Higher School of Economics
National Research University
School of Software Engineering



# Процедурно-параметрический полиморфизм и его интеграция с языком программирования C

*А.И. Легалов*
*П.В. Косов*

# Парадигмы программирования

1. Методы алгоритмизации (МА)
   - императивное программирование, функциональное программирование, автоматное программирование, декларативное программирование
2. ***Методы композиции (МК)***
   - абстрактные типы данных + функции, классы = данные + методы, модули, пространства имен
3. Методы задания однозначности (МЗО)
   - статическая типизация (однозначность на уровне описания типов данных), динамическая типизация (однозначность на уровне вычисляемых тегов), бестиповая (операционная однозначность, на уровне кодов операций компьютера)
4. Стратегии управления вычислениями (МУВ)
   - последовательное программирование, параллельное программирование (разнообразие вариантов: 27 стратегий в архитектурах ВС, 8 стратегий в языках программирования)
5. Уровни абстракции (УА)
   - Непосредственное отображение, Абстракция типов, Метапрограммирование
6. Модели памяти (МП)
   - Методы выделения, Однородность, Организация, Использование

...

**Парадигмы = |МА| * |МК| * |МЗО| * |СУВ| * |УА| * |МП| * …**

# Языки программирования: смесь парадигм

***Язык программирования — инструмент предоставляющий поддержку зафиксированного в нем подмножества парадигм программирования по каждому из ортогональных критериев***

**Язык программирования =**

**{ма ⊆ МА}, {мк ⊆ МК}, {мзо ⊆ МЗО},**
**{мув ⊆ МУВ}, {уа ⊆ УА},...**

***Монопарадигменный язык — средство в большей степени теоретического изучения***

# Пример. Язык программирования C

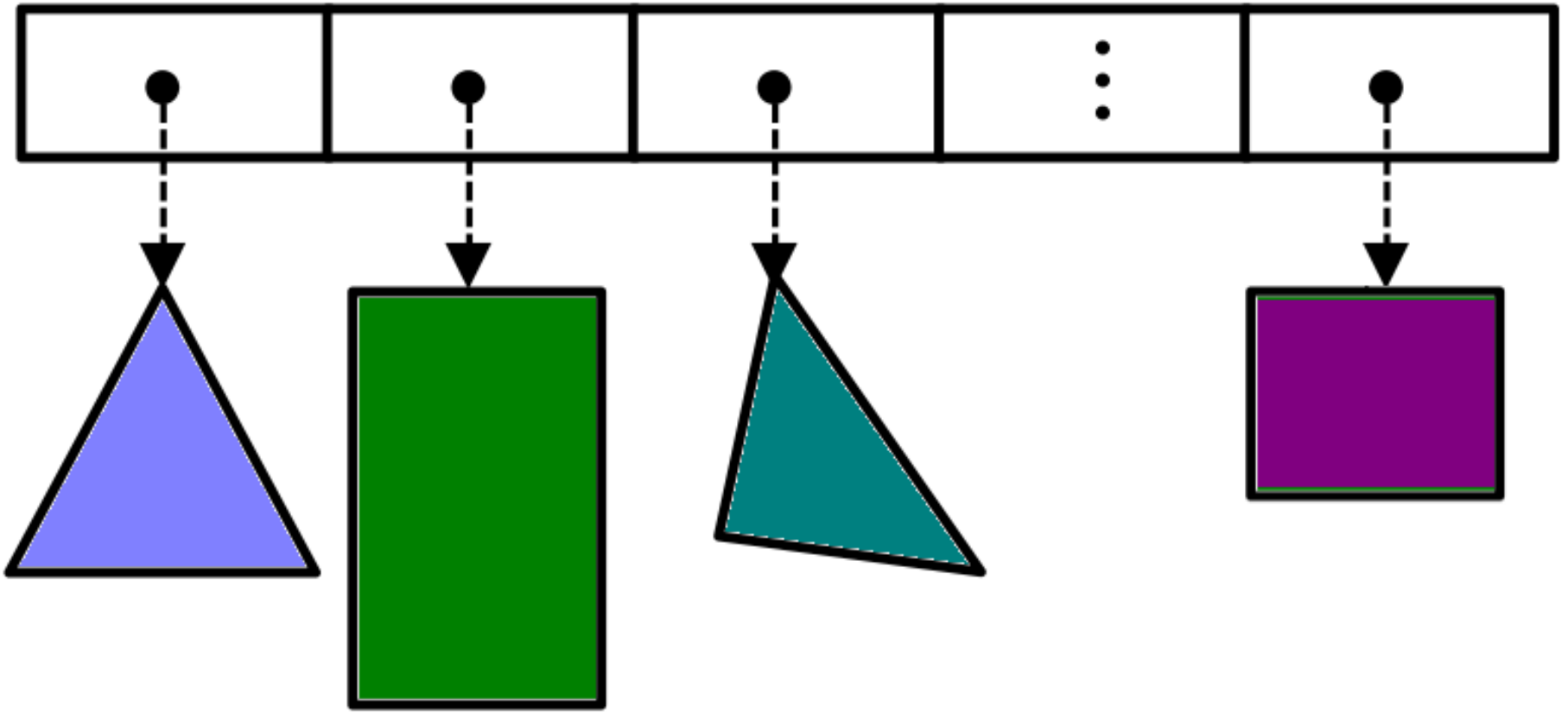
МА: {Императивный, Частично функциональный в выражениях}
МК: {Функции, АТД, Единицы компиляции, Заголовочный интерфейс}
МЗО: {Статическая типизация, Бестиповой (void*)}
СУВ: {Последовательная, директивная}
УА: {Прямое отображение, Препроцессор, Абстракции (АТД + Функции}
МП: {
{Глобальная, Локализованная, Автоматическая, Динамическая},
{Общая, регистры}, {Линейная}, {Многократная}
}

# Парадигмы = |МА| * |МК| * |МЗО| * |СУВ| * |УА| * |МП| * …

| *Методы композиции* | *Методы алгоритмизации* | | | |
|---|---|---|---|---|
| | *Императивный* | *Функциональный* | *...* | *Функционально-потоковый* |
| *Объектно-ориентированный стиль (классы)* | Java, C#, ... | ... | ... | |
| *Процедурный стиль (АТД + функции)* | C++, Python, ... | Ocaml | ... | |
| | C, Modula-2, Oberon-7, ... | Haskell, ML, ... | ... | |
| *Стиль Go (Структуры + интерфейсы, типажи)* | Go, Rust | ... | ... | |
| *...* | ... | ... | ... | |
| *Процедурно-параметрический* | ! | + | ... | ! |

# Инопланетное вторжение

# Инопланетное вторжение



# Инопланетное вторжение

```
typedef struct Rectangle {
  int x, y; // ширина, высота
} Rectangle;

typedef struct Triangle {
  // стороны треугольника
  int a, b, c;
} Triangle;

typedef struct Figure {
  key k; // ключ
  union {
    Rectangle r;
    Triangle t;
  };
} Figure;
```

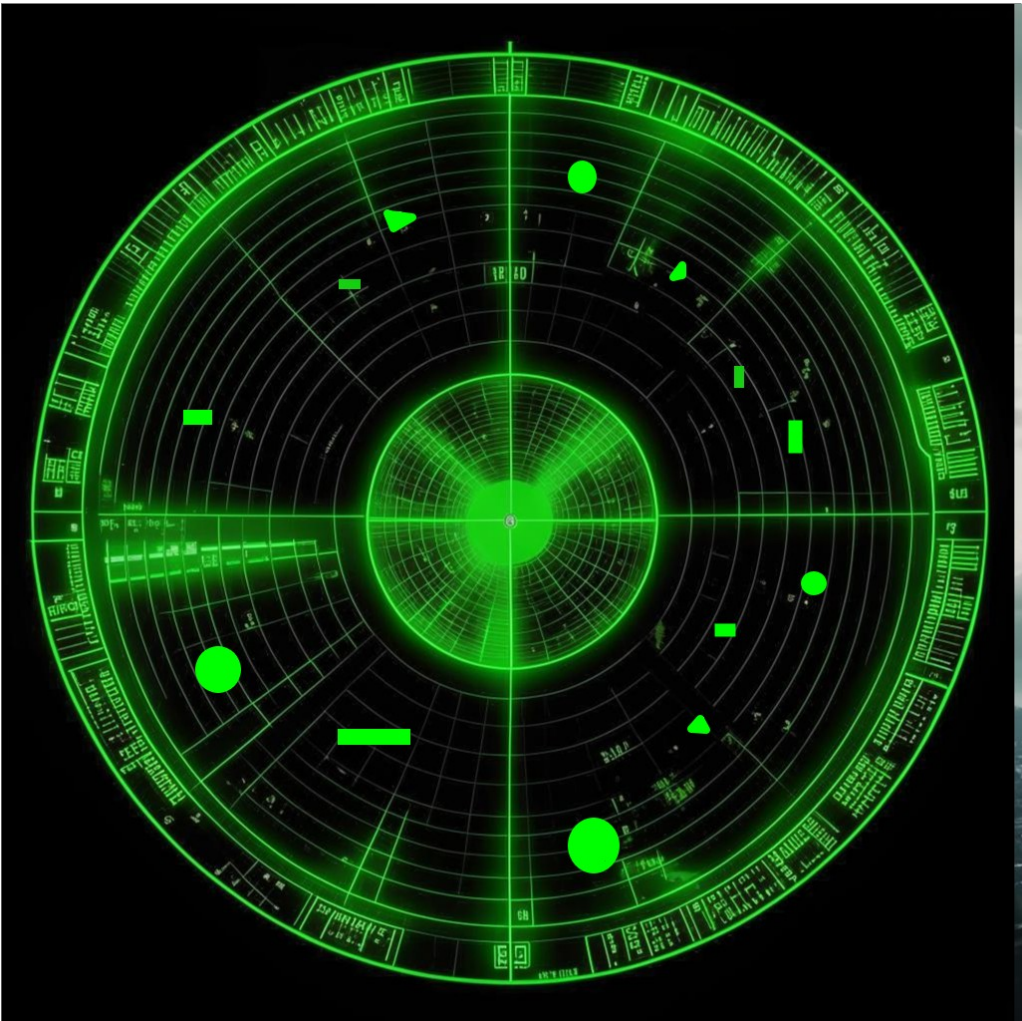
```
class Figure {
public:
  ...
  virtual void Out
   (std::ofstream &ofst) = 0;
  ...
};

class Rectangle: publicFigure {
  double x, y; // ширина, высота
public:
  ...
  virtual void Out
    (std::ofstream &ofst);
  ...
};

class Triangle ...
```

# Инопланетное вторжение

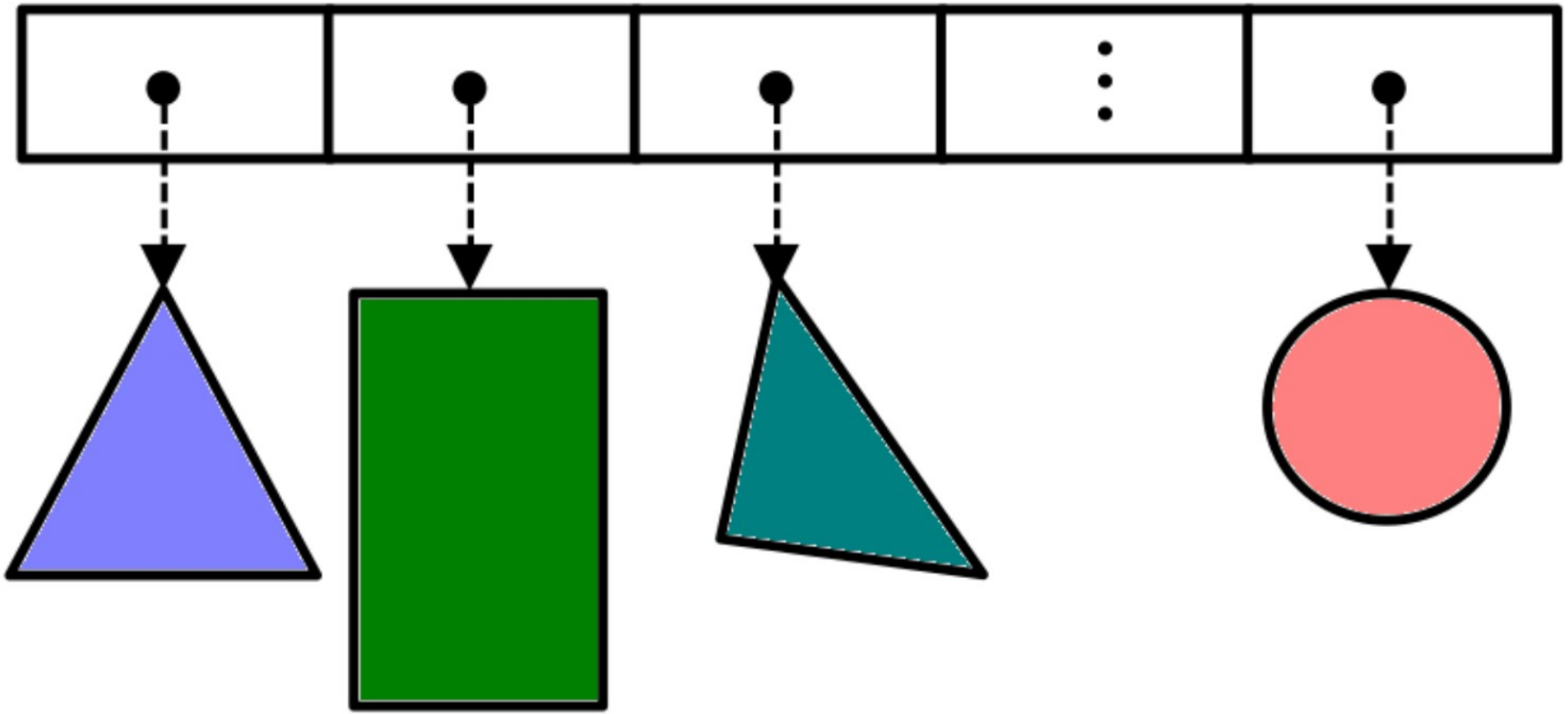
```
void FigureOut(Figure *s, FILE*
ofst)
{
  switch(s->k) {
  case RECTANGLE:
    RectangleOut(&(s->r), ofst);
    break;
  case TRIANGLE:
    TriangleOut(&(s->t), ofst);
    break;
  default:
    fprintf(ofst, "Incorrect
figure!\n");
  }
}
```

```
// Вывод параметров прямоугольника
void Rectangle::Out
   (std::ofstream &ofst) {
  ofst <<
    "It is Simple Rectangle: x = "
     << x << ", y = " << y << "\n";
}

// Вывод параметров треугольника
void Triangle::Out
  (std::ofstream &ofst) {
  ofst <<
     "It is Simple Triangle: a = "
      << a << ", b = " << b
      << ", c = " << c << "\n";
}
```

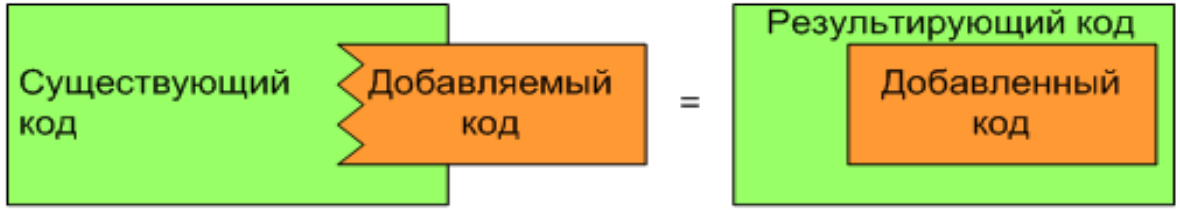
# Инопланетное вторжение



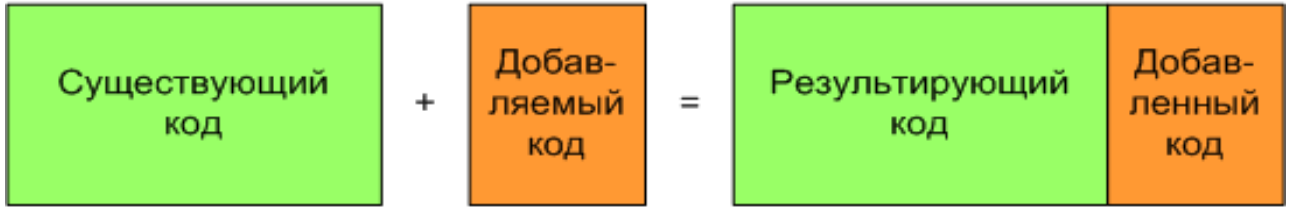
# Инопланетное вторжение



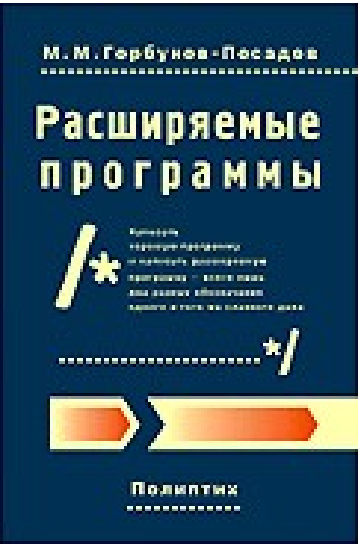
# Общая Специфика эволюционного расширения




а) Изменение существующего кода



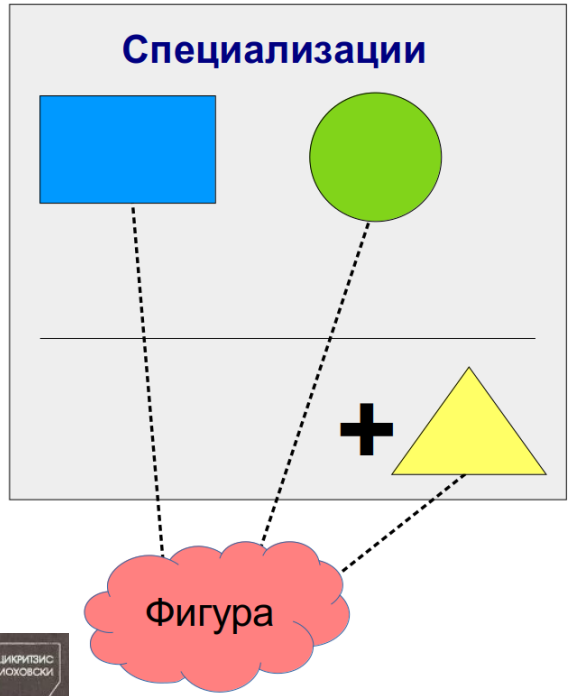

б) Эволюционное расширение


Михаил Михайлович Горбунов-Посадов
РАСШИРЯЕМЫЕ ПРОГРАММЫ
ООО «Полиптих», Москва. 1999.

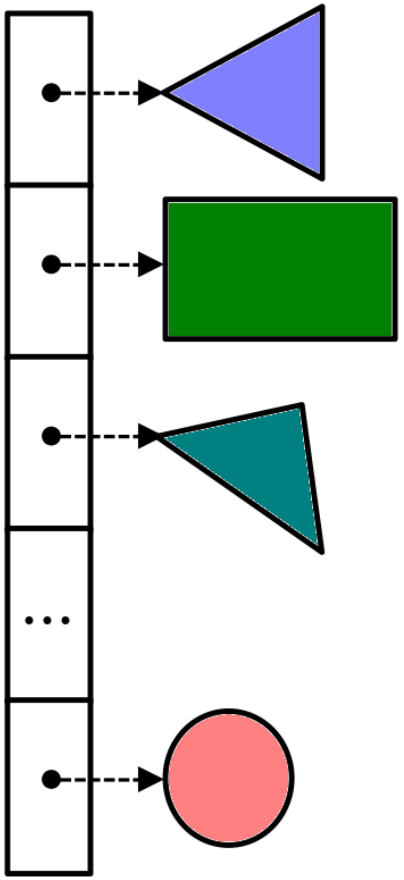
Фуксман А.Л. Технологические аспекты создания программных систем. — М.: Статистика, 1979. — 184 с.

# Специфика использования обобщений (динамическое связывание)




Обобщение

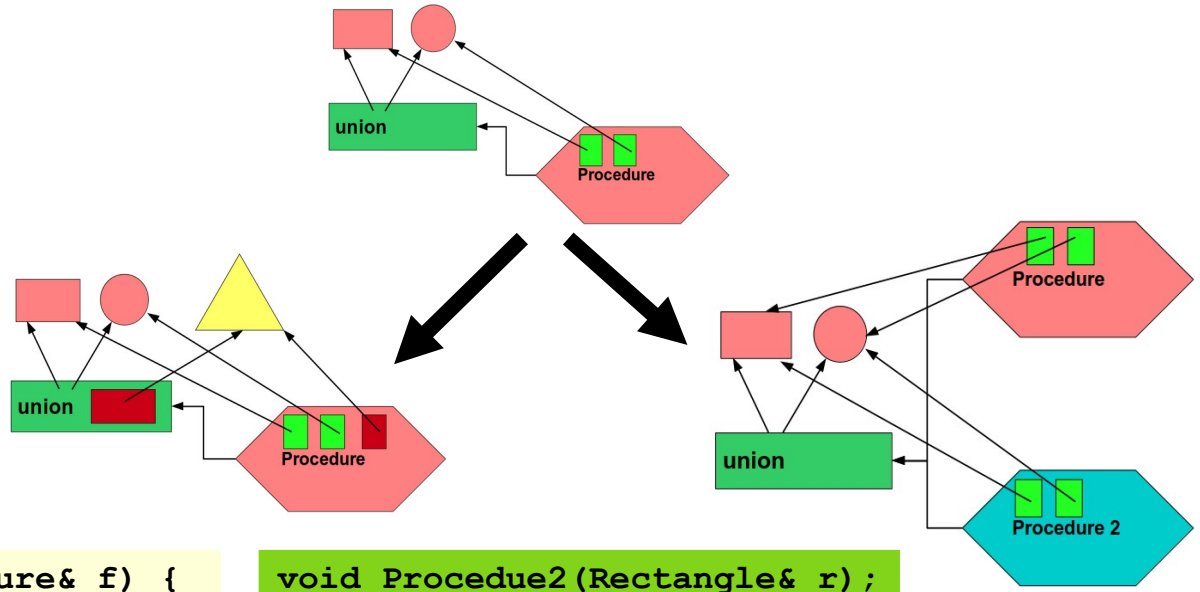

Обобщающие процедуры




Цикритзис Д., Лоховски Ф. Модели данных.
Пер. с англ. - М.: Финансы и статистика,
1985. - 344 с.

# Процедурное программирование

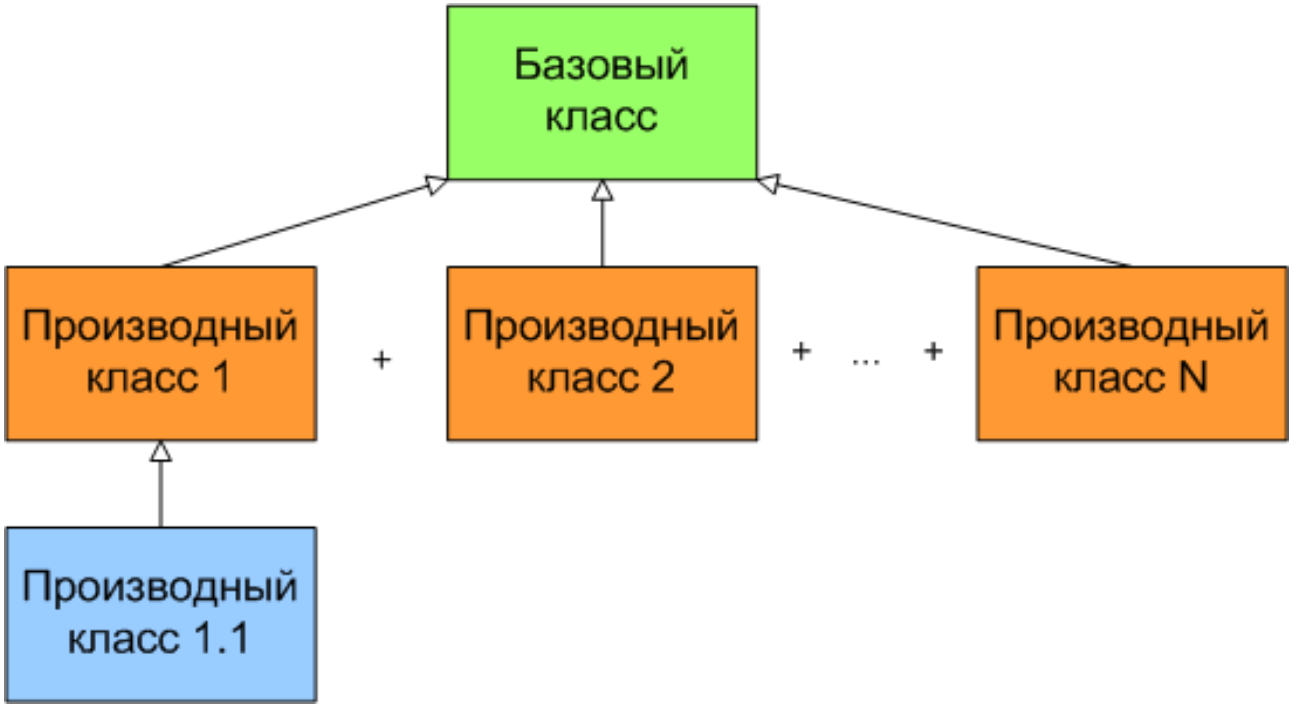
```
struct Rectangle { int x; int y; };
struct Circle { int r; };
struct Triangle {
     int a; int b; int c;
};
void Procedue(Rectangle& r);
void Procedue(Circle& r);
void Procedue(Triangle& r);
enum key {
    rectangle, circle, triangle
};

struct Figure {
     key k;
     union {
          Rectangle r;
          Circle c;
          Triangle t;
     };
};
```




```
void Procedure(Figure& f) {
     switch(key) {
     case rectangle: P
          Procedure(f.r);
     break;
     case circle:
          Procedure(f.c);
     break;
     case triangle:
          Procedure(f.t);
     break;
     }
}
```

```
void Procedue2(Rectangle& r);
void Procedue2(Circle& r);

void Procedure2(Figure& f) {
     switch(key) {
     case rectangle: P
          Procedure2(f.r);
     break;
     case circle:
          Procedure2(f.c);
     break;
     }
}
```

# Объектно-ориентированное программирование




- Добавление новых специализаций без изменения кода.
- Модификация классов при добавлении методов.
- Прямое использование специализаций в других обобщениях.

```
// Мультиметод
// Проявление множественного полиморфизма
bool Multimethod(Figure& first, Figure& second) {
    ...
}
```

**Процедурный подход:**

- вложенные if или switch

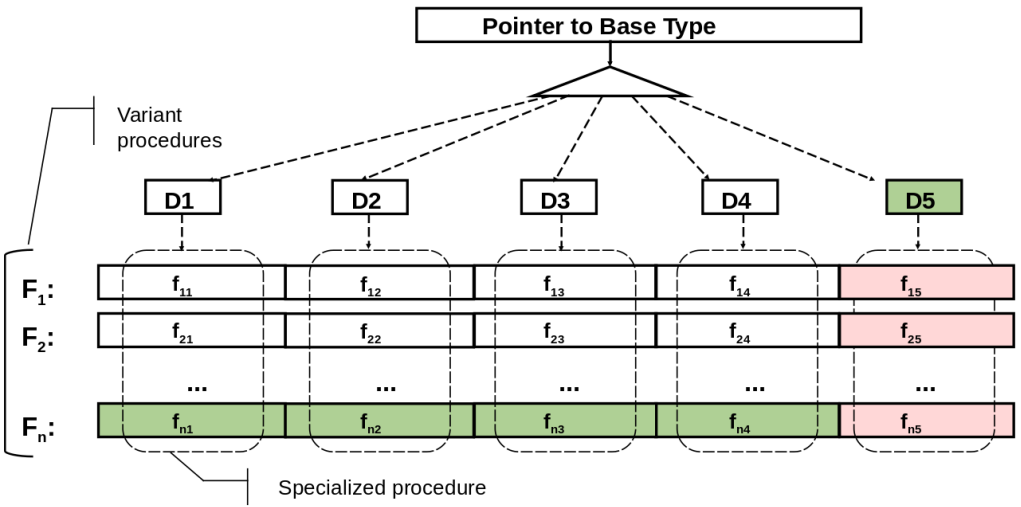
**Объектно-ориентированный подход:**

- диспетчеризация

**В обоих случаях при расширении мультиметода изменение уже написанного кода!!!**

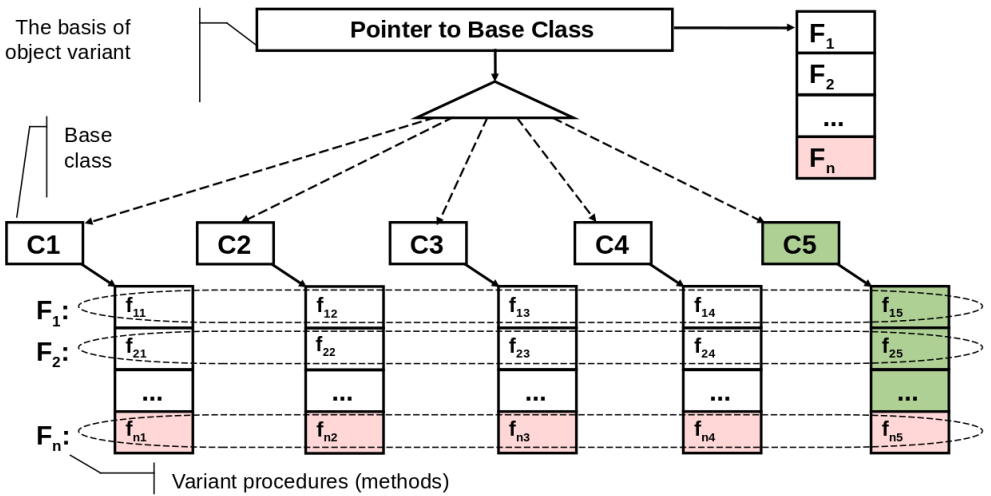
## Процедурный стиль F(X)

# F(X)
# F(X, Y, Z)



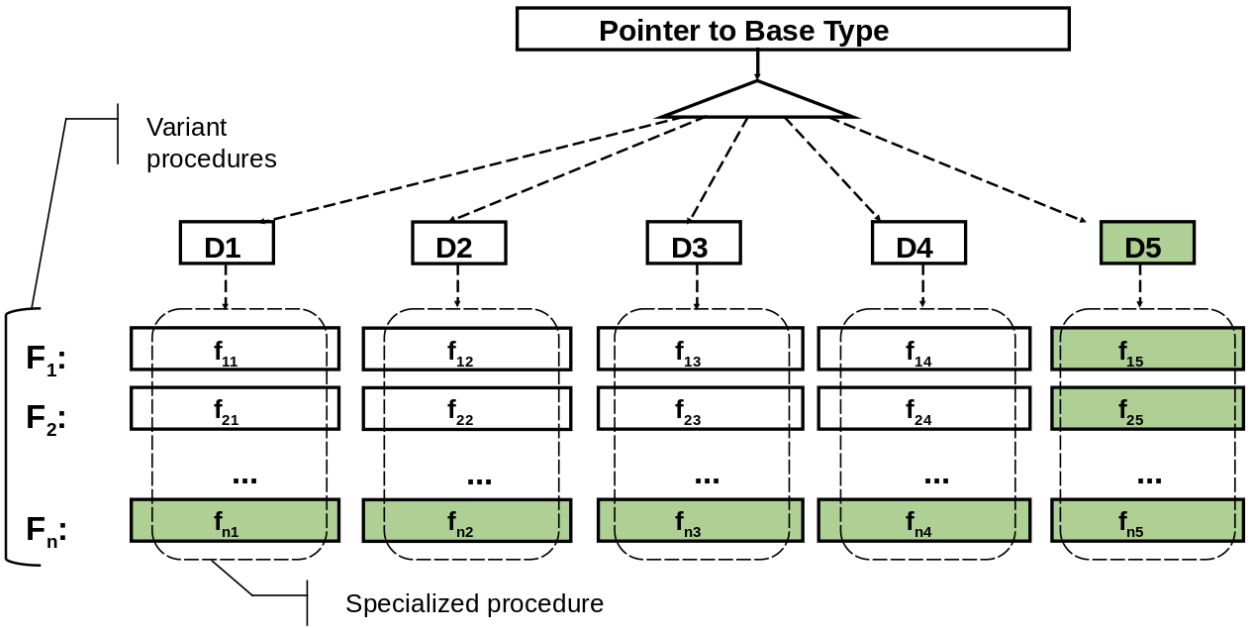

## Объектно-ориентированный стиль

# X.F()
# X.F(Y,Z)



# Процедурно-параметрический стиль

# F<X>()
# F<X, Y, Z>()
# F<X, Y>(Z)



# Процедурно-параметрическая парадигма

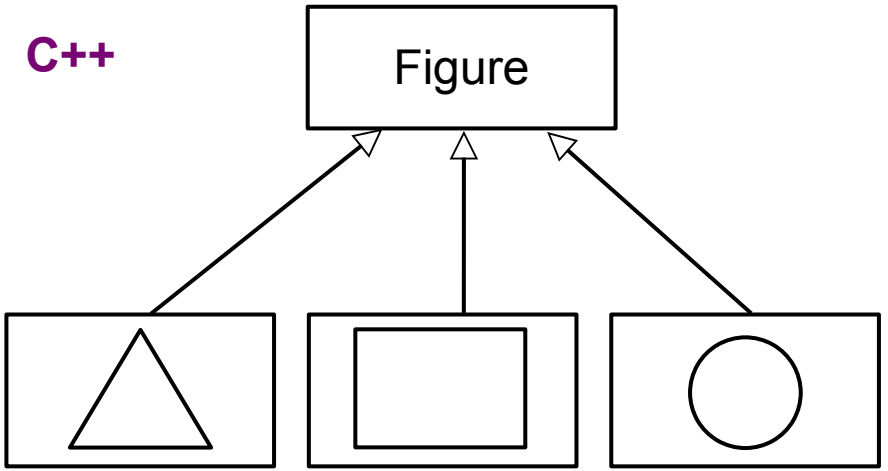
**В отличие от концепции базового типа, расширяемого за счет добавления в производных типах, использование параметрического обобщения предполагает сохранение исходного типа, а альтернативные специализации вводятся как его уточнения.**

**Внешне все специализации имеют единый тип, а их разнообразное толкование используется только внутри него. Это позволяет убрать глобальную идентификацию типов, применяемую при расширении записей или наследовании, и обеспечивает поддержку концепции статической типизации.**

**==> Возможность обращения к обработчика через массивы (одномерные, многомерные)**
**==> Возможность преобразований в мономорфный монолит (if, switch)**
**==> Возможность расширения обобщений и создания новых обобщений от уже существующих обобщений (расширение вверх и вниз)**

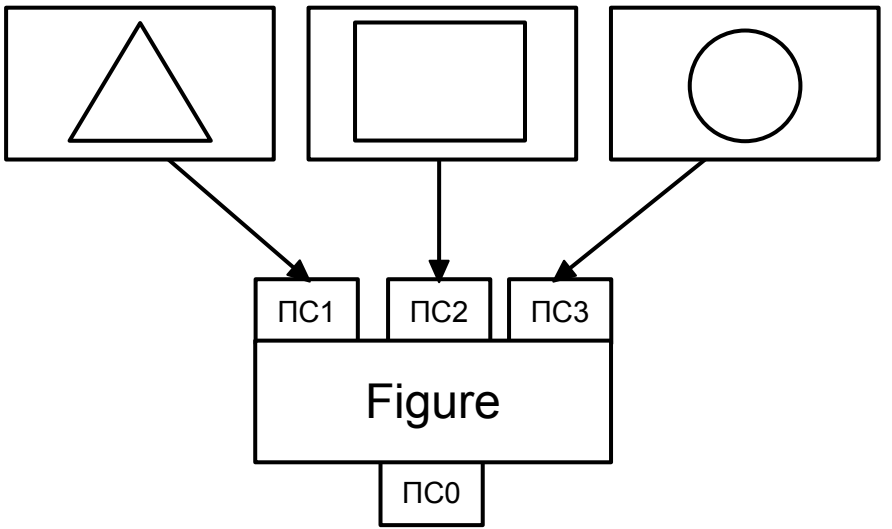
# ООП vs ППП

C++




```
struct Figure {};
struct Triangle: Figure {int a, b, c;};
struct Rectangle: Figure {int x, y;};
struct Circle: Figure {int r;};
```

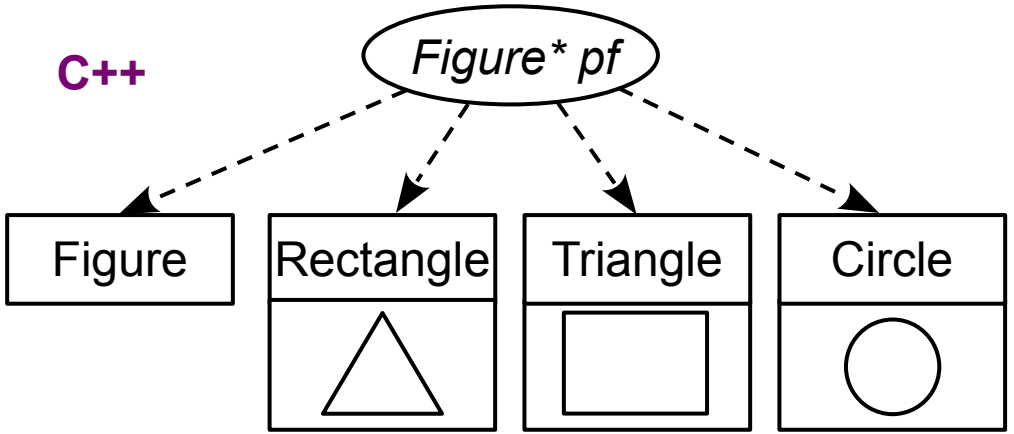
P2C




```
typedef struct Triangle {int a, b, c;} Triangle;
typedef struct Rectangle {int x, y;} Rectangle;
typedef struct Circle {int r;} Circle;
Typedef struct Figure {}<Triangle; Rectangle;> Figure;
Figure + <Circle;>;
```

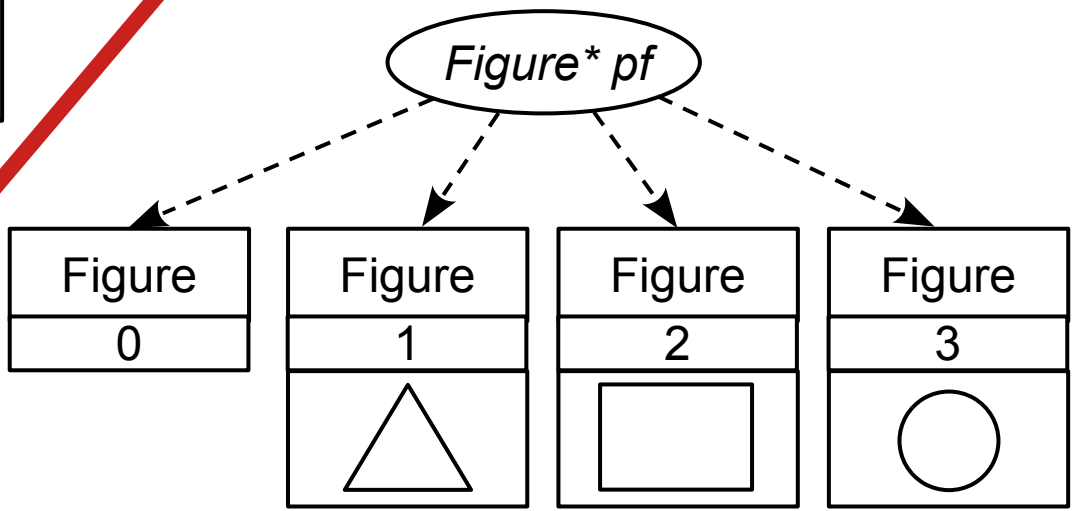
# ООП vs ППП

**C++**




```
Figure *pf = new Rectangle;
pf = new Triangle;
pf = new Circle;
```

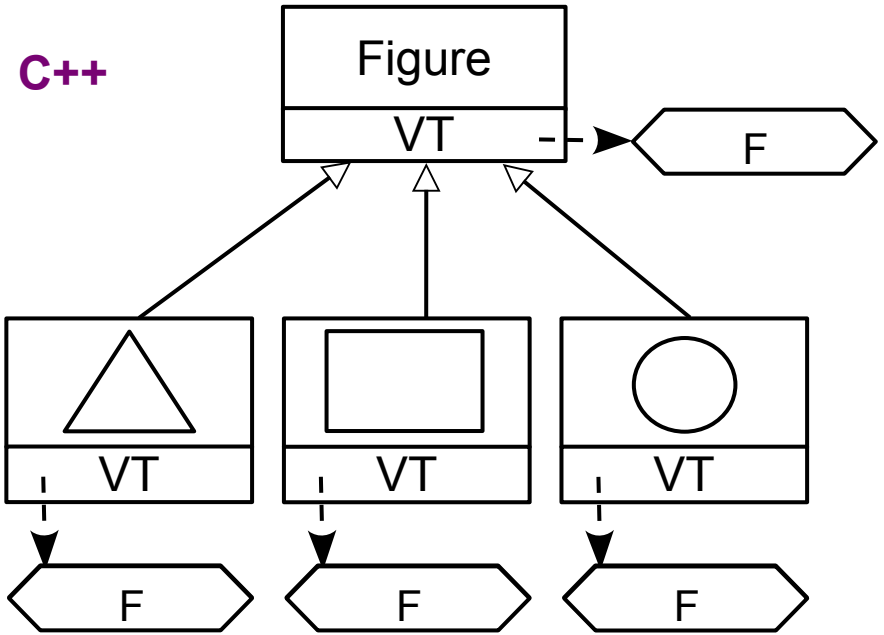
**P2C**




```
Figure *pf = create_spec Figure<Rectangle>;
Figure *pf = create_spec Figure<Triangle>;
Figure *pf = create_spec Figure<Circle>;
```

# Косвенная адресация --- Индексация
## Декоратор --- Стратегия

**C++**

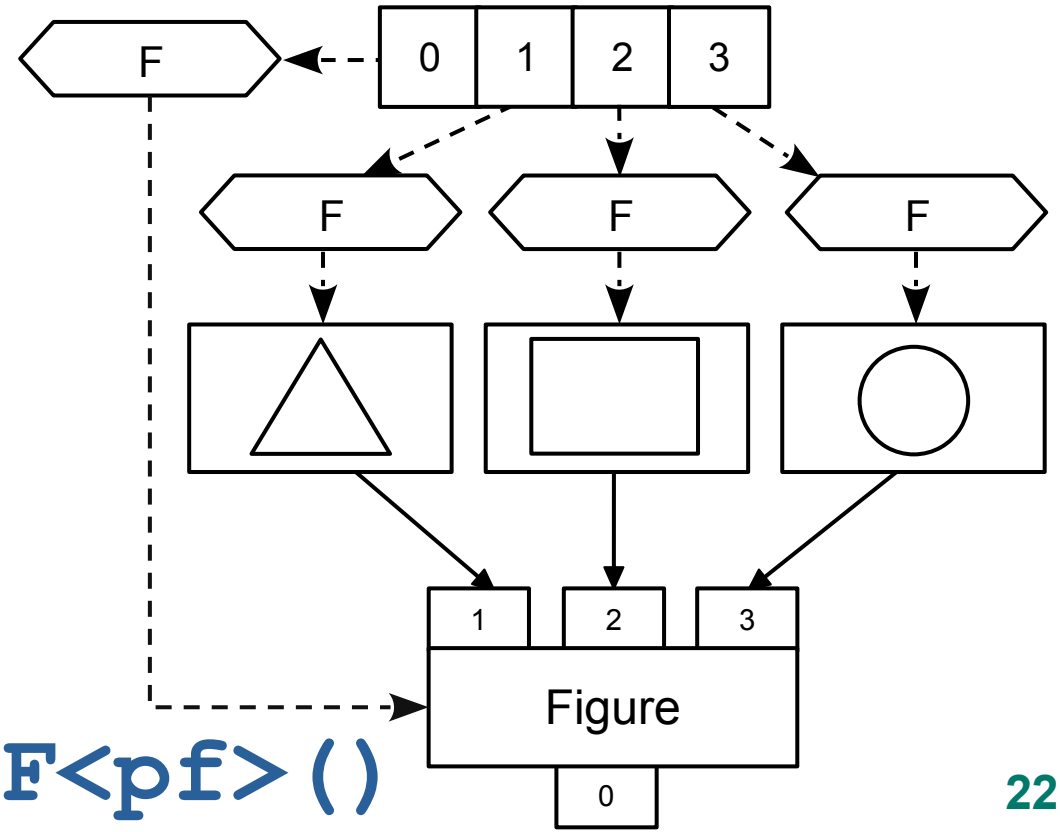



```
struct Figure{... virtual F()= 0;}
struct Rectangle{... virtual F();};
struct Triangle{... virtual F();};
struct Circle{... virtual F();};
```

**pf->F()**

**P2C**

```
F<Figure *pf>();
F<Figure<Rectangle> *pr>();
F<Figure<Triangle> *pt>();
F<Figure<Circle> *pt>();
```




**F<pf>()**

# Обобщенная структура и ее расширение

*ОбобщеннаяСтруктура = Структура "<" СписокСпециализаций ">" ["const"]*
*| "<" [ШаблонПризнака] ">".*
*Шаблон признака = ":".*
*СписокСпециализаций = СписокСпециализацийПоТипу*
*| СписокСпециализацийПоПризнаку.*
*СписокСпециализацийПоТипу = ИмяТипа ";" {ИмяТипа ";"}.*
*СписокСпециализацийПоПризнаку = Признак {"," Признак} ":" Тип ";"*
*{Признак {"," Признак} ":" Тип ";"}.*
*Признак = идентификатор.*
*Тип = ИмяТипа | ОписаниеНеименованногоТипа.*

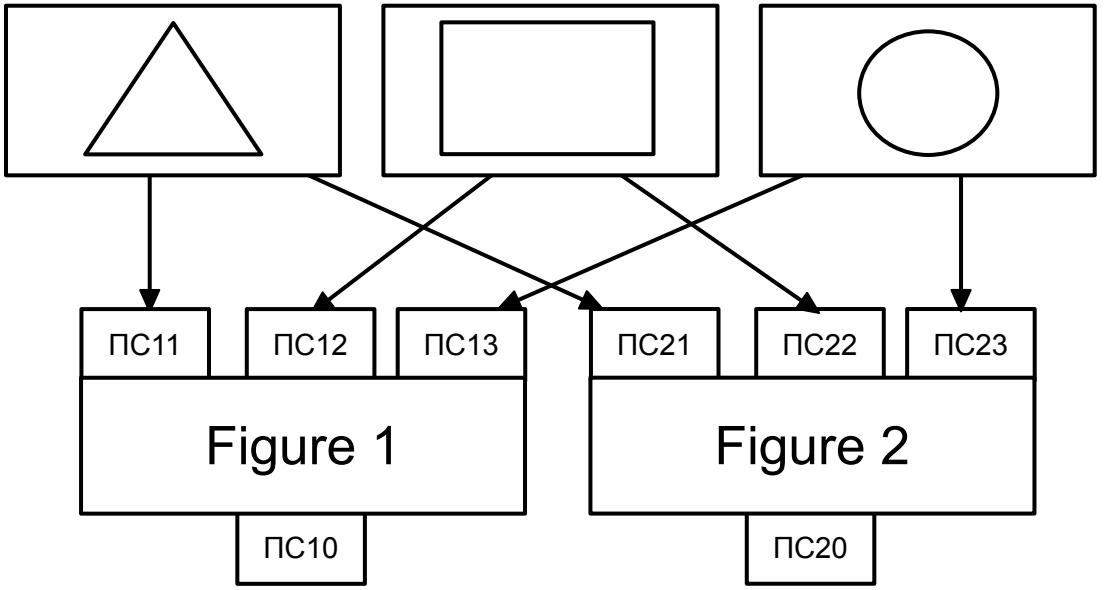
```
struct Triangle {int a, b, c;};    // Треугольник
struct Rectangle {int x, y;};      // Прямоугольник
struct Circle {int r;};            // Круг

struct Figure {}<struct Triangle; struct Rectangle; struct Circle;>;
```

# Обобщенная структура и ее расширение

*ДобавлениеСпециализаций = ССылкаНаОбобщеннуюСтруктуру*
*"+" "<" СписокСпециализаций ">".*

```
struct Figure {}<struct Triangle;>;
struct Figure + <struct Rectangle; struct Circle;>;

// В обобщенной структуре отсутствуют специализации
struct Figure {}<>;
// Включение треугольника в обобщенную фигуру
struct Figure + <struct Triangle;>;
```

# Обобщенная структура и ее расширение




```
typedef struct Figure1 {}<> Figure1;
Figure1 + <Rectangle;>;
Figure1 + <Triangle;>;
Figure1 + <Circle;>;
```

```
typedef struct Figure2 {}<:> Figure2;
Figure2 + <rect: Rectangle;>;
Figure2 + <tran: Triangle;>;
Figure2 + <circ: Circle;>;
```

**Возможность создания нескольких обобщений и их специализаций от одних и тех же основ специализаций**

# Обобщенная структура и ее расширение

```
typedef struct Triangle {int a, b, c;} Triangle;    // Треугольник
typedef struct Rectangle {int x, y;} Rectangle;     // Прямоугольник
typedef struct Circle {int r;} Circle;              // Круг

typedef struct Figure2 {}<:> Figure2;

// Добавление прямоугольника, ромба, отрезка
Figure2 + <rect, rhomb: Rectangle; section: Circle;>;
// Добавление треугольника и круга
Figure2 + <trian: Triangle; circ: Circle;>;


// Дни недели
struct WeekDay {int week_number;}
    <Monday, Tuesday, Wednesday, Thursday,
     Friday, Saturday, Sunday: void;> const;
```

# Экземпляры параметрических обобщений

*ОписаниеСпециализированнойПеременнной = ССылкаНаОбобщеннуюСтруктуру*
*"<" [ ИмяТипа | Признак ] ">" СписокСпециализированныхПеременных.*
*СписокСпециализированныхПеременных =*
*СпециализированнаяПеременная {"," СпециализированнаяПеременная}.*
*СпециализированнаяПеременная =*
*{"*"} ИмяСпециализированнойПеременной {"[" Размерность "]"} ["=" Инициализатор].*
*ССылкаНаОбобщеннуюСтруктуру = struct ИмяОбобщенной структуры | ИмяTypedef.*
*ОписаниеУказателейНаОбобщения = ТипОбобщения СписокОбобщенныхУказателей.*
*СписокОбобщенныхУказателей = ОбобщенныйУказатель {"," ОбобщенныйУказатель}.*
*ОбобщенныйУказатель =*
*"*"{"*"} ИмяОбобщенногоУказателя {"[" Размерность "]"}["=" Инициализатор].*

```
struct Figure<struct Triangle> v1 = {}<{3,4,5}>, v2;
struct Figure<struct Circle> c1 = {}<{10}>, *pc1 = &c1, c[10];
Figure2<rect> r1, r2[3] = {{}<{1,2}>,{}<{3,4}>,{}<{5,6}>},
              r3 = <{7,4}>, pr1 = &r1;
struct Figure *pf1 = &t1, **ppf1 = &pf1; // Указатели на обобщение
struct WeekDay<Monday> monday = {35}<>;  // Задается номер недели
```

# Рекурсивное расширение специализаций

```
typedef struct T {int x, y;}<:> T;
T + <t0: _Bool; t1: struct{double r; char s;}>;

typedef struct T0 {int z;}<:> T0;
struct T + <t2: T0;>;  // T{int x,y;}<t0:...; t1:...; t2:T0>

// Потенциал для расширения
typedef struct T00 {int a;}<:> T00;
T0 + <t00: T00;>; // T0{int a;}<t00:...;>

T<t0>, T<t1>, T<t2>, T<t2<t00>>, …

// Рекурсивное расширение
struct T + <t3: T>;

// Декорирование на этапе компиляции
T<t3>, T<t3<t3>>, T<t3<t3<t3<t3<t2<t00>>>>>>, ...
```

# Операции над специализированными переменными

```
struct Figure<struct Triangle> v1 = {}<{3,4,5}>, v2;
v1.@a = 5;
v2 = v1;
pf1 = pc1;

struct WeekDay<Monday> monday = {35}<>;  // Задается номер недели
monday.week_number = 24; // Понедельник 24-й недели


struct T<t0> b = {}<1>;  // Инициализация специализации базового типа
b.@ = 0;   // Изменение значения специализации базового типа
```

# Обобщающие параметрические функции

*ОбобщающаяФункция = ТипВозвращаемогоПараметра ИмяФункции*
*"<" СписокОбобщенныхПараметров ">" "(" [ СписокФормальныхПараметров ] ")"*
*ТелоОбобщающейФункции.*

*СписокОбобщенныхПараметров = ОбобщенныйПараметр {"," ОбобщенныйПараметр }.*
*ТелоОбобщающейФункции = ПустоеТело | ОбработчикПоУмолчанию.*

*ПустоеТело = "=" "0".*

*ОбобщенныйПараметр = ТипОбобщения "*" ИмяОбобщения.*

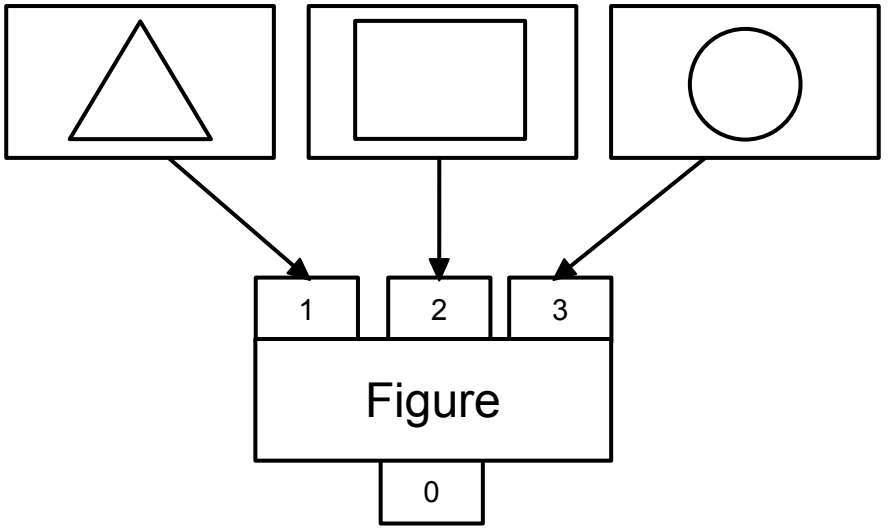
```
void PrintFigure<struct Figure *f>() = 0;

void PrintFigure<struct Figure *f>() {
    printf("Incorrect Argument!!!\n")
    exit(-1);
}
```

# Обработчики параметрических специализаций

*СпециализированнаяФункция = ТипВозвращаемогоПараметра ИмяФункции*
*"<" СписокСпециализаций ">" "(" СписокФормальных параметров ")"*
*ТелоСпециализированнойФункции.*
*СписокСпециализаций = СпециализированныйПараметр*
*{"," СпециализированныйПараметр }.*
*СпециализированныйПараметр = ТипСпециализации "*" ИмяСпециализации.*

```
// Вывод параметров прямоугольника
void PrintFigure<struct Figure<struct Rectangle> *r>() {
    printf("Rectangle: x = %d, y = %d", r->@x, r->@y);
}
// Вывод параметров треугольника
void PrintFigure<struct Figure<struct Triangle> *t>() {
    printf("Triangle: a = %d, b = %d, c = %d",
                (*t).@a, (*t).@b, (*t).@c);
}
// Вывод параметров круга
void PrintFigure<struct Figure<struct Circle> *r>() {
    printf("Circle: r = %d", c->@r);
}
```

# Эволюционная поддержка множественного полиморфизма




| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | F00 | F01 | F02 | F03 |
| 1 | F10 | F11 | F12 | F13 |
| 2 | F20 | F21 | F22 | F23 |
| 3 | F30 | F31 | F32 | F30 |

Формируются многомерные таблицы для обработчиков специализаций

# Эволюционная поддержка множественного полиморфизма

```
typedef struct Figure3 {}<:> Figure3;
// Добавление прямоугольника и треугольника
Figure3 + <rect: Rectangle; trian: Triangle;>;
// Обобщенная функция, требующая обязательного переопределения
bool Multimethod<Figure3 *first, Figure3 *second>() = 0;

// Прямоугольник разместится внутри прямоугольника
bool Multimethod<Figure3<rect> *r1, Figure3<rect> *r2>() {
    return ((r1->@x < r2->@x) && (r1->@y < r2->@y)) ||
           ((r1->@x < r2->@y) && (r1->@y> < r2->@x));
}
// Прямоугольник разместится внутри треугольника
bool Multimethod<Figure3<rect> *r1, Figure3<trian> *t2>() {...}
// Треугольник разместится внутри прямоугольника
bool Multimethod<Figure3<trian> *t1, Figure3<rect> *r2>() {...}
// Треугольник разместится внутри треугольника
bool Multimethod<Figure3<trian> *t1, Figure3<trian> *t2>() {...}
```

# Эволюционная поддержка множественного полиморфизма

```
// Добавление круга (позднее)
Figure3 + <circ: Circle;>;
// Прототип. Там, где не видно определения
bool Multimethod<Figure3*, Figure3*>();

// Прямоугольник разместится внутри круга
bool Multimethod<Figure3<rect> r1*, Figure3<circ> *c2>() {
    return ((r1->@x*r1->@x + r1->@y*r1->@y) < (c2->@r*c2->@r));
}
// Треугольник разместится внутри круга
bool Multimethod<Figure3<trian> *t1, Figure3<circ> *c2>() {...}
// Круг разместится внутри прямоугольника
bool Multimethod<Figure3<circ> *c1, Figure3<rect> *r2>() {...}
// Круг разместится внутри треугольника
bool Multimethod<Figure3<circ> *c1, Figure3<trian> *t2>() {...}
// Круг разместится внутри круга
bool Multimethod<Figure3<circ> *c1, Figure3<circ> *c2>() {
    return c1->@r < c2->@r;
}
```

# Реализация динамического полиморфизма

1) Изменение компилятора clang. Трансформация абстрактного синтаксического дерева в процессе компиляции программы. Вводимые конструкции заменяются на конструкции языка С, обеспечивающие их реализацию. Это обеспечивает дальнейшую безболезненную генерацию кода (в llvm)

2) Формирование необходимых связей  до запуска функции main. Обеспечивается за счет возможностей операционной системы Linux, использовать аналоги конструкторов, выполняемых до запуска main, и деструкторов, запускаемых после ее завершения

*Легалов А.И., Косов П.В.*
*Расширение языка С для поддержки процедурно-параметрического полиморфизма. - Моделирование и анализ информационных систем.*
*2023;30(1):40-62. https://doi.org/10.18255/1818-1015-2023-1-40-62.*

***Проект:*** *https://github.com/kpdev/llvm-project/tree/pp-extension-v2*

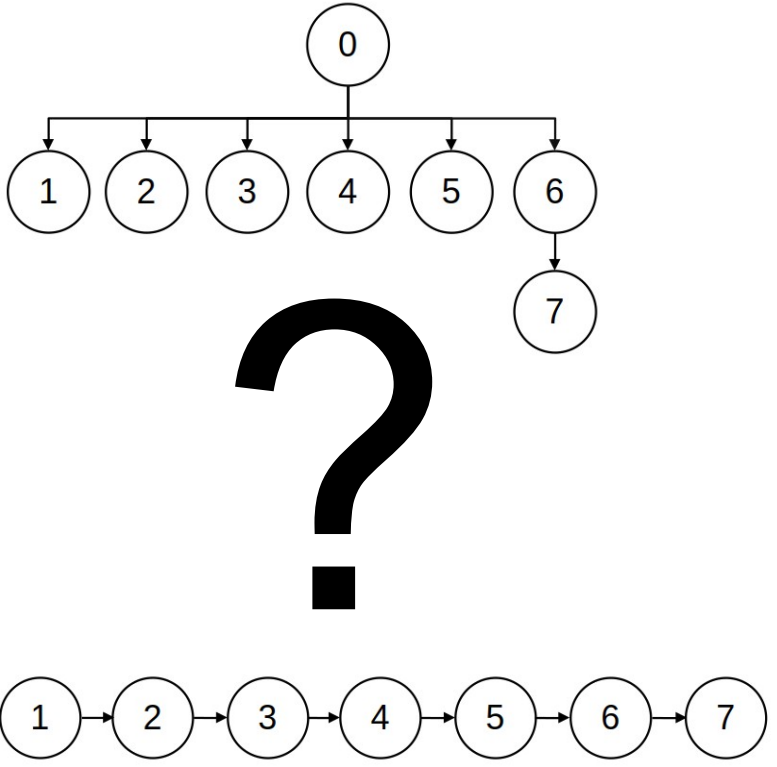
# Пример формирования обобщения и специализации

```
typedef struct figure {
    unsigned tag;        // тег, задающий признак специализации
    struct {} head;      // первичная структура, возможен список полей
} figure;

static unsigned _figure_number_ = 0;

typedef struct figure_rectangle {
    // Повторение полей обобщения
    unsigned tag;   // поле тега на том же месте
    struct {} head; // поле структуры, идентичное структуре обобщения
    struct rectangle tail; // поле, определяющее содержание специализации
} figure_rectangle;

// Регистрация очередной специализации
void _figure_spec_register(void (*set_spec_tag)(unsigned));
// Получение текущего числа специализаций
unsigned get_figure_number();
// Возврат признака специализации через указатель на обобщение
unsigned get_figure_tag(figure *fig);
```

# Пример использования конструкторов и деструкторов

```
// Конструктор, обеспечивающий создание массива указателей
// на обработчики специализаций
void __attribute__ ((constructor(201))) register_print_figure_array() {
  unsigned figure_number = get_figure_number();
  // тестовый вывод конструктора
  printf("%s: array size = %u\n", __FUNCTION__, figure_number);
  _print_figure_array =
  malloc(figure_number * sizeof(print_figure_pointer));
  for(unsigned i = 0; i < figure_number; ++i) {
    _print_figure_array[i] = print_figure_default;
  }
}

// Деструктор, обеспечивающий создание памяти, выделенной
// под массив указателей на обработчики специализаций
void __attribute__ ((destructor(201))) delete_print_figure_array() {
  // тестовый вывод деструктора
  printf("%s\n", __FUNCTION__);
  free(_print_figure_array);
}
```

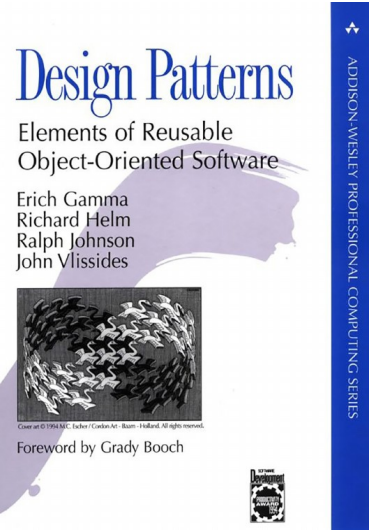
# Эволюционное расширение с поддержкой динамического связывания

| Ситуация | Подходы | | | |
|---|---|---|---|---|
| | Процедурный | ОО | Go-поход | ПП |
| 1 Добавление специализации и ее обработчиков |  | | | |
| 2 Добавление процедур с дополнительной функциональностью | | | | |
| 3 Добавление новых полей в существующий тип | | | | |
| 4 Добавление новых процедур для обработки конкретных специализаций внутри существующих обобщений | | | | |
| 5 Создание обобщения на основе существующих специализаций | | | | |
| 6 Добавление в программу мультиметода | | | | |
| 7 Изменение мультиметода при добавлении специализации | | | | |

# Эволюционное расширение с поддержкой динамического связывания

| Ситуация | Подходы | | | |
|---|---|---|---|---|
| | Процедурный | ОО | Go-поход | ПП |
| 1 Добавление специализации и ее обработчиков | *нет* | *есть* | *есть* | *есть* |
| 2 Добавление процедур с дополнительной функциональностью | *есть* | *нет* | *есть* | *есть* |
| 3 Добавление новых полей в существующий тип | *косвенное, для расширяемых типов* | *косвенное, при наличии динамической проверки типов во время выполнения* | *нет* | *косвенное, при использовании обобщенной записи* |
| 4 Добавление новых процедур для обработки конкретных специализаций внутри существующих обобщений | *есть* | *нет* | *есть* | *есть процедурный и параметрический* |
| 5 Создание обобщения на основе существующих специализаций | *есть* | *косвенное* | *есть* | *есть* |
| 6 Добавление в программу мультиметода | *есть* | *нет* | *есть* | *есть* |
| 7 Изменение мультиметода при добавлении специализации | *нет* | *нет* | *нет* | *есть* |

http://www.softcraft.ru/ppp/simplesituations/





# *Паттерны проектирования*

Дополнительные возможности по сравнению с ОО паттернами проектирования:

- Эволюционное расширение функциональности с поддержкой полиморфизма для уже сформированных структур за счет обобщающих функций.
- Другие подходы к реализации паттернов, обеспечивающие разнообразные специализированные варианты (мультиметод, монолитные декораторы).
- Ряд паттернов уже оказались реализованными при реализации эволюционно расширяемых программ (декоратор, посетитель)

# Повышение надежности программирования

**Позволяет избавиться от разыменования (на примере разыменования типов в контейнере и его ненадежного восстановления)**

```
typedef struct FigRectangle { Key k; Rectangle r; } FigRectangle;
typedef struct FigTriangle { Key k; Triangle t; } FigTriangle;
typedef struct FigCircle { Key k; Circle c; } FigCircle;

void PrintFigure(void *f) {
  if((FigRectangle*)f->k==RECTANGLE) {
    printf("Rectangle: x = %d, y = %d",
        (FigRectangle*)f->r.x, (FigRectangle*)f->r.y);
  } else if((FigTriangle*)f->k==TRIANGLE) {
    printf("Triangle: a = %d, b = %d, c = %d",
          (FigTriangle*)f->t.a, (FigTriangle*)f->t.b, (FigTriangle*)f->t.c);
  } else if((FigCircle*)f->k==CIRCLE) {
    printf("Circle: r = %d",
          (FigCircle*)f->c.r);
  } else {
    printf("Incorrect Figure!!!\n");
  }
}
```

# Повышение надежности программирования

**Позволяет сформировать надежную оболочку вокруг ненадежного кода (на примере pthread)**

```
int pthread_create( pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine) (void *),
    void *arg );

int pthread_join( pthread_t thread, void **retval );
```

# Повышение надежности программирования

```
class Thread {
public:
  virtual ~Thread () {}
  virtual void run () = 0;
  int start () {
    return pthread_create(
      &_ThreadId, nullptr,
      Thread::thread_func, this );
  }
  int wait () {
    return pthread_join( _ThreadId, NULL );
  }
protected:
  pthread_t _ThreadId;
  static void* thread_func(void* d) {
    (static_cast <Thread*>(d))->run();
    return nullptr;
  }
};
```

```
typedef struct ThreadData
    {pthread_t threadId;}
<> ThreadData;

void RunThread <ThreadData* d>() = 0;

void* ThreadFunc(void* d) {
  RunThread<(ThreadData*) d>();
  return NULL;
}

int StartThread(ThreadData* td) {
  return pthread_create(&(td->threadId),
                NULL, ThreadFunc, td);
}

int WaitThread(ThreadData* td) {
  return pthread_join(td->threadId, NULL);
}
```

# Повышение надежности программирования

```
class ThreadRect : public Thread {
public:
  ThreadRect (int xx, int yy):
                  x{xx}, y{yy} {}

  virtual void run () {
    p = double(2*(x+y));
  }

  void print(const char* str) {
    std::cout << "Perimeter of "
    << str << " = " << p << "\n";
  }

protected:
  int x;
  int y;
  double p;
};
```

```
typedef struct Rectangle{int x, y;} Rectangle;
typedef struct RectPerimeter
        {Rectangle r; double p;} RectPerimeter;

ThreadData + <RectPerimeter;>;

void RunThread <ThreadData<RectPerimeter> *rp>() {
  rp->p = (double)((rp->r.x+rp.r->y)*2);
}

void PrintRectPerimeter(
    RectPerimeter* rp, const char* str)
{
  printf("Perimeter of %s = %f", str, rp->p);
}

// Специализация для потока
ThreadData + <RectPreimeter;>;

// Обработчик специализации, запускаемый в потоке.
// Вычисляет периметр прямоугольника.
void RunThread<ThreadData<RectPreimeter> *rp>() {
  rp->@p = (double)((rp->@r.x+rp->@r.y)*2);
}
```

# Повышение надёжности программирования

```
int main (int argc, char *argv[], char
*envp[]) {
  ThreadRect thread1{3, 5};
  ThreadRect thread2{13, 15};
  ThreadRect thread3{4, 10};

  thread1.start();
  thread2.start();
  thread3.start();

  thread1.wait();
  thread2.wait();
  thread3.wait();

  thread1.print("Tread1");
  thread2.print("Tread2");
  thread3.print("Tread3");

  return EXIT_SUCCESS;
}
```

```
int main () {
    ThreadData<RectPerimeter> thread1 =
        {0}<{3,5,0.0}>;
    ThreadData<RectPerimeter> thread2 =
        {0}<{7,4,0.0}>;
    ThreadData<RectPerimeter> thread3 =
        {0}<{6,8,0.0}>;

    StartThread(&thread1);
    StartThread(&thread2);
    StartThread(&thread3);

    WaitThread(&thread1);
    WaitThread(&thread2);
    WaitThread(&thread3);

    PrintRectPerimeter(&(tread1.@), "Tread1");
    PrintRectPerimeter(&(tread2.@), "Tread2");
    PrintRectPerimeter(&(tread3.@), "Tread3");

    return 0;
}
```

# Итоги и варианты развития

1. Иная разновидность полиморфизма с непосредственной инструментальной поддержкой мультиметодов.
2. Гибкое эволюционное расширение программ без изменения ранее написанного кода как при нисходящем, так и при восходящем проектировании.
3. Более мелкие фракции данных и функций, используемые в процессе инкрементального наращивания кода.
4. Относительная независимость формирования данных и функций, позволяющая использовать ППП с другими парадигмами и, как следствие, возможность ее встраивания даже в уже существующие языки программирования (есть ли смысл в дублировании иного полиморфизма?).
5. Много обобщений от одних и тех же основ специализаций.
6. Возможность реализации в бестиповых (ассемблерных) языках. Это позволяет из базовых данных вывести систему типов, превратив язык ассемблера в типизированный язык (с ООП и АТД это было. Нужно ли здесь?).
7. Концептуально ничто не мешает интегрировать в языки с динамической типизацией (но есть ли в этом смысл?).
8. Возможность окончательного формирования параметрических отношений (связывания воедино) на различных стадиях (компоновка <o2m>, начальная загрузка <p2c>, во время выполнения <?>.
9. Если в вызове обработчика специализаций подставлены конкретные специализации, то можно убрать из параметричесой таблицы соответствующие измерения. Вплоть до непосредственной подстановки нужного обработчика.
10. Повышение надежности и качества когда для ненадежных языков за счет использования ППП в качестве обертки ненадежных конструкций (в качестве примера можно назвать язык C).
11. Перекрытие возможностей ОО парадигмы <монометодов>, плюс дополнительные возможности по повышению эффективности альтернативных решений (альтернативная реализация паттернов ОО проектирования).
12. Возможность гибкой трансформации структуры ПП программ, что позволяет легко заменить параметрические таблицы на другие методы реализации, включая использование конструкций, применяемых в традиционном процедурном (монолитном) программировании.
13. В качестве специализаций можно использовать именованные типы, указатели на них, неименованные структуры, указатели на функции <?>. Указатели и неименованные структуры используются с явно задаваемыми признаками.
14. Параметризация значениям <??>
15. Дополнительные возможности по автоматизации программирования с использованием ИИ за счет гибкости при модификации кода и формировании новых подстановок в код <???>

# Дополнительная информация


1. Легалов А.И. Процедурно-параметрическая парадигма программирования. Возможна ли альтернатива объектно-ориентированному стилю? - Красноярск: 2000. Деп. рук. № 622-В00 Деп. в ВИНИТИ 13.03.2000. - 43 с. <http://www.softcraft.ru/ppp/pppfirst/>
2. Легалов А.И. ООП, мультиметоды и пирамидальная эволюция (2002) <http://www.softcraft.ru/coding/evo/>
3. Легалов А.И. Эволюция мультиметодов при процедурном подходе (2002) <http://www.softcraft.ru/coding/evp/>
4. Легалов И.А. Применение обобщенных записей в процедурно-параметрическом языке программирования. / И.А. Легалов // Научный вестник НГТУ. – 2007. – № 3 (28). – С. 25-38. <http://www.softcraft.ru/ppp/genrecords/>
5. Легалов А.И., Бовкун А.Я., Легалов И.А. Расширение модульной структуры программы за счет подключаемых модулей. / Доклады АН ВШ РФ, № 1 (14). – 2010. – С. 114-125. <http://www.softcraft.ru/ppp/inclmodules/>
6. Легалов А.И., Легалов И.А., Солоха А.Ф. Эволюционное расширение программ при различных парадигмах программирования. / Труды XVI Байкальской Всероссийской конференции «Информационные и математические технологии в науке и управлении». Часть III. - Иркутск: ИСЭМ СО РАН, 2011. ISBN 978-5-93908-094-1. - С. 42-49. <http://www.softcraft.ru/ppp/simplesituations/>
7. Легалов А.И., Косов П.В. Эволюционное расширение программ с использованием процедурно-параметрического подхода // Вычислительные технологии. 2016. Т. 21. № 3. С. 56-69. <http://www.ict.nsc.ru/jct/content/t21n3/Legalov_n.pdf>
8. Легалов А.И., Косов П.В. Расширение языка С для поддержки процедурно-параметрического полиморфизма. Моделирование и анализ информационных систем. 2023;30(1):40-62. <https://doi.org/10.18255/1818-1015-2023-1-40-62>

# Благодарю за внимание!

http://www.softcraft.ru/ppp/