# 1 Язык Си++

Язык программирования Си++ разрабатывался как развитие языка Си с начала 80-х годов. Страуструп экспериментировал с добавлением в процедурный язык достаточно новых в то время объектно-ориентированных возможностей языков Smalltalk и Simula. С другой стороны необходимым условием признания нового языка была как можно более полная совместимость с уже широко распространённым языком Си. Такая совместимость, по мысли автора, дала бы возможность постепенного перехода с Си на Си++ с постепенным отмиранием Си.

В конце 80-х годов был создан комитет по стандартизации языка Си++, который в 1998 году принял стандарт языка Си++. Следует заметить, что за время работы комитета язык достаточно сильно изменился: появились множественное наследование, шаблоны, исключения, пространства имён. В результате язык Си++ отличается очень высокой сложностью для как для освоения программистами, так и для реализации. За время, прошедшее с момента официального принятия стандарта, так пока и не появилось компилятора, в полной мере поддерживающего стандарт языка.

Хотя комитеты по языкам Си и Си++ работали в сотрудничестве друг с другом, языки развивались по-разному, что было зафиксировано как стандартом Си++ 1998 года, так и стандартом Си 1999 года. Правильнее говорить те то, что язык Си++ является надмножеством языка Си, а то, что они имеют общие корни и существует достаточно большое общее подмножество этих языков.

Язык Си++ предоставляет возможности вызывать из программы на Си++ функции из модулей, написанных на Си. Таким образом из программы на Си++ можно использовать все библиотеки, написанные на языке Си без использования каких-либо дополнительных средств, обеспечивающих согласование формата передачи параметров и имен функций для разных языков.

Язык Си++ является «объектно-ориентированным» языком, то есть языком, в котором средства, необходимые для объектно-ориентированного программирования, встроены в язык. С другой стороны, язык сохранил все средства модульного программирования в том виде, в котором они присутствуют в языке Си.

## 1.1 Классы

Одним из важных свойств языка является открытость его системы типов. Другими словами программист должен иметь возможность добавлять новые типы таким образом, что их использование не будет отличаться от использования стандартных типов.

Если мы посмотрим на, например, тип **`int`**, то, с одной стороны, мы, конечно, знаем, что значение типа **`int`** представляется определённым количеством бит, но с другой стороны в подавляющем большинстве случаев при написании программ (особенно если это не низкоуровневое программирование) внутреннее представление значения типа **`int`** нам безразлично. Нас интересуют только операции, определённые над значениями этого типа. Таким образом мы можем рассматривать некоторый тип как набор операций, определённых над значениями типа, и набор констант.

Так возникает поняетие *абстрактного типа данных*, то есть типа, использование которого возможно только с помощью определённых над типом операций. Однако абстрактный тип данных — это взгляд с точки зрения использования типа. Естественно, при определении операций над типом нам может потребоваться знание внутреннего представления значений типа. Таким образом требуется разделение прав доступа, чтобы предоставить реализации

операций над типом полный доступ к внутренней структуре, но, с другой стороны, ограничить доступ со стороны. В объекно-ориентированной терминологии такой подход называется *инкапсуляцией*.

В качестве первого примера попробуем написать реализацию типа данных комплексных чисел. Будем его рассматривать как абстрактный тип, то есть все его пользователи могут использовать только операции работы с комплексными числами, но не имеют доступа к внутреннему представлению чисел.

```
class complex
{
 private:
  double re, im;
 public:
  double abs();
  //...
};
```

По сути определение класса — это определение полей структуры и методов для работы с этой структурой. Для полей и методов можно указывать права доступа к ним. **private** означает, что доступ к соответствующим полям или методам имеют только методы класса и дружественные функции (которые мы рассмотрим позже), **public** означает, что соответствующие методы или поля общедоступны.

*Класс* — это тип данных, состоящий из полей и методов, то есть функций, определённых для работы с этими полями. *Объект* — это экземпляр класса, существующий в работающей программе, под который выделена память.

Идентификатор `complex` по аналогии с структурами языка Си — это тег класса. То есть далее в программе тег можно использовать вместе с ключевым словом **class** для указания имени типа, например `class complex a;`. Однако язык Си++ идёт дальше и для того, чтобы сделать пользовательские типы и встроенные типы неразличимыми разрешает использовать теги классов, структур, объединений и перечислимых типов без указания соответствующего ключевого слова тогда, когда это не приводит к конфликтам. Например, вместо `class complex a;` можно написать просто `complex a;`.

Теперь напишем реализацию метода `abs` для вычисления абсолютного значения комплексного числа. У нас есть две возможности: мы можем разместить реализацию метода непосредственно в определении класса `complex`, а можем вынести реализацию из объявления. Сначала попробуем второй вариант:

```
double complex::abs()
{
  return sqrt(this->re*this->re+this->im*this->im);
}
```

Обратите внимание, что в отличие от Си пустая пара скобок `()` в заголовке метода означает, что метод не принимает параметров, и такая запись эквивалентна указанию ключевого слова **void**, например `double complex::abs(void)`.

Идентификатор `complex` в названии метода указывает то, что метод `abs` является методом именно класса `complex`, а не какого-либо другого класса или глобальным методом. Поскольку мы определяем метод вне определения класса, указание класса, к которому относится метод, обязательно.

Нестатические методы класса (такие как `abs`) всегда применяются к некоторому объекту класса, поэтому в метод необходимо каким-либо образом передать информацию о том, к

какому объекту метод применяется. Поэтому у всех нестатических методов помимо параметров, задаваемых пользователем, всегда присутствует неявный параметр **this**, имеющий тип указателя на соответствующий класс. **this** — это ключевое слово, поэтому оно не может быть никогда переопределено в программе.

Внутри метода мы можем обращаться к полям и методам нашего класса посредством указателя **this**, как показано на примере. Для удобства **this** мы можем опустить, если это не приведёт к неоднозначностям. Таким образом метод `abs` запишется следующим образом:

```
double complex::abs()
{
  return sqrt(re*re+im*im);
}
```

При анализе программы всякий идентификатор, использованный в методе, сначала ищется в блочных и локальных переменных метода, затем в определениях полей и методов класса, затем в глобальных переменных и функциях.

У нас есть ещё одна возможность определить метод `abs` — непосредственно при определении класса. Выглядеть это будет следующим образом:

```
class complex
{
 private:
  double re, im;
 public:
  double abs() { return re*re+im*im; }
};
```

Если метод определяется внутри определения класса, то указание имени класса не требуется. С точки зрения семантики программы оба способа определения методов эквивалентны друг другу. Однако, способ, при котором метод определяется внутри класса, может приводить к генерации чуть более эффективного кода. Дело в том, что компилятор может выполнять подстановку тела метода в точку его вызова, если метод достаточно прост по структуре. Того же эффекта можно добится и определением метода вне определения класса, но тогда нужно использовать специальное ключевое слово **inline** и выполнить несколько других условий.

Использование методов аналогично использованию полей структур и классов. Например:

```
complex v, *pv;
// ...
x = v.abs();
y = pv->abs();
```

То есть для вызова метода для объекта используется либо `.`, либо `->` в зависимости от того, обращаемся мы к объекту напрямую или через указатель.

В первом случае в теле метода `abs` значение указателя **this** будет равно `&x`, а во втором случае — `pv`.

## 1.2 Конструкторы

Поскольку поля данных класса `complex` закрыты для доступа извне, потребуется предоставить метод, с помощью которого можно выполнять установку полей метода, например

```
class complex
{
 public:
  //...
  void setValue(double _re, double _im);
};
```

Такой метод имеет право на существование и в некоторых случаях может быть полезным. Однако при таком подходе возникает проблема в том, что определение переменной класса окажется отделённой от её инициализации. В случае сложных классов инициализация полей класса может быть достаточно сложной, и её может оказаться необходимо выполнять сразу при создании объекта. Поэтому в языке Си++ вводится понятие *конструктора*. Конструктор — это специальный метод, который вызывается неявно в момент выделения памяти под объект. Например, если объект локальный для блока, конструктор для объекта будет вызываться при входе в блок, а если объект глобальный, то конструктор для такого объекта будет вызываться после запуска программы ещё до вызова функции `main`.

Конструктор определяется в классе как метод, имя которого совпадает с именем класса. Как и обычные методы, он может быть определён как внутри, так и вне класса.

```
class complex
{
  // ...
 public:
  complex(double _re, double _im);
};
complex::complex(double _re, double _im)
{
  re = _re; im = _im;
}
```

Обратите внимание, что у конструктора не может быть типа возвращаемого значения. Даже указание **void** является неверным. Конструктор класса не может быть вызван к уже существующему объекту. Явный вызов конструкторов допускается только при создании объектов.

Теперь мы можем при создании объекта класса `complex` указывать начальные значения полей следующим, например так:

```
  complex c1(1.0, 2.0);
  complex c2 = complex(1.0, 2.0); // то же самое, что и выше
```

или даже так:

```
void bar(complex x);
//...
  bar(complex(1.0, 2.0));
```

здесь при передаче параметров в функцию `bar` создаётся временный объект класса `complex`, который инициализируется вызовом конструктора.

Если мы объявили конструктор с параметрами, то попытка определить переменную без инициализации приведёт к ошибке.

```
  complex z;
```

Дело в том, что если в классе не определён ни один конструктор, то компилятор Си++ должен сгенерировать необходимые конструкторы (а таких два: конструктор по умолчанию

и конструктор копирования) сам. Но если в классе определены пользовательские конструкторы, никаких конструкторов компилятор автоматически не генерирует.

Конструктор по умолчанию — это конструктор, не принимающий аргументов. Например:

```
class complex
{
  // ...
 public:
  complex() { re = im = 0.0; }
};
```

Конструктор по умолчанию вызывается при создании объекта, если он не инициализируется пользовательским конструктором или конструктором копирования. Если в классе конструкторы отсутствуют, компилятор при необходимости генерирует пустой (то есть не выполняющий никаких действий) конструктор по умолчанию.

## 1.3 Перегрузка

Теперь возникла ситуация, когда в классе есть два конструктора, один с двумя параметрами, другой — без параметров. В языке Си такая ситуация недопустима, но Си++ допускает *перегрузку* методов и функций. У класса может быть несколько методов с одним именем отличающихся по количеству или типам передаваемых параметров. В точке использования методов компилятор подберёт наиболее подходящий метод. Например рассмотрим несколько вариантов функции `sqrt`, возвращающей квадратный корень числа:

```
int sqrt(int x);
double sqrt(double x);
```

Теперь, если вызывается `sqrt(1)`, то будет вызван вариант `sqrt` для целого параметра, а если `sqrt(1.0)`, то для параметра **`double`**. Однако, перегрузка может приводить к неоднозначностям. Например, если `sqrt` вызывается как `sqrt(1L)`, то есть для аргумента типа **`long`**, ни один из существующих вариантов не подходит, поэтому компилятор пытается подобрать необходимую операцию приведения типа, и здесь подходящими оказываются и приведение из **`long`** в **`int`**, и из **`long`** в **`double`**. Когда компилятор сталкивается с неоднозначностью, выдаётся ошибка.

Перегруженные функции не могут различаться только по типу возвращаемого значения, то есть перегрузка

```
int rand();
double rand();
```

недопустима. Кроме того, не могут быть перегружены функции, имеющие соглашение о связях языка Си (extern "C"), а также функции имеющие неограниченное количество параметров, если они отличаются только в части, соответствующей неограниченному количеству параметров.

```
// неверно!
void func(int a, ...);
int  func(int a, int b);
// а так можно
void func2(int a, ...);
void func2(double a, ...);
```

Главное требование к перегрузке функций — однозначность. При любом количестве параметров и любых типов параметров компилятор должен однозначно определить, какую функцию из перегруженных можно вызвать.

## 1.4 Параметры по умолчанию

Язык Си++ позволяет задавать значения параметров по умолчанию для функций и методов, то есть параметры, которые могут быть опущены в точке вызова соответствующей функции или метода.

Например, конструктор класса `complex` можно определить следующим образом:

```
class complex
{
 public:
  complex(double _re = 0.0, double _im = 0.0);
};
```

Мы определили конструктор с двумя параметрами, каждый из которых имеет значение по умолчанию и таким образом может отсутствовать в точке вызова этого конструктора. Теперь этот конструктор будет использоваться и как конструктор по умолчанию, так как допускается вызов без параметров — и `_re`, и `_im` получат значения `0.0`, и как конструктор с одним аргументов, и как конструктор с двумя аргументами, например:

```
complex z1, z2 = 5.0, z3(0.0, 1.0);
```

Если в точке вызова функции или метода задано меньше фактических параметров, чем требуется функции, то параметры получают значения в порядке их появления в списке параметров слева направо, те формальные параметры, фактических значений которых не хватило, получают значения по умолчанию. Не существует способа пропустить несколько первых параметров по умолчанию и задать значения только следующих за ними. Таким образом, в вызове конструктора `z2 = 5.0` параметр `_re` получит значение `5.0`, а параметр `_im` — значение по умолчанию `0.0`.

На определение параметров со значениями по умолчанию накладываются некоторые ограничения. Во-первых, значения по умолчанию могут быть определены для последних формальных параметров функции или метода. Нельзя, например, задать значение по умолчанию для первого параметра, но не для остальных параметров:

```
void func(int x = 0, int y); // неверно!
```

Во-вторых, задание значений параметров по умолчанию не должно приводить к неоднозначностям при подборе наиболее подходящего варианта из перегруженных функций или методов. Так, в случае конструктора класса `complex` с двумя аргументами по умолчанию уже нельзя будет определить конструктор без аргументов или конструктор с одним аргументом.

```
class complex
{
 public:
  complex(double _re = 0.0, double _im = 0.0);
  complex(); // неверно! в случае 'complex z;' - неоднозначность
};
```

## 1.5 Константы в программе

В языке Си++ если переменная определена как константная, то её значение может использоваться в константных выражениях в программе. В языке Си такое было невозможно. Например:

```
static const int MAX_SIZE = 100;
static int arr[MAX_SIZE];
```

## 1.6 Ссылочные типы

Язык Си++ вводит ссылочные типы, которые позволяют реализовать передачу параметров в функции или методы по ссылке, как во многих других языках. Определение ссылочного типа похоже на определение указательного типа, только вместо * используется &. Например:

```
int a;
int &b = a;
```

Здесь определяется ссылочная переменная b, которая ссылается на переменную a. Обратите внимание, ссылочная переменная может быть проинициализирована только в точке определения. После определения ссылочной переменной b к ячейке памяти, в которой находится значение переменной a, можно обратиться и по имени a, и по имени b.

```
a = 10;
cout << b; // будет напечатано 10
b = 20;
cout << a; // будет напечатано 20
```

Изменение переменной b на самом деле означает изменение переменной a. Таким образом, ссылочный тип позволяет задать синонимы (альтернативные имена) для переменных. С точки зрения программиста обращение к значению переменной по основному имени и по синонимичным переменным эквивалентно.

Основное назначение ссылочных типов — для передачи параметров по ссылке.

```
void swap(int &a, int &b)
{
  int t = a;
  a = b; b = t;
}
```

Функция swap принимает два параметра ссылочного типа. В точке вызова функции swap формальные параметры a и b станут ссылками на соответсвующие переменные, переданные в качестве фактических параметров функции, и работая с параметрами a и b в функции swap, мы фактически работаем с теми переменными, которые будут переданы в качестве фактических параметров при вызове функции.

```
int a = 1, b = 2;

  swap(a, b);
  cout << a << ' ' << b; // напечатает "2 1"
```

Ссылки могут реализовываться как указатели, при этом при передаче по ссылке может неявно добавляться операция взятия адреса, а при обращении по ссылке — операция разыменования. Таким образом, функция swap может быть оттранслирована в следующий код:

```
void swap(int * const a, int * const b)
{
  int t = *a;
  *a = *b; *b = t;
}
```

А вызов функции `swap` будет оттранслирован в код:

```
int a = 1, b = 2;

  swap(&a, &b);
  cout << a << ' ' << b;
```

Поскольку при передаче по ссылке фактически берётся адрес некоторого объекта, на месте формального параметра, передаваемого по ссылке при вызове функции должно находиться выражение, для которого определена операция взятия адреса.

```
int a, b, c;

  swap(a, 1); // неверно - операция &1 не определена
  swap(a, b + c); // неверно - операция &(b + c) не определена
```

Ссылки могут быть константными, например:

```
void print_int(const int &x)
{
  cout << x;
}
```

На месте формального параметра-константной ссылки позволяется указывать выражение, к которому операция взятия адреса неприменима. В этом случае компилятор создаст временную переменную, которой присвоит значение выражения, и затем ссылка на эту временную переменную будет передана в функцию. Например,

```
int a, b;

  print_int(1); // означает: t1 = 1; print_int(t1);
  print_int(a + b); // означает: t2 = a + b; print_int(t2);
```

## 1.7 Конструктор копирования

Предположим, что мы хотим написать функцию, складывающую два комплексных числа. Пусть функция принимает два параметра типа `complex` и возвращает результат типа `complex`. Мы пока предположим, что функция имеет полный доступ ко внутреннему представлению комплексных чисел.

```
complex add(complex a, complex b)
{
  complex res = complex(a.re + b.re, a.im + b.im);
  return res;
}
```

Использовать такую функцию можно, например, следующим образом:

```
complex z1, z2, z3;

//...
z3 = add(z1, z2);
```

При вызове функции `add` необходимо скопировать значения переменных `z1` и `z2`, передаваемых в качестве фактических параметров, в формальные параметры `a` и `b` соответственно, при этом копирование может выполняться одновременно с выделением места на стеке под параметры. Поскольку копирование одного объекта в другой может требовать нетривиальных операций, для реализации копирования может быть определён специальный *конструктор копирования*.

Конструктор копирования для класса `complex` может быть определён следующим образом.

```
class complex
{
  double re, im;
 public:
  complex(const complex &v) { re = v.re; im = v.im; }
};
```

Конструктор копирования — это конструктор, принимающий один параметр, тип которого — константная ссылка на тот же самый класс. Конструктор копирования используется всякий раз, когда необходимо создать новый объект по уже существующему объекту того же типа, например, при передаче параметров по значению в функции, при возврате значения из функции, при определении переменной, инициализируемой значением того же типа. Так, в примере реализации функции `add` выше, конструктор копирования будет вызван не только при передаче значений в функцию, но и в операторе `return res;` для того, чтобы скопировать значение локальной переменной `res` в область памяти, в котором хранится возвращаемое значение.

Можно ли уменьшить число вызовов конструкторов копирования при вызове функции `add`? Для этого можно, во-первых, передавать параметры не по значению, а по константной ссылке, и, во-вторых, создавать возвращаемый объект непосредственно в операторе **`return`** следующим образом:

```
complex add(const complex &a, const complex &b)
{
  return complex(a.re + b.re, a.im + b.im);
}
```

Обратите внимание, что возвращать из функции ссылку на локальный объект нельзя, так как это эквивалентно возврату адреса локальной переменной, которая перестанет существовать сразу же после выхода из функции.

```
complex &add(const complex &a, const complex &b)
{
  complex res = complex(a.re + b.re, a.im + b.im);
  return res; // НЕВЕРНО! возвращается ссылка на локальную перем.
}
```

## 1.8 Автоматическая генерация конструкторов

Как было сказано выше, конструктор по умолчанию и конструктор копирования имеют особое положение среди всех прочих конструкторов. Компилятор может (а в некоторых случаях и должен) сгенерировать эти конструкторы автоматически.

Если в классе не определён конструктор копирования, он определяется неявно как побайтовое копирование полей объекта-аргумента конструктора в новый объект. В некоторых случаях такое поведение конструктора копирования является корректным, но иногда требуется явное задание конструктора копирования. Такая необходимость, в частности, возникает, когда класс использует какие-либо ресурсы, которые необходимо захватывать или освобождать явно, например, файловые дескрипторы, динамическую память и пр.

Компилятор Си++ генерирует конструктор по умолчанию, если в классе не определено никаких конструкторов. Сгенерированный таким образом конструктор по умолчанию не выполняет никаких действий.

## 1.9 Константные методы

Рассмотрим метод `abs` в классе `complex`, который вычисляет модуль комплексного числа. Этот метод вычисляет свой результат по полям класса `complex`, однако не модифицирует их. Можно сказать, что метод `abs` не изменяет состояние объекта класса `complex`. В этом случае метод `abs` следует определить как константный.

```
class complex
{
 public:
  double abs() const;
};
double complex::abs() const
{
  return re*re+im*im;
}
```

Признаком константного метода является ключевое слово **`const`** после списка параметров метода. Обратите внимание, что если метод определяется вне тела класса, то ключевое слово **`const`** нужно повторить и при определении метода.

Константные методы применимы и к константным, и к неконстантным объектам, в то время, как неконстантные методы применимы только к неконстантным объектам. Например, если для класса `complex` определён константный метод `abs` и неконстантный метод `neg`:

```
class complex
{
 public:
  double abs() const { return re*re+im*im; }
  void neg() { re = -re; im = -im; }
};

complex c;
const complex cc;

  c.abs(); // верно
  c.neg(); // верно
```

```
  cc.abs(); // верно
  cc.neg(); // ошибка: переменная cc - константная
```

Константность метода является частью его сигнатуры, то есть в классе может быть определено два метода с одинаковым списком параметров, но один из которых константный, а другой — нет. Тогда в точке вызова метода компилятор выберет подходящий в зависимости от константности объекта:

```
class complex
{
 public:
  double abs() const; // определение вынесено из класса
  double abs(); // определение вынесено из класса
};

complex c;
const complex cc;

  c.abs(); // вызывается метод для неконстантных объектов
  cc.abs(); // вызывается метод для константных объектов
```

## 1.10 Дружественные функции

Функция `add`, которая складывает два комплексных числа, не является членом класса `complex` и, как следствие, не имеет доступа к закрытым `private` полям класса. Конечно, можно использовать в этой функции какие-нибудь публичные методы работы с классом, но, с другой стороны, функция `add`, по сути, тесно связан с классом `complex`, и ему можно открыть доступ к закрытым полям.

Для этого функция `add` может быть сделана дружественной для класса `complex`. Дружественные функции имеют полный доступ к закрытым полям и методам класса. Чтобы сделать функцию дружественной, нужно её прототип поместить в класс, пометив его ключевым словом **`friend`**, как в следующем примере:

```
class
{
  friend complex add(const complex &a, const complex &b);
};
```

Дружественная функция не является членом класса, поэтому её реализация не может находиться в классе, а должна быть вынесена из него, кроме того, имя функции `add` не должно квалифицироваться именем класса, то есть запись `complex::add` неправильна. Одна и та же функция может быть дружественной для нескольких классов.

## 1.11 Перегрузка операций

Как было сказано выше, одной из целей, преследовавшихся при создании языка Си++, было обеспечение «равенства» между встроенными типами и типами, определёнными пользователем. Для этого требуется, чтобы пользователь мог определять семантику операций, доступных в выражениях, для пользовательских типов. Например, для типа комплексных

чисел требуется определить все арифметические операции +, - и т. д, для строкового типа также можно определять некоторые операции.

Язык Си++ позволяет определять операции для пользовательских типов. Определены могут быть почти все операции, кроме `.` (доступ к полю), `.*` (доступ к полю по указателю на функцию), `::` (доступ к области видимости), `?:` (условная операция). Все прочие операции могут быть определены пользователем. Для любого класса по умолчанию определены операции присваивания (=), взятия адреса (`&`) и следования (`,`), но и они могут быть переопределены пользователем.

На определение операций накладываются следующие ограничения:

- Нельзя изменить синтаксическую структуру выражений, то есть нельзя добавить новые знаки операций (например, ввести новую операцию `@`, бинарную операцию `!` или унарную операцию `^`), нельзя изменить приоритет и ассоциативность операций. Эти ограничения гарантируют, что для синтаксического анализа программы на Си++ не потребуется информация об операциях, определённых для класса.

- Операции =, `[]`, `()` и `->` должны определяться как нестатические методы класса, а не как дружественные функции. Таким образом гарантируется наличие объекта, над которым выполняется операция.

- Нельзя переопределить операции над встроенными типами, то есть, например, нельзя определить пользовательскую операцию сложения двух значений типа **`int`**.

Операции, которые можно определить перечислены ниже: + (унарная и бинарная), - (унарная и бинарная), * (унарная и бинарная), / (бинарная), `%` (бинарная), `^` (бинарная), `&` (унарная и бинарная), `|` (бинарная), ~ (унарная), = (бинарная), < (бинарная), > (бинарная), += (бинарная), -= (бинарная), *= (бинарная), /= (бинарная), %= (бинарная), ^= (бинарная), |= (бинарная), &= (бинарная), <<= (бинарная), >>= (бинарная), << (бинарная), >> (бинарная), == (бинарная), != (бинарная), <= (бинарная), >= (бинарная), `&&` (бинарная), `||` (бинарная), ++ (унарная префиксная и постфиксная), -- (унарная префиксная и постфиксная), `->*` (бинарная), `,` (бинарная), -> (унарная), `[]` (бинарная), `()` (бинарная), `new`, `new[]`, `delete`, `delete[]`.

Рассмотрим некоторую бинарную операцию, которую мы обозначим `@`. Применение бинарной операции к своим аргументам, например, `a @ b`, может рассматриваться одним из двух способов:

1. Как применение метода с именем `operator @` и аргументом `b` к объекту `a`, то есть как `a.operator @(b)`.

2. Как применение функции (не члена класса) с именем `operator @` и аргументами `a` и `b`.

Несколько операций (=, `[]`, `()`) допускают только первое толкование. Компилятор при анализе программы проверяет оба варианта.

Например, определить бинарную операцию + для класса `complex` как член класса можно следующим образом:

```
class complex
{
public:
  complex operator+ (const complex &b) const;
```

```
};
complex complex::operator+ (const complex &b) const
{
  return complex(re + b.re, im + b.im);
}
```

Бинарную операцию + для класса `complex` как функцию можно следующим образом:

```
class complex
{
public:
  friend complex operator+ (const complex &a, const complex &b);
};
complex operator+ (const complex &a, const complex &b)
{
  return complex(a.re + b.re, a.im + b.im);
}
```

Какой из вариантов предпочтительнее мы рассмотрим ниже.

Рассмотрим теперь унарные операции +, -, *, ~, &, !, ++ (префиксная), -- (префиксная). Обозначим эти операции как `@`. Применение унарной операции `@ a` к своему аргументу может рассматриваться либо как применение метода `operator @` без параметров к объекту `a`, то есть как `a.operator@()`, либо как вызов функции `operator@` с одним аргументом `a`, то есть `operator@(a)`.

Таким образом, определить унарную операцию - для класса `complex` как метод этого класса можно следующим образом:

```
class complex
{
public:
  complex operator- () const;
};
complex complex::operator- () const
{
  return complex(-re, -im);
}
```

Унарную операцию - как функцию-не метод класса определить можно следующим образом:

```
class complex
{
public:
  friend complex operator- (const complex &a);
};
complex operator- (const complex &a)
{
  return complex(-a.re, -a.im);
}
```

При определении операций инкремента ++ и декремента -- возникает сложность, так как и префиксный и постфиксный вариант операций — это унарная операция. Поэтому чтобы отличить префиксную и постфиксную операцию используется приём с фиктивным формальным параметром. Префиксные операции инкремента и декремента определяются как

обычные унарные операции, как показано выше, а для определения постфиксных операций требуется, чтобы соответствующие методы или функции принимали дополнительный параметр типа **int**.

Определение операции постфиксного инкремента для класса `complex` как метода класса:

```
class complex
{
public:
  complex operator++ (int b) { return complex(re++,im); }
};
```

Обратите внимание, что `operator++` возвращает значение типа `complex`. Это необходимо, чтобы в выражении можно было написать, например `-a++`.

Определение операции постфиксного инкремента для класса `complex` как дружественной функции:

```
class complex
{
public:
  friend complex operator++ (complex &a, int b);
};
complex operator++ (complex &a, int b)
{
  return complex(a.re++, a.im);
}
```

Далее мы рассмотрим некоторые особые случае переопределения операций для классов.

## 1.12 Операция присваивания

Операция присваивания может быть определена для класса `complex` следующим образом:

```
class complex
{
public:
  complex &operator=(const complex &v);
};
complex &complex::operator=(const complex &v)
{
  re = v.re; im = v.im;
  return *this;
}
```

Операция присваивания должна быть нестатическим методом класса. В качестве параметра метода передаётся константная ссылка на собственный класс. Метод возвращает ссылку на собственный класс для того, чтобы результат операции присваивания можно было использовать в выражении, например `a = b = c;`. Оператор `return *this;` возвращает ссылку на объект, для которого был вызван метод операции присваивания, поскольку в качестве возвращаемого значения требуется ссылка на объект (что с точки зрения записи в программе на Си++ то же самое, что и просто объект), а `this` — это указатель на объект,

запись `*this` необходима для согласования типов. Никакого разыменования указателя при этом, конечно, выполняться не будет.

Операция присваивания очень похожа на конструктор копирования, и в случае нашего класса `complex` реализуются они одинаково, но между ними есть принципиальное различие. Конструктор копирования вызывается при создании объекта для того, чтобы корректно задать значения полей объекта. Операция присваивания вызывается, чтобы изменить значение уже существующего объекта, поля которого уже содержат некоторые значения. Если при создании объекта захватываются ресурсы, то оператор присваивания должен сначала корректно освободить ресурсы, уже занятые объектом, и после этого выполнить присваивание новых значений полей.

В качестве примера рассмотрим простой класс для чтения из файлов. В конструкторе открывается файловый дескриптор, который и используется для операций с файлами.

```
class InputFile
{
private:
  int fd;
public:
  InputFile(const char *path);
  InputFile(const InputFile &f);
  InputFile &operator=(const InputFile &f);
};
```

Конструктор копирования должен сделать копию файлового дескриптора, например так:

```
InputFile::InputFile(const InputFile &f)
{
  fd = dup(f.fd);
}
```

Операция присваивания должна перед созданием копии файлового дескриптора закрыть тот файловый дескриптор, который сохранён в поле `fd` текущего объекта.

```
InputFile& InputFile::operator=(const InputFile &f)
{
  close(fd);
  fd = dup(f.fd);
  return *this;
}
```

## 1.13 Деструкторы

В нашем простом примере класса для чтения из файла необходим метод, который будет закрывать файловый дескриптор `fd` класса. Можно реализовать метод `close`, который выполнит все необходимые операции, однако использовать такой метод неудобно по двум причинам: во-первых, программист должен не забыть вызвать метод `close` перед уничтожением объекта класса `InputFile`, во-вторых, программист может случайно использовать объект этого метода после вызова `close`.

Для решения проблемы освобождения ресурсов в момент уничтожения объекта предназначены *деструкторы*. Деструктор — это метод, который вызывается автоматически непосредственно перед освобождением памяти, выделенной под объект. Если объект — локальная или блочная переменная, то деструктор будет вызван при выходе из соответствующего

блока, если объект — глобальная переменная, то деструктор будет вызван при завершении программы после возврата из функции `main` или после вызова функции `exit`.

Деструктор — это метод без возвращаемого значения и без параметров. Деструктор всегда имеет имя `~<имя класса>`. В отличие от конструктора, деструктор не может быть перегружен.

В примере класса `InputFile` деструктор может быть определён следующим образом:

```
class InputFile
{
  // ...
public:
  ~InputFile();
};
InputFile::~InputFile()
{
  close(fd);
}
```

## 1.14 Неявные преобразования типов

Продолжим работать с типом `complex`. Мы уже определили операцию сложения для двух комплексных слагаемых, таким образом фрагмент программы

```
complex a, b, c;

c = a + b;
```

даст ожидаемый результат. Однако, если одно из слагаемых будет другого типа, например **`double`**, как будет вычисляться выражение в этом случае?

Из всех реализаций операции сложения компилятор должен подобрать единственную реализацию, подходящую по параметрам. В данном случае операция сложения комплексного и вещественного числа не определена. Тогда компилятор пытается подобрать такое привидение типов, после которого подходящая операция сложения бы нашлась. В нашем случае если было бы можно преобразовать вещественное число к комплексному, то после такого преобразования стала бы применима операция сложения двух комплексных чисел. Естественно, что для пользовательских типов, каким является тип `complex`, операции привидения типов могут быть предоставлены только пользователем, компилятор не в состоянии их сгенерировать автоматически.

Предположим, что в классе `complex` определён конструктор, допускающий вызов с одним аргументом типа **`double`**. В случае нашей реализации класса `complex` — это конструктор с двумя параметрами со значениями по умолчанию. По сути, такой конструктор позволяет получить из значения типа **`double`** значение типа `complex`. Значит, компилятор может создать временную переменную типа `complex`, передав в конструктор для неё второе слагаемое операции сложения. В результате получатся два объекта типа `complex`, для которых и будет вызвана операция сложения.

Таким образом, если в классе определён конструктор, который можно использовать с одним аргументом, такой конструктор будет использоваться в качестве операции приведения типа, если компилятор сочтёт это необходимым. Например, если добавить в класс `complex` конструктор

```
complex(const char *str);
```

появится возможность автоматического преобразования типов от типа `char *` к типу `complex`. Соответственно, выражение

```
complex a, b;
a = b + "(0,1)";
```

станет допустимым выражением для компилятора Си++ и будет успешно оттранслировано.

Поскольку в большой программе структура классов может быть очень сложной, ничем не ограниченный поиск подходящей цепочки преобразований типов может оказаться очень ресурсоёмким, требуется наложить ограничения на подбор типов при трансляции выражений.

Будем считать, что все возможные преобразования между встроенными типами равнозначны и всегда выполняются за один шаг. Например, преобразования из **`short`** в **`double`** и из **`double`** в **`short`** равнозначны и требуют одной операции, хотя, конечно, в машинном коде такие преобразования могут потребовать несколько инструкций процессора. Такие преобразования типов назовём *стандартными*. В отличие от языка Си, Си++ не поддерживает неявные преобразования между целыми и указательными типами и между разными указательными типами (за некоторыми исключениями).

Пусть даны два типа $T_1$ и $T_2$. Тип $T_1$ приводим к типу $T_2$, если существует цепочка не более чем из одного стандартного и одного пользовательского преобразования типов, которая из типа $T_1$ позволяет получить тип $T_2$. Такая цепочка должна быть единственной, так как в противном случае компилятор не может из нескольких цепочек выбрать одну.

Так, в нашем случае класса `complex` с конструктором-операцией преобразования типа из **`double`** в `complex` все целочисленные и вещественные типы оказываются приводимыми к типу `complex`, так как для любого такого типа можно сначала выполнить приведение к типу **`double`**, а затем — приведение к типу **`complex`**.

Однако, если мы добавим в класс `complex` конструктор

```
complex(int val);
```

то только типы **`int`** и **`double`** останутся приводимыми к типу `complex`, так как для любого другого встроенного типа возникнут две возможных цепочки приведений, например, для типа **`char`**: **`char`** → **`int`** → `complex` и **`char`** → **`double`** → `complex`.

Такой алгоритм подбора преобразований типов действует, естественно, не только для бинарных операций, но для любых выражений, в которых требуемый тип не соответствует фактическому типу выражения, в частности, при передаче параметров в функции и методы.

Поскольку функции могут быть перегруженными, и могут существовать разные пути неявных привидений типов, при подборе подходящей функции из перегруженных действуют следующие правила:

- Если существует перегруженная функция, типы формальных параметров которой в точности совпадают с типами фактических параметров, выбирается эта функция.
- Иначе если существует единственная перегруженная функция такая, что типы можно согласовать только с помощью стандартных преобразований, выбирается эта функция. Если подобных функций несколько, компилятор диагностирует ошибку.
- Иначе если существует единственная перегруженная функция такая, что типы фактических параметров приводимы к типам формальных параметров, выбирается эта функция. Если подобных функций несколько, компилятор диагностирует ошибку.

- Если ни один из вышеперечисленных вариантов не удовлетворяет, диагностируется ошибка.

## 1.15 Явные конструкторы

Как было сказано выше, конструкторы, которые могут быть вызваны с одним аргументом, используются компилятором там, где требуется преобразования типов от типа аргумента конструктора к типу самого конструктора. Эти конструкторы проверяются при подборе подходящих преобразований типа, и как следствие, их наличие может приводить к неоднозначностям.

Может оказаться полезным запретить неявное использование конструктора как оператора преобразования типов. Для этого используется ключевое слово **explicit**. Например, если в классе `complex` мы желаем определить новый конструктор из типа `const char *`, но не хотим, чтобы этот конструктор использовался компилятором неявно, мы можем определить его следующим образом:

```
class complex
{
public:
  explicit complex(const char *);
};
```

Определение конструктора с ключевым словом **explicit** запрещает всяческое неявное использование его компилятором, но, естественно, такой конструктор может вызваться в точках создания объекта явно. Например:

```
complex a, b;
  a = b + "1";          // неверно: конструктор вызван неявно
  a = b + complex("1"); // верно: конструктор вызван явно
```

## 1.16 Операции преобразования типов

Конструкторы с одним аргументом используются для приведения типов в направлении от уже существующих к новому типу. Такой подход не работает для преобразования ко встроенным или ранее написанным типом. Например, мы не можем дописать новый конструктор в стандартный класс `string`, поэтому конструкторы для этой цели не подходят.

Чтобы реализовать операции преобразования данного типа к другим типам можно использовать перегрузку оператора привидения к типу по аналогии с тем, как перегружаются операции. Например, преобразование из `complex` в **double** может быть определено следующим образом:

```
class complex
{
public:
  operator double () const { return re; }
};
```

Обратите внимание на отсутствие типа возвращаемого значения и на отсутствие аргументов у оператора преобразования. Также можно определить операцию преобразования к ранее определённым пользовательским типам, например, к стандартному классу `string`.

```
class complex
```

```
{
public:
  operator string () const { return string("complex"); }
};
```

Заметим, что перегруженные операции приведения к типу должны использоваться с осторожностью, так как в совокупности с конструкторами-операторами преобразования типов легко могут приводить к неоднозначностям. Например, если в классе `complex` определён и конструктор, который может вызываться с одним аргументом типа **`double`**, и операция преобразования к типу **`double`**, то такое выражение приведёт к неоднозначности:

```
complex a, b;
  a = b + 5.0; // неоднозначность!
```

здесь неоднозначность возникает потому, что мы можем комплексное значение `b` преобразовать к типу **`double`** и выполнить сложение в типе **`double`**, с одной стороны, и, с другой стороны, преобразовать `5.0` к типу `complex` и выполнить сложение в типе **`complex`**.