Язык алгоритмических суперпозиций ASDIEL.

(Руководство программиста)

α-версия

17 января 2002 г.

Содержание

1 Введение

Пакет программ ASDIEL предназначен для построения и исследования алгоритмов интеллектуального анализа данных. Аббревиатура ASDIEL означает Algorithmic Superpositions Description and Investigation Environment and Language и переводится как «Язык и среда для описания и исследования алгоритмических суперпозиций».

Исходным материалом для построения процедур интеллектуального анализа данных в ASDIEL являются библиотеки методов.

Библиотеки могут пополняться несколькими способами. Первый способ состоит в определении нового метода как наследника класса `CMethod` и требует перекомпиляции ядра (точнее, только его линковки). Этот способ описан в разделе 2. Второй способ основан на использовании интерфейсных функций ядра, описанных в разделе 4. В этом случае методы реализуются либо в виде отдельной динамической библиотеки, либо в виде составной части прикладной программы, либо как часть ядра. Перекомпилировать ядро требуется только в третьем случае.

Наравне с алгоритмами в суперпозицию включаются обычные арифметические выражения. Библиотеку элементарных функций, используемых в выражениях, также можно расширить. Способ реализации дополнительных функций описан в разделе 3.

Ядро ASDIEL реализовано в виде библиотеки функций и не имеет ни пользовательского интерфейса, ни развитой системы ввода-вывода. Задача взаимодействия с пользователем полностью переложена на оболочку ASDIEL. Раздел 4 предназначен для разработчика оболочки.

Те дополнения, которые требуют перекомпиляции исходного кода, должны быть реализованы на языке C++. Компилятором может быть либо `Microsoft Visual C++` не ниже 2.0, либо `GNU C++` под `UNIX` или `MS DOS`.

2 Методы

С точки зрения пользователя языка *метод* — это совокупность алгоритмов с общим набором параметров. Каждый *алгоритм* задаётся своими входными и выходными аргументами. Работа алгоритма состоит в том, чтобы считать данные из входных аргументов, выполнить некоторые вычисления и записать результат в выходные аргументы. *Аргументы* алгоритма — это матрицы данных, в общем случае произвольной размерности. *Параметры* метода — это локальные переменные, доступные для чтения и записи всем алгоритмам метода и используемые для передачи данных между алгоритмами.

С точки зрения разработчика метод — это класс, выведенный путём наследования из класса `CMethod`. Дочерний класс реализует вычислительные функции алгоритмов. В то время как пользователь языка ASDIEL работает с алгоритмом как с

«чёрным ящиком», имея доступ лишь к его входам, выходам и параметрам, разработчик метода, наоборот, реализует «начинку» чёрного ящика, не зная в точности, какие данные будут подаваться на вход алгоритма, и в каких задачах этот метод будет использоваться.

Разработчик определяет структуру метода, а именно:

- количество алгоритмов и их имена,
- количество аргументов у каждого алгоритма и их размерности,
- дополнительные ограничения на размерности аргументов, такие как «число столбцов во входном аргументе алгоритма `calc` должно совпадать с числом столбцов во входном аргументе алгоритма `tune`»,
- количество параметров метода и их имена.

Пользователь метода обязан строго придерживаться данных требований, чтобы правильно применять его в своих задачах.

Для добавления нового метода в библиотеку ASDIEL разработчик должен выполнить следующие шаги.

1. Вывести путём наследования новый класс из класса `CMethod`. Переопределить его виртуальные функции, отвечающие за работу алгоритмов.
2. Определить класс-регистратор метода и зарегистрировать метод в библиотеке ASDIEL.
3. Исходный код метода и класса-регистратора поместить в отдельный файл и добавить его к проекту ASDIEL. Скомпилировать новый файл и собрать (`link`) проект.

Проиллюстрируем это на примере реализации очень простого метода — нормировки признаков. Следующий параграф взят из документации по библиотеке методов ASDIEL.

2.1 Пример: метод нормировки признаков

Метод применяется для образования новых признаков $F_0^* = \{f_1^*, \ldots, f_n^*\}$ путём нормировки заданных признаков $F_0 = \{f_1, \ldots, f_n\}$. Метод имеет три алгоритма, форма описания которых должна соответствовать следующему шаблону:

`USE F[S], P[F];`
`METHOD Norm;`
`tune` S_t `|` F_0`;`
`calc` S_c `|` F_0 `->` S_c `|` F_0^*`;`
`save ->` F_0 `|` $P\{f_{\min}, f_{\max}\}$`;`

Алгоритм настройки `tune` вычисляет максимальные и минимальные значения признаков f_1, f_2, …, f_n на объектах поднабора S_t. Эти значения запоминаются

как параметры метода `@min@1`, `@max@1`, ..., `@min@`n, `@max@`n, где n равно мощности поднаборов F_0 и F_0^*.

Алгоритм вычисления `calc` для всех объектов $s \in S_c$ и всех $i = 1, \ldots, n$ вычисляет значения $f_i^*(s)$ по формуле

$$f_i^*(s) = \begin{cases} \dfrac{f_i(s) - \texttt{@min@}}{\texttt{@min@}i - \texttt{@min@}i} & \text{если } \texttt{@DoShift} = 1 \text{ и } \texttt{@DoScale} = 1; \\ f_i(s) - \texttt{@min@}i & \text{если } \texttt{@DoShift} = 1 \text{ и } \texttt{@DoScale} = 0; \\ \dfrac{f_i(s)}{\max\{|\texttt{@min@}i|, |\texttt{@min@}i|\}} & \text{если } \texttt{@DoShift} = 0 \text{ и } \texttt{@DoScale} = 1; \\ f_i(s) & \text{если } \texttt{@DoShift} = 0 \text{ и } \texttt{@DoScale} = 0; \end{cases}$$

Метод имеет управляющие параметры, `@DoShift` и `@DoScale` типа `bool`. Первый параметр разрешает сдвиг, второй — масштабирование нормируемых значений.

Алгоритм записи параметров `save` заносит минимальные и максимальные значения признаков из F_0 в массив $P \times F$.

2.2 Определение класса `MNorm`

Для реализации метода выведем из класса `CMethod` класс `MNorm`. В новом классе переопределим виртуальные функции инициализации метода `InitMethod`, расчёта выходного подмассива `Run` и расчёта одной ячейки в выходном подмассиве `RunValue`.

```
class MNorm : public CMethod {
public:
  MNorm () {}
  ~MNorm () {}
  void InitMethod ();
  void Run (int iAlgor);
  void RunValue (int iAlgor, int iArgum, const Index* ids, CDatum& result);
private:
  void normalize (CDatum& result, int j);
};
```

2.3 Регистрация метода

Регистрация нужна для того, чтобы сообщить ядру ASDIEL всю необходимую информацию о методе: имя, количество алгоритмов и их имена, структуру каждого алгоритма, ограничения на размерности аргументов.

Сначала необходимо определить функцию создания нового экземпляра метода:

```
CMethod* MNormCreate ()  {
  return new MNorm;
}
```

Затем определяется класс-регистратор метода с единственным членом — конструктором, и тут же создаётся экземпляр этой структуры.

```
struct RNorm {
  RNorm () {
    CKernel::RegMethod ("Norm", 3, 3, &MNormCreate);
    CKernel::RegAlgorithm ("Tune 0:St|F");
    CKernel::RegAlgorithm ("Calc 1:Sc|F -> 2:Sc|F");
    CKernel::RegAlgorithm ("Save -> 3:F|{min,max}");
  }
} RegNorm;
```

Для регистрации метода и алгоритмов используются статические функции ядра `RegMethod` и `RegAlgorithm`.

`void CKernel::RegMethod (const char* name, int nalgors, int max_argum, FMethodCreate fcreate)`
— функция регистрации метода, где:
`name` — имя метода;
`nalgors` — число алгоритмов метода;
`max_argum` — максимальный порядковый номер аргумента;
`fcreate` — указатель на функцию создания экземпляра метода.

`void CKernel::RegAlgorithm (const char* descr);`
— функция регистрации алгоритма, где:
`descr` — шаблон алгоритма.

Шаблон алгоритма описывает структуру алгоритма аналогично тому, как это делается в документации по библиотеке методов (см. пример на стр. 3).

Перед каждым аргументом через двоеточие должен быть указан его внутренний номер. Внутренние номера используются разработчиком метода при обращении к аргументам. Нумерация аргументов явная и сквозная для всех алгоритмов метода. Нумерация алгоритмов неявная — внутренние номера присваиваются в порядке регистрации, начиная с нуля.

Размерность аргумента может быть описана двумя способами: идентификатором или перечислением.

В первом случае все размерности с одинаковыми именами во всех аргументах-подмассивах должны иметь один и тот же размер. Например, согласно шаблону

```
Calc 1:Sc|F -> 2:Sc|F
```

в методе `Norm` входная и выходная матрицы алгоритма `Calc` должны быть одинакового размера.

Во втором случае поднабор, соответствующий данной размерности, должен состоять из фиксированного числа элементов. Например, согласно шаблону

```
Save -> 3:F|{min,max}
```

выходная матрица алгоритма `Save` должна иметь ровно 2 столбца.

Знак `?` перед аргументом означает, что он необязательный и может быть опущен.

Знак `?` перед описанием размерности означает, что данная размерность, и все последующие, необязательные. Например, подмассив, описанный как `S|S|?{dist}`, может быть либо квадратной матрицей, либо трёхмерным подмассивом размера $|S| \times |S| \times 1$.

Знак `?` перед элементом перечисления означает, что все элементы, начиная с данного, необязательные. Например, матрица `F|{a,b,?weight}` может состоять либо из двух, либо из трёх столбцов.

2.4 Абстрактный метод `CMethod`

Для создания нового метода необходимо вывести производный класс из абстрактного класса `CMethod`:

```
class CMethod {
public:
  // виртуальные функции, переопределяемые разработчиком метода
  virtual void InitMetod ();
  virtual void InitAlgorithm (int iAlgor);
  virtual void Run (int iAlgor);
  virtual void RunValue (int iAlgor,
                int iArgum, const Index* ids, CDatum& result);
  // доступ к аргументам алгоритмов
  Index DimSize (int iArgum, Dim d);
  void Get (int iArgum, CDatum &dat, Index i0);
  void Get (int iArgum, CDatum &dat, Index i0, Index i1);
  void Get (int iArgum, CDatum &dat, Index i0, Index i1, Index i2);
  void Put (int iArgum, CDatum &dat, Index i0);
  void Put (int iArgum, CDatum &dat, Index i0, Index i1);
  void Put (int iArgum, CDatum &dat, Index i0, Index i1, Index i2);
  int GetMatrix (int iArgum, CMatrix& A, double err_value=0.0);
  int PutMatrix (int iArgum, CMatrix& A);
  CSubarray* Argument (int iArgum);
  // доступ к параметрам метода
  CParamTable Param;
  // функции, информирующие оболочку
  int progress;
  int progress_max;
  Bool CallGui (double *run_seconds=0);
  char* SetProgressMessage (const char* messkey=0);
};
```

В производном классе разрешается определять любые новые данные-члены и функции-члены. Однако *данные-члены нельзя использовать ни для хранения параметров метода, ни для передачи информации из одного алгоритма в другой. Для этих целей служит исключительно вектор параметров* `Param`.

Для работы с аргументами алгоритмов в большинстве случаев хватает трёх функций: `DimSize`, `Get` и `Put`. Функции `Get` и `Put` передают значение через структуру типа `CDatum`, которая может содержать значение любого типа (см. 2.7). Вся работа с параметрами метода происходит через вектор параметров `Param` (см. 2.9), также состоящий из элементов типа `CDatum`.

Ниже перечислены эти и другие члены класса `CMethod`, которыми должен пользоваться разработчик производного класса.

`Index DimSize (int iArgum, Dim d)`

Возвращает размер `iArgum`-го аргумента по `d`-ой размерности. Сквозная нумерация аргументов устанавливается при регистрации метода (стр. 4). Аргументы и размерности нумеруются, начиная с нуля.

`void Get (int iArgum, CDatum &dat, Index` i_0`, Index` i_1`)`

Возвращает в `dat` значение ячейки (i_0, i_1) в `iArgum`-ом аргументе. Индекс i_d может пробегать значения от 0 до `DimSize(iArgum,`d`)-1`. Имеются 4 версии функции `Get` для аргументов размерностей от 1 до 4. Для работы с аргументами большей размерности придётся пользоваться функцией `Argument` и членами класса `CSubarray`.

`void Put (int iArgum, CDatum &dat, Index` i_0`, Index` i_1`)`

Функция, обратная `Get`. Записывает значение `dat` в выходной аргумент.

`int GetMatrix (int iArgum, CMatrix& A, double err_value=0.0)`

Предоставляет альтернативный по отношению к `Get` способ считывания данных из `iArgum`-го аргумента. Целиком загружает аргумент в двумерную числовую матрицу `A` (см. 2.8). Если в аргументе встречаются нечисловые, ошибочные или пропущенные значения, на их место в матрице `A` записывается `err_value`. Возвращает число ошибок.

`int PutMatrix (int iArgum, CMatrix& A)`

Предоставляет альтернативный по отношению к `Put` способ записи данных в `iArgum`-ый аргумент. Целиком переписывает двумерную числовую матрицу `A` в соответствующий подмассив (см. 2.8). Подмассив обязан быть двумерным, и его размеры должны совпадать с размерами матрицы `A`. Возвращает число ошибок, возникших при записи данных.

`CSubarray* Argument (int iArgum)`

Возвращает `iArgum`-ый аргумент метода (нумерация сквозная, заданная при регистрации метода, стр. 4). Применяется только в исключительных случаях, когда для работы с аргументом не достаточно функций `DimSize`, `Get` и `Put`. Функция возвращает 0 в двух случаях: либо аргумент необязательный и опущен в

описании метода; либо соответствующий алгоритм ещё не описан (выполнение программы ещё не дошло до команды описания алгоритма).

`CParamTable Param`

Вектор параметров. Ядро ASDIEL должно иметь доступ ко *всем* параметрам метода. Поэтому разработчику метода *запрещается* хранить параметры как данные-члены. Единственный корректный способ работы с параметрами метода — через вектор параметров `Param`. Порядок работы с вектором параметров описан в разделе 2.9.

`int progress, progress_max`

Переменная `progress` представляет собой счётчик проделанной работы, который может принимать значения от 0 до `progress_max`. Разработчик метода устанавливает эти переменные в теле функции `Run` (см. ниже) для информирования ядра ASDIEL о текущем состоянии расчёта.

`char* SetProgressMessage (const char* messkey=0)`

Устанавливает код сообщения, выводимого оболочкой во время выполнения функции `Run`. Текст сообщения, соответствующий коду `messkey`, может быть записан в файлах `sdl_eng.err` и `sdl_rus.err`.

`Bool CallGui (double *run_seconds=0)`

Должна периодически (несколько раз в секунду) вызываться в теле функции `Run` для временной передачи управления оболочке ASDIEL. При этом `CallGui` выполняет следующую работу.

- Отображает текущее состояние расчёта, используя `progress`, и выводит последнее сообщение, заданное вызовом `SetProgressMessage`.
- В переменную по указателю `run_seconds` записывает время выполнения текущей команды в секундах. Это значение можно использовать для прерывания итерационного процесса.
- Возвращает `1`, если выполнение алгоритма можно продолжить, и `0`, если выполнение прервано пользователем. В последнем случае разработчик метода должен обеспечить немедленное завершение алгоритма с корректным освобождением выделенной памяти.

2.5 Переопределение виртуальных функций. Общий случай

В производном классе разработчик метода должен определить конструктор, деструктор и 4 виртуальные функции:

- `InitMetod` — инициализация метода,
- `InitAlgorithm` — инициализация алгоритма,

- `Run` — расчёт выходного подмассива,
- `RunValue` — расчёт одной ячейки в выходном подмассиве.

Реализация может не переопределять виртуальную функцию, если она не нужна; например функция `InitAlgorithm` часто опускается.

Вернёмся к нашему примеру с методом нормировки.

`void InitMethod ()`

Инициализация метода происходит во время выполнения команды `METHOD`. При этом вызывается виртуальная функция-член `InitMethod`. Обычно она устанавливает значения параметров «по умолчанию».

```
void MNorm::InitMethod () {
  Param["DoShift"] = long(0);
  Param["DoScale"] = long(1);
}
```

`void Run (int iAlgor)`

Функция выполняет `iAlgor`-ый алгоритм метода (нумерация алгоритмов начинается с нуля). Она должна осуществлять полный перерасчёт всех значений во всех выходных подмассивах данного алгоритма.

```
void MNorm::Run (int iAlgor) {
  CDatum dat;
  double min, max, Xij;
  int nF, nS, i, j;
  switch (iAlgor) {
    case 0:   // Алгоритм настройки
      nF = DimSize (0,1);
      nS = DimSize (0,0);
      for (j=0; j<nF; j++) {
        min = max = 0.0;
        for (i=0; i<nS; i++) {
          Get (0, dat, i, j);
          Xij = double (dat);
          if (i==0 || Xij<min) min = Xij;
          if (i==0 || Xij>max) max = Xij;
        }
        Param["min"][j] = min;
        Param["max"][j] = max;
      }
      break;
    case 1:   // Алгоритм вычисления
      nF = DimSize (1,1);
      nS = DimSize (1,0);
      for (j=0; j<nF; j++)
        for (i=0; i<nS; i++) {
```

```
            Get (1, dat, i, j);
            normalize (dat, j);
            Put (2, dat, i, j);
          }
        break;
      case 2:   // Алгоритм сохранения параметров
        nF = DimSize (3,0);
        for (j=0; j<nF; j++) {
          Put (3, Param["min"][j], j, 0);
          Put (3, Param["max"][j], j, 1);
        }
        break;
    }
  }
```

`void RunValue (int iAlgor, int iArgum, const Index* ids, CDatum& result)`

Функция выполняет `iAlgor`-ый алгоритм метода. В отличие от `Run` она вычисляет только одну ячейку в `iArgum`-ом выходном подмассиве. Координаты (индексы) ячейки передаются в массиве `ids`. Функция должна записать результат вычислений в структуру `result`. Записывать вычисленное значение в выходной массив с помощью функции `Put` не нужно.

```
void MNorm::RunValue (int iAlgor, int iArgum,
                      const Index* ids, CDatum& result) {
  switch (iAlgor) {
    //case 0:    Алгоритм настройки никогда не вызывается
    case 1:   // Алгоритм вычисления
      Get (1, result, ids[0], ids[1]);
      normalize (result, ids[1]);
      break;
    case 2:   // Алгоритм сохранения параметров
      if (ids[1]==0) result = Param["min"][ids[0]];
      if (ids[1]==1) result = Param["max"][ids[0]];
      break;
  }
}
```

Заметим, что реализация ветви `case 0`, соответствующей алгоритму настройки, в данном случае полностью бессмысленна, так как он не имеет выходных аргументов, следовательно для него функция `RunValue` никогда не будет вызвана.

Функции расчёта `Run` и `RunValue` должны производить идентичные вычисления. Смысл их дублирования в том, чтобы оптимизировать расход времени и памяти. В зависимости от имеющихся ресурсов выбирается тот или иной способ вычисления. Использование функции `RunValue` менее эффективно по времени, но зато не требует памяти для хранения промежуточных данных. Значение, выданное функцией `RunValue`, может быть «забыто» сразу после его использования.

`void normalize (CDatum& result, int j)`

Функция-член `normalize` нормирует значение j-го признака, передаваемое ей в ячейке `result`.

```
void MNorm::normalize (CDatum& result, int j) {
  long is_shifted = Param["DoShift"];
  long is_scaled = Param["DoScale"];
  double min = Param["min"][j];
  double max = Param["max"][j];
  double Max = max>-min ? max : -min;
  if (!result.IsNumber()) return;
  if (is_shifted && is_scaled && max>min)
    result = (double(result)-min) / (max-min);
  else if (is_shifted)
    result = double(result) - min;
  else if (is_scaled && Max!=0.0)
    result = double(result) / Max;
}
```

2.6 Переопределение виртуальных функций. Методы обучения по прецедентам

Методы обучения по прецедентам удовлетворяют двум дополнительным ограничениям. Во-первых, они имеют стандартный набор алгоритмов:

- `calc` — вычисление (имеется всегда),
- `tune` — настройка (имеется почти всегда),
- `more` — дополнительная настройка (у некоторых методов),
- `save` — запись параметров (у некоторых методов),
- `load` — загрузка параметров (у некоторых методов),
- `init` — инициализация (в редких случаях).

Во-вторых, из этих 6 алгоритмов только `calc` и `save` могут иметь выходные подмассивы со свободными поднаборами или вычислимыми признаками (именно в этих случаях ядро вызывает функцию `RunValue` вместо `Run`). Связано это с тем, что по смыслу алгоритмы `tune`, `more`, `load` и `init` не вычисляют никакой выходной информации и предназначены для установки внутренних параметров метода.

Для реализации этого частного случая разработчику предоставляется возможность вывести собственный класс из класса `CTunableMethod`:

```
class CTunableMethod : public CMethod {
public:
  virtual void Calc ();
  virtual void Tune ();
  virtual void More ();
```

```
  virtual void Save ();
  virtual void Load ();
  virtual void Init ();
  virtual void CalcValue (int iArgum, const Index* ids, CDatum& result);
  virtual void SaveValue (int iArgum, const Index* ids, CDatum& result);
};
```

Теперь вместо `Run` и `RunValue` разработчик переопределяет некоторые из перечисленных 8 функций. Первые 6 функций являются аналогами `Run`, и к ним в полной мере относятся все замечания, касавшиеся реализации функции `Run`. Точно так же, функции `CalcValue` и `SaveValue` являются аналогами `RunValue`. Единственное отличие заключается в том, что теперь вместо передачи номера алгоритма через параметр `iArgum` ядро сразу вызывает нужную функцию.

Переопределять нужно те и только те виртуальные функции, которые соответствуют зарегистрированным алгоритмам метода. Если метод по замыслу разработчика должен обладать какими-то алгоритмами кроме перечисленных шести, он уже не сможет воспользоваться классом `CTunableMethod` в качестве базового.

2.7 Значения типа `CDatum`

Объекты класса `CDatum` предназначены для хранения значений произвольных типов. В частности, они используются для хранения параметров и передачи данных при работе с аргументами методов.

Класс `CDatum` совместим по присваиванию с тимами `long` и `double` языка C++ и основными внутренними типами ASDIEL.

```
class CDatum {
public:
  DatumType Type : 8;    // Тэг типа данных
  union {
    ErrorCode Code;      // Код ошибки
    long Int;            // Все целочисленные типы
    double Real;         // Все вещественные типы
    time_t DateTime;     // Дата и Время
    char* Str;           // Строка
    COutStream *Stream;  // Поток вывода
    CSubset *Subset;     // Поднабор
    CSubspace *Subspace; // Подпространсво (подмассив)
    CParamTable *ParTab; // Таблица параметров
    CDatumArray *DatArr; // Массив датумов
  };
  CDatum ();
  CDatum (CDatum& right);
  ~CDatum ();
  // преобразования к другим типам
  operator long () const;
```

```
    operator double () const;
    operator char* ();
    operator COutStream& () const;
    // преобразования в дату и из даты
    time_t GetDateTime ();
    void SetDateTime (time_t timer);
    // присваивания значений других типов
    void Copy (CDatum& right);    // копируется содержимое, а не ссылка
    void Error (ErrorCode code);  // присваивание кода ошибки
    CDatum& operator= (CDatum& right);
    CDatum& operator= (double right);
    CDatum& operator= (long right);
    CDatum& operator= (char* right);
    CDatum& operator= (CSubset* right);
    CDatum& operator= (COutStream* os);
    // работа с таблицей параметров
    void CreateParamTable (Index block_size);
    CDatum& operator[] (Index i);          // создать если не найден
    CDatum& operator[] (const char* name); // создать если не найден
    CDatum& GetValue (Index i);            // ошибка  если не найден
    CDatum& GetValue (const char* name);   // ошибка  если не найден
    // работа с массивом датумов
    void CreateDatumArray (Index block_size);
    CDatum& DatArrElem (Index i); // значение элемента в массиве датумов
    // проверки
    Bool IsEmpty ();   // является ли пустым значением (EMPTY) ?
    Bool IsNumber ();  // является ли числом?
    Bool IsTime ();    // является ли датой и/или временем?
};
```

Поле `Type` может принимать значения, перечисленные в таблице 1

Функция `CDatum::IsNumber()` возвращает `1` тогда и только тогда, когда значение может быть корректно преобразовано к типу `double`.

Функция `CDatum::IsEmpty()` возвращает `1` тогда и только тогда, когда значение пусто или содержит код ошибки. В обоих случаях `Type` равно `DT_ERROR`.

Передача значений в функциях `CMethod::Get` и `CMethod::Put` через структуру типа `CDatum` позволяет корректно обрабатывать матрицы, содержащие разнотипные, некорректные или пропущенные данные. Операторы преобразования типа `CDatum` к типам `long` и `double` по умолчанию возвращают нуль, если значение не является числовым, при этом не происходит никакой ошибки. Таким образом обнаружение и обработка пропусков в данных полностью возлагается на разработчика метода.

2.8 Числовые матрицы `CMatrix`

Кроме функций `Get` и `Put` разработчику метода предоставляется ещё один механизм считывания и записи аргументов, более подходящий для работы с числовыми дан-

DT_CODE=0	Код сообщения об ошибке, char*
DT_BOOL=1	булевский, 1 бит
DT_2BIT=2	2-битовый $\{0, 1, 2, 3\}$
DT_4BIT=3	4-битовый $\{0, 1, \ldots, 15\}$
DT_BYTE=4	байт
DT_SHORT=5	короткое целое (2 байта)
DT_LONG=6	длинное целое (4 байта)
DT_DATE=7	только дата
DT_TIME=8	только время
DT_DATETIME=9	дата и время
DT_SINGLE=10	вещественное одинарной точности (4 байта)
DT_DOUBLE=11	вещественное двойной точности (8 байт)
DT_STRING=12	строка
DT_STREAM=13	поток вывода
DT_DATARR=14	массив датумов
DT_DISTRIB=15	массив вещественных чисел (распределение)
DT_PARTAB=16	вектор параметров
DT_BASET=17	базовый набор
DT_SUBSET=18	поднабор
DT_SUBSPACE=19	подмассив
DT_ERROR=DT_CODE	синоним ошибки
DT_INT=DT_LONG	синоним самого большого из целых
DT_REAL=DT_DOUBLE	синоним самого большого из вещественных

Таблица 1: Допустимые типы данных CDatum

ными (см. 2.4):

```
int GetMatrix (int iArgum, CMatrix& A, double err_value=0.0)
int PutMatrix (int iArgum, CMatrix& A)
```

Класс CMatrix реализует двумерную матрицу элементов типа double:

```
class CMatrix {
public:
  CMatrix (Index rows, Index cols=1);
  CMatrix (CMatrix& right);
  CMatrix ();
  ~CMatrix ();
  void Init (Index rows, Index cols);
  void AddColumn (Index cols=1);
  T& operator() (Index row, Index col=0);
  Index Rows();
  Index Cols();
  void Zeroize ();
```

```
  void Inverse (CMatrix& A);
  void Mult (CMatrix& A, CMatrix& B);
};
```

Функция `Init` инициализирует матрицу новыми размерами, уничтожая старое содержимое матрицы. Функция `AddColumn` добавляет в матрицу `cols` столбцов, не затрагивая предыдущих столбцов.

Оператор `operator()` предоставляет доступ к элементам матрицы. Строки и столбцы нумеруются, начиная с нуля. Причём как конструктор матрицы, так и оператор доступа имеют значения второго аргумента «по умолчанию» такие, что объектом типа `CMatrix` можно пользоваться как вектор-столбцом.

Функция `Zeroize` обнуляет матрицу. Функция `Inverse` вычисляет псевдообратную матрицу $(A^TA)^{-1}A^T$, причём A не обязана быть квадратной матрицей. Функция `Mult` вычисляет матричное произведение AB.

2.9 Параметры метода

Параметры метода хранятся в векторе параметров `Param`. Вектор параметров является членом класса `CMethod` и достаётся по наследству любому методу, выводимому из этого класса. Вектор параметров состоит из значений типа `CDatum`, каждое из которых может иметь имя. Доступ к элементу вектора возможен как по порядковому номеру, так и по имени параметра.

Тип `CDatum` может содержать любые значения, включая вектор параметров. Это позволяет организовать вектор параметров иерархически. Число уровней вложенности не ограничено.

Обращение к параметру довольно компактно как в C++, так и в ASDIEL:

```
long a = Param["DoShift"];        // на C++
a = @DoShift;                     !  на SDL
Param["max"][12] = double(150);   // на C++
@max@12 = 150;                    !  на SDL
```

При обращении к параметру по несуществующему имени он создаётся и добавляется в конец вектора. При обращении к параметру по номеру, выходящему за пределы вектора, в конец вектора добавляется нужное количество параметров; причём все они не наделяются именами. Это правило распространяется как на C++, так и на ASDIEL.

Основное назначение вектора параметров — служить для передачи информации из одного алгоритма в другой. Ещё раз подчеркнём, что для этой цели нельзя использовать другие данные-члены класса, производного от `CMethod`. Более того, для этого в общем случае не подходят также и массивы. Эти запреты необходимы для того, чтобы алгоритмы могли работать независимо друг от друга, и ограничения на последовательность их выполнения были менее жёсткими. Тем самым мы предоставляем пользователю больше свободы в выборе способа настройки метода.

Например, если бы работа алгоритма `calc` явным образом зависела от выполняемого перед ним `tune`, пользователь языка ASDIEL был бы вынужден всегда вызывать эти алгоритмы в паре — сначала `tune`, затем `calc`. Если же `calc` зависит только от вектора параметров, то становится неважным, как был получен этот вектор. Его можно было настроить с помощью того же `tune`, либо загрузить из массива алгоритмом `load`, либо непосредственно задать в программе с помощью команд присваивания.

2.10 Исключительные ситуации

Исключительные ситуации в методах могут сигнализировать о неверно сформированных аргументах, ошибках в вычислениях, отсутствии сходимости алгоритма, и т. д. Порождение исключительных ситуаций, связанных с методом, возлагается на разработчика метода, однако их обработка осуществляется ядром ASDIEL.

Исключительная ситуация порождается оператором `throw`, которому передаётся экземпляр класса `CMethodError`, например:

```
if (min==max)
  throw CMethodError ("Norm_CANT_TUNE", this);
```

Конструктор класса `CMethodError` принимает на входе два параметра: код сообщения и указатель на метод, в котором возникла ошибка. Коду сообщения может соответствовать текст сообщения в файлах `sdl_eng.err` и `sdl_rus.err`.

2.11 Отладочная печать

Отладочная печать производится в специализированный поток — объект класса `COutStream`. Поток может быть связан с текстовым файлом, консолью, окном оболочки или другим приложением. Кроме того, он может игнорироваться или распараллеливаться в несколько потоков. Во всех случаях класс `COutStream` работает одинаково, так что разработчику метода нет необходимости знать, куда реально производится печать. Также разработчику нет необходимости перед каждым оператором печати проверять, выводить отладочную информацию или нет. Если он оставит команды вывода в коде, пользователь языка ASDIEL всегда сможет переопределить поток вывода как нулевой, если отладочная информация его не интересует.

Значения стандартных типов выводятся в поток `COutStream` операцией `«` точно так же, как и для стандартного типа `ostream`.

Поток можно передать методу через параметр. Для этой цели все методы используют параметр с именем `log`. Для получения потока, скажем, в теле функции `Run`, достаточно вставить строку:

```
COutStream& os = Param["log"];
```

и далее пользоваться ссылкой `os`:

```
Get (0,dat,i,j);
os « "A[" « i « "," « j « "] = " « dat;
```

Для приверженцев функции `printf` оставлена возможность форматного вывода:

```
os « format ("A[%3d,%3d] = %6.3f\n", i, j, double(dat));
```

Длина строки, формируемой и возвращаемой функцией `format`, не должна превышать 4 Кбайт, в противном случае произойдёт переполнение внутреннего буфера и результат операции будет непредсказуем.

2.12 Второй пример: метод наименьших квадратов

Метод наименьших квадратов применяется для образования нового признака y, линейно зависящего от заданной совокупности признаков $F_0 = \{f_1, \ldots, f_n\}$ и аппроксимирующего заданный целевой признак g. Метод имеет алгоритмы `tune` и `calc`, форма описания которых должна соответствовать следующему шаблону:

```
USE F[S], P[F];
METHOD LeastSquares;
tune S_t | F_0, S_t | {g};
calc S_c | F_0 -> S_c | {y};
save -> F_0 | P{a};
load F_0 | P{a};
```

Алгоритм вычисления `calc` для всех объектов $s \in S_c$ определяет значение нового признака $y \in F$ как линейную комбинацию всех признаков поднабора F_0:

$$y(s) = \sum_{i=1}^{n} a_i f_i(s),$$

где $a_1, \ldots, a_n$ — действительные числа, внутренние параметры метода, которые должны быть настроены к моменту вычисления.

Алгоритм настройки `tune` настраивает коэффициенты линейной зависимости a_i из условия аппроксимации целевого признака $g \in F$ на заданной обучающей выборке объектов $S_t \subseteq S$. Для этого решается задача минимизации функционала

$$Q(a_1, \ldots, a_n) = \sum_{s \in S_t} (y(s) - g(s))^2.$$

Алгоритм записи параметров `save` переписывает параметров $(a_1, \ldots, a_n)$ в заданный подмассив размером $n \times 1$.

Алгоритм загрузки параметров `load` выполняет обратную операцию — переписывает заданный подмассив размером $n \times 1$ в вектор параметров $(a_1, \ldots, a_n)$.

Параметры метода $a_1, \ldots, a_n$ доступны в ASDIEL как `@a@1`, ..., `@a@`n а в C++, соответственно, как `Param["a"][1]`, ..., `Param["a"][`n`]`.

Ниже целиком приводится текст исходного файла `MLsm.cpp`:

```
#include "Kernel.h"  // ядро SDL
#include "Analyt.h"  // файл, в котором определяется CMatrix

// LeastSquares - Метод наименьших квадратов
class MLeastSquares : public CTunableMethod {
public:
  MLeastSquares () {}
  ~MLeastSquares () {}
  void Tune ();
  void Calc ();
  void CalcValue (int iArgum, const Index* ids, CDatum& result);
  void Save ();
  void SaveValue (int iArgum, const Index* ids, CDatum& result);
  void Load ();
private:
  double calc_linear_combination (int i);
};

// создание нового экземпляра метода
CMethod* MLeastSquaresCreate ()  {
  return new MLeastSquares;
}

// вызов регистратора
struct RLeastSquares {
  RLeastSquares () {
    CKernel::RegMethod ("LeastSquares", 4, 7, &MLeastSquaresCreate);
    CKernel::RegAlgorithm ("Tune 0:St|F, 1:St|{goal}, ?2:St|{weight}");
    CKernel::RegAlgorithm ("Calc 3:Sc|F -> 4:Sc|{result}");
    CKernel::RegAlgorithm ("Save -> 5:F|{coeff}");
    CKernel::RegAlgorithm ("Load 6:F|{coeff}");
}
} RegLeastSquares;

// запуск алгоритма TUNE для полного пересчёта
void MLeastSquares::Tune () {
  COutStream &out = Param["log"];
  CMatrix A, g;
  GetMatrix(0,A);
  GetMatrix(1,g);
  CMatrix Ainv, coeff;
  Ainv.Inverse (A);
  out « "Inverse of A matrix Ainv:\n" « Ainv;
  coeff.Mult (Ainv, g);
  int nF = DimSize (0,1);
  for (int j=0; j<nF; j++) Param["a"][j] = coeff(j);
}
```

```
// запуск алгоритма CALC для полного пересчёта
void MLeastSquares::Calc () {
  CDatum dat;
  int nS = DimSize (3,0);
  for (int i=0; i<nS; i++) {
    dat = calc_linear_combination (i);
    Put (3, dat, i, 0);
  }
}

// запуск алгоритма CALC для расчёта значения, iArgum==3 всегда!
void MLeastSquares::CalcValue (int iArgum, const Index* ids, CDatum& result){
  result = calc_linear_combination (ids[0]);
}

// запуск алгоритма SAVE для полного пересчёта
void MLeastSquares::Save () {
  int nF = DimSize (4,0);
  for (int i=0; i<nF; i++) Put (4, Param["a"][i], i, 0);
}

// запуск алгоритма SAVE для расчёта значения, iArgum==4 всегда!
void MLeastSquares::SaveValue (int iArgum, const Index* ids, CDatum& result){
  result = Param["a"][ids[0]];
}

// запуск алгоритма LOAD для полного пересчёта
void MLeastSquares::Load () {
  int nF = DimSize (5,0);
  for (int i=0; i<nF; i++) Get (5, Param["a"][i], i, 0);
}

// скалярное произведение i-ой строки Argument(2) и вектора Param["a"]
double MLeastSquares::calc_linear_combination (int i) {
  int j, nF = DimSize (0,1);
  double sum = 0.0;
  CDatum dat;
  for (j=0; j<nF; j++) {
    Get (2, dat, i, j);
    sum += double (Param["a"][j]) * double (dat);
  }
  return sum;
}
```

3 Функции

Чтобы добавить в ASDIEL новую функцию, необходимо выполнить следующие шаги:

1. Описать новую функцию в файле `Funcs.cpp`.
2. Зарегистрировать её в ядре ASDIEL, добавив одну строку в том же файле.
3. Скомпилировать файл `Funcs.cpp` и собрать (`link`) проект.

3.1 Определение функции

Все функции ASDIEL имеют переменное число аргументов произвольного типа. Каждой из них соответствует ровно одна функция на `C++`, определяемая в файле `Funcs.cpp`. Все `C++`-функции имеют один и тот же прототип

```
void Fn (CDatum** arg);
```

Массив указателей `arg` устроен следующим образом. Датум `*arg[0]` предназначен для записи результата. Датумы `*arg[1]`, `*arg[2]`, и т.д. содержат первый, второй, и т.д. аргументы соответственно. Если число аргументов, переданных функции равно n, то указатель `arg[`$n+1$`]` будет равен нулю. Этот факт используется при обработке списка аргументов у таких функций, как `if`, `min` и `max`.

Пример 1. Рассмотрим определение функции синуса. Она имеет один аргумент типа `double` и возвращает `double`.

```
// функция синуса
// double = double
void FnSin (CDatum** arg) {
  *arg[0] = sin (double(*arg[1]));
}
```

Пример 2. Условная функция `if` принимает произвольное число аргументов. Если значение 1-го аргумента может трактоваться как `true`, то возвращается значение 2-го аргумента. Иначе рассматривается 3-й аргумент, и если его значение `true`, то возвращается значение 4-го аргумента. И так далее. Если значения всех нечётных аргументов кроме последнего равны `false`, то результатом является последний аргумент.

```
// условная функция
// any = boolean any {boolean any} any
void FnIf (CDatum** arg) {
  int i;
  for (i=1; arg[i] && arg[i+1]; i+=2)
    if (double(*arg[i])>0) {
      arg[0]->Copy(*arg[i+1]); return;
    }
  if (arg[i])
    arg[0]->Copy(*arg[i]);
  else
    *arg[0] = long(0);
}
```

3.2 Регистрация функции

Чтобы зарегистрировать новую функцию, необходимо вызвать до её первого использования специальный регистратор, передав ему имя `name` и адрес `oper` новой функции:

```
void CKernel::RegistrateOperator (const char* name, FOperator oper);
```

Указатель на функцию определён как

```
typedef void (*FOperator) (CDatum**);
```

Все функции ASDIEL, реализованные в `Funcs.cpp`, регистрируются внутри функции-члена `registrate_functions` класса `CKernel`, определяемой в том же файле. Например, функции `sin` и `if` регистрируются следующим образом:

```
RegistrateOperator ("sin", &FnSin);
RegistrateOperator ("if", &FnIf);
```

4 Интерфейс ядра ASDIEL

Интерфейс ядра ASDIEL предназначен для выполнения ASDIEL-программ из внешних приложений. Он реализован в виде динамической библиотеки и не является объектно-ориентированным.

Набор интерфейсных функций позволяет выполнять следующие действия:

- загрузить программу или несколько программ одновременно;
- выполнить программу целиком или пошагово;
- считывать и записывать значения из переменных и массивов;
- вычислять значения выражений и выполнять фрагменты программы, заданные тектовой строкой;
- определять новые методы и динамические библиотеки методов.

Интерфейс ядра ASDIEL достаточен для реализации таких приложений, как

- отладчик, позволяющий трассировать выполнение программ, просматривать и изменять значения переменных и массивов, а также выводить отладочную информацию о состоянии ядра, памяти и текущего метода;
- простейший интерпретатор, в диалговом режиме исполняющий команды и вычисляющий выражения;
- прикладная программа, использующая алгоритмические суперпозиции для решения конкретных прикладных задач;
- динамическая библиотека методов, подключаемая командой `LIBRARY`.

4.1 Инициализация и удаление ядра

`int SDL_Init (int argc, char *argv[], char *envp[], char* SDLinit)`

Создать ядро ASDIEL. Функция должна быть вызвана только один раз перед первым обращением к другим интерфейсным функциям ядра. На входе указывается список аргументов командной строки `argv`, список переменных среды `envp` и последовательность ASDIEL-команд `SDLinit`. Переменная среды `SDLDIR` должна указывать директорию, в которой был инсталлирован ASDIEL. Команды `SDLinit` будут выполнены перед чтением файла `init.sdl`. Обычно они используются для присваивания значений глобальным переменным `APPLICATION` (тип приложения: диалоговый интерпретатор, отладчик, прикладная программа и т.д.) и `KERLINK` (способ загрузки ядра: статический или динамический).

Возвращаемые значения:

0 — инициализация ядра прошла успешно;

1 — невозможно открыть файл `init.sdl`;

2 — не найдена переменная среды `SDLDIR`;

3 — код инициализации `SDLinit` содержит ошибку;

4 — невозможно открыть файл сообщений `*.err`;

5 — невозможно открыть файл `sdl_mem.log` для отладочной печати.

`int SDL_Done ()`

Удалить ядро ASDIEL. Функция должна быть вызвана только один раз по завершении работы с ядром.

Возвращаемые значения:

0 — удаление ядра прошло успешно;

1 — невозможно открыть файл `sdl_mem.log` для продолжения записи.

4.2 Работа с файлами программ

Ядро позволяет загрузить одновременно несколько программ, однако в каждый момент времени активной может быть только одна.

`int SDL_ProgramLoad (const char* filename)`

Загрузить программу из файла `filename`. Программа становится активной.

Возвращаемые значения:

0 — загрузка прошла успешо;

1 — файл не найден;

2 — файл пуст.

`int SDL_ProgramReLoad (const char* filename)`

Перезагрузить программу из файла `filename`. Программа становится активной.

Выполнение программы автоматически позиционируется на начало того раздела, в котором обнаружена первая модификация.

Возвращаемые значения:

0 — загрузка прошла успешно;

1 — файл не найден;

2 — файл пуст;

3 — изменений не обнаружено, перезагружать не было надобности;

4 — файл не был загружен ранее.

`int SDL_ProgramSetCurrent (const char* filename)`

Сделать ранее загруженную программу `filename` активной.

Возвращаемые значения:

0 — программа активизирована успешно;

1 — файл не был предварительно загружен.

`int SDL_ProgramSaveTuned (const char* filename)`

Записать настроенную программу в файл `filename`. В каждый метод добавляется блок установки параметров.

Возвращаемые значения:

0 — программа записана успешно;

1 — в программе остались ненастроенные разделы;

2 — невозможно сохранить раздел;

3 — пустое имя файла.

4.3 Выполнение активной программы

`int SDL_ProgramRun ()`

Выполнить активную программу целиком.

Возвращаемые значения:

0 — программа выполнена успешно;

2 — в программе обнаружена ошибка;

3 — выполнение программы прервано пользователем.

`int SDL_CommandRun ()`

Выполнить следующую команду активной программы.

Возвращаемые значения:

0 — команда выполнена успешно;

1 — следующей команды нет, программа завершена;

2 — в команде обнаружена ошибка;

3 — выполнение команды прервано пользователем.

`int SDL_CommandPosBegin ()`

Выдать позицию начала текущей команды в активной программе.

`int SDL_CommandPosEnd ()`

Выдать позицию конца текущей команды в активной программе.

`char* SDL_CommandLastMessage ()`

Выдать последнее сообщение об ошибке в активной программе.

`int SDL_CommandLastErrorPos(int* pos, int* len)`

Выдать начальную позицию `pos` и длину ошибочного участка `len` в последней выполненной команде активной программы.

Возвращаемые значения:

0 — последняя команда выполнилась с ошибкой;

1 — ошибки не было, при этом `len=0`;

`double SDL_CommandTime ()`

Выдать время выполнения текущей команды в секундах.

4.4 Информационные функции

`int SDL_Evaluate (const char* SDL, char* valuebuf, long bufsize)`

Оценить значение выражения `SDL` и записать результат в буфер `valuebuf` длины `bufsize`. В выражении можно использовать глобальные переменные, базовые наборы и локальные переменные активной программы. Если длины буфера не хватило для записи значения переменной, то возвращается требуемая длина буфера.

Возвращаемые значения:

0 — выражение корректно, значение записано в буфер;

1 — выражение ошибочно;

`int SDL_Execute (char* SDL)`

Выполнить последовательность команд, записанную в текстовой строке `SDL`. При выполнении задействуются переменные, режимы и базовые наборы активной программы. Это означает, что команды выполняются так, как если бы они были записаны внутри активной программы в том месте, на котором в настоящий момент остановлено её выполнение.

Возвращаемые значения:

0 — последовательность команд выполнена успешно;

1 — невозможно загрузить последовательность команд;

2 — в последовательности команд обнаружена ошибка.

`int SDL_GetInfo (const char* filename, int infotype)`

Записать информацию в файл с заданным именем. Тип информации задаётся параметром `infotype`, который может принимать следующие значения:

```
SDL_INFO_KERNEL = 1;     // состояние ядра
SDL_INFO_MEMORY = 2;     // текущий расход памяти
SDL_INFO_LIBRARIES = 3;  // доступные методы и библиотеки
SDL_INFO_FUNCTIONS = 4;  // доступные функции
SDL_INFO_PARAMETERS = 5; // параметры текущего метода
```

Возвращаемые значения:

0 — файл записан успешно;

1 — невозможно открыть файл.

`char* SDL_GetMemoryReport ()`

Выдать отчёт о текущем расходе памяти.

`char* SDL_GetString (char* key, SDL_Method* method)`

Выдать строковую константу по ключу `key`. Строковые константы, используемые ядром ASDIEL, хранятся в отдельных файлах с расширением `err` и загружаются вместе с ядром. По умолчанию строка ищется в файлах `sdl_*.err`. Если `method!=0`, то сначала просматриваются файлы ⟨`lib`⟩`_*.err`, где ⟨`lib`⟩ — имя библиотеки методов, из которой был загружен `method`. Возвращает `key`, если строка не найдена.

4.5 Запись и считывание значений переменных

`int SDL_VariableGetStr (char* varname, char* valuebuf, long bufsize)`

Преобразовать значение переменной с именем `varname` к типу `STRING` и записать в буфер `valuebuf` длины `bufsize`. Если длины буфера не хватило для записи значения переменной, то возвращается требуемая длина буфера.

Возвращаемые значения:

0 — значение переменной записано успешно;

1 — переменная не найдена.

`int SDL_VariableGetReal (char* varname, double* value)`

Преобразовать значение переменной `varname` к типу `REAL` и записать в `value`.

Возвращаемые значения:

0 — значение переменной преобразовано и записано успешно;

1 — переменная не найдена;

2 — переменная не может быть корректно преобразована в `REAL`.

`int SDL_VariablePutStr (char* varname, char* value, int can_create)`

Присвоить переменной `varname` значение `value` строкового типа. Если перед именем через пробел указано ключевое слово `global`, переменная считается глобальной; иначе — локальной в текущей программе. Если переменная не существует и `can_create`= 1, то она создаётся.

Возвращаемые значения:

0 — присваивание выполнено успешно;

1 — переменная не найдена.

`int SDL_VariablePutReal (char* varname, double value, int can_create)`

Присвоить локальной переменной `varname` значение `value` вещественного типа. Если перед именем через пробел указано ключевое слово `global`, переменная считается глобальной; иначе — локальной в текущей программе. Если переменная не существует и `can_create`= 1, то она создаётся.

Возвращаемые значения:

0 — присваивание выполнено успешно;

1 — переменная не найдена.

4.6 Запись и считывание данных из массивов

Интерфейс ядра ASDIEL позволяет работать с произвольными подмассивами без ограничений размерности, считывая и записывая значения отдельных ячеек. Тип подмассива определён как

```
typedef void SDL_Subarray;
```

Внутреннее представление подмассива скрыто от пользователя интерфейса, все операции с подмассивами производятся только через интерфейсные функции, которым указатель на `SDL_Subarray` передаётся в качестве аргумента.

`SDL_Subarray* SDL_SubarrayInit (const char* SDL)`

Оценить ASDIEL-выражение, которое должно быть подмассивом, и вернуть указатель на подмассив. При вычислении выражения задействуются переменные активной программы. Возвращает 0, если выражение ошибочное или его результат неприводим к типу `SUBARRAY`.

`int SDL_SubarrayFree (SDL_Subarray* subarray)`

Удалить подмассив.

Возвращаемые значения:

0 — подмассив успешно удалён;

-1 — подмассив не существует.

`int SDL_SubarrayDimension (SDL_Subarray* subarray)`

Выдать размерность подмассива.

Возвращаемые значения:

−1 — подмассив не существует.

`int SDL_SubarrayDimSize (SDL_Subarray* subarray, int dim)`

Выдать размер подмассива по `dim`-ой размерности.

Возвращаемые значения:

−1 — подмассив не существует;

−2 — `dim` выходит за пределы размерности подмассива.

`int SDL_SubarrayGetElemStr (SDL_Subarray* subarray,`
`const int* ind, char* valuebuf, long bufsize)`

Выдать значение ячейки подмассива, преобразовав его к типу `STRING` и записать в буфер `valuebuf` длины `bufsize`. Если длины буфера не хватило для записи значения переменной, то возвращается требуемая длина буфера. Вектор `ind` должен содержать координаты ячейки в массиве. Число координат должно быть равно значению `SDL_SubarrayDimension`.

Возвращаемые значения:

0 — значение ячейки считано успешно;

−1 — подмассив не существует;

2 — элемент не существует, выход за пределы матрицы;

3 — ячейка содержит ошибочное значение.

`int SDL_SubarrayGetElemReal (SDL_Subarray* subarray,`
`const int* ind, double* value)`

Выдать значение ячейки подмассива, преобразовав его к типу `REAL` и записать в `value`. Вектор `ind` должен содержать координаты ячейки в массиве.

Возвращаемые значения:

0 — значение ячейки считано успешно;

−1 — подмассив не существует;

2 — элемент не существует, выход за пределы матрицы;

3 — ячейка содержит ошибочное значение;

4 — ячейка содержит нечисловое значение.

`int SDL_SubarrayPutElemStr (SDL_Subarray* subarray,`
`const int* ind, char* value)`

Записать строковое значение `value` в ячейку подмассива. При необходимости преобразование типа выполняется автоматически. Вектор `ind` должен содержать координаты ячейки в массиве.

Возвращаемые значения:

0 — значение записано успешно;

−1 — подмассив не существует;

2 — элемент не существует, выход за пределы матрицы.

```
int SDL_SubarrayPutElemReal (SDL_Subarray* subarray,
     const int* ind, double value)
```

Записать вещественное значение `value` в ячейку подмассива. При необходимости преобразование типа выполняется автоматически. Вектор `ind` должен содержать координаты ячейки в массиве.

Возвращаемые значения:

0 — значение записано успешно;

-1 — подмассив не существует;

2 — элемент не существует, выход за пределы матрицы.

Функции, предназначенные для записи и считывания значений из отдельных ячеек подмассивов, используют целочисленные векторы типа `const int*` для указания координат ячеек. Если динамический метод реализуется на `C++`, то можно воспользоваться классом `CIndex`, который предоставляет удобный интерфейс для описания координат ячеек многомерных массивов. Это очень простой класс, который описан и реализован в заголовочном файле `SDLlib.h`:

```
class CIndex {
public:
  CIndex (int MaxDim=8);
  ~CIndex ();
  operator const int* ();
  CIndex& operator[] (int dim_index);
};
```

С помощью этого класса координату ячейки можно записывать прямо на месте аргументов функций, требующих указатель на `int`. Например, вот так в ячейку с координатами $[2, i, j]$ трёхмерного массива `A` записывается число π:

```
CIndex iA;
SDL_SubarrayPutElemReal (A, iA[2][i][j], 3.14159265);
```

По умолчанию число координат в многомерном индексе не может превышать 8. Если нужен более длинный индекс, максимальное число компонент следует явным образом указать конструктору класса `CIndex`.

4.7 Регистрация динамических методов

Описанный ниже способ реализации методов существенно отличается от рассмотренного в разделе 2. Он не является объектно-ориентированным, использует только интерфейсные функции ядра ASDIEL, и позволяет создавать динамически загружаемые библиотеки методов.

Для добавления динамического метода его необходимо зарегистрировать с помощью функции `SDL_RegistrateMethod`. Затем функцией `SDL_RegistrateAlgorithm`

последовательно регистрируются все алгоритмы метода. Регистрация производится единственный раз на стадии загрузки приложения или динамической библиотеки. Допускается производить регистрацию методов как до, так и после вызова функции `SDL_Init`. После добавления метода он становится доступен для использования в ASDIEL-программах посредством команды `METHOD`.

`int SDL_RegistrateMethod (const char* name, int nalgors, int max_argum)`
Зарегистрировать метод в ядре ASDIEL под именем `name` (именно это имя должно указываться в команде `METHOD`). Повторная регистрация под тем же именем переопределяет метод (тем не менее разные динамичекие библиотеки могут определять методы с одинаковыми именами). Параметр `nalgors` задаёт число алгоритмов в методе. После вызова функции `SDL_RegistrateMethod` должны следовать `nalgors` вызовов функции `SDL_RegistrateAlgorithm`, последовательно регистрирующие отдельные алгоритмы метода. Аргументы-подмассивы всех алгоритмов нумеруются с помощью единой сквозной нумерации; параметр `max_argum` задаёт максимальный номер аргумента.
Возвращаемые значения:
0 — метод успешно зарегистрирован;
1 — невозможно создать метода.

`int SDL_RegistrateAlgorithm (const char* scheme, SDL_FuncRun* func_run)`
Зарегистрировать в ядре ASDIEL алгоритм последнего зарегистрированного метода. Параметр `scheme` задаёт шаблон алгоритма. Правила записи шаблонов описаны в разделе 2.3. Параметр `func_run` является указателем на функцию выполнения алгоритма, см. раздел 4.9.
Возвращаемые значения:
0 — алгоритм успешно зарегистрирован;
1 — последний зарегистрированный метод не определён;
2 — число алгоритмов превысило заданное при регистрации метода;
3 — число аргументов превысило заданное при регистрации метода.

4.8 Динамические библиотеки методов

Несколько методов можно объединить в динамическую библиотеку. Чтобы использовать в ASDIEL-программе метод, находящийся в динамической библиотеке с именем `LibName.dll`, имя библиотеки указывают перед именем метода в команде `METHOD`:

```
METHOD LibName.MethodName;
```

Способ реализации динамической библиотеки зависит от операционной системы. В системе `Windows` она представляет собой обычный `dll`-модуль, экпортирующий единственную функцию `SDL_LibraryRegistrate`. Эта функция должна зарегистрировать все определяемые в библиотеке методы с помощью `SDL_RegistrateMethod`

и SDL_RegistrateAlgorithm. Разработчик библиотеки определяет данную функцию, но сам не вызывает её. Она вызывается ядром в момент первого обращения к какому-либо методу библиотеки с помощью команды METHOD.

`int SDL_LibraryRegistrate ()`

Функция возвращает число ошибок, допущенных при регистрации методов и их алгоритмов. Регистрация считается успешной только в том случае, если оно равно нулю.

Пример реализации простейшей библиотеки приводится ниже в разделе 4.11.

4.9 Реализация динамических методов

Интерфейсные функции, описанные в данном разделе, используются при реализации динамических методов и их алгоритмов. Все они в качестве первого аргумента принимают указатель на метод.

Функция выполнения алгоритма должна соответствовать прототипу

```
typedef void SDL_FuncRun(SDL_Method* M, const int* index, SDL_Datum* result);
```

Если index==0, то функция должна вычислить все выходные аргументы-подмассивы данного алгоритма. В противном случае она должна вычислить значение единственной ячейки в одном выходном подмассиве. Если алгоритм имеет несколько выходных подмассивов, то номер нужного подмассива можно узнать с помощью функции SDL_MethodArgumID. Местоположение ячейки в нём задаётся векторным индексом index. Вычисленное значение записывается не в ячейку, а в переменную result с помощью одной из функций SDL_DatumPut***.

Для записи и считывания аргументов и параметров метода функция выполнения алгоритма обращается к ядру ASDIEL с помощью функций, перечисляемых ниже.

`SDL_Subarray* SDL_MethodArgument (SDL_Method* method, int argum)`

Выдать указатель на подмассив — argum-ый аргумент метода method. Напомним, что номера аргументов задаются в шаблоне при регистрации алгортма. Функция возвращает 0, если аргумент не задан или не принадлежит выполняемому алгоритму. Для работы с подмассивом следует использовать функции, описанные в разделе 4.6.

`int SDL_MethodArgumID (SDL_Method* method)`

Выдать номер выходного аргумента, который требуется рассчитать. Если алгоритм имеет менее двух выходных подмассивов, то вызывать данную функцию не нужно. Возвращает -1, если метод не задан.

Следующие функции предназначены для работы с параметрами метода. Параметры каждого метода сведены в деревовидную структуру, поэтому доступ к ним

осуществляется через структрурные имена. Структурное имя параметра представляет собой последовательность имён и индексов, задающую его положение в дереве параметров подобно тому, как полное имя файла задаёт его местоположение в файловой системе. Однако, в отличие от директорий файловой системы, отдельные уровни структурного имени могут задаваться как символьным именем, так и индексом (порядковым номером). Например структурное имя параметра `@coeff@57@x` состоит из имени `coeff` (верхний уровень), индекса `57` (второй уровень) и имени `x` (третий уровень). Дерево параметров можно также представлять себе как сложную структуру данных, в которой допускаются массивы и записи, состоящие из массивов, записей и значений.

При обращении к параметрам через интерфейсные функции структурные имена задаются с помощью массива элементов вида

```
struct SDL_Param  {
    char* Name;
    long Index;
};
```

Поле `Name` задаёт строковое имя уровня. Если `Name==0` или `*Name==0`, то строковое имя игнорируется и считается, что уровень задаётся индексом `Index`. Если `Index<0`, то это означает конец структурного имени параметра.

Имя параметра передаётся посредством указателя на первый элемент массива элементов типа `SDL_Param`. Например, чтобы обратиться к параметру `@coeff@57@x`, необходимо сформировать массив

```
SDL_Param coeff[3] = {{"coeff",0},{0,57},{"x",0},{0,-1}};
```

и в качестве имени параметра передать `coeff` (указатель на `coeff[0]`).

Если динамический метод реализуется на `C++`, то можно воспользоваться классом `CParam`, который предоставляет удобный интерфейс для описания структурных имён параметров:

```
class CParam {
public:
    CParam (int MaxLength=7);
    ~CParam ();
    operator SDL_Param* ();
    CParam& operator[] (int index);
    CParam& operator[] (char* name);
};
```

С помощью этого класса структурные имена параметров записываются прямо на месте аргументов функций, требующих указатель на `SDL_Param`. Например, вот так в параметр `@coeff@57@x` метода `M` записывается число π:

```
SDL_ParamPutReal (M, CParam()["coeff"][57]["x"], 3.14159265);
```

По умолчанию число компонент в структурном имени параметра не может превышать 7. Если нужен более длинный параметр, максимальное число компонент следует явным образом указать конструктору класса `CParam`.

Во всех перечисленных ниже функциях можно обращаться к параметру, который ещё не существует в дереве параметров метода. В таком случае будет создан не только сам параметр, но и все не существовавшие до этого момента уровни, записанные в структурном имени.

```
int SDL_ParamGetReal (SDL_Method* method, SDL_Param* ind,
        double* value, double default_value)
```

Записать значение вещественного параметра в переменную `value`. Если параметр не существует, он будет создан и проинициализирован заданным вещественным значением `default_value`.

Возвращаемые значения:

0 — значение параметра считано успешно;

1 — метод не задан;

2 — значение параметра не приводится к вещественному типу.

```
int SDL_ParamGetStr (SDL_Method* method, SDL_Param* ind,
        char* valuebuf, long bufsize, char* default_value)
```

Записать значение строкового параметра в переменную `value`. Если параметр не существует, он будет создан и проинициализирован значением `default_value`. Если длины буфера не хватило для записи значения параметра, то возвращается требуемая длина буфера.

Возвращаемые значения:

0 — значение параметра считано успешно;

1 — метод не задан.

```
int SDL_ParamPutReal (SDL_Method* method,
        SDL_Param* ind, double value)
```

Задать значение `value` вещественного параметра. Если параметр не существует, он будет создан.

Возвращаемые значения:

0 — нет ошибки;

1 — метод не задан.

```
int SDL_ParamPutStr (SDL_Method* method,
        SDL_Param* ind, char* value)
```

Задать значение `value` строкового параметра. Если параметр не существует, он будет создан.

Возвращаемые значения:

0 — нет ошибки;

1 — метод не задан.

`int SDL_MethodLog (SDL_Method* method, const char* text)`

Вывести текстовую строку `text` в поток вывода отладочной информации, задаваемый параметром метода `@log`. Разработчик метода может не опасаться вставлять отладочную печать с помощью функции `SDL_MethodLog`: если параметр `@log` не определён в ASDIEL-программе, то отладочный вывод будет проигнорирован без особого ущерба для эффективности выполнения.

Возвращаемые значения:

0 — нет ошибки;

1 — метод не задан.

```
int SDL_MethodSetProgress (SDL_Method* method,
    double progress, const char* messkey)
```

Передать в ядро ASDIELинформацию о текущем состоянии вычислений: комментарий к выполняемой работе `messkey` и долю проделанной работы `progress` (в диапазоне от 0 до 1). Строка `messkey` является идентификатором сообщения, хранящегося в `err`-файле. Если `messkey==0`, то сохраняется комментарий, заданный при предыдущем вызове данной функции для данного метода.

Возвращаемые значения:

0 — нет ошибки;

1 — метод не задан.

Функции, перечисленные ниже, предназначены для записи значения в переменную типа `SDL_Datum`. Они используются функцией выполнения алгоритма в том случае, когда требуется рассчитать значение единственной ячейки в выходном подмассиве алгоритма (при `index!=0`).

`int SDL_DatumPutLong (SDL_Datum* dat, long value)`

Записывает целочисленное значение value в датум. Возвращает 0, если запись прошла успешно.

`int SDL_DatumPutReal (SDL_Datum* dat, double value)`

Записывает вещественное значение value в датум. Возвращает 0, если запись прошла успешно.

`int SDL_DatumPutStr (SDL_Datum* dat, char* value)`

Записывает строковое значение value в датум. Возвращает 0, если запись прошла успешно.

4.10 Использование двумерных числовых матриц

Двумерные числовые матрицы часто используются в качестве аргументов алгоритмов. Поэтому в ядре ASDIEL предусмотрена специальная группа функций, существенно облегчающая работу с подмассивами для данного частного случая. В отличие от подмассивов типа SDL_Subarray интерфейс ядра не предоставляет функций доступа к отдельным элементам матрицы. Вместо этого матрица определяется как класс с очевидной функцией доступа к элементам:

```
struct SDL_Matrix {
  int nr;   // число строк
  int nc;   // число столбцов
  double **data;
  double default_element;
  double& operator() (int r, int c=0) {return data[c][r];}
};
```

Для работы с матрицами в C++ необходимо дополнительно к SDLlib.h присоединить заголовочный файл Analyt.h из проекта ASDIEL.

`SDL_Matrix* SDL_MatrixInit (SDL_Subarray* subarray)`

Инициализировать матрицу с размерами, совпадающими с размерами данного подмассива. Возвращает 0 если не удалось создать матрицу (размерность подмассива не равна двум, либо не хватило памяти). По окончании работы с выданной матрицей её необходимо освободить функцией SDL_MatrixFree.

`SDL_Matrix* SDL_MatrixInitRC (int rows, int columns)`

Инициализировать матрицу с заданными размерами. Возвращает 0 если не удалось создать матрицу. По окончании работы с выданной матрицей её необходимо освободить функцией SDL_MatrixFree.

`int SDL_MatrixFree (SDL_Matrix* matrix)`

Освободить матрицу после использования. Матрица должна быть проинициализирована SDL_MatrixInit или SDL_MatrixInitRC.

Возвращаемые значения:

0 — OK;

−2 — матрица не существует;

2 — ошибка при удалении памяти.

`int SDL_SubarrayGetMatrix (SDL_Subarray* subarray, SDL_Matrix* matrix)`

Загрузить матрицу из массива. Матрица должна быть проинициализирована функцией SDL_MatrixInit.

Возвращаемые значения:

0 — OK;

−1 — подмассив не существует;

−2 — матрица не существует;
2 — размерность подмассива не равна двум;
3 — размеры матрицы и подмассива не соответствуют друг другу.

`int SDL_SubarrayPutMatrix (SDL_Subarray* subarray, SDL_Matrix* matrix)`
Записать матрицу в массив. Матрица должна быть проинициализирована функцией `SDL_MatrixInit`.
Возвращаемые значения:
0 — OK;
−1 — подмассив не существует;
−2 — матрица не существует;
2 — размерность подмассива не равна двум;
3 — размеры матрицы и подмассива не соответствуют друг другу.

`SDL_Matrix* SDL_MethodGetMatrix (SDL_Method* method, int argum)`
Возвращает `argum`-ый аргумент метода в виде заполненной числовой матрицы. Возвращает 0, если аргумент не задан, не принадлежит выполняемому алгоритму, или не является двумерным подмассивом.

`int SDL_MethodPutMatrix (SDL_Method* method, int argum, SDL_Matrix* mat)`
Записывает числовую матрицу целиком в `argum`-ый аргумент метода.
Возвращаемые значения:
0 — OK;
1 — метод не определён;
−2 — матрица не существует;
2 — размерность подмассива не равна двум;
3 — размеры матрицы и подмассива не соответствуют друг другу;
4 — аргумент не задан или не принадлежит выполняемому алгоритму.

`SDL_Matrix* SDL_ParamGetMatrix (SDL_Method* method, SDL_Param* ind)`
Выдать поддерево параметров в виде двумерной матрицы. Возвращает 0, если метод не задан или поддерево не существует или не является матрицей. По окончании работы с выданной матрицей её необходимо освободить функцией `SDL_MatrixFree`.

`int SDL_ParamPutMatrix (SDL_Method* method, SDL_Param* ind, SDL_Matrix*)`
Записать поддерево параметров как двумерную матрицу. Пример: если поддерево параметров `ind` задано как `CParam()["a"][7]`, то каждый (i, j)-ый элемент двумерной матрицы запишется как отдельный числовой параметр с именем `@a@7@`i`@`j. Если матрица имеет только один столбец (то есть является вектором), запись каждого i-го элемента вектора производится в параметр с именем `@a@7@`i. Все несуществующие параметры создаются.

Возвращаемые значения:

1 — метод не определён;

2 — матрица не существует;

4.11 Примеры

Ниже приводятся примеры простых приложений, использующие интерфейсные функции ядра ASDIEL.

Пример 1. Запуск программы

Простейшее приложение предназначено для запуска ASDIEL-программ. Имя программы передаётся в командной строке при запуске приложения.

Для правильной отработки программы необходимо выполнение двух условий. Во-первых, переменная среды `SDLDIR` должна указывать на директорию, в которой был установлен ASDIEL. Во-вторых, файл `$SDLDIR$\bin\init.sdl` должен существовать и быть корректной ASDIEL-программой.

После выполнения программы в текущей директории остаются два отладочных файла. Файл `sdl.log` содержит протокол выполнения программы. Файл `sdl_mem.log` содержит отчёт об использовании оперативной памяти.

```
#include "SDLlib.h"
void main(int argc, char *argv[], char *envp[]) {
  SDL_Init (argc, argv, envp, "");
  if (SDL_ProgramLoad(argv[1])==0)
    SDL_ProgramRun ();
  SDL_Done ();
}
```

Пример 2. Считывание значений из массива

Данное приложение не загружает программу из файла. Вместо этого выполняется последовательность команд ASDIEL, создающая и заполняющая массив $[\![F \times S]\!]$. Значения из этого массива считываются поэлементно и выводятся в стандартный поток вывода. Затем массив выводится в тот же поток, но уже на уровне ASDIEL — с помощью команды `cout«S|F;`. Тест заключается в том, что выведенные значения должны быть идентичны.

```
#include <iostream.h>
#include "SDLlib.h"
void main(int argc, char *argv[], char *envp[]) {
  SDL_Init (argc, argv, envp, "");
  SDL_Execute ("USE F[S]; =S{new a::10}; =F{x=S/card(S), y=1/(1+x*x)};");
  SDL_Subarray* sub = SDL_SubarrayInit ("S|F");
  int nR = SDL_SubarrayDimSize (sub,0);
```

```
  int nC = SDL_SubarrayDimSize (sub,1);
  int ind[2], &r=ind[0], &c=ind[1];
  double value;
  for (r=0; r<nR; r++) {
    cout « r;
    for (c=0; c<nC; c++) {
      SDL_SubarrayGetElemReal (sub, ind, &value);
      cout « "\t" « value;
    }
    cout « "\n";
  }
  SDL_Execute ("cout « S|F;");
  SDL_SubarrayFree (sub);
  SDL_Done ();
}
```

Пример 3. Запись значений в массив

Данное приложение, аналогично предыдущему, не загружает программу из файла, а выполняет последовательность команд, создающую пустой массив $[\![F \times S]\!]$. Массив заполняется и распечатывается поэлементно на уровне C++, затем выводится в стандартный поток cout на уровне ASDIEL. Тест снова заключается в том, чтобы выведенные массивы совпадали.

```
#include <iostream.h>
#include "SDLlib.h"
void main(int argc, char *argv[], char *envp[]) {
  SDL_Init (argc, argv, envp);
  SDL_Execute ("USE F[S]; =S{new a::10}; =new memory F{x, y};");
  SDL_Subarray* sub = SDL_SubarrayInit ("S|F");
  int ind[2];
  int nR = SDL_SubarrayDimSize (sub,0);
  double x, y;
  for (ind[0]=0; ind[0]<nR; ind[0]++) {
    ind[1] = 0;
    SDL_SubarrayPutElemReal (sub, ind, x = ind[0]/double(nR));
    ind[1] = 1;
    SDL_SubarrayPutElemReal (sub, ind, y = 1/(1+x*x));
    cout « ind[0] « \t « x « \t « y « \n;
  }
  SDL_Execute ("cout « S|F;");
  SDL_SubarrayFree (sub);
  SDL_Done ();
}
```

Пример 4. Простейшая динамическая библиотека

В этом примере приводится текст простейшей библиотеки, которая определяет единственный метод `Copy` с единственным алгоритмом, копирующим входной двумерный подмассив в выходной модмассив того же размера.

```
#include"SDLlib.h"
// Алгоритм вычисления метода Copy
void Copy_calc (SDL_Method* M, const int* index, SDL_Datum* result) {
  // получение аргументов
  SDL_Subarray* SFinput = SDL_MethodArgument (M, 0);
  SDL_Subarray* SFoutput = SDL_MethodArgument (M, 1);
  double value;
  if (index) {
    // расчёт единственного значения в выходном аргументе
    SDL_SubarrayGetElemReal (SFinput, index, &value);
    SDL_DatumPutReal (result, value);
  }
  else {
    // расчёт выходного аргумента целиком
    int m = SDL_SubarrayDimSize (SFinput,0);
    int n = SDL_SubarrayDimSize (SFinput,1);
    int ind[2], &i=ind[0], &j=ind[1];
    for (i=0; i<m; i++)
      for (j=0; j<n; j++) {
        SDL_SubarrayGetElemReal (SFinput, ind, &value);
        SDL_SubarrayPutElemReal (SFoutput, ind, value);
      }
  }
}
// Функция регистрации всех методов библиотеки в ядре SDL
int SDL_LibraryRegistrate () {
  int res = 0;
  // Метод копирования
  res += SDL_RegistrateMethod ("Copy", 1, 1, 10);
  res += SDL_RegistrateAlgorithm ("Calc 0:X|Y -> 1:X|Y", Copy_calc);
  return res;
}
```

Для компиляции динамической библиотеки `TestLib.dll` необходим ещё файл определения модуля `TestLib.def`:

```
LIBRARY TestLib
EXPORTS
    SDL_LibraryRegistrate
```

Скомпилированный файл `TestLib.dll` необходимо переместить в системную директорию `%SDLDIR%\bin`. После этого метод `Copy` можно сделать доступным с помощью команды `METHOD TestLib.Copy`.

4.12 Дополнения

Функции, перечисленные в дополнениях, используются только при реализации пользовательского интерфейса ASDIEL и впоследствие могут быть измененны.

Работа с разделами текущей программы

`int SDL_SectionsCount ()`
Выдать число разделов в текущей программе

`char* SDL_SectionGetText (int sect)`
Выдать текст `sect`-ого раздела (текущего если `sect`< 0) активной программы.

`int SDL_SectionSetText (int sect, char* SDL)`
Задать текст `sect`-ого раздела (текущего если `sect`< 0) активной программы.

`char* SDL_SectionGetName (int sect)`
Выдать имя `sect`-ого метода (текущего если `sect`< 0) активной программы.

`int SDL_SectionGetIndex ()`
Выдать номер текущего раздела в активной программе.

Потоки вывода

`int SDL_StreamFind (const char* name)`
Выдать номер потока по имени или -1 если не найден.

`char* SDL_StreamFlush (int streamID)`
Выдать содержимое буфера для потока с заданным номером. Возвращает 0 в случае ошибки. По окончании использования буфера необходимо вызвать функцию очистки потока `SDL_StreamClear`.

`int SDL_StreamClear (int streamID)`
Очистить буфер потока с заданным номером. Функция должна быть вызвана после `SDL_StreamFlush` и использования буфера.
Возвращаемые значения:
0 — буфер очищен;
1 — неверный номер потока.

`int SDL_StreamSetCout (void *Cout)`
Задать стандартный поток вывода. `Cout` должен быть указателем на `C++`-класс — наследник `ostream`.
Возвращаемые значения:
0 — поток успешно установлен;
1 — невозможно использовать поток в качестве стандартного вывода.

Установка функции оболочки

`int SDL_SetGuiFunction (SDL_GuiFunction guifun)`

Задать функцию оболочки, которая будет вызываться из ядра.

Возвращаемые значения:

0 — OK

1 — функция не задана

Функция оболочки определяется следующим образом

```
typedef void (*SDL_GuiFunction) (GuiInfo* guiinfo);
```

Эта функция реализуется разработчиком оболочки, но вызывается из ядра во время длительных вычислений. Она отображает текущее состояние в интерфейсе оболочки и проверяет, не прервал ли пользователь выполнение ASDIEL-программы.

При вызове ядро передаёт в функцию оболочки указатель на структуру `GuiInfo`. Все поля этой структуры, кроме последнего, содержат входную информацию для функции оболочки. Последнее поле `is_userstop` формируется самой функцией оболочки, то есть является её выходом.

Информационная запись для обмена данными между ядром ASDIEL и оболочкой:

```
struct GuiInfo {
  int progress;         // продвижение работы
  int progress_max;     // верхний предел продвижения
  char* progress_mess;  // сообщение о текущей выполняемой работе
  long run_ticks;       // время выполнения программы в тиках
  double run_seconds;   // время выполнения программы в секундах
  int is_userstop;      // пользователь прервал выполнение?
};
```