видимо во всех областях, содержащихся в данной, в которых нет описания того же имени. Операция спецификация области видимости, `::`, позволяет обращаться к перекрытым локальными объявлениями глобальным именам.

```
#include <iostream>
using namespace std;
int i = 1, &ri = i, *p = &i;
main() {
   int i = 2;
   cout << i << ::i << ri;
```

```
   {
      int i = 3;
      cout << i << ::i << *p;
   }
   cout << i << endl;
} //2113112
```

Листинг localsvisibility.cpp

Модульное программирование и пространства имён

Си++ — это модульный язык. При трансляции исходного текста создаётся файл (объектный модуль) с расширением o (в средах Microsoft может быть obj). При помощи редактора связей (компоновщика) объектный модуль, содержащий определение функции main, можно превратить в исполняемую программу.

Идентификаторы констант, шаблонов, переменных, типов, макросов и функций, определённые в стандартных библиотеках, группируются по назначению в заголовках. Например, если в программе используется стандартная функция strcpy, то в одной из её начальных строк должна быть директива #include <cstring>[14].

Если нужно использовать определённые в объектном модуле идентификаторы в других программах, то нужно сначала создать файл-заголовок, содержащий объявления этих идентификаторов. Затем в тексты программ, которые используют идентификаторы из этого модуля, добавляется директивой #include содержимое этого файла-заголовка.

На этапе компоновки необходимо указывать расположение модулей и библиотек, используемых при сборке нового модуля, в файловой системе, так как в заголовках нет никакой информации о том, где хранятся соответствующие им модули. Большинство стандартных библиотек и объектных модулей хранится в каталогах lib. В средах программирования информация о местонахождении модулей и библиотек находится в автоматическом или в полуавтоматическом режимах.

Программа на стадии разработки состоит из следующих компонент: текстов программ, текстов файлов-заголовков, описаний ресурсов (тексты или бинарные данные), описания последовательности операций трансляции и связей между всеми требуемыми для успешной трансляции файлами (текст или бинарные данные). В процессе разработки часть текстов заменяется на объектные модули.

Классическим средством для описания последовательности операций трансляции и связей между всеми требуемыми для успешной трансляции файлами является программа make.

Примером ошибок, которые возникают на этапе компоновки, могут служить следующие два текста модулей (единиц трансляции), которые предполагается использовать совместно.

Модуль 1.

```
int x;
char b;
extern char d;
```

Листинг module1.cpp

Модуль 2.

```
int x;
extern char d;
```

Листинг module2.cpp

Для проверки работы с этими модулями нужен ещё модуль, где используются объявленные в них объекты.

```
#include<iostream>
using namespace std;
extern int x;
extern double b;
extern char d;
main () {
   cout << x + b + d << endl;
}
```

Листинг module.cpp

Переменная x определяется дважды — одно из определений следует заменить на объявление extern int x; . Нет определения d — нужно убрать одно из extern, если d не определяется в другом модуле. В модули не переносится информация о типах, поэтому несовпадение типов в объявлении и определении b не вызовет ошибки при компоновке, но может вызвать некорректое поведение программы — ошибку времени исполнения, чтобы избежать этого заменим double на char.

[14] Можно, конечно, вместо подключения заголовка добавить объявление (прототип) этой функции, но это плохой стиль.

Компиляцию и компоновку можно осуществить командой make, для которой нужно подготовить файл с именем makefile.

```
module: module1.o module2.o module.o
        g++ -o $@ $↑
module1.o: module1.cpp
module2.o: module2.cpp
module.o: module.cpp
```

Значки $@ и $↑ означают соответственно результат (module) и файлы из которых он собирается (module1.o, module2.o и module.o) — вместо них файлы можно было прописать явно.

Модули в стиле си образуются из отдельных файлов, а служебные слова static и extern позволяют управлять видимостью имён.

Служебное слово namespace управляет видимостью имён как в одном, так и в группе файлов. Например, в

```
namespace Stack {
   int array[100], sp = 0;
   void push(int d) {array[sp++] = d;}
   int pop() {return array[--sp];}
}
```

Листинг Stack.cpp

определятся реализация стека, а в

```
namespace Stack {
   void push(int);
   int pop();
}
```

Листинг Stack.h

стек только объявляется, что предоставляет интерфейс. Можно работать со стеком, имея только интерфейс к нему, что позволяет разработчикам улучшать реализацию, без влияния на программы пользователя. Если интерфейс стека описан в файле Stack.h, то пользователь может написать

```
#include "Stack.h"
int f() {
   return Stack::pop();
}
```

Листинг with-stack.cpp

Пример — вложенные пространства имён.

```
#include <iostream>
namespace S1 {
  int a = 5, b = 3;
}
namespace S2 {
  int a = 4, c = 7;
  using namespace S1;
    //добавляет имена из S1 к текущему пространству имен (S2)
  void shows () {
    using namespace std;
      //std - имя пространства стандартной библиотеки
    cout << b << S1::a << a << endl;
  }
  namespace S3 {
    int deep = 6;
  }
}
main () {
  int a = 1, d = 2;
  std::cout << a << S1::a << S2::S3::deep << std::endl;
  using S2::shows;
    //открывает имя shows в текущем пространстве имен
  shows();
```

```
} //156
  //354
```

Листинг namespace.cpp

Можно использовать `extern "C"` в объявлениях для указанения того, что функцию следует компилировать по соглашениям языка си, т. е. без поддержки перегрузки. Эта возможность позволяет связывать при компоновке функции си++ и си.

Средства форматирования ввода-вывода через классы-потоки

Это либо функции-члены классов-потоков, либо манипуляторы. Например, функция `width` позволяет задать ширину поля ввода-вывода данных, а функция `precision` — количество выводимых цифр после десятичной точки.

```
#include <iostream>
using namespace std;
main() {
  cout << "10/7 =";
  cout.width(5);
  cout.precision(2);
  cout.setf(ios::fixed);  //устанавливает тип точности
  cout << 10./7 <<endl;
} // 10/7 = 1.43
```

Листинг cout-form.cpp

Манипуляторы (это функции или указатели на функции) можно непосредственно вставлять в потоки ввода-вывода, что снижает громоздкость операций форматирования потоков. Предыдущий пример можно с манипуляторами переписать.

```
#include <iostream>
#include <iomanip>
using namespace std;
main() {
   cout << "10/7 =" << fixed << setw(5) << setprecision(2) << 10./7
      << endl;
} // 10/7 = 1.43.
```

Листинг cout-manip1.cpp

Здесь `setw`, `setprecision`, `fixed` и `endl` — манипуляторы. Все манипуляторы с аргументами объявлены в `<iomanip>`, а без аргументов — в `<iostream>`.

Ещё несколько полезных манипуляторов на примере.

```
#include <iostream>
using namespace std;
main() { //системы счисления
   int i = 15;
   cout << i << ' ' << hex << i << ' ' << i << ' ' << oct << i
      << ' ' << dec << i << endl;
} // 15 f f 17 15
```

Листинг cout-manip2.cpp

Беспараметрный манипулятор `flush` обеспечивает немедленный вывод содержимого буфера вывода — такой же эффект имеет вывод маркера конца строки.

В заголовке `<iostream>` объявлены четыре потоковые переменные:

1) `cin` — класса `istream`, стандартный поток ввода;
2) `cout` — класса `ostream`, стандартный поток вывода;
3) `clog`, `cerr` — класса `ostream`, стандартные потоки вывода сообщений об ошибках соответственно с буферизацией и без;

Классы

Их нужно использовать тогда, когда моделируемая в программе сущность не может быть выразительно и цельно описана при помощи стандартных типов. Например, в программе для финансовых расчётов нужно моделировать деньги. Деньги в программе выразительно и цельно могут быть описаны при помощи числовых стандартных типов и использовать для их описания класс необязательно. Однако, если диапазон необходимых для расчётов денежных величин не вписывается в ограничения стандартных типов, то создание класса для

таких величин — это необходимость. Кроме того, специальный класс позволяет иметь расширяемый набор специальных операций, манипуляторов и т. п.

Другой пример. Для работы с числовыми матрицами, которые часто встречаются на практике, можно использовать стандартный тип массив, но лучше создать соответствующий матрицам класс, так как для стандартного типа массив нет стандартных характерных для матриц операций, например, их сложения, умножения, вычисления ранга и т. п.

Сущностям, соответствующим объектам, имеющим не только количественные характеристики, или сущностям, к которым применимы не только стандартные операции, следует в программе сопоставлять класс.

Инкапсуляция данных

Все компоненты любого класса имеют один из трёх атрибутов доступа: `private`, `protected` или `public`. Компоненты с атрибутом `private` доступны только в функциях-членах или в функциях-друзьях этого класса. Компоненты с атрибутом `protected` доступны ещё в функциях-членах и функциях-друзьях дочерних классов. Компоненты с атрибутом `public` не имеют никаких ограничений на доступ к себе. В классе, описываемом со служебным словом `class`, все компоненты имеют по умолчанию атрибут `private`, а в классе, описанном со служебным словом `struct` или `union`, — `public`.

Пример.

```
#include <iostream>
using namespace std;
class {
  int a;
} x; //бессмысленное описание - компонента a
    //объекта x нигде не доступна
class {
  int a;
public:
  int get_a() {return a;}
  void set_a(int b) {a = b;}
} y;
struct {
  int a;
} z;
main() {
  z.a = 4;
  //y.a = 6; //нельзя!
  y.set_a(6);
  cout << z.a << y.get_a() << endl;
} //46
```

Листинг incapsulation.cpp

Динамические данные

В отличие от данных статических, память для которых выделяется в момент компиляции и не может быть увеличена или уменьшена в процессе исполнения программы, или автоматических, память для которых выделяется и освобождается по однозначным правилам, которые не могут меняться, контроль над памятью для динамических данных полностью осуществляется стандартными операциями. Эти операции связаны с использованием служебных слов `new` (выделение памяти) и `delete` (освобождение памяти). Операция `new` выделяет память для хранения переменной указанного типа. Результат этой операции (значение new-выражения) — это указатель на начало выделенной памяти или генерация исключения (исключение, если его не перехватывать специальными средствами, приводит к аварийной остановке программы). Если перед типом поставить `(nothrow)` , то если память выделить не удалось, возвращается 0. Типичные параметры `new` — это тип и опциональный инициализатор.

Массивы нельзя инициализировать. Память, выделенную `new`, можно освободить только при помощи операции `delete`, типа void с единственным аргументом-указателям. Для удаления массива перед указателем ставят `[]`. Вызов `delete` с аргументом 0 не приводит ни к какому результату.

```
main() {
   const int *p1;
   p1 = new const int (77); //создание целой константы 77
   long double *p4, *p5, **p6;
   p4 = new long double;  //создание вещественного числа
```

```
   p6 = new long double *;
      //создание указателя на вещественное число
   p4 = new long double (1.2);
      //создание вещественного переменной со значением 1.2
   p5 = new long double[4];
      //создание массива из 4 вещественных чисел
   struct Demo {
      int i;
   } *p2, *p3;
   p2 = new Demo[12]; //создание массива из 12 структур demo
   p3 = new struct Demo;
   void **p7, (**p8)(int);
   p7 = new void*; //создание указателя на void
   //*p7 = new void; //ошибка!
   p8 = new (void (*[3])(int));
      //создание массива из 3 указателей
   char *p9;
   p9 = (char*) new Demo; //создание структуры demo
   p2[4].i = *p1; //присваивание 77
   delete p1;
   delete p4;
   delete [] p5;
   delete p6; delete [] p2;
   delete p3; delete p7; delete p8;
   delete (Demo*) p9; //уничтожение структуры demo
}
```

Листинг dynamic1.cpp

Пример — создание, заполнение, распечатка и уничтожение однонаправленного списка.

```
#include <iostream>
using namespace std;
struct TElem {
   TElem *next;
   int value;
};
struct List {
   TElem *first, *curr, *temp;
} list;
main() {
  //создание и заполнение
  list.curr = list.first = new TElem;
  list.curr->next = 0;
  for(;;) {
     int i;
     cin >> i;  //ввод данных с клавиатуры
     if (i) { //признак конца ввода - 0
        list.curr->value = i;
        list.curr->next = list.temp = new TElem;
        list.curr = list.temp;
        list.curr->next = 0;
     } else break;
  }
  //печать
  list.curr = list.first;
  while (list.curr->next) {
     cout << list.curr->value << ' ';
     list.curr = list.curr->next;
```

```
  }
  cout<<endl;
  //уничтожение
  do {
     list.curr = list.first;
     list.first = list.first->next;
     delete list.curr;
  } while (list.first);
}
```

Листинг dynamic2.cpp

Переопределение операций

Для типов, определяемых пользователем, большинство лексем операций можно рассматривать как названия функций, которые допускают определение и перегрузку.

Рассмотрим пример класса, соответствующего числовым матрицам.

```
/*  МОДЕЛИРОВАНИЕ МАТРИЦ РАЗМЕРА ДО 255X255:
   СОЗДАНИЕ МАТРИЧНОЙ АЛГЕБРЫ. АВТОР - В.ЛИДОВСКИЙ */
#include <cstdlib>
#include <iostream>

using namespace std;

void halt(int ErrNo) {
  cout << "Ошибка - недопустимая операция.\n";
  exit(ErrNo);
}

class Array {
  int **p;
public:
  unsigned char HSize, VSize;
  Array(unsigned char, unsigned char);
  Array(const Array&);
  int* operator[](unsigned char) const;
  Array& operator=(const Array&);
  Array& operator=(int*);
  Array& operator+=(const Array&);
  Array operator+(Array) const;
  Array operator-() const;
  Array operator-(Array) const;
  Array operator*(const Array&) const;
  Array operator*(int) const;
  friend Array operator*(int, Array);
  ~Array();
};

Array::Array(unsigned char n, unsigned char m) {
  VSize = n;
  HSize = m;
  p = new int*[VSize];
  for (int i = 0; i < VSize; i++)
    p[i] = new int[HSize];
}

Array::Array(const Array& Data) {
  HSize = Data.HSize;
  VSize = Data.VSize;
```

```
43:   p = new int*[VSize];
44:   for (int i = 0; i < Data.VSize; i++) {
45:     p[i] = new int[Data.HSize];
46:     for (int j = 0; j < Data.HSize; j++)
47:       p[i][j] = Data.p[i][j];
48:   }
49: }
50:
51: Array::~Array() {
52:   for (int i = 0; i < VSize; i++)
53:     delete [](p[i]);
54:   delete []p;
55: }
56:
57: int* Array::operator[](unsigned char i) const {
58:   if (i >= VSize)
59:     halt(2);
60:   return p[i];
61: }
62:
63: Array& Array::operator=(int* InitData) {
64:   for (int i = 0; i < VSize; i++)
65:     for (int j = 0; j < HSize; j++)
66:       p[i][j] = InitData[i*HSize + j];
67:   return *this;
68: }
69:
70: Array& Array::operator=(const Array& Data) {
71:   if (VSize == Data.VSize && HSize == Data.HSize)
72:     for (int i = 0; i < VSize; i++)
73:       for (int j = 0; j < HSize; j++)
74:         p[i][j] = Data.p[i][j];
75:   else
76:     halt(1);
77:   return *this;
78: }
79:
80: Array Array::operator-() const {
81:   Array T(*this);
82:   for (int i = 0; i < VSize; i++)
83:     for (int j=0; j < HSize; j++)
84:       T.p[i][j] = -p[i][j];
85:   return T;
86: }
87:
88: Array& Array::operator+=(const Array& Data) {
89:   if (VSize == Data.VSize && HSize == Data.HSize)
90:     for (int i = 0; i < VSize; i++)
91:       for (int j = 0; j < HSize; j++)
92:         p[i][j] += Data.p[i][j];
93:   else
94:     halt(1);
95:   return *this;
96: }
97:
98: Array Array::operator+(Array Data) const {
99:   if (VSize == Data.VSize && HSize == Data.HSize)
100:     for (int i = 0; i < VSize; i++)
```

```
101:       for (int j = 0; j < HSize; j++)
102:         Data.p[i][j] += p[i][j];
103:   else
104:     halt(1);
105:   return Data;
106: }
107:
108: Array Array::operator-(Array Data) const {
109:   if (VSize == Data.VSize && HSize == Data.HSize)
110:     for (int i = 0; i < VSize; i++)
111:       for (int j = 0; j < HSize; j++)
112:         Data.p[i][j] = p[i][j] - Data.p[i][j];
113:   return Data;
114: }
115:
116: Array Array::operator*(const Array& Data) const {
117:   Array T(VSize, Data.HSize);
118:   if (HSize == Data.VSize)
119:     for (int i = 0; i < T.VSize; i++)
120:       for (int j = 0; j < T.HSize; j++) {
121:         T.p[i][j] = 0;
122:         for (int k = 0; k < HSize; k++)
123:           T.p[i][j] += p[i][k]*Data.p[k][j];
124:       }
125:   else
126:     halt(1);
127:   return T;
128: }
129:
130: Array Array::operator*(int k) const {
131:   Array T(VSize,HSize);
132:   for (int i = 0; i < VSize; i++)
133:     for (int j = 0; j < HSize; j++)
134:       T.p[i][j] = k*p[i][j];
135:   return T;
136: }
137:
138: Array operator*(int k, Array Data) {
139:   for (int i = 0; i < Data.VSize; i++)
140:     for (int j = 0; j < Data.HSize; j++)
141:       Data.p[i][j] *= k;
142:   return Data;
143: }
144:
145: ostream& operator<<(ostream& s, const Array& Data) {
146:   s << endl;
147:   for (int i = 0; i < Data.VSize; i++) {
148:     for (int j = 0; j < Data.HSize; j++)
149:       s << '\t' << Data[i][j];
150:     s << endl;
151:   }
152:   s << endl;
153:   return s;
154: }
155:
156: int main() {
157:   Array A(3,3), B(3,3), C(3,2), D(3,3), E(3,3), G(2,3);
158:   {
```

```
159:      int data[] = {0,1,2,3,4,3,2,1,0};
160:      B = A = data;
161:      C = data + 2;
162:      G = data + 3;
163:    }
164:    Array F(A);
165:    for (int i = 0; i < 3; i++)
166:      for (int j = 0; j < 3; j++)
167:        E[i][j] = i == j;
168:    B += E;
169:    D = F - E;
170:    C[2][1] = -5;
171:    D = -(A*A - 4*B*C*G + E*12)*D;
172:    cout << "A = F =" << A << "B = " << B << "C = " << C
173:      << "G =" << G << "D = -(A*A - 4*B*C*G + E*12)*D = " << D;
174:    return 0;
175: } // D = -(A*A - 4*B*C*G + E*12)*D =
176:    //    579     600     435
177:    //    1836    2346    2052
178:    //    563     760     707
```

Листинг matrix.cc

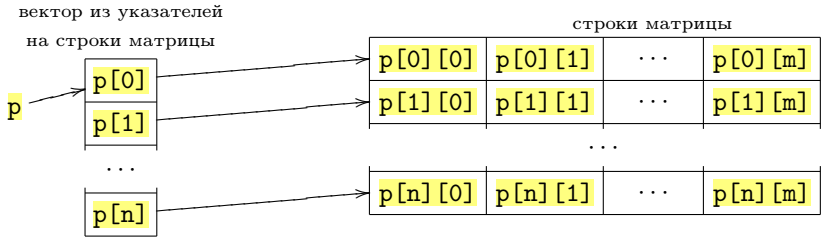
Получаем матрицы, с которыми можно работать, используя стандартный синтаксис выражений. В строке 171 примера `A`, `E`, `B` и `D` — матрицы размерности 3×3, `G` — матрица размерности 2×3 и `C` — матрица размерности 3×2, т. е. размерности матриц в выражениях могут быть любыми (в данной версии программы до 255×255).

В первых строках программы, как правило, описываются её назначение, а также данные об авторских правах (1–4). Далее (5–6) идут директивы препроцессора, включающие заголовки в текст программы.

Функция `halt` (8–11) вызывается в случаях, когда предпринята попытка выполнить недопустимую операцию, например, сложить матрицы разных размерностей. Она выводит сообщение об ошибке и прекращает при помощи стандартной функции `exit` из `<cstdlib>` работу всей программы.

Объявление класса `Array` (13–30) для представления матриц является костяком, вокруг которого строится вся программа.

Внутреннее, машинное представление матричных данных (14), описывается в закрытой части объекта. Указатель на указатель на целое число (`p`) позволяет работать с любым двумерным массивом. Если предоставить пользователю объекта непосредственный доступ к `p`, то нельзя будет проконтролировать случаи, когда индексы при обращении к элементу массива выходят за допустимые для этого массива пределы. Кроме того, изменяя значение `p` или указателей `*p`, можно разрушить машинное представление матрицы и привести программу к «зависанию». На следующей схеме изображена связь указателя `p` с матрицей, размерности $(n+1)\times(m+1)$.



Со строки 16 описаны средства, доступные пользователю объекта, т. е. свойства и методы. Матрица имеет два свойства: вертикальный размер (`VSize`) и горизонтальный размер (`HSize`).

В строках 17 и 18 размещены заголовки-объявления конструкторов. Синтаксически конструкторы — это функции, имеющие то же имя, что и класс, и которые не возвращают никакого (даже `void`) значения. Конструкторов может быть несколько — они различаются по параметрам (типам и количеству). Конструктор вызывается всегда при создании каждого экземпляра объекта (в строках 98, 105*, 108, 113*, 117, 127*, 131, 135*, 138, 142*, 157, 164), причём этот вызов всегда происходит неявно. Конструктор помимо специального эффекта (создание таблицы виртуальных функций) обычно производит и инициализацию данных. Если не описать конструкторы явно, то они будут созданы автоматически, в частности, конструктор копирования.

Конструктор из строки 17 создает матрицу, заданной параметрами размерности, с неопределёнными компонентами. Определение этого конструктора приводится в строках 32–38. После установки значений `HSize` и `VSize` создаётся внутреннее представление матрицы. Сначала создаётся массив указателей на строки матрицы, а затем для каждой строки матрицы выделяется необходимое количество памяти. Этот конструктор используется при первоначальном описании матриц (117, 131, 157).

---

* эти вызовы могут быть отброшены компилятором при оптимизиции кода

Конструктор из строки 18 создает матрицу-копию уже имеющейся матрицы. Определение этого конструктора приводится в строках 40–49 — это почти полная копия определения предыдущего конструктора плюс код для копирования значений компонент одной матрицы в другую. Конструктор с заголовком, идентичным рассматриваемому, называется конструктором копирования. Он автоматически создаётся для каждого класса. При своём выполнении он копирует покомпонентно все поля из объекта-аргумента в создаваемый объект. Но, как правило, этот стандартный конструктор переопределяют, так как он не может адекватно работать с указателями. Что, например, произойдет в строке 164 программы, если не переопределить этот конструктор? Полям `VSize` и `HSize` матрицы `F` присвоятся значения те же, что и в `A`, — это правильно. С полем `p` случится то же самое, но это приведет к тому, что матрицам `A` и `F` будет соответствовать одно и то же машинное представление, т. е. получится не создание новой матрицы-копии `A`, а введение в программу другого имени для матрицы `A`.

В строке 19 объявляется функция-член класса `Array`, предназначенная для обращения к компонентам массива. Если `A` объект класса `Array`, то обращение `A[2]` означает вызов этой функции, т. е. вызов `operator[](2)` . Определение этой функции находится в строках 57–61: после проверки допустимости индекса функция возвращает указатель на соответствующую индексу строку массива. Таким образом, в записи `C[2][1]` (170) 1-е квадратные скобки — это обращение к функции `Array::operator[]`, а 2-е — это операция обращения к элементу массива. Следовательно, допустимость второго индекса проверить невозможно.

В строке 20 объявляется функция `operator=` (см. определение в строках 70–78). Если `A` и `B` объекты класса `Array`, то запись `A = B` — это форма записи вызова `A.operator=(B)`. В этой функции используется служебное слово `this`. Оно — указатель на объект, вызвавший функцию-член класса. Функции-члены класса существуют в единственном числе для всех экземпляров этого класса (объектов), поэтому при вызове функции-члена в указатель `this` копируется значение, позволяющее определить, какой конкретно объект её вызвал. Указатель `this` можно использовать только внутри нестатических функций-членов класса. Если не описать функцию `operator=` явно, то она будет создана автоматически и, обеспечивая лишь покомпонентное копирование, будет иметь недостатки аналогичные тем, что показаны у создаваемого по умолчанию конструктора копирования.

В строке 21 также объявляется функция `operator=` (см. определение в строках 63–68), т. е. функции-операции можно перегружать. Эта функция позволяет инициализировать данные в матрице, используя стандартные массивы си, так как непосредственно инициализировать объекты значением списка констант нельзя. Пример инициализации смотри в строках 158–163. Переменная-массив `data` уничтожается в 163 строке, так как она нужна только для инициализации.

Строка 22 объявляет операцию `+=`. Запись `A += B` лучше всего использовать для сокращения записи `A = A + B`. Однако, то, что в действительности произойдет при выполнении `A += B`, зависит от реализации функции `operator+=` (88–96).

Строка 24 объявляет операцию унарный минус для матриц. Если `A` объект класса `Array`, то запись `-A` — это форма записи вызова `A.operator-()` (80–86).

В строке 23 объявляется функция `operator+` (см. определение в строках 98–106). В отличие от функций `operator=` или `operator+=` эта функция порождает новый, временный объект-матрицу и возвращает его значение. Рассмотрим, например, матричное выражение `A = B + C`. `B + C` — это вызов `B.operator+(C)`. Результат этого вызова — это не `B` и не `C`, а некая временная, безымянная матрица, значение которой нужно присвоить `A`.

В строках 25–27 объявляются матричные операции, реализация которых (в строках 108–114, 116–128 и 130–136) аналогична операции `operator+`. Операцию умножение матрицы на число можно использовать только в форме сначала матрица, а затем число. Например, если `A` — матрица, то запись `A*4` означает вызов `A.operator*(4)`, а запись `4*A` недопустима.

Последнее неудобство преодолевается в строке 28, в которой объявляется функция-друг класса `Array` (определение в строках 138–143), которая не есть член этого класса. Функции-друзья как и функции-члены класса могут обращаться к закрытой части объекта. Функция может быть другом нескольких классов.

Служебное слово `operator` для переопределения операций можно использовать только в двух случаях: 1) для функций-членов классов; 2) для функций, хотя бы один из параметров которых является объектом. Первый случай был рассмотрен ранее. Во-втором случае, для рассматриваемого примера, запись `4*A` означает вызов `operator*(4, A)`. Использование спецификатора `friend` допустимо только в объявлении класса.

Невозможно переопределять операции для стандартных типов и сделать, например, чтобы результатом выражения 2 + 2 стало 5.

Кроме `operator=` предопределённое значение имеют ещё только `operator,` и унарный `operator&`. Операции `::`, `?:`, `.`, `.*` и `->*` переопределять нельзя.

В строке 29 объявляется деструктор (определение — в строках 51–55). У объекта может быть только один деструктор, который всегда автоматически вызывается при уничтожении объекта (в строках 106*, 114* и 143*, когда уничтожается автоматическая переменная `Data`; в строках 86*, 128* и 136*, когда уничтожается автоматическая переменная `T`; в строках 169 и 171, когда уничтожаются временные переменные-результаты работы функций-операций; в строке 175 уничтожаются объекты-матрицы `A`, `B`, `C`, `D`, `E`, `F` и `G`). В отличие от конструктора деструктор можно вызывать и явно. Имя деструктора начинается со знака тильда, после которого следует имя его класса. Деструктор не может иметь параметров и не возвращает никакого значения. Если не определить

---

* эти вызовы также могут быть отброшены компилятором при оптимизации

деструктор явно, но он будет создан автоматически. Однако, такой деструктор не сможет, например, освободить динамически выделенную объекту память.

В строках 145–154 доопределяется функция `operator<<` для аргументов-матриц, что позволяет в программе выводить матрицы на печать (172) так же как и данные стандартных типов.

В строках 165–167 происходит заполнение матрицы `E` так, чтобы она стала единичной. Эти строки можно заменить на

```
{
   int data[] = {1,0,0, 0,1,0, 0,0,1};
   E = data;
}
```

В последних строках листинга приводится результат вычисления матрицы `D`, выводимый на печать.

В приведённой программе есть существенный недостаток: при обращении к элементу-строке матрицы возвращается стандартный массив, а не объект матрица-строка. Поэтому в программе нельзя, например, написать `C[0] = A[2]*C` или просто `B[1]=D[3]`[15]. Чтобы избавится от этого недостатка нужно, чтобы функция `operator[]` возвращала значение типа `Array`. Если перегружать не операцию `operator[]`, а операцию `operator()`, то добиться этого будет легче.

Другим менее важным и проще устранимым недостатком является открытость полей `HSize` и `VSize`, что позволяет их бесконтрольно менять. Эти поля следовало бы перенести в закрытую часть и определить две функции, возвращающие размерности.

Для полноты алгебры следует ещё реализовать `operator+` (унарный плюс), `operator==` и `operator!=`.

Использование конструкторов для преобразования типов

В предыдущем примере операцию умножения матрицы на число можно заменить на операцию умножения заданной матрицы размерности $N \times M$ на квадратную матрицу размерности $M \times M$, у которой все компоненты главной диагонали равны заданному числу, а прочие компоненты равны нулю. Таким образом, если при умножении матрицы на число будет происходить автоматическое неявное преобразование типа число в тип матрица, то это позволит обойтись без описания двух операций для матриц. Если описать конструктор с параметром-числом, который строит нужную квадратную матрицу, то это обеспечит выполнение требуемого неявного преобразования типов. Однако, для предыдущего примера этот способ не годится, так как в нем матрицы могут иметь произвольный размер. Но если реализовывать арифметику квадратных матриц фиксированного размера, то этот способ можно использовать.

Если же рассматривать, например, реализацию арифметики комплексных чисел, которые совместимы и с вещественными и с целыми числовыми типами, то использование здесь конструкторов-преобразователей типов — это наиболее рациональное решение.

Наследование

Как правило, удобно рассматривать объект входящим в ту или иную иерархию родо-видовых отношений. Например, конкретная собака Шарик — это частный случай общего понятия собака. Собака, в свою очередь, частный случай млекопитающегося и т.д. Для отражения отношений подобного типа в ОО ЯП входит аппарат поддержки наследования, который заключается в предоставлении возможности строить по базовым, родовым типам, типы-наследники, к которым автоматически переходят все, кроме конструкторов и деструктора, компоненты родительского типа. В типах-наследниках можно вводить новые компоненты. Практически очень важной в аппарате наследования является концепция совместимости типов-родителей с типами-потомками из четырёх пунктов:

1) указатель на родительский тип можно использовать как указатель на любой из его типов-потомков, включая непрямые;
2) объекту типа-предка можно присваивать значение объекта типов-потомков;
3) ссылки можно инициализировать объектами-потомками;
4) список доступных для использования компонент объекта определяется типом величины, хранящей объект или указывающей на него.

Эти ограничения происходят из-за необходимости контроля соответствия декларируемых величин используемым на этапе компиляции.

Таким образом, можно создавать списки разнотипных объектов, входящих в одну родо-видовую иерархию, имея тип-указатель на вершину этой иерархии, что дает возможность представить все объектные данные программы или даже ОС в виде списочных структур, что типично для программирования. Примером такой структуры являются видимые элементы графического интерфейса пользователя (GUI) сред Microsoft Windows или X Window. При использовании наследования более естественной является работа не с объектами класса, а с указателями на них.

Пример использования наследования — напечатает три строки: 17 12 14 14 7 1, 12 17 14 4, 4 7 4.

```
#include <iostream>
```

[15] последняя операция возможна в паскале

```
using namespace std;
int f(void) {return 1;}
class Ancestor {
public:
   int a;
   int c;
   int f(void) {return 4;}
} *p, a;
class Descendant: public Ancestor {
public:
   int a;
   int f2(void) {return 2;}
   int f(void) {return ::f() + f2() + Ancestor::f();}
} b, *q;
main() {
   a.a = 11;
   a.c = 15;
   b.Ancestor::a = 17;
   b.a = 12;
   b.c = 14;
   a = b;
   p = &b;
   q = &b;
   p = q;
   //q = &a; q = p; b = a; //нельзя, т.к. при компиляции будет
      //невозможен контроль допустимости q->f2() или b.f2()
   cout << a.a << ' ' << b.a << ' ' << a.c << ' ' << b.c
      << ' ' << b.f() << ' ' << f() << endl;
   cout << q->a << ' ' << p->a << ' ' << q->c << ' '
      << b.Ancestor::f() << endl;
   cout << p->f() << ' ' << q->f() << ' ' << q->Ancestor::f()
      << endl;
   //cout << p->Descendant::f() << endl; //нельзя! //*
}
```

Листинг inheritance1.cpp

При использовании наследования естественно применяется операция спецификации области видимости, `::`. Перед ней нужно ставить имя базового класса, к компонентам которого необходим доступ. При использовании указателя на базовый класс, указывающий на объект производного типа, при обращении к компоненте объекта по умолчанию выбирается компонента базового класса. Таким образом, имея указатель на базовый класс, невозможно без специальных средств (виртуальных функций) получить доступ к компонентам производного класса, перекрывающих базовые (см. строку //*), что очень неудобно, так как именно указатели на базовые типы являются самыми совместимыми. Класс может иметь несколько родителей.

Синтаксис описания наследования следующий: после имени описываемого класса и двоеточия следует список базовых классов, элементы которого отделяются друг от друга запятой. Каждому имени класса в этом списке может предшествовать служебное слово, атрибут наследования, `private`, `protected` или `public` — по умолчанию берется атрибут, соответствующий виду базового класса (`struct` — `public`, `class` — `private`). Атрибут наследования может лишь ограничить права доступа к компонентам базового класса, т. е. атрибут наследования имеет меньший приоритет, чем атрибут компоненты в производном классе. Функции-члены и функции-друзья производного класса могут иметь доступ только к компонентам с атрибутами `public` и `protected` базового класса. Прочие функции могут иметь доступ только к компонентам с атрибутом `public`.

Вместе с использованием общего атрибута наследования можно каждой наследуемой компоненте явно приписать атрибут доступа, отменяющий значение общего атрибута. Отменять можно только значение атрибута, полученного от атрибута дополнительного ограничиния при описании наследования.

```
class Basis {
   int value; //private
   int k;
public:
   int b;
```

```
};
class Inherited: /* private */ Basis {
   int value;
   //Basis::k; //нельзя!
public:
   Basis::b; //это не описание компоненты, а уточнение
      //атрибута доступа к компоненте базового класса
};
```

При наследовании функции не перегружаются.

```
class Base {
public:
   int func(int i) {return i*i;}
};
class Next: public Base {
public:
   int func(char *c) {return *c+1;}
} t;
main() {
   //t.func(3);     //ошибка
   t.Base::func(3); //правильно
}
```

Листинг inheritance2.cpp

Используя служебное слово `using`, можно организовать перегрузку функций при наследовании, так как класс естественным образом формирует пространство со своим именем, хотя его нельзя использовать с `namespace`. Для того, чтобы `t.func(3)` в предыдущем примере можно было вызвать непосредственно, нужно в раздел `public` класса `Next` добавить строку `using Base::func;`.

При использовании производных классов возникает проблема с конструкторами, которые не вызываются явно. Как из конструктора производного класса вызвать конструктор базового класса? Для этого используется операция инициализации класса, `:`, которое ставится после имени конструктора в его определении и после которого через запятую следуют вызовы конструкторов базовых классов.

```
class Basis {
   int val1, val2;
public:
   int get_val(void) {return val1;}
   Basis() {val1 = val2 = 1;}
   Basis(int n) {val1 = val2 = n;}
};
class Inherited: public Basis {
   int sum;
public:
   Inherited() {sum = 0;}
     //тут по умолчанию вызывается конструктор Basis()
   Inherited(int n, int s): Basis(n) {sum = s;}
   void add(Basis& b) {sum += b.get_val();}
};
```

Виртуальные функции и полиморфизм

Позволяют указателям на базовые классы, указывающим на производные классы, вызывать функции из этих производных классов, перекрывающие функции базовых классов. Виртуальными могут быть только функции, но не поля. Виртуальные функции описываются служебным словом `virtual`. Если при помощи указателя на базовый класс вызывается виртуальная функция, то происходит вызов именно функции-члена того класса, на который указывает этот указатель. Спецификатор `virtual` нужно указывать при всех виртуальных функциях только в базовом классе. Виртуальные функции должны иметь идентичные заголовки (допускается лишь незначительное отличие в типе возвращаемого значения). Типы с виртуальными функциями называют *полиморфными*.

Конструкторы и статические функции не могут быть виртуальными, а деструктор, как правило, описывают виртуальным.

```
#include <iostream>
```

```
using namespace std;
class Basis {
public:
  virtual void identify(void) {
    cout << "Basis\n";
  }
};
class Inherited: public Basis {
public:
  void identify(void) {
    cout << "Inherited\n";
  }
};
main() {
  Inherited inherited;
  Basis *p = &inherited;
  p -> identify();
} //Inherited
```

Листинг virtual.cpp

Если из текста этого примера убрать слово `virtual`, то будет напечатано `Basis`.

Связь кода вызова функции с кодом самой функции, которая устанавливается в момент компоновки, называется *статической связью*. Установка статической связи — это подстановка адреса кода вызываемой функции в команду вызова этой функции. При *динамической связи* вызов функции и её код связываются во время исполнения программы, в два этапа: 1) получение адреса той функции, которую нужно вызвать, из таблицы виртуальных функций; 2) вызов функции по полученному адресу. Динамическую связь называют также *поздним связыванием*.

Код для вызова виртуальных функций незначительно больше и медленнее, чем невиртуальных. Но хорошим стилем является полный отказ от использования невиртуальных методов.

Виртуальные функции наряду с перегружаемыми дают возможность реализовывать *полиморфные операции*, т. е. операции одинаковой синтаксической структуры, но действующие по разному на разных типах данных.

Абстрактные классы

Это классы, содержащие абстрактные функции-члены. Нельзя создавать объекты абстрактных классов. Абстрактные функции не имеют тела и, следовательно, определения. Объявление абстрактной функции — это объявление виртуальной функции, после списка формальных параметров которой следует инициализация нулем.

```
struct MetaBasis {
   virtual int func(float) = 0;
};
```

Абстрактные классы используют в качестве вершин в иерархии классов для придания иерархии единообразия. Абстрактные функции часто называют *чисто виртуальными*.

Вложенные классы

Если класс содержит компоненту-объект, то возникает проблема её инициализации, подобная той, что возникает при инициализации производных классов. Эта проблема решается схожим с ранее рассмотренным методом: в определения конструкторов вставляется инициализирующая часть, вызов конструктора вложенного класса. Синтаксис вызова отличается от случая для классов-потомков — вместо имени класса в вызове приводится имя объекта-компоненты.

```
class Address {
   long zipCode;
public:
   Address(long n) {zipCode = n;}
   long getZipCode();
};
class Student {
   Address address;
public:
   Student(long n): address(n) {}
};
```

Этот пример можно было бы изменить так, чтобы класс `Student` стал производным от класса `Address`, но это не отражало бы реальную связь между студентом и его адресом: не следует использовать аппарат наследования в подобных случаях.

Подобную инициализацию можно использовать с любыми членами класса.

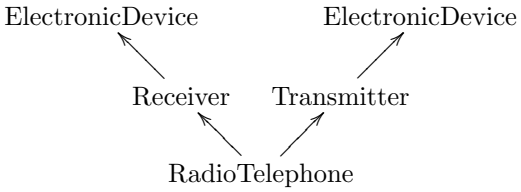
```
class X {
   int i;
public:
   X(int n): i(n) {}
};
```

Множественное и виртуальное наследование

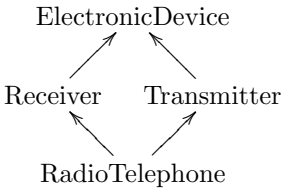
Производный класс может иметь несколько базовых. При описании такого производного класса базовые классы с опциональными атрибутами наследования перечисляются через запятые.

```
class ElectronicDevice {
public:
   int x;
};
class Receiver: public ElectronicDevice {};
class Transmitter: public ElectronicDevice {};
class RadioTelephone: public Receiver, public Transmitter {} z;
```

Связи между этими классами можно изобразить следующей схемой



Таким образом, объект `z` унаследует по два экземпляра каждой компоненты `ElectronicDevice`, в частности, по два поля `x`: доступ к первому осуществляется через `z.Receiver::x`, ко второму — `z.Transmitter::x`. Если же требуется, чтобы наследование осуществлялось в соответствии со схемой



то объявления классов `Receiver` и `Transmitter` нужно записать так:

```
class Receiver: public virtual ElectronicDevice {};
class Transmitter: public virtual ElectronicDevice {};
```

Добавление служебного слова `virtual` как атрибута наследования базового класса в декларации множества производных от него классов приводит к тому, что все классы этого множества будут совместно использовать единственную копию заданного базового класса. Применительно к рассматриваемому примеру это означает, что после внесения указанных изменений обращения `z.Receiver::x` и `z.Transmitter::x` будут означать одно и то же.

Наследование называется *множественным* при использовании более одного непосредственного базового класса. При использовании ровно одного базового класса наследование называется *одиночным*. Класс, наследуемый с атрибутом `virtual`, называется виртуальным базовым классом. Использование виртуального базового класса — это *виртуальное* наследование. Виртуальное и множественное наследования являются противоположными по смыслу: первое позволяет в графе наследования создавать сходящиеся направления, а второе — расходящиеся. Комбинацией этих трёх типов наследования можно получить структуру наследования, описываемую любым заранее заданным, ориентированным, не содержащим циклов графом.

Указатели на компоненты структур

Имеется возможность использовать специальные типы указателей, которые позволяют указывать не на объект в целом, а только на его одну выбранную компоненту.

```
#include <iostream>
using namespace std;
class Extreme {
public:
```

```
   int save_min, save_max;
   int max(int v1, int v2) {
      return v1 > v2 ? v1 : v2;
   }
   int min(int v1, int v2) {
      return v1 < v2 ? v1 : v2;
   }
};
main() {
   Extreme t, *pt = &t;
   int v1 = 4, v2 = 8;
   int (Extreme::*p1)(int, int),
        //p1 - указатель на компоненту-метод
      Extreme::*p2 = &Extreme::save_max;
        //p2 - указатель на компоненту-поле
   p1 = &Extreme::max;
   t.*p2 = (t.*p1)(v1, v2);
   cout << "max=" << t.save_max << ", ";
   p1 = &Extreme::min;
   p2 = &Extreme::save_min;
   t.*p2 = (pt->*p1)(v1, v2);
   cout << "min=" << t.save_min << endl;
} // max=8, min=4
```

Листинг pointer2member.cpp

Указатели на компоненту класса не похожи на обычные указатели. Указатель на компоненту экземпляра класса — это не абсолютный адрес памяти, а адрес относительно начального адреса объекта класса. Такой относительный адрес называют смещением, относительно абсолютного адреса объекта — базы. Преобразование величины типа указатель на компоненту класса в обычный указатель или число можно провести, например, используя объединение. Указатель на компоненту класса функцию совместим с обычными указателями.

Шаблоны

Шаблоны бывают двух видов: для функций и для классов. Они предназначены для описания общих (родовых) определений, по которым компилятор автоматически создает конкретную версию функции или класса.

Шаблоны во многом похожи на макросы, но они дают более разнообразные возможности и более контролируемы.

Шаблонами следует всюду, где это возможно, заменять перегружаемые функции. Например, рассмотрим функцию `max1(x,y)`, которая возвращает больший из своих аргументов. Тип аргументов и возвращаемого значения у этой функции хотелось бы иметь любым, допускающим сравнение своих конкретных реализаций. Си++ — это язык со строгим контролем типов, поэтому необходимо, чтобы типы параметров были описаны до времени компиляции. Без шаблонов пришлось бы писать много перегружаемых версий функции `max1`, хотя их определения получались бы практически идентичными. Используя шаблоны, можно определить образец для семейства перегружаемых функций, задавая тип как параметр.

```
#include<iostream>
using namespace std;
template <class T> //T - это аргумент-тип
   T max1(T x, T y) {
      return x > y ? x : y;
   }
struct X {
   int a;
   X(int v): a(v) {}
   int operator>(X v) {
      return a > v.a;
   }
};
ostream& operator<<(ostream& s, const X& x) {
   return s << x.a;
}
```

```
main() {
   int i = 12;
   double r = 7.5;
   X x(8), y(11);
   cout << max1(i, 14) << ' ' << max1(8.3, r) << ' '
      << max1(x, y) << endl;
}
```

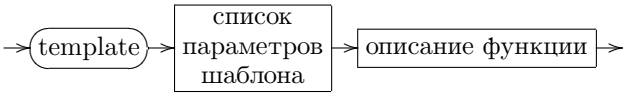
Листинг func-template.cpp

Используя этот шаблон, компилятор создает соответствующую функцию, согласно типу данных, использованному в вызове функции, т. е. вызов функции приводит к генерации её тела. Этот шаблон позволяет использовать `max1` с любым типом, для которого определена операция `>` или `operator>`. Тип-параметр в шаблоне может быть любым: числовым, указателем или классом.

Вместо рассмотренного шаблона можно использовать макрос
`#define max1(x,y) ((x) > (y) ? (x) : (y))`.

Макросы не обеспечивают проверки соответствия типов формальных и фактических параметров, поэтому их использование затрудняет поиск ошибок.

Далее — общий синтаксис описания шаблона функций — шаблон получается из декларации функции добавлением template-строки.



Список параметров шаблона заключается в угловые скобки, компоненты списка отделяются друг от друга запятыми. Параметры-типы предваряются служебным словом `class`. Могут быть ещё и параметры-константы. Параметрам всех видов может после знака `=` назначаться предопределённое значение. Последняя возможность для шаблонов функций не имеет практического смысла.

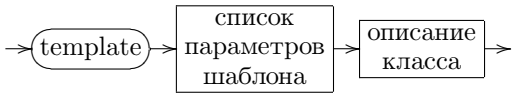
Шаблон класса, называемый также родовым классом или *генератором* классов, задает образцы для определения классов.

Типичным примером использования шаблона класса является работа с классами-контейнерами. Операции с такими классами практически не меняются при смене типа хранимых в них величин. Можно описать один родовой класс-контейнер и позволить системе генерировать конкретные описания классов «на лету».

```
#include <iostream>
using namespace std;
template <class T = int, int d = 10>
   class Container {
      T *v;
      int size;
   public:
      Container(int);
      T& operator[](int i) const {return v[i];}
      int get_size() const {return size;}
  };
template <class T, int d>
   Container<T, d>::Container(int n = d): size(n)  {
      v = new T[n];
   }
main() {
   Container<float, 5> w;
   Container<> x(20);
   Container<double> y(30);
   Container<> z;
   x[3] = 7;
   y[3] = 2.5;
   z[1] = 8;
   cout << w.get_size() << ' ' << x.get_size() << ' '
      << y.get_size() << ' ' << z.get_size() << ' '
      << x[3] + y[3] + z[1] << endl;
} //5 20 30 10 17.5
```

Листинг class-template.cpp

Общий синтаксис описания шаблона классов следующий.



— таким образом, определение шаблона тоже получается из определения класса добавлением template-строки.

Некоторые типы данных могут потребовать особенной реализации — такая реализация называется специализацией шаблона.

```
#include <iostream>
#include <cstring>
using namespace std;
template <class T>
  class Data {
    T v;
  public:
    Data() : v(0) {}
    T& operator=(T);
    operator T () {return v;}  //преобразование типа
};
template <class T> T& Data<T>::operator=(T d) {
    return v = d;
}
template<> char*& Data<char*>::operator=(char* d) {
     //специализация для типа char*
   delete v;
   v = new char [strlen(d) + 1];
   strcpy(v, d);
   return v;
}
main() {
   Data<long> x;
   Data<double> y;
   Data<char*> s;
   x = 7;
   y = 3.1;
   s = (char*) " ok\n";  //преобразование необязательно
   cout << x*y << s; //Data преобразуется в числовой
} //21.7 ok
```

Листинг template-spec.cpp

Для шаблона функции компилятор не генерирует код до тех пор, пока функция не будет фактически вызвана. Имеется специальное средство для помещения «кодов» шаблонов в объектные модули. Шаблон, помещаемый в модуль, должен предваряться служебным словом `export`[16].

В параметрах шаблона вместо слова `class` можно использовать `typename`.

Кроме того, при определении шаблона класса может возникнуть неоднозначность при интерпретации назначения идентификатора компоненты класса — `typename` квалифицирует идентификатор как тип.

Шаблоны абстрактных типов являются самым высоким уровнем абстракции для типов.

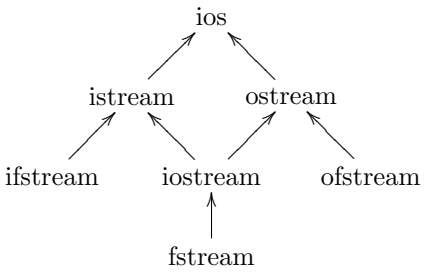
Стандартная библиотека си++ содержит множество готовых шаблонов контейнеров и средств для работы с ними, в частности, контейнеры `vector`, `list`, `deque`, `stack`, `map` (ассоциативный массив, индексы в нем могут быть любого типа и не обязательно должны идти подряд), `set`, `multiset` (набор).

Потоки ввода-вывода

Стандартный ввод-вывод осуществляется через объекты потоковых классов с буферизацией. Средства ввода-вывода потоками объявлены в заголовках `<fstream>`, `<iostream>` и `<iomanip>` (используется совместно с `<iostream>`) — это классы `ios`, `istream`, `ostream`, `iostream`, `ifstream`, `ofstream`, `fstream` и `streambuf`. Последний класс — это класс-буфер, а прочие — это собственно классы-потоки. Класс `ios` является базовым для классов-потоков.

Схемы наследования потоковых классов

[16] Не поддерживается текущим (2010) GNU си++.



Классы `ostream` (для вывода), `istream` (для ввода) и `iostream` (для ввода и вывода) предназначены для работы со стандартными устройствами (экран, клавиатура, ...).

Классы `ofstream` (для вывода в файл), `ifstream` (для ввода из файла) и `fstream` (для файлового ввода и вывода) предназначены для работы с файлами.

Работа с файлами в объектном стиле, пример.

```
#include <fstream>
#include <iostream>
using namespace std;
int main () {
  ofstream fo("4write.txt");
  if (!fo) {
    cerr << "file can't be opened for write\n";
    return 1; //код возврата для среды вызова (ОС),
          //0 - ошибок нет, не 0 - номер ошибки
  }
  fo << "Hello\n";
  fo.close();
  ifstream fi("4write.txt");
  if (!fi) {
    cerr << "file can't be opened for read\n";
    return 2;
  }
  char c, s[80];
  fi >> s;
  cout << s;
  fi.seekg(0);  //для ofstream - seekp
  while (fi.get(c)) cout << c;
  fi.close();
  return 0;
} //HelloHello
```

Листинг fstream.cpp

Для потоков ввода метод `get` может осуществлять посимвольный ввод, а метод `unget` возвращает последний введённый символ в поток. Метод `get` может также использоваться для ввода в буфер данных — в этом случае первым аргументом должен быть буфер (указатель или массив), вторым — размер буфера, а третьим — символ-маркер конца вводимых данных (по умолчанию — это маркер конца строки). Методы `seekg` (seek get) и `seekp` (seek put) — это средства прямого доступа — они устанавливают позиции для чтения и записи соответственно.

Работа со строковыми потоками, пример.

```
#include <sstream>
#include <iostream>
#include <string>
using namespace std;
main () {
  ostringstream os;
  string s;
  os << "2x2=5\n";
  s = os.str(); //преобразование содержимого потока в строку
  s[4] = '4';
  os.str("Hello"); //установка содержимого потока
  cout << os.str() << endl << s;
}
```

```
//Hello
//2x2=4
```

Листинг sstream.cpp

Строковые потоки можно использовать для преобразования любых типов данных в тип `string` с одновременным опциональным форматированием.

Для ввода произвольных текстовых строк очень удобен метод `getline`. При вводе через операцию `<<` пробелы рассматриваются как концы строк. У `getline` два параметра: указатель на массив символов и предел длины для вводимой строки. Можно ещё использовать функцию `getline` с аргументами типа поток ввода и объектная строка.

```
#include <iostream>
#include <string>
using namespace std;
main() {
   char s1[80];
   string s2;
   cin.getline(s1, 80);
   cout << s1 << endl;
   getline(cin, s2);
   cout << s2 << endl;
}
```

Листинг getline.cpp

Форматный ввод-вывод в стиле си

Использует объявления из `<cstdio>`, более компактен, но менее нагляден, из-за отсутствия возможности для проверки типов, менее надёжен и, главное, его нельзя приспособить для работы с типами, определяемыми пользователем.

Пример.

```
#include <cstdio>
using namespace std;
main() {
   printf("10/7 =%5.2f\n",10./7);
   int i = 15;
   printf("%d %x %X %o %d\n", i, i, i, i, i); //15 f F 17 15
}
```

Листинг printf.cpp

Таким образом, 1-й параметр `printf` — это строка-спецификатор формата, в которой каждая спецификация размещения следующих параметров расположена между знаками процента и буквы, определяющей способ показа. Используются буквы: `c` (символ), `s` (строка), `d` (десятичное число со знаком), `u`, `o`, `x`, `X` (целое число соответственно без знака десятичное, восьмеричное, 16-е и 16-е с заглавными латинскими буквами-цифрами), `f`, `e`, `E` (вещественное число соответственно без порядка, с порядком и с порядком с заглавной буквой `E` для порядка). Между `%` и буквой можно опционально ввести знак, ширину и количество цифр после десятичной точки.

Функции `fprintf` и `sprintf` отличаются от `printf` наличием дополнительного параметра, типов `FILE*` и `char*` соответственно. Они печатают не в стандартный поток вывода, а соответственно в файл и в строку. Пример.

```
#include<cstdio>
using namespace std;
main() {
   FILE *out;
   char s[200];
   sprintf(s, "%dx%d=%d\n" , 2, 2, 5); //установит s в "2x2=5"
   s[4] = '4';
   printf("%s", s); //2x2=4
   out = fopen("testfile", "w");
   fputs(s, out);
   fprintf(out, "%.1f x %.1f = %.2f\n", 5.2, 1.6, 5.2*1.6);
   fclose(out);
} //в текущем каталоге должен появиться файл testfile из 2 строк
```

Листинг sf-printf.cpp

Форматированный ввод осуществляется функцией `scanf`, схожей по параметрам с `printf`. Пример.

```
#include<cstdio>
using namespace std;
main() {
  int i;
  scanf("%d", &i);
  fprintf(stdout, "%e\n", i*1000.1);
     //fprintf(stdout,...) эквивалентно printf(...)
     //вместо stdout можно использовать stderr для
     //записи в поток ошибок
}
```

Листинг scanf.cpp

Параметры `scanf`, кроме первого, — это адреса переменных. Есть также функции `sscanf` и `fscanf`. Пример, должно быть напечатано 1001 7 − two.

```
#include <cstdio>
using namespace std;
main() {
   char input[] = "These are two numbers 1001 and 7", s[32];
   int n1, n2;
   sscanf(input, "These are %s numbers %d %*s %d", s, &n1, &n2);
   printf("%d %d - %s\n", n1, n2, s);
}
```

Листинг sscanf.cpp

Знак `*` используется для величин, которые выделяются при сканировании строки, но не используются. Пробел в строке форматирования означает любое число (и пустое) пробелов, концов строк и табуляций.

Исключения

Дают возможность контролировать ошибки. Кроме того, исключения можно использовать для выхода из вложенных конструкций как циклов, так и вызовов функций. Исключения можно перехватывать. Неперехваченное исключение аварийно прекращает работу программы. Для работы с исключениями используются служебные слова `throw` (генерировать исключения), `catch` (перехватить исключение), `try` (создает блок, допускающий перехват исключений).

Пример, печатающий 9 строк: bad division, -5, bad logarithm, 1, 0, other error, 0, 0, impossible.

```
#include <iostream>
#include <cmath>
using namespace std;
int div(int n, int d) {
  if (d==0) throw 1;
  return n/d;
}
double ln(double x) {
  if (x<=0) throw 2;
  return log(x);
}
main() {
  int i = -5, j = 0;
  while (1) {
    try {
      cout << div(i, j) << endl;
      cout << ln(i) << endl;
      if (j == 1)
        throw 7.4;
      throw 5;
    }
    catch (int k) {
      switch (k) {
```

```
      case 1:
        cerr << "bad division\n";
        j = 1;
        continue;
      case 2:
        cerr << "bad logarithm\n";
        i = 1;
        continue;
      default:
        cerr << "impossible\n";
      }
    }
    catch (...) {
       cerr << "other error\n";
       j = 2;
       continue;
    }
    break; //выход из цикла
  }
}
```

Листинг exceptions.cpp

После try-блока следуют catch-операторы, в которых нужно указывать в скобках тип или тип и переменную этого типа, либо многоточие. Затем в обязательных фигурных скобках приводится код обработки исключения. Переходы во внутрь try-блока запрещены. Если ни один из catch-операторов не соответствует сгенерированному исключению, то исключение передается дальше, в объемлющий try-блок. При обработке исключения можно использовать `throw` без аргумента, что означает передать существующее исключение дальше. Исключение обрабатывается первым подходящим catch-оператором (остальные игнорируются), поэтому важен порядок этих операторов. После обработки исключения управление передается на следующий после catch-команд оператор. Многоточие соответствует любому исключению.

Как правило, аргументами исключений являются классы. Например, при использовании следующей иерархии ошибок

```
struct MathErr {}; // все математические ошибки
struct DivErr: MathErr {}; // ошибки деления
```

`catch (MathErr)` будет перехватывать и исключения типа `DivErr`. Есть стандартная иерархия классов ошибок в `<stdexcept>`. Свойства класса можно естественным образом использовать для передачи уточняющей информации об исключении в обработчик. Например, с классом

```
struct Error {
   int line, pos;
   Error(int l, int p) : line(l), pos(p) {}
};
```

можно сгенерировать исключение командой

`throw Error(3,4);`

перехватить и обработать его catch-оператором

```
catch (Error e) {
   cerr << "Error at line " << e.line << " in position "
      << e.pos << endl;
} //Error at line 3 in position 4
```

Листинг catch.cpp

В `<csetjmp>` содержатся средства для работы со стеком в стиле си, дающие возможности, похожие на исключения.

Дополнительные средства для работы с типами

Вызов вида `T(e)` в выражении, где `T` — имя типа, а `e` — выражение, эквивалентен использованию в этом выражении переменной `t`, определённой декларацией `T t(e);`. Вместо `T(e)` можно использовать `static_cast<T>(e)` . Таким образом, для типов определяемых пользователем такое преобразование типа сводится к вызову соответствующего конструктора. Преобразование `static_cast` не может манипулировать квалификаторами `const` и `volatile`.

Преобразование `const_cast` позволяет отбрасывать или навешивать `const` или `volatile`. Пример.

```
int *p1, *const* p2, **p4, a[2];
const int *p3, i = 4;
main () {
  p2 = p4;
  //p4 = p2; //ошибка
  p4 = const_cast<int**>(p2); // p4 = (int**) p2;
  const_cast<int*const*&>(p4) = p2; // (int*const*&) p4 = p2;
  //p1 = p3; //ошибка
  p1 = const_cast<int*>(p3);
  p1 = (int*) p3;
  p2 = new int * const (a);
  //*p2 = p1; //ошибка
  (int*&) *p2 = p1;
  p3 = p1;
  const_cast<int&>(i) = 7; // (int&) i = 11;
}
```

Листинг type-cnv-a1.cpp

Служебное слово `reinterpret_cast` позволяет преобразовывать числа в указатели и наоборот, например, результатом преобразования `reinterpret_cast<unsigned int*>(4)` будет указатель на целое со значением физического адреса памяти 4. Подобные преобразования без ограничений можно также получить, используя объединения или ссылки.

```
#include<iostream>
using namespace std;
char v[10];
main() {
   for (int i = 0; i < 10; i++)
      v[i] = 7;
  (long&) v[1] = 55; //присваивание 4-м элементам массива,
      //начиная с идексируемого 1, значения
      //машинного представления целого числа 55
   for (int i = 0; i < 10; i++)
      cout << ' ' << (int) v[i];
   cout << endl;
} // 7 55 0 0 0 7 7 7 7 7  -- 32-разрядный режим
// или 7 55 0 0 0 0 0 0 0 7  -- 64-разрядный
```

Листинг type-cnv-a2.cpp

Преобразования `static_cast<T>(e)`, `reinterpret_cast<T>(e)` и `const_cast<T>(e)` сводятся к преобразованию `(T) e` — они лишь позволяют более точно оформить цель преобразования.

Преобразование `dynamic_cast` позволяет проверять допустимость преобразования полиморфного типа во время исполнения программы — ранее рассмотренные преобразования осуществлялись на этапе компиляции. Если преобразование недопустимо, то результат будет 0 для указателя или генерация исключения для других типов.

Пример — печатает пять строк: bad cast, 2, 3, 12, ok.

```
#include <iostream>
using namespace std;
struct A {
  virtual int f() {return 1;}
} a;
struct B : A {
  int f() {return 2;}
} b, *q;
struct C : B {
  int f() {return 3;}
} c;
main() {
  A *p[] = {&a, &b, &c};
```

```
  for (int i = 0; i < 3; i++) {
    q = dynamic_cast<B*>(p[i]);
    if (q)
      cout << q->f() << endl;
    else
      cerr << "bad cast\n";
  }
  a = b;
  A &x = b;
  cout << a.f() << x.f() << endl;
  dynamic_cast<B&>(x);
  try {
    dynamic_cast<B&>(a);
  }
  catch (...) {
    cerr << "ok\n";
  }
}
```

Листинг dynamic-cast.cpp

Вызов преобразований `_cast` синтаксически эквивалентен вызову шаблона функции.

Для работы с типами во время исполнения программы используется служебное слово `typeid`. Для работы с ним необходимо подключать заголовок `<typeinfo>`. Пример.

```
#include<iostream>
#include<typeinfo>
using namespace std;
struct A {
   int i;
};
struct B : A {};
int * const ** a, b[10], (*f)(int), i, &j = i;
main() {
   if (typeid(i) == typeid(j))
      cout << "ok\n";
   cout << typeid(A).name() << ' ' << typeid(B).name() << ' '
      << typeid(a).name() << ' ' << typeid(b).name() << ' '
      << typeid(f).name() << ' ' << typeid(i).name() << endl;
} //1A 1B PPKPi A10_i PFiiE i -- может быть другой
```

Листинг typeinfo.cpp

Инициализация классов с конструкторами

Экземпляры классов можно инициализировать, используя `=`. Пример.

```
struct A {
  int a, b, c;
  A(int x, int y, int z) : a(x), b(y) { c = z;}
  A(int x) { a = b = c = x;}
};
A a1 = A(2, 3, 4); //A a1(2,3,4);
A a2 = a1; //A a2(a1);
A a[2] = {A(1,3,5), A(4)};
A a3 = 6; // A a3(6);
```

При использовании `=` сначала создаётся временный объект, который затем присваивается при помощи `operator=`. Если перед вторым конструктором поставить служебное слово `explicit`, то последняя строчка программы станет недопустимой — нужно будет использовать `A a3(6)`. Точнее `explicit` запрещает неявный вызов конструктора для преобразования типа.

Стандартная библиотека

Контейнеры — это шаблоны, они могут содержать данные любого типа. Они полностью контролируют работу с динамической памятью. Ассоциативные контейнеры (бинарные деревья) могут использовать только данные, для которых определена операция `<` или заданная явно. Вот некоторые стандартные контейнеры:

- `<vector>` — одномерный массив с динамической верхней границей, т. е. данные вектора располагаются в памяти непрерывно, т. е. если `v` — вектор, то `&v[n]` = `&v[0] + n`;
- `<list>` — двунаправленный список;
- `<deque>` — двунаправленная очередь, одномерный массив с динамическими границами. Время добавления или удаления крайних элементов не зависит от размера дека;
- `<queue>` — очередь — это адаптация других контейнеров, например, `list` или `deque`;
- `<stack>` — стек — это адаптация других контейнеров, например, `vector`, `list`, `deque`;
- `<map>` — ассоциативный массив `map` (индексы такого массива называют также ключами) и ассоциативный массив с повторяющимся ключом (`multimap`). Время поиска соответствующего ключу значения пропорционально логарифму от размера массива. Допускается обход всех элементов массива в прямом и обратном порядке;
- `<set>` — множество (`set`) и набор (`multiset`). Множество — это вырожденный ассоциативный массив, в котором значения элементов игнорируются — важно только наличие или отсутствие ключа. В наборах значения ключей могут повторяться. Теоретико-множественные операции определены в заголовке `<algorithm>` .

Операции для контейнеров включают сравнения, присваивание и приводимые в следующей таблице.

Определение операции	**Имена контейнеров**	**Имя операции**
Присваивание с неявным вызовом конструктора с несколькими аргументами	`vector`, `list`, `deque`	`assign`
Прямой доступ к элементам без проверки допустимости индекса	`vector`, `deque`, `map`	`operator[]`
Прямой доступ к элементам с проверкой	`vector`, `deque`	`at`
Добавление элемента в начало	`list`, `deque`	`push_front`
Добавление элемента в конец	`vector`, `list`, `deque`	`push_back`
	`stack`, `queue`	`push`
Добавление элемента	все кроме `queue`, `stack`	`insert`
	`map`	`operator[]`
Удаление начального элемента	`list`, `deque`	`pop_front`
	`queue`	`pop`
Удаление последнего элемента	`vector`, `list`, `deque`	`pop_back`
	`stack`	`pop`
Удаление элемента	все кроме `queue`, `stack`	`erase`
Указатель на 1-й элемент	все кроме `queue`, `stack`	`begin`
Указатель на последний элемент с обходом в обратном порядке	все кроме `queue`, `stack`	`rbegin`
1-й элемент	`vector`, `list`, `deque`, `queue`	`front`
Указатель на следующий за последним элемент	все кроме `queue`, `stack`	`end`
Указатель на предшествующий 1-у элементу при обходе в обратном порядке	все кроме `queue`,`stack`	`rend`
Последний элемент	`vector`, `list`, `deque`, `queue`	`back`
	`stack`	`top`

Текущий размер	все	`size`
Максимально возможный размер	все кроме `queue`, `stack`	`max_size`
Изменить размер	`vector`, `list`, `deque`	`resize`
Зарезервировать размер	`vector`	`reserve`
Зарезервированный размер	`vector`	`capacity`
Проверка на пустоту	все	`empty`
Опустошение контейнера	все кроме `queue`, `stack`	`clear`
Проверка наличия элемента	(`multi`)`map`, (`multi`)`set`	`find`
Счётчик элементов с одинаковым ключом	(`multi`)`map`, (`multi`)`set`	`count`
Начало элементов с одинаковым ключом	(`multi`)`map`, (`multi`)`set`	`lower_bound`
Конец элементов с одинаковым ключом	(`multi`)`map`, (`multi`)`set`	`upper_bound`
Границы элементов с одинаковым ключом	(`multi`)`map`, (`multi`)`set`	`equal_range`
Перенос элементов	`list`	`splice`
Удаление элементов с заданным значением	`list`	`remove`
Соединение списков	`list`	`merge`
Инвертирование списка	`list`	`reverse`
Удаление неуникальных элементов	`list`	`unique`
Быстрая сортировка	`list`	`sort`

Псевдоуказатели при работе с контейнерами принято называть *итераторами*. Итераторы — это объекты классов, ведущие себя подобно указателям (операции `++`, `--`, унарная `*`, сравнения, `->`, `+`, `-`), они имеют некоторые дополнительные операции и не имеют некоторых операции, например, `+` и `-`, для некоторых типов. Ассоциативные контейнеры состоят из пар «ключ-значение». Поле для доступа к ключу называется `first`, а к значению — `second`.

```
#include<iostream>
#include<map>
#include<vector>
#include<string>
using namespace std;
struct D {
   string animal;
   int q;
} data1[] = {{"wolf",4}, {"fox",10}, {"boar",40}, {"lion",1}};
int data2[] = {2, 3, 5, 7, 11, 13, 17};
main() {
   vector<int> v;
   for (int i = 0; i < sizeof(data2)/sizeof(int); i++)
      v.push_back(data2[i]); //заполнение вектора
         //v[i] = data2[i] - нельзя, память еще не выделена
   for (int i = 0; i < v.size(); i++)
      cout << ' ' << v[i];
         //распечатка вектора, прямой доступ
   cout << " -- vector by indices\n";
   for (vector<int>::iterator i = v.begin();
                                          i != v.end(); i++)
      cout << ' ' << *i; //распечатка вектора,
                         //последовательный доступ
   cout << " -- vector, direct order\n";
```

```
   for (vector<int>::reverse_iterator i = v.rbegin();
                                          i != v.rend(); i++)
      cout << ' ' << *i; //распечатка, последовательный
                               //доступ в обратном порядке
   cout << " -- vector, reversed order\n";
   map<string,int> m;
   for (int i = 0; i < sizeof(data1)/sizeof(D); i++)
      m[data1[i].animal] = data1[i].q; //заполнение
                //ассоциативного массива парами
             //слово-количество - исходный порядок теряется!
   m["lynx"] = 2; //добавление элемента
   m["boar"] += 10; //изменение количества
   m.erase("fox"); //удаление элемента
   for (map<string,int>::iterator i = m.begin();
                                          i != m.end(); i++)
      cout << ' ' << i->first << '-' << i->second;
         //распечатка по-порядку
   cout << " -- map, direct order\n";
   for (map<string,int>::reverse_iterator i = m.rbegin();
                                         i != m.rend(); i++)
      cout << ' ' << i->first << '-' << i->second;
         //распечатка в обратном порядке
   cout << " -- map, reversed order\n";
   if (m.find("lion") != m.end()) //проверка наличия элемента
      cout << "a lion is found ";
   if (m.find("tiger") == m.end())
      cout << "but tiger is absent\n";
} /*
 2 3 5 7 11 13 17 -- vector by indices
 2 3 5 7 11 13 17 -- vector, direct order
 17 13 11 7 5 3 2 -- vector, reversed order
 boar-50 lion-1 lynx-2 wolf-4 -- map, direct order
 wolf-4 lynx-2 lion-1 boar-50 -- map, reversed order
a lion is found but tiger is absent */
```

Листинг stdcpplib.cpp

Функция `find` в случае успеха возвращает итератор, указывающий на найденный элемент, а в случае неуспеха — результат функции `end`.

Многомерные динамические массивы можно конструировать, использую образцы, подобные `deque<vector<int> > mda` или для динамичности одной размерности — `vector<int> mda[10];`.

Заголовок `<string>` содержит средства для строк: все сравнения (6 стандартных и `compare` с результатом как у `strcmp`), присваивание, `+`, итераторы, прямой доступ. Тип `string` основан на шаблоне `basic_string`, имеет много общих операций с шаблоном `vector`: `assign`, `size`, `max_size`, `resize`, `empty`, `capacity`, `reserve`, `clear`, `operator[]`, `at`, `insert`, `erase`. Преобразование в си-строку производится функцией `c_str`. Вставка в конец — `append` и `+=`. Поиск — `find`, которая возвращает `string::npos` (может использоваться как индикатор конца строки) при неуспехе. Замена — `replace`. Выделение подстроки — `substr`. Вместо `size` можно использовать `length`. Пример.

```
#include <iostream>
#include <string>
#include <cstring> //для strcpy
using namespace std;
main() {
   string s1("Hello"), s2, s3(1,'a'), s4 = "Hello World";
   char sc[100];
   cout << s1.length() << ' ' << s2.length() << ' '
      << s3.length() << ' ' << s4.length() << endl; //5 0 1 11
   if (s2.empty())
```

```
    s2 = s1;
  s2 += s2 + s4.substr(6,5);
  cout << s3 << s2 << (s4.compare(s2) < 0) << (s1 < s2)
    << s4.substr(5, string::npos) << endl;
                                  //aHelloHelloWorld11 World
  s2.erase(2, 7); //Held
  s2[1] = 'o';
  s4.insert(6, "Big ");
  cout << s2 + ' ' + s4 << endl; //Hold Hello Big World
  cout << s4.find("Big") << endl; //6
  if (s4.find("Small") == string::npos)
    cout << "missed\n";
  s4.replace(0, 5, "Hi!");
  cout << s4 << endl;  //Hi! Big World
  strcpy(sc, s4.c_str());
  s4.clear(); //s4 = "";
  s4 = sc;
  cout << s4 << endl; //Hi! Big World
}
```

Листинг string.cpp

Первый аргумент `substr`, `replace`, `erase`, `insert` — это число-позиция (нумерация с 0). Второй аргумент `substr`, `replace`, `erase` — это длина, если взять длину большей доступной, например, `npos`, то будет взята вся оставшаяся часть строки. Для поиска можно ещё использовать функции `rfind` — обратный поиск, `find_first_of` — поиск 1-го символа из заданного строкой множества, `find_last_of` — поиск последнего символа, `find_first_not_of` — 1-го символа не из множества, `find_last_not_of` — последнего не из множества. Все функции поиска могут использовать опциональный второй аргумент — позицию начала поиска. Примеры,

```
string s = "bcabdeabfd";
s.find("ab", 3); //6
s.rfind("ab"); //6
s.find_first_of("adf"); //2
s.find_last_of("abf"); //8
s.find_last_of("abf", 5); //3
s.find_first_not_of("abcd"); //5
s.find_last_not_of("abcd"); //8
```

В заголовке `<cctype>` объявлены средства классификация символов: предикаты `isalpha` (буквы), `isupper`, `isdigit`, `isxdigit`, `isspace` (разделители) и подобные. Кроме того, `<cctype>` содержит функции преобразования регистра букв, `toupper` и `tolower`.

Заголовок `<comlpex>` вводит поддержку комплексных чисел

Заголовок `<valarray>` вводит числовые векторы, которые похожи на `vector` со специальными операциями. Позволяют работать со срезами (n-ми членами последовательности, что, в частности, дает возможность работать с вектором как с матрицей), подмножествами компонент вектора и массивами индексов.

Операции: `sum` — сумма элементов, `shift` — логический сдвиг элементов, `cshift` — циклический сдвиг, `apply` — применение заданной функции к каждому элементу, `=`, `-=`, `+`, ` `, `!`, `*`, `/`, `&`, `==`, `cos`, `atan`, `exp` и т. п. — применение заданных операций к каждому элементу, `min`, `max` — наименьший, наибольший элемент

Заголовок `<algorithm>` — это типовые алгоритмы, позволяют работать с практически любыми контейнерами и избегать явных циклов. Некоторые операции:

- `for_each` — выполнение операции для каждого элемента последовательности;
- `find` — поиск элемента в последовательности;
- `count` — счётчик вхождений;
- `equal` — сравнение последовательностей;
- `search` — поиск подпоследовательности в последовательности;
- `swap` — замена двух элементов;
- `fill` — заполнение последовательности;
- `remove` — удаление элементов;
- `unique` — удаление равные смежных элементы;
- `reverse` — инвертирует последовательность;
- `rotate` — вращает последовательность;
- `random_shuffle` — случайно перемещает элементы последовательности, “тасует”;

- `sort` — быстрая сортировка;
- `binary_search` — поиск элемента в отсортированной последовательности;
- `includes` — проверка на включения подмножества в множество;
- `set_intersection`, `set_difference`, `set_symmetric_difference`, `set_union` — соответственно пересечение, разность, симметрическая разность и объединение множеств;
- `min`, `max` — меньшее, большее из двух;
- `min_element`, `max_element` — меньшее, большее значение в последовательности;
- `lexicographical_compare` — словарное сравнение элементов последовательностей;
- `next_permutation`, `prev_permutation` — соответственно следующая и предыдущая перестановка, для полного перебора вариантов.

Для некоторых операций (`sort`, `random_shuffle`, ...) требуется прямой доступ к элементам.
Пример.

```
#include<iostream>
#include<list>
#include<algorithm>
using namespace std;
int data[] = {2, 3, 5, 7, 11};
const int N = sizeof(data)/sizeof(int);
struct Print {
   int z;
   void operator()(int i) {cout << ' ' << i;}
   Print() {z = 1;}
   Print(const Print&) {z = 0;}
   ~Print() {if (z) cout << endl;}
}; //это класс-функция, лучше сделать шаблоном
   //и не превязывать к int
   //если p - объект класса Print, то p(5) -- печатает 5
struct Fill {
   list<int>::iterator p;
   Fill (list<int> &l) {p = l.begin();}
   void operator()(int i) {*p++ = i;}
};
struct Plus {
   int v;
   Plus (int i) : v(i) {}
   void operator()(int &i) {i = v += i;}
};
main() {
   list<int> l(N), l2;
   for_each(data, data + N, Print());  //2 3 5 7 11
   for_each(data, data + N, Fill(l));
   l2 = l;
   for_each(l.begin(), l.end(), Plus(4));
   for_each(l.begin(), l.end(), Print()); //6 7 9 11 15
   reverse(l.begin(), l.end());
   for_each(l.begin(), l.end(), Print()); //15 11 9 7 6
   rotate(l.begin(), ++l.begin(), l.end());
       //циклический сдвиг влево
   for_each(l.begin(), l.end(), Print()); //11 9 7 6 15
   cout << *min_element(l.begin(), l.end()) << endl; //6
   cout << *find(l.begin(), l.end(), 11) << endl; //11
   cout << (find(l.begin(), l.end(), 10) == l.end()) << endl;
       //1
   cout << count(l.begin(), l.end(), 4) << endl; //0
}
```

Листинг algorithm.cpp

Таким образом, `find(l.begin(), l.end(), 10) == l.end()` — это форма записи `l.find(10) == l.end()`. Если определить макрос `#define xfind(value,obj) (obj.find(value)==obj.end())`, то можно будет писать ещё компактнее — `xfind(10, l)`.

Стандарт 2011 года вместо `l.begin()` и `l.end()` допускает ещё и запись `begin(l)` и `end(l)`.

Функция-класс типа `Print`, `Fill`, `Plus` имеет специальное названия — (лексическое или функциональное) замыкание. Такое замыкание представляет собой функцию вместе с некоторым окружением. Замыкание можно использовать как обычную функцию, но у каждого экземпляра замыкания своё окружение.

```
Plus a(5), b(7);
int i = 5, j = 8;
a(i);  //i += 5;
b(j);  //j += 8;
cout << i << ' ' << j << endl; //10 15
```

Замыкания делают ненужным использование глобальных переменных функциями, т. е. позволяют избежать нежелательного побочного эффекта. В последнем примере программы вместо замыкания `Plus` в `for_each` можно было бы использовать функцию `pluszg`, записывая `for_each(l.begin(), l.end(), pluszg);`.

```
int z = 4; //глобальная переменная
void pluszg(int& i) {
   i = z += i;
}
```

Тип значение у `pluszg` может быть любым — он игнорируется при использовании таким образом. Вместо такой функции с побочным эффектом можно было бы использовать функцию `pluszs` со статическими переменными.

```
void pluszs(int& i) {
   static int z = 4;
   i = z += i;
}
```

Такая функция ничем не хуже экземпляра замыкания, но если понадобится иметь несколько одинаковых функций с разными окружениями, то замыканию удобной альтернативы нет.

Замыкания позволяют связываться с внешними объектами через поля-ссылки.

```
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;
struct Print {
   string &title;
   string notitle = "";
   int z = 0;
   void operator()(int i) {
      if (z) cout << ' '; else cout << title;
      cout << i;
      z = 1;
   }
   Print(): title(notitle) {}
   Print(const Print& p): title(p.title) {}
   Print(string &s): title(s) {}
   ~Print() {
      if (z) cout << endl;
   }
};
main() {
   vector<int> v = {1, 2, 3, 4, 5};
   string s = "list: ";
   Print p(s);
   for_each(begin(v), end(v), p);  //list: 1 2 3 4 5
   s = "values: ";
   for_each(begin(v), end(v), p);  //values: 1 2 3 4 5
}
```

Листинг closure.cpp

Рассмотрим ещё пример — генерацию всех вариантов раздачи подарков.

```
#include <iostream>
#include <algorithm>
//#include <string>
#include <vector>
using namespace std;
main() {
   vector<string>
      guests = {"Mike", "Jane", "David", "Judit"},
      gifts = {"dog", "candy", "hat", "fruit"};  //std=c++11
   sort(begin(gifts), end(gifts));
   do {
      for (int i = 0; i < guests.size(); i++)
         cout << "A " << gifts[i] << " for " << guests[i] << endl;
      cout << endl;
   }
   while (next_permutation(begin(gifts), end(gifts)));
}
```

Листинг algorithm2.cpp

В <cstdlib> содержатся средства в стиле си, в частности, для поиска элемента в отсортированной последовательности — bsearch и для быстрой сортировка — qsort.

Регулярные выражения

Существует несколько способов для использования таких выражений. Рассмотрим самые известные из них:

1) си-библиотека с заголовком ⟨regex.h⟩. Она соответствует стандарту POSIX. Не поддерживаются многие расширения;
2) стандартная библиотека си++ по стандарту от 2011 года с заголовком ⟨regex⟩*;
3) библиотека си++ boost с заголовком ⟨boost/regex.hpp⟩;
4) си-библиотека языка перл с заголовком ⟨pcre.h⟩. Поддерживает практически все способы работы с регулярными выражениями. Эту библиотеку можно использовать в объектном синтаксисе, используя заголовок ⟨pcrecpp.h⟩ от Google.

При работе с регулярными выражениями их сначала нужно скомпилировать. Рассмотрим последнюю библиотеку поподробнее. Она вводит пространство имён pcrecpp. Тип-объект регулярного выражения — RE. У конструктора один аргумент, задающий регулярное выражение. Можно использовать и второй аргумент с опциями. Для задания аргументов можно использовать как си-строки, так и объектные. Вызов конструктора — затратная по времени операция, так как конструктор компилирует заданное регулярное выражение.

Компилировать программы, использующие ⟨pcrecpp.h⟩, нужно с ключом -lpcrecpp.

Метод FullMatch используется для сопоставления текстовой строки и регулярного выражения. Если сопоставление успешно, то результат 1, если нет — 0. Например,

```
RE re("h.*o");
cout << re.FullMatch("hello") << endl;
```

должно напечатать 1.

У FullMatch могут быть и дополнительные параметры, позволяющие выделять части строки, соответствующие выделенным частям регулярного выражения. Например, следующий код должен напечатать ruby1234.

```
int i;
string s;
RE re("([a-z]+):([0-9]+)");
if (re.FullMatch("ruby:1234", &s, &i))
   cout << s << i << endl;
```

Если соответствие какой-то части регулярного выражения находить не надо, то вместо параметра-указателя используется 0. При использовании нестроковых переменных при их переполнении или несоответствии результат сопоставления будет 0. В качестве переменных можно использовать указатели на любые скалярные типы или на типы string, stringPiece и T. Можно использовать до 16 переменных. Если нужно больше, то следует использовать механизм интерфейса DoMatch. Тип T — это любой объектный, в котором есть член bool ParseFrom(const char*, int).

* Не поддерживается gcc от 2015.

Метод `QuoteMeta` используется для цитирования всех символов заданной строки. Например, `string quoted = RE::QuoteMeta("1.5-2.0?");` сделает значением строки *quoted* выражение `"1\.5\-2\.0\?"`.

Метод `PartialMatch` используется для частичного сопоставления, например,

```
cout << RE("ell").PartialMatch("Hello") << endl;  //1
```

должно напечатать 1. В примере

```
int i;
RE re("([0-9]+)");
if (re.PartialMatch("x*100 + 20", &i))
   cout << i << endl;
```

должно напечататься 100.

Регулярному выражению можно сопоставить ряд опций:

- `PCRE_CASELESS` — игнорировать различие в регистре букв;
- `PCRE_MULTILINE` — работать с многострочным текстом;
- `PCRE_DOTALL` — точка соответствует и концу строки;
- `PCRE_DOLLAR_ENDONLY` — $ соответствует только концу текста;
- `PCRE_EXTRA` — сообщать об ошибке, если после \ следует символ, образующий неизвестную escape-последовательность;
- `PCRE_EXTENDED` — игнорировать пробелы и табуляции;
- `PCRE_UTF8` — использовать Unicode;
- `PCRE_UNGREEDY` — инвертировать смысл `*?` и `*`;
- `PCRE_NO_AUTO_CAPTURE` — не считать выражение в скобках выделенным. Можно использовать `(?:` и `)` вместо простых скобок только для группировки без выделения;

Опции можно установить при конструировании, например, `RE re("ши", UTF8().set_multiline(1));`.

Для работы с опциями предназначен объект `RE_Options`. Рассмотрим пример.

```
RE_Options opts(PCRE_MULTILINE | PCRE_DOLLAR_ENDONLY);
opts.set_caseless(1).set_utf8(1).set_multiline(0);
cout << opts.caseless();  //1
cout << RE("Hello", opts).PartialMatch("hello world") << endl;  //1
```

Методы `Consume` и `FindAndConsume` полезны для сканирования строки. Они работают с типом `StringPiece`, представляющем собой надстройку над сканируемой строкой. Данные этого типа можно рассматривать как ещё неотсканированную часть строки, доступную через метод `data`, который преобразует эту часть в си-строку. Нельзя менять базовую строку, пока используется связанный с ней объект типа `StringPiece`. `Consume` выбирает с начала текста данные, соответствующие регулярному выражению, и отбрасывает их от дальнейшего рассмотрения.

```
string contents = "2 5 7 11";
StringPiece input(contents);
int value;
RE re("([0-9]+) *");
while (re.Consume(&input, &value))
   cout << value;  //25711
```

`FindAndConsume` выбирает данные не сначала, а с найденной позиции соответствия, что, например, позволяет выделить все слова или числа из текста.

```
string contents = "2 a 5 bc 7 d 11";
StringPiece input(contents);
string value;
RE re("([a-z]+)");
while (re.FindAndConsume(&input, &value))
   cout << value;  //abcd
```

Можно выделять из текста числа в разных системах счисления, используя соответствующие методы. Рассмотрим пример.

```
int a, b, c, d, e;
RE re("(.*) (.*) (.*) (.*) (.*)");
re.FullMatch("100 40 0100 0x40 64", Octal(&a), Hex(&b), CRadix(&c),
   CRadix(&d), &e);
cout << a << b << c << d << e << endl; //6464646464
```

Метод `CRadix` соответствует нотации чисел языка си.

Методы `Replace` и `GlobalReplace` позволяют делать замены в тексте. На выделенные подвыражения можно ссылаться, использую обозначения от \1 до \9, а \0 означает весь текст, соответствующий регулярному выражению. Метод `GlobalReplace` делает замены сразу по всему тексту, а `Replace` заменяет только первое соответствие.

```
string s = "yabba dabba doo";
RE("(b+)a").GlobalReplace("a\\1", &s);
cout << s; //yaabb daabb doo
```

Рассмотрим программу, удаляющую из текстового файла на си++ все комментарии.

```
#include <iostream>
#include <string>
#include <pcrecpp.h>
using namespace std;
using namespace pcrecpp;
int main() {
   RE re("^(.*?)(\"(\\\\.|[^\"\\\\])*\"|'(\\\\.|[^'\\\\])*'|\
      //[^\\n]*|/\\*.*?\\*/)", UTF8().set_multiline(1).set_dotall(1));
   string s = "", r1, r2, r = "";
   while (!cin.eof()) {
      const int bsz = 1024;
      char b[bsz + 1];
      cin.read(b, bsz);    //читать не более bsz байт в b
      b[cin.gcount()] = 0;
        //метод gcount возвращает число байт, прочитанных read или get
      s += b;
   }
   StringPiece in(s);
   while (re.Consume(&in, &r1, &r2)) {
      r += r1;
      if (r2[0] != '/') r += r2;
   }
   r += in.data();
   cout << r;  //можно было вместо двух строк одну cout << r << in;
}
```

Листинг no-c++-comments.cpp

Возможности базового си-интерфейса через ⟨`pcre.h`⟩ шире объектного.

Графика

Стандартных средств для рисования нет. Существует множество библиотек средств для работы с графикой:

X11 — низкоуровневые средства, остальные библиотеки, использующие X Window работают через них, си-ориентированная;

GTK+ — на ней написан менеджер окон GNOME, ориентирована на работу в стиле си, мультиплатформенная, имеет расширения для ООП;

OpenGL — ориентирована на разработку игр и полноэкранный режим, прямая поддержка аппаратуры видеосистемы, интерфейс в стиле си, низкоуровневая, с ней работают, как правило, через другие библиотеки;

Qt — на ней написан менеджер окон KDE, мультиплатформенная, объектно-ориентированная, использование метакомпилятора;

wxWidgets – ориентирована на использование графических элементов, свойственных ОС разработки;

SDL — ориентирована на разработку игр, поддержка работы со звуком и мультимедия, на си;

Fltk (“fulltick”) — мультиплатформенная, поддержка GL, небольшой размер, статическая компоновка, предоставляет большое количество виджетов (widget — элемент изображения), объектно-ориентированная. Пример.

```
#include <FL/Fl.H>
#include <FL/Fl_Window.H>
#include <FL/Fl_Box.H>
int main(int argc, char **argv) {
   Fl_Window *window = new Fl_Window(340, 180); //размер окна
      Fl_Box *box = new Fl_Box(20, 40, 300, 100, "Hello, World!");
         //координаты углов прямоугольника для текста
```

```
      box->box(FL_UP_BOX);
      //box->labelfont(FL_BOLD + FL_ITALIC); //гарнитура шрифта
      box->labelsize(36);  //размер шрифта
      //box->labeltype(FL_SHADOW_LABEL);
   window->end();
   window->show(argc, argv);
   return Fl::run(); //передача управления графической системе
}
```

Листинг fltk1test.cpp

Характерной чертой работы с графической системой является необходимость работать в согласовании с устанавливаемыми этой системой соглашениями. Все существующие популярные средства определяют среду программирования, основанную на обработке сообщений. Сама графическая система может генерировать и обслуживать сообщения и поэтому не нужно, например, программировать стандартные изменения размеров окна, перемещения окна и т. п. Можно создавать новые события и переопределять или доопределять реакции на стандартные. Как и в рассмотренном примере в любой программе для графической среды управление должно передаваться этой среде — программируются только определения новых событий, виджетов и т. п. Последние может приводить к тому, что программы для некоторых систем (wxWidgets, Microsoft Windows SDK) могут не содержать определения функции `main`, которая определяется графической библиотекой.

Расширим пример добавлением кнопки и функции, вызываемой при нажатии на эту кнопку.

```
#include <FL/Fl_Button.H>
void button_action(Fl_Widget *w) {
   w->position(w->x() + 1, w->y()); //сдвиг кнопки
   w->parent()->redraw();
}
```

```
      Fl_Button *button = new Fl_Button(10, 20, 30, 70, "Ok");
      button->labelsize(7);
      button->callback(button_action);
```

Как правило, для библиотеки для поддержки работы с графикой существует специальный редактор виджетов, где по нарисованным мышкой из заданных элементов виджетов создаются соответствующие скелетные программы на выбранном языке программирования. Для fltk такой редактор называется `fluid`.

Новые возможности по стандарту 2011 года

Перемещающие конструкторы и присваивания

Рассмотрим присваивание `A=B+C`. Сначала создаётся (конструируется) временный объект, равный сумме `B` и `C`, затем этот объект копируется в `A` и затем уничтожается. Поучается очень не эффективно, если размер объекта велик. Было бы естественнее просто перенести данные из временного объекта. Для проведения такого эффективного присваивания следует решить проблемы правых и левых значений (rvalue и lvalue). Левые значения существуют независимо от выражений, где их используют, им обычно сопоставлены идентификаторы и им можно присваивать значения. Правые значения — это всё прочее, например, константы или временные переменные, возникающие при вычислении выражений. По константной ссылке можно передавать любые выражения, а по неконстантной только левые значения.

Пример с перегрузкой функций.

```
#include <iostream>
using namespace std;
void f(int &a) {
   cout << "1\n";
}
void f(const int &a) {
   cout << "2\n";
}
main() {
   int a = 5;
   f(a); //вызовет первую функцию
   f(7); //вторую
   f(2 + 2); //вторую
}
```

Листинг byref.cpp

Первую функцию нельзя использовать с аргументом 7 или $2 + 2$, а вторую можно с любыми. Но в теле второй функции нельзя менять аргумент.

Если нужно при присваивании перемещать временный объект, то его надо передавать по ссылке. При перемещении он меняется, поэтому его нужно передавать как неконстантную ссылку, что невозможно для временного объекта.

Специальный синтаксис ссылки на правое значение решает проблему.

```
void f(int &&a) {
   cout << "3\n";
}
```

В такую `f` можно передавать по ссылке только правые значения. Эта функция будет вызвана при аргументе $2 + 2$. С этим аргументом совместима и вторая функция, но новая, третья, подходит лучше.

Рассмотрим содержательный пример создания класса для матриц размера 7×7.

```
#include <iostream>
#include <cstring>
using namespace std;
class Array7 {
   int *p;
public:
   Array7(void) {//конструктор
      cout << "create constructor\n";
      p = new int[7*7];
   }
   Array7(const Array7& a) {//копирующие конструктор
      cout << "copy constructor\n";
      p = new int[7*7];
      memcpy(p, a.p, 7*7*sizeof(int));
   }
   Array7& operator=(const Array7& a) {  //копирующее присваивание
      cout << "copy assignment\n";
      memcpy(p, a.p, 7*7*sizeof(int));
      return *this;
   }
#if __cplusplus > 201100
   Array7(Array7&& a) {  //перемещающий конструктор
      cout << "move constructor\n";
      p = a.p;
      a.p = 0;
   }
   Array7& operator=(Array7&& a) {  //перемещающее присваивание
      cout << "move assignment\n";
      delete [] p;
      p = a.p;
      a.p = 0;
      return *this;
   }
#endif
   ~Array7() {
      cout << "destructor\n";
      delete [] p;
   }
   int& operator()(int i, int j) {
      return p[i*7 + j];
   }
   Array7 operator+(Array7 a) {
      cout << "+\n";
      for (int i = 0; i < 7; i++)
         for (int j = 0; j < 7; j++)
```

```
            a.p[i*7 + j] += p[i*7 + j];
      return a;
   }
   friend ostream& operator<<(ostream& out, const Array7& a) {
      for (int i = 0; i < 7; i++) {
         for (int j = 0; j < 7; j++)
            out << a.p[i*7 + j] << '\t';
         out << endl;
      }
      return out;
   }
};
main() {
   Array7 A, E;
   for (int i = 0; i < 7; i++)
      for (int j = 0; j < 7; j++) {
         A(i,j) = i + j;
         E(i,j) = i == j;
      }
   Array7 B(A), D(A + E);
   A = D;
   B = B + E + D;
   cout << "2(A + E) =\n" << B;
}
```

Листинг moveops.cpp

При использовании опции `-std=c++11` компилятора `g++` будут задействованы перемещающие операции.

Новые значения служебных слов delete, default и auto

При помощи `delete` можно указывать запрет на перегрузку функции, например,

```
int f(double a) {return a;}
int f(int) = delete;
```

Теперь, при вызов `f(2)` — ошибочен. Без запрещающего определения 2 было бы преобразовано в вещественное и был бы вызов `f(2.0)`.

Слово `default` можно использовать для явного указания на использование автоматически генерируемых функций: конструктора, присваивания, ...

```
struct A {
   int a;
   A() = default;
   A(int b) : a(b) {}
} c;
```

Без строки с `default` будет ошибка, т.к. есть правило, если в классе определить какой-то конструктор, то простейшие конструкторы неявно не генерируются.

Слово `auto` позволяет выводить тип, что полезно для сокращений.

```
auto a = 7; //будет выведен тип int
map<int,int> m;
for (auto i = m.begin(); i != m.end(); ++i)
    cout << i->second << endl; //auto вместо map<int,int>::iterator
```

Новые служебные слова decltype, constexpr, nullptr

Слово `decltype` позволяет выводить тип по аргументу.

```
int a;
decltype(a) b;  //int b ;
```

Слово `constexpr` — это спецификатор для функции для обозначения того, что результат функции — это константа, вычислимая во время компиляции.

```
constexpr int n() {return 5;}
int array[n()];  //без constexpr - ошибка
```

Слово `nullptr` вводится для обозначения нулевого указателя для избежания неоднозначности при перегрузке функций.

```
int f(void*);
int f(int);
f(0);  //f(int)
f(nullptr);  //f(void*)
```

При отсутствии `f(void*)` `f(nullptr)` не вызовет `f(int)` — случится ошибка. При отсутствии `f(int)` `f(0)` вызовет `f(void*)`.

Инициализация

Можно инициализировать структуру с конструкторами, при соблюдении некоторых ограничений, например, если все конструкторы создаются автоматически и у класса нет виртуальных функций.

Можно из одного конструктора вызывать другой.

```
class SomeType  {
    int number;
public:
    SomeType(int new_number) : number(new_number) {}
    SomeType() : SomeType(42) {}
};
```

Можно наследовать конструкторы.

```
struct BaseClass {
   BaseClass(int value);
};
struct DerivedClass: BaseClass {
   using BaseClass::BaseClass;
};
```

Члены класса можно инициализировать согласно примеру.

```
class SomeClass {
   int value = 5;
public:
   SomeClass() {}
   explicit SomeClass(int new_value) : value(new_value) {}
};
```

Все конструкторы, которые явно не устанавливают `value`, неявно устанавливают значение `value` в 5.

Синтаксис общей инициализации позволяет единообразно задавать начальные значения классам с конструкторами и без.

```
struct BasicStruct {
   int x;
   double y;
};
class AltStruct {
   int x;
   double y;
public:
   AltStruct(int x0, double y0) : x{x0}, y{y0} {}
};
BasicStruct var1 = {5, 3.2};
AltStruct var2 = {2, 4.3};  //var2(2, 4.3);
```

Для поддержки инициализации произвольных классов в `<initializer_list>` определяются соответствующие средства, имеющие больший приоритет, чем общая инициализация.

```
#include <iostream>
#include <vector>
#include <initializer_list>
using namespace std;
struct T {
   int v[7], size;
   T(initializer_list<int> l): size(l.size()) {
```

```
      for (auto i = l.begin(); i != l.end(); ++i)
         v[i - l.begin()] = *i;
   }
   void append(initializer_list<int> l) {
      for (auto i = l.begin(); i != l.end(); ++i)
         v[size++] = *i;
   }
};
struct S {
   vector<int> v;
   S(initializer_list<int> l) : v(l) {}
   void append(initializer_list<int> l) {
      v.insert(v.end(), l.begin(), l.end());
   }
};
main() {
    T t = {2, 3, 5, 7};
    t.append({11, 13, 17});
    for (int i = 0; i < 7; ++i)
       cout << t.v[i] << ' ';   //for (int i: t.v) cout << i << ' ';
    cout << endl;
    S s = {1, 2, 3, 4, 5};
    s.append({6, 7, 8});
    for (auto i = s.v.begin(); i != s.v.end(); ++i)
       cout << ' ' << *i;   //for (auto i: s.v) cout << ' ' << i;
    cout << endl;
    vector<int> v(initializer_list<int>({1, 3, 5}));
    cout << v.size() << v[0] << v[1] << v[2] << endl; //3135
}
```

Листинг initializer-list.cpp

Новые средства синтаксиса

Цикл `for` можно использовать, указывая множество значений для переменной, например,

```
int array[5] = {1, 2, 3, 4, 5};
for (int &i: array) i *= 2; //удвоит все числа в массиве
```

В последнем примере со списком инициализации циклы `for` можно было заменить на закомментированные, более короткие операторы.

Тип функции можно указывать после параметров, что может быть необходимым при определении некоторых шаблонов.

```
auto f(int i) -> int {return i};
```

С шаблонами можно использовать `>>`, например, `vector<vector<int>> v;`.

В шаблонах можно использовать списки параметров неопределенной длины. Рассмотрим пример с рекурсивным шаблоном, симулирующим функцию printf.

```
#include <iostream>
using namespace std;
void print(const char *s) {
   while (*s) {
      if (*s == '%' && *++s != '%')
         throw "wrong format string or extra arguments";
      cout << *s++;
   }
}
template<class T, class... Args>
void print(const char *s, T value, Args... args) {
   while (*s) {
      if (*s == '%') {
         if (*++s == 0)  //можно добавить проверки на знаки формата
```

```
            throw "trailing %";
         else if (*s != '%') {
            cout << value;
            ++s;
            print(s, args...);
            return;
         }
      }
      cout << *s++;
   }
   throw "extra arguments";
}
main() {
   try {
      print("Hello World\n");
      print("%s %d times\n", "Hello World", 100);
      print("Hello World\n%", 100); //error
      print("%d");                  //error
      print("%", 5);                //error
      print("%");                   //error
   }
   catch (const char *s) {
      cerr << "error: " << s << endl;
   }
}
```

Листинг vartemplate.cpp

Лексема ... используется с шаблонами похожим на функции с неограниченным числом аргументов образом. Отличие в необходимости указания имени для списка параметров в шаблоне.

Энки

В заголовке <tuple> вводятся средства (get, tie, make_tuple, tuple_element, tuple_size, ...) для работы с упорядоченными энками. Энки реализуются на основе шаблонов с неопределённым числом параметров.

```
#include<tuple>
#include<string>
#include<iostream>
using namespace std;
main() {
   tuple<char, short, const char *> tuple1('X', 2, "Hola!");
   get<0>(tuple1) = 'Y';
   string s = get<2>(tuple1);
   cout << get<0>(tuple1) << ' ' << s << endl; //Y Hola!
   auto record = make_tuple("Hari Ram", "New Delhi", 3.5, 'A');
   string name;
   tuple_element<2, decltype(record)>::type gpa;
   char grade;
   tie(name, ignore, gpa, grade) = record;  //ignore пропускает позицию
   cout << name << ' ' << gpa << grade << endl; //Hari Ram 3.5A
   cout << "size = " << tuple_size<decltype(record)>::value << endl; //4
   cout << "element #1 is " << get<1>(record) << endl; //New Delhi
}
```

Листинг tuple.cpp

Однотипные энки можно сравнивать и присваивать.

В заголовке <map> есть функция pair, а в <utility> более удобная функция make_pair для работы с частным случаем энок — парами.

```
#include <utility>
#include <map>
```

```
#include <iostream>
using namespace std;
main () {
  pair<int,int> pair1 = make_pair(10, 20), pair2, pair3;
  pair2 = pair<int, int>(10.4, 'A');  //pair2 = make_pair(10.4, 'A');
  pair1.first = pair2.second;
  pair3 = pair1;
  cout << pair3.first << ", " << pair3.second << endl; //65, 20
  cout << (pair1 == pair2) << endl; //0
  map<int,int> map1;
  map1.insert(pair2); //map1[pair2.first] = pair2.second;
  cout << map1[10] << endl; //65 = 'A'
}
```

Листинг pairs.cpp

Лямбда-функции

В 1930 Черч предложил специальную форму записи для функции, вводить символическое обозначение для которой не нужно. Например, для записи значения функции $\sin 2x$ в точке 1 используется следующая конструкция $\lambda \sin 2x(1)$. Сегодня для записи того же чаще используется конструкция $\sin 2x|_{x=1}$.

Использование лямбда-записи позволяет избавить программу от введения ненужных имен для функций. Получается, что лямбда-функция — это просто функция без имени или анонимная функция.

В следующем примере, печатающем 125 и 11, показано примитивное использование лямбда-функций.

```
#include <iostream>
using namespace std;
main() {
   int i = [](int x) {return x*x*x;}(5);  //i = 5*5*5
   cout << i << ' '
      << [](int x, int y){return x + y;}(5, 6)  //5 + 6
      << endl;
}
```

Листинг lambda1.cpp

Лямбда-функции очень полезны с конструкциями, подобными предлагаемым заголовком `<algorithm>`. Без анонимных функции нужно искусственно определять функции с именами. Например, нужно возвести все числа в заданном списке в квадрат.

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
void sqr(int &i) {i *= i;}
struct Print {
   int z = 0;
   void operator()(int i) {
      if (z) cout << ' ';
      cout << i;
      z = 1;
   }
   Print() {}
   Print(const Print&) {}
   ~Print() {
      if (z) cout << endl;
    }
};
main() {
   vector<int> v = {1, 2, 3, 4, 5};
   for_each(begin(v), end(v), Print());  //1 2 3 4 5
   for_each(begin(v), end(v), sqr);
   for_each(begin(v), end(v), Print());  //1 4 9 16 25
```

```
   for_each(begin(v), end(v), [](int &i){i *= i;});
   for_each(begin(v), end(v), Print());  //1 16 81 256 625
}
```

Листинг lambda2.cpp

Лямбда-функции являются замыканиями. Их среду можно связывать с другими переменными. Связь может быть по значению или по ссылке. Связи задаются в квадратных скобках. Лямбда-функция не может работать с переменными вне замыкания без явной связи с ними — этим она отличается от обычных функций. Связь устанавливается в момент определения функции. Рассмотрим типовые случаи таких связей:

`[]` — нет связей, функции недоступны внешние переменные;
`[x, &y]` — связь с внешней переменной x по значению и с y по ссылке;
`[&]` — все используемые внешние переменные доступны через связь по ссылке;
`[=]` — все используемые внешние переменные доступны через связь по значению;
`[&, x]` — x связывается по значению, а остальные используемые по ссылке;
`[=, &z]` — z связывается по ссылке, а остальные используемые по значению.

Лямбда-функция без связей совместима с обычными функциями.

```
#include <iostream>
using namespace std;
void g5(long (*fp)(int)) {
   cout << (*fp)(5) << endl;
}
main() {
   int d = 5, i = 7;
   auto f = [d, &i](int x)->long {return i = d + x;};
      //d в функции - это константа 5
   auto h = [](int x) {return (long)x*x;};
   d = -5;
   g5(h);  //25, f так использовать нельзя
   g5([](int z)->long {return z*z*z;}); //125
   cout << f(11) << endl;  //16, i=16
   cout << d + i << endl;  //11
   cout << [](int x, int y){return x + y;}(5, 7) << endl; //12
```

Листинг lambda3.cpp

Основные отличия си от си++

Нет операций `::`, `.*`, `->*`.

Нет ссылок, наследования, классов, шаблонов, пространств имён, перегрузки функций.

Перечисления считаются разновидностью целого типа.

Переменные можно объявлять только в глобальной области или в начале блока.

Нет преобразований типов `_cast` и таких, которые используют имя типа как имя функции.

Перед именами типов структур, объединений и перечислений нужно всегда ставить соответственно `struct`, `union` и `enum`.

Структуры и объединения могут иметь только компоненты-переменные, поля.

Нет статических членов в структурах.

Если тип объекта не указан, то считается, что его тип — `int`.

Нет исключений. Вместо них можно использовать функции `<setjmp.h>` `setjmp` и `longjmp`.

Нет операций `new` и `delete`. Вместо них можно использовать функции `<stdlib.h>` `calloc`, `malloc`, `free`, `realloc`.

Нет аргументов функции по умолчанию.

Все указатели и целые числа считаются совместимыми, но разнотипные присваивания вызывают предупреждения.

Имена файлов заголовков получаются отбрасыванием первой “c” и добавлением “.h” в конец, например, вместо `<cmath>` используем `<math.h>`

Составные литералы можно использовать в операторах. Эта возможность поддерживается GNU си++. Пример.

```
#include <cstdio>
using namespace std;
struct X {
   int a,b;
};
```

```
main(){
   int *p, i;
   struct X b;
   p = (int[]){1,2,3,4,5,6,7}; //создается анонимный массив
   for (i = 0; i < 7; i++)
      printf("%d",p[i]);
   b = (struct X){8,9};
   printf("%d%d\n", b.a, b.b);
} //123456789
```

Листинг compound-literals.cpp

Можно инициализировать структуры и массивы по именам полей и индексам соответственно. Пример.

```
struct X {int a; const char *s;} x = { .s = "abc", .a = 7};
int z[10] = {[7] = 11};
```

Препроцессор

Может выполнять следующую работу:

1) склеивать строки;
2) делать замены в тексте, в частности, используя макрорасширения;
3) удалять определённые части текста (условная компиляция);
4) включать текст из других файлов;
5) передавать команды компилятору.

Склейка строк — это удаление последовательности знаков “обратная косая черта” (backslash — \) и переход на новую строку (`'\n'`) в исходном тексте программы.

```
char BigString[] = "0123456789\
ABCDEF"; //BigString инициализируется значением
   //"0123456789ABCDEF"
```

Делать замены в исходном тексте можно, используя директиву `#define`, которую можно использовать тремя способами, для определения:

1) констант (макросов-значений);
2) имён (пустых макросов);
3) макросов с аргументами.

Общий синтаксис определения макроса весьма прост — после слова `#define` следует имя-идентификатор макроса, после которого следует опциональная последовательность лексем. Определение заканчивается концом строки, но для макросов, не помещающихся в одну строку, можно использовать склейку строк. В последовательности лексем все символы, включая пробелы, являются значимыми. Если препроцессор встречает в тексте программы *идентификатор* определённого ранее макроса, то он заменяет его на соответствующую ему последовательность лексем. В случае макросов-значений и пустых макросов эта замена проходит чисто механически, а в случае макросов с аргументами — происходит ещё подстановка аргументов. Для имён пустых макросов и макросов-значений принято использовать заглавные буквы.

Пример — должен печатать 16,49,36,400,EMPTY_STRING,5397.

```
#include <iostream>
using namespace std;
#define EMPTY_STRING
#define ARRAY_SIZE 128
#define DECLARATION int k = 0;
#define sqr(x) ((x)*(x))
#define pos(x,y) ((x) + 80*(y))
main() {
   DECLARATION
   unsigned array[ARRAY_SIZE];
   char string[] = "EMPTY_STRING";
      //заменяются только идентификаторы!
   long sum = 0;
   for (; k < ARRAY_SIZE; sum += (array[k++] = sqr(k)));
      //array[n]=n*n
   cout << sqr(4) << ',' << sqr(5 + 2) << ',' << 4*sqr(3)
```

```
      << ',' << pos(0, 5) << ',' << string << ','
      << sum/ARRAY_SIZE << endl;
         //среднее квадратов чисел от 0 до 127
}
```

Листинг macro1.cpp

Внешние скобки в определении макросов `sqr` и `pos` необходимы для того, чтобы эти макросы можно было использовать в выражениях, а скобки, окружающие формальные аргументы, необходимы для того, чтобы фактические аргументы могли быть выражениями.

Пример — макросы без "лишних" скобок.

```
#define m1(x) (x*x)
#define m2(x) (x)+(x)
main() {
   int i;
   i = m1(2+4); //получится i=(2+4*2+4), а не i=((2+4)*(2+4))
   i = m2(5)*2; //получится i=5+5*2, а не i=(5+5)*2
}
```

Листинг macro2.cpp

При использовании макросов нужно соблюдать некоторую осторожность — с их помощью текст программы можно сделать совершенно не похожим на текст на си++.

С макросами можно использовать две специальные операции: 1) `#` — подстановка строки; 2) `##` — склейка лексем. Если аргумент макроса начинается с `#`, то это означает, что при подстановке он всегда заменяется на символьную строку. Если две лексемы в макросе соединены знаком `##` и, возможно, пробелами, то они соединяются препроцессором в одну лексему.

```
#include <iostream>
using namespace std;
#define ShowString(s) cout << #s << endl
#define ShowVar(v) cout << i##v << endl
main() {
   int i1 = 12, i2 = 16, i3 = 22;
   ShowString(ok.); //ok.
   ShowString("ok."); //"ok."
   ShowVar(1); //12, т.е. значение i1
   ShowVar(3); //22
}
```

Листинг macro3.cpp

Макросы можно переопределять (это может вызвать предупреждающее сообщение) или удалять директивой `#undef`, после которой нужно указывать имя удаляемого макроса.

```
#include <iostream>
using namespace std;
#define m1(x) ((x)*(x))
main() {
   cout << m1(4) << endl; //16
   #define m1(x) ((x)*(x)*(x))
   cout << m1(4) << endl; //64
   #undef m1
          //далее в программе использовать m1 нельзя
}
```

Листинг macro4.cpp

Макросы не могут контролировать соответствие типов аргументов, что не позволяет обнаруживать ряд ошибок на этапе компиляции, поэтому рекомендуется вместо макросов использовать inline-функции, шаблоны и константы.

Директивы препроцессору `#ifdef`, `#ifndef`, `#if`, `#else`, `#elif`, `#endif` позволяют организовать условную компиляцию. Начинается условная компиляция со строки `#if условие`, после которой следует текст, а заканчивается строкой `#endif`. Текст может делиться на фрагменты строками `#elif`, а после всех `#elif` можно ещё использовать строку `#else`.

Если условие после `#if` истинно (не 0), то все кроме текста за ним удаляется из конструкции, иначе если условие после `#elif` истинно, то удаляется все кроме текста за ним и т. д. Если же все условия ложны,

то удаляется все кроме текста за `#else`. Слово elif — это сокращение от else if. Директивы `#elif` и `#else` могут не использоваться. При отсутствии директивы `#else` и ложности всех условий в текст программы ничего не включается. В качестве условия может быть взято любое константное выражение, не содержащее операций `sizeof`, преобразования типа, а также enum-констант. Кроме того, только с `#if` и `#elif` можно использовать операцию `defined` с одним аргументом-именем: она возвращает 1, если это имя было ранее определено директивой `#define`, и 0 в противном случае. Вместо `#if условие` можно использовать директивы `#ifdef идентификатор` и `#ifndef идентификатор`. Две последние директивы полностью эквиваленты следующим конструкциям: `#if defined(идентификатор)` и `#if !defined(идентификатор)`. Круглые скобки с `defined` использовать необязательно. Конструкции условной компиляции могут быть вложенными, но должны содержаться в одном файле.

Средства условной компиляции позволяют использовать один и тот же текст программы для генерации кода:

1) для конечного продукта. Этот код оптимизируется по скорости исполнения и почти не содержит средств проверки правильности выполнения операций, например, обычно не проверяется соответствие значения индекса массива допустимому диапазону этого индекса;

2) для отладки — медленный код, который содержит в себе все возможные проверки правильности исполнения операций;

3) соответствующего различной аппаратуре, ОС или компилятору.

```
#include <iostream>
using namespace std;
#define UNIX
//#define MICROSOFT_WINDOWS
#ifdef UNIX
   #define DIR_SIGN '/'
#else
   #define DIR_SIGN '\\'
#endif
main() {
   cout << "В этой ОС знаком каталога является "
      << DIR_SIGN << endl;
}
```

Листинг macro5.cpp

Как уже было рассмотрено ранее, вставка текста из других файлов производится директивой `#include`.

Пустая директива препроцессору состоит из одного символа `#`. Она ничего не делает и используется только для визуального выделения пустых строк.

Директива `#error` может иметь аргумент-строку текста. При встрече такой директивы компиляция исходного файла прекращается и на печать (в канал ошибок) выводится аргумент.

```
#define ERROR
#if defined ERROR && !defined OK
   #error The file cannot be compiled!
#endif
```

Листинг macro6.cpp

Препроцессор может использовать некоторые предопределённые идентификаторы, например:

`__DATE__` — дата компиляции;

`__TIME__` — текущее время;

`__FILE__` — имя обрабатываемого файла;

`__LINE__` — номер строки компилируемого файла.

Директива `#line` имеет один или два аргумента: первый номер строки, второй — имя файла в двойных кавычках. Выполнение этой директивы заключается в том, что идентификатору `__LINE__` присваивается указанный номер строки, а идентификатору `__FILE__` — имя указанного файла. Если второй аргумент отсутствует, то `__FILE__` не меняется. Эта директива может быть полезна для опытных пользователей при отладке программ, а также при генерации текста на си/си++ из исходников на другом языке.

```
#include <iostream>
using namespace std;
main() {
   cout << "Date = " << __DATE__ << endl;
   cout << "Time = " << __TIME__ << endl;
   cout << "File:Line = " << __FILE__ << ':' << __LINE__
      << endl;
   #line 22 "NOFILE.CPP"
```

```
   cout << "----" << endl;
   cout << "File:Line = " << __FILE__ << ':' << __LINE__
      << endl;
}
```

Листинг macro7.cpp

Директива `#pragma` используется для передачи команд компилятору, например, в ней можно указать как оптимизировать код: по размеру или скорости исполнения. Набор команд для `#pragma` меняется от компилятора к компилятору. Если команда в этой директиве не распознаётся, то она игнорируется.

В системах программирования на си++ препроцессор объединяется с компилятором в одном файле, но, как правило, в них есть и отдельный препроцессор, который позволяет программисту увидеть результат его работы в отдельном файле. В системе GNU вызов только препроцессора обеспечивается опцией `-E` транслятора `g++`. Например, если файл из предыдущего примера назвать `example.cpp`, то вызов `g++ -E example.cpp >example.ii` создаст файл `example.ii`, содержащий текст, обработанный препроцессором.

Функции с произвольным числом параметров

Могут иметь любое количество параметров, от нуля до неограниченного. Если у функции нет параметров, то её список формальных параметров надо описывать как `()` или `(void)`. Если у функции неограниченное количество параметров, то список её формальных параметров нужно заканчивать лексемой `...` (эллипсис), перед которой можно не ставить запятую.

Для работы с функциями, допускающими неограниченное число параметров, в заголовке `<cstdarg>` описаны следующие имена:

- `va_list` — стандартный тип для работы со списком параметров неизвестной длины;
- `va_start` — это макрос с двумя аргументами: 1-й — переменная типа `va_list`; 2-й — имя параметра, предшествующего первому неопределённому. Он устанавливает первый параметр-указатель на начало списка необязательных параметров;
- `va_arg` — это макрос с двумя аргументами: 1-й — переменная типа `va_list`; 2-й — тип. Он возвращает значение текущего, указываемого 1-м параметром, необязательного параметра и переводит этот свой параметр-указатель на следующий необязательный параметр. Тип должен соответствовать типу текущего необязательного параметра.
- `va_end` — это макрос с одним аргументом-переменной типа `va_list`. Он заканчивает работу со списком параметров неизвестной длины — его можно не использовать.

```
#include <iostream>
#include <cstdarg>
using namespace std;
main() {
   int average(unsigned, ...);
   int n = average(5, 1, 2, 3, 4, 5);
   cout << n << ',' << average(4, -4, 0, 16, 8) << endl;
} //3,5
int average(unsigned list_size, ...) { //list_size -
   //количество величин, среднее которых надо подсчитать
   va_list arg_ptr;
   unsigned sum = 0, n = list_size;
   va_start(arg_ptr, list_size);
   while (n-- > 0)
      sum += va_arg(arg_ptr, int);
   va_end(arg_ptr);
   return sum/list_size;
}
```

Листинг stdarg.cpp

В стандартной библиотеке есть ряд функций с неограниченным числом параметров, например, `printf` и `scanf`.

Работа со строками в стиле си

Строка в си — это массив символов, завершающийся нулем, — не символом `'0'`, а числом 0, т. е `'\0'`. Константные строки-литералы задаются последовательностью символов в кавычках. К каждой последовательности символов в кавычках транслятор автоматически добавляет нулевой символ. Знак обратная косая черта играет

в таких последовательностях такую же роль как и при определении символьных констант: он присоединяется к следующему или следующим символам, вместе с которым или которыми он образует одиночный символ.

Пример — печатает 5 строк: ABC, 3\4, 86'40, XYZ и "TEST".

```
#include <iostream>
using namespace std;
main() {
   char s1[] = "ABC\n3\\4\012", s2[] = "86\'40\x0A",
      s3[] = {'X', 'Y', 'Z', '\n', '\0'}, s4[] = "\"TEST\"";
   cout << s1 << s2 << s3 << s4 << endl;
}
```

Листинг c-string1.cpp

Строки — это массивы, поэтому к ним можно применять только операцию выделения элемента, индексацию. Нельзя, например, в предыдущем примере присвоить строку `s2` строке `s1`. Строки также нельзя сравнивать операциями `==`, `!=`, `<=`, `>=`, `<`, `>`, так как будут сравнивать не строки, а указатели на них.

Стандартным заголовком `<cstring>` объявляются ряд функций, которые предназначены для работы со строками:

- `char* strcat(char *result, const char *s)` — string catenation — добавляет строку `s` к строке `result`, возвращает указатель на строку-результат;
- `char* strncat(char *result, const char *s, unsigned n)` — добавляет первые `n` символов строки `s` к `result`, возвращает указатель на строку `result`;
- `int strcmp(const char *s1, const char *s2)` — string comparison — сравнивает посимвольно строки `s1` и `s2`, возвращает отрицательное значение, если `s1 < s2`, положительное, если `s1 > s2`, ноль, если `s1` равна `s2`, например, `strcmp("AB","ABC")` меньше нуля;
- `int strncmp(const char *s1, const char *s2, unsigned n)` — сравнивает начальные подстроки длиной не более `n` символов строк `s1` и `s2`, возвращает значение такое же как и `strcmp`, вызванная к определённым подстрокам;
- `int strcasecmp(const char *s1, const char *s2)`
- `int strncasecmp(const char *s1, const char *s2, unsigned n` — аналогичны `strcmp` и `strncmp`, но сравнивают, игнорируя регистр;
- `char* strcpy(char *result, const char *s)` — string copy — копирует строку `s` в `result`, возвращает указатель на `result`. Эта функция реализует присваивание для строк разного типа, например, `char[10]` и `char[20]`;
- `char* strncpy(char *result, const char *s, unsigned n)` — копирует начальную подстроку длины `n` строки `s` в `result`, возвращает указатель на `result`. Если среди `n` первых байт `s` нет нулевого, то конечный 0 к `result` не добавляется;
- `int strlen(const char *s)` — string length — возвращает длину строки `s` без завершающего 0;
- `char* strchr(const char *s, int c)` — string character — ищет символ `c` в строке `s`, возвращает указатель на первое вхождение `c` в `s` или 0, если такого вхождения нет;
- `char* strstr(const char *s1, const char *s2)` — string string — поиск строки `s2` в строке `s1`, возвращает указатель на начало первой найденной подстроки в `s1` или 0, если такой подстроки не найдено.
- `char* strrchr(const char *s, int c)` — string reverse character — ищет символ `c` в строке `s`, возвращает указатель на последнее вхождение `c` в `s` или 0, если такого вхождения нет;
- `char* strpbrk(const char *s, const char *chrset)` — string position break — поиск символа из строки `chrset` в строке `s`, возвращает указатель на первый найденный символ или 0, если не найден ни один символ из `chrset`;
- `int strspn(const char *s, const char *chrset)` — string span — возвращает количество начальных символов `s`, которые все входят в `chrset`, например, `strspn("ABCDE","DBXAZ")` равно 2;
- `int strcspn(const char *s, const char *chrset)` — string complement span — возвращает количество начальных символов `s`, которые все не входят в `chrset`, например, `strcspn("ABCDE","DBXAZ")` равно 0, а `strcspn("123BCE","DBXAZ")=3`;
- `char* strtok(char *s, const char *chrset)` — string token — выделяет из строки `s` слова, разделённые символами из `chrset`. При первом вызове эта функция возвращает указатель на первое выделенное в `s` слово. Последующие вызовы `strtok` с первым параметром, равным 0, будут возвращать указатели на следующие слова или 0, после того как в `s` не останется ни одного не выделенного слова. Эта функция модифицирует свой первый аргумент.

```
#include <iostream>
#include <cstring>
using namespace std;
main() {
   char s1[] = "ABCDEFG", *s2, s3[20];
```

```
   strcpy(s3, s1); //присваивание s3 значения s1
   s2 = s1;        //s2 становится синонимом s1
   cout << strlen(s1) << ' ' << s1 << ' ' << s2 << ' ' << s3
      << endl;
   s1[1] = '-';
   s2[2] = '>';
   s3[3] = '=';
   cout << strlen(s1) << ' ' << s1 << ' ' << s2 << ' ' << s3
      << endl;
   s3[3] = 0;
   cout << strlen(s3) << ' ' << s3 << endl;
   //strcpy(s1 + 7, s3) //нельзя - в s1 не хватит места,
       //но компилятор об ошибке не сообщит
   strcpy(s3 + 3, s2);
   cout << strlen(s3) << ' ' << s3 << ' ' << strchr(s3, 'E')
      << ' ' << strstr(s3, "DE") << endl;
} /* 7 ABCDEFG ABCDEFG ABCDEFG
     7 A->DEFG A->DEFG ABC=EFG
     3 ABC
     10 ABCA->DEFG EFG DEFG */
```

Листинг c-string2.cpp

Стандартные объектные строки реализуются на основе объектно-ориентированного подхода и шаблонов. Средства для работы с ними носят более высокоуровневый и в большинстве случаев удобный характер.

Использование параметров вызова программы

Функцию `main` можно вызывать с двумя параметрами: 1-й, который принято называть `argc`, — это число параметров, считая имя самой программы; 2-ой, который принято называть `argv`, — это указатель на массив строк, в которые копируются параметры. С индексом 0 в `argv` храниться имя программы, с 1 — 1-й параметр и т.д.

```
#include <cstdio>
using namespace std;
main(int argc, char *argv[]) {
   printf("Программа %s вызвана с %d параметрами: ", argv[0],
                                                     argc - 1);
   for (int i = 1; i < argc; i++)
      printf("%s ", argv[i]);
   printf("\n");
}
```

Листинг main-params.cpp

Если полученная после компиляции программа называется `main-params`, то вызов

```
main-params A B CD 555
```

приведет к печати сообщения: “Программа main-params вызвана с 4 параметрами: A B CD 555”.

Стандартные функции для преобразование типов

Преобразование может быть явным, неявным и функцией. Последнее позволяет преобразовывать любые типы друг в друга стандартными или определяемыми пользователем функциями.

```
#include <iostream>
using namespace std;
long str2int(const char *s) { //преобразование строки в число
   long n = 0, i = 0;
   while (s[i])
     if (s[i] <= '9' && s[i] >= '0')
        n = n*10 + s[i++] - '0';
     else
        throw 0;
   return n;
}
main() {
```

```
   char s1[] = "123", s2[] = "4";
   int i = str2int(s1)*str2int(s2);
   cout << i << endl;
} //492
```

Листинг type-cnv1.cpp

Следующие, описанные в заголовке `<cstdlib>`, стандартные функции и макросы предназначены для преобразования типов:

- `long strtol(const char *s, char **p, int base)` — преобразует строку `s` в длинное целое. Через `p` возвращается указатель на символ в `s`, на котором преобразование было прекращено. Параметр `base` — это основание системы счисления (от 2 до 36);
- `long atol(const char *s)` — преобразует строку `s` в длинное целое. Упрощённый вариант `strtol`;
- `int atoi(const char *s)` — преобразует строку `s` в целое (`int`);
- `double strtod(const char *s, char **p)` — преобразует строку `s` в вещественное. Параметр `p` имеет тот же смысл, что и в `strtol`;
- `double atof(const char *s)` — преобразует строку `s` в вещественное. Упрощённый вариант `strtod`.

```
#include <iostream>
#include <cstdlib>
using namespace std;
main() {
   char s1[] = "123z", s2[] = "4", *p;
   long i = strtol(s1, &p, 10)*atol(s2);
   cout << i << *p << endl;
} //492z
```

Листинг type-cnv2.cpp

Работа с файлами в стиле си

В стандартном заголовке `<cstdio>` объявлены все необходимые для работы с файлами средства: определение типа данных `FILE` (структуры), объявления функций, макросов, констант и переменных для работы с файлами.

Рассмотрим некоторые файловые функции:

- `FILE* fopen(const char* filename, const char* mode)` — открывает файл с именем `filename` со способом доступа, определяемым строкой `mode`, которая может иметь следующие значения: `r` — только для чтения, `r+` и `w+` — для чтения и записи, `w` — для записи, `a` — записи в конец файла, `a+` — для чтения и записи в конец файла. Для использования `r` необходимо существование открываемого файла, использование `w` уничтожает существующий файл с именем `filename`, при использовании `a` открывается существующий файл или, если файла с именем `filename` нет, создаётся новый. При неудачном открытии файла `fopen` возвращает 0, при удачном — указатель на открытый файл;
- `int fclose(FILE* fp)` — закрывает файл `fp`, возвращает 0, если файл успешно закрыт, или `EOF` (стандартная константа) в противном случае;
- `int fseek(FILE* fp, long offset, int origin)` — изменяет текущую позицию в файле `fp` на смещение `offset` относительно начала отсчёта `origin`, которое может принимать три значения: `SEEK_SET` = 0 — начало файла, `SEEK_CUR` = 1 — текущая позиция, `SEEK_END` = 2 — конец файла. Функция возвращает 0 при успешном выполнении и ненулевое значение при неуспехе;
- `long ftell(FILE* fp)` — возвращает текущую позицию файла `fp`;
- `int feof(FILE* fp)` — возвращает 0, если конец файла `fp` не достигнут, и не 0, если достигнут;
- `int fputc(int c, FILE* fp)` — запись символа `c` в текущую позицию файла `fp` и сдвиг текущей позиции на один байт. При успехе возвращает `c`, при неуспехе или при достижении конца файла — `EOF`;
- `int fgetc(FILE* fp)` — возвращает символ с текущей позиции файла и сдвигает текущую позицию на следующий символ-байт. При неуспехе возвращает `EOF`.
- `int fputs(const char *s, FILE *fp)` — записывает строку `s` в файл `fp` без конечного `'\0'`. Возвращает неотрицательное число при успехе и `EOF` при неуспехе.
- `char *fgets(char *s, int size, FILE *fp)` — читает `size` или менее байт из `fp` в строку `s`. Данные читаются до конца строки. Символ `'\0'` добавляется к `s` автоматически. Возвращает `s` при успехе и 0 при неуспехе.
- `int fread(void *p, int s, int q, FILE *fp)` — пытается считать в буфер по указателю `p` `q` элементов размера `s` байт из файла `fp`. Результат — число считанных элементов.
- `int fwrite(void *p, int s, int q, FILE *fp)` — пытается записать из буфера по указателю `p` `q` элементов размера `s` байт в файл `fp`. Результат — число записанных элементов.

Пример. Вывод на экран дисплея содержимого файла passwd.

```
#include <cstdio>
using namespace std;
main() {
  FILE *file;
  char filename[] = "/etc/passwd";
  int c;
  if (file = fopen(filename, "r")) {
    do {
      c = fgetc(file);
      printf("%c", c);
    } while (c != EOF); //здесь нужен тип int для c
    printf("\n");
  }
}
```

Листинг c-files.cpp

Работа с отдельными битами

Биты принято нумеровать с 0 от младшего разряда к старшим. Для работы с отдельными битами можно использовать поразрядные логические операции: `&` позволяет проверять и сбрасывать в 0, `|` — устанавливать в 1, `^` — изменять, например,

`z & 3` — проверит оба ли бита `z` 0 и 1 нулевые и установит биты с 2 до последнего в 0,

`z | 6` — установит в `z` биты 1 и 2 в 1,

`z ^ 8` — изменит в `z` бит 3.

Массивы бит

В стандартной библиотеке в заголовке `<bitset>` объявлены средства для работы с массивом бит, задаваемым типом `bitset<N>`, где `N` — это размер массива. Для инициализации битового массива можно использовать целое число, строку или часть строки.

Для массивов бит определены сравнения (`==`, `!==`), все поразрядные операции (`~`, `&`, `|`, `^`, `<<`, `>>`), присваивание и комбинации присваивания с поразрядными операциями. Сдвиги нециклические. Можно испорльзовать `([])` для прямого доступа к битам по их индексу. Другие операции:

с результатом `bitset&`: `set()` — установить все биты в 1, `set(unsigned)` — установить заданный бит в 1, `reset(unsigned)` и `reset()` — в 0, `flip()` и `flip(unsigned)` — инвертирование;

`unsigned count()` — счётчик количества 1-ц;

`unsigned size()` — размер;

`bool test(unsigned)` — истина, если в заданной позиции 1;

`bool any()` — истина, если `count()` > 0;

`bool none()` — истина, если все 0;

`basic_string<char> to_string()` — конвертировать в строку;

`unsigned to_ulong()` — конвертировать в число.

Пример.

```
#include<iostream>
#include<string>
#include<bitset>
using namespace std;
main() {
   bitset<10> b0; //заполняется 0-ми
   bitset<10> b1 = 0xa;
   bitset<10> b2("10101"); //в строках - только 0 и 1
   bitset<10> b3("110011001100",1,5); //с позиции 1 5 знаков
   bitset<160> b4;
   cout << b0 << ' ' << b1 << ' ' << b2 << ' ' << b3 << endl;
                //0000000000 0000001010 0000010101 0000010011
   cout << (~b1&b2|b3) << endl; //0000010111
   cout << b0.set() << ' ' << b1.reset(1) << ' ' << b2.flip()
      << ' ' << b3.flip(0) << endl;
                //1111111111 0000001000 1111101010 0000010010
   cout << b3.count() << ' ' << b4.any() << ' ' << b1.none()
```

```
      << ' ' << b2.test(3) << ' ' << sizeof(b4) << endl;
                                                  //2 0 0 1 20
   cout << b3.to_ulong() << " " + b3.to_string() << endl;
                                                //18 0000010010
}
```

Листинг bitset.cpp

Поля бит

В структурах можно задавать поля целых типов заданного в битах размера. Это позволяет избегать явных поразрядных операций для доступа к битам. Пример.

```
#include<iostream>
using namespace std;
struct BIOSHARDWARE {
   unsigned floppy_boot: 1;
   unsigned s80x87: 1;
   unsigned ramsize: 2;
   unsigned video_mode: 2;
   unsigned floppies: 2;
   unsigned dma: 1;
   unsigned ports: 3; //RS-232
   unsigned game: 1; //joystick
   unsigned: 1; //пустое поле для выравнивания
   unsigned printers: 2;
} *p;
main() {
  int d410h = 0x9867;
  p = (BIOSHARDWARE*) &d410h;
  cout << sizeof(BIOSHARDWARE) << endl; //4 - int
  if (p->s80x87)
    cout << "math co-processor presents\n";
  else
    cout << "no math co-processor\n";
  p->floppies = 3;
  cout << p->floppies << " diskdrive(s)\n";
  cout << p->printers << " parallel printer(s)\n";
}
```

Листинг bit-fields.cpp

С битовыми полями можно работать как со значениями соответствующего целого типа.

Управление размещением объектов в памяти

Можно использовать `new` для размещения данных в указанном месте — такое использование предполагает подключения заголовка `<new>`. Не следует использовать `delete` с данными, размещёнными таким образом, так как для них память не выделялась, а использовалась имеющаяся. С такими данными естественно использование явного вызова деструктора.

Программа из следующего примера должна два раза напечатать 5550777.

```
#include <iostream>
using namespace std;
struct X {
   int l;
   X(): l(0) {}
   X(int n): l(n) {}
   ~X() {}
};
char mem[1024];
main () {
   X *p1 = new (mem) X(555);
   X *p2 = new (&mem[sizeof(X)]) X;
```

```
   X *p3 = new (&mem[2*sizeof(X)]) X(777);
   cout << p1->l << p2->l << p3->l << endl;
   cout << (int&) mem[0] << (int&) mem[sizeof(int)]
      << *(((int*)mem) + 2) << endl;
   p2->~X();
}
```

Листинг alloc1.cpp

Операции `new` и `delete` можно переопределять. Результат `new` должен быть `void*`, её первый аргумент всегда неявный, типа `size_t`, т. е. беззнакового целого, возвращаемого операцией `sizeof`. Остальные аргументы могут быть любыми. У `delete` аргумент должен быть типа `void*`. Рассматриваемые операции можно определять для классов или переопределять на глобальном уровне. Если для класса нужна глобальная операция, то используется `::`. Любой вызов `delete` сначала пытается вызывать деструктор.

Рассмотрим пример простой системы управления памятью, в которой можно освобождать только последний занятый элемент — первое удаление объекта по указателю `px1` будет проигнорировано.

```
#include <iostream>
using namespace std;
struct Memory {
  static const int max = 65536;
  char mem[max];
  int first_free, total_free;
  Memory(): first_free(0), total_free(max) {}
} m;
struct X {
   int d;
   X(): d(0) {}
   X(int n): d(n) {}
   void* operator new(size_t, Memory&);
   void operator delete(void *);
};
void* X::operator new(size_t t, Memory &m) {
   X* p;
   if (m.total_free < t) throw "no memory";
   p = ::new (&m.mem[m.first_free]) X;
   m.first_free += t;
   m.total_free -= t;
   return p;
}
void X::operator delete(void *p) {
   if (p == &m.mem[m.first_free - sizeof(X)]) {
      m.first_free -= sizeof(X);
      m.total_free += sizeof(X);
      cout << "delete\n";
   }
   else
      cout << "ignore\n";
}
main () {
   cout << m.total_free << endl;  //65536
   X *px1 = new (m) X;
   cout << m.total_free << ' ' << px1->d << endl;  //65532 0
   X *px2 = new (m) X(5);
   cout << m.total_free << ' ' << px2->d << endl;  //65528 5
   delete px1;    //нет связи с m
   delete px2;
   delete px1;
   cout << m.total_free << endl;  //65536
}
```

Листинг alloc2.cpp

Здесь new определён как член класса, однако его можно было бы определить и вне класса с аналогичным заголовком. Если нужно удалить объект, размещённый в заданном месте, то нужно для этого определять специальную функцию — нет средств для доопределения delete для использования во всех подобных случаях.

```
#include <iostream>
#include <new>
using namespace std;
struct Arena {
   char b[1024];
   void* alloc(size_t) {return b;}
   void free() {}
};
void* operator new(size_t sz, Arena* a) {
   return a->alloc(sz);
}
struct Data {
   int d;
   Data(int n): d(n) {}
};
void destroy(Data *p, Arena *a) {
   p->~Data();
   a->free();
}
main () {
  Arena *a = new Arena;
  int *p = new (a) int (7);
  cout << *p << endl;  //7
  a->free();  //стандартный тип, нет деструктора
  Data *data = new (a) Data (5);
  cout << data->d << *p << endl;  //5
  destroy(data, a);
}
```

Листинг alloc3.cpp

С помощью new можно выделять заданное число байт памяти для объекта без вызова конструктора.

Рассмотрим пример по размещению объектов, суммирующий способы вызова new. Конструктор должен вызываться 4 раза, а деструктор 3. Должно быть напечатано 1334.

```
#include <iostream>
#include <new>
using namespace std;
struct X {
  int d;
  X(int n): d(n) {cout << "constructed [" << this << "] = "
     << n << endl;}
  ~X() {cout << "destructed [" << this << "] = " << d << endl;}
};
main () {
  X *p1 = new X(1);  //operator new(sizeof(X))
  X *p2 = new (nothrow) X(2); //operator new(sizeof(X), std::nothrow)
  X *p3 = new (p2) X(3);  //operator new(sizeof(X), p2)
  X *p4 = (X*) operator new (sizeof(X));
                  //operator new(sizeof(X)), не вызывает конструктора
  new (p4) X(4); //вызов конструктора
  cout << p1->d << p2->d << p3->d << p4->d << endl;
  delete p1;
  delete p2;
  delete p4;
```

```
}
```

Листинг alloc4.cpp

Операцию `delete` также можно использовать в низкоуровневом варианте, без вызова деструктора. В таком варианте у `delete` один аргумент-указатель в скобках и необходимо использовать слово `operator`.

Стандартный распределитель памяти

В заголовке `<memory>` объявлен шаблон класса `allocator`, который используется по умолчанию операцией `new`. Он предоставляет, в частности, следующие операции:

- `T* allocate(size_t n)` — выделить память для n объектов типа `T`. Если $n \neq 1$, то выделяется память для массива. Конструктор не вызывается;
- `void deallocate(T* p, size_t n)` — освобождает память от n объектов типа `T`, начиная с `p`. Память не возвращается ОС, а остаётся в пользовании для всех данных типа `T`;
- `construct(T* p, const T& val)` — конструирует объект по указателю `p` со значением `val`;
- `destroy(T* p)` — уничтожает объект `*p`.

Кроме того, вводится структура `rebind`, позволяющая распределителю динамически подстраиваться под требуемый тип. Распределители можно сравнивать на равенство.

Рассмотрим пример использования этих средств. Должно быть напечатано 7 и hello world.

```
#include <memory>
#include <iostream>
#include <string>
using namespace std;
int main() {
   allocator<int> a1;
   int* a = a1.allocate(10); //массив для 10 целых
   a[4] = 7;
   cout << a[4] << '\n';
   a1.deallocate(a, 10);
   allocator<string> a2;  //аналогично следующей строке
   //allocator<int>::rebind<string>::other a2;
   string* s = a2.allocate(2); //массив из 2 строк
   a2.construct(s, "hello");
   a2.construct(s + 1, "world");
   cout << s[0] << ' ' << s[1] << endl;
   a2.destroy(s);
   a2.destroy(s + 1);
   a2.deallocate(s, 2);
}
```

Листинг alloc5.cpp

Можно написать другой распределитель памяти и использовать его вместо стандартного. Недостаток стандартного распределителя в том, что он выделяет память для каждого объекта отдельно через вызовы ОС. Это может приводить к излишнему расходу памяти (каждое выделение может требовать отметки в структурах данных ОС) и снижению скорости работы (из-за вызовов средств ОС). Можно оптимизировать работу распределителя, выделяя заранее память для будущих размещений.

Рассмотрим распределитель, который будет выделять память кусками, достаточными для хранения нескольких элементов заданного типа, и добавим к этому распределителю интерфейс, совместимый со стандарной библиотекой. При создании распределителя будет напечатан размер его элемента данных, а затем строки 8-24-1000 и 7.

```
#include <iostream>
#include <memory>
#include <map>
#include <vector>
using namespace std;
class Pool {
   struct Link {Link *next;};
   struct Chunk {
      static const unsigned size = 8192 - sizeof(Chunk*);
         //выравнивание по границе страницы, 256 байт
      Chunk *next;
```

```
      char mem[size];
   } *chunks;
   Link *head;  //указатель на 1-й свободный элемент памяти
   void grow();    //увеличение пула
public:
   const unsigned int elsize;
   Pool(unsigned n);  //n - запрашиваемый размер элементов
   ~Pool();
   void* alloc();  //выделить память для элемента
   void free(void*);
      //помещение элемента обратно в пул свободной памяти
};
void* Pool::alloc() {
   if (head == 0) grow();
   Link *p = head;
   head = p->next;
   return p;
}
void Pool::free(void *b) {
   Link *p = static_cast<Link*>(b);
   p->next = head;
   head = p;
}
Pool::Pool(unsigned sz):
   elsize(sz < sizeof(Link*) ? sizeof(Link*) : sz) {
     cout << "element size = " << elsize << " bytes\n";
     head = 0;
     chunks = 0;
}
Pool::~Pool() {
   Chunk *p = chunks;
   while (p) {
      Chunk *q = p;
      p = p->next;
      delete q;
   }
}
void Pool::grow() {
   Chunk *p = new Chunk;
   p->next = chunks;
   chunks = p;
   const unsigned noe = Chunk::size/elsize;
      //число элементов в куске, number of elements
   char *start = p->mem, *last = start + (noe - 1)*elsize;
   for (char *p = start; p < last; p += elsize)
      reinterpret_cast<Link*>(p)->next
         = reinterpret_cast<Link*>(p + elsize);
   reinterpret_cast<Link*>(last)->next = 0;
   head = reinterpret_cast<Link*>(start);
}
template<class T> class Pool_alloc : public allocator<T> {
   static Pool pool;
      //static для взаимодействия с контейнерами стандартной библиотеки
public:
   template<class U> struct rebind {
      typedef Pool_alloc<U> other;
```

```
   };
   template<class U> Pool_alloc(const Pool_alloc<U>&) {}
   Pool_alloc() {}
   T* allocate(size_t, void*);
   void deallocate(T*, size_t);
};
template<class T> Pool Pool_alloc<T>::pool(sizeof(T));
template<class T> T* Pool_alloc<T>::allocate(size_t n, void* = 0) {
   T* p;
   if (n == 1)
      p = static_cast<T*>(pool.alloc());
   else
      p = static_cast<T*>(allocator<T>::allocate(n));    //STL level
      //p = static_cast<T*>(operator new (sizeof(T)*n));  //OS level
   return p;
}
template<class T> void Pool_alloc<T>::deallocate(T* p, size_t n) {
   if (n == 1)
      pool.free(p);
   else
      allocator<T>::deallocate(p, n);  //STL level
      //operator delete(p);  //OS level
}
main() {
   map<int, int, less<int>, Pool_alloc<pair<int, int> > > m;
   for (int i(0); i < 1000; ++i)
      m[i*i] = 2*i;
   m.erase(36);
   m.insert(pair<int,int>(7, 8));  //m[7] = 8;
   cout << sizeof(pair<int, int>) << '-' << m[144] << '-'
      << m.size() << endl;
   vector<int, Pool_alloc<int> > v(1000);
   v[4] = 7;
   cout << v[4] << endl;
}
```

Листинг pool.cpp

Поле `pool` статическое, что позволяет использовать данный распределитель всеми объектами с однотипными динамическими элементами.

Этот распределитель использует стандартные средства для работы с массивами, а значит и с векторами. Для их поддержки распределителем нужно будет, например, ввести методы `alloc(unsigned)` и `free(void*, unsigned)`. Особая проблема возникнет с массивами, размер элемента которых меньше `sizeof(Link*)`. Однако, распределитель спроектирован именно для оптимизации работы со скалярными элементами и поддержка им массивов не будет лучше, чем у стандартных средств.

Рассматриваемый распределитель будет выделять пул данных для каждого уникального типа, но было бы лучше, если бы он выделял память, ориентируясь только на размер типа данных.

Для прямой работы с памятью можно использовать следующие шаблоны функций, которые действуют, не вызывая конструктор:

- `template<class In, class For> uninitialized_copy(In first, In last, For res)` — копирует память с $first$ до $last$ в res;
- `template<class For, class T> uninitialized_fill(For first, For last, const T& val)` — заполняет память с $first$ до $last$ значением val;
- `template<class For, class Size, class T> uninitialized_fill_n(For first, Size n, const T& v)` — заполняет память длиной n, начиная с $first$, значением v.

Следующий пример должен напечатать hurry huffy hussy.

```
#include<cstdio>
#include<cstring>
#include<memory>
using namespace std;
main() {
   char s[] = "hurry", *p = (char*)operator new(strlen(s) + 1);
   uninitialized_copy(s, s + strlen(s) + 1, p);
      //или memcpy(p, s, strlen(s) + 1)
   puts(p);  //печатает строку-аргумент с добавлением '\n'
   uninitialized_fill(p + 2, p + 4, 'f');
   puts(p);
   uninitialized_fill_n(p + 2, 2, 's'); //или memset(p + 4, 's', 2);
   puts(p);
   operator delete(p);
}
```

Листинг lowlevelmem.cpp

Работа с памятью в стиле си

В `<cstdlib>` объявлены функции:

- `void* malloc(size_t n)` — memory allocation — выделяет память размером n байт и возвращает, как и две следующие функции, указатель на её начало;
- `void* calloc(size_t n, size_t s)` — call allocation — выделяет n раз по s байт, инициализированных 0;
- `void* realloc(void *p, size_t n)` — reallocation — изменяет рамер данных, указываемых p, на n байт;
- `void free(void *p)` — освобождает данные по указателю p.

Нельзя память, выделенную `new`, освобождать вызовом `free` и т. п.
В `<cstring>` объявлены функции:

- `void memchr(const void *p, int b, size_t n)` — memory character — подобна `strchr`, но ищёт байт b в отрезке памяти длиной n, начиная с p;
- `int memcmp(const void *p, const void *q, size_t n)` — memory compare — подобна `strcmp`;
- `void* memset(void *p, int b, size_t n)` — memory set — заполняет участок памяти от p длиной n байтом b;
- `void* memcpy(void *p, const void *q, size_t n)` — memory copy — подобна `strcpy`;
- `void* memmove(void *p, const void *q, size_t n)` — memory move — подобна `memcpy`, но области, начинающиеся с p и q, могут перекрываться.

Рассмотрим пример, печатающий Hello.

```
#include<cstdlib>
#include<cstdio>
#include<cstring>
using namespace std;
main() {
   char s[] = "Hello", *p = (char*)malloc(strlen(s) + 1);
   strcpy(p, s);  //или memcpy(p, s, strlen(s) + 1)
   puts(p);
   free(p);
}
```

Листинг malloc.cpp

Сопроцессы или процессорные нити

Работа с ними обеспечивается несколькими способами:

1) стандартной библиотекой по стандарту от 2011 года, предоставляющей, в частности, заголовок `<thread>`*;
2) си-библиотеками по стандарту POSIX, предоставляющими, в частности, заголовок `<pthread.h>`.

* На 2015 она не поддерживается транслятором gcc.

Средства из последнего заголовка включают следующие функции:

- `int pthread_create(pthread_t *p, pthread_attr_t *attr, void *(*start)(*void), void *arg)` — создание процессорной нити. Аргумент *p* указывает на переменную для хранения информации о сопроцессе, он подобен аргументу типа `FILE*` для файловых операций. Аргумент *attr* используется для передачи опциональных атрибутов в нить, *start* — это указатель на функцию, исполняемую сопроцессом, а *arg* указатель на её аргумент. Результат функции 0, если создание нити прошло без ошибок;
- `void pthread_exit(void* data)` — заканчивает исполнение сопроцесса, передавая указатель на данные основному процессу. Если не использовать эту функцию, то она будет вызвана неявно при завершении сопроцесса с указателем на результат в `return`;
- `int pthread_join(pthread_t thread, void **ret)` — подобна функции `wait` для процессов, ждёт окончания выполнения заданной аргументом *thread* нити. Завершившись, сопроцесс передаёт данные через *ret*. Результат функции 0, если соединение нити с основным процессом прошло успешно.

В следующем примере должно распечататься 7x11=77 global=16 bye.

```
#include<pthread.h>
#include<iostream>
using namespace std;
int global;
long mul(void *a) {
   global = 5;
   return ((int*)a)[0]*(((int*)a)[1]);
}
void* test(void*) {
   global = 16;
   pthread_exit((void*)"bye\n");
}
int main() {
   pthread_t thread1, thread2;
   int a[2] = {7, 11};
   long d = 52;
   char *s;
   if (pthread_create(&thread1, 0, (void*(*)(void*))mul, (void*)a))
      throw 1;
   while (global != 5);
   if (pthread_create(&thread2, 0, test, 0)) throw 2;
   if (pthread_join(thread1, (void**)&d)) throw 3;
   if (pthread_join(thread2, (void**)&s)) throw 4;
   cout << a[0] << 'x' << a[1] << '=' << d << " global="
      << global << ' ' << s << endl;
}
```

Листинг pthread1.cpp

Компилировать программы, использующие `<pthread.h>` нужно с ключами `-D_REENTRANT -lpthread`.

В `<pthread.h>` объявлены также средства для работы с мьютексами, обеспечивающими блокировку ресурсов при использовании нитей:

- `int pthread_mutex_init(pthread_mutex_t *pm, const pthread_mutexattr_t *attr)` — создание мьютекса по указателю *pm*. Аргумент *attr* используется для передачи опциональных атрибутов;
- `int pthread_mutex_lock(pthread_mutex_t *pm)` — делает следующий код доступным только для одной нити;
- `int pthread_mutex_unlock(pthread_mutex_t *pm)` — снимает блокировку, установленную предыдущей функцией;
- `int pthread_mutex_destroy(pthread_mutex_t *pm)` — уничтожение мьютекса.

Функции с результатом для мьютексов возвращают 0 при успешном завершении.

Расмотрим пример, в котором два конкурирующих сопроцесса набирают очки.

```
#include<pthread.h>
#include<iostream>
using namespace std;
pthread_mutex_t mr;
unsigned sum, sum1, sum2, i1, i2;
```

```
void incsum() {
   pthread_mutex_lock(&mr);
   ++sum;
   pthread_mutex_unlock(&mr);
}
void* task1(void*) {
   for (i1 = 0; i1 < 1000000; i1++)
      if (i2%2)
         incsum(), sum1++;
}
void* task2(void*) {
   for (i2 = 0; i2 < 1000000; i2++)
      if (i1%2)
         incsum(), sum2++;
}
int main() {
   pthread_t thread1, thread2;
   if (pthread_mutex_init(&mr, 0)) throw 0;
   if (pthread_create(&thread1, 0, task1, 0)) throw 1;
   if (pthread_create(&thread2, 0, task2, 0)) throw 2;
   if (pthread_join(thread1, 0)) throw 3;
   if (pthread_join(thread2, 0)) throw 4;
   if (pthread_mutex_destroy(&mr)) throw 5;
   cout << sum1 << '+' << sum2 << '=' << sum << ' ' << 100 - sum/20000. << "% skipped\n";
   if (sum1 > sum2) cout << "the 1st thread";
      else if (sum1 < sum2) cout << "the 2nd thread";
      else cout << "no one";
   cout << " has won\n";
}
```

Листинг pthread2.cpp

Нитями можно управлять средствами, подобными сигналам. В `<pthread.h>` есть средства:

- `int pthread_cancel(pthread_t thread)` — запрос на завершение нити *thread*;
- `void pthread_setcancelstate(int state, int *oldstate)` — значение `PTHREAD_CANCEL_ENABLE` для *state* делает нить доступной для управления предыдущей функцией, а значение `PTHREAD_CANCEL_DISABLE` — недоступной. Через опциональный параметр *oldstate* можно получить прежнее значение *state*;
- `void pthread_setcanceltype(int type, int *oldtype)` — значение `PTHREAD_CANCEL_ASYNCHRONOUS` для *type* означает, что сопроцесс должен завершиться немедленно после соответствующего запроса, но система не гарантирует этого. Значение `PTHREAD_CANCEL_DEFERRED` — только во время вызова заданных в стандарте функций, например, `pthread_testcancel` без аргументов, которая не имеет никакого другого эффекта, кроме обеспечения точки выхода из нити по запросу.

Рассмотрим пример управления сопроцессом.

```
#include<pthread.h>
#include<unistd.h>  //sleep
#include<iostream>
using namespace std;
void* task(void*) {
   pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, 0);
   pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, 0);
   for (int i = 1; i < 10; i++) {
      sleep(1);
      cout << "the thread is running for " << i << " seconds\n";
   }
}
int main() {
   pthread_t thread;
   if (pthread_create(&thread, 0, task, 0)) throw 1;
```

```
    sleep(5);
    if (pthread_cancel(thread)) throw 2;
    if (pthread_join(thread, 0)) throw 3;
    cout << "ok\n";
}
```

Листинг pthread3.cpp

Литература

1) International standard ISO/IEC 14882. *Programming languages — C++.* //Third edition 2011-09-01. — 1356 p.
2) International standard ISO/IEC 14882. *Programming languages — C++.* //Second edition 2003-10-15. — 757 p.
3) International standard ISO/IEC 9899. *Programming languages — C.* //Second edition 2000-5-22. — 554 p.
4) М. И. Болски *Язык программирования Си. Справочник* /Пер. с англ. — М.: «Радио и связь», 1988 г. — 96 с.
5) Р. Лафорте *Объектно-ориентированное программирование в C++. Классика Computer Science. 4-е изд.* — СПб.: Питер, 2011. — 928 с.
6) Бьерн Страуструп *Язык программирования C++, 3-е изд.* /Пер. с англ. — СПб.; М.: «Невский диалект» — «Издательство БИНОМ», 1999 г. — 991 с.
7) Дерк Луис *C и C++* — М.: Бином, 1997.
8) Б. В. Керниган, Р. Пайк *Unix — универсальная среда программирования* — М.: Финансы и статистика, 1992.
9) Романовская Л. М., Русс Т. В., Свитковский С. Г. *Программирование в среде Си для ПЭВМ ЕС* — М.: Финансы и статистика, 1992.
10) Собоцинский В. В. *Практический курс Turbo C++* — М.: ИПК Московская правда, 1993.

ОГЛАВЛЕНИЕ