Введение в C++.
От структуры к классу.

СПбГЭТУ «ЛЭТИ»
ПЕРВЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ

Понятие и смысл ООП

1967 г. - Simula-67 — первые идеи по объединению данных и процедур их обработки в одном программном модуле

С 1969 по 1980 — развитие языка Smalltalk — первого полностью «объектно-ориентированного».

Наибольшая популярность — в C++. Полностью все концепции ООП в C++ сформировались к 1983 г., стандарт ISO — в 1996 г.

Подход ООП: данные существуют в виде некоторых объектов — «чёрных ящиков» (внутреннее устройство объекта извне недоступно или просто неизвестно).

Всё, что можно сделать с объектом — это послать ему сообщение и получить ответ.

Объекты, внутреннее устройство которых одинаково, образуют ***классы.***

Класс — описание внутреннего устройства объекта

При создании нового объекта указывается, к какому классу он принадлежит (объектом какого класса, он будет являться).

Методы (функции-члены)

Для поддержки объектно-ориентированного программирования в C++ к понятию *классов* добавляется понятие «*функций-членов*».

Пусть имеется структура для представления комплексного числа через действительную и мнимую части:

```
struct str_complex {
    double re, im;
};
```

Если требуется операция вычисления модуля комплексного числа, то можно написать функцию (пример **U00.cpp**), но C++ позволяет добавить нужную функцию как поле структуры:

```
struct str_complex {
    double re, im;
    double modulo() { return sqrt(re*re + im*im); }
};
```

Методы (функции-члены)

Функция **`modulo()`** называется *функцией-членом* или *методом*, структуры **`str_complex`**.

(*Если говорят о «методе объекта», то это означает метод того типа, к которому принадлежит объект*).

Метод вызывается не сам по себе, а для конкретного объекта, и как раз из этого объекта берутся поля, когда в теле метода имеет место обращение к полю структуры по имени.

Вызов метода — это именно то, что в теории ООП понимается как «отправка сообщения объекту».

Термины «вызов метода» и «передача сообщения» являются синонимами.

Как с этим работать: пример **`U01.cpp`**

Указатель *this*

Вызов функции-члена (метода) объекта с точки зрения реализации представляет собой абсолютно то же самое, что и вызов обычной функции, первым параметром которой является адрес объекта.

Если заменить внешнюю функцию **`modulo()`**, получающую адрес структуры, на метод **`modulo()`**, описанный внутри структуры, а вызов с явной передачей адреса — на вызов метода для объекта, то *на уровне машинного кода никаких изменений не произойдет*.

Функциям-членам при вызове их для объекта передаётся *неявный параметр* — адрес объекта, для которого функция вызывается. К этому параметру можно обратиться по имени **`this`**.

Пример: **`U02.cpp`**

`this` требуется, когда из метода нужно вызвать какую-либо функцию (обычную или метод другого объекта), аргументом которой должен стать как раз объект, для которого делается этот метод.

Защита. Конструктор.

В предыдущих примерах поля **`re`** и **`im`** (детали реализации) доступны из любого места в программе, где доступна сама структура **`str_complex`** и могут быть произвольно изменены.

Для того, чтобы скрыть детали реализации объекта, в C++ введен механизм *защиты*, который позволяет запретить доступ к некоторым частям структуры (полям и методам) из любых мест программы, кроме тел методов.

Для обеспечения защиты в язык введены ключевые слова **`public`** и **`private`**, которыми в описании структуры могут быть помечены поля и методы, доступные извне структуры (**`public:`**) и, наоборот, доступные только из тел функций-методов (**`private:`**).

Если описать поля **`re`** и **`im`** в секции **`private:`**, то явная установка их значений становится невозможной (пример **`U03.cpp`**).

Защита. Конструктор.

Можно добавить в структуру метод для установки значение полей в секции `public:`, например

```
public:
    void dset(double a_re, double a_im)
    {
        re=a_re;
        im=a_im;
    }
    double modulo() { return sqrt(re*re + im*im); }
```

(см. пример `U04.cpp`).

Bug: если перед использованием метода `modulo()` забыть вызвать метод `dset()`, то значения `re` и `im` будут неопределенными (случайными). *Объект создан, но не инициализирован.*

Fix: провести инициализацию (установку начальных значение полей) прямо в момент создания объекта.

Защита. Конструктор.

Конструктор объекта — это функция-член (метод) специального вида, которая может, как и обычная функция, иметь параметры или не иметь их.
Тело этой функции представляет собой порядок действий, которые необходимо выполнить всякий раз, когда создаётся объект описываемого типа.

Конструктор «объясняет» компилятору, как (в соответствии с какой инструкцией) создавать новый объект данного типа, какие параметры должны быть для этого заданы и как воспользоваться значениями этих параметров.

Компилятор отличает конструкторы от обычных методов по *имени*, которое совпадает с *именем описываемого типа* (в данном случае структуры).
Поскольку конструктор играет специальную роль и в явном виде не вызывается, *тип возвращаемого значения для конструктора указывать нельзя* (он не возвращает никаких значений, результатом работы конструктора является сам объект, для которого его вызвали).

Защита. Конструктор.

Если в структуре имеется конструктор, то описание переменных такого структурного типа уже не нужны. Каждая переменная (объект) создается вызовом конструктора.

Пример: **`U05.cpp`**

В C++ любая переменная создается с помощью конструктора. Во многих случаях компилятор считает конструктор существующим («неявно»), несмотря на то, что в программе конструктор не описан.

Поэтому возможны операторы **`int k=7;`** в середине тела модуля, т. к. это равносильно **`int k(7);`** (вызов конструктора для переменной типа **`int`**).

Описание структуры можно дополнить методами вывода вещественной и мнимой части, а также аргумента комплексного числа (пример **`U06.cpp`**).

Статические и динамические экземпляры класса

Если все поля должны быть доступны в процессе работы программы, рекомендуется использовать структуры (можно — с методами).

Если нужна защита — рекомендуется использовать классы.

Разница — в синтаксисе описаний переменных (экземпляров).

При создании объекта (экземпляра класса) можно использовать «статическое» или «динамическое» выделение памяти.

При «статическом» выделении памяти специально описывать переменную не нужно, она создается в результате работы конструктора при его вызове в любом месте модуля.

При «динамическом» выделении памяти нужно описать указатель на класс и создать экземпляр класса с помощью операции **`new`**.

Тогда для очистки динамической памяти после работы с объектом нужно использовать операцию **`delete`**.

См. пример **`U07.cpp`**.

Деструкторы

Конструкторы в C++ контролируют процесс создания (инициализации) объектов.

В C++ предусмотрены также ***деструкторы***, предназначенные для контроля над процессом уничтожения объекта.

Если объект в процессе своей деятельности захватывает некий ресурс (например, открывает файл, выделяет динамическую память и т. п.), причём об этом известно только самому объекту, т. е. на захваченный ресурс ссылаются только закрытые поля объекта, то о том, что объект что-то захватил, никто кроме него не знает! Тогда только сам объект может освободить всё, что он себе захватил, поэтому ему необходима возможность освободить захваченные ресурсы перед исчезновением, когда бы и по каким бы причинам это исчезновение ни случилось.

Деструктор — это метод класса, вызов которого автоматически вставляется компилятором в код в любой ситуации, когда объект прекращает существование.

Функция-деструктор имеет имя, представляющее собой имя описываемого типа (класса или структуры), к которому спереди добавлен знак ~ (тильда). Деструктор не возвращает никаких значений и не имеет параметров.

Примеры: `U08.cpp, U09.cpp`

Перегрузка имён функций

C++ позволяет в одной области видимости (модуле, структуре, классе) описать несколько различных функций, имеющих одно имя и различающиеся количеством и/или типами параметров.

Например:

```
void print(int n) { printf("%d\n", n); }
void print(char *s) { printf("%s\n", s); }
void print() { printf("Hello World!\n"); }
```

При обработке вызова функции компилятор определяет, какую функцию из имеющих одно имя необходимо вызвать в каждом конкретном случае, используя количество и типы фактических параметров.

Конструктор по умолчанию

Описать переменную типа **`Complex`** без параметров нельзя, т.к. для существующего конструктора класса **`Complex`** требуются два параметра.

А как тогда создавать массивы?

Компилятор считает конструктором метод класса (или структуры), имя которого совпадает с именем класса (или структуры).

Благодаря свойству перегрузки функций в C++ можно описать в одной области видимости несколько функций с одним и тем же именем; это относится и к конструкторам. Главное, чтобы функции, имеющие одинаковые имена (в данном случае — конструкторы), различались количеством и / или типом параметров.

Тогда можно добавить в класс **`Complex`** ещё один конструктор (см. пример **`U10.cpp`**).

и можно создать переменную без параметров

```
Complex z;
```

или массив объектов

```
Complex arr[50];
```

Динамические массивы объектов

При создании динамических массивов объектов также используется операция **new** и конструкторы по умолчанию.

Операция **new** имеет в этом случае «векторную» форму:

aaa = new Complex[n]; (n — количество объектов, может быть переменной)

Для освобождения памяти используется «векторная» форма операции **delete**:

delete [] aaa; (скобки должны быть пустыми!).

См. пример **U11.cpp**.

Конструктор преобразования

При создании программ на языках, обладающих типизацией, возникает потребность использовать значение одного типа там, где по смыслу предполагается значение другого типа. Простейший пример - арифметические операции для числа с плавающей точкой и целого числа, требующие *неявного преобразования типов*.

Конструктор преобразования — средство указания правил неявного преобразования для пользовательких типов данных (классов).

Задать правило преобразования значений типа A в значения типа B — это то же самое, что задать инструкцию по созданию объекта типа B по имеющемуся значению типа A.

Пример для объекта типа **`Complex`** — работа с частным случаем вещественного (целого) числа. См. пример **`U12.cpp`**.

При этом функция, работающая с объектами типа **`Complex`**, корректно обработает разные варианты своих аргументов.

Раскрытие области видимости. h-файлы.

Если методы класса содержат много кода, то трудно понять, что происходит в классе (описание реализации мешает понять назначение).

Если нужно использовать некоторый класс или структуру в нескольких модулях, следует поместить описание этого класса (структуры) в заголовочный файл и включить этот файл директивой **`#include "..."`** во все нужные модули.

Если при этом класс (структура) содержит тела методов, то код, составляющий тела методов, будет откомпилирован в каждом из модулей.

Если таких модулей много, в итоговом исполняемом файле окажется значительное количество дублируемого кода: методы класса (один и тот же код!) будут присутствовать в таком количестве копий, сколько модулей используют класс.

Раскрытие области видимости. h-файлы.

Обе проблемы (громоздкость описания класса и дублирование объектного кода методов) решаются вынесением тел методов за пределы описания класса (называемого также ***заголовком класса***). При этом в заголовке класса оставляется только прототип (заголовок) метода, после чего вместо тела метода ставится точка с запятой, как и после обычного прототипа функции. Тело функции-метода при этом описывается в другом месте, за пределами заголовка класса, возможно даже, что в другом файле (чаще всего это происходит, если заголовок класса вынесен в заголовочный файл; тела методов при этом описываются в файле реализации соответствующего модуля).

При описании метода за пределами заголовка класса необходимо указать компилятору, что речь идёт именно о методе определённого класса, а не о простой функции. Это делается с помощью символа ***раскрытия области видимости***, в роли которого в C++ выступает два двоеточия «`::`» («четвероточие»).

Описания заголовков классов и реализаций методов можно выносить в h-файлы (расширения `.h` или `.hpp`).

Пример: `U13.cpp` + защита от повторного включения

Переопределение стандартных операций

При работе с комплексными числами можно использовать операции (например, арифметические) обычными образом, но для этого их нужно переопределить в соответствующем классе (пример **`U014.cpp`**).

Получаются функции с названиями

- **`operator+`**
- **`operator-`**
- **`operator*`**
- **`operator/`**

(**`operator`** — ключевое слово в C++).

Но в применении к данным типа **`Complex`** это можно записать как обычные арифметические операции, причем для данных типа **`int`** (**`double`**) знаки арифметических операций будут иметь прежний смысл (*компилятор различает функции по типам аргументов, см. также про конструктор преобразования*).

Объект как параметр функции

При передаче объекта (экземпляра класса) в функцию ***по имени*** в качестве параметра создается копия объекта в области памяти для функции (передача по значению).

При этом при изменении значений полей исходного объекта в функции с объектом вне функции ничего не происходит, если в объекте нет полей с динамическим выделением памяти (пример `U15.cpp`).

Если объект содержит поля с динамической памятью, то после работы функции могут возникнуть проблемы (пример `U16.cpp` — ошибка времени выполнения *double free*). Причина — передается указатель на массив, при завершении функции работает деструктор и указатель теряется, но в процессе передачи управления функция пытается еще раз освободить объект.

Выход: передать объект в функцию по ссылке, тем самым дублируя и динамический массив.

Ссылки в C++

Ссылки в C++ - особый вид объектов данных, хранящие адреса переменных, но в отличие от указателей, ссылки эквивалентны тем переменным, для которых они созданы.

Любые операции со ссылкой будут на самом деле производиться с переменной, для которой создана ссылка.

Ссылку невозможно изменить, ее значение (переменная, на которую она сделана) устанавливается в момент создания ссылки и не меняется в течение «срока существования» ссылки.

Например

```
int i;
int *p=&i; // указатель на i
int &r=i; // ссылка на i (синоним имени переменной)
// …
i++; // увеличение i на 1
(*p)++; // то же самое через указатель
r++; // то же самое по ссылке
```

Конструктор копирования

Конструктор копирования — инструкция компилятору, как создавать копии объектов.

В примере класса с динамическим массивом конструктор копирования формирует копию массива в ссылке на объект. Такую ссылку можно передать как параметр функции и ее использование не вызовет ошибки *double free* (пример **`U17.cpp`**), т. к. динамический массив тоже копируется и функция работает с копией всех полей.

Конструктор копирования имеет один параметр — ссылку на объект описываемого типа. Ссылку обычно снабжают модификатором **`const`**, подчеркивая, что она не должна изменяться в течение существования объекта.

Специальные виды конструкторов

В зависимости от набора и значений параметров компилятор «придает смысл» конструкторам:

- Конструктор ***без параметров*** воспринимается как ***конструктор по умолчанию***
- Конструктор ***с параметрами, отличающимися по количеству и типу*** от описываемого типа, воспринимается как ***конструктор преобразования***
- Конструктор ***с одним параметром, имеющим тип «ссылка на описываемый объект»***, воспринимается как ***конструктор копирования***.

Значения параметров по умолчанию

В C++ при описании прототипа функции можно для всех или некоторых ее параметров задать значения по умолчанию (для ***формальных параметров***), тогда при вызове функции можно указать меньше ***фактических параметров***. *Недостающие элементы компилятор подставит сам*.

Пример:

```
void f(int i=3, char *b="string", int c=5);
```

Тогда возможны вызовы:

```
f(5, "HELLO!", 10);
f(5, "HELLO!"); // то же что и f(5, "HELLO!", 5);
f(5); // то же что и f(5, "string", 5);
f(); // то же что и f(3, "string", 5);
```

Для функции может быть задано любое количество параметров по умолчанию, но *все параметры, следующие за первым параметром по умолчанию, должны иметь значения по умолчанию*.

Параметры по умолчанию в конструкторе

Наличие возможность использовать параметры по умолчанию позволяет уменьшить количество описаний конструкторов при описании класса.

Для класса **Complex** можно описать один конструктор

```
Complex(double a_re=0, double a_im=0)
  {
      re=a_re;
      im=a_im;
  }
```

Он является и обычным конструктором для двух аргументов, и конструктором по умолчанию, и конструктором преобразования (пример **U18.cpp**).

Еще про переопределение символов операций

Ранее рассматривалась возможность переопределения арифметических операций для комплексных чисел.

C++ позволяет переопределять и другие операции:

- Присваивание
- Присваивание, совмещенное с действиями (`+=`, `-=`, `|=`, `<<=` и т.п.)
- Инкремент и декремент
- Индексирование
- «Стрелка» (`->`)

Подробности см. А.В.Столяров «Введение в C++», 5-е издание.

Неявные конструкторы

Неявный конструктор — конструктор, который генерируется компилятором автоматически при отсутствии соответствующего конструктора в коде класса или структуры.

Компилятор C++ неявно генерирует два вида конструкторов: конструктор по умолчанию и конструктор копирования.

Конструктор копирования неявно генерируется для любого класса или структуры, в которых он не описан. По умолчанию поля, имеющие конструкторы копирования, их и используют, остальные поля копируются побитово.

Конструктор по умолчанию генерируется неявно, ***только если*** в структуре (классе) ***не описан ни один конструктор***.