Учебно-методические пособия Учебно-научного центра ОИЯИ Дубна

УНЦ-2003-22

Т. М. Соловьева

ВВЕДЕНИЕ В ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ НА ПРИМЕРЕ ПАКЕТА **ROOT**

Учебное пособие

2003

Учебно-научный центр ОИЯИ

Т. М. Соловьева

ВВЕДЕНИЕ В ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ НА ПРИМЕРЕ ПАКЕТА ROOT

Учебное пособие

Дубна 2003

Соловьева Т. М. Введение в объектно-ориентированный анализ на примере пакета ROOT: Учебное пособие. — Дубна: ОИЯИ, 2003. — 87 с.

Предлагаемое учебное пособие соответствует в основном программе курса «Введение в объектно-ориентированный анализ на примере пакета ROOT», который несколько лет читался в УНЦ ОИЯИ. В нем излагаются основные возможности прикладного пакета программ ROOT, созданного для анализа и визуализации данных в физике высоких энергий. Раздел, содержащий основные сведения о C++, на котором написан ROOT, ни в коей мере не претендует на полноту изложения средств этого языка программирования. Существует очень много хороших книг по C++, наиболее популярные из которых приведены в списке литературы этого пособия. При изучении ROOT помимо данного пособия рекомендуется пользоваться «The ROOT Users Guide» под редакцией Р. Брана и др., а также материалами web-сайта http://root.cern.ch.

Solovieva T. M. Introduction to Object-Oriented Analysis Taking ROOT Framework as an Example: Textbook. — Dubna: JINR, 2003. — 87 p.

The proposed teaching manual corresponds for the most part to the program of the course «Introduction to Object-Oriented Analysis Taking ROOT Framework as an Example», which has been lectured on for a few years at the JINR UC. The manual presents basic possibilities of the Framework ROOT created for data analysis and visualization in high energy physics. The part containing basic information about C++, in which ROOT has been written, does not pretend to be a complete course on the means of this programming language. There exist a rich variety of good books on C++, the most popular of which are listed in the reference literature to this manual. While studying ROOT, it is recommended that «The ROOT Users Guide» edited by R. Brun et al. and the web site http://root.cern.ch be used besides this teaching manual.

Содержание

1. Предисловие

Многофункциональный пакет программ ROOT был создан для анализа экспериментальных данных и их визуализации в физике высоких энергий. Авторами этого программного продукта является группа физиков: Рене Бран (Rene Brun (ЦЕРН)), Фон Рademакер (Fons Rademakers (ФНАЛ)), Масахару Готу (Masaharu Goto (Япония)), Валерий Файн (Valery Fine (ЦЕРН)), Ненад Банкик (Nenad Buncic), Филипп Кэнал (Philippe Canal), Сюзанна Панасек (Suzanne Panacek) (ФНАЛ) и др. Являясь разработчиками таких известных пакетов, как PAW, PIAF и GEANT, Р.Бран и Ф.Рademакер пришли к выводу, что дальнейшее развитие программного обеспечения для обработки данных физических экспериментов ввиду усложнения их структуры требует внедрения объектно-ориентированных технологий программирования. Поэтому для написания ROOT был использован язык программирования общего назначения C++. Разработчики новой системы поставили перед собой задачи поддержки полного цикла анализа данных, управления комплексными структурами со сложной иерархией объектов и сохранения таких ранее использовавшихся методов анализа (например в PAW), как гистограммирование, фитирование и визуализация. Эти задачи были успешно решены. Современные эксперименты в области физики высоких энергий характеризуются большой статистикой и большой множественностью событий. На стадии анализа ROOT позволяет выделить из массива данных только интересующие исследователя события и тем самым существенно упростить их обработку. Поэтому в настоящее время ROOT является одним из самых популярных пакетов прикладных программ, использующихся в физических лабораториях всего мира.

Программный пакет ROOT хорошо документирован. Информационный узел ROOT http://root.cern.ch своевременно обновляется. Исходный текст ROOT позволяет легко генерировать документацию, и каждый класс имеет в сети свою собственную страницу, содержащую описание класса и объяснение каждого метода. Эта страница всегда соответствует последней официальной версии ROOT. По адресу http://root.cern.ch/root/html/ClassIndex.html можно найти информацию о классах. В дополнение к этому web-сайт ROOT содержит самую последнюю версию руководства пользователя ROOT, список публикаций по ROOT, прикладные и обучающие программы, примеры использования ROOT в крупных научных коллаборациях.

При составлении данного учебного пособия были использованы материалы web-сайта ROOT и "The ROOT Users Guide 3.05" [1].

Большую помощь автору при подготовке рукописи оказал А.Г.Соловьев, внеся значительные улучшения в раздел о языке C++ и устранив ряд неточностей. За это и за тщательное редактирование всей книги я приношу ему глубокую благодарность. Я также признательна Н.Б.Скачкову и Д.В.Бандурину за постановку этого учебного курса в УНЦ ОИЯИ и А.В.Стаднику за полезные советы.

2. Введение

1. В тексте пособия приняты следующие шрифтовые выделения:

   (a) шрифтом с символами, имеющими одинаковую ширину, выделены имена классов и методов ROOT, исходный код в скриптах, имена скриптов и каталогов, например `TObject::Draw()`,
   (b) курсивом выделены глобальные переменные, например *gDirectory*,
   (c) рубленым шрифтом выделены адреса в Интернете, например http://root.cern.ch.

2. В диалоговой системе подсказка ROOT имеет номер строки, например root[5], ради простоты этот номер в большинстве случаев отброшен.
3. Когда используется переменная, она показывается между угловыми скобками, например <lib>.

2.1. Соглашения кодирования

С первых дней развития ROOT было решено использовать набор соглашений, используемых при написании кода. Это позволило добиться единообразия внутри исходного текста. С помощью этого набора пользователю проще определить, с какой информацией он имеет дело, и, следовательно, понять код лучше и быстрее. Конечно, можно использовать другие соглашения кодирования, но если Вы собираетесь представлять некоторый код для включения в состав исходного кода ROOT, Вы будете должны их использовать. Итак, в ROOT приняты следующие соглашения кодирования:

1. Классы начинаются с прописной буквы `T`: `TTree, TBrowser`.
2. Фундаментальные типы заканчиваются строчной буквой `_t`: `Int_t`.
3. Члены-данные начинаются со строчной буквы `f`: `fTree`.
4. Функции-члены начинаются с прописной буквы: `Loop()`.
5. Константы начинаются со строчной буквы `k`: `kInitialSize, kRed`.
6. Глобальные переменные начинаются со строчной буквы `g`: `gEnv`.
7. Статические члены данных начинаются с сочетания букв `fg`: `fgTable`.
8. Типы перечисления начинаются с прописной буквы `E`: `EColorLevel`.
9. Локальные переменные и параметры начинают со строчных букв: `nbytes`.
10. Методы получения (getters) и установки (setters) начинаются с `Get` и `Set`, например, `GetFirst()` и `SetLast()`.

Чтобы гарантировать размер переменных, используйте для них предопределенные типы в ROOT:

1. `Char_t` - символьная переменная со знаком 1 байт,
2. `UChar_t` - символьная переменная без знака 1 байт,

3. `Short_t` - короткое целое число со знаком 2 байта,
4. `UShort_t` - короткое целое число без знака 2 байта,
5. `Int_t` - целое число со знаком 4 байта,
6. `UInt_t` - целое число без знака 4 байта,
7. `Long_t` - длинное целое число со знаком 8 байтов,
8. `ULong_t` - длинное целое число без знака 8 байтов,
9. `Float_t` - число с плавающей точкой 4 байта,
10. `Double_t` - число с плавающей точкой двойной точности 8 байтов,
11. `Bool_t` - логическая переменная (0=false, 1=true).

Если Вы не собираетесь сохранять переменную на диске, то можно использовать не только `Int_t`, но и `int`, результат будет один и тот же, интерпретатор или компилятор обработают их одинаково.

В заголовочных файлах ROOT `Gtypes.h`, `Htypes.h` и `Rtypes.h` с помощью `typedef` введены специальные типы для графических и гистограммных классов, например `Option_t`, `Color_t` и `Axis_t`.

2.2. Установка ROOT

Чтобы установить ROOT на вашем компьютере, зайдите на web-сайт http://root.cern.ch/root/Availability.html. После выбора версии между `new`, `pro` и `old` (для новых пользователей рекомендуется версия `pro`) Вы должны будете также выбрать между загрузкой исходного кода или скомпилированной версии.

Исходный дистрибутив ROOT машинно независим. Порядок его инсталляции можно найти по адресу: http://root.cern.ch/root/Install.html. После создания `root` каталога прочтите файл README/INSTALL для детального описания процедуры построения ROOT в нужной конфигурации.

Бинарный дистрибутив ROOT зависит от платформы. После загрузки соответствующего файла распакуйте его командами:

```
gunzip root_<name_version>.<name_platform>.tar.gz
tar xvf root_<name_version>.<name_platform>.tar
```

Это создаст `root` каталог. Прочтите файл README/README перед началом работы.

Порядок установки ROOT можно также найти в приложении A в [1].

2.3. Структура ROOT

После инсталляции ROOT в каталоге `root` будут находиться следующие каталоги:

- `bin` содержит выполняемые программы:
  - `root` - показывает заставку ROOT и вызывает `root.exe`;

  - `root.exe` - основная выполняемая программа, обеспечивающая интерактивный режим работы с пакетом;
  - `rootcint` - утилита ROOT, используемая для создания словаря классов для CINT;
  - `rmkdepend` - модифицированная версия makedepend из C++, используется ROOT для построения системы;
  - `root-config` - скрипт, возвращающий флаги, необходимые для компиляции проектов, которые компилируются вместе с ROOT;
  - `cint` - C++ интерпретатор, независимый от ROOT;
  - `makecint` - чистая CINT версия `rootcint`, используемая некоторыми CINT скриптами для генерации словаря для внешних системных библиотек;
  - `proofd` - демон для подтверждения подлинности пользователя в процессе параллельной обработки ROOT (PROOF);
  - `proofserv` - основной демон для PROOF;
  - `rootd` - демон для отдаленного доступа к ROOT файлам.

- `lib` содержит библиотеки ROOT. Часть из них загружается по умолчанию, сразу после входа в ROOT. Некоторые нужно загружать дополнительно, например: `libEGPythia.so` и `libEGPythia6.so` - интерфейс к пакетам `Pythia5` и `Pythia6`, `libRFIO.so` - интерфейс к системе отдаленного ввода/вывода CERN и `libRGL.so` - интерфейс к `OpenGl`.
- `tutorials` содержит много скриптов с обучающими программами. Они содержат некоторые элементарные знания ROOT.
- `test` содержит набор примеров, которые представляют все области ROOT. Когда выходит новая версия, примеры в этом каталоге компилируются и выполняются, чтобы проверить совместимость новой версии ROOT со старой.
- `include` содержит все заголовочные файлы ROOT.

2.4. Глобальные переменные

ROOT имеет набор глобальных переменных, которые используются в сеансе. Все глобальные переменные начинаются со строчной буквы "g", за которой следует заглавная буква.

- *gROOT* - указатель, с помощью которого пользователь получает доступ к каждому объекту, созданному в программе ROOT. В течение ROOT сеанса *gROOT* сохраняет ряд совокупностей объектов. К ним можно обращаться с помощью `gROOT::GetListOf...` методов.
- *gFile* - указатель на текущий открытый файл.
- *gDirectory* - указатель на текущий каталог.
- *gPad* - указатель на активный объект класса `TPad`, на который будут выводиться все графические объекты.

- *gRandom* - указатель на текущий генератор случайных чисел.
- *gStyle* - указатель на текущий стиль.
- *gEnv* - глобальная переменная со всеми параметрами настройки среды для текущего сеанса.

2.5. Преобразование HBOOK/PAW файлов в ROOT формат

Для пакета ROOT был разработан свой собственный формат файлов. Огромное количество информации в физике высоких энергий было обработано пакетами прикладных программ HBOOK и PAW. Чтобы конвертировать файлы с гистограммами и другими данными, созданными этими пакетами, в файлы ROOT, существует утилита `h2root`. Чтобы ее использовать, наберите команду `shell`:

```
h2root <hbookfile> <rootfile>
```

Если второй параметр не определен, то имя файла сгенерируется автоматически. То есть если `hbookfile` имеет форму `file.hbook`, то файл будет называться `file.root`.

Утилита `h2root` преобразует HBOOK гистограммы в объекты ROOT класса `TH1F`, HBOOK профильные гистограммы - в объекты ROOT класса `TProfile`, а HBOOK ntuples - в объекты ROOT класса `TTree`. В последнем случае каждый столбец ntuple станет ветвью ROOT дерева. В некоторых случаях, если каждый элемент ntuple сам является многомерным массивом, конвертация HBOOK файлов может пройти не полностью. Имена гистограмм в ROOT файле будут иметь вид `hID (h_ID)`, где `ID` есть целый положительный (отрицательный) идентификатор HBOOK гистограммы.

3. Необходимые сведения о C++

В этом разделе мы познакомим читателей с основными концепциями C++, необходимыми для работы с ROOT и понимания документации. Если Вы уже знакомы с C++, смело пропускайте этот раздел и переходите к следующему.

3.1. Основные средства

3.1.1. Типы и объявления

В C++ имеется набор фундаментальных типов, отражающих характерные особенности организации памяти большинства компьютеров и наиболее распространенные способы хранения данных:

- логический тип (`bool`);
- символьные типы (например, `char`);

- целые типы (например, `int`);
- типы с плавающей точкой (например, `double`).

Прежде чем имя (идентификатор) может быть использовано в программе на C++, оно должно быть объявлено. То есть должен быть указан тип имени, чтобы компилятор знал, на сущность какого вида ссылается имя. Объявление состоит из четырех частей: необязательного спецификатора, базового типа, объявляющей части и, возможно, инициализатора (начального значения). Например:

```
int a=1;
```

В этом примере базовым типом является `int` , объявляющей частью - `a`, инициализатором - `1`. В качестве спецификаторов могут выступать ключевые слова, например `virtual`. Они приводятся в начале объявления и описывают характеристики, не связанные с типом. Объявляющая часть состоит из имени, которое может содержать последовательность букв, цифр и подчеркиваний, но не может начинаться с цифры. Ключевые слова C++ нельзя использовать в качестве имени сущности, определяемой пользователем. Разрешается объявлять несколько имен, разделенных запятыми, в одном операторе объявления.

3.1.2. Область видимости

Объявление вводит имя в область видимости. Это значит, что имя может использоваться только в определенной части программы. Для имени, объявленного в теле функции (такое имя часто называют **локальным**), область видимости начинается с места объявления имени и заканчивается в конце блока, в котором это имя объявлено. Блоком называется фрагмент текста, заключенный в фигурные скобки. Имя называется **глобальным**, если оно объявлено вне любой функции или класса. Область видимости глобальных имен простирается от места их объявления до конца файла, содержащего объявление.

3.1.3. Размещение инструкций и комментарии

Текст программы размещается свободно, на одной строке может находиться несколько инструкций, каждая инструкция заканчивается знаком ";" (точка с запятой). Комментарии в C++ начинаются с `//` (два слеша) и заканчиваются в конце строки. Можно также использовать комментарии в стиле C:

```
 /* Это комментарий  */
 /*
Это тоже комментарий
 */
```

3.1.4. Арифметические операторы

Арифметические операторы C++ в большинстве подобны арифметическим операторам других языков программирования. Исключение составляют операторы

инкремента (++) и декремента(- -). Оператор ++ (по которому язык C++ и получил свое название) используется для явного указания операции увеличения вместо менее явной записи, состоящей из комбинации сложения и присваивания. По определению, `++lvalue` означает `lvalue = lvalue + 1`. Подобным же образом записывается оператор декремента --. Инкремент и декремент могут быть как постфиксными, так и префиксными. Постфиксный инкремент (декремент) состоит из двух шагов: 1 - использовать переменную, 2 - увеличить (уменьшить) ее. Префиксный инкремент (декремент): 1 - увеличить (уменьшить) переменную, 2 - использовать ее. Существует укороченная запись комбинации арифметических действий и присваивания:

`x=x+y -> x+=y, x=x-y -> x-=y, x=x*y -> x*=y, x=x/y -> x/=y`

Правила установления приоритета арифметических операторов аналогичны правилам приоритета соответствующих арифметических действий.

3.1.5. Инструкции передачи управления

C++ предоставляет обычный набор инструкций для реализации ветвления и циклов.

1. Инструкция выбора:
   if *(условие) выражение_1;* **else** *выражение_2;*
   Первое выражение выполняется, если условие истинно, второе выражение выполняется, если условие ложно.
2. Цикл с заранее заданным числом повторений (цикл со счетчиком):
   for *(объявление и инициализация переменной счетчика; условие продолжения цикла; выражение для модификации переменной счетчика) выражение, выполняемое много раз.*
3. Циклы, выполняющие блок операторов до тех пор, пока указанное условие истинно, бывают двух типов:

   - цикл **while** *(условие) выражение;*
     используется, если есть вероятность, что операторы цикла могут не выполняться;
   - цикл **do** *выражение* **while** *(условие);*
     используется, если операторы цикла выполняются по крайней мере один раз.

В качестве выражения может использоваться набор инструкций, заключенный в фигурные скобки. В этом случае после закрывающей фигурной скобки точка с запятой не ставится.

3.1.6. Логические операторы и операторы отношения

В инструкциях передачи управления применяются следующие логические операторы, расположенные в порядке убывания приоритета:

- ! — логическое "не",
- && — логическое "и",
- || — логическое "или"

и операторы отношения:

- < — меньше,
- > — больше,
- <= — меньше или равно,
- >= — больше или равно,
- == — равно,
- != — не равно.

Первые четыре оператора отношения имеют более высокий приоритет, чем два последних. Арифметические операторы имеют более высокий приоритет, чем логические операторы и операторы отношения. При составлении сложных выражений рекомендуется применять скобки.

3.2. Стандартная библиотека

Ни одна программа приличных размеров не пишется с использованием только "голых" конструкций языка. Каждый компилятор C++ поставляется со стандартной библиотекой C++, которая

- обеспечивает поддержку свойств языка, таких как управление памятью и информация о типах во время выполнения;
- предоставляет функции, которые не могут быть написаны оптимально для всех систем собственно на языке C++, например `sqrt`;
- предоставляет программисту нетривиальные средства, на которые он сможет рассчитывать, заботясь о переносимости, например, списки, функции сортировки и потоки ввода/вывода;
- служит общим фундаментом для других библиотек.

Средства стандартной библиотеки представлены набором заголовочных файлов. Они идентифицируют основные части библиотеки. Полное их перечисление можно найти в [2]. Например, библиотека `<cmath>` содержит общие математические функции, в частности функцию возведения в степень `pow(d,e)` (`d` в степени `e`). (В C++ нет оператора возведения в степень, как в ФОРТРАНе и многих других языках программирования.) Для использования возможностей каждого средства стандартной библиотеки в программу должен быть включен соответствующий заголовочный файл.

3.2.1. Операторы ввода-вывода

В заголовочном файле <iostream> стандартной библиотеки объявлены оператор >> - оператор ввода и оператор << - оператор вывода. Кроме того, <iostream> предоставляет стандартные потоки ввода cin и вывода cout, а также стандартный поток ошибок cerr. Строка #include <iostream> в начале программы дает указание компилятору включить объявления средств ввода/вывода стандартной библиотеки <iostream>. Без этого мы не могли бы воспользоваться операторами ввода/вывода.

Приведем здесь небольшой пример кода с оператором вывода. Объявим переменную a целого типа, присвоим ей значение 5 и выведем ее.

```
#include <iostream>
int a;
a = 5;
cout < < "a = " < < a < < endl;
```

3.3. Производные типы

Для каждого типа, как для встроенного (фундаментального), так и для определенного пользователем, существуют производные типы.

- Массив - линейная последовательность объектов одного типа. Элементы массива доступны через [индекс]. Если количество элементов массива равно n, то первый индекс равен 0 , а последний n-1. Можно объявлять многомерные массивы, например Double_t x[20][10] - объявление двумерного массива.
- Указатель (pointer) - объект, который указывает на другой объект, то есть содержит адрес другого объекта. Чтобы использовать объект, на который указывает указатель, необходимо применить оператор *.
  *ptr - объект, на который указывает указатель ptr.
  Чтобы использовать адрес объекта, то есть инициализировать указатель, необходимо применить оператор &.
  &obj - адрес объекта obj.
  Методы объекта доступны через оператор . (точка).
  object.member()
  Методы объекта, объявленного через указатель, доступны через оператор ->.
  objectpointer->member().
- Ссылка (reference) - другое имя объекта. Не существует нулевых ссылок. Ссылка должна всегда ссылаться на какой-либо объект.

И указатели, и ссылки позволяют неявно ссылаться на другие объекты. Однако между этими типами есть несколько различий [3]. Если ваша переменная обеспечивает доступ к объекту, которого может и не быть, необходимо использовать

указатель, потому что это позволит приравнять его к нулю. С другой стороны, если переменная должна всегда ссылаться на существующий объект, то есть не должна иметь нулевого значения, то, скорее всего, лучше использовать в качестве такой переменной ссылку. Другое важное различие между указателями и ссылками состоит в возможности присваивать указателям различные значения для доступа к разным объектам. Ссылка же всегда указывает на один и тот же объект, заданный при ее инициализации.

3.4. Создание и уничтожение объектов

Оператор `new` выделяет память для объекта, размещает в ней объект и возвращает указатель на этот объект. Оператор `delete` уничтожает объект, на который указывает указатель. Каждый `new` должен быть спарен только с одним `delete`. Двойное уничтожение может привести вашу программу к прерыванию. Если не производить уничтожение, то программа будет занимать все больше и больше памяти, а этого лучше избегать. Массив создается оператором `new name[size]`, а уничтожается оператором `delete[]name` , здесь `name` - имя массива, `size` - его размер.

В зависимости от своего времени жизни объекты подразделяются на следующие категории.

- Если пользователь не указал явно иное, то объект, объявленный в функции, создается, когда встречается его определение, а уничтожается в момент выхода его имени из области видимости. Такие объекты называются **автоматическими**.
- Объекты, объявленные глобально и живущие до конца программы, называются **статическими**.
- Объекты, создаваемые в области свободной памяти, размещаемые и удаляемые в произвольном порядке под контролем пользователя, называются **динамическими**. Ясно, что объекты, создаваемые `new`, являются динамическими.

3.5. Функции

Определить функцию в C++ - значит задать способ выполнения операции. К функции нельзя обратиться, если она не была предварительно объявлена.

Объявление функции задает имя функции, тип возвращаемого значения (указание `void` в качестве возвращаемого значения означает, что функция не возвращает значения) и количество и типы аргументов, которые должны присутствовать при вызове функции. Объявление функции может содержать и имена аргументов, но для компилятора это не важно. Если функция не имеет аргументов, то она объявляется с пустым списком аргументов.

Например, объявим функцию с именем `mult` и двумя аргументами типа `Double_t`, возвращающую величину типа `Double_t`.

```
Double_t mult(Double_t, Double_t);
```

Каждая функция, вызываемая в программе, должна быть где-нибудь определена (и только один раз). Определение функции является объявлением функции, в котором присутствует тело функции. Тело функции заключается в фигурные скобки. Типы в определении и объявлениях функции должны совпадать. Однако имена аргументов не являются частью типа и совпадать не обязаны. Например, определение функции, объявленной выше, может выглядеть так:

```
Double_t mult(Double_ta, Double_t b) {
return a*b;
}
```

Важной особенностью C++ являются перегруженные функции. Что это означает? Как правило, разным функциям дают разные имена, но когда функции выполняют концептуально аналогичные задачи для объектов различных типов, может оказаться удобным присвоить им одно и то же имя. Использование одного имени для операции, выполняемой с различными типами, называется перегрузкой. Следовательно, в C++ допускается определять функции с одним и тем же именем, но с разным типом или количеством аргументов. Однако возвращаемое значение не участвует в процессе перегрузки.

У функций общего назначения часто больше аргументов, чем требуется в простых случаях. В частности, функции, создающие объекты, для увеличения гибкости обычно предоставляют несколько возможностей. В этом случае используются аргументы по умолчанию. Они всегда задаются только в конце списка аргументов. Тип аргумента по умолчанию проверяется в месте объявления функции и вычисляется в момент вызова функции.

3.6. Классы

Класс - это тип, определенный пользователем. Часто класс представляет собой структуру, которая содержит группу связанных переменных, расширяемую функциями, специфичными для этой структуры (класса).

Например, определим класс, представляющий собой линию

```
class TLine {
protected: Double_t x1;
Double_t y1;
Double_t x2;
Double_t y2;
public: TLine(int x1, int y1, int x2, int y2);
virtual void Draw();
};
```

Первая строка данной записи содержит имя класса. В последующих четырех строках объявляются члены-данные, перед которыми стоит модификатор досту-

па, в данном случае protected. Далее с модификатором доступа public введены две функции, которые мы будем называть методами или функциями-членами класса TLine. Первый метод называется конструктором и используется для инициализации объекта линии. Второй метод предназначен для рисования. Поэтому, чтобы построить и нарисовать линию, мы должны сделать следующее:

```
TLine l(1.,1.,2.,2);
l.Draw();
```

В первой строке данного кода строится объект l (вызывается его конструктор). Во второй строке вызывается метод Draw()[1] этого объекта, параметры для этого метода передавать не нужно, так как он связан с объектом l, который знает координаты линии - внутренние переменные, которые были инициализированы конструктором.

Ключевые слова public (открытый), private (закрытый) и protected (защищенный) регулируют права доступа к членам-данным и методам класса. Каждое имя, объявленное public, доступно для пользователя класса. Если имя объявлено private, то оно может быть доступно только в классе, в котором объявлено. Чтобы обратиться к нему из внешнего мира, в ROOT используются специальные методы "getters" ("получатели") и "setters" ("устанавливатели"). Объявление protected отличается от private лишь тем, что имена, объявленные protected, доступны из дочерних классов. Например, если метод Draw() объявлен как public, то каждый пользователь будет способен его видеть и использовать. Если переменная x1 в классе TLine объявлена private, то к этой переменной могут обратиться только методы (члены-функции) класса TLine.

3.7. Наследование, полиморфизм и инкапсуляция

Рассмотрим одно из основных понятий C++ - наследование. Мы определили класс TLine, который содержит все необходимое, чтобы нарисовать линию. Теперь мы хотим нарисовать стрелку. Стрелка является линией с треугольником на одном из концов линии. Было бы неэффективно определять класс TArrow на пустом месте. Поэтому применим наследование и определим класс TArrow как производный класс от существующего класса TLine.

```
class TArrow: public TLine{
int ArrowHeadSize;
void Draw();
void SetArrowSize(int arrowsize);
};
```

TArrow наследует public и protected методы и данные от TLine. То есть содержит все, что имеет класс TLine и добавляет еще пару вещей, размер треуголь-

[1]Значение ключевого слова virtual будет объяснено при рассмотрении полиморфизма в следующем параграфе.

ника и функцию, чтобы изменять его. (Все, что объявлено `private` в `TLine`, недоступно в `TArrow`.) `Draw()` метод `TArrow` нарисует треугольник и вызовет метод рисования класса `TLine`. То есть мы должны написать код только для рисунка треугольника.

Задание одного и того же названия метода в дочернем классе `TArrow` и родительском классе `TLine` не приводит ни к каким проблемам. Функция `Draw()` в `TArrow` переопределяется для рисования стрелки вместо линии. Метод обращается к объекту, тип которого известен. Если мы имеем объект `l` типа `TLine` и объект `a` типа `TArrow`, то при вызове `l.Draw()` рисуется линия, а при вызове `a.Draw()` рисуется стрелка.

Функция `Draw()` является виртуальной функцией. Виртуальные функции предоставляют возможность программисту объявить в базовом классе функции, которые можно заместить в каждом производном классе. Компилятор и загрузчик гарантируют правильное соответствие между объектами и функциями, применяемыми к ним. Для того, чтобы объявление виртуальной функции работало в качестве интерфейса к функциям, определенным в производных классах, типы аргументов функции в производном классе не должны отличаться от типов аргументов, объявленных в базовом классе, и только очень небольшие изменения допускаются для типа возвращаемого значения. Виртуальную функцию-член, как уже говорилось выше, иногда называют методом. Виртуальная функция должна быть определена для класса, в котором она впервые объявлена.

Когда функции родительского класса ведут себя правильно, независимо от того, какой конкретно производный от него класс используется, это называется полиморфизмом [4]. Тип, имеющий виртуальные функции, называется полиморфным типом. Для достижения полиморфного поведения в C++ вызываемые функции-члены должны быть виртуальными и доступ к объектам должен осуществляться через ссылки или указатели.

Чаще всего члены-данные классов объявляются `private` или `protected`, а методы `public`. Это называется сокрытием данных или инкапсуляцией и является одним из преимуществ объектно-ориентированного программирования. Сокрытие данных позволяет программисту неоднократно изменять реализацию классов, а для пользователя это будет оставаться незамеченным.

3.8. Заголовочные файлы

В тексте данного пособия уже встречалось такое понятие, как заголовочный файл. Что оно означает? C++ поддерживает соглашение C о раздельной компиляции. Это можно использовать для организации программы в виде почти независимых частей. Принято помещать объявления типов, функций и классов, определение констант и комментарии в отдельный файл, носящий название заголовочного файла и имеющий расширение `.h`[2]. Файл, содержащий определения

[2]Заголовочные файлы стандартной библиотеки не имеют расширение `.h`.

функций, различные инструкции и другой исполняемый код, называется исполняемым файлом и, как правило, имеет расширение .c. Чтобы собрать фрагменты исходного кода программы в одну единицу, используется директива
#include "включаемая_компонента"
которая заменяет строку, содержащую **#include**, на содержимое файла **"включаемая_компонента"**. Для включения заголовочных файлов стандартной библиотеки вместо кавычек используются угловые скобки < и >. Например:

```
#include <iostream>
#include "myheader.h"
```

Пробелы внутри < > и " " не допускаются.

4. CINT - интерпретатор C++

Встроенный в ROOT интерпретатор CINT переводит C++ на машинный язык. Благодаря ему пользователь общается с ROOT на языке C++. CINT является также процессором скриптов - программ, которые исполняют определенные задачи. CINT организован в виде командной строки и охватывает около 85% конструкций языка C++. Он поддерживает многократное наследование, виртуальные функции, перегрузку функций и операторов, заданные по умолчанию параметры, шаблоны и многое другое. CINT оптимизирован для кода, который часто изменяется, в то же время он способен быстро загружать исходные файлы и обрабатывать очень большой исходный текст. В отличие от большинства других интерпретаторов, имеющих отдельный язык, разработанный специально для интерпретации, CINT использует тот же язык, что и компилятор. Преимущество наличия одного языка для интерпретации и компиляции состоит в том, что как только опытный образец кода отлажен, он может быть сразу откомпилирован. CINT допускает смешивание интерпретации C++ и своих собственных команд, обеспечивает идентификацию типа во время выполнения и имеет встроенный отладчик.

Вообще CINT является автономным изделием и может быть установлен отдельно от ROOT. Далее мы будем обсуждать версию CINT, встроенную в ROOT.

4.1. Основные команды CINT

Интерпретатор поддерживает три типа команд.

1. CINT команды начинаются с точки.
   (a) .? или .help - выводит список всех команд CINT,
   help/[keyword] - выводит информацию о keyword;
   (b) .> <filename> - перенаправляет выход в файл <filename>;
   (c) .f <filename> - показывает исходный код файла <filename>;
   (d) .v <line> - показывает исходный код вокруг строки <line>;

(e) .p <expr> - оценивает выражение или показывает текущее значение переменной;
(f) .L <filename> - загружает файл <filename>;
(g) .x <filename> - загружает <filename> и исполняет утверждения, заключенные в фигурные скобки;
(h) .g - выдает список глобальных переменных;
(i) .l - выдает список локальных переменных;
(j) .class - показывает все классы ROOT;
(k) .class <name> - показывает определение класса <name> (один уровень);
(l) .Class <name> - показывает определение класса <name> (все уровни);
(m) .function - показывает все функции интерпретатора;
(n) .file - показывает список загруженных файлов;
(o) .b - устанавливает точки прерывания;
(p) .db - уничтожает точки прерывания;
(q) .s - шаг внутри функции или цикла;
(r) .c <line> - продолжает со строки <line>;
(s) .e - выход из функции;
(t) .q - выход из интерактивного сеанса ROOT.

2. SHELL команды начинаются с точки и восклицательного знака, например `root [] .!ls`
3. C++ команды следуют синтаксису C++ (почти всегда).

Интерпретатор знает все классы, функции, переменные и определяемые пользователем типы. Это облегчает пользователю формирование командной строки. Для этого используется клавиша [Tab], позволяющая завершить набранное выражение. Например, наберите в командной строке часть названия класса и нажмите клавишу [Tab]. Если начинающиеся с набранного сочетания букв классы существуют, то интерпретатор их перечислит. Или наберите метод, откройте скобку и нажмите клавишу [Tab]. В результате получите список всех параметров метода.

Команды CINT регистрируются в файле хронологии `.root_hist`. Этот файл содержит последние 100 команд. Это текстовый файл, и его можно редактировать. Чтобы обратиться к предыдущей и последующей команде, находясь в сеансе ROOT, используйте стрелки вверх и вниз в командной строке.

CINT допускает использование команды с несколькими строками. Такая команда должна начинаться и заканчиваться фигурной скобкой {...}. Каждая строка в команде должна заканчиваться точкой с запятой. Все создаваемые в команде объекты будут иметь глобальную область видимости. В такой команде невозможно вернуться на предыдущую строку.

4.2. ROOT/CINT расширения C++

Встроенный в ROOT интерпретатор ослабляет некоторые требования синтаксиса C++, а именно:

1. При создании объекта конструктором **new** может быть опущено объявление этого объекта, например

   ```
   root [] l=new TLine(0.1,0.1,0.9,0.9)
   ```

2. При вызове методов запись " -> " эквивалентна " . ". Так, несмотря на то, что l - указатель на TLine, допустимо использовать

   ```
   root [] l.Print()
   ```

3. Можно опускать точку с запятой в конце строки.
4. В случае, если CINT не может найти упоминаемый объект, он будет просить ROOT искать объект с идентичным названием в пути поиска, определенном `TROOT::FindObject()`. Если ROOT находит объект, он возвращает CINT указатель на этот объект и указатель на его определение класса, и CINT выполняет требуемую функцию-член.

Когда интерпретатор заменяется транслятором, эти расширения не работают. Поэтому при написании скриптов от этих расширений нужно отказаться.

4.3. CINT - процессор скриптов

Скрипты, написанные для ROOT, должны содержать чистый C++ код. Скрипты могут содержать простую последовательность утверждений, определения функции и определения классов.

4.3.1. Неименованные скрипты

Пусть файл с именем `script1.C` содержит следующие утверждения

```
{
#include<iostream>
Double_t x=2.;
Int_t i=25;
cout <"x = " < x < "i = " < i <endl;
}
```

Этот скрипт состоит из списка инструкций, заключенных в фигурные скобки. Объекты, которые определяются в таком скрипте, имеют глобальную область видимости. Они остаются доступными с командной строки после того, как скрипт выполнен. При выполнении неименованного скрипта снова и снова необходимо очищать глобальную среду. Это делается вызовом `gROOT->Reset()`. Желательно начинать этой командой каждый неименованный скрипт. `gROOT->Reset()` вызывает деструктор объекта, если объект был создан в стеке. Если объект был создан через **new**, он не удаляется, но переменная перестает быть связанной с ним. Об этом нужно помнить, чтобы не допустить потери памяти. Чтобы выполнить вышеупомянутые инструкции, напечатайте в командной строке:

```
root [] .x script1.C
```

Интерпретатор ищет скрипты в Root.MacroPath, как определено в вашем .rootrc файле.

4.3.2. Именованные скрипты

Создадим файл script2.C, добавив в файл script1.C утверждение функции:

```
#include<iostream>
int main()
{
Double_t x=2.;
Int_t i=25;
cout <"x = " < x < "i = " < i <endl;
return 0;
}
```

Обратите внимание на расположение фигурных скобок в данном случае. Чтобы выполнить функцию main() в script2.C,напечатайте:

```
root [] .L script2.C
root [] main()
```

Как только функция, объявленная в скрипте, была загружена, она становится частью системы точно так же, как откомпилированная функция. Это можно проверить, набрав

```
root [] .func
```

Создадим файл script3.C, изменив в файле script2.C имя функции с main() на script3.C():

```
#include<iostream>
int script(int j=10)
{
Double_t x=2.;
Int_t i=j;
cout <"x = " < x < "i = " < i <endl;
return 0;
}
```

Чтобы выполнить функцию script3() в файле script3.C, напечатайте:

```
root [] .x script3.C(5)
```

Эта команда загружает файл script3.C и выполняет функцию script3(5). Имя файла (без расширения) и имя функции должны быть одинаковыми. Объекты, которые определяются в именованном скрипте, имеют локальную область видимости. Чтобы избежать исчезновения объектов после выхода из функции, необходимо создавать их через указатель с помощью оператора new.

4.3.3. Выполнение скрипта из скрипта

Чтобы выполнить скрипт внутри другого скрипта, необходимо вызвать интерпретатор. Это можно сделать командой

```
gROOT->ProcessLine(".x script.C");
```

Для примера смотрите `tutorials/cernstaff.C`.

4.3.4. Скрипт, содержащий определение класса

Давайте создадим небольшой класс `TMyClass` и класс `TChild`, дочерний от него. Пусть в классе `TChild` виртуальный метод TMyClass::Print() будет перегружен. Скрипт, содержащий определения этих двух классов, назовем `script4.C`.

```
#include<iostream>
class TMyClass {
private:
float fX;
float fY;
public:
TMyClass() fX = fY = -1;
virtual void Print() const;
void SetX(float x) fX = x;
void SetY(float y) fY = y;
};
void TMyClass::Print() const
{
cout << "fX = " << fX << ",fY = " << fY << endl;
}


class TChild : public TMyClass {
public:
void Print() const;
};
void TChild::Print() const
{
cout << "This is TChild::Print()" << endl;
TMyClass::Print();
}
```

Загрузим файл в среду интерпретатора.

```
root[] .L script4.C
```

Создадим объект класса `TChild`.

```
root[] TMyClass *a = new TChild
```

Вызовем для него метод `Print()`

```
root[] a->Print()
```

Проверим выполнение методов родительского класса для нашего объекта `a`:

```
root [] a->SetX(10)
root [] a->SetY(12)
root [] a->Print()
```

Посмотрим определение созданного класса:

```
root [] .class TMyClass
```

Таким образом убеждаемся, что интерпретируемый класс ведет себя точно так же, как откомпилированный класс. Есть два ограничения для классов, созданных в скрипте. Они не могут наследовать от `TObject` и, следовательно, они не могут быть записаны в ROOT файл.

4.3.5. Отладка скриптов

CINT имеет возможность отладить скрипты, содержащие функции, посредством установки контрольных точек и способности к пошаговому выполнению кода с выводом на печать промежуточных значений переменных. Для этого командой `.b` устанавливается точка прерывания на отлаживаемый метод или строку. Далее метод запускается на выполнение, и когда CINT попадает на точку прерывания, он возвращает командную строку, на которой можно запускать любую из команд отладчика, например `.s`, `.c`, `.p`, `.db`. Смотри пример на стр.89 [1].

4.4. ACLiC - автоматический компилятор библиотек для CINT

Для работы со скриптами в ROOT помимо CINT можно использовать C++ компилятор. Преимущества этого способа состоят в существенно большей скорости выполнения скриптов и возможности использования конструкций C++, которые полностью не поддерживаются CINT. ACLiC создает CINT словарь и динамическую библиотеку из вашего C++ скрипта. Он также добавляет классы и функции, объявленные в `include` файлах с тем же самым именем, как файл скрипта, и любым из расширений `.h`, `.hh`, `.hpp`, `.hxx`, `.hPP`, `.hXX`.

При использовании ACLiC в скрипт необходимо добавить `include` инструкции для всех классов, используемых в скрипте. Далее, загружаем скрипт и добавляем в конец файла знак "+".

```
root [] .L MyScript.C+
```

Опция + генерирует динамическую библиотеку и именует ее, беря название файла, но заменяя точку перед расширением символом подчеркивания и добавляя расширение `.so`, характерное для динамических библиотек - `MyScript_C.so`.

При выполнении + команды, если дата создания header файла скрипта изменилась, динамическая библиотека перестраивается. Но даты создания всех включенных файлов автоматически не проверяются. Чтобы гарантировать, что динамическая библиотека восстановлена после всех изменений, следует использовать ++ синтаксис:

```
root [] .L MyScript.C++
```

Чтобы строить, загружать и выполнять функцию с тем же именем, что и файл, используйте команду `.x (.X)`. Функция при этом может иметь параметры.

```
root [] .x MyScript.C+ (100)
```

Вместо `.L` можно использовать команду `gROOT->LoadMacro("MyScript.C+")`. Обычно это применяется, если в одном скрипте нужно использовать ACLiC, чтобы скомпилировать и загрузить другой скрипт. Опции + и ++ имеют прежнее значение.

Важное замечание: Нельзя вызывать ACLiC со скриптом, имеющим функцию с именем `main()`.

4.5. CINT - генератор словаря ROOT

В ROOT большинство классов наследует от основного класса `TObject`. Этот класс обеспечивает заданное по умолчанию поведение и протокол для всех объектов в системе. В частности, `TObject` обеспечивает абстрактные функции-члены для следующих операций:

- ввод-вывод объекта - Read(), Write();
- обработка ошибок - Warning(), Error();
- возможность сортировки - IsSortable();
- осмотр - Inspect();
- печать - Print();
- рисование - Draw();
- способ выделения памяти - IsOnHeap();
- просмотр объекта - Browse ().

Несмотря на идентификацию типа во время выполнения (RTTI) доступ к информации относительно структуры данных, переменных и функций объекта не обеспечивается во время работы программы. Поэтому должен существовать дополнительный механизм для сбора этой информации. Для этого ROOT соединяется с каждым классом через специальный объект, описывающий его структуру. Этот специальный объект является представителем класса `TClass`, который содержит следующую информацию о классе:

- имя и заголовок класса;
- размер в байтах;

- его родительский класс(ы);
- имена, заголовки и дескрипторы его переменных;
- имена и синтаксис его функций-членов;
- адрес объекта, используемого для создания нового объекта.

Таким образом с помощью CINT образуется так называемый словарь класса. То есть CINT помимо функций интерпретатора командной строки и процессора скриптов является еще и генератором словаря. Словарь класса генерируется утилитой `rootcint`.

5. Графические объекты

5.1. Объекты TCanvas и TPad

5.1.1. Основное графическое окно

Объекты ROOT, наследующие от `TObject`, который имеет виртуальный метод `Draw()`, могут быть визуализированы (нарисованы) в специальном окне, которое называется canvas[3] [kænvas]. Это графическое окно реализовано с помощью класса `TCanvas`. Конструктор

```
TCanvas(const char* name, const char* title, Int_t wtopx, Int_t
wtopy, Int_t ww, Int_t wh)
```

где `wtopx, wtopy` - координаты левого верхнего угла окна в пикселях, `ww, wh` - размеры окна по оси `X` и `Y` в пикселях. Если определены сразу несколько объектов `TCanvas`, то одновременно только один из них является активным. Объект выводится на активный `TCanvas` командой `object.Draw()`. Если открытых графических окон нет, то при выполнении этой команды устанавливается `TCanvas` с именем `c1`, заданный по умолчанию.

Основное графическое окно имеет строку главного меню с большим количеством опций. Рассмотрим некоторые из них.

Меню "File" дает возможность открыть файл специального ROOT формата (о котором будет идти речь в разделе"Ввод-вывод")- пункт "Open...", спасти изображение на экране в файл формата PostScript - пункты "Save As canvas.ps" или "Save As canvas.eps", спасти команды , создавшие это изображение,в скрипт - пункт "Save As canvas.C", а также в специальный ROOT файл - пункт "Save As canvas.root".

При выборе пункта "Editor" в меню "Edit" стартует встроенный в графический редактор, который предназначен для вывода и редактирования простейших графических объектов. Он будет рассмотрен в отдельном параграфе данного раздела.

[3]Одно из значений английского слова "canvas" - "холст". Мы будем выводить в этом графическом окне наши графики, гистограммы и детали физических установок так же, как художник рисует на холсте свои картины.

Меню "View" позволяет просмотреть набор доступных цветов "Colors" , маркеров "Markers" и фонтов "Fonts" и их идентификационные номера.

Пункт "Event status" меню "Option" вызывает так называемую строку состояния, расположенную внизу объекта TCanvas. На ней показывается имя и заголовок объекта, на который указывает курсор, координаты курсора, а также дополнительная информация об объекте в зависимости от вида объекта.

Меню "Help" предоставляет пользователю различного рода информацию об объектах ROOT.

5.1.2. Объект TBrowser

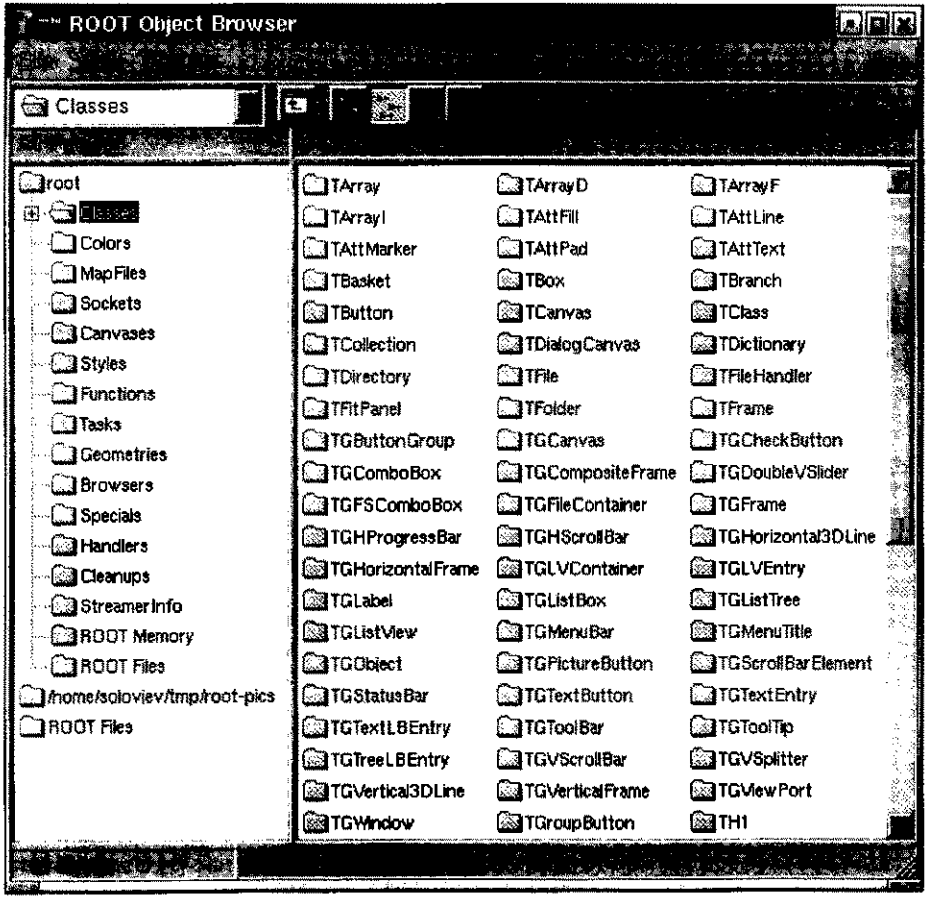

Рис. 1: Окно просмотра

С помощью строки главного меню можно вызвать также браузер ROOT (см. рис. 1). При выборе пункта "Start Browser" меню "Inspect" стартует окно просмотра (или программа просмотра) `ROOT Object Browser` - объект класса `TBrowser`. Окно просмотра, показывающее объекты ROOT, доступные для просмотра, можно также вызвать из командной строки

```
root [] TBrowser b
```

5.1.3. Деление основного окна на подокна

Основное графическое окно (объект класса `TCanvas`) можно разделить на несколько подокон, которые могут содержать другие подокна или графические объекты (см. рис. 2). Подокно называется pad [pæd] (дорожка, заплатка) и реализовано с помощью класса `TPad`. Конструктор:

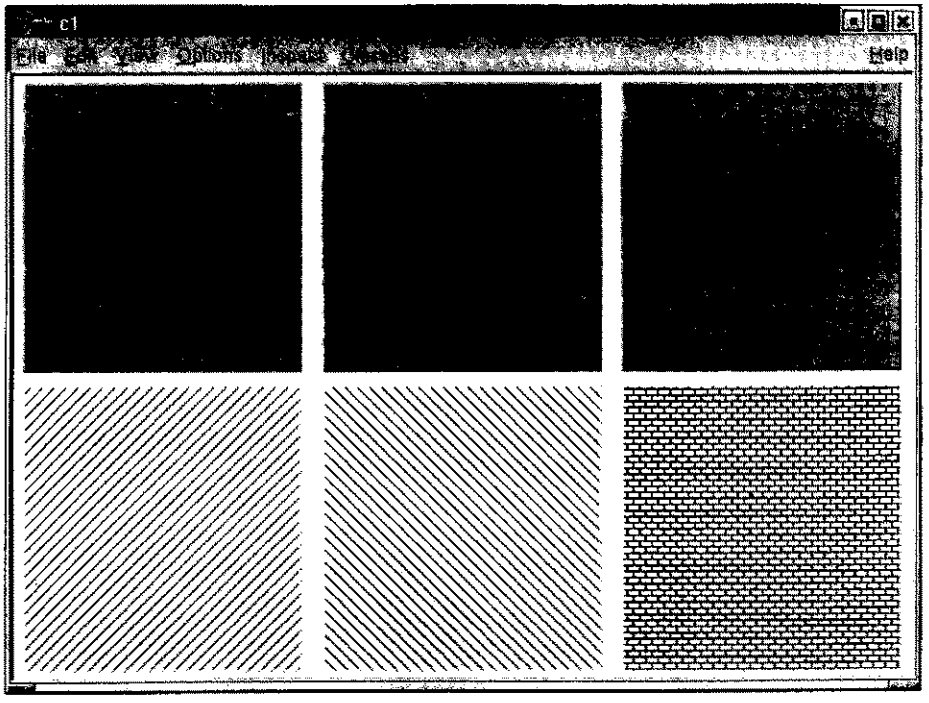

Рис. 2: Пример разбиения основного графического окна

```
TPad(const char* name, const char* title, Double_t xlow, Double_t
ylow, Double_t xup, Double_t yup, Color_t color = -1, Short_t
bordersize = -1, Short_t bordermode = -2)
```

Параметры: `xlow` - x координата нижней левой точки подокна в системе координат TPad, `yloy` - y координата этой точки, `xup` - x координата верхней правой точки подокна в системе координат TPad, `yup` - y координата этой точки, `color` - цвет подокна, `bordersize` - размер границы подокна в пикселях, `bordermode` - тип границы. Деление TCanvas на TPad может производиться несколькими способами. Можно строить подокна на объекте TCanvas, явно задавая их размер и положение, пользуясь при этом конструктором и методом `Draw()`. Второй способ состоит в автоматическом делении на горизонтальные и вертикальные ряды. Например, команда

```
root [] c1->Divide(3,2)
```

разделит TCanvas на три столбца и две строки. Вообще же метод `Divide()` имеет 5 параметров, третий и четвертый из них задают поля между подокнами (в единицах, при которых родительский TCanvas имеет размер 1 x 1), а пятый - цвет полученных объектов TPad. Сгенерированные подокна получат имена `c1_i`. В нашем случае:`c1_1, c1_2, ..., c1_6`.

Объект класса TPad - графический контейнер, имеющий связанный список указателей на объекты, которые он содержит. Рисование объекта есть не что иное, как добавление указателя на объект в этот список.

5.1.4. Взаимодействие с графическими объектами

Когда объект нарисован, можно с ним взаимодействовать. Преобразование объекта на экране (в окне TCanvas) также преобразует его в памяти. Фактически мы взаимодействуем с реальным объектом, а не с его копией на экране. Перемещение или изменение размеров объекта делается **левой** кнопкой мыши. Курсор может изменять свою форму, показывая, что сейчас может быть сделано.

Средняя кнопка мыши применяется для выбора объекта, то есть придания ему статуса активного. Чтобы активизировать графическое окно из командной строки, введите команду

```
root [] c1->cd()
```

В случае объектов TCanvas и TPad, когда они становятся активными, их границы высвечиваются. Глобальная переменная **gPad** всегда указывает на активный TPad.

При наведении курсора на объект и нажатии **правой** кнопки мыши появляется контекстное меню этого объекта, содержащее его основные методы. Чтобы метод появился в контекстном меню объекта, в заголовочном файле класса этот метод должен быть отмечен `//*MENU*`. Контекстное меню разделено линиями. Сначала идут методы этого класса, во втором отделении - методы родительских

классов, и последующие разделы содержат методы других родительских классов в случае многократного наследования. При выборе из контекстного меню метода, который имеет параметры, появится диалоговое окно, в которое пользователь должен будет ввести определенное значение соответствующей опции.

5.1.5. Системы координат объекта TPad

В графических окнах используются три системы координат: координаты пользователя, нормализованные координаты (NDC) и координаты в пикселях.

Большинство методов объекта TPad использует **координаты пользователя**. По умолчанию левый нижний угол пустого выведенного TPad имеет координаты (0,0), и диапазон их по осям x и y устанавливается от 0 до 1. Далее, используя метод TPad::Range(Float_t x1, Float_t y1, Float_t x2, Float_t y2), можно определить новые координаты пользователя, например

```
root [] gPad->Range(0,100,-50,50)
```

Нормализованные координаты (NDC) независимы от размера окна и системы координат пользователя. Левый нижний угол объекта TPad имеет координаты (0,0), и диапазон их по осям x и y устанавливается от 0 до 1. Некоторые графические объекты имеют методы DrawObjectNDC() для вывода в системе координат NDC, даже если предварительно были установлены отличные от нее координаты пользователя. Например, если Вы хотите выводить текст всегда в одном и том же месте экрана, независимо от того, какие координаты установлены, используйте метод DrawTextNDC().

Пиксельная система координат. Левый верхний угол объекта TPad имеет координаты (0,0), все размеры определяются в пикселях.

Для преобразования между системами координат TPad существует несколько методов. Пусть (px, py) - координаты точки в пиксельной системе, (ux, uy) - в системе координат пользователя, (ndcx, ndcy) - в нормализованной системе координат. Методы PixeltoX(Int_t px), PixeltoY(Int_t py) преобразуют пиксельные координаты в координаты пользователя (тип возвращаемой величины - Double_t). Методы UtoPixel(Float_t ndcx), VtoPixel(Float_t ndcy) преобразуют нормализованные координаты в пиксельные (тип возвращаемой величины - Int_t). Методы XtoPixel(Double_t ux), YtoPixel(Double_t uy) преобразуют координаты пользователя в пиксельные координаты (тип возвращаемой величины - Int_t).

Линейная или логарифмическая шкала есть атрибут TPad, а не какого-либо изображения на нем. Чтобы установить логарифмическую шкалу, например в направлении x, наберите команду

```
root [] gPad->SetLogx(1)
```

Чтобы вернуться к линейной шкале, например в направлении y, наберите в командной строке

```
root [] gPad->SetLogy(0)
```

Можно также вызвать эти методы через контекстное меню TPad или TCanvas.

5.1.6. Обновление TPad

По причинам эффективности обновление объекта TPad не происходит с каждым изменением. Автоматическая перерисовка происходит в следующих случаях:

1. Наведение курсора на TPad и щелканье мышью, например при изменении его размеров.
2. Окончание выполнения скрипта.
3. Добавление нового графического примитива или изменение некоторых старых, например названия или заголовка какого-либо объекта.
4. Вызов метода `Draw()`.

Можно явно вызвать обновление графического подокна вызовом метода `TPad::Modified`. Последующий вызов `TCanvas::Update` просматривает список объектов TPad и перерисовывает те из них, которые были объявлены измененными.

5.2. Графические примитивы

5.2.1. Класс TLine

Линия определяется координатами начальной `x1, y1` и конечной `x2, y2` точек. Конструктор:

```
TLine(Double_t x1, Double_t y1, Double_t x2, Double_t y2)
```

5.2.2. Класс TArrow

Стрелка определяется координатами начальной `x1, y1` и конечной `x2, y2` точек, размером `arrowsize` в процентах от высоты объекта TPad и своим видом - `option`. Значения, принимаемые `option` - ">", "<", "|>", "<|", "<>", "<|>". Конструктор:

```
TArrow(Double_t x1, Double_t y1, Double_t x2, Double_t y2, Float_t
arrowsize, Option_t* option)
```

Метод `SetFillColor(icolor)` устанавливает цвет заполнения стрелки. По умолчанию `icolor =0`. Метод `SetAngle(angle)` устанавливает угол между двумя сторонами стрелки, `angle` выражается в градусах, по умолчанию его значение 60.

5.2.3. Класс TPolyLine

Ломаная линия определяется набором точек, между которыми расположены ее сегменты, n - количество точек, x и y - массивы с координатами точек. Конструктор:

```
TPolyLine(Int_t n, Double_t* x, Double_t* y)
```

5.2.4. Класс TEllipse

Эллипс определяется своим центром x1, y1 и двумя радиусами r1 и r2. Эллипс может вращаться и быть усеченным, phimin и phimax - минимальный и максимальный угол. Конструктор:

```
TEllipse(Double_t x1, Double_t y1, Double_t r1, Double_t r2, Double_t
phimin, Double_t phimax)
```

5.2.5. Класс TBox

Плоский прямоугольник определяется своими нижними левыми координатами x1, y1 и верхними правыми координатами x2, y2. Конструктор:

```
TBox(Double_t x1, Double_t y1, Double_t x2, Double_t y2)
```

5.2.6. Класс TWbox

Объемный прямоугольник определяется своими нижними левыми координатами x1, y1, верхними правыми координатами x2, y2, размером границы bordersize и режимом границы bordermode. Конструктор:

```
TWbox(Double_t x1, Double_t y1, Double_t x2, Double_t y2, Int_t
color, Int_t bordersize, Int_t bordermode)
```

5.2.7. Класс TMarker

Маркер определяется координатами x, y и типом marker. Конструктор:

```
TMarker(Double_t x, Double_t y, Int_t marker)
```

Метод SetMarkerSize(size) устанавливает размер маркера, size задается в пикселях.

5.2.8. Класс TPolyMarker

Набор маркеров определяется двумя массивами из n элементов, содержащими координаты точек x[i] и y[i]. Конструктор:

```
TPolyMarker(Int_t n, Double_t* x, Double_t* y, Option_t* option)
```

5.2.9. Класс TCurlyLine

Волнистая линия определяется координатами начальной x1, y1 и конечной x2, y2 точек, длиной волны wavelenght и амплитудой amplitude. Две последние величины задаются в процентах от высоты объекта TPad. Конструктор:

```
TCurlyLine(Double_t x1, Double_t y1, Double_t x2, Double_t y2,
Double_t wavelenght, Double_t amplitude)
```

5.2.10. Класс TCurlyArc

Волнистая дуга определяется своим центром x1, y1 и радиусом rad. Длина волны wavelenght и амплитуда amplitude задаются процентами от длины линии. Угол начала phimin и угол окончания phimax дуги задаются в градусах. Конструктор:

```
TCurlyArc(Double_t x1, Double_t y1, Double_t rad, Double_t phimin,
Double_t phimax, Double_t wavelenght, Double_t amplitude)
```

5.2.11. Класс TLatex

Этот класс был построен для изображения текста в графических окнах TCanvas или TPad. Его синтаксис подобен математическому режиму известной системы для верстки текстов с формулами Latex. Нижние и верхние индексы создаются _ и ^ командами. Чтобы получить строчные греческие буквы, необходимо перед названием буквы, записанным латиницей, поставить знак #. Для получения прописной греческой буквы напечатайте первый символ названия прописной буквой. Например:

#alpha – α, #Omega – Ω, #beta – β, #Delta – Δ.

Для изображения дробей используется команда $\#frac\{$числитель$\}\{$знаменатель$\}$. Команда $\#sqrt\{\}$ производит квадратный корень от своего параметра. Необязательный первый параметр используется для изображения корней другой степени $\#sqrt[]\{\}$. Математические символы, диакретические знаки и стрелки также изображаются с помощью команд, аналогичных командам Latex (см.стр.118 [1]).

5.2.12. Класс TPaves

Текст может изображаться в графическом окне самостоятельно, а может располагаться в объемных рамках (панелях). Pave(панель) - это блок с размером границы и характером расположения тени.

5.2.13. Класс TPaveLabel

Панель, содержащая только одну строку текста, определяется координатами нижнего левого угла x1, y1, верхнего правого угла x2, y2, изображаемым тек-

стом label и видом панели option. Значения, принимаемые option: "T" - тень сверху, "B" - тень снизу, "R" - тень справа, "L" - тень слева, "NDC" - x1, y1, x2, y2 задаются в NDC, "ARC" - углы панели округляются. Конструктор:

```
TPaveLabel(Double_t x1, Double_t y1, Double_t x2, Double_t y2, const
char *label, Option_t *option)
```

Метод SetBorderSize() устанавливает размер границы. По умолчанию размер границы - 5, а option - "BR".

5.2.14. Класс TPaveText

Этот класс отличается от предыдущего только тем, что объект TPaveText может содержать несколько строк текста.

5.2.15. Класс TPavesText

Стопка текстовых панелей определяется своими координатами, числом панелей в стопке npaves и видом option. По умолчанию npaves=5, option - "BR". Конструктор:

```
TPavesText(Double_t x1, Double_t y1, Double_t x2, Double_t y2, Int_t
npaves, Option_t *option)
```

5.2.16. Пример

Приведем здесь пример кода, строящего некоторые графические объекты. Изображение, полученное на TCanvas в этом случае, показано на рис. 3.

```
root [] TArrow ar(0.1,0.1,0.3,0.3,0.05,"<|>")
root [] ar.SetAngle(45)
root [] ar.Draw()
root [] TEllipse el(0.65,0.2,0.2,0.1)
root [] el.SetFillStyle(3005)
root [] el.Draw()
root [] TWbox wb(0.1,0.6,0.4,0.9,16,10,5)
root [] wb.Draw()
root [] TMarker m(0.6,0.6,3)
root [] m.SetMarkerSize(5)
root [] m.Draw()
root [] TCurlyArc cyr(0.7,0.6,0.2,270,360)
root [] cyr.Draw()
root [] TPaveLabel pv(0.45,0.8,0.95,0.9,"Graphical objects","brarc")
root [] pv.Draw()
```

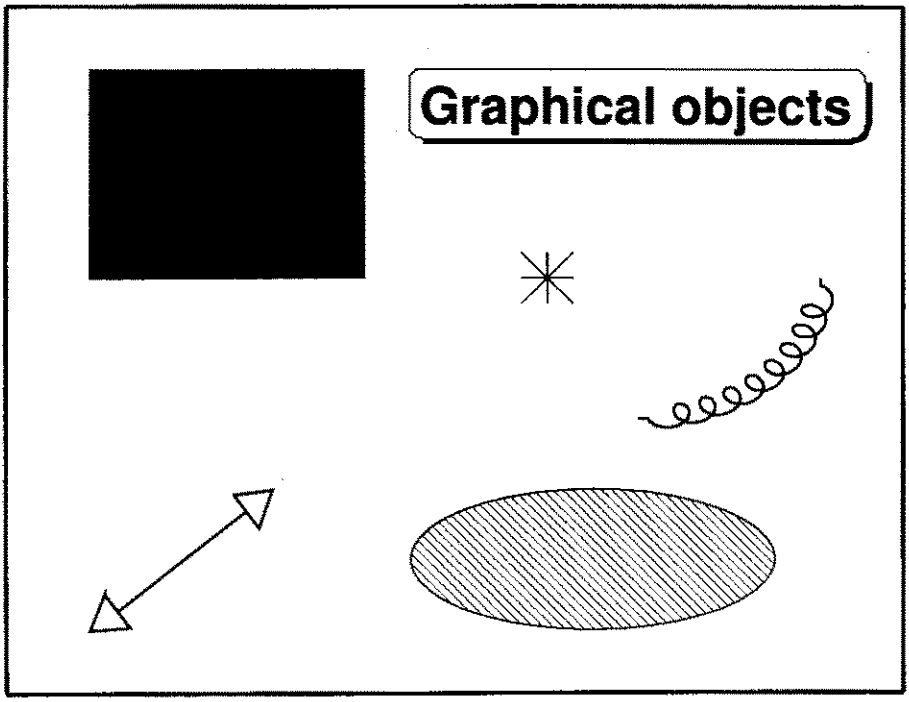

Рис. 3: Графические примитивы

5.2.17. Класс TSlider

Полоса прокрутки (сдвиг), реализованная с помощью класса `TSlider`, используется для показа развития процесса. Полоса прокрутки содержит движущийся блок, который может быть перемещен или изменен. Конструктор:

```
TSlider(const char name, const char* title, Double_t x1, Double_t y1,
Double_t x2, Double_t y2, Int_t color, Int_t bordersize, Int_t
bordermode)
```

Текущая позиция полосы прокрутки может быть найдена с помощью методов `GetMinimum()` и `GetMaximum()`, возвращаемые числа лежат в диапазоне от 0 до 1. Если вызван метод `SetMethod()`, то при перемещении полосы выполняется установленный метод или скрипт (если в качестве параметра используется строка "`.x script.C`"). В скрипте обычно считывается текущая позиция полосы прокрутки и в зависимости от нее устанавливается диапазон вывода какого-либо изображения. Пример использования объекта класса `TSlider` можно найти в `tutorials/xyslider.C`

5.2.18. Классы TAxis и TGaxis

По историческим причинам имеются два класса, реализующие ось. Во-первых, объекты оси автоматически строятся различными объектами высокого уровня, например гистограммами или графами. В этом случае ось x (y, z) гистограммы - объект класса TAxis, и она может быть получена при вызове метода TH1::GetXaxis() (TH1::GetYaxis(), TH1::GetZaxis())[4]. Ось x (y) графа вызывается методом TGraph::GetXaxis() (TGraph::GetYaxis())[5].

Также ось может быть выведена независимо от гистограммы или графа. В этом случае нужно использовать класс графического представления оси TGaxis, конструктор которого имеет вид

```
TGaxis (Double_t xmin, Double_t ymin, Double_t xmax, Double_t ymax,
Double_t wmin, Double_t wmax, Int_t ndiv = 510, Option_t* chopt,
Double_t gridlehght = 0)
```

Параметры xmin, ymin (xmax, ymax) - начальные (конечные) координаты оси в системе координат пользователя, wmin и wmax - минимальное в начале и максимальное в конце оси значения, которые будут представлены на оси, ndiv - число делений,
ndiv = N1+100*N2+10000*N3,
где N1 - число первичных делений, N2 - число вторичных делений, N3 - число третичных делений.

Параметр chopt принимает следующие значения:

- "G" - логарифмический масштаб оси;
- "B" - рисуются только метки и цифры, ось не выводится;
- "+" - метки делений выведены на положительной стороне оси (значение по умолчанию);
- "-" - метки делений выведены на отрицательной стороне оси;
- "+-"- метки делений выведены с обеих сторон оси;
- "U" - рисуется только ось и метки делений, цифры не выводятся;
- "W" - задает сетку на графическом экране;
- "=" - цифры выводятся на той же стороне оси, на которой расположены метки делений (по умолчанию на стороне напротив меток делений);
- "M" - цифры стоят в середине интервалов между метками;
- "N" - нет оптимизации разбиения оси на бины (по умолчанию это разбиение оптимизировано);
- "I" - целочисленные метки;
- "t" - создается ось с координатами в формате времени.

Если xmin = xmax, то

[4]TH1 - класс одномерных гистограмм.
[5]TGraph - класс графов.

- "R" - метки делений сдвинуты вправо относительно цифр (значение по умолчанию);
- "L" - метки делений сдвинуты влево относительно цифр;
- "C" - цифры центрированы относительно меток оси.

Если значения координат оси очень большие или очень маленькие, то по умолчанию используется экспоненциальное представление для этих чисел. Но его можно отключить методом `TAxis::SetNoExponent(kTRUE)`. Можно также изменить заданное по умолчанию максимальное число цифр, при котором координаты оси все еще не будут иметь экспоненциальное представление. Для этого используется метод `TGaxis::SetMaxDigits(Int_t maxd=5)`, `maxd` - задаваемое пользователем число цифр.

Скрипт `gaxis.C`, находящийся в каталоге `tutorials`, иллюстрирует построение осей с различными параметрами.

5.3. Графические атрибуты объектов

5.3.1. Текстовые атрибуты

Когда класс содержит текст или происходит от класса, содержащего текст, требуется, чтобы он был способен устанавливать такие атрибуты текста, как тип, размер и цвет шрифта. Поэтому такой класс должен наследовать от класса `TAttText`.

Метод `SetTextAlign(align)` устанавливает выравнивание текста, где
`align= 10 * HorizontalAlign + VerticalAlign`.
Для `HorizontalAlign` применяется следующее соглашение:

- 1-выравнивает по левому краю,
- 2-центрирует,
- 3-выравнивает по правому краю.

Для `VerticalAlign` применяется следующее соглашение:

- 1-выравнивает по нижнему краю,
- 2-центрирует,
- 3-выравнивает по верхнему краю.

Метод `SetTextAngle(angle)` устанавливает угол текста, `angle` - градусы от диагонали. Метод `SetTextColor(color)` устанавливает цвет текста, `color` - индекс цвета. Метод `SetTextFont(font)` устанавливает шрифт, `font` - код шрифта, объединяющий ID шрифта и точность: `font=10 * fontID + precision`. ID шрифта может изменяться от 1 до 14, таблицу доступных шрифтов можно найти на стр. 137 в [1]. Точность может иметь значения 0, 1, 2. От точности зависит качество масштабирования, способность к вращению шрифтов и быстрота появления надписи. Метод `SetTextSize(size)` устанавливает размер текста, `size` - размер текста, выраженный в процентах от текущего размера объекта `TPad`.

5.3.2. Атрибуты линии

Все классы, манипулирующие с линиями, должны иметь дело с атрибутами линии. Поэтому такие классы должны наследовать от класса `TAttLine`.

Метод `SetLineColor(color)` устанавливает цвет линии, `color` - индекс цвета. Метод `SetLineStyle(style)` устанавливает тип линии, `style` - индекс, принимающий значения от 1 до 4, 1 - сплошная линия, 2 - линия из тире, 3 - пунктирная линия, 4 - линия из тире и точек. Метод `SetLineWidth(width)` устанавливает ширину линии, `width` - ширина, выраженная в пикселях.

5.3.3. Атрибуты заполнения

Многие графические объекты имеют заполняющиеся области. Поэтому классы, их описывающие, должны иметь дело с заполняющими атрибутами и, следовательно, должны наследовать от класса `TAttFill`.

Метод `SetFillColor(color)` устанавливает цвет области заполнения, `color` - индекс цвета. Метод `SetFillStyle(style)` устанавливает стиль, которым область будет заполняться, `style` - индекс, принимающий значения от 3000 до 3025. Различные образцы стиля заполнения можно найти на стр.140 [1].

5.3.4. Установка графических атрибутов в интерактивном режиме

Атрибуты текста, линии и заполнения доступны через контекстное меню графических объектов. При щелчке по объекту, содержащему текст (имеющему некоторые атрибуты линии, заполняющуюся область), один из последних пунктов появляющегося меню есть `SetTextAttributes (SetLineAttributes, SetFillAttributes)`. При выборе этих пунктов появятся диалоговые окна с доступными для данных графических объектов атрибутами. Выбирая мышью необходимый тип атрибутов и нажимая затем на клавишу `Apply` диалогового окна, пользователь может добиться требуемого изображения.

5.4. Графический редактор

ROOT имеет встроенный графический редактор для рисования и редактирования графических объектов. Он показан на рис. 4. Редактор стартует при выборе пункта "Editor" в меню "Edit". При этом появляется независимое окно с меню. Выбрав требуемый пункт меню, нажмите на него левой кнопкой, далее:

- Круг (**Arc**) - перемещайте мышь, держа нажатой ее левую кнопку. Редактор будет выводить растягивающийся блок, в который затем поместит круг. Отпустите кнопку, когда нужный размер будет достигнут. Вызвав контекстное меню круга, можно преобразовать его в дугу.
- Линия или стрелка (**Line** or **Arrow**) - подведя курсор к точке начала линии, нажмите на левую кнопку мыши и, держа кнопку нажатой, перемещайте мышь до точки окончания линии, где отпустите кнопку.

- Кнопка (**Button**) - перемещайте мышь, держа нажатой ее левую кнопку. Редактор будет выводить растягивающийся блок, предлагая контур кнопки, которую Вы хотите расположить в графическом окне. Отпустите кнопку мыши, когда нужный размер будет достигнут. Введите текст надписи на кнопке и нажмите на клавишу **"Enter"**. Вызвав контекстное меню, можно установить метод, вызываемый при нажатии на эту кнопку графического окна, и другие параметры.
- Ромб (**Diamond**) - поместив курсор в нужную точку, нажмите на левую кнопку мыши и, держа кнопку нажатой, перемещайте мышь. Редактор будет выводить растягивающийся блок, в который затем поместит ромб. Отпустите кнопку, когда нужный размер будет достигнут.
- Эллипс (**Ellipse**) - перемещайте мышь, держа нажатой ее левую кнопку. Редактор будет выводить растягивающийся блок, в который затем поместит эллипс. Отпустите кнопку, когда нужный размер будет достигнут. Эллипс можно растянуть или сжать, подведя курсор к чувствительным точкам, которые высвечиваются.
- Объект TPad (**Pad**) - перемещайте мышь, держа нажатой ее левую кнопку. Редактор будет выводить растягивающийся блок, предлагая контур объекта TPad.
- Панель (**Pave**) - перемещайте мышь, держа нажатой ее левую кнопку, до достижения нужного размера панели.
- Панель с меткой (**PaveLabel**) - то же самое, как для простой панели. При завершении процесса редактор выведет вертикальную черту на панели и знак вопроса. Введите текст и нажмите на клавишу **"Enter"**.
- Панель или панели с текстом (**PaveText** or **PavesText**) - то же самое, как для простой панели. Вызвав контекстное меню объекта `TPave(s)Text`, выберите опцию `InsertText` и введите текст.
- Ломаная линия (**PolyLine**) - нажмите левой кнопкой на первую точку, переместите мышь, снова нажмите левой кнопкой на новую точку и так далее. Закройте ломаную линию двойным щелчком.
- Волнистая линия (**CurlyLine**) - то же самое, как для линии/стрелки. Вызвав контекстное меню, можно изменить амплитуду, длину и другие характеристики волны.
- Волнистая дуга (**CyrlyArc**) - нажмите левой кнопкой на точку, в которой должен располагаться центр дуги, и, держа кнопку нажатой, переместите мышь до позиции начала дуги. Редактор выведет волнистый эллипс, который можно преобразовать в волнистую дугу и изменить такие характеристики, как смещение, амплитуда, длина волны и другие с помощью контекстного меню.
- Текстовая строка (**Text/Latex**) - нажмите левой кнопкой по позиции, в которой должен быть выведен текст, затем напечатайте его и нажмите на

клавишу **"Enter"**. Текст можно перемещать, вращать и растягивать.

- Маркер (**Marker**) - нажмите левой кнопкой мыши на точку, в которой хотите разместить маркер.
- Графическая вырезка (<...**Graphical Cut**...>) - нажмите левой кнопкой по каждой точке многоугольника, ограничивающего выбранную область, по последней точке щелкните дважды.

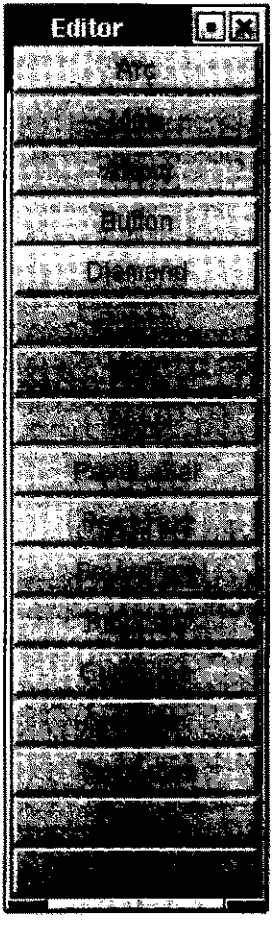

Рис. 4: Графический редактор

5.5. Копирование графических объектов

Для копирования графических объектов с одного экрана на другой в ROOT разработаны методы `DrawClone` и `DrawClonePad`. Последний из них применяется, если мы хотим сделать копию целого объекта `TCanvas` на новый экран (активный `TPad`). Список параметров в методе `TCanvas::DrawClonePad()` пустой. Метод `DrawClone` более универсален, он применяется для всех объектов, происходящих от `TObject`. Опция в методе `TObject::DrawClone (Option_t* option)` есть опция `Draw` для гистограммы или графа. Объекты `TCanvas` или `TPad`, на которые производится копирование, должны быть активными. Копии независимы от первоначальных объектов. Если изменить, например, первоначальную гистограмму, ее копия сохранит все прежние атрибуты.

Оба метода копирования доступны через контекстное меню графических объектов. Последовательность действий при этом такова:

1. Создайте новый `TCanvas` или `TPad`, на который будет производиться копирование (или активизируйте один из существующих, нажимая по нему средней кнопкой мыши).
2. Вызовите контекстное меню копируемого объекта, нажимая по нему правой кнопкой мыши, и выберите метод `DrawClone` или `DrawClonePad` (последний для полного `TCanvas`).
3. В появившемся диалоговом окне оставьте опцию пустой (или наберите одну из опций `Draw()`, если хотите, чтобы копия отличалась от оригинала) и нажмите OK.

5.6. Как создать PostScript файл

Чтобы создать PostScript файл одного изображения в графическом окне при работе в интерактивном режиме, нужно выбрать пункт "Save As canvas.ps" в меню "File" объекта `TCanvas` или набрать в командной строке

```
c1->Print("filename").
```

Эта команда сгенерирует PostScript файл с изображением `TCanvas`, указанного как `c1`. Конструктор PostScript файла

```
TPostScript(char* filename, Int_t type),
```

где `type` есть опция формата: 111 - portret, 112 - landscape, 113 - eps. Метод `Range(xsize,ysize)` устанавливает размер изображения. Метод `Text(x,y,"string")` добавляет текст `string` к изображению в позиции `x, y`.

Рассмотрим два примера, как получить многократные изображения в PostScript файле. Пусть `h1` уже существующая гистограмма. В первом случае PostScript файл будет содержать две страницы, с одним изображением на каждой странице.

```
root [] TCanvas c1
```

```
root [] TPostScript myps("my1.ps")
root [] myps.Range(20,30)
root [] h1.Draw()
root [] c1.Update()
root [] h1.Draw("lego")
root [] c1.Update()
root [] myps.Close()
```

Генерация новой страницы PostScript файла происходит автоматически, когда метод `TCanvas::Clear`, очищающий окно, вызывается командой `object->Draw()`.

Во втором случае PostScript файл содержит две страницы, каждая из которых состоит из двух изображений.

```
root [] TCanvas c1
root [] TPostScript mps("my2.ps",112)
root [] c1.Divide(2,1)
root [] mps.NewPage()
root [] c1.cd(1)
root [] h1.Draw()
root [] c1.cd(2)
root [] h1.Draw("c")
root [] c1.Update()
root [] mps.NewPage()
root [] c1.cd(1)
root [] h1.Draw("*h")
root [] c1.cd(2)
root [] h1.Draw("lego")
root [] c1.Update()
root [] mps.Close()
```

В этом случае для генерации новой страницы должен вызываться метод `TPostScript::NewPage`, поскольку вызов `object->Draw()` теперь очищает только `TPad`, а не весь `TCanvas` целиком.

5.7. Создание и изменение стиля

При создании объектов заданные по умолчанию атрибуты этих объектов принимаются от текущего стиля. Текущий стиль - объект класса `TStyle` и может быть вызван через глобальную переменную `gStyle`. ROOT обеспечивает следующие стили:

- "Default" - заданный по умолчанию;
- "Bold" - с более жирными линиями;
- "Plain" - черно-белый стиль;
- "Pub" - подходящий для публикаций;

- "Video" - подходящий для создания html изображений.

Можно переопределять заданные по умолчанию параметры стиля командами вида `gStyle->Set...` Например, для линии - `SetLineColor()`, `SetLineStyle()`, `SetLineWidth()`. Для других объектов аналогично.

Дополнительные стили создаются конструктором

`TStyle(char* name, char* title)`.

Далее следует определить атрибуты стиля, например, цвет объектов `TCanvas` и `TPad`, размер и тип их границ методами `SetCanvasColor()`, `SetPadColor()`, `SetCanvasBorderSize()`, `SetPadBorderMode()`, цвет маркеров - `SetMarkerColor()`, угол наклона текста - `SetTextAngle()`, стиль статистики - `SetStatStyle()`, всю палитру цветов - `SetPalette(Int_t ncolor)` (`ncolor` - номер палитры, он может изменяться от 0 до 49) и так далее. Полный список атрибутов, которые могут быть установлены в одном стиле, находится в описании класса `TStyle`. Созданный стиль устанавливается командой `gROOT->SetStyle(name)`.

Если Вы читаете объект из файла, то его атрибуты будут приниматься от стиля, который был установлен при записи этого объекта в файл. Чтобы установить текущий стиль для ранее созданных объектов, следует до чтения объектов из файла вызвать `gROOT->ForseStyle()`.

6. Графы

Граф - графический объект, созданный двумя массивами, содержащими `x` и `y` координаты `n` точек. Имеются несколько классов графа: `TGraph`, `TGraphErrors`, `TGraphAsymmErrors` и `TMultiGraph`.

6.1. Класс TGraph

Граф определяется числом точек `n` и массивами координат `x`, `y`, заданными заранее. Конструктор

`TGraph (Int_t n, Float_t x, Float_t y)`

Координаты могут быть представлены также массивами с двойной точностью. Параметры метода `Draw()` для графа могут быть следующими:

1. "A" - выводятся оси графа,
2. "L" - между точками рисуется простая ломаная линия,
3. "C" - между точками рисуется сглаженная кривая линия,
4. "P" - в каждой точке рисуется текущий маркер,
5. "*" - в каждой точке рисуется звезда,
6. "B" - в каждой точке рисуется гистограмма,
7. "F" - выводится заполняющая область.

Для двух последних опций заполняющий цвет по умолчанию белый, поэтому разумно установить заполняющий цвет до рисования графика, используя метод `SetFillColor(icolor)`. Можно установить ширину линии (`SetLineSize(size)`), цвет линии (`SetLineColor(icolor)`), стиль маркера (`SetMarkerStyle(mstyle)`), его размер (`SetMarkerSize(size)`) и цвет (`SetMarkerColor(icolor)`). При рисовании двух и более графов в одном объекте `TPad` оси следует выводить только один раз и, следовательно, в методе `Draw()` опцию "A"задавать только для одного графа.

6.2. Класс TGraphErrors

Граф с симметричными ошибками определяется, помимо числа точек и их координат, еще и массивами с координатами ошибок `ex`, `ey`. Конструктор

```
TGraph (Int_t n, Float_t x, Float_t y, Float_t ex, Float_t ey)
```

В этом случае параметры метода `Draw()` такие же, как для графа без ошибок. И кроме них может быть использована опция "Z", которая стирает маленькие линии в конце полос ошибок.

6.3. Класс TGraphAsymmErrors

Граф с асимметричными ошибками определяется шестью массивами: `x`, `y` - координаты точек, `exl`, `exh` - массив левых и правых координат `X`-полосы ошибок, `eyl`, `eyh` - массив нижних и верхних координат `Y`-полосы ошибок. Конструктор

```
TGraph (Int_t n, Float_t x, Float_t y, Float_t exl, Float_t exh,
Float_t eyl, Float_t eyh)
```

6.4. Класс TMultiGraph

Объект `TMultiGraph` есть совокупность нескольких `TGraph`, которая имеет эти объекты в своем списке. Чтобы добавить к списку новый граф, используйте метод `TMultiGraph::Add()`.

6.5. Установка заголовков оси графа

Чтобы установить заголовок оси графа, сначала нужно создать и вывести сам граф, только после этого образуется объект оси графа. Методы `GetXaxis()` и `GetYaxis()` вызывают `X` и `Y` оси существующего графа. Метод `TAxis::SetTitle (title)` устанавливает заголовок оси. Методом `TAxis::CenterTitle()` можно отцентрировать заголовок.

6.6. Легенда для графа

Объект TLegend - панель для введения какого-либо текста. Легенду может иметь не только граф, но и любой объект, имеющий линию, маркер или заполняющий атрибут. Конструктор:

```
TLegend(Double_t x1, Double_t y1, Double_t x2, Double_t y2, const char
*header, Option_t *option)
```

Опции x1, y1, x2, y2 задают координаты легенды в текущем объекте TPad (в координатах NDC по умолчанию), header - заголовок легенды, option - такие же, как у объектов TPave. Как только блок легенды создан, можно добавить текст AddEntry() методом:
`TLegend::AddEntry(TObject *obj, const char *label, Option_t *option)`

- *obj - указатель на объект, имеющий линию, маркер или заполняющий атрибут (например граф или гистограмму);
- label- метка, которая будет привязана к объекту;
- option:
  - "l" выводит линию, связанную с атрибутами линии obj, если obj имеет их (наследует от TAttLine),
  - "p" выводит полимаркер, связанный с атрибутами маркера obj, если obj имеет их (наследует от TAttMarker),
  - "f" выводит блок с заполнением, связанный с заполняющими атрибутами obj, если obj имеет их (наследует от TAttFill).

6.7. Пример

Приведем на рис. 5 пример графа и код интерактивного сеанса ROOT, в котором был получен этот граф. Другие примеры графов можно найти в каталоге tutorials, например в скриптах graph.C, gerrors.C, zdemo.C, gerrors2.C.

```
root [] Int_t n=20
root [] Double_t x[n],y1[n],y2[n],ex[n],ey2[n]
root []{
for(Int_t i=0; i<n; i++) {
x[i]=i*0.5;
y1[i]=5*cos(x[i]+0.2);
y2[i]=5*sin(x[i]+0.2);
ex[i]=0.3;
ey2[i]=0.07*i;
}
}
```

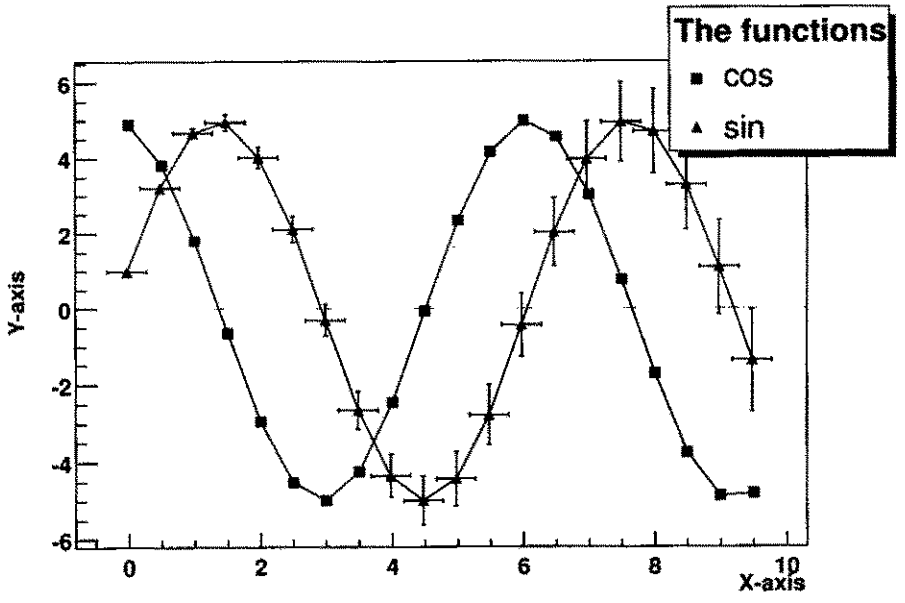

Рис. 5: Граф

```
root [] g1=new TGraph(n,x,y1)
root [] g2=new TGraphErrors(n,x,y2,ex,ey2)
root [] g1->SetMarkerStyle(21)
root [] g2->SetMarkerStyle(22)
root [] mg=new TMultiGraph()
root [] mg->Add(g1)
root [] mg->Add(g2)
root [] mg->Draw("ALP")
root [] mg->GetXaxis()->SetTitle("X-axis")
root [] mg->GetYaxis()->SetTitle("Y-axis")
root [] mg->GetYaxis()->CenterTitle()
root [] mg->Draw("ALP")
root [] l=new TLegend(0.75,0.75,0.99,0.99)
root [] l->Draw()
root [] l->SetHeader("The functions")
root [] l->AddEntry(g1,"cos","p")
root [] l->AddEntry(g1,"sin","p")
root [] l->Draw()
```

7. Гистограммы

7.1. Классы гистограмм

ROOT поддерживает следующие типы гистограмм:
одномерные гистограммы:

1. TH1C - гистограммы с одним байтом на канал. Максимальное содержание бина - 255;
2. TH1S - гистограммы с одним числом типа short на канал. Максимальное содержание бина - 65535;
3. TH1F - гистограммы с одним числом типа float на канал. Максимальная точность 7 знаков;
4. TH1D - гистограммы с одним числом типа double на канал. Максимальная точность 14 знаков;

двумерные гистограммы:

1. TH2C - гистограммы с одним байтом на канал. Максимальное содержание бина - 255;
2. TH2S - гистограммы с одним числом типа short на канал. Максимальное содержание бина - 65535;
3. TH2F - гистограммы с одним числом типа float на канал. Максимальная точность 7 знаков;
4. TH2D - гистограммы с одним числом типа double на канал. Максимальная точность 14 знаков;

трехмерные гистограммы:

1. TH3C - гистограммы с одним байтом на канал. Максимальное содержание бина - 255;
2. TH3S - гистограммы с одним числом типа short на канал. Максимальное содержание бина - 65535;
3. TH3F - гистограммы с одним числом типа float на канал. Максимальная точность 7 знаков;
4. TH3D - гистограммы с одним числом типа double на канал. Максимальная точность 14 знаков;

профильные гистограммы:

1. TProfile - одномерная профильная гистограмма;
2. TProfile2D - двумерная профильная гистограмма.

Все классы гистограмм получены из основного класса TH1. Классы гистограмм также наследуют от класса массива TArray.

7.2. Создание гистограмм

Одномерная гистограмма определяется именем name, заголовком title, числом бинов nbins и границами по оси x - xlow, xup. Конструктор для гистограмм с числами с плавающей точкой:

```
TH1F(const char name, const char* title, Int_t nbins, Float_t xlow,
Float_t xup)
```

Конструктор для гистограмм классов TH1C, TH1S, TH1D аналогичен конструктору для класса TH1F.

Двумерная гистограмма определяется именем name, заголовком title, числом бинов и границами по оси x - nbinsx, xlow, xup и числом бинов и границами по оси y - nbinsy, ylow, yup. Конструктор для гистограмм с числами с плавающей точкой:

```
TH1F(const char name, const char* title, Int_t nbinsx, Float_t xlow,
Float_t xup, Int_t nbinsy, Float_t ylow, Float_t yup)
```

Гистограммы могут быть также созданы вызовом метода Clone() у существующей гистограммы, чтением гистограммы из файла и созданием проекции из двумерной или трехмерной гистограммы.

Все типы гистограмм поддерживают фиксированный или переменный размер бина. Двумерные гистограммы могут иметь фиксированный размер бина по оси x и переменный размер бина по оси y и наоборот. Конструктор гистограммы с переменным размером бина имеет вид:

```
TH1(const char name, const char* title, Int_t nbins, Float_t* xbins)
```

Здесь xbins - массив нижних границ для каждого бина. Размерность этого массива - nbins+1.

7.3. Заполнение гистограмм

Для заполнения гистограмм используется метод Fill(). Пусть h1 - одномерная гистограмма, а h2 и h3 соответственно двумерная и трехмерная. x, y, z - значения, которыми заполняется гистограмма, w - вес. Тогда эти гистограммы будут заполняться при выполнении следующих команд:

```
root [] h1.Fill(x)
root [] h1.Fill(x,w)
root [] h2.Fill(x,y)
root [] h2.Fill(x,y,w)
root [] h3.Fill(x,y,z)
root [] h3.Fill(x,y,z,w)
```

Fill() метод вычисляет номер бина, соответствующий данному значению x, y или z и увеличивает этот бин на заданный вес. Метод AddBinContent(Int_t

bin, Int_t w) увеличивает содержание бина с номером bin на вес w. Метод SetBinContent(Int_t bin, Int_t content) устанавливает содержание content бина с номером bin. Метод GetBinContent(Int_t bin) получает содержание бина с номером bin. Гистограммы всех типов могут иметь положительное и/или отрицательное содержание бинов.

Для заполнения гистограмм случайным распределением используется метод FillRandom(const char* fname, Int_t ntimes=5000). Параметр fname определяет имя функции (объекта класса TF), соответственно которой происходит распределение. ROOT имеет четыре встроенные функции, используемые для случайных распределений:

1. gaus - распределение Гаусса - $F(x) = p0 * exp(-0.5 * ((x - p1)/p2)^2)$,
2. expo - экспонента с двумя параметрами - $F(x) = exp(p0 + p1 * x)$,
3. poly - полиномиал степени N - $F(x) = p0 + p1 * x + p2 * x^2 + ...$,
4. landau - функция Ландау.

Параметр ntimes указывает, сколько случайных чисел должно быть сгенерировано.

7.4. Рисование гистограмм

Рисование гистограмм производится с помощью метода TH1::Draw. По умолчанию вызов TH1::Draw очищает объект TPad от всех объектов перед рисованием каждого нового образа гистограммы. Гистограммы используют текущий стиль gStyle, который является глобальным объектом класса TStyle. Чтобы изменить текущий стиль для гистограмм, класс TStyle обеспечивает множество методов. Методы SetFillStyle(Int_t styl=0) и SetFillColor(Int_t color=1) устанавливают стиль и цвет заполнения гистограммы. Методы SetLineStyle (Int_t styl=0), SetFillColor(Int_t color=1) и SetLineWidth(Int_t width=1) устанавливают стиль, цвет и ширину контура гистограммы. Чтобы распространить замену текущего стиля на предварительно созданную гистограмму, необходимо вызвать для нее UseCurrentStyle() метод. Изображение изменится после модификации объекта TPad, то есть после щелчка внутри него или после вызова TPad::Update().

7.4.1. Параметры метода Draw()

Следующие опции рисования поддержаны для всех классов гистограмм:

1. "axis" - вывести только оси;
2. "hist" - вывести только контур гистограммы без ошибок;
3. "same" - наложить гистограмму на предыдущее изображение в тот же самый TPad;

4. "lego" - вывести график, в котором содержание ячейки представлено в виде трехмерного блока с высотой, пропорциональной содержанию ячейки;
5. "lego1" - вывести график, в котором содержание ячейки представлено в виде трехмерного блока с высотой, пропорциональной содержанию ячейки, и одной окрашенной гранью;
6. "lego2" - вывести график с использованием цветов, чтобы показать содержание ячейки.

Для одномерных гистограмм можно использовать дополнительно следующие опции:

1. "b" - вывести прозрачную гистограмму;
2. "l" - провести линию через содержание бина;
3. "c" - провести гладкую кривую через бины гистограммы;
4. "p" - вывести маркеры в каждом бине, используя текущий стиль маркера гистограммы;
5. "*h" - вывести гистограмму со звездочкой в каждом бине;
6. "e1" - вывести только ошибки гистограммы;
7. "e2" - вывести ошибки в виде прозрачных прямоугольников;
8. "e3" - вывести прозрачную область через конечные точки вертикальных линий ошибок;
9. "e4" - вывести сглаженную область через конечные точки линий ошибок.

Следующие опции рисования поддержаны для классов двумерных гистограмм:

1. "scat" - вывести для каждой ячейки число точек, пропорциональное содержанию ячейки (значение по умолчанию);
2. "arr" - вывести для каждой ячейки стрелку, показывающую градиент между смежными ячейками;
3. "box" - вывести для каждой ячейки прямоугольник с поверхностью, пропорциональной содержанию ячейки;
4. "col" - вывести для каждой ячейки прямоугольник, цвет которого изменяется в зависимости от содержания ячейки;
5. "colz" - то же самое, как "col", но с палитрой цветов;
6. "cont" - вывести контурный график с использованием цветов поверхностей, чтобы отличить контуры;
7. "contz" - то же самое, как "cont", но с палитрой цветов;
8. "cont1" - вывести контурный график с использованием цветов линий, чтобы отличить контуры;
9. "cont2" - вывести контурный график с использованием типов линий, чтобы отличить контуры;
10. "cont3" - вывести контурный график с использованием одного и того же типа линии;

11. "cont4" - вывести контурный график с использованием цветов, заполняющих области;
12. "surf" - вывести поверхностный график;
13. "surf1" - вывести поверхностный график с использованием цветов, чтобы показать содержание ячейки;
14. "surf2" - вывести поверхностный график с использованием цветов и удаленными линиями между поверхностями;
15. "surf3" - вывести поверхностный график вместе с контурным графиком, использующим цвета;
16. "surf4" - вывести поверхностный график с использованием Gouraud оттенения;
17. "text" - вывести содержание ячейки как текст.

Большинство параметров связывается без пробелов или запятых, например:

```
h->Draw("e1same")
```

Параметры независимы от регистра. Помимо декартовых координат, используемых по умолчанию, поверхностные и lego графики можно получать и в других системах координат, соединяя соответствующие опции с опциями "sirf..." и "lego...". Это действительно и для одномерных, и для двумерных гистограмм. Используются следующие координаты:

1. "cyl" - цилиндрические;
2. "pol" - полярные;
3. "sph" - сферические;
4. "psr" - psevdo-rapid/phi координаты.

Параметры "cont", "surf" и "lego" имеют по умолчанию 20 равноудаленных уровней. Можно изменять число уровней методом `SetContour(Int_t nlevels)`.

Приведем пример построения одномерных и двумерных гистограмм. Результат выполнения этого кода показан на рис. 6.

```
root [] TCanvas c1
root [] c1.Divide(2,2)
root [] c1_1.cd()
root [] TH1F h1("h1","ex_1",30,-3,3)
root [] h1.FillRandom("gaus",8000)
root [] h1.Draw()
root [] h1.Draw("e1same")
root [] c1_2.cd()
root [] h1.Draw("lego")
root [] c1_3.cd
root [] TH2F h2("h2","ex_2",20,0,20,20,0,20)
root [] h2.FillRandom("landau",5000)
root [] h2.Draw("lego1pol")
```

```
root [] c1_4.cd()
root [] h2.Draw("surf1sph")
```

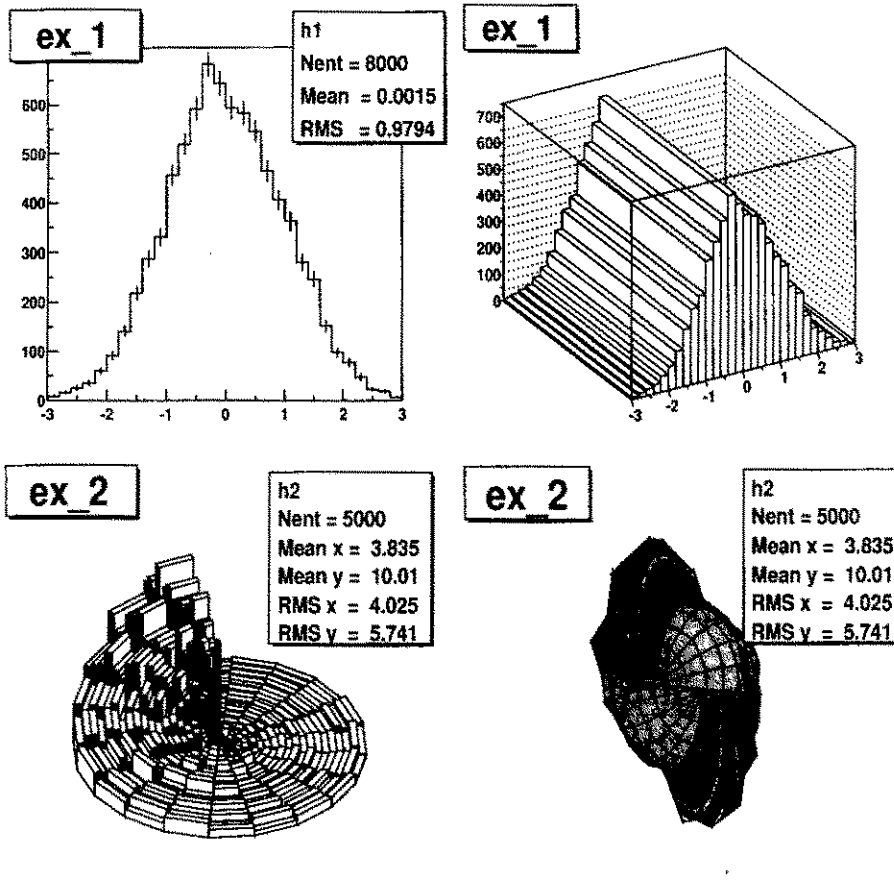

Рис. 6: Одномерные и двумерные гистограммы

7.4.2. Гистограммы с опцией "bar"

Одномерные гистограммы могут быть выведены в виде вертикальных или горизонтальных полос. Для этого в методе `Draw()` используются параметры "bar" "hbar" соответственно. Полосы заполняются цветом, заполняющим гистограмму. Левая (нижняя) сторона полосы выводится светлым заполняющим цветом,

а правая (верхняя) сторона полосы выводится темным заполняющим цветом. Процент полосы, выведенной темным цветом, следующий:

- "bar0", "hbar0" - 0 процентов;
- "bar1", "hbar1" - 10 процентов;
- "bar2", "hbar2" - 20 процентов;
- "bar3", "hbar3" - 30 процентов;
- "bar4", "hbar4" - 40 процентов.

Метод `TH1F::SetBarWidth(Float_t width=0.5)` управляет шириной полосы. Метод `TH1F::SetBarOffset(Float_t offset=0.25)` управляет смещением полосы. Пример гистограммы с опцией "bar" можно найти в `tutorials/hbars.C`.

7.4.3. Как дать заголовки X-, Y- и Z-осям

Поскольку заголовок оси есть атрибут оси, сначала необходимо получить ось, а затем вызвать метод `TAxis::SetTitle`, например

```
h->GetXaxis()->SetTitle("X axis title")
```

Заголовок гистограммы и заголовки оси могут быть любой `TLatex` строкой.

7.4.4. Установка меток на оси

Метод `TPad::SetTicks(Int_t tx=0, Int_t ty=0)` определяет тип меток на осях гистограммы. Значения параметров `tx, ty`:

- `tx=ty=0` - выведены нижняя X-ось и левая Y-ось;
- `tx=1` - выведены деления на внутренней стороне верхней X-оси;
- `tx=2` - выведены цифры на внешней стороне верхней X-оси;
- `ty=1` - выведены деления на внутренней стороне правой Y-оси;
- `ty=2` - выведены цифры на внешей стороне правой Y-оси.

7.4.5. Трехмерные гистограммы

По умолчанию трехмерная гистограмма рисуется в виде рассеянного графика. Если определена опция "box", выводится трехмерный блок с объемом, пропорциональным содержанию ячейки.

7.4.6. Профильные гистограммы

Профильная гистограмма показывает среднее значение величины Y и ее среднеквадратичное отклонение (RMS) для каждого бина величины X. Во многих случаях использовать профильные гистограммы более удобно, чем двумерные. Конструктор:

```
TProfile(const char* name, const char* title, Int_t nbins, Axis_t
```

`xlow, Axis_t xup, Option_t* option)`

Первые пять параметров аналогичны параметрам одномерных гистограмм. Последний параметр `option` задает способ вычисления ошибок. Для значения этого параметра по умолчанию ошибки определяются: `отклонение`/$\sqrt{N}$ - в случае отклонения, отличного от нуля, $\sqrt{Y}/\sqrt{N}$ - в случае, когда отклонение равно нулю, здесь `Y` - значения точек данных, а `N` - число точек данных. Другие значения параметра `option` можно посмотреть на стр.47 в [1]. Чтобы вывести профильную гистограмму без ошибок, используйте опцию "hist" в методе `TProfile::Draw()`. Для заполнения профильных гистограмм используется метод `TProfile::Fill(Axis_t x,Axis_t y)`. В конструкторе объекта `TProfile` можно добавить шестым и седьмым параметрами минимальное и максимальное значения по `Y`-оси. Но при этом необходимо помнить, что при заполнении гистограммы `Fill` метод проверяет, находится ли переменная `y` между `ylow` и `yup`, и все значения ниже минимального и выше максимального будут отвергнуты.

Вместо трехмерных гистограмм часто можно использовать двумерные профильные гистограммы, реализованные классом `TProfile2D`. Конструктор:

```
TProfile2D(const char* name, const char* title, Int_t nbinsx, Axis_t
xlow, Axis_t xup, Int_t nbinsy, Axis_t ylow, Axis_t yup, Option_t*
option)
```

Объект `TProfile` отображает среднее значение величины `Z` и ее RMS для каждой ячейки `[X,Y]`.

7.4.7. Проекции

Можно сделать одномерную проекцию двумерной гистограммы или ее профиль методами `TH2::ProjectionX()`, `TH2::ProjectionY()`, `TH2::ProfileX()`, `TH2::ProfileY()`, `TProfile::ProjectionX()`. Если все параметры этих методов заданы по умолчанию, то имена получающихся гистограмм образуются от имени первоначальной двумерной гистограммы и расширения `_px`, `_py`, `_pfx`, `_pfy`, `_px` соответственно. Методы `TH3::ProjectionZ()`, `TH3::Project3D (Option_t* option=x)` создают одномерные проекции трехмерных гистограмм.

7.5. Статистический дисплей

По умолчанию изображение гистограммы включает блок статистики, так называемый статистический дисплей. Тип отображаемой в нем информации можно выбрать командой

`gStyle->SetOptStat(mode)`

Параметр `mode` состоит из семи цифр, которые могут быть установлены на 1 или на 0, `mode = iourmen` (значение по умолчанию = 0001111),

1. n=1 - напечатано название гистограммы;

2. e=1 - напечатано число точек;
3. m=1 - напечатано среднее значение;
4. r=1 - напечатано значение квадратного корня;
5. u=1 - напечатано число;
6. o=1 - напечатано число переполнений;
7. i=1 - напечатана сумма (интеграл) бинов.

Если первые цифры mode равны 0, то в методе SetOptStat(mode) их можно не записывать. При использовании опции "same" статистический дисплей повторно не выводится. Чтобы при наложении второго изображения гистограммы на первое получить блок статистики второй гистограммы, используйте опцию "sames". И наконец приведем пример кода, отображающего сразу два блока статистики:

```
root [] TPaveStats *st=(TPaveStats*)gPad->GetPrimitive("stats")
root [] st->SetName(new) //new типа Char_t, например new="name2"
root [] st->SetX1NDC(newx1) //newx1 типа Float_t, например newx1=0.2
root [] st->SetX2NDC(newx2) //newx2 типа Float_t, например newx2=0.4
root [] newhist->Draw("sames") //newhist - вторая гистограмма.
```

7.6. Сложение, деление и умножение гистограмм

Для гистограмм поддерживается несколько типов арифметических операций. Метод TH1::Add(TH1* h1, TH1* h2, Double_t c1, Double_t c2) обеспечивает добавление гистограммы к текущей гистограмме, сложение двух гистограмм с коэффициентами и сохранение в текущей гистограмме. Можно также складывать гистограммы с функцией, в этом случае следует вызывать TH1::Add(TF1* f1, Double_t c1). h1, h2, f1 - указатели на гистограммы и функцию, а c1, c2 - коэффициенты. Методы Divide() и Multiply() с аналогичным синтаксисом обеспечивают деление и умножение гистограмм, а также деление и умножение гистограммы на функцию.

Приведем пример построения трехмерной гистограммы (см. рис. 7), создания ее проекций по осям и сложения этих одномерных гистограмм с коэффициентами.

```
root [] TCanvas c1
root [] c1.Divide(2,2)
root [] c1_1.cd()
root [] TH3F h3("h3","ex_3",3,0,6,3,0,6,3,0,6)
root [] h3.FillRandom("pol3",10)
root [] h3.Draw("box")
root [] c1_2.cd()
root [] h3.Project3D("x")
root [] c1_2.cd()
root [] h3_x.Draw()
```

```
root [] c1_3.cd()
root [] h3.Project3D("y")
root [] h3_y.Draw()
root [] c1_4.cd()
root [] TH1F h1("h1","add",3,0,6)
root [] h1.Add(h3_x,h3_y,2,5)
root [] h1.GetXaxis()->SetTitle("Sum X- and Y-projection")
root [] h1.GetXaxis()->CenterTitle()
root [] h1.Draw()
```

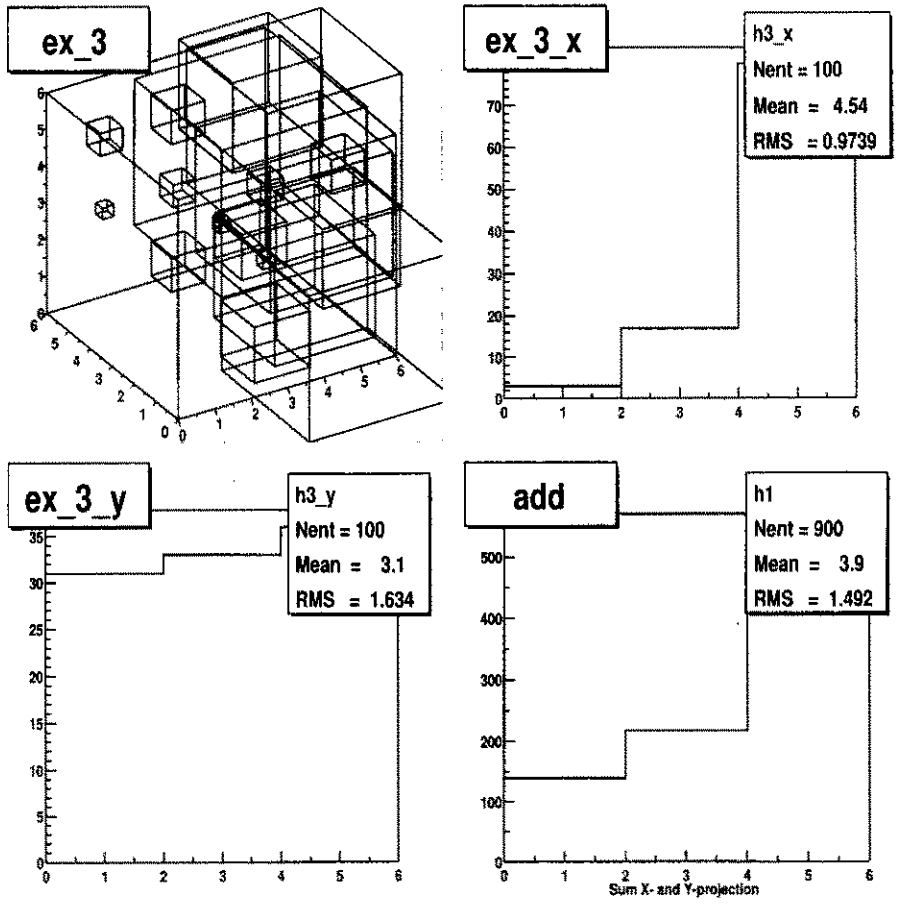

Рис. 7: Трехмерная гистограмма, ее проекции и их сумма

8. Фитирование гистограмм и графов

8.1. Фитирование в интерактивном режиме

Фитирование в интерактивном режиме проводится с помощью диалогового окна, которое называется `Fit` панель. Оно показано на рис. 8.

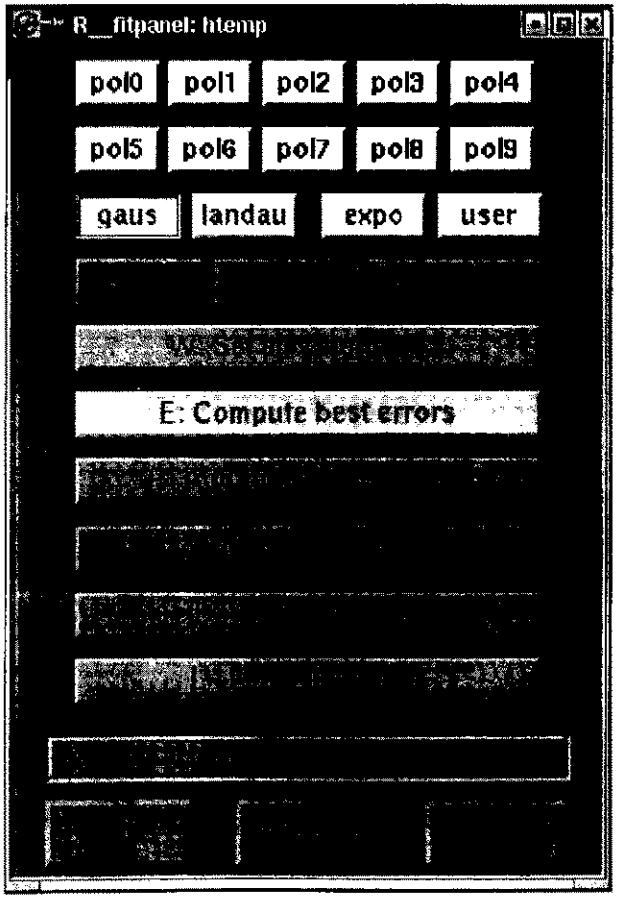

Рис. 8: Fit панель

Гистограмма должна быть выведена на графический экран до начала процесса фитирования, тогда Fit панель становится доступной. Вызовите контекстное меню гистограммы, щелкнув правой кнопкой по верхней границе любого бина. Из меню выберите опцию FitPanel. Первые три ряда кнопок - предопределенные (встроенные) функции ROOT для фитирования гистограмм: распределение Гаусса, экспонента, несколько полиномов и функция Ландау. Можно также самостоятельно определить функцию фитирования и связать ее с кнопкой "user". Можно регулировать режим вывода количества информации, напечатанной на командной строке в результате фитирования, выбирая опции "Quiet" ("Краткий") или "Verbose" ("Подробный"). По умолчанию после выполнения фитирования изображение гистограммы заменяется на изображение функции. Чтобы одновременно увидеть гистограмму и фитировавшую ее функцию, следует нажать кнопку "Same Picture". Чтобы установить все ошибки на 1, выберите опцию "W"; чтобы использовать специальную методику для вычисления лучших ошибок, выберите опцию "E". При фитировании гистограммы несколько раз прежняя функция по умолчанию удаляется и заменяется на самую последнюю. Опция "+ " добавляет новую функцию к списку фитирующих функций. При выборе опции "N" фитирующая функция не добавляется к гистограмме и не выводится, при выборе опции "O" фитирующая функция не рисуется на экране. Опция "L" запускает использование регистрационного метода вероятности. Полоса прокрутки внизу панели позволяет устанавливать диапазон для фитирования путем перетаскивания граней ползунка к центру. Чтобы вернуться к первоначальным установкам, необходимо нажать на Defaults. Чтобы применить фитирование, нажмите кнопку Fit.

8.2. Fit метод

Чтобы фитировать гистограммы в скриптах, используется метод

```
TH1::Fit(const char* fname, Option_t* option, Option_t* goption,
Axis_t xxmin, Axis_t xxman)
```

Первый параметр fname - название фитирующей функции, заранее предопределенной ROOT или определяемой пользователем.

Второй параметр option - опция фитирования. Помимо выведенных на Fit панель и уже рассмотренных значений этой опции, она может принимать следующие значения: "I" - использовать вместо значения в центре бина интеграл функции в бине, "M" - улучшить результаты фитирования, "R" - использовать при фитировании диапазон, указанный в диапазоне конструктора функции, "B" - отключить автоматическое вычисление начальных значений параметров для стандартных предопределенных функций.

Третий параметр goption - графическая опция, ее параметры такие же, как у метода Draw(); xxmin, xxmax - диапазон, в котором применяется фитирование.

8.2.1. Фитирование заранее определенной функцией

В этом случае нужно просто передать имя функции в первом параметре Fit метода, например

```
root [] hist.Fit("gaus")
```

Начальные значения параметров для встроенных функций устанавливаются автоматически.

8.2.2. Фитирование функцией, определенной пользователем

Функции в ROOT реализованы с помощью класса TF1. Для фитирования гистограммы необходимо создать объект TF1 и использовать его при вызове Fit метода. Конструктор:

```
TF1(const char* name, const char* formula, Double_t xmin = 0,
Double_t xmax = 1)
```

Здесь name - имя функции, formula - ее математическое выражение, xmin и xmax - диапазон, в котором задана функция. В математическое выражение могут быть добавлены параметры. Индекс параметра заключается в квадратные скобки. Для явной установки начальных значений параметров используется метод SetParameter(Int_t ipar, Double_t parvalue), где ipar - индекс параметра, parvalue - его начальное значение. Для установки начального значения параметра в 0 существует специальный метод FixParameter(ipar,0). Чтобы установить значения сразу всех параметров, используйте:

```
SetParameters(Double_t p0, Double_t p1, ... , Double_t p10 = 0)
```

Метод SetParLimits(Int_t ipar, Double_t parmin, Double_t parmax) устанавливает границы для одного параметра, ipar - индекс параметра, parmin и parmax - границы его изменения. Если нижний и верхний пределы равны, параметр является фиксированным. Пределы для всех параметров функции устанавливать необязательно.

Еще один способ построить объект TF1 состоит в том, чтобы определить функцию самостоятельно и затем дать ее имя конструктору. Функция для TF1 конструктора должна иметь следующую сигнатуру:

```
Double_t fitf(Double_t *x, Double_t *par)
```

где Double_t *x - указатель на массив измерения, а Double_t *par - указатель на массив параметров. Для одномерной гистограммы используется x[0], для двумерной - x[0], x[1], для трехмерной - x[0], x[1], x[2], ntuple - может иметь до 10 измерений.

Скрипт tutorials/myfit.C иллюстрирует, как фитировать одномерную гистограмму с помощью определенной пользователем функции.

Можно объединять функции, чтобы фитировать гистограмму их суммой. Примеры суммирования функций находятся в tutorials/multifit.C и tutorials/FittingDemo.C.

8.3. Доступ к параметрам функции и результатам

Фитирующая функция добавляется к списку функций, связанных с гистограммой. Метод `TH1::GetFunction(const char* name)` отыскивает функцию, связанную с данной гистограммой, `name` - имя функции. Если указатель на функцию найден, например `*f`, то параметры фитирования для этой функции находятся следующим образом:

```
root [] Double_t chi2=f->GetChisquare()
root [] Double_t p1=f->GetParameter(0)
root [] Double_t e1=f->GetParError(0)
```

Здесь `p1` и `e1` - величина и ошибка первого параметра. `TStyle::SetOptFit(mode)` метод задает, как параметры фитирования отображаются в блоке статистики, `mode = pcev` (значение по умолчанию - 0111), где

- v=1 - печатает названия и величины параметров;
- e=1 - печатает ошибки;
- c=1 - печатает число степеней свободы;
- p=1 - печатает вероятность.

9. Папки

Папка (реализованная с помощью класса `TFolder`) - совокупность объектов, видимых и раскрываемых в браузере ROOT. Папки имеют название и заголовок и идентифицируются в иерархии папок подобно UNIX. Основная папка - `//root` видна слева на верху браузера (см. рис. 1). Под ней показано еще несколько папок. К папке могут добавляться или удаляться из нее новые папки. Папки используются для уменьшения зависимости класса и улучшения модульности создаваемых приложений.

Рассмотрим, как создать иерархию папок. Сначала следует добавить верхнюю папку иерархии к папке `//root`.

```
root [] TFolder *myFold = gROOT->GetRootFolder()->AddFolder("MyFold",
"my top folder")
```

Затем добавить все остальные папки к существующей с помощью метода `TFolder::AddFolder()`, имеющего два параметра: название и заголовок папки.

```
root [] TFolder *data = myFold->AddFolder("data", "experiense data")
root [] TFolder *const = data->AddFolder("const", "Detector const")
root [] TFolder *conf=data->AddFolder("conf", "Configuration")
root [] TFolder *test =myFold->AddFolder("test","test")
root [] TFolder *new =test->AddFolder("new","test in May")
```

```
root [] TFolder *new_24 =new->AddFolder("new_24","test in 24 May")
```

Можно добавить личную папку к списку объектов, доступных программе просмотра, командой:

```
root [] gROOT->GetListOfBrowsables()->Add(myFold,"MyFold")
```

Тогда эта папка становится видна в программе просмотра на верхнем уровне.

Некоторые объекты ROOT автоматически добавляются к иерархии папок. Например, при старте существуют следующие папки: `//root/ROOT Files` - со списком открытых файлов, `//root/Classes` - со списком активных классов, `//root/Styles` со списком графических стилей и другие. Чтобы найти папку или объект в папке, воспользуйтесь методом `TROOT::FindObjectAny()`.

10. Ввод/вывод

10.1. Файл в формате ROOT

В пакете ROOT реализован специальный вид файлов, применяемых для хранения объектов, это так называемые ROOT файлы. Обычно такой файл имеет расширение `root`. ROOT файл может содержать объекты и каталоги, организованные в неограниченное количество уровней, подобно UNIX. Класс `TFile` описывает ROOT файл. Конструктор:

```
TFile (const char* fname, Option_t* option, const char* ftitle, Int_t
compress=1)
```

Параметр `option` может принимать следующие значения:

- NEW или CREATE - создает новый файл и открывает его для записи; если файл уже существует, то файл не откроется;
- RECREATE - создает новый файл; если файл уже существует, то он будет перезаписан;
- UPDATE - открывает существующий файл для записи;
- READ - открывает существующий файл для чтения.

Параметр `compress` определяет уровень сжатия.

10.1.1. Как устроен ROOT файл

Рассмотрим физическое размещение ROOT файла. Первые 64 байта заняты заголовком файла. Первые четыре байта заголовка содержат строку "root", по ней система опознает файл как файл ROOT. Поэтому расширение `.root` для ROOT файла необязательно, а служит лишь подсказкой для пользователя. Далее стоит номер версии файла `fVersion`, указатель на первый рекорд с данными `fBEGIN`, указатель на первое свободное слово `fEND`, указатель на свободные рекорды

fSeekFree, число битов в свободных рекордах fNBytesFree, число свободных рекордов nfree, число битов во время создания fNbytesName, число битов для указателей файла fUnits, уровень сжатия fCompress.

84 байта после заголовка файла содержат описание верхнего каталога, включая название, дату и время, когда он был создан, и дату и время последних модификаций. Далее следуют рекорды с информацией об объектах, записанных в ROOT файл. Эта информация включает в себя длину сжатого объекта, дату и время, когда объект был записан в файл, число битов в имени класса, имя класса объекта, число битов в имени заголовка объекта, заголовок объекта и некоторые другие данные. Просмотреть физическое размещение файла можно методом TFile::Map(), который печатает информацию относительно каждого рекорда при просмотре файла. Это выполнимо потому, что ROOT поддерживает прямой доступ к объектам. Чтобы сделать так, TFile сохраняет список объектов TKey, которые являются по существу индексами к объектам в файле.

Рекорды, занятые информацией об объекте, сопровождаются списком описания класса по имени StreamerInfo. Этот список содержит описание каждого класса, записанного в файл, всех предков класса и всех членов-данных объекта. Метод TFile::ShowStreamerInfo() печатает список описания класса.

10.1.2. Восстановление файла

Предельный размер ROOT файла 2 ГБ. Если эта квота превышена или если есть сбои в работе, то файл может разрушиться, его невозможно будет закрыть правильно и записать на диск. В этом случае необходимо сохранить как можно больше информации. ROOT обеспечивает механизм восстановления файла, использующий информацию каталога в рекордах заголовка. При новом открытии файла, закрытого не должным образом, алгоритм восстановления читает файл и создает сохраненные объекты в памяти согласно информации заголовка. Чтобы при новом закрытии файла сделанные исправления были записаны на диск, файл должен быть открыт в режиме записи. Метод TFile::Recover() явно вызывает процедуру восстановления.

10.1.3. Просмотр содержания файла

Для просмотра содержания файла используется метод TFile::ls(Option_t* option)). Чтобы найти определенный объект в файле, следует использовать метод TFile::Get(). Он находит объект согласно его индексу, описанному классом TKey. TFile имеет методы перечисления содержания, вывода имен на печать и создания подкаталогов. Когда Вы создаете объект TFile, он становится текущим каталогом. Текущий каталог сохраняется в глобальной переменной *gDirectory*. Команда

```
root [] gDirectiry->pwd()
```

покажет ваш текущий каталог.

10.1.4. Объекты в памяти и объекты на диске

Опции метода `TFile::ls()` следующие: "-d" - перечисляет объекты на диске, "-m" - перечисляет объекты в памяти. Если никакая опция не задана, метод перечисляет оба списка. Строка, содержащая объект в памяти, начинается с OBJ. Строка, содержащая объект на диске, начинается с KEY. Синтаксис этой строки следующий:

```
KEY: <class> <variable>;<cycle number> <title>
```

`<class>` - имя класса ROOT, `<variable>` - название объекта, `<cycle number>` - номер цикла, `<title>` - строка, данная в конструкторе объекта как заголовок. Чтобы перенести объект из диска в память, его нужно "получить" явно или использовать его каким-то способом. Когда мы используем объект, ROOT получает его для нас. Любая ссылка на объект, например рисование, будет считывать объект с диска и создавать его в памяти. Объекты в памяти независимы от объектов на диске. Можно изменять объект в памяти, но это не будет распространяться на объект на диске. Новую версию объекта нужно специально спасти на диск.

10.2. Спасение объекта на диск

В памяти может храниться только одна версия объекта, а на диске может сохраняться несколько версий. Для записи объектов на диск используется метод `Write()`. Команда

```
root [] f->Write()
```

записывает в файл весь список объектов, созданных в текущем каталоге. Команда

```
root [] h->Write()
```

записывает в файл только объект с определенным именем, в данном случае с именем `h`. Если объект с этим именем уже существовал на диске, то появится еще один объект с таким же именем и номером версии, увеличенным на единицу. Команда

```
root [] h->Write("NewName")
```

записывает в файл объект с именем `h`, присваивая ему при записи новое имя `NewName`. Команда

```
root [] h->Write(" ",TObject::kOverwrite)
```

записывает в файл объект с именем `h` поверх существующего. Имя объекта и номер версии сохраняются, а предыдущая версия перестает существовать. Команда

```
root [] f->Close()
```

закрывает ROOT файл. После ее выполнения текущий каталог становится пустым, и теперь невозможно ссылаться на объекты ROOT файла. Следует помнить, что метод `TFile::Close()` не вызывает автоматически `TFile::Write()`.

Если ROOT файл содержит многократные версии объекта с одним и тем же названием, то при открытии файла и использовании объекта CINT отыскивает версию с самым большим номером цикла. Чтобы считать предыдущую версию, нужно явно получить ее и приписать ее значение новой переменной. Пусть, например, файл `hist.root` содержит две версии одной и той же одномерной гистограммы `h;1` и `h;2` и мы хотим нарисовать `h;1`. Тогда порядок действий следующий:

```
root [] TFile *f = new TFile("hist.root")
root [] TH1F *his = (TH1F*) f->Get("h;1")
root [] his->Draw()
```

10.3. Подкаталоги и навигация

Класс `TDirectory`, от которого наследует `TFile`, позволяет организовывать свое содержание в подкаталоги. Чтобы добавить подкаталог `name` к ROOT файлу, используется метод `TDirectory::mkdir(const char* name, const char* title)`. Можно изменить текущий каталог, перейдя в подкаталог, методом `TFile::cd (const char* path)`. Чтобы вернуться к верхнему каталогу файлов, используется `cd()` без любых параметров. Чтобы удалить подкаталог, используется метод `TDirectory::Delete(const char *namecycle)`. Строка `namecycle` имеет формат `name; cycle`, где `name` - имя переменной, `cycle` - номер цикла.

Когда гистограмма (или другой объект) создана, она по умолчанию добавляется к списку объектов в текущем каталоге. Можно изменить каталог гистограммы методом `TH1::SetDirectory(TDirectory* dir)`.

Приведем небольшой пример навигации по подкаталогам.

```
root [] TFile *f = new TFile("hist.root","NEW")
root [] f->mkdir("my_dir")
root [] f->cd("my_dir")
root [] TH1F *h = new TH1F("h","title",100,0,10)
root [] f->cd()
root [] f->mkdir("old_dir")
root [] h->SetDirectory(old_dir)
```

В этом примере мы открываем новый файл `hist.root`, затем создаем в нем подкаталог `my_dir`, перемещаемся в него и создаем гистограмму `h`. Далее мы поднимаемся в верхний каталог, создаем в нем еще один подкаталог `old_dir` и перемещаем в него гистограмму `h`.

10.4. Уровень сжатия

ROOT использует алгоритм сжатия, основанный на известном gzip алгоритме. Он поддерживает девять уровней сжатия. Значение по умолчанию - 1, если значение равно нулю, то сжатие не производится. Уровень сжатия устанавливается либо конструктором `TFile`, либо методом `TFile::SetCompressionLevel(Int_t level)` и может быть изменен в любое время. Новый уровень сжатия применяется только к записи объектов, создаваемых после его установки. Следовательно, ROOT файл может содержать объекты с различными уровнями сжатия. Чем выше уровень сжатия, тем больше экономия дискового пространства, но тем больше время, затраченное на чтение и запись объектов. Поэтому при выборе уровня сжатия следует обращать внимание на тип данных. Если данные редки, то есть имеется много нулей, или есть очень много однотипных данных, разумно установить более высокое значение уровня сжатия. Если же время, затраченное на ввод-вывод, большое по сравнению со временем обработки данных, то слишком высокий уровень сжатия будет снижать эффективность программы.

11. Деревья

11.1. Объекты классов `TNtuple` и `TTree`

Для хранения большого количества объектов одного и того же класса в ROOT разработаны специальные классы `TTree` и `TNtuple`. Понятие `Ntuple` появилось еще при разработке пакетов HBOOK и PAW. В `Ntuple` записывается каждая переменная каждого события, а затем при анализе данных эти переменные можно обрабатывать по любым алгоритмам, используя любые критерии отбора. Объект класса `TNtuple` можно считать деревом, ограниченным числами с плавающей запятой. Объект класса `TTree` может содержать все виды данных, объекты и массивы в дополнение ко всем простым типам.

При использовании дерева каждый объект не записывается отдельно в файл, а собирается в связку, и эта связка объектов записывается одновременно. При записи объекта `TTree` в ROOT файл, поскольку единица сжатия есть буфер, размер файла будет намного меньше, чем если бы объекты, составляющие дерево, были записаны индивидуально.

Преимущество использования объектов класса `TTree` проявляется также при организации доступа к данным, объединенным в совокупности, называемые ветвями и листьями. Дерево использует иерархию ветвей, и каждая ветвь может читаться независимо от другой ветви. Поэтому при обработке данных нет необходимости считывать каждый раз событие полностью, что особенно существенно в современных физических экспериментах с большой статистикой и большим количеством измеряемых величин в каждом событии. Конструктор дерева:

```
TTree(const char* name, const char* title).
```

Можно также создать дерево из иерархии папок с ветвями для каждой подпапки. Например, если "MyFolder" - верхняя папка в иерархии, то команда

```
root [] TTree folder_tree("MyFolderTree","/MyFolder")
```

создаст дерево, а команда

```
root [] folder_tree.Fill()
```

заполнит его данными, содержащимися в подпапках. Обратите внимание на наклонную черту перед вторым параметром в конструкторе. Она сообщает ROOT, что это есть папка, а не просто заголовок.

11.2. Ветви

Для создания ветви дерева используется метод `TTree::Branch()`, который имеет несколько видов записи в зависимости от вида ветви. Если две переменные независимы и на этапе проектирования уже известно, что переменные не будут использоваться вместе, то имеет смысл разместить их в отдельные ветви. Если же переменные связаны, например, три координаты одной точки, то более эффективно создать одну ветку с несколькими переменными на ней, организованными в виде листьев (объектов класса `TLeaf`). Вид ветки зависит от того, что сохранено в ней. Ветка может содержать список простых переменных, полный объект, папки с объектами `TLeaf` или массив объектов. Рассмотрим несколько примеров.

11.2.1. Добавление ветвей, содержащих список переменных

В этом случае синтаксис метода имеет следующий вид:

```
TBranch(const char* name, void* address, const char* leaflist)
```

Здесь `name` - имя ветви, `address` - адрес, с которого должен считываться первый лист, `leaflist` - список листьев, составляющих ветку, с именем и типом каждого листа.

Для того, чтобы на одной ветке было несколько листьев, каждый из которых представляет собой одну переменную, эти переменные должны быть объединены в структуру. Пусть, например, `event` - структура, содержащая `n` - переменную целого типа и `x, y, z` - переменные с плавающей точкой. Тогда для создания ветви с именем `NameBr` у дерева `tree` используется следующая команда:

```
tree->Branch("NameBr",&event, "n/I:x:y:z/F")
```

Здесь второй параметр - адрес, откуда начинается считывание, третий параметр - строка со списком листьев. Название листа не используется для выбора переменной из структуры, а используется только как имя для листа. Следовательно, список переменных в третьем параметре должен иметь такой же порядок, в каком переменные описаны в структуре. Если структура не определена, то для

каждой переменной должна быть создана своя ветка.

Листья в третьем параметре отделены друг от друга двоеточием. Каждый лист имеет название и тип, отделенные друг от друга наклонной чертой.

```
<Variable>/<type>:<Variable>/<type>
```

Тип может быть опущен, в этом случае переменной листа присваивается значение типа такое же, как у переменной предыдущего листа. Если тип опущен у первого листа, то его переменной присваивается тип `Float_t`. Символы, используемые для типа в синтаксисе метода `Branch()`:

1. `C` - символьная строка, законченная символом 0,
2. `B` - 8–разрядное целое число со знаком,
3. `b` - 8-разрядное целое число без знака,
4. `S` - 16–разрядное целое число со знаком,
5. `s` - 16-разрядное целое число без знака,
6. `I` - 32-разрядное целое число со знаком,
7. `i` - 32-разрядное целое число без знака,
8. `F` - 32-разрядное число с плавающей точкой,
9. `D` - 64–разрядное число с плавающей точкой.

Тип используется ROOT, чтобы решить, сколько места выделить под переменную. Можно также добавлять листы, содержащие массивы переменных постоянной и переменной длины, например,в коде ниже `f` - массив постоянной длины, а `E` - массив переменной длины:

```
{
Float_t f[10];
Int_t NP;
Float_t T[NP];
tree->Branch("br1",f,"f[10]/F");
tree->Branch("br2",&NP,"NP/I");
tree->Branch("br2",E,"E[NP]/F");
}
```

Обратите внимание на второй параметр метода `Branch` в случае листа с массивом. Поскольку в C++ переменная массива содержит адрес первого элемента, определять `&f` и `&E` не требуется.

11.2.2. Добавление ветвей, содержащих объект

При добавлении ветви, содержащей объект, метод записывается следующим образом:

```
TBranch(const char* name, const char* classname, void* addobj, Int_t
bufsize = 32000, Int_t splitlevel = 99)
```

В этом случае ROOT должен знать определение класса объекта, а следовательно, должна быть загружена библиотека, содержащая определение нужного класса. Для того, чтобы объект был в дереве со своим определением класса, в заголовочный и исполняемый файлы класса необходимо включить макрокоманды `ClassDef` и `ClassImp`. Об этих макрокомандáх более подробно будет рассказано в разделе "Добавление класса".

Допустим, мы хотим создать ветку, содержащую объект `Event` созданного нами класса `Event`. Сначала должна быть создана и загружена библиотека `libEvent.so`. Далее, как и раньше, откроем файл и создадим дерево:

```
root [] TFile *f = new TFile("AFile.root","recreate")
root [] TTree *tree = new TTree("T","A Root Tree")
```

Далее создадим указатель на объект `Event`, который будет использоваться как ссылка на `Tree::Branch()` метод. Затем создаем ветвь:

```
root [] Event *event = new Event()
root [] tree->Branch("EventBranch","Event",&event,32000,99)
```

Первый параметр метода - имя ветки, второй - имя класса объекта, который будет сохранен. Третий параметр - адрес указателя на сохраняемый объект, четвертый - размер буфера, по умолчанию это 32000 байтов. Пятый параметр - уровень разбиения.

11.2.3. Уровень разбиения

При установке уровня разбиения в ноль целый объект записывается полностью на одну ветку, то есть разбиение отключено. При установке уровня разбиения от 1 до 99 число указывает на глубину разбиения. Когда уровень разбиения - 1, члены-данные объекта приписываются ветке. Когда уровень разбиения - 2, члены-данные объекта также будут разбиты. Когда уровень разбиения - 3, объекты членов-данных объекта будут разбиты. Значение по умолчанию для уровня разбиения - 99, это означает, что объект будет разбит по максимуму.

Расщепление ветки может быстро сгенерировать много ветвей. Каждая ветка имеет свой собственный буфер в памяти, поэтому при наличии многих ветвей (>50) необходимо скорректировать размер буфера так, чтобы не произошло превышение памяти.

Разбиение ветки ускоряет чтение, потому что переменные одного и того же типа сохранены последовательно и тип не должен читаться каждый раз. Но разбиение ветки слегка замедляет запись из-за большого количества буферов. Исходя из этого были выработаны правила расщепления веток, при которых достигается наибольшая эффективность работы. Рекомендуем вам придерживаться этих правил.

1. Члены-данные основного типа не разбиваются, становясь единой веткой.
2. Члены-данные - массивы основных типов не разбиваются.

3. Указатели на члены-данные не разбиваются за исключением указателей на `ClonesArray`.
4. Члены-данные - объекты разбиваются на ветви согласно количеству членов-данных объекта.
5. Если члены-данные - указатели на объект, то создается специальная ветка.
6. Основные классы разбиваются, если объект разбивается.

Объект, который не разбивается, требует много времени при просмотре.

11.2.4. Идентичные имена веток

Когда объект верхнего уровня имеет два члена данных одного и того же класса, бывает, что подветви заканчиваются идентичными названиями. Чтобы различить подветви, следует дать им вместо простых названий, например `sub`, сложные со включением в название имени главной ветки, например `master.sub`. Это достигается путем добавления точки в конце имени главной ветки в методе `Branch()`.

Например, пусть дерево имеет две ветки `Trigger` `MuonTrigger`, и каждая из веток имеет листья `T1`, `T2` и `T3`. Тогда команды

```
tree->Branch("Trigger.","Trigger",&b1,8000,1)
tree->Branch("MuonTrigger.","Trigger",&b2,8000,1)
```

сгенерируют объекты `Trigger.T1`, `Trigger.T2`, `Trigger.T3`, `MuonTrigger.T1`, `MuonTrigger.T2`, `MuonTrigger.T3`.

11.2.5. Добавление ветки с папкой

```
tree->Branch("/aFolder")
```

Этот метод создает одну ветку для каждого элемента в папке.

11.3. Некоторые полезные методы класса TTree

Метод `Show(Int_t entry)` показывает одно событие дерева с номером `entry`. Метод `Print()` показывает структуру дерева, то есть число событий, количество и имена ветвей и листьев, размер дерева и уровень сжатия данных. Метод `Scan(const char* name_select)` показывает все значения списка выбранных листьев, имена листьев отделены друг от друга двоеточием.

11.4. Средство просмотра дерева - TreeViewer

Существует простой и эффективный способ исследовать дерево - вызвать `TTreeViewer` - средство просмотра дерева. Для этого необходимо открыть ROOT файл, содержащий объект `TTree`, и запустить программу просмотра объектов, затем из контекстного меню `TTree` выбрать метод `StartViewer`. Можно также

вызвать средство просмотра дерева с командной строки.

```
root [] TFile f("name.root")
root [] tree->StartViewer()
```

Средство просмотра для дерева со списком простых переменных показано на рис. 9.

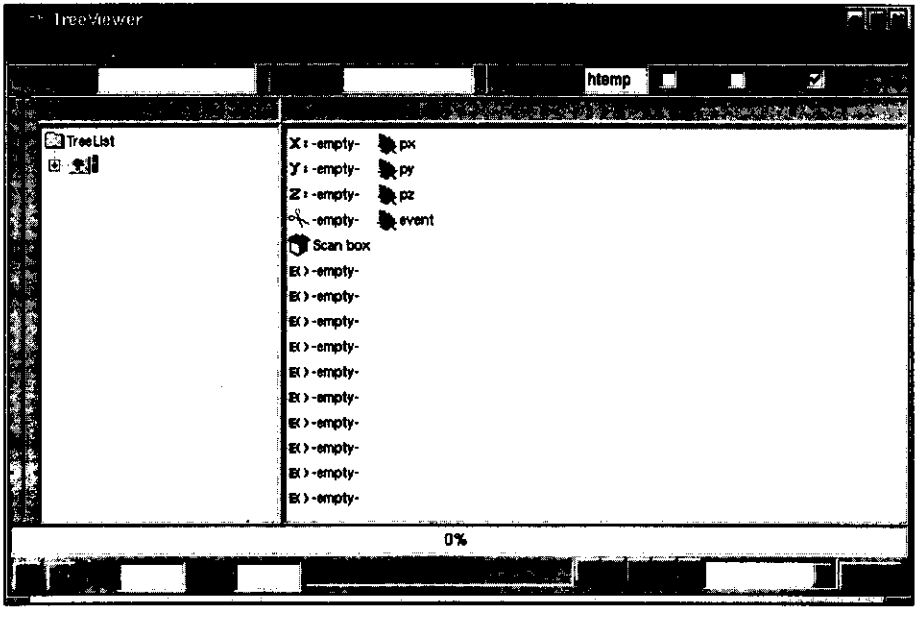

Рис. 9: Средство просмотра дерева

Левая панель содержит список деревьев и их ветвей. В данном примере имеется только одно дерево. Можно добавить другие деревья, выбрав в строке меню **TreeViewer** пункт **File** и из выпадающего меню выбрать пункт **Open tree file**. В открывшемся диалоговом окне выбрать имя необходимого файла и открыть его. Затем активизировать контекстное меню **TreeViewer** щелчком правой кнопкой мыши по его правой панели и выбрать метод **SetTreeName**. (Необходимая подсказка, кстати, появляется в командной строке.) Остается только ввести в открывшееся диалоговое окно имя дерева, и тогда оно появится на левой панели.

Правая панель **TreeViewer** содержит список листьев или переменных дерева. Чтобы получить гистограмму листа, следует дважды щелкнуть левой кнопкой

мыши по этому листу. Чтобы вывести двух- или трехмерную гистограмму, перетащите любой лист к X:, Y: и Z: координатам и нажмите кнопку Draw в нижней строке TreeViewer, окрашенную фиолетовым цветом.

"Блок вырезки", отмеченный значком с ножницами, служит для добавления какого-либо условия на гистограмму. Выберите пункт EditExpression из контекстного меню блока вырезки, в открывшееся диалоговое окно введите условие вырезки и присвойте ему название или номер.

Можно создать новое выражение из переменных листьев. Для этого щелкните дважды левой кнопкой мыши по любой из E() рамок, в открывшееся диалоговое окно введите выражение и присвойте ему название или номер. Это диалоговое окно открывается также при выборе пункта EditExpression из контекстного меню E() рамки. Создаваемые выражения можно помещать в любую из рамок X:, Y:, Z:, Cat или Scan.

Можно просматривать одну или большее количество переменных, переместив их в блок Scan и дважды щелкнув левой кнопкой мыши по этому блоку. Если в следующей за основным меню строке отмечен блок Rec, то команды Draw и Scan регистрируются в файле хронологии и отражаются в командной строке.

Текстовое поле "Histogram" содержит имя построенной гистограммы, по умолчанию htemp.

Пункт "Option" основного меню содержит список Draw параметров для гистограмм. При вызове эти параметры отражаются в текстовом поле "Option" строки, следующей за строкой основного меню. Можно также вводить их в это текстовое поле непосредственно.

Текстовое поле "Commands" позволяет вводить любую команду, которую можно ввести в командную строку ROOT.

Кнопка "RESET" сбрасывает содержание X:, Y:, Z:, E() рамок, блока "Scan" и блока вырезки.

11.5. Создание, заполнение и запись дерева

Приведем пример последовательности действий при создании дерева с простыми переменными:

```
root [0] TFile f("tree.root","recreate")
root [1] TTree t("t","simple tree")
root [2] Float_t px,py,pz
root [3] Int_t event
root [4] t.Branch("px",&px,"px/F")
root [5] t.Branch("py",&py,"py/F")
root [6] t.Branch("pz",&pz,"pz/F")
root [7] t.Branch("event",&event,"event/I")
root [8] { for (Int_t i=0;i<1000;i++){ gRandom->Rannor(px,py);
pz=px*px+py*py;
event=i;
```

```
t.Fill();
}
}
root[9] t.Write()
```

Комментарии для этого примера следующие:

1. Создание ROOT файла [0].
2. Создание дерева [1].
3. Объявление переменных веток (листьев) [2-3].
4. Организация дерева в ветви. Присвоение веткам имен, указание адреса, по которому каждая ветка получает свои значения, объявление имен и типов листьев [4-7].
5. Присвоение переменным веток их значений [8].
6. Заполнение дерева (метод `TTree::Fill()`). Поскольку дерево уже организовано в ветви и каждая ветка знает, где получить значение, вызов `Fill()` заполняет все ветви в дереве [8].
7. Запись дерева в файл [9].

После выполнения этих команд мы будем иметь ROOT файл, называемый `tree.root`, с деревом, называемым `t`.

11.6. Чтение дерева

Рассмотрим порядок действий при чтении дерева, то есть как обратиться к каждому входу и каждому листу дерева.

```
root [0] TFile *f=new TFile("tree.root")
root [1] TTree *t=(TTree*)f->Get("t")
root [2] Float_t px,py,pz
root [3] Int_t event
root [4] t->SetBranchAddress("px",&px)
root [5] t->SetBranchAddress("py",&py)
root [6] t->SetBranchAddress("pz",&pz)
root [7] t->SetBranchAddress("event",&event)
root [8] TH2F *hpxpz=new TH2F("hpxpz","title",30,-3,3,20,0,10)
root [9] Int_t nentries=(Int_t)t->GetEntries()
root [10] {
for (Int_t i=0;i<nentries;i++){
t->GetEntry(i);
hpxpz->Fill(px,pz);
}
}
```

Мы проделали следующие действия:

1. Открытие файла и отыскание в нем дерева [0-1].
2. Объявление переменных, чьи значения необходимо прочитать [2-3].
3. Указание адреса, по которому должно проходить заполнение переменных значениями листьев при чтении дерева. Для этого используется метод `TTree::SetBranchAddress(const char* bname, void* add)` [4-7]. Первый параметр - название ветки, второй параметр - адрес переменной, куда данные ветки должны быть помещены.
4. Создание гистограмм, в которые будут помещаться считываемые данные [8].
5. Считывание количества входов, содержащихся в дереве (необязательное действие) [9].
6. Считывание переменных с определенного входа и заполнение данного адреса методом `TTree::GetEntry(n)`, `n` - номер входа [10].
7. Заполнение гистограмм методом `Fill()` [10].

Чтение отдельных ветвей более быстрое, чем чтение полного входа. Если необходимо считать данные только с одной ветки, можно использовать метод `TBranch::GetEntry(n)`. В этом случае сначала находим эту ветку, устанавливаем ее адрес и считываем ее данные. Например:

```
root [] TBranch *x=t->GetBranch("px")
root [] x->SetAddress(&px)
. . .
root [] x->GetEntry(i)
```

Примеры записи, заполнения и чтения различных деревьев можно найти в скриптах `tree1.C`, `tree2.C`, `tree3.C`, `tree4.C` в каталоге `tutorials`.

11.7. Добавление ветки к уже существующему дереву

Может появиться необходимость добавить ветку к существующему дереву. Например, если одна переменная была вычислена по одному алгоритму, а Вы хотите попробовать другой алгоритм и сравнить результаты. Это можно сделать двумя способами. Пример ниже показывает, как добавить новую ветку, заполнить ее и заново сохранить дерево.

```
root [0] TFile f("tree.root","update")
root [1] Float_t nb
root [2] TTree *t=(TTree*)f->Get("t")
root [3] TBranch *newBr=t->Branch("nb",&nb,"nb/F")
root [4] Int_t nentries=(Int_t)t->GetEntries()
root [5] {
for(Int_t i=0;i<nentries;i++){
nb=gRandom->Gaus(0,1);
newBr->Fill();
```

```
}
}
root [6] t->Write(,TObject::kOverwrite)
```

Обратите внимание на опцию `kOverwrite` в методе `Write`, она записывает новое дерево поверх существующего. Если эта опция не задана, то в файле сохраняются две копии дерева.

Добавление новой ветки часто невозможно. Дерево может находиться в файле, открытом только для чтения, и тогда невозможно сохранить измененное дерево с новой веткой. К тому же файлы с деревьями имеют обычно большой размер, при добавление ветки он может превысить допустимые 2 ГБ. И тогда, даже если пользователь обладает полными правами доступа, при неудачной попытке сохранения модификации можно потерять первоначальное дерево. В любом случае добавление ветки к дереву увеличивает дерево и, следовательно, увеличивается объем памяти, необходимый для считывания входа, и поэтому уменьшается эффективность. По этим причинам в ROOT разработана концепция "друзей" для деревьев и метод `TTree::AddFriend()`.

11.8. Дружественные деревья

В контексте ROOT дружба означает неограниченный доступ к данным друзей. Концепция дружественного дерева позволяет добавлять ветки к дереву без риска повреждения его с помощью метода

```
TTree::AddFriend(const char* treename, const char* filename)
```

Первый параметр - имя дерева, второй - имя ROOT файла, в котором сохранится дружественное дерево. `AddFriend()` автоматически открывает дружественный файл. Если никакое имя файла не задано, то новое дерево сохраняется в том же файле, что и первоначальное. Если дружественное дерево имеет одно и то же имя с первоначальным, то можно давать ему псевдоним (алиас) в контексте дружбы.

```
tree.AddFriend("tree1=tree","friendfile1.root")
```

Как только дерево приобрело друзей, можно использовать `TTree::Draw()` так, как будто переменные друга имеются в первоначальном дереве. Чтобы определить, какое дерево использовать в `Draw()` методе, для первого параметра этого метода применяется синтаксис:

```
<treeName>.<branchName>.<variableName>
```

Если `<variableName>` достаточно, чтобы уникально идентифицировать переменную, можно не учитывать имена дерева и/или ветки.

Число входов в дружественном дереве должно быть равно или больше числа входов первоначального дерева. Если дружественное дерево имеет меньшее количество входов, дается предупреждение и отсутствующие входы не включаются в гистограмму.

Когда дерево записано в файл методом TTree::Write(), список друзей сохраняется с ним. И когда дерево восстанавливается, деревья из списка друзей также восстанавливаются, и восстанавливается дружба. Когда дерево удаляется, элементы дружественного списка удаляются также. Чтобы отыскать список друзей дерева, используется метод TTree::GetListOfFriends().

Возможно так объявить дружественное дерево, чтобы оно имело ту же самую внутреннюю структуру, что и первоначальное дерево, и сравнивать одни и те же значения переменных.

11.9. Деревья в анализе

Методы TTree::Draw(), TTree::MakeClass(), TTree::Make Selector() применяются для анализа данных с использованием деревьев. TTree::Draw() метод является простым способом посмотреть и вывести содержание деревьев. Он дает возможность получить график переменной (листа) только одной линией кода. Однако этот метод не подходит для просмотра каждого входа и проектирования более сложных критериев для анализа данных. Для этих случаев используется метод TTree::MakeClass(). Он создает класс, который прокручивает входы деревьев один за другим. Метод TTree::Make Selector() рекомендуется для анализа большого набора данных в конфигурации с параллельной обработкой, где анализ распределен по нескольким процессорам и пользователь может определять, какие входы посылать каждому процессору.

11.9.1. Простой анализ, использующий TTree::Draw

Рассмотрим возможности метода

```
TTree::Draw(const char* varexp, const char* selection, Option_t*
option, Int_t nentries = 1000000000, Int_t firstentry = 0)
```

Пусть MyTree есть указатель на исследуемое дерево. Команда с одним параметром, именем листа,

```
MyTree->Draw("nameLeaf_1")
```

строит гистограмму этого листа. Стиль гистограммы наследуется от атрибутов TTree, а текущий стиль (gStyle) игнорируется. Дерево получает свои атрибуты из потока TStyle в то время, когда оно создается. Можно вызвать метод TTree::UseCurrentStyle(), чтобы изменить текущий стиль. Для построения двумерной гистограммы используется команда

```
MyTree->Draw("nameLeaf_1:nameLeaf_2")
```

Ее синтаксис имеет все еще только один параметр, но он теперь имеет два измерения, отделенных двоеточием. При записи этого параметра могут использоваться не только простые переменные, но и выражения, составленные из nameLeaf.

Вообще, первый параметр метода TTree::Draw() - строка, содержащая до трех выражений, по одному для каждого измерения, отделенных двоеточием.

Второй параметр метода TTree::Draw() есть параметр выбора, в качестве него можно использовать выражение, содержащее названия листьев и других переменных, любой C++ оператор, некоторые функции, определенные в TFormula, а также объект TCut. Например

```
root [] MyTree->Draw("nameLeaf_1:nameLeaf_2", "nameLeaf_3>0")
```

Или

```
root [] TCut c1="nameLeaf_3>0")
root [] MyTree->Draw("nameLeaf_1:nameLeaf_2",c1)
root [] MyTree->Draw("nameLeaf_1:nameLeaf_2",c1 || "nameLeaf_4<0")
```

Метод TTree::Draw() создает гистограмму, называемую по умолчанию htemp, и помещает ее в текущий каталог и активный TPad, если открыт графический режим. Команда

```
root [] MyTree->Draw("nameLeaf_1>h1")
```

создает гистограмму с именем h1 и заполняет ее переменной nameLeaf_1. Если необходим определенный диапазон гистограммирования и размер бина, то можно заранее создать гистограмму с нужными параметрами конструктором объектов TH. Тогда метод Tree::Draw() только заполнит ее нужной переменной.

Следующий параметр - опция рисования для гистограмм. В дополнение к опциям, уже рассмотренным в разделе 5.7, имеются еще три. "prof" и "profs" (разница между ними в способе представления ошибок) выводят профильную гистограмму от выражения с двумя переменными и гистограмму класса TProfile2D от выражения с тремя переменными. Опция "goff" подавляет производство графики.

Четвертый параметр TTree::Draw() - число рассматриваемых входов, и пятый - номер входа, с которого начинается вывод.

11.9.2. Список событий

Когда первому параметру метода TTree::Draw() предшествует оператор ">>", ROOT знает, что эта команда предназначена не для рисования, а для создания списка событий с названием, данным первым параметром. Это так называемый объект TEventList, в котором сохраняются входы, удовлетворяющие условию выбора, заданному во втором параметре метода TTree::Draw(). TEventList используется, чтобы ограничить дерево событиями в списке. После создания он добавляется в текущий каталог. Метод TTree::SetEventList(TEventList* list) сообщает дереву, что при дальнейших вызовах любых методов для этого дерева необходимо использовать только входы, содержащиеся в списке событий с указателем list. Приведем пример кода, создающий для дерева t список событий,

отвечающий условию, наложенному на переменную x.

```
root [] t->Draw(">myList", "x>100")
root [] TEventList *list = (TEventList*)gDirectory->Get("myList")
root [] t->SetEventList(list)
root [] t-> .....
```

11.10. Цепочки

Когда приходится иметь дело с очень большим объемом информации, удобно использовать объект класса `TChain`. Конструктор

```
TChain(const char* name, const char* title)
```

Цепочка - это несколько ROOT файлов с одним и тем же деревом. ROOT файлы объединяются в цепочку с помощью метода

```
TChain::Add(const char* name, Int_t nentries = -1)
```

Класс `TChain` получен из класса `TTree`, следовательно, все его методы доступны для объектов `TChain`.

12. Добавление класса

Иногда возникает необходимость создать свой собственный класс. Для правильной интеграции собственных классов пользователя в ROOT и обеспечения использования таких удобных инструментов этого пакета, как идентификация типа во время выполнения (RTTI) и ROOT ввод-вывод объектов, необходимо связать собственные классы со словарем, сгенерированным CINT. Это делается с помощью макрокоманд `ClassDef` и `ClassImp`. В заголовочном файле класса следует добавить строку

```
ClassDef (ClassName,ClassVersionID) //заголовок класса
```

`ClassName`-имя класса, `ClassVersionID`-номер версии, который используется системой ввода-вывода ROOT. Каждый раз, когда пользователь изменяет члены-данные класса, он должен увеличить его `ClassVersionID` на единицу. Если нет необходимости в операции ввода/вывода для класса, установите `ClassVersionID=0`.

В исполняемом файле класса следует добавить строку

```
ClassImp (ClassName)
```

`ClassDef` и `ClassImp` макрокоманды определены в файле **Rtypes.h**. Этот файл упомянут во всех ROOT `include` файлах, и пользователь автоматически получает эти макрокоманды, если использует ROOT `include` каталог.

Для возможности использования ROOT ввода/вывода в собственном классе в нем необходимо обеспечить заданный по умолчанию конструктор, то есть

конструктор с нулевыми параметрами или с параметрами со всеми значениями по умолчанию. Этот заданный по умолчанию конструктор вызывается всякий раз, когда объект читается из ROOT базы данных.

Далее необходимо с помощью `rootcint` создать файл словаря:

```
root [] .!rootcint NameDict.cxx -c TNameYourClass.h LinkDef.h
```

Здесь `NameDict.cxx` - имя словаря, который будет сгенерирован. Это имя должно быть уникальным. `TNameYourClass.h` - имя собственного класса пользователя. `LinkDef.h` - специальный файл, сообщающий `rootcint`, для каких классов должен быть создан интерфейс.
Чтобы получить справку по `rootcint`, напечатайте

```
root [] .!rootcint -?
```

Следующий шаг - с помощью компилятора C++ получите объектные файлы вашего класса и его словаря.

```
.!c++ -c TNameYourClass.cxx NameDict.cxx $(root-config -cflags)
```

Затем должна быть построена динамическая библиотека `LibName.so` , содержащая ваш класс и словарь класса для CINT.

```
.!c++ -shared -O TNameYourClass.o NameDict.o -o LibName.so
```

Для компиляции класса можно использовать утилиту `make` [5]. В каталоге `test` приведен пример `Makefile`, в котором компилируются классы, определенные в этом каталоге. Компиляция в этом случае производится командой

```
gmake -f Makefile
```

Например, определим небольшой класс `SClass`. Пусть файл заголовка этого класса `SClass.h` имеет вид

```
#include <iostream.h>
#include "TObject.h"
class SClass : public TObject{
private:
Float_t fX;
Float_t fY;
Int_t fTemp;
public:
SClass() {fX=fY=-1 ;}
void Print() const;
void SetX(float x) {fX=x;}
void SetY(float y) {fY=y;}

ClassDef (SClass,1)
};
```

Исполняемый файл класса SClass.cxx следующий:

```
#include "SClass.h"
ClassImp (SClass);
void SClass::Print() const {
cout<<"fX="<<fX<<", "fY="<<fY<<endl;
}
```

Обратите внимание на макрокоманды в этих файлах и на конструктор класса, заданный по умолчанию, в заголовочном файле.

И файл LinkDef.h в этом случае будет иметь вид

```
{
#ifdef __CINT__

#pragma link off all globals;
#pragma link off all classes;
#pragma link off all functions;

#pragma link C++ class SClass;
#endif
}
```

Далее в командной строке наберите следующие команды:

```
root [] .!rootcint SClassDict.cxx -c SClass.h LinkDef.h
root [] .!c++ -c SClass.cxx SClassDict.cxx $(root-config -cflags)
root [] .!c++ -shared -O SClass.o SClassDict.o -o LibSClass.so
```

После создания динамической библиотеки загрузите ее и, создав объект вашего собственного класса, начинайте с ним работать.

```
root [] .L libSClass.so
root [] SClass *sc = new SClass()
root [] TFile *f = new TFile("name.root","update")
root [] sc->Write()
```

13. Физические векторы

13.1. Классы физических векторов

Для проведения расчетов в области физики высоких энергий в ROOT существует несколько специальных классов, предназначенных для описания физических векторов в трех и четырех измерениях и алгоритмов их преобразований. Эти классы были перенесены в ROOT из CLHEP. Они не загружаются по умолчанию, и чтобы получить к ним доступ, необходимо загрузить библиотеку libPhysics.so. Мы рассмотрим следующие классы:

- `TVector3` - общий класс трехмерных векторов,
- `TLorentzVector` - общий класс четырехмерных векторов,
- `TRotation` - класс, описывающий вращение объекта `TVector3`,
- `TLorentzRotation` - класс, описывающий преобразования Лоренца,
- `TVector2` - класс векторов, определенных на плоскости.

13.2. Класс `TVector3`

13.2.1. Декларация и доступ к компонентам

`TVector3` - класс, описывающий различные векторы в трехмерном пространстве. Трехмерный вектор может быть задан в декартовых, сферических или цилиндрических координатах. Методы класса вычисляют модуль, скалярное и векторное произведение векторов, углы между векторами, вращения и увеличения, а также специфические величины, используемые в физике высоких энергий, такие как псевдоскорость, поперечная компонента, инвариантная масса и др. Конструктор:

```
TVector3(Double_t x = 0.0, Double_t y = 0.0, Double_t z = 0.0)
```

Компоненты вектора можно получить по имени или по индексу, например, если `v` - объект класса `TVector3`, то

```
x=v.X(); x=v(0);
y=v.Y(); y=v(1);
z=v.Z(); z=v(2);
```

Методы `SetX()`, `SetY()`, `SetZ()` и `SetXYZ()` позволяют устанавливать компоненты вектора.

13.2.2. Переход в недекартовы координаты

Чтобы получить координаты вектора $v(x, y, z)$ в других системах, сферической $v(\rho, \phi, \Theta)$ и цилиндрической $v(\tilde{\rho}, \phi, z)$, используются методы

- `Mag()` - вычисляет $\rho = \sqrt{x*x+y*y+z*z}$;
- `Mag2()` - вычисляет квадрат величины ρ;
- `Theta()` - вычисляет полярный угол;
- `CosTheta()` - вычисляет косинус Θ;
- `Phi()` - вычисляет азимутальный угол;
- `Perp()` - вычисляет поперечную компоненту $\tilde{\rho} = \sqrt{x*x+y*y}$;
- `Perp2()` - вычисляет квадрат поперечной компоненты.

Поперечную компоненту вектора `v1` относительно вектора `v2` и ее квадрат можно получить, используя `v1.Perp(v2)` и `v1.Perp2(v2)`. Методы `Eta()` или `PseudoRapidity()` вычисляют псевдоскорость $\eta = -\ln(\tan\phi/2)$. Методы

SetTheta(), SetPhi(), SetMag() и SetPerp() изменяют одну (соответствующую названию) из недекартовых координат при сохранении прежними двух других.

13.2.3. Арифметические операции и вращение

TVector3 класс имеет операторы, чтобы складывать, вычитать, масштабировать и сравнивать векторы. Для этого используются соответствующие операторы C++. Также можно получить следующие величины:

- v2=v1.Unit() - единичный вектор, параллельный v1;
- v2=v1.Orthogonal() - единичный вектор, ортогональный v1;
- s=v1.Dot(v2) - скалярное произведение векторов v1 и v2;
- v=v1.Cross(v2) - векторное произведение векторов v1 и v2;
- Double_t a=v1.Angle(v2) - угол между векторами v1 и v2.

Методы RotateX(angle), RotateY(angle), RotateZ(angle) поворачивают вектор вокруг соответствующей оси на угол angle, angle - величина типа Double_t. Метод Rotate(angle, v) вращает вектор вокруг другого вектора v.

13.3. Класс TRotation

13.3.1. Декларация и доступ к компонентам

Объект класса TRotation представляет собой матрицу 3 на 3 из величин типа Double_t

$$\begin{vmatrix} xx & xy & xz \\ yx & yy & yz \\ zx & zy & zz \end{vmatrix}$$

По умолчанию объект TRotation инициализирован диагональной матрицей с диагональными элементами, равными единице, и остальными элементами, равными нулю. Класс TRotation описывает вращение объекта TVector3 внутри статической системы координат. Если необходимо повернуть систему отсчета и узнать координаты объектов во вращаемой системе, к объектам применяется обратное вращение. Чтобы преобразовать координаты из вращаемой системы в первоначальную, применяется прямое преобразование. Вращение вокруг указанной оси означает вращение против часовой стрелки вокруг положительного направления оси. Объявим переменную r объектом рассматриваемого класса:

```
root [] TRotation r.
```

Прямых методов установки матричных элементов объекта TRotation не существует, но можно получить эти значения методами XX(), XY(), ... ZZ().

13.3.2. Вращение вокруг осей и произвольных векторов

Матрицы

$$Rx(angle) = \begin{vmatrix} 1 & 0 & 0 \\ 1 & cos(angle) & -sin(angle) \\ 0 & sin(angle) & cos(angle) \end{vmatrix},$$

$$Ry(angle) = \begin{vmatrix} cos(angle) & 0 & sin(angle) \\ 0 & 1 & 0 \\ -sin(angle) & 0 & cos(angle) \end{vmatrix}$$

и

$$Rz(angle) = \begin{vmatrix} cos(angle) & -sin(angle) & 0 \\ cos(angle) & -sin(angle) & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

описывают вращения против часовой стрелки вокруг координатных осей. Эти вращения реализуются методами: `RotateX(angle)`, `RotateY(angle)` и `RotateZ (angle)`. Метод `Rotate(angle, vector)` совершает вращение вокруг произвольного вектора, например

```
root [] r.Rotate(TMath::Pi()/3, TVector3(2,2,2))
```

Чтобы узнать результат, воспользуйтесь методами `XX()`, `XY()`, ... `ZZ()`. Метод `RotateAxes(const TVector3& newX, const TVector3& newY, const TVector3& newZ)` добавляет вращение локальных осей к текущему вращению и возвращает результат. Методы `ThetaX()`, `ThetaY()`, `ThetaZ()`, `PhiX()`, `PhiY()`, `PhiZ()` возвращают азимутальный и полярный углы вращаемых осей.

13.3.3. Обратное и составные вращения

Если `r1` и `r2` объекты класса `TRotation`, то произвести обратное вращение можно двумя способами:

```
r2=r1.Inverse()
```

в этом случае матрица `r2` вычисляется как обратная к `r1`, а `r1` не изменяется,

```
r2=r1.Invert()
```

в этом случае матрица `r1` обращается и происходит присвоение `r2=r1`.

Оператор `*` в классе `TRotation` был реализован согласно математическим правилам произведения двух матриц, описывающих два последовательных вращения. Поэтому при умножении двух объектов класса `TRotation` сначала записывается второе вращение, а затем первое, например `r = r2 * r1`.
Если `r` и `v` объекты класса `TRotation` и `TVector3`, то синтаксис `v = r * v` со-

ответствует математической записи

$$\begin{vmatrix} x' \\ y' \\ z'' \end{vmatrix} = \begin{vmatrix} xx & xy & xz \\ yx & yy & yz \\ zx & zy & zz \end{vmatrix} * \begin{vmatrix} x \\ y \\ z \end{vmatrix}.$$

13.4. Класс TLorentzVector

13.4.1. Декларация и доступ к компонентам

Этот класс описывает четырехмерные векторы и используется для определения как позиции и времени - x, y, z, t, так и импульса и энергии - px, py, px, E. Конструктор

```
TLorentzVector(Double_t x = 0.0, Double_t y = 0.0, Double_t z = 0.0,
Double_t t = 0.0)
```

или

```
TLorentzVector(const TVector3& vector3, Double_t t)
```

Есть два набора функций доступа к компонентам объекта TLorentzVector: X(), Y(), Z(), T() и Px(), Py(), Pz(), E(). Оба набора возвращают одни и те же значения, но первый удобно применять для четырехмерного вектора, описывающего позицию и время, а второй - для вектора энергии-импульса. Компоненты TLorentzVector могут быть также доступны через индексы, аналогично случаю объектов TVector3. Метод Vect() получает векторную компоненту от TLorentzVector. Для установки компонентов имеются методы SetX(), ... и SetPx(), ... Чтобы устанавливать все компоненты одним вызовом, используются методы SetXYZT() и SetPxPyPzE(), а для установки TVector3 части - метод SetVect().

13.4.2. Переход в недекартовы координаты

Методы получения и установки параметров TVector3 части в сферических координатах следующие:

- Rho() - вычисляет величину $\rho = \sqrt{x * x + y * y + z * z}$;
- Theta() - вычисляет полярный угол;
- CosTheta() - вычисляет косинус Θ;
- Phi() - вычисляет азимутальный угол;
- SetRho(Double_t rho) - устанавливает величину ρ;
- SetTheta(Double_t theta) - устанавливает полярный угол;
- SetPhi(Double_t phi) - устанавливает азимутальный угол.

В цилиндрической системе координат можно получить величину $\tilde{\rho} = \sqrt{x*x+y*y}$ и ее квадрат методами `Perp2()` и `Perp2()`. Если список параметров этих методов не пустой, а в нем указан другой вектор, то методы вычисляют поперечную компоненту первого вектора относительно второго.

Также имеются методы для установки сразу нескольких параметров

`SetPtEtaPhiE(Double_t pt, Double_t eta, Double_t phi, Double_t e)` и `SetPtEtaPhiM(Double_t pt, Double_t eta, Double_t phi, Double_t m)`.

13.4.3. Арифметические действия, инвариантная масса, β и γ

Для сложения, вычитания и сравнения четырехмерных векторов используются соответствующие операторы C++.

Скалярное произведение двух векторов вычисляется согласно формуле $s = v1*v2 = t1*t2 - x1*x2 - y1*y2 - z1*z2$.
Соответствующий метод - `s = v1.Dot(v2)`. Инвариантная масса вычисляется согласно формуле $m = \sqrt{e*e - px*px - py*py - pz*pz}$. Соответствующий метод - `M()` или `Mag()`, а для вычисления квадрата от массы используется `M2()` или `Mag2`. Метод `Angle(const TVector3& v)` вычисляет угол между четырехмерным и трехмерным векторами и возвращает величину типа `Double_t`.

13.4.4. Вращения

Следующие четыре метода вращают `TVector3` компоненту `TLorentzVector` вектора вокруг `X-`, `Y-`, `Z-` осей и произвольного вектора: `RotateX(Double_t angle)`, `RotateY(Double_t angle)`, `RotateZ(Double_t angle)` и `Rotate(Double_t a, const TVector3& v)`.

Метод `RotateUz(TVector3& direct)` совершает преобразование из вращаемой системы отсчета, при этом вектор `direct` должен быть единичным.

13.5. Класс TLorentzRotation

13.5.1. Декларация и доступ к компонентам

`TLorentzRotation` класс описывает лоренцевские преобразования, включая лоренцевские увеличения и вращения. Объект класса `TLorentzRotation` представляет собой матрицу 4 на 4.

$$\begin{vmatrix} xx & xy & xz & xt \\ yx & yy & yz & yt \\ zx & zy & zz & zt \\ tx & ty & tz & tt \end{vmatrix}$$

По умолчанию объект `LorentzTRotation` инициализирован диагональной матрицей с диагональными элементами, равными единице, и остальными элемента-

ми, равными нулю. Доступ к матричным компонентам осуществляется методами `XX()`, `XY()`, ..., `TT()`.

13.5.2. Различные преобразования объекта TLorentzRotation

Если `a`, `b` и `c` являются объектами `TLorentzRotation`, то найти произведение двух `TLorentzRotation` преобразований можно несколькими способами:

```
c=a*b
c=a.MatrixMultiplication(b)
a *=b
c=a.Transform(b)
```

В первых двух случаях матрица не изменяется.

Вращения объекта `TLorentzRotation` вокруг `X-`, `Y-`, `Z-` осей и произвольного вектора производится следующими методами: `RotateX(Double_t angle)`, `RotateY(Double_t angle)`, `RotateZ(Double_t angle)` и `Rotate(Double_t a, const TVector3& v)`.

При обратном преобразовании, чтобы сохранить первоначальный объект `TLorentzRotation` неизменяемым, используйте метод `Inverse()`. `Invert()` инвертирует первоначальный `TLorentzRotation`. Чтобы применить `TLorentzRotation` к объекту `TLorentzVector`, можно использовать метод `VectorMultiplication(const TLorentzVector& p)` или `*` оператор.

Пример использования классов физических векторов можно найти в `test/TestVectors.cxx`.

14. Заключение

ROOT является постоянно развивающимся инструментом обработки данных. В течение года обычно выходит несколько версий этого программного продукта. И каждый день появляется что-то новое. Поэтому невозможно в одном издании отразить все, что входит в этот пакет программ. Так, в данную книгу не вошли такие компоненты ROOT, как геометрический пакет, использование шаблонов, ROOT GUI классы и автоматическая генерация HTML документации. Но в помощь пользователю существует очень много открытых программ, в которых можно увидеть изящные приемы работы с ROOT. Например, в библиотеке программ ОИЯИ "JINRLIB" находятся программы, написанные с использованием ROOT GUI классов: "Пакет программ для фильтрации данных с помощью лифтинг-схемы WALF" [6] (автор А.Г.Соловьев) и "Программа для анализа угловых распределений вторичных частиц с помощью вейвлет-преобразований WASP" [7] (авторы А.Г.Соловьев, А.В.Стадник, Г.А.Ососков, М.В.Алтайский). Надеюсь, что работа с ROOT принесет Вам истинное удовольствие и поможет решить стоящие перед Вами задачи.

15. Литература

1. The ROOT Users Guide 3.05./ *Ed. by R.Brun, F.Rademakers, S.Panacek, I.Antcheva, D.Buskulic* — 2003.

2. *Б.Страуструп.* Язык программирования C++. — СПб.; М.:"Невский Диалект" - "Издательство БИНОМ", 2000.

3. *С.Мейерс.* Эффективное использование C++. — М.:ДМК, 2000.

4. *С.Мейерс.* Наиболее эффективное использование C++. — М.:ДМК, 2000.

5. *В.Игнатов.* Эффективное использование GNU Make. — 2000.

6. http://www.jinr.ru/programs/jinrlib/walf/index.html

7. http://www.jinr.ru/programs/jinrlib/wasp/index.html

Получено 15 сентября 2003 г.

Учебное издание

Т. М. Соловьева

Введение в объектно-ориентированный анализ на примере пакета ROOT

Редактор *Е. В. Калинникова*
Макет *Е. В. Сабаевой*

Подписано в печать 07.10.2003.
Формат 60 × 90/16. Бумага офсетная. Печать офсетная.
Усл. печ. л. 5,6. Уч.-изд. л. 6,9. Тираж 200 экз.
Заказ № 54132.

Издательский отдел Объединенного института ядерных исследований
141980, г. Дубна, Московская обл., ул. Жолио-Кюри, 6.
E-mail: publish@pds.jinr.ru
www.jinr.ru/publish/