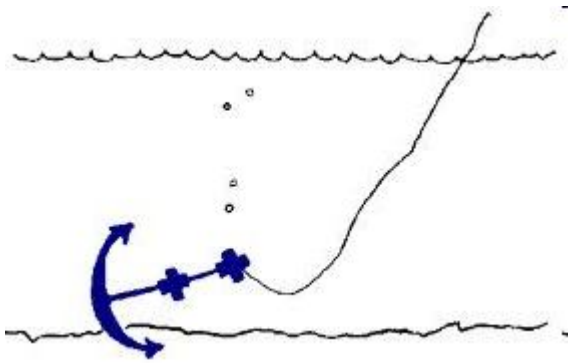
Карпов В.Э.

Проблемы ООП



"Попинаем" C++

- Диссертация Б. Страуструпа, программа на яз. Симула.
- Программа была написана очень быстро и легко.
- Скорость работы этой программы крайне мала (говорят, ее не хватало даже для того, чтобы насчитать необходимые данные к моменту защиты диссертации).

  =>
- Желание создать свой язык, который бы совмещал в себе высокую скорость разработки (от Симулы) и высокую скорость выполнения (от C).

Цели нового языка

- Язык должен быть объектно-ориентированным.
- Язык должен быть эффективным.
- Язык должен быть таким, чтобы программисты на C могли легко на него перейти.
- Компилятор должен находить все возможные ошибки

Технология создания первого компилятора C++:

- На языке C был написан препроцессор, преобразующий подмножество нового языка в обычный язык C.
- На подмножестве C++ и был разработан первый транслятор с C++ и затем «раскручен» в C-код, пригодный к компиляции в исполняемую программу.
- Целевой машиной для первого C++ транслятора была любая машина, способная выполнять C-программу (результатом трансляции был C-код).

«Исторические» претензии к C++

- "C++ **мутирует** чуть ли не каждый квартал, постепенно превращаясь в игрушку-трансформер, а сам автор C++ вынужден выпускать одну книжку за другой для толкования своего детища."
- "С внедрением языка C++ в **коммерческую разработку резко** упала надежность программного обеспечения."
- "Чрезмерная **избыточность** языка C++ привела к тому, что один и тот же алгоритм каждый программист кодирует по-своему и не самым оптимальным способом."
- "C++ - просто бескрайнее море перегруженных операторов."

Объекты

- *"C++ - не "настоящий" объектный язык"*

Безуспешные попытки построения единой иерархии классов с общим базовым классом Object (идея Smalltalk).

В C++ тщательно продуманные иерархии библиотек классов оказывались негибкими, а работа классов — неочевидной. Для того чтобы библиотеками классов можно было пользоваться, их приходилось поставлять в **исходных текстах**.

Шаблоны и объем кода

Появление шаблонных классов поставило крест на этом направлении развития:

- Наследованием стали пользоваться только в тех случаях, когда требовалось порождение специализированной версии имеющегося класса.
- Библиотеки стали составляться из отдельных классов и небольших несвязанных друг с другом иерархий.

  =>
- Стало снижаться повторное использование кода, т.к. в C++ невозможно полиморфное использование классов из независимых иерархий.
- Повсеместное применение шаблонов ведет к недопустимому росту объема скомпилированного кода (шаблоны реализуются методами **макрогенерации**).

Проблемы двоичной совместимости

Один из тяжелейших недостатков C++, унаследованный им от синтаксиса C, состоит в доступности компилятору описания внутренней структуры всех использованных классов.

- => Изменение внутренней структуры представления какого-нибудь библиотечного класса приводит к необходимости перекомпиляции всех программ, где эта библиотека используется.
- => Это сильно ограничивает разработчиков библиотек в части их модернизации (выпуская новую версию, они должны сохранять двоичную совместимость с предыдущей).
- => C++ "в чистом виде" не пригоден для ведения больших проектов.

C++ по-прежнему ненадежен

```
1.  void main(void)
2.  {
3.    int len = 10;
4.    char *pSource = "abcd";
5.    char c;
6.    int n = 10;
7.    switch (len % 8)
8.    {
9.      case 0:
10.     {
11.       do
12.       {
13.         c = *pSource++;
14.         case 7: c = *pSource++;
15.         case 1: c = *pSource++;
16.       }
17.       while (--n > 0);
18.     }
19.   }
20. }
```

Непредсказуемость кода

- Проблема "скрытого кода"

  void f (const std::string& s);

При вызове

f ("Hello, world!")

будет создан объект класса std::string, он передается функции, а потом уничтожается. Т.е. вполне корректный и правильный код несет в себе скрытые затраты на создание и уничтожение объекта.

- Старые "претензии" к ссылкам:.

  void func1 (A& a);

  void func2 (A a);

Проблемы объектной парадигмы

- Основа ООП:

«Всё суть объекты» подразумевает универсальность ООП.

Однако могут ли существовать иные подходы?

"Эры" программирования

- кодирование в машинных кодах;
- ассемблеры, интерпретирующие программы и ранние компиляторы;
- императивное, процедурное и функциональное программирование (языки, ориентированные на компиляторы);
- объектно-ориентированное программирование;
- ???

Особенность всех эр – "монолитные" программы; ориентация на программы, работающие на одном компьютере.

Особенности ближайшей перспективы

Тенденции

- Возрастающая сложность систем
- Переход к распределенным вычислениям
- Гетерогенные вычислительные комплексы

Вывод:

- Вычислительные системы будут состоять из множества различных по программной и аппаратной платформ компонент (ОС, приложения, архитектуры, рецепторы и эффекторы).

Следствия:

- Устойчивость (надежность)
- Гибкость (реконфигурация)
- Адаптируемость (самоорганизация).

ООП

Изначально декларировался принципиальный динамизм ООП:

- "Программы" выполнялись непосредственно в среде программирования (Смолток).
- Модификация поведения объектов (и программы) непосредственно в ходе выполнения.
- Парадигма **интерпретируемой** системы

Дальнейшее развитие (регресс):

- Возврат на статическую (компилируемую) платформу.
- Программа и среда программирования отделяются друг от друга.
- Эффективность кода приносится в жертву гибкости.

Причины:

- Коммерческие соображения;
- Отсутствие теоретического базиса (надежные системы из ненадежных элементов; принципы самоорганизации и т.п.);
- "Проклятие" фон-неймановской архитектуры.

Некоторые признаки вырождения ООП

- Сложившийся ОО поход неадекватен требованиям вычислительных процессов будущего.
- ОО языки потеряли простоту в угоду мнимому удобству программирования.
- Сплошь и рядом нарушаются и обходятся базовые концепции ООП: инкапсуляция, защита данных, строгое наследование
- На первый план выходят шаблоны – суррогаты "истинных" динамических объектов.
- Как следствие, концепция открытого исходного кода лучше справляется с подобной ситуацией. Похоже, что только модульность — разбиение на составные части так, чтобы люди могли их понять — вот что действительно важно в инкапсулировании.
- Объекты не решили проблему повторного использования кодов. Классы поставляются в виде исходных кодов.
- Динамизм объектов свелся к компиляции статического кода.

Прототипное программирование

Прототипное программирование — стиль ООП, при котором отсутствует понятие класса, а повторное использование (наследование) производится путём клонирования существующего экземпляра объекта — **прототипа**.

Примеры прототип-ориентированных языков:

- Self
- JavaScript,
- Lua
- Io
- REBOL и др.

ПП и ООП

ООП. Все объекты разделены на два основных типа — *классы* и *экземпляры*.

- *Класс* определяет структуру и функциональность (*поведение*).
- *Экземпляр* - носитель данных, т.е. обладает *состоянием*, меняющимся в соответствии с поведением, заданным классом.

ПП. Языки, основанные на классах, приводят к излишней концентрации на **таксономии** классов и на **отношениях** между ними.

- Прототипирование заостряет внимание на поведении некоторого (небольшого) количества «образцов», которые затем классифицируются как «базовые» объекты и используются для создания других объектов.
- Многие прототип-ориентированные системы поддерживают изменение прототипов во время выполнения программы.

Язык Lua

Lua (*«луна»*) — интерпретируемый язык программирования, Католический университет Рио-де-Жанейро (*Computer Graphics Technology Group of Pontifical Catholic University of Rio de Janeiro in Brazil*).

Интерпретатор является свободно распространяемым, с открытыми исходными текстами на языке Си.

- LucasArts. Игровой движок GrimE.
- World of Warcraft.
- Описание уровней игры-головоломки Enigma.
- S.T.A.L.K.E.R.

Lua. Некоторые примеры

```
-- Функция возвращает несколько значений
function swap(a, b)
  return b, a
end

x, y = swap(x, y)
-- или так:
x, y = y, x

-- Игнорирование ненужного:
a, _, _, d = f()
```

```
do
  -- Сохраняем текущую функцию print
     как oldprint
  local oldprint = print
  -- Переопределяем функцию print
  function print(s)
    if s == "foo" then
      oldprint("bar")
    else
       oldprint(s)
    end
  end
end
```

Lua. Объекты = функции + таблицы

```
lamp = {  on = false }

-- Функции для работы со структурой
function turn_on(l)
  l.on = true
end

function turn_off(l)
  l.on = false
end

turn_on(lamp)
turn_off(lamp)
```

```
lamp = {
  on = false
  turn_on = function(l) l.on = true end
  turn_off = function(l) l.on = false end
}
lamp.turn_on(lamp)
lamp.turn_off(lamp)
```

```
lamp:turn_on() – эквивалентные формы записи
lamp.turn_on(lamp)
lamp["turn_on"](lamp)
```

```
lamp = { on = false }

-- '.': аргумент надо указывать
function lamp.turn_on(l) l.on = true end

-- ':' аргумент неявно задается как переменная "self"
function lamp:turn_off() self.on = false end
```

Lua. Наследование

```
superlamp = {   brightness = 100 }
-- указываем родительскую таблицу
setmetatable(superlamp, lamp)
-- методы родителя доступны
superlamp:turn_on()
superlamp:turn_off()
```

Вывод

Революционная ситуация созрела:

- Верхи (ОО-системы) не могут
- Низы (программисты) не хотят

"Наши цели ясны, задачи определены. За работу, товарищи!"

Из выступления на 22 съезде КПСС (1962) Первого секретаря ЦК КПСС Никиты Сергеевича Хрущева (1894—1971).

(Бурные, продолжительные аплодисменты, переходящие в овацию. Все встают.)