Архитектура уровня статически типизированного процедурного языка

Статическая однозначность

Статическая однозначность операций формируется за счет того, что с каждым значением в программе сопоставляется его тип. Этот тип задается при описании переменных и может быть проверен во время компиляции. Для всех временных и промежуточных значений тип может быть также выведен во время компиляции. Поэтому его не имеет смысла проверять во время выполнения. Одна и та же операция может быть задана с разными типами, но все вопросы по ее конкретному выполнению решаются во время компиляции (статический полиморфизм). С каждой переменной сопоставляется только один тип. Допускает эффективную трансформацию в бестиповые архитектуры уровня системы команд. Используется в языках компилируемого типа.

Примеры:

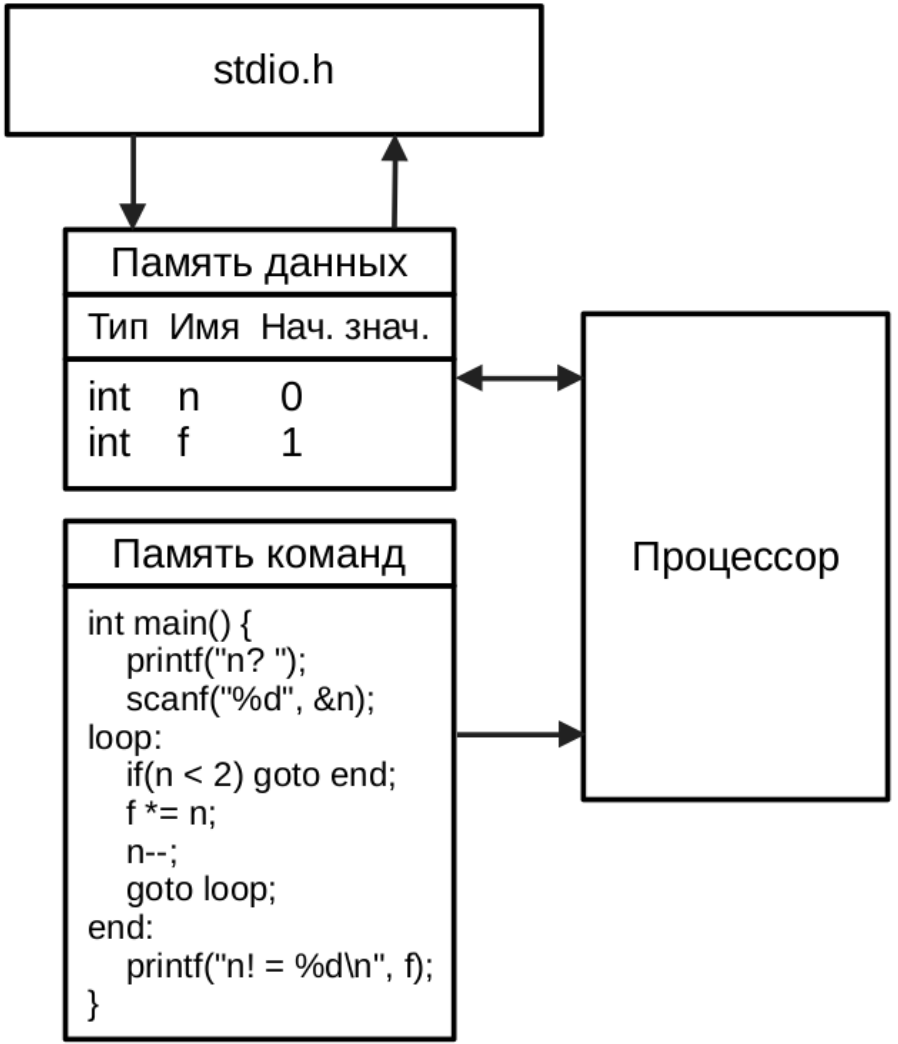
1) Императивные языки программирования:C, C++, Pascal, Oberon family, Java, C#, Rust, Go...

2) Языки функционального программирования: ML, Haskell...

Программа для компьютера

```
1   #include <cstdio>
2
3   int n;
4   int f = 1;
5
6   int main() {
7       printf("n? ");
8       scanf("%d", &n);
9   loop:
10      if(n < 2) goto end;
11      f *= n;
12      n--;
13      goto loop;
14  end:
15      printf("n! = %d\n", f);
16  }
```

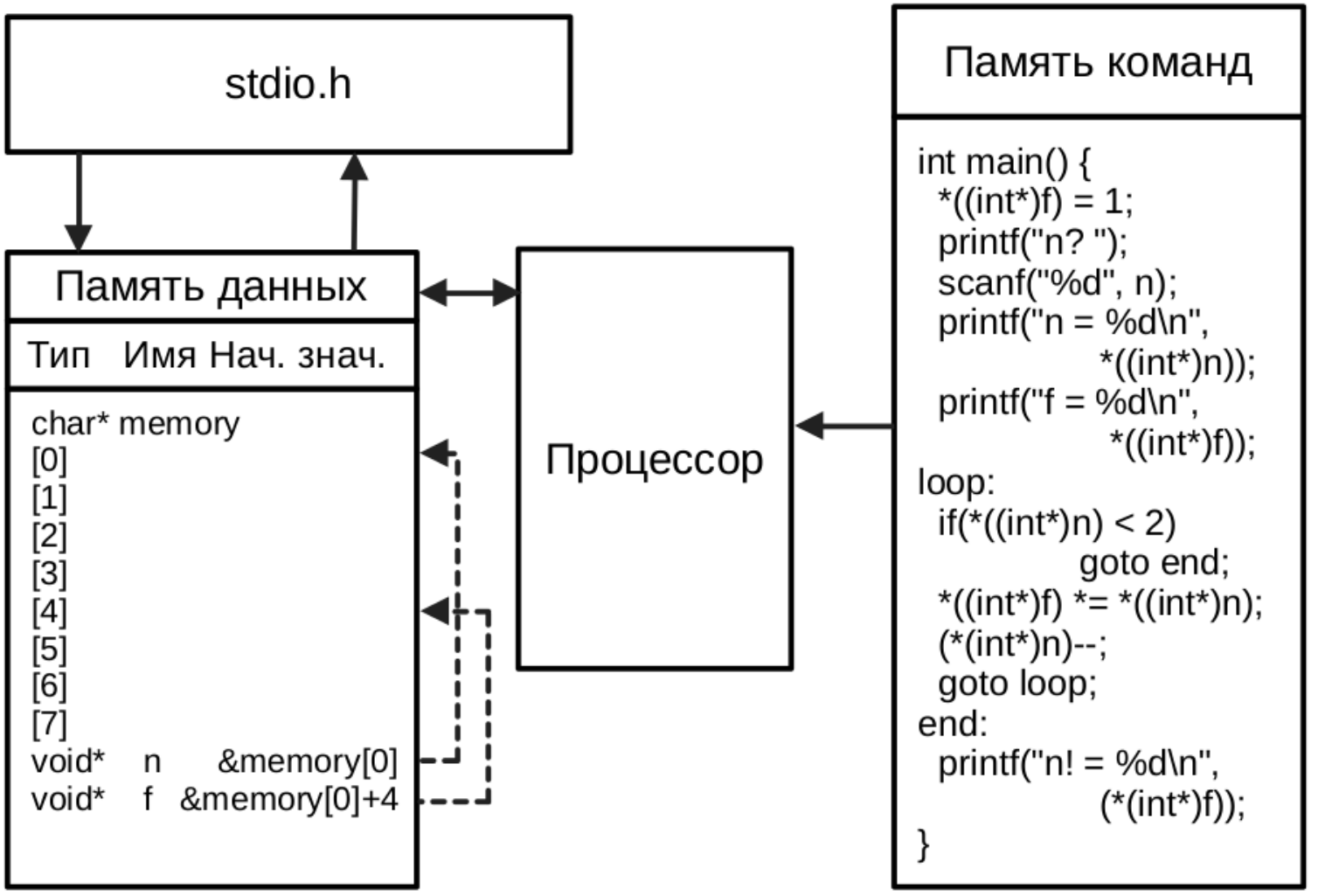
Отображение программы на структуру



Бестиповое программирование на C++

```
1  #include <cstdio>
2
3  char memory[2*sizeof(int)];      // Память для n и f
4  void* n = memory;                // Адрес на область для n
5  void* f = memory + sizeof(int); // Адрес на область для f
6
7  int main() {
8      *((int*)f) = 1;
9      printf("n? ");
10     scanf("%d", n);
11     printf("n = %d\n", *((int*)n));
12     printf("f = %d\n", *((int*)f));
13 loop:
14     if(*((int*)n) < 2) goto end;
15     *((int*)f) *= *((int*)n);
16     (*(int*)n)--;
17     goto loop;
18 end:
19     printf("n! = %d\n", (*(int*)f));
20 }
```

Отображение бестиповой программы на структуру



Избыточность статической типизации

```
1  #include <cstdio>
2
3  int n;
4  int f = 1;
5
6  int main() {
7      printf("n? ");
8      scanf("%d", &n);
9  loop:
10     if(n < 2) goto end;
11     f *= n;
12     n--;
13     goto loop;
14 end:
15     printf("n! = %d\n", f);
16 }
```

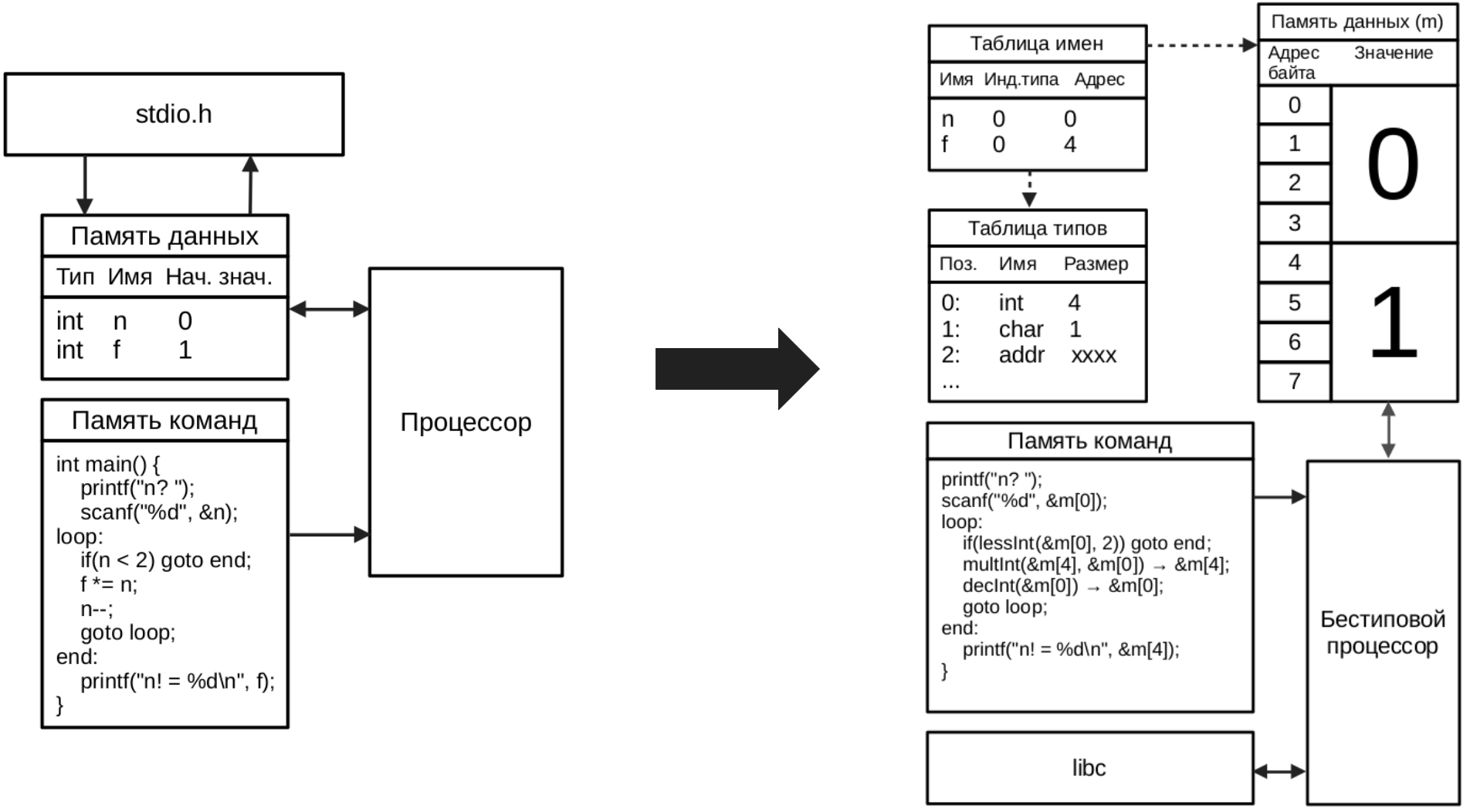
man 3 printf
man 3 scanf

```
int printf(char *, …) →
int scanf(char *, …) →

< (int, int) → bool
*(int, int) → int; = (int) → int
--(int) → int
```

Трансформация к операционной однозначности:

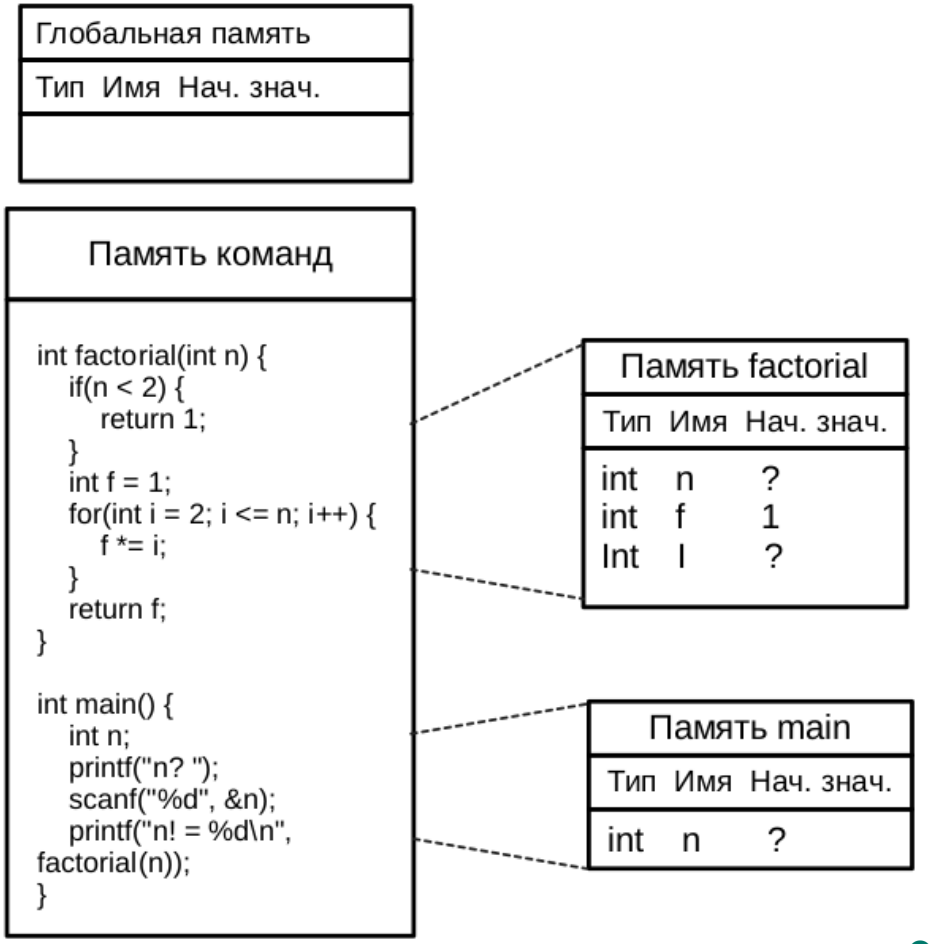
```
lessInt(void*, void*) → void*
multInt(void*, void*) → void*
movInt(void*) → void*
decInt(void*) → void*
```

Трансформация статической типизации



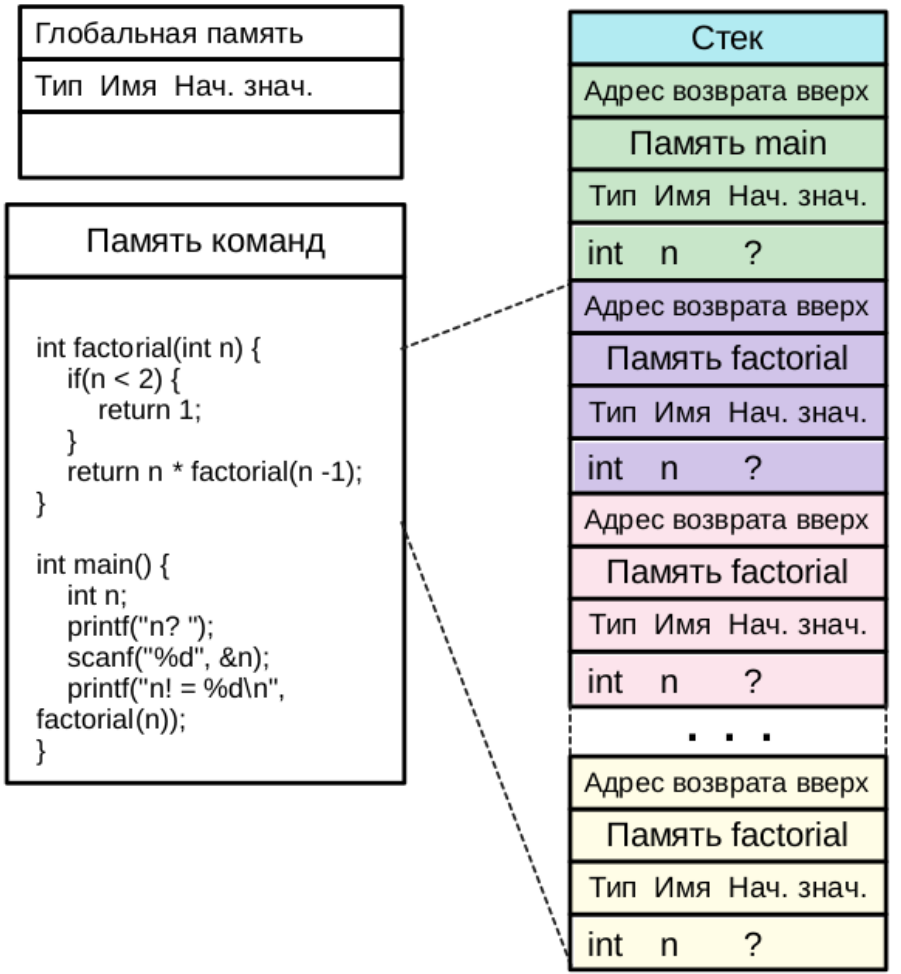
Статическая типизация и локальные данные

```
1  #include <cstdio>
2
3  int factorial(int n) {
4      if(n < 2) {
5          return 1;
6      }
7      int f = 1;
8      for(int i = 2; i <= n; i++) {
9          f *= i;
10     }
11     return f;
12 }
13
14 int main() {
15     int n;
16     printf("n? ");
17     scanf("%d", &n);
18     printf("n! = %d\n", factorial(n));
19 }
```



Статическая типизация и рекурсия

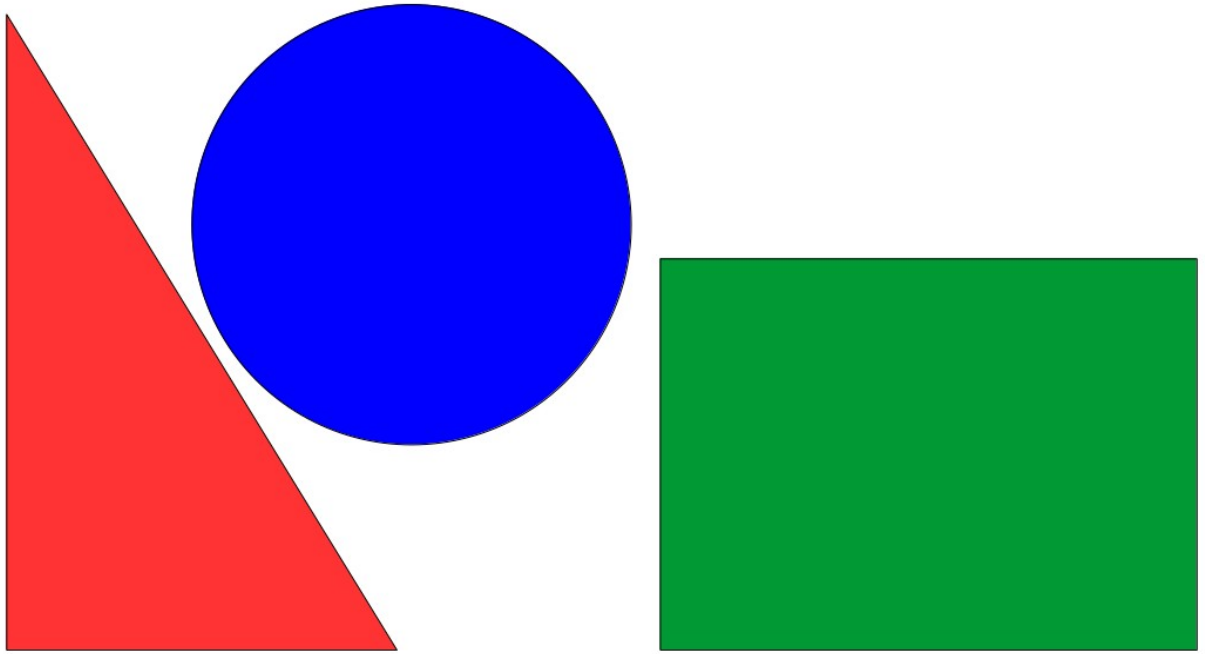
```cpp
#include <cstdio>

int factorial(int n) {
    if(n < 2) {
        return 1;
    }
    return n * factorial(n -1);
}

int main() {
    int n;
    printf("n? ");
    scanf("%d", &n);
    printf("n! = %d\n", factorial(n));
}

```



Статическая типизация и абстрактные типы данных

Базовые понятия для конструирования составных программных объектов:

- *агрегаты*
- *обобщения (категории – «is a»)*



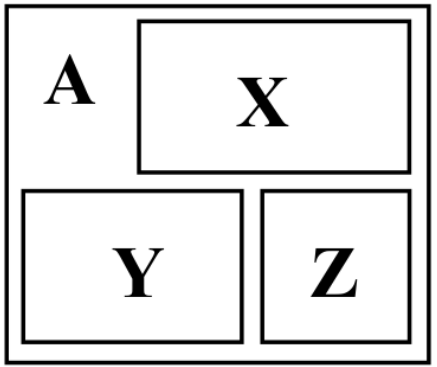
Конструирование агрегатов

Агрегация (агрегирование) - это абстрагирование, посредством которого один объект конструируется из других [Цикритзис].

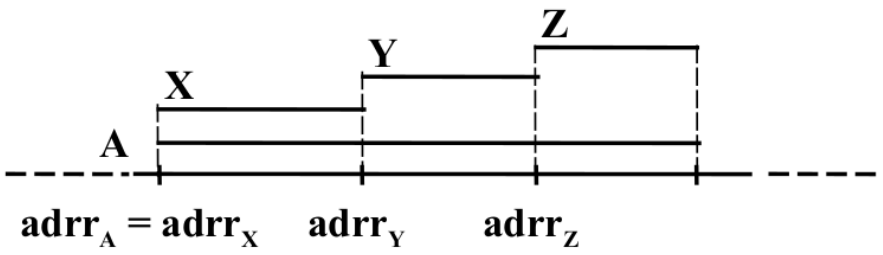
Агрегирование обеспечивает формирование программных объектов одним из способов:

- **непосредственным включением**,
- **косвенным (ссылочным) связыванием**,
- **с применением наследования (расширения)**,
- **образного восприятия**.

Агрегат A из элементов X, Y, Z, построенный с использованием непосредственного включения



а) условное изображение на плоскости



б) отображение на одномерное адресное пространство

Непосредственное включение

Воспринимается как единый, **монолитный ресурс**, занимающий **неразрывное пространство**. Цельность и законченность данной конструкции не требует выполнения ***дополнительных алгоритмов***, связанных с формированием его структуры.
Можно **сразу** приступать к **использованию** (инициализации, обработке).

Агрегаты данных и действий построенные с использованием непосредственного включения

```
// прямоугольник
struct rectangle {
    int x, y; // ширина, высота
};
```

```
// Вычисление периметра прямоугольника
double Perimeter(rectangle &r) {
    return 2.0 * (r.x + r.y);
}
```

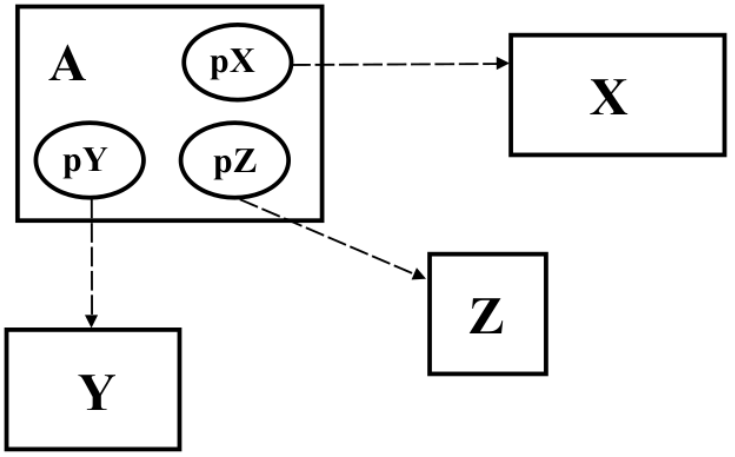
```
// треугольник
struct triangle {
    int a, b, c; // стороны
};
```

```
// Вычисление периметра треугольника
double Perimeter(triangle &t) {
    return double(t.a + t.b + t.c);
}
```

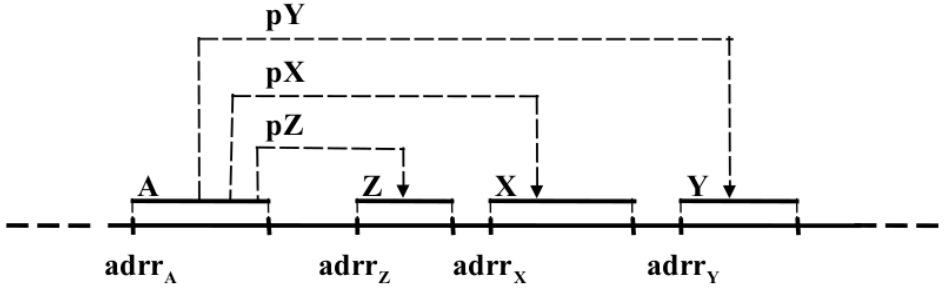
```
// Простейший контейнер на основе одномерного массива
struct container {
    enum {max_len = 10000}; // максимальная длина
    int len; // текущая длина
    shape cont[max_len];
};
```

```
// Вычисление суммы периметров всех фигур в контейнере
double PerimeterSum(container &c) {
    double sum = 0.0;
    for(int i = 0; i < c.len; i++) {
        sum += Perimeter(c.cont[i]);
    }
    return sum;
}
```

Агрегат A из элементов X, Y, Z, построенный с использованием косвенного связывания



а) условное изображение на плоскости



б) отображение на одномерное адресное пространство

Косвенное связывание

Позволяет формировать агрегаты из **отдельных элементов**, уже располагаемых в пространстве. Взаимосвязь осуществляется **алгоритмическим заданием ссылок**.
Специфика: **выполнение кода обеспечивающего конструирование объекта** из разрозненных экземпляров абстракций. Этот код выполняется один раз, после чего работа с агрегатом осуществляется как и при непосредсвенном включении.

В ряде случаев, когда элементы и агрегат создаются статически, инициализация связей может быть проведена во время компиляции программы.

Агрегаты данных и действи построенные с использованием косвенного связывания (версия 1)

```
// Простейший контейнер на основе одномерного массива
struct container {
    enum {max_len = 10000}; // максимальная длина
    int len; // текущая длина
    shape *cont[max_len];
};
```

```
// Ввод содержимого контейнера из указанного потока
void In(container &c, ifstream &ifst) {
    while(!ifst.eof()) {
        if((c.cont[c.len] = In(ifst)) != 0) {
            c.len++;
        }
    }
}
```

Агрегаты данных и действи построенные с использованием косвенного связывания (версия 2)

```
// Простейший контейнер на основе одномерного массива
struct container {
    int len; // текущая длина
    shape *cont;
};
```

```
// Инициализация контейнера
void Init(container &c, int n) {
    c.cont = new shape[n];
    c.len = 0;
}
```

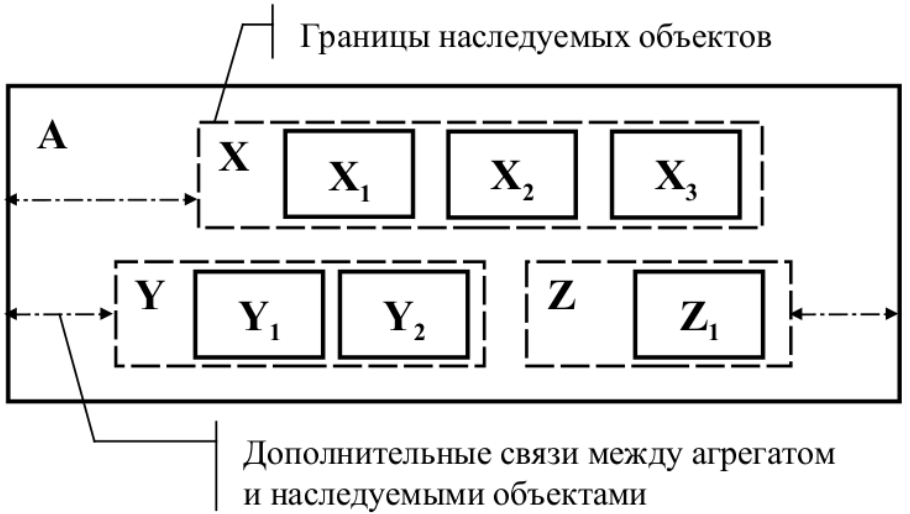
Агрегаты данных и действи построенные с использованием косвенного связывания (версия 3)

```
// Простейший контейнер на основе одномерного массива
struct container {
    int len; // текущая длина
    shape **cont;
};
```

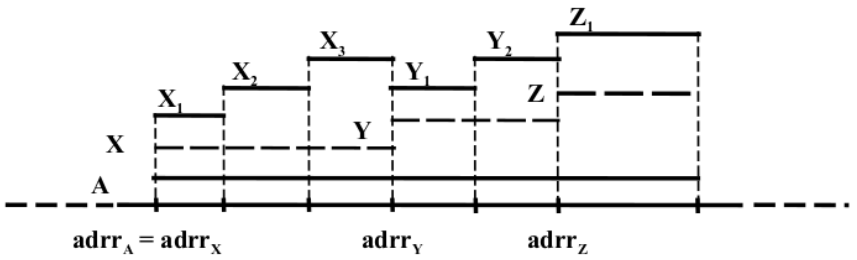
```
// Инициализация контейнера
void Init(container &c, int n) {
    c.cont = new shape*[n];
    c.len = 0;
}
```

```
// Ввод содержимого контейнера из указанного потока
void In(container &c, ifstream &ifst) {
    while(!ifst.eof()) {
        if((c.cont[c.len] = In(ifst)) != 0) {
            c.len++;
        }
    }
}
```

Агрегат A из элементов X, Y, Z, построенный с использованием наследования



а) условное изображение на плоскости



б) отображение на одномерное адресное пространство

Наследование

Позволяет создавать артефакт, эквивалентный непосредственному включением. Но поддерживает свойство **конкатенации**, в результате чего обращение к элементам подключаемых абстракций осуществляется **напрямую**. **Сохраняется информация о включенных абстракциях**, которая может использоваться для разрешения неоднозначности доступа к элементам при совпадении их имен. В отличие от непосредственного включения, каждому наследуемому элементу "предоставляется" дополнительная информация о формируемом агрегате. Это позволяет правильно манипулировать агрегатом, используя лишь сведения об одном из его базовых (включаемых) элементов.

Агрегаты данных и действий построенные с использованием наследования

```
// структура, обобщающая все имеющиеся фигуры
struct shape {
    // значения ключей для каждой из фигур
    enum key {RECTANGLE, TRIANGLE};
    key k; // ключ
};
```

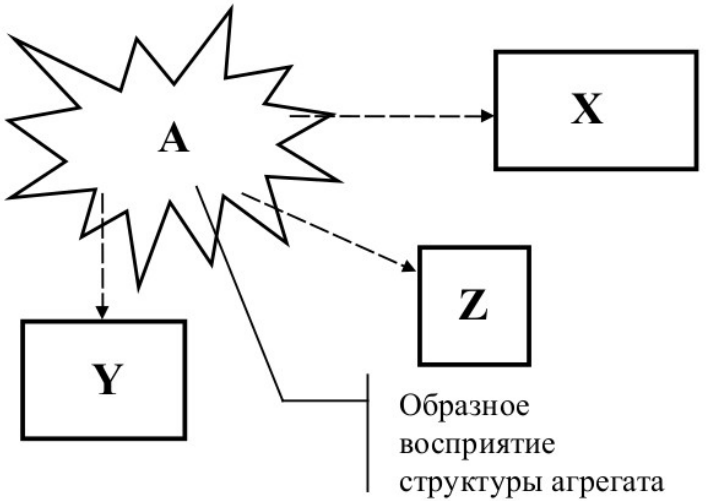
```
// прямоугольник
struct rectangle: shape {
    int x, y; // ширина, высота
};
```

```
// треугольник
struct triangle: shape {
    int a, b, c; // стороны
};
```

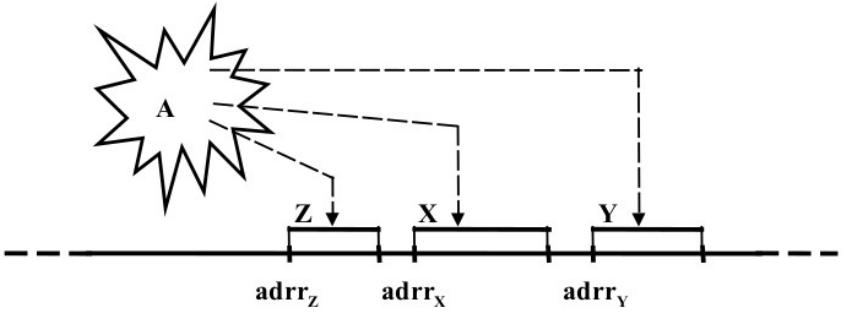
```
// Вычисление периметра прямоугольника
double Perimeter(rectangle &r) {
    return 2.0 * (r.x + r.y);
}
```

```
// Вычисление периметра треугольника
double Perimeter(triangle &t) {
    return double(t.a + t.b + t.c);
}
```

Агрегат A из элементов X, Y, Z, построенный на основе образного восприятия



а) условное изображение на плоскости



б) отображение на одномерное адресное пространство

Образное агрегирование

Связано с отсутствием специально созданной абстракции. Вместо этого **агрегат воссоздается только в мыслях программиста**, а на уровне программы имеются его разрозненные элементы, обрабатываемые как единое целое. Например, точку на плоскости можно представить как две независимые целочисленные переменные x и y. Уходит корнями в далекое прошлое (эпоху Фортрана и Алгола-60). И сейчас некоторым "лень" вписать лишнюю абстракцию. Или код на языке уровня системы команд.

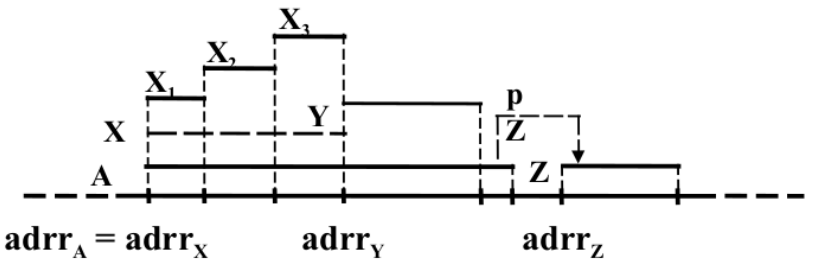
Агрегаты данных и действий построенные на основе образного восприятия

```
// Простейший контейнер на основе одномерного массива
struct container {
    enum {max_len = 10000}; // максимальная длина
    int len; // текущая длина
    void *cont[max_len];
};
    // прямоугольник
    const int RECTANGLE = 1;

    struct rectangle {
        int k;
        int x, y; // ширина, высота
    };

// треугольник
const int TRIANGLE = 2;

struct triangle {
    int k;
    int a, b, c; // стороны
};
```

```
// Вычисление периметра фигуры
double Perimeter(void *s) {
    switch(((rectangle*)s)->k) {
        case RECTANGLE:
            return Perimeter(*reinterpret_cast<rectangle*>(s));
            break;
        case TRIANGLE:
            return Perimeter(*reinterpret_cast<triangle*>(s));
            break;
        default:
            return 0.0;
    }
}
```

Агрегат A из элементов X, Y, Z, построенный с использованием наследования, непосредственного включения и косвенного связывания

а) условное изображение на плоскости



б) отображение на одномерное адресное пространство

Смешанное агрегирование

Наряду с **"чистыми"** методами, агрегаты могут строиться по **смешанному** принципу.

Такой подход является **наиболее популярным** в настоящее время, так как позволяет рационально использовать различную технику в зависимости от организации и назначения агрегируемых элементов.

Агрегаты данных и действий построенные с использованием комбинаций

```
// Простейший контейнер на основе одномерного массива
struct container {
    int len; // текущая длина
    shape *cont;
};
```

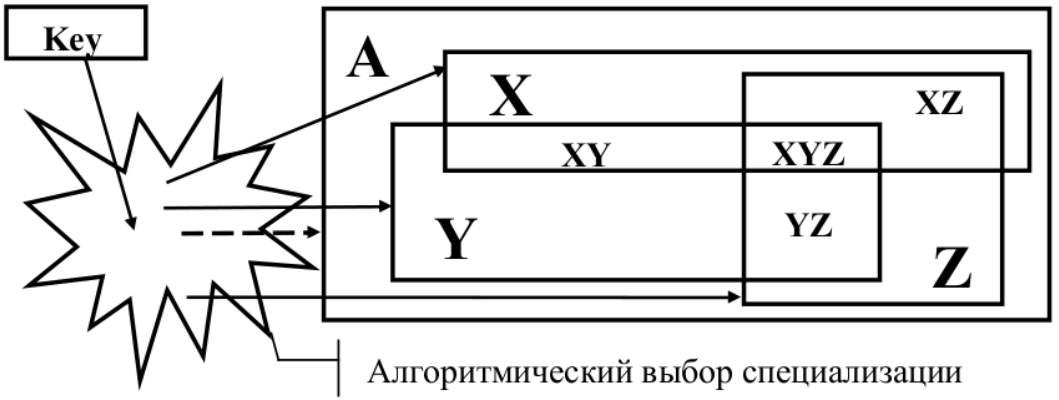
```
// Инициализация контейнера
void Init(container &c, int n) {
    c.cont = new shape[n];
    c.len = 0;
}
```

Конструирование обобщений

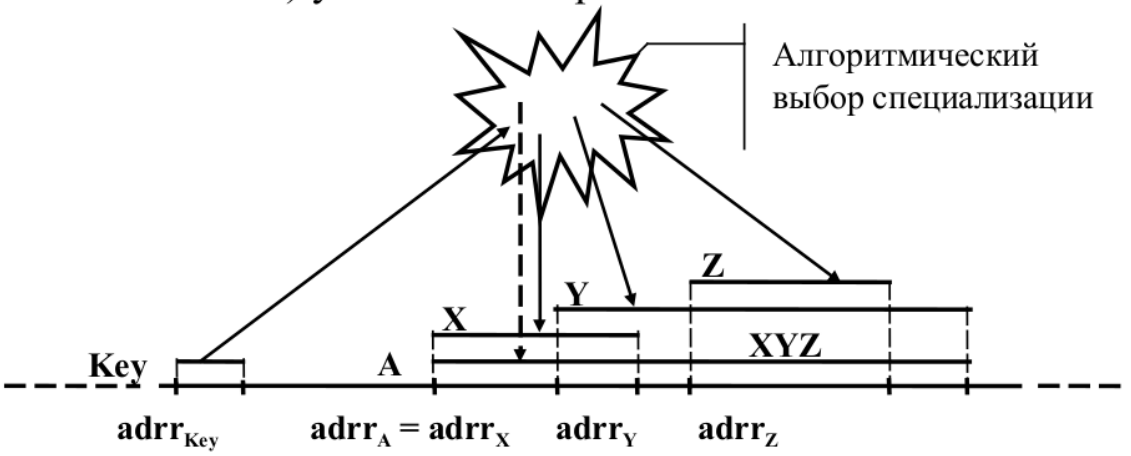
Обобщение - это композиция альтернативных по своим свойствам программных объектов, принадлежащих к единой категории в некоторой системе классификации.

Выделяются:
обобщение данных и ***обобщение процедур (процедурное обобщение)***. Обобщение данных состоит из ***основы обобщения***, к которой присоединяются различные ***основы специализаций***.

Обобщение A, построенное с использованием объединения на основе общего ресурса (начало)



а) условное изображение на плоскости



б) отображение на одномерное адресное пространство

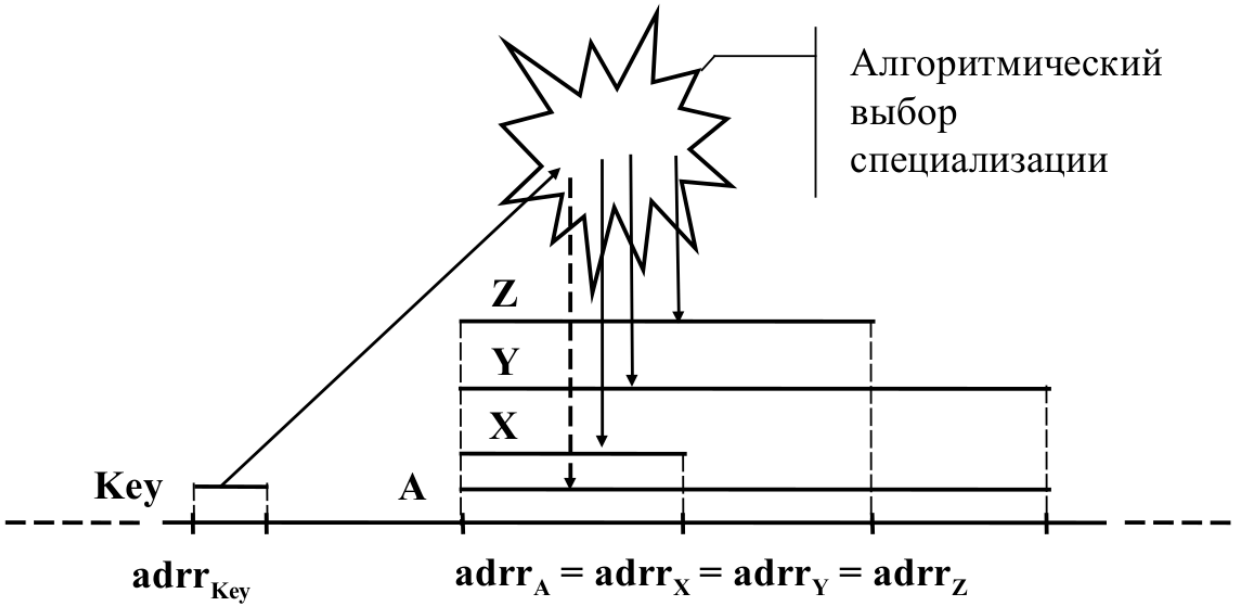
Обобщение на основе общего ресурса

Размещение специализаций обобщения в едином адресном пространстве.

При этом обычно существует часть ресурса, одновременно перекрываемая всеми размещаемыми программными объектами.

Но чаще всего обобщения на основе общего ресурса строится таким образом, что начальный адрес для всех размещаемых объектов является одинаковым.

Обобщение A, построенное с использованием объединения на основе общего ресурса (окончание)



в) отображение на одномерное адресное пространство при размещении специализаций по одному начальному адресу

Два варианта использования

1. Хранение одного из альтернативных объектов. Выделенное пространство предоставляется экземпляру абстракции только одного типа, который хранит свое значение, никоим образом по семантике не пересекающееся с семантикой альтернативных объектов.

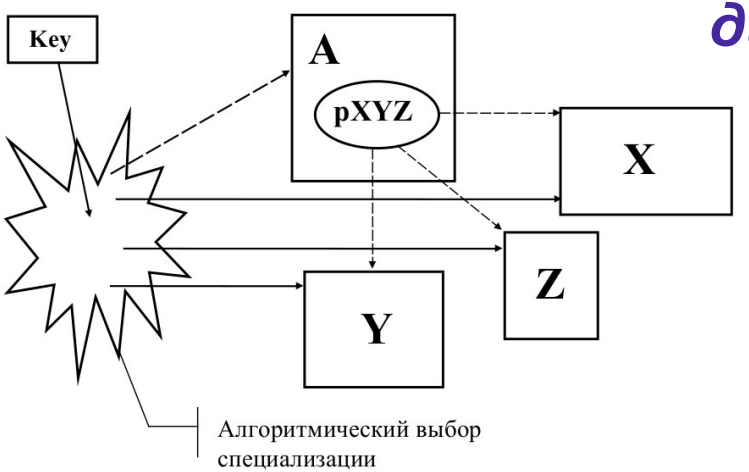
2. Использование различных трактовок одного и того же пространства ресурсов (целое число длиной два байта может интерпретироваться как единое слово, для операций сохранения-восстановления в двоичном формате, как младший и старший байты).

Обобщения данных и действий построенные с использованием объединения на основе общего ресурса

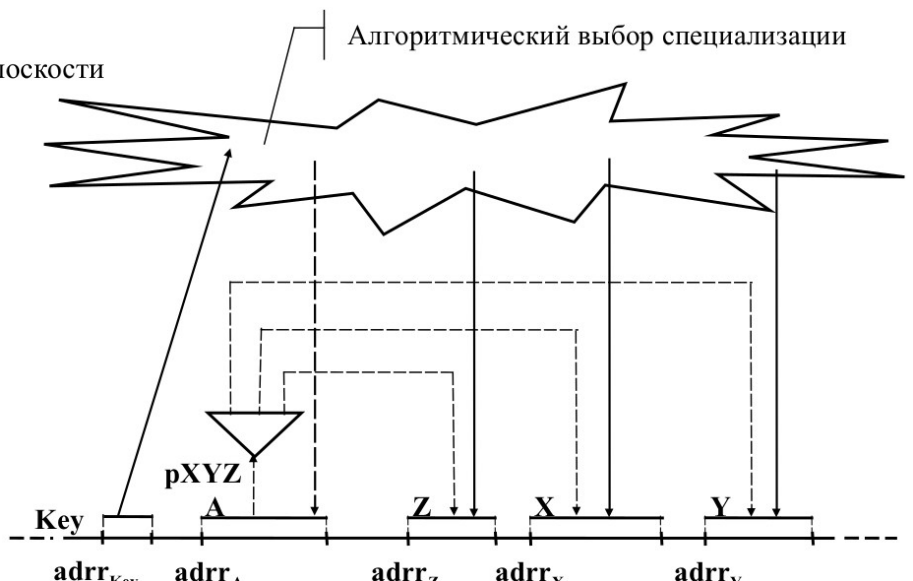
```
// структура, обобщающая все имеющиеся фигуры
struct shape {
    // значения ключей для каждой из фигур
    enum key {RECTANGLE, TRIANGLE};
    key k; // ключ
    // используемые альтернативы
    union { // используем простейшую реализацию
        rectangle r;
        triangle t;
    };
};
```

```
// Вычисление периметра фигуры
double Perimeter(shape &s) {
    switch(s.k) {
        case shape::RECTANGLE:
            return Perimeter(s.r);
            break;
        case shape::TRIANGLE:
            return Perimeter(s.t);
            break;
        default:
            return 0.0;
    }
}
```

Обобщение A, построенное с использованием динамического связывания



а) условное изображение на плоскости



б) отображение на одномерное адресное пространство

Альтернативное связывание

Независимое размещение специализаций. Использование ссылки или указателя обеспечивает доступ только к одному избранному объекту. Связывание обычно осуществляется алгоритмически. Выполнение связывания, происходит для отдельного экземпляра обобщения только один раз. Работа с обобщением осуществляется на основе анализа значения ключа. Это позволяет правильно обработать специализацию через "обезличенный" указатель.

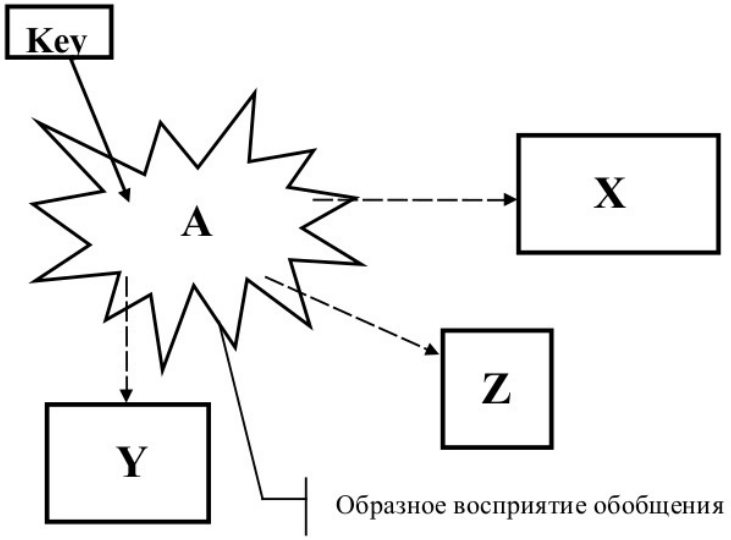
Обобщения данных и действий построенные с использованием динамического связывания

```
// структура, обобщающая все имеющиеся фигуры
struct shape {
    // значения ключей для каждой из фигур
    enum key {RECTANGLE, TRIANGLE};
    key k; // ключ
    // используемые альтернативы
    union { // используем простейшую реализацию
        rectangle *pr;
        triangle *pt;
    };
};
```

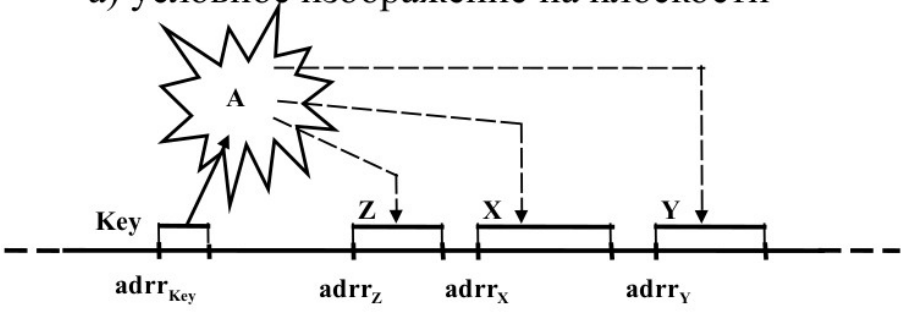
```
// Очистка специализаций обобщенной фигуры
void Clear(shape &s) {
    switch(s.k) {
        case shape::RECTANGLE:
            delete s.pr;
            return;
        case shape::TRIANGLE:
            delete s.pt;
            return;
        default:
            return;
    }
}
```

```
// Вычисление периметра фигуры
double Perimeter(shape &s) {
    switch(s.k) {
        case shape::RECTANGLE:
            return Perimeter(*s.pr);
        case shape::TRIANGLE:
            return Perimeter(*s.pt);
        default:
            return 0.0;
    }
}
```

Обобщение A из элементов X, Y, Z, построенное на основе образного восприятия



а) условное изображение на плоскости



б) отображение на одномерное адресное пространство

Образное обобщение

Как и образное агрегирование, связано с мысленными ассоциациями программиста. Используя ключевой признак, можно связать с отдельными специализациями подмножества разрозненных объектов программы, семантическая связь между которыми поддерживается на уровне алгоритма. При этом одни и те же программные объекты могут использоваться в альтернативных специализациях, имея при этом различную семантическую трактовку.

Как и в случае с образным агрегированием, такой подход к написанию программ является **"пережитком прошлого"**. Достаточно часто он использовался и для экономии памяти, когда одни и те же переменные интерпретировались различным образом. Используется также в бестиповых языках (Ассемблеры)

Агрегаты данных и действий построенные на основе образного восприятия

```
// Простейший контейнер на основе одномерного массива
struct container {
    enum {max_len = 10000}; // максимальная длина
    int len; // текущая длина
    void *cont[max_len];
};
    // прямоугольник
    const int RECTANGLE = 1;

    struct rectangle {
        int k;
        int x, y; // ширина, высота
    };

// треугольник
const int TRIANGLE = 2;

struct triangle {
    int k;
    int a, b, c; // стороны
};
```

```
// Вычисление периметра фигуры
double Perimeter(void *s) {
    switch(((rectangle*)s)->k) {
        case RECTANGLE:
            return Perimeter(*reinterpret_cast<rectangle*>(s));
            break;
        case TRIANGLE:
            return Perimeter(*reinterpret_cast<triangle*>(s));
            break;
        default:
            return 0.0;
    }
}
```

Обобщения данных и действий построенные с использованием образного восприятия (версия 1)

```
// Вывод параметров текущей фигуры в поток
void Out(void *s, ofstream &ofst) {
    switch(((rectangle*)s)->k) {
        case RECTANGLE:
            Out(*((rectangle*)s), ofst);
            break;
        case TRIANGLE:
            Out(*((triangle*)s), ofst);
            break;
        default:
            ofst << "Incorrect figure!" << endl;
    }
}

//-----------------------------------------------------------
// Вычисление периметра фигуры
double Perimeter(void *s) {
    switch(((rectangle*)s)->k) {
        case RECTANGLE:
            return Perimeter(*reinterpret_cast<rectangle*>(s));
            break;
        case TRIANGLE:
            return Perimeter(*reinterpret_cast<triangle*>(s));
            break;
        default:
            return 0.0;
    }
}
```

```
// Удаление обобщенной фигуры
void DeleteShape(void *s) {
    switch(((rectangle*)s)->k) {
        case RECTANGLE:
            delete (rectangle*)s;
            break;
        case TRIANGLE:
            delete (triangle*)s;
            break;
        default:
            break;
    }
}
```

Агрегаты данных и действий построенные с использованием наследования

```
// структура, обобщающая все имеющиеся фигуры
struct shape {
    // значения ключей для каждой из фигур
    enum key {RECTANGLE, TRIANGLE};
    key k; // ключ
};
```

```
// прямоугольник
struct rectangle: shape {
    int x, y; // ширина, высота
};
```

```
// треугольник
struct triangle: shape {
    int a, b, c; // стороны
};
```

```
// Вычисление периметра прямоугольника
double Perimeter(rectangle &r) {
    return 2.0 * (r.x + r.y);
}
```

```
// Вычисление периметра треугольника
double Perimeter(triangle &t) {
    return double(t.a + t.b + t.c);
}
```

Обобщения данных и действий построенные с использованием образного восприятия (версия 2)

```
// Вывод параметров текущей фигуры в поток
void Out(shape &s, ofstream &ofst) {
    switch(s.k) {
        case shape::RECTANGLE:
            Out(*((rectangle*)&s), ofst);
            break;
        case shape::TRIANGLE:
            Out(*((triangle*)&s), ofst);
            break;
        default:
            ofst << "Incorrect figure!" << endl;
    }
}

//-----------------------------------------------------------------
// Вычисление периметра фигуры
double Perimeter(shape &s) {
    switch(s.k) {
        case shape::RECTANGLE:
            return Perimeter(*reinterpret_cast<rectangle*>(&s));
            break;
        case shape::TRIANGLE:
            return Perimeter(*reinterpret_cast<triangle*>(&s));
            break;
        default:
            return 0.0;
    }
}
```

```
// Удаление обобщенной фигуры
void DeleteShape(shape *s) {
    switch(((rectangle*)s)->k) {
        case shape::RECTANGLE:
            delete (rectangle*)s;
            break;
        case shape::TRIANGLE:
            delete (triangle*)s;
            break;
        default:
            break;
    }
}
```

Использование автоматически выделяемых массивов

```
int main() {
    int len;
    std::cout << "input length: ";
    std::cin >> len;
    in_out_vector(len);
    return 0;
}
```

```
void sort(int *vec, int len) {
    for(int i = 0; i < len-1; i++) {
        int i_min = i;
        for(int j = i + 1; j < len; j++) {
            if(vec[i_min] > vec[j]) {
                int tmp = vec[i_min];
                vec[i_min] = vec[j];
                vec[j] = tmp;
            }
        }
    }
}
```

```
void in_out_vector(int len) {
    int vec[len];
    std::cout << "length = " << len << "\n";
    for(int i = 0; i < len; i++) {
        vec[i] = len - i;
    }
    sort(vec, len);
    for(int i = 0; i < len; i++) {
        printf("%d  ", vec[i]);
    }
    printf("\n");
}
```

Использование автоматически выделяемых массивов

```
if(!strcmp(argv[1], "-f")) {
    ifstream ifst(argv[2]);
    // Создание фиксированного одномерного массива
    shape arr1[100];
    Init(c, arr1);
    In(c, ifst);
    goto m1;
}
else if(!strcmp(argv[1], "-n")) {
    auto size = atoi(argv[2]);
    if((size < 1) || (size > 10000)) {
        cout << "incorrect numer of figures = "
             << size
             << ". Set 0 < number <= 10000\n";
        return 3;
    }
    // Создание массива переменной размерности
    shape arr2[size];
    Init(c, arr2);

    // системные часы в качестве инициализатора
    srand(static_cast<unsigned int>(time(0)));
    // Заполнение контейнера генератором случайных чисел
    InRnd(c, size);
}
else {
    errMessage2();
    return 2;
}
// Вывод содержимого контейнера в файл
ofstream ofst1(argv[3]);
ofst1 << "Filled container:\n";
Out(c, ofst1);
```

```
[al@parsys build]$ ../bin/task01 -f ../tests/test01.txt 1.txt 2.txt
Start
Start Init
End Init
Stop
[al@parsys build]$ cat 1.txt 2.txt
Filled container:
Container contains 8 elements.
0: It is Rectangle: x = 3, y = 4. Perimeter = 14
1: It is Triangle: a = 1, b = 1, c = 1. Perimeter = 3
2: It is Rectangle: x = 30, y = 40. Perimeter = 140
3: It is Triangle: a = 2, b = 2, c = 1. Perimeter = 5
4: It is Rectangle: x = 13, y = 14. Perimeter = 54
5: It is Triangle: a = 10, b = 10, c = 10. Perimeter = 30
6: It is Rectangle: x = 330, y = 49. Perimeter = 758
7: It is Triangle: a = 3, b = 4, c = 5. Perimeter = 12
Perimeter sum = 0
[al@parsys build]$ ../bin/task01 -n 5 1.txt 2.txt
Start
Start Init
End Init
Stop
[al@parsys build]$ cat 1.txt 2.txt
Filled container:
Container contains 5 elements.
0: Incorrect figure!
1: Incorrect figure!
2: Incorrect figure!
3: It is Rectangle: x = 0, y = 393049312. Perimeter = 7.86099e+08
4: Incorrect figure!
Perimeter sum = 3.47045e+09
```

Использование автоматически выделяемых массивов

```
int main(int argc, char* argv[]) {
    if(argc != 5) {
        errMessage1();
        return 1;
    }

    cout << "Start"<< endl;
    container c;
    int size = 100;
    bool isFile;

    ////cout << "argv[1] = " << argv[1] << "\n";
    if(!strcmp(argv[1], "-f")) {
        isFile = true;
    }
    else if(!strcmp(argv[1], "-n")) {
        isFile = false;
        size = atoi(argv[2]);
        if((size < 1) || (size > 10000)) {
            cout << "incorrect numer of figures = "
                 << size
                 << ". Set 0 < number <= 10000\n";
            return 3;
        }
    }
    else {
        errMessage2();
        return 2;
    }
```

```
    // Автоматическое выделение памяти для массива
    shape arr[size];
    Init(c, arr);
    std::cout << "size = " << size << "\n";

    // Проверка источника данных
    if(isFile) {
        ifstream ifst(argv[2]);
        In(c, ifst);
    } else {
        // системные часы в качестве инициализатора
        srand(static_cast<unsigned int>(time(0)));
        // Заполнение контейнера генератором случайных чисел
        InRnd(c, size);
    }

    // Вывод содержимого контейнера в файл
    ofstream ofst1(argv[3]);
    ofst1 << "Filled container:\n";
    Out(c, ofst1);

    // The 2nd part of task
    ofstream ofst2(argv[4]);
    ofst2 << "Perimeter sum = " << PerimeterSum(c) << "\n";

    Clear(c);
    cout << "Stop"<< endl;
    return 0;
}
```

Использование автоматически выделяемых массивов

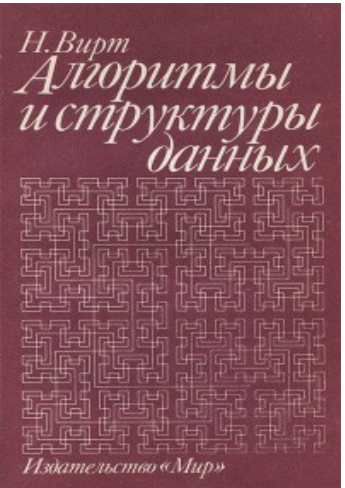
```
[al@parsys build]$ ../bin/task01 -n 5 1.txt 2.txt
Start
Start Init
End Init
size = 5
Start Rnd
End Rnd
Stop
[al@parsys build]$ cat 1.txt 2.txt
Filled container:
Container contains 5 elements.
0: It is Rectangle: x = 11, y = 16. Perimeter = 54
1: It is Triangle: a = 12, b = 12, c = 17. Perimeter = 41
2: It is Rectangle: x = 14, y = 1. Perimeter = 30
3: It is Rectangle: x = 16, y = 3. Perimeter = 38
4: It is Rectangle: x = 16, y = 9. Perimeter = 50
Perimeter sum = 213
[al@parsys build]$ ../bin/task01 -f ../tests/test01.txt 1.txt 2.txt
Start
Start Init
End Init
size = 100
Stop
[al@parsys build]$ cat 1.txt 2.txt
Filled container:
Container contains 8 elements.
0: It is Rectangle: x = 3, y = 4. Perimeter = 14
1: It is Triangle: a = 1, b = 1, c = 1. Perimeter = 3
2: It is Rectangle: x = 30, y = 40. Perimeter = 140
3: It is Triangle: a = 2, b = 2, c = 1. Perimeter = 5
4: It is Rectangle: x = 13, y = 14. Perimeter = 54
5: It is Triangle: a = 10, b = 10, c = 10. Perimeter = 30
6: It is Rectangle: x = 330, y = 49. Perimeter = 758
7: It is Triangle: a = 3, b = 4, c = 5. Perimeter = 12
Perimeter sum = 1016
```

Выводы

1. Независимо от техники и парадигм программирования создаются программные объекты и отношения между ними, с использованием таких понятий как агрегат и обобщение. К одному и тому же конечному результату можно прийти различными путями.
2. Процедурное агрегирование обладает большой гибкостью, позволяя создавать программу с применением разнообразных альтернативных подходов.
3. Процедурное программирование слишком универсально и разнообразно по используемым техническим решениям, чтобы эффективно использоваться для написания прикладных программ!??


МАТЕМАТИЧЕСКОЕ ОБЕСПЕЧЕНИЕ ЭВМ
Н. Вирт
АЛГОРИТМЫ + СТРУКТУРЫ ДАННЫХ = ПРОГРАММЫ


Классика программирования
Никлаус Вирт
Алгоритмы и структуры данных
BHV


Н. Вирт
Алгоритмы и структуры данных
Издательство «Мир»


"Object-oriented programming
is an exceptionally bad idea
which could only have
originated in California."
-- Edsger Dijkstra

Список источников информации по данной теме

1. [Википедия] Статическая типизация
   https://ru.wikipedia.org/wiki/Статическая_типизация
2. Статическая и динамическая типизация
   https://habr.com/ru/post/308484/
3. Стандартная библиотека языка Си
   https://ru.wikipedia.org/wiki/Стандартная_библиотека_языка_Си
4. Легалов А.И. Разнорукое программирование
   http://softcraft.ru/paradigm/dhp/
5. Цикритзис Д., Лоховски Ф. Модели данных. Пер. с англ. - М.: Финансы и статистика, 1985. - 344 с.
6. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е изд./Пер. с англ. - М.: "Издательства Бином", СПб: "Невский диалект", 1998 г. - 560 с., ил.

Вопросы для обсуждения на семинаре

1. Основные виды памяти, используемые при статической типизации.
2. Избыточность статической типизации.
3. Трансформация статической типизации в бестиповую.
4. Агрегаты. Методы агрегирования.
5. Обобщения. Методы формирования обобщений.