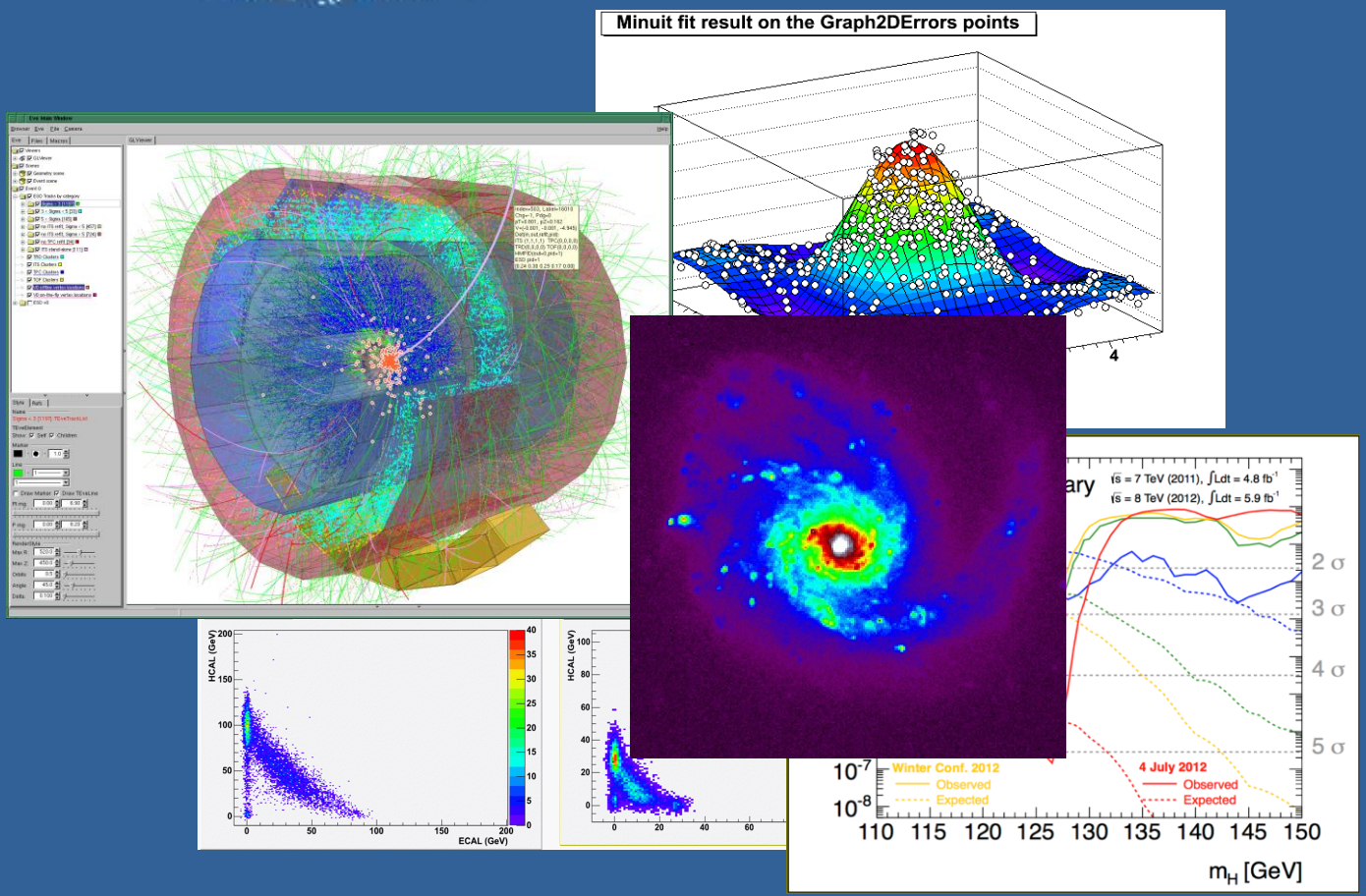
Nhóm NMTP

(Tài liệu lưu hành nội bộ)

Hướng dẫn cơ bản sử dụng

ROOT

Data Analysis Framework



Hướng dẫn cơ bản sử dụng

ROOT

Data Analysis Framework

Đặng Nguyên Phương

TPHCM – 04/2015

"One of the things I really like about programming languages is that it's the perfect execuse to stick your nose into any field. So if you're interested in high energy physics and the structure of the universe, being a programmer is one of the best ways to get in there. It's probably easier than becoming a theoretical physicist."

— Bjarne Stroustrup

MỤC LỤC

LỜI NÓI ĐẦU

Tập tài liệu ***Hướng dẫn cơ bản sử dụng ROOT*** được biên soạn nhằm mục đích giới thiệu cho các bạn sinh viên về gói phần mềm mã nguồn mở ROOT, một trong những công cụ xử lý số liệu phổ biến nhất hiện nay trong ngành Vật lý Hạt nhân và Năng lượng cao. Mặc dù đây là một công cụ khá thông dụng, đã và đang được sử dụng trong hầu hết các thí nghiệm lớn trên thế giới, tuy nhiên lại khá lạ lẫm với các bạn sinh viên chuyên ngành Vật lý Hạt nhân tại Việt Nam. Do đó qua tập tài liệu này, tôi hi vọng sẽ giới thiệu đến các bạn thêm một công cụ xử lý số liệu hữu ích nữa bên cạnh các công cụ mà các bạn đang sử dụng như Origin, Matlab,...

Nội dung của tài liệu được tổng hợp từ nhiều nguồn khác nhau, với phần lớn các ví dụ được lấy từ ROOT User's Guide. Với mục đích trình bày sơ lược cho các bạn những kiến thức và kĩ năng cơ bản nhất trong việc sử dụng ROOT, nội dung của tài liệu khá cô đọng và không đi quá sâu vào chi tiết. Các chủ đề được lựa chọn trình bày trong tài liệu này là các vấn đề cơ bản cần nắm vững mà tôi đã đúc kết được sau một thời gian sử dụng ROOT. Mặc dù vẫn còn rất nhiều chủ đề chưa được đề cập đến trong tài liệu, tuy nhiên với những gì được trình bày, hi vọng các bạn sẽ nắm bắt được phần nào những kiến thức nền tảng của ROOT để từ đó có thể tìm hiểu sâu hơn về chương trình này thông qua các tài liệu khác được phổ biến trên mạng internet. Đặc biệt đối với những bạn có định hướng nghiên cứu về lĩnh vực thực nghiệm hạt cơ bản, đây gần như là một công cụ không thể thiếu trong công việc của các bạn sau này.

Mặc dù đã rất cố gắng, nhưng do thời gian hạn hẹp nên chắc chắn tài liệu này vẫn còn nhiều thiếu sót về nội dung cũng như hình thức trình bày. Rất mong nhận được sự đóng góp ý kiến của các bạn để việc biên soạn tài liệu ngày càng tốt hơn.

Đặng Nguyên Phương

CHƯƠNG 1

GIỚI THIỆU VỀ ROOT

ROOT là một *framework* (thư viện chứa các mã lệnh) dùng cho việc phân tích và xử lý số liệu, chủ yếu cho ngành Vật lý hạt nhân và hạt cơ bản (hay còn gọi là Vật lý năng lượng cao). Chương trình được phát triển đầu tiên bởi René Brun và Fons Rademakers vào giữa những năm 1990 tại Tổ chức Nghiên cứu Nguyên tử Châu Âu (*Conseil Européen pour la Recherche Nucléaire* – CERN) dựa trên ngôn ngữ lập trình C++ với khả năng lập trình hướng đối tượng (*object-oriented programing*).

1.1 Lịch sử hình thành

ROOT được phát triển trong bối cảnh dự án thí nghiệm NA49 tại Trung tâm Nghiên cứu Nguyên tử Châu Âu (CERN). Thí nghiệm này đòi hỏi phải xử lý một lượng dữ liệu rất lớn, khoảng 10 Terabytes[1]/run, do đó đòi hỏi phải phát triển một thế hệ công cụ xử lý dữ liệu mới để thay thế cho công cụ hiện đang có.

Vào giữa những năm 1990, René Brun và Fons Rademakers sau nhiều năm phát triển các công cụ xử lý và mô phỏng cho các thí nghiệm vật lý năng lượng cao (*particle physics* hay *high energy physics*) như PAW, PIAF và GEANT đã nhận ra rằng các thư viện ngôn ngữ lập trình Fortran dường như đã đạt tới giới hạn của nó. Dù cho vẫn còn rất thông dụng, các công cụ này cần phải được nâng cấp để có thể tương thích cho lượng dữ liệu khổng lồ của các thí nghiệm trong tương lai, đặc biệt là các thí nghiệm tại *Large Hadron Collider* (LHC) sắp tới.

Cùng lúc đó, sự phát triển của khoa học máy tính đặc biệt là kĩ thuật lập trình hướng đối tượng đã tạo tiền đề để các nhà khoa học hướng tới việc xây dựng một công cụ dựa trên nền tảng kĩ thuật này. Năm 1994, Brun và Rademakers bắt đầu tiến hành xây dựng và phát triển ROOT; đến khoảng những năm đầu thế kỉ 21 nó đã thay thế hoàn toàn cho các framework được xây dựng bằng ngôn ngữ Fortran trước đó tại CERN[2].

Sự phát triển của ROOT phần lớn là nhờ vào sự tương tác của những người dùng (*user*) và người phát triển (*developer*). Vì đây là mã nguồn mở (*open source*) cho nên bản thân người dùng cũng có thể trở thành người phát triển bằng cách xây dựng và tích hợp các mã nguồn của mình vào trong ROOT, nó được gọi là *"phong cách Bazaar"* (*Bazaar style*)[3].

[1] Kí hiệu là TB, 1 TB = 10^{12} hay 2^{40} B

[2] CERN đã hoàn toàn ngừng việc phát triển các thư viện xây dựng bằng ngôn ngữ Fortran kể từ năm 2003.

[3] Dựa theo thuật ngữ trong cuốn sách ***"The Cathedral and the Bazaar"*** của Eric S. Raymond (các bạn có thể tham khảo thêm về cuốn sách này tại đây http://www.catb.org/~esr/writings/cathedral-bazaar/), trong đó

Từ năm 1995 đến nay, đã có rất nhiều phiên bản của ROOT được ra đời

- Phiên bản đầu tiên ROOT 0.5 được công bố vào tháng 11/1995.
- Bắt đầu sử dụng trình thông dịch CINT cho ROOT từ năm 1996.
- Các phiên bản ROOT 1.0 (1997), ROOT 2.0 (1998), ROOT 3.0 (2001), ROOT 4.0 (2004) và ROOT 5.0 (2005) lần lượt ra đời.
- Phiên bản ROOT 5.34 ra đời vào năm 2013.
- Phiên bản ROOT 6.00 ra đời vào tháng 7/2014 và sau đó là 6.02 vào tháng 9/2014.

Mặc dù được xây dựng ban đầu cho mục đích xử lý số liệu hạt cơ bản, ROOT đã nhanh chóng lan sang nhiều lĩnh vực ứng dụng khác như thiên văn học (*astronomy*), khai phá dữ liệu (*data mining*),... Danh sách một số ứng dụng tiêu biểu của ROOT có thể xem tại đây http://root.cern.ch/drupal/content/example-applications.

Để có thể tìm hiểu về ROOT một cách trọn vẹn hơn, các bạn có thể vào trang web của ROOT tại http://root.cern.ch/drupal/; hoặc tham gia trao đổi, thảo luận với các chuyên gia tại http://root.cern.ch/drupal/content/roottalk-digest.

1.2 Một số tính năng cơ bản của ROOT

Một điểm đặc trưng dễ nhận thấy nhất của ROOT đó là việc tổ chức, sắp xếp dữ liệu dưới dạng cây (*tree*) nhằm giúp cho việc lưu trữ và xử lý một lượng lớn dữ liệu được hiệu quả hơn. Hiện nay, chương trình này đang được sử dụng trong các thí nghiệm tại Large Hadron Collider (LHC) và tại hầu hết các thí nghiệm vật lý năng lượng cao trên thế giới.

- **Lưu trữ dữ liệu:** chúng ta có thể lưu trữ các dữ liệu (thô hoặc đã qua xử lý) và thậm chí là các đối tượng C++ vào trong các file nén dạng *.root*. Các dữ liệu được lưu lại dưới dạng cấu trúc cây (*tree*) giúp cho tốc độ truy xuất dữ liệu được tăng lên nhanh chóng so với việc truy xuất các dữ liệu dạng thông thường.
- **Truy xuất dữ liệu:** dữ liệu được lưu trữ trong các file root có thể được truy xuất từ máy tính hoặc thông qua mạng internet. Với các dữ liệu có kích thước lớn, ta có thể lưu trữ trên mạng grid. Các dữ liệu có thể được chia nhỏ ra nhiều file root khác nhau và được liên kết lại trong quá trình xử lý.
- **Xử lý dữ liệu:** ROOT là một thư viện lớn tập hợp rất nhiều các công cụ toán học và thống kê giúp chúng ta có thể xử lý dữ liệu một cách tốt nhất. ROOT cũng hỗ trợ các xử lý theo hướng lập trình hướng đối tượng hoặc lập trình song song.
- **Trình bày kết quả:** các kết quả có thể được trình bày theo rất nhiều dạng khác nhau (histogram, chấm điểm, đường,...) và có thể được hiệu chỉnh trực tiếp trên hình vẽ. Các hình vẽ trong ROOT có thể được xuất ra theo nhiều định dạng khác nhau (pdf, eps, png,...).
- **Mô phỏng:** ROOT có khả năng mô tả các hình học không gian phức tạp, việc mô phỏng trong ROOT được thực hiện thông qua các gói mô tả hình học, tracker và Monte Carlo ảo.

---

phân chia các mô hình phát triển phần mềm mã nguồn mở thành hai loại

- Mô hình Cathedral (*Cathedral model*): mã nguồn mở cho từng phiên bản phát hành (*release*), việc phát triển mã nguồn của từng phiên bản chỉ được giới hạn bên trong đội ngũ phát triển (ví dụ: Emacs, GCC,...)
- Mô hình Bazaar (*Bazaar model*): mã nguồn được phát triển chung thông qua mạng internet, mọi người đều có thể tham gia (ví dụ: nhân Linux)

1.3 Cách thức cài đặt ROOT

ROOT là gói phần mềm mã nguồn mở miễn phí, có thể được download tại trang web http://root.cern.ch/drupal/, thích hợp cho rất nhiều hệ điều hành như Linux, Windows, Mac OS,... Trong phần này tôi sẽ trình bày cách thức cài đặt ROOT cho hai hệ điều hành thông dụng là Windows và Linux.

Để cài đặt được ROOT trước tiên ta sẽ vào trang
http://root.cern.ch/drupal/content/downloading-root
kéo xuống nhấp chọn phiên bản (*version*) cần cài đặt (Hình 1.1).

- **Pro**, version 6.02/00 experimental (see also the release notes)
- **Old**, version 6.00/02 experimental (see also the release notes)
- **Pro**, version 5.34/21 recommended (see also the release notes)
- **Old**, version 5.32/04 (see also the release notes and development notes)
- **Old**, version 5.30/06 (see also the release notes and development notes)
- **Old**, version 5.28/00h (see also the release notes and development notes)
- **Old**, version 5.26/00 (see also the release notes and development notes)
- **Old**, version 5.24/00 (see also the release notes and development notes)
- **Old**, version 5.22/00 (see also the release notes and development notes)
- **Old**, version 5.20/00 (see also the release notes and development notes)
- **Old**, version 5.18/00 (see also the release notes and development notes)
- **Old**, version 5.16/00 (see also the release notes and development notes)
- **Old**, version 5.14/00 (see also the release notes and development notes)
- **Old**, version 5.12/00 (see also the release notes and development notes)
- **Old**, version 5.10/00 (see also the release notes and development notes)
- **Old**, version 5.08/00b (see also the release notes and development notes)
- **Old**, version 4.04/02g (see also the release notes and development notes)
- **Old**, version 4.02/00 (see also the release notes and development notes)
- **Old**, version 4.00/08 (see also the release notes and development notes)
- **Old**, version 3.10/02 (see also the release notes and development notes)
- **Old**, version 3.05/07 (see also the development notes)
- **Old**, version 3.04/02 (see also the release notes)
- **Old**, version 3.03/09 (see also the release notes)
- **Old**, version 3.02/07 (see also the release notes)
- **Old**, version 3.01/06 (see also the release notes)
- **Old**, version 3.00/06 (see also the release notes)

Hình 1.1: Các phiên bản của ROOT

Ứng với mỗi phiên bản sẽ có nhiều file nguồn khác nhau để cài đặt, dưới đây sẽ là hướng dẫn chi tiết cách cái đặt ứng ứng với từng hệ điều hành.

1.3.1 Hệ điều hành Windows

Đối với các phiên bản ROOT 5.34 trở về trước, ta có thể cài đặt thẳng file binary đã được biên dịch bằng cách chọn các file có đuôi **.msi** (*Microsoft Installer*) tại phần **Binaries** như trong Hình 1.2.

Sau khi đã tải file **.msi** về, ta nhấp đôi chuột vào file và cài đặt như những chương trình bình thường.

Tuy nhiên với các phiên bản ROOT 6.00 trở về sau hiện tại chưa có file **.msi**, file binary hiện có được biên dịch trong môi trường Cygwin, do đó để có thể sử dụng được ta phải cài đặt Cygwin.

	Release	Debug
VC++ 9 (2008)	MSI (65.9 MB) tar (66.1 MB)	MSI (157 MB) tar (158 MB)
VC++ 10 (2010)	MSI (69 MB) tar (69 MB)	MSI (146 MB) tar (147 MB)
VC++ 11 (2012)	**MSI (72 MB)** **tar (72 MB)**	MSI (160 MB) tar (162 MB)

Hình 1.2: Các file đã được biên dịch

1.3.2 Hệ điều hành Linux

Đối với hệ điều hành Linux, cách tốt nhất là biên dịch từ mã nguồn (*source code*). Đầu tiên ta cần tải mã nguồn bằng cách vào trang web như được mô tả trong Hình 1.1), sau đó ta sẽ tìm đến phần **Source** và nhấp vào đường link *ROOT 6.xx.yy complete source tree* để tải file mã nguồn (có dạng `root_v6.xx.yy.source.tar.gz`).

Trước khi tiến hành biên dịch mã nguồn, ta cần phải đảm bảo rằng trong hệ thống đã có sẵn một số gói ứng dụng (*package*) cần thiết cho quá trình biên dịch. Mỗi một hệ điều hành sẽ đòi hỏi các gói ứng dụng khác nhau, ta có thể kiểm tra các gói cho hệ điều hành tương ứng tại http://root.cern.ch/drupal/content/build-prerequisites. Để cài đặt các gói ứng dụng này, ta sử dụng các lệnh sau

- Đối với **Fedora**

```
$ yum install <package1> <package2> ...
```

- Đối với **Ubuntu**

```
$ sudo apt-get install <package1> <package2> ...
```

- Đối với **openSUSE**

```
$ sudo zypper install <package1> <package2> ...
```

- Đối với **AIX**

```
$ rpm -Uvh <package1> <package2> ...
```

Sau khi đã download mã nguồn và các gói ứng dụng cần thiết, ta tiến hành giải nén file mã nguồn bằng lệnh

```
$ tar -zxvf root_v6.xx.yy.source.tar.gz
```

Sau khi giải nén, một thư mục *root/* sẽ được tạo ra tại vị trí đặt file mã nguồn. Ta di chuyển vào bên trong thư mục *root/* này để thực hiện việc cài đặt

```
$ cd root
```

Ở đây có hai phương thức cài đặt: cài đặt vị trí không cố định và cài đặt với vị trí cố định. Phương thức thứ nhất thích hợp với mục đích cài đặt ROOT để sử dụng cho cá nhân, phương thức thứ hai thích hợp với mục đích cài đặt cho toàn hệ thống.

- Để cài đặt theo phương thức thứ nhất, ta gõ các lệnh sau

```
$ ./configure
$ make
```

- Để cài đặt theo phương thức thứ hai, ta gõ các lệnh sau

```
$ ./configure --prefix=/usr/local
$ make
$ sudo make install
```

Đến đây là quá trình cài đặt ROOT đã kết thúc, tuy nhiên để có thể sử dụng được ROOT, ta cần phải khai báo đường dẫn của thư mục ROOT vào trong các biến môi trường `PATH` (tới thư mục *bin/*) và `LD_LIBRARY_PATH` (tới thư mục *lib/*) để chương trình có thể tìm được các file thực thi và thư viện của ROOT. Để làm được điều đó ta có thể sử dụng một trong các cách sau

- Cách 1: sử dụng lệnh `export`

```
$ export ROOTSYS=<duong dan toi thu muc root>
$ export PATH=$ROOTSYS/bin:$PATH
$ export LD_LIBRARY_PATH=$ROOTSYS/lib:$LD_LIBRARY_PATH
```

- Cách 2: sử dụng script có sẵn của ROOT, tại thư mục *root/* ta gõ

```
$ . bin/thisroot.sh
```

hoặc

```
$ source bin/thisroot.csh
```

- Cách 3: cập nhật cache bằng cách gõ lệnh (trong trường hợp cài theo phương thức thứ hai)

```
$ ldconfig
```

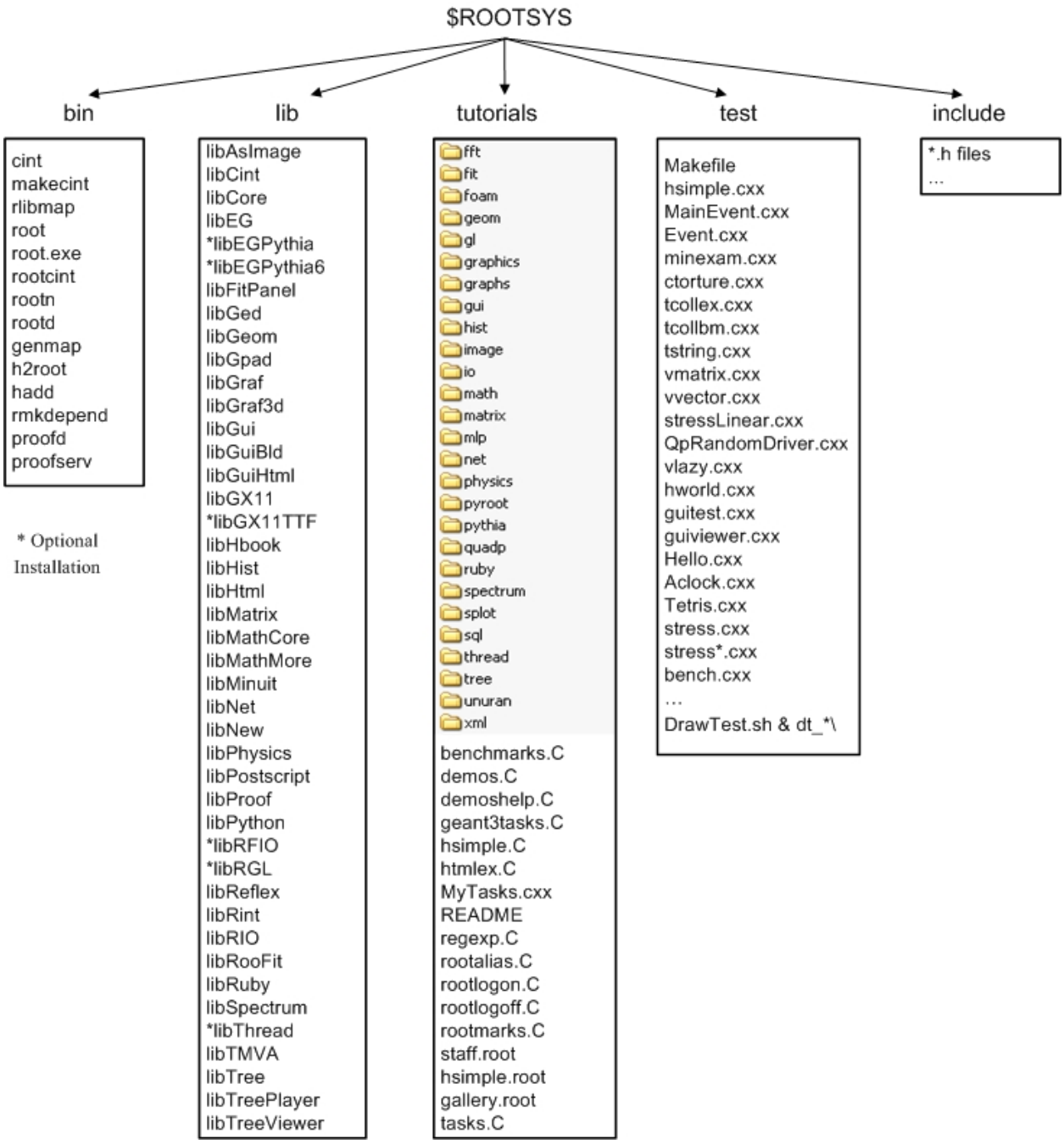
Hai cách đầu tiên phải được thực hiện mỗi lần khởi động terminal. Để thuận tiện cho việc sử dụng ROOT mà không cần phải khai báo nhiều lần, ta có thể mở file *∼/.bashrc* tại thư mục gốc và thêm các lệnh trên vào trong file này (ở cách 2 cần khai báo đường dẫn đầy đủ của các script). Với cách này, chương trình sẽ tự động thực hiện các lệnh trên mỗi khi khởi động một terminal mới.

1.4 Cấu trúc của ROOT

Hình 1.3 trình bày cấu trúc của ROOT. Các thư mục chính của ROOT gồm có: *bin*, *lib*, *include*, *tutorials* và *test*.

bin chứa các file thực thi (*executable*). Một số file quan trọng trong thư mục bin:

- **root.exe:** chạy root
- **rootcint:** cung cấp thư viện các lớp cho CINT
- **rmkdepend:** phiên bản của `makedepend` cho ROOT
- **root-config:** script xác định các flag và thư viện cho việc biên dịch chương trình có sử dụng ROOT
- **cint:** chạy trình thông dịch C++
- **makecint:** xác định các flag và thư viện của rootcint
- **proofd:** chương trình daemon tạo user cho quá trình tính toán song song bằng ROOT (PROOF)
- **proofserv:** thực hiện tính toán PROOF
- **rootd:** chương trình daemon dùng để điều khiển việc truy xuất các file



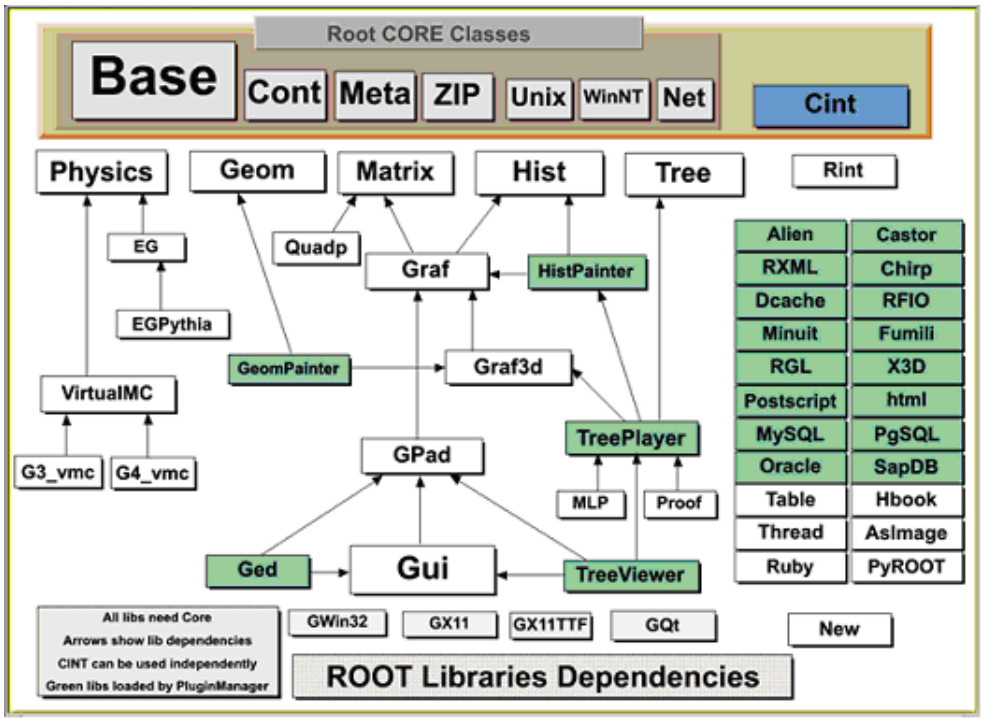
Hình 1.3: Cấu trúc của ROOT

lib chứa thư viện chia sẻ (*shared object*) của các lớp được định nghĩa trong ROOT, người dùng có thể liên kết các thư viện này với chương trình của mình. Hình 1.4 trình bày sự phụ thuộc giữa các thư viện với nhau.

include chứa các file header mô tả định nghĩa của các lớp.

tutorials chứa các file ví dụ giúp người dùng nắm rõ thêm về các khái niệm, kĩ thuật trong ROOT.

test chứa các file ví dụ để test framework. Khi một phiên bản mới của framework được xây dựng, các ví dụ trong thư mục này sẽ được biên dịch và thực thi để test khả năng của phiên bản mới.



Hình 1.4: Sự phụ thuộc của các thư viện trong ROOT

1.5 Các lớp của ROOT

Trong ROOT hiện có khoảng hơn 250 lớp (*class*) được chia thành 11 loại:

- Các lớp cơ bản: bao gồm các lớp ở tầng thấp nhất trong cấu trúc lớp của ROOT, làm nền tảng để xây dựng nên các lớp khác. VD: `TObject`, `TClass`,...
- Các lớp lưu trữ: bao gồm các lớp tạo cấu trúc dữ liệu (*array*, *list*, *set*,...)
- Các lớp histogram:bao gồm các lớp histogram 1D, 2D, 3D, các lớp làm khớp dữ liệu, hàm.
- Các lớp ntuple: bao gồm các lớp như `TTree`, `TNtuple`,...
- Các lớp đồ họa 2D: bao gồm các đồ họa 2D cơ bản (đường thẳng, hình hộp, ellipse,...)
- Các lớp đồ họa 3D: bao gồm có khối hình học 3D và các hình học detector.
- Các lớp giao diện người dùng MOTIF: chứa các thành phần giao diện tương tự như các công cụ khác như cửa sổ, nút bấm, menu,...
- Các lớp giao diện tương tác: chứa các ứng dụng tương tác, trình thông dịch C++.
- Các lớp tương tác với hệ điều hành: các lớp tương tác với hệ điều hành thông qua `TSystem` (`TUnixSystem`, `TWinNTSystem`, `TMacSystem`).
- Các lớp mạng (network): các ứng dụng xây dựng hệ thống mạng.
- Các lớp tài liệu: cho phép tạo các tài liệu (dạng HTML), file header và source C++,...

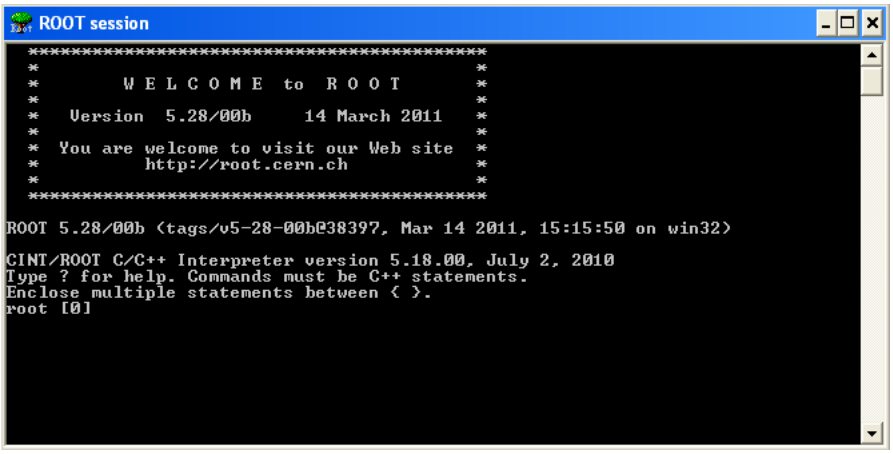
1.6 Chạy chương trình

1.6.1 Cách chạy chương trình trong ROOT

Để khởi động ROOT, ta có thể thực hiện 1 trong 2 cách sau:

- **Cách 1:** double click vào biểu tượng ROOT được tạo ra sau khi cài đặt.

- **Cách 2:** mở cửa sổ **Command Prompt**, gõ lệnh *root* hoặc *root -l*.
- Gõ lệnh *.q* để thoát ra khỏi ROOT.



Hình 1.5: Giao diện khi khởi động của ROOT

Để thực thi các lệnh, ta có thể gõ các lệnh này trực tiếp tại dấu nhắc lệnh của ROOT hoặc tạo một file macro và chạy file này tại dấu nhắc lệnh của ROOT. Thông thường các file macro được sử dụng trong ROOT là các file C++, cho nên thường có đuôi dạng như *.c*, *.cpp* hoặc *.cxx*. Các lệnh trong file macro được bao lại bởi một cặp ngoặc nhọn *{}*.

Ví dụ: *tạo một file macro* ***ten.cxx*** *yêu cầu người dùng nhập vào tên của mình:*

```
{
    Char_t ten[100];
    cout << "Moi ban nhap vao ten cua minh: ";
    gets(ten);
    cout << "Ten cua ban la " << ten;
}
```

Ta cũng có thể đặt các lệnh này trong một hàm, chẳng hạn như

```
void ten()
{
    Char_t ten[100];
    cout << "Moi ban nhap vao ten cua minh: ";
    gets(ten);
    cout << "Ten cua ban la " << ten;
}
```

Để thực thi file macro, tại dấu nhắc lệnh của ROOT, ta gõ `.x tenfile`. Để load macro, ta gõ `.L tenfile`.

Ví dụ: *Load macro*

```
root [0] .L ten.cxx
```

Thực thi macro

```
root [0] .x ten.cxx
```

Ngoài ra ta cũng có thể gọi thực thi macro ngay từ bên ngoài ROOT bằng cách gõ lệnh như sau

```
$ root ten.cxx
```

Trong trường hợp thực thi macro, nếu file macro có chứa nhiều hàm bên trong, ta cần đặt tên hàm chính trong file trùng với tên của file.

1.7 Một số chú ý khi sử dụng ROOT

Trong phần này tôi sẽ trình bày một số quy ước kí hiệu và những điều cần lưu ý trong ROOT để các bạn có thể dễ dàng theo dõi và nắm bắt ý nghĩa của các ví dụ ở những phần sau.

1.7.1 Quy ước

Các quy ước kí hiệu trong ROOT gồm có:

- Các lớp bắt đầu bằng kí tự `T`, ví dụ như `TH1`, `TTree`,...
- Các khai báo không thuộc lớp được kết thức bằng `_t`, ví dụ như `Int_t`, `Float_t`,...
- Các thành phần của dữ liệu được bắt đầu bằng kí tự `f`, ví dụ như `fTree`,...
- Các hàm, lệnh trong lớp được bắt đầu bằng kí tự hoa, ví dụ như `Loop()`, `Integral()`,...
- Các hằng số được bắt đầu bằng kí tự `k`, ví dụ như `kRed`, `kInitialSize`,...
- Các biến toàn cục được bắt đầu bằng kí tự `g`, ví dụ như `gEnv`, `gSystem`,...
- Các thành phần tĩnh của dữ liệu được bắt đầu bằng kí tự `fg`, ví dụ như `fgTokenClient`,...
- Các biến cục bộ và tham số được bắt đầu bằng kí tự thường, ví dụ như `nbytes`,...
- Các lệnh, hàm gán giá trị và truy xuất giá trị được bắt đầu bằng các từ `Set` và `Get`, ví dụ như `SetLast()`, `GetFirst()`,...

1.7.2 Một số biến toàn cục quan trọng

- `gROOT` lưu giữ thông tin của hiện tại trong ROOT, bằng cách sử dụng `gROOT` ta có thể truy cập tới tất cả các đối tượng đang được tạo ra trong ROOT.
- `gSystem` tương tác với hệ điều hành.
- `gFile` trỏ đến file đang mở trong ROOT.
- `gDirectory` trỏ đến thư mục đang làm việc.
- `gPad` trỏ đến pad đang hoạt động.
- `gRandom` trỏ đến bộ phận phát số ngẫu nhiên đang hoạt động.
- `gEnv` chứa tất cả các thiết lập môi trường trong quá trình sử dụng ROOT.

Ví dụ:

```
root [0] gROOT->Reset();
root [1] gSystem->Load("../lib/libQFramework.so");
root [2] gStyle->SetFrameBorderMode(0);
```

1.8 CINT

CINT là một trình thông dịch C++ được tạo bởi Masa Goto, và được tích hợp trong ROOT để thông dịch các lệnh C++ do người dùng tạo ra. Trình thông dịch này có nhiệm vụ đọc và thực thi từng câu lệnh do người dùng tạo ra, nhưng không biên dịch thành ngôn ngữ máy. Mặc dù việc này làm cho tốc độ chạy một chương trình thông dịch chậm hơn rất nhiều so với một chương trình được biên dịch, tuy nhiên việc này sẽ thuận tiện hơn cho người dùng trong trường hợp phải chỉnh sửa mà nguồn nhiều lần do thời gian biên dịch lâu hơn nhiều so với thời gian thông dịch.

Mặc dù được tích hợp trong ROOT nhưng CINT là một chương trình độc lập và có thể download tại http://root.cern.ch/drupal/content/cint.

Cách viết nhiều dòng lệnh

- Các cụm dòng lệnh bắt đầu và kết thúc bằng cặp ngoặc nhọn `{}`
- Các lệnh bên trong kết thúc bằng dấu `;`
- Tất cả các đối tượng được tạo ra là toàn cục

Kể từ phiên bản ROOT6 trở đi, trình thông dịch Cling[4] sẽ được dùng để thay thế cho CINT (http://root.cern.ch/drupal/content/cling).

1.9 ACLiC

ACLiC (*Automatic Compiler of Libraries for CINT*) là trình biên dịch có nhiệm vụ tạo ra các thư viện chia sẻ dựa trên mã nguồn C++ do người dùng viết ra, sử dụng các tùy chỉnh được thiết lập tương tự như khi thực thi ROOT.

Cách biên dịch macro với ACLiC Để có thể biên dịch được các macro với ACLiC, điều đầu tiên là chúng ta cần khai báo thêm các file header phù hợp vào trong script của macro (điều này là không cần thiết đối với CINT).

Ví dụ:

Để tạo ra các thư viện chia sẻ, ta thêm dấu `+` vào sau tên file macro khi load, chẳng hạn như

```
root [0] .L ten.cxx+
```

Chương trình sẽ biên dịch và tạo ra các file chia sẻ có đuôi *.so*. Ví dụ như file macro của chúng ta có tên là *macro1.cxx* thì file thư viện tạo ra sẽ có tên là *macro1_ cxx.so*

[4] Cling là một trình thông dịch được xây dựng dựa trên cơ sở các thư viện của LLVM (*Low Level Virtual Machine*) và Clang

CHƯƠNG 2

C++ CĂN BẢN CHO ROOT

ROOT là một framework được xây dựng trên nền tảng C++, do đó một số kiến thức về ngôn ngữ lập trình C++là cần thiết để có thể sử dụng được ROOT. Thực tế từ kinh nghiệm của bản thân tôi cho thấy, việc viết một chương trình xử lý bằng ROOT đôi khi không khác gì nhiều so với việc viết một chương trình bằng C++. Đó có thể sẽ là một trong những nguyên nhân làm cho các bạn ngại ngần khi sử dụng ROOT vào việc phân tích số liệu, bởi vì đơn giản là nó quá gần với C++ và không phải ai cũng yêu thích công việc lập trình (đặc biệt là lập trình C++ ☺). Tuy nhiên, lời khuyên của tôi dành cho các bạn là hãy trở lên đầu tài liệu này và đọc lời trích dẫn từ Bjarne Stroustrup của tôi, có thể các bạn sẽ tìm được lý do để yêu thích công việc này.

Trong chương này, tôi sẽ chỉ trình bày một số điểm chính căn bản của ngôn ngữ lập trình C++ để bổ sung kiến thức nền tảng cho việc sử dụng ROOT trong những phần sau[1]. Nếu các bạn thấy rằng các kiến thức về C++ được trình bày ở đây là quá đơn giản hoặc đã từng biết trước đó rồi thì các bạn hoàn toàn có thể bỏ qua phần này mà đi thẳng vào những phần sau.

2.1 Giới thiệu ngôn ngữ lập trình C++

C++ là một ngôn ngữ lập trình đa mục đích (*general purpose*) được Bjarne Stroustrup phát triển từ những năm 1980 tại Bell Labs. Đây là một trong những ngôn ngữ lập trình phổ biến nhất trên thế giới (cùng với C, Objective-C, và Java). Các đặc trưng chính của ngôn ngữ này gồm có

- Dạng tự do (*free-form*): có thể sử dụng khoảng trắng tùy ý
- Kiểu tĩnh (*static type*): tất cả các biến phải được gán kiểu trước khi biên dịch, và kiểu của biến sẽ được kiểm tra trong thời gian biên dịch (*compile-time*)[2]
- Hỗ trợ lập trình thủ tục (*procedural programing*) hay lập trình cấu trúc (*structured programming*): phân chia công việc chính thành các công việc nhỏ hơn và giao cho một hàm đảm nhiệm, chương trình chính sẽ gọi các hàm này vào những thời điểm cần thiết.
- Dữ liệu trừu trượng (*abstract data type*): có sử dụng các kiểu dữ liệu do người dùng tự định nghĩa.

[1] Nội dung của phần này được trích từ tài liệu ***"Ngôn ngữ lập trình C++ (chuẩn 2011)"*** mà tôi đã viết trước đó, nếu muốn tìm hiểu sâu hơn thì các bạn có thể đọc tài liệu tại http://goo.gl/ng8KyS.

[2] Ngược lại là kiểu động (*dynamic type*), các biến có thể mang bất kì kiểu nào, việc gán kiểu cho biến sẽ được thực hiện trong thời gian chạy chương trình (*run-time*)

- Lập trình hướng đối tượng (*object-oriented programming*): coi chương trình là tập hợp của các đối tượng (*object*) có quan hệ nào đó với nhau, mỗi đối tượng có dữ liệu (*data*) và phương thức (*method*) của riêng mình.
- Lập trình đa hình (*polymorphism*): một biến có thể có nhiều kiểu (thông qua con trỏ) hoặc có thể dùng nhiều hàm có cùng tên.

Ngôn ngữ C++ gồm có hai thành phần chính

- Phần ngôn ngữ cốt lõi (*core language*) bao gồm ngôn ngữ lập trình, một số thư viện gốc và các danh định (*identifier*) được biết đến với tên gọi "từ khóa" (*keyword*).
- Thư viện chuẩn C++ (*Standard Template Library – STL*) là một tập hợp các lớp và các hàm được viết bằng ngôn ngữ cốt lõi. Thư viện này cung cấp các container, hàm để làm tiện ích, các đối tượng hàm, các dãy kí tự tổng quát và các dòng dữ liệu (bao gồm I/O tương tác và tập tin), hỗ trợ một số tính năng ngôn ngữ (bao gồm cả thư viện chuẩn C).

2.2 Một số kiến thức cơ bản

2.2.1 Cấu trúc một chương trình C++

Cấu trúc phổ biến của một file C++ như sau

```
#include <header>
int main ()
{
  // Noi dung
}
```

- Các dòng bắt đầu bằng kí tự '#' được gọi là các chỉ thị tiền xử lý (*preprocessor directive*), dùng để báo hiệu cho trình biên dịch. Ví dụ như lệnh `#include <header>` sẽ báo cho trình biên dịch sử dụng các file `header` trong khi biên dịch chương trình.
- Kí tự '//' có tác dụng comment toàn bộ kí tự sau nó trên cùng 1 dòng, trong trường hợp ta muốn comment nhiều hơn 1 dòng thì ta sử dụng cặp kí tự '/*' và '*/' để mở đầu và kết thúc.
- Hàm `main()` là hàm chính của chương trình, đây là nơi mà chương trình bắt đầu thi hành các lệnh. Hàm này bắt buộc phải có khi muốn biên dịch một chương trình viết bằng C++. Hàm `main()` có thể được đặt ở bất kì vị trí nào trong chương trình (đầu, cuối hoặc giữa) và luôn là hàm được thực hiện đầu tiên khi chạy chương trình.

2.2.2 Biến và hằng

Biến và các kiểu dữ liệu của biến

Biến (*variable*) có thể được xem như là vùng nhớ chứa dữ liệu tạm thời trong khi thực thi chương trình. Các dữ liệu được lưu trữ trong biến có thể là các giá trị số, chuỗi kí tự,... và các dữ liệu này có thể thay đổi được trong quá trình thực thi chương trình. Các loại biến mà chúng ta thường gặp gồm có

- Biến toàn cục (*global variable*) có thể được sử dụng ở bất kì đâu trong chương trình, ngay sau khi nó được khai báo.
- Biến cục bộ (*local variable*) tầm hoạt động của biến bị giới hạn trong phần mã (thường là hàm hoặc vòng lặp) mà nó được khai báo.
- Biến ngoài (*external variable*) không những được dùng trong một file mã nguồn mà còn trong tất cả các file được liên kết trong chương trình (thường được khai báo với từ khóa `extern`).

- Biến tự động (*automatic variable*) là các biến cục bộ mà sự tồn tại của nó kết thúc ngay khi việc thực thi kết thúc.
- Biến tĩnh (*static variable*) là các biến mà tồn tại đến cuối chương trình (thường được khai báo với từ khóa `static`).

Các kiểu biến được chia làm 4 nhóm chính là nhóm các biến kiểu kí tự (*character*), kiểu số nguyên (*integer*), kiểu số thực (*real*) và kiểu luận lý (*boolean*). Danh sách các kiểu dữ liệu trong C++ gồm có

Nhóm	Kiểu dữ liệu	Kích thước (B)
Kí tự	`char`	1
	`char16_t`	2
	`char32_t`	4
	`wchar_t`	kích thước lớn nhất cho kiểu `char`
Số nguyên[3]	`int`	2[4]
	`short`	2
	`long`	4
	`long long`	8
Số thực	`float`	4
	`double`	8
	`long double`	10
Luận lý	`bool`	
Trống	`void`	0
Con trỏ trống	`decltype(nullptr)`	0

Bên cạnh các kiểu biến đặc trưng của C++ như trên, chúng ta còn có thể sử dụng các kiểu biến được xây dựng dành riêng cho ROOT: `Char_t` (biến kiểu kí tự), `Int_t` (biến kiểu nguyên), `Float_t` (biến kiểu thực), `Double_t` (biến kiểu thực kép),... (xem thêm ở `http://root.cern.ch/root/html/ListOfTypes.html`).

Để có thể sử dụng một biến, đầu tiên chúng ta cần phải khai báo biến đó. Cách thức khai báo một biến là ghi ra tên kiểu (vd: `Int_t`, `Float_t`,...) và sau đó là tên của biến.

Một tên biến được xem là hợp lệ khi nó là một chuỗi gồm các chữ cái, chữ số hoặc kí tự gạch dưới (không chứa kí tự trống hoặc kí tự đặc biệt). Chiều dài của tên biến là không giới hạn. Một số lưu ý khi đặt tên cho biến

- Tên biến thường bắt đầu bằng một chữ cái.
- Các tên bắt đầu bằng kí tự gạch dưới '_' thường được dành cho các liên kết bên ngoài (*external link*).
- Không bao giờ bắt đầu tên biến bằng một chữ số.
- Không được đặt trùng tên biến với các từ khóa của C++ (xem danh sách tại `http://en.cppreference.com/w/cpp/keyword`) hay từ khóa của ROOT .
- ROOT phân biệt chữ hoa và chữ thường.

Ví dụ: *khai báo hai biến kiểu nguyên x, y (khởi tạo giá trị ban đầu của y là 0) và một biến kiểu kí tự a*

```
root [0] Int_t x, y=0;
```

[3] Các kiểu số nguyên có thể là số có dấu hay không dấu tuỳ theo miền giá trị mà chúng ta cần biểu diễn. Vì vậy khi xác định một kiểu số nguyên chúng ta đặt từ khoá `signed` hoặc `unsigned` trước tên kiểu dữ liệu. Nếu ta không chỉ rõ `signed` hoặc `unsigned` nó sẽ được coi là có dấu.

[4] Kích thước kiểu dữ liệu phụ thuộc vào hệ thống, nhưng không nhỏ hơn kiểu `short`, vd: với hệ thống 32bit thì kiểu dữ liệu này có kích thước là 4byte (32bit).

```
root [1] Char_t a;
```

Hằng

Hằng (*constant*) là bất kì một biểu thức nào mang một giá trị cố định, chẳng hạn như

- Các số

```
123
0.545
0x4b
3.0e-12
```

- Kí tự, chuỗi kí tự

```
'a'
"Good morning!"
```

- Mã điều khiển: `\n` (xuống dòng), `\b` (backspace), `\r` (lùi về đầu dòng), `\t` (tab), `\v` (căn thẳng theo chiều dọc), `\f` (sang trang), `\a` (kêu bíp), `\'` (dấu nháy đơn), `\''` (dấu nháy kép),...

Để định nghĩa các hằng, ta có thể

- Sử dụng chỉ thị tiền xử lý `#define`, ví dụ:

```
root [0] #define PI 3.14159265
root [1] Float_t r = 2.0 // khai bao ban kinh r
root [2] float c = 2 * PI * r // tinh chu vi
```

  Khi khai báo hằng bằng `#define`, chương trình sẽ thay thế các tên hằng bằng giá trị của nó tại bất kì chỗ nào chúng xuất hiện. Các hằng số được định nghĩa theo cách này ược coi là các hằng số macro.

- Sử dụng từ khóa `const` và khai báo các hằng với một kiểu xác định như là làm với một biến, ví dụ:

```
root [0] const float pi = 3.14159265
root [1] const Int_t zip = 12440
```

2.2.3 Toán tử

Một số toán tử thông dụng trong ROOT

- Các toán tử logic: && (*and*), || (*or*), ! (*not*)
- Toán tử gán: =
- Các toán tử số học: +, −, *, /, % (chia lấy dư)
- Các toán tử gán phức hợp:

  ++ tăng lên 1 đơn vị
  - - giảm đi 1 đơn vị
  += tăng lên một lượng
  -= giảm đi một lượng
  *= nhân thêm một lượng
  /= chia cho một lượng

- Các toán tử so sánh:

== so sánh bằng
> so sánh lớn hơn
< so sánh nhỏ hơn
>= so sánh lớn hơn hoặc bằng
<= so sánh nhỏ hơn hoặc bằng
!= so sánh khác (không bằng)

- Toán tử điều kiện: ? , có cấu trúc như sau
  điều kiện ? kết quả 1 : kết quả 2
  (nếu điều kiện cho giá trị *true* thì trả về kết quả 1, còn nếu *false* thì trả về kết quả 2)

```
root [2] c = (a == b ? 2 : 3) // neu a = b thi c = 2, con khong thi c
   = 3
```

- Các toán tử thao tác bit: & (*logical and*), | (*logical or*), ∧ (*logical xor*), ∼ (*logical not*), >> (dịch bit sang phải), << (dịch bit sang trái)
- Toán tử chuyển đổi kiểu dữ liệu: *(kiểu dữ liệu)*

```
root [0] int i
root [1] float f = 3.14
root [2] i = (int) f; // hoac i = int (f)
```

- Kích thước dữ liệu (theo byte): `sizeof()`

```
root [0] int i
root [1] sizeof(i)
(const int)4
```

2.2.4 Các lệnh xuất nhập dữ liệu

Các lệnh xuất nhập dữ liệu căn bản gồm có

- `cin`: lệnh nhập dữ liệu (từ bàn phím)
- `cout`: lệnh xuất dữ liệu (lên màn hình)
- `cerr`/`clog`: lệnh xuất báo lỗi (lên màn hình), `cerr` không lưu trong buffer còn `clog` thì có

```
root [0] cout << "Xin chao!" << endl << "Toi ten la A." << endl;
Xin chao!
Toi ten la A.
```

- Toán tử << được gọi là toán tử chèn vì nó chèn dữ liệu đi sau nó vào dòng dữ liệu đứng trước.
- Để xuống dòng ta có thể sử dụng kí tự `\n` hoặc tham số `endl`

```
root [0] cout << "Xin chao " << ten << " ! \n";
```

2.2.5 Viết nhiều dòng lệnh một lúc

Trong ROOT, để có thể viết được nhiều dòng lệnh cùng một lúc, ta sử dụng cặp ngoặc nhọn `{}` để mở đầu và kết thúc các dòng lệnh.

Ví dụ:

```
root [0] {
end with '}', '@':abort > Int_t x = 0, y = 3;
end with '}', '@':abort > x = y - 2;
end with '}', '@':abort > cout << "x = " << x << endl;
end with '}', '@':abort > }
x = 1
```

2.2.6 Thư viện chuẩn

Thư viện chuẩn (*Standard Template Library – STL*) được xây dựng đầu tiên bởi Alexander Stepanov (1979) với mục đích phát triển phương pháp lập trình tổng quát. Thư viện đầu tiên được xây dựng với ngôn ngữ lập trình Ada bởi Stepanov và Musser năm 1987, tuy nhiên ngôn ngữ Ada lại không được phát triển mạnh nên họ đã chuyển sang ngôn ngữ C++. Với sự giúp đỡ của Meng Lee và Andrew Koenig, bộ thư viện chuẩn đã được hoàn thiện và đưa vào chuẩn ANSI/ISO C++ từ năm 1994.

Lưu ý: các file thư viện chuẩn của C++ đều không có phần mở rộng *.h*, ví dụ

```
#include<string>  // thu vien chuan C++
#include<string.h>  // thu vien chuan C
```

Các thư viện chuẩn C được đưa vào trong bộ thư viện chuẩn C++ STL với kí tự 'c' ở đầu

```
#include<cstring>  // thu vien chuan C++ cua cac ham C
```

2.2.7 Các hàm toán học

Trong ROOT sử dụng nhiều hàm toán học từ thư viện chuẩn của C++, để sử dụng các hàm này ta cần khai báo thư viện *cmath* (hoặc *math.h*)

`sqrt(x)`	hàm căn bậc hai
`pow(x,a)`	hàm luỹ thừa x^a
`exp(x)`	hàm e^x
`ln(x)`	hàm logarit tự nhiên

Ví dụ:

```
root [0] Int_t x=3;   // x = 3
root [1] cout << x++ << endl;    // x = 4
4
root [2] cout << sqrt(x) << endl;   // x = 4
2
root [3] cout << pow(x,3) << endl;   // x = 4
64
```

Danh sách các hàm toán học thông dụng trong C++ có thể được xem ở đây http://en.cppreference.com/w/cpp/numeric/math.

2.2.8 Các chỉ thị tiền xử lý

Bộ tiền xử lý (*preprocessor*) được ra đời do sự hạn chế của bộ nhớ máy tính trước đây, không thể chứa hết toàn bộ chương trình nguồn để dịch, do vậy một số ngôn ngữ thế hệ thứ 3 như C/C++ đã chia giai đoạn xử lý ra làm 2 phần: giai đoạn tiền xử lý sẽ xử lý các file nguồn, ghi nhận các tùy chọn (*option*) cho việc biên dịch; và giai đoạn xử lý sẽ thực hiện việc biên dịch mã nguồn.

Các chỉ thị tiền xử lý trong C/C++ đề bắt đầu bằng kí tự '#', các chỉ thị này không được xem như là dòng lệnh nên không có dấu ';' ở cuối dòng. Trong phần trên ta đã làm quen với các chỉ thị `#include` và `#define`, phần này sẽ trình bày chi tiết hơn về một số chỉ thị thông dụng

- Các chỉ thị định nghĩa (`#define` và `#undef`) dùng để định nghĩa hay hủy bỏ định nghĩa các macro, ví dụ như

```
#define SIZE 100
int table1[SIZE];  // tuong duong voi khai bao: int table1[100]
#undef SIZE  // bo dinh nghia macro SIZE
#define SIZE 200  // dinh nghia lai macro SIZE
int table2[SIZE];  // table2[200]
```

  Ta cũng có thể định nghĩa một macro hàm có tham số

```
#define getmax(a,b) a>b?a:b
```

- Các chỉ thị điều kiện (`#ifdef`, `#ifndef`, `#if`, `#endif`, `#else`, `#elif`) cho phép bao gồm hoặc loại bỏ một phần các dòng lệnh nếu những điều kiện được thỏa

```
#ifdef SIZE
int table[SIZE];  // neu macro SIZE da duoc dinh nghia thi khai bao
   table[SIZE]
#endif
```

```
#ifndef SIZE
#define SIZE 100  // neu macro SIZE chua duoc dinh nghia thi dinh
   nghia no
#endif
```

2.2.9 Không gian tên

Không gian tên (*namespace*) cho phép chúng ta gộp một nhóm các lớp, các đối tượng toàn cục và các hàm dưới một cái tên, hoặc được dùng để phân chia phạm vi hoạt động của các đối tượng. Cú pháp của nó như sau

namespace <không gian tên> {
... khai báo ...
}

Giả sử như ta có hai hàm có cùng tên `func()` trong một chương trình, để giúp cho trình biên dịch phân biệt được hàm nào được gọi ta sử dụng không gian tên như sau

```
// Khong gian ten dau tien
namespace first{
  void func(){
    cout << "Inside first space" << endl;
  }
}

// Khong gian ten thu hai
namespace second{
  void func(){
    cout << "Inside second space" << endl;
  }
}
```

Để gọi hàm với không gian tên tương ứng ta sử dụng kí hiệu '::'

```
int main ()
{
  // Goi ham voi khong gian ten thu nhat
  first::func();
  // Goi ham voi khong gian ten thu hai
  second::func();
  return 0;
}
```

Để có thể truy xuất trực tiếp mà không cần đặt thông qua không gian tên, ta sử dụng từ khóa `using`

```
int main ()
{
  using namespace first;
  // Goi ham voi khong gian ten thu nhat
  func();
  // Goi ham voi khong gian ten thu hai
  second::func();
  return 0;
}
```

Chúng ta cũng có thể định nghĩa lại các không gian tên đã khai báo, chẳng hạn như

```
namespace dau_tien = first;
```

Tất cả định nghĩa của các lớp, đối tượng và hàm của thư viện chuẩn (STL) đều được định nghĩa trong không gian tên `std`, chẳng hạn như các lệnh `cin` và `cout` mà ta nói ở trên, nếu viết đầy đủ thì sẽ là

```
std::cout << "Nhap ten cua ban: ";
std::cin >> ten;
std::cout << "Xin chao " << ten << " !" << std::endl;
```

hoặc ta cũng có thể lược đi bằng cách sử dụng từ khóa `using`

```
using namespace std;
cout << "Nhap ten cua ban: ";
cin >> ten;
cout << "Xin chao " << ten << " !" << endl;
```

2.2.10 Các cấu trúc điều khiển

Khi xây dựng chương trình, đôi lúc ta cần thực hiện một nhóm các lệnh tương ứng với một điều kiện nào đó hay là lặp lại nhóm lệnh một số lần nhất định. Trong trường hợp đó, ta sẽ sử dụng cấu trúc điều khiển. Nhóm các lệnh sẽ được gộp lại với nhau bằng cặp ngoặc nhọn `{ }`

Cấu trúc điều kiện *if...else...*

Cú pháp:

if(điều kiện) { các lệnh thực thi }

hoặc

if(điều kiện) { các lệnh thực thi } else { các lệnh thực thi }

Ví dụ: *kiểm tra xem x la số chẵn hay số lẻ*

```
int x=5;
if(x%2 == 0) {
  std::cout << "So chan" << std::endl;
} else {
  std::cout << "So le" << std::endl;
}
```

Cấu trúc lặp *for*

Cú pháp:

for (giá trị ban đầu; điều kiện ; bước nhảy) { các lệnh thực thi }

Ví dụ: *in ra các số tăng dần từ 0 đến 4*

```
for(int i=0; i < 5; i++) {
  std::cout << i << std::endl;
}
```

Cấu trúc lặp *while*

Cú pháp

while (điều kiện) { các lệnh thực thi }

hoặc

do { các lệnh thực thi } while (điều kiện)

Ví dụ: *in ra các số giảm dần từ 5 đến 0*

```
int x=5;
do {
  std::cout << x-- << std::endl;
} while(x>0)
```

Các lệnh cho vòng lặp

- Lệnh `continue` được dùng để kết thúc lần lặp hiện tại và chuyển sang lần lặp tiếp theo
- Lệnh `break` được dùng để kết thúc hoàn toàn vòng lặp.

Ví dụ: *các số được in ra sẽ là 0, 1, 2, 4 (không có 3)*

```
for(int i=0; i<5; i++) {
  if(i==3) continue;
  std::cout << i << std::endl;
}
```

Ví dụ: *các số được in ra sẽ là 0, 1, 2 (không có từ 3 trở đi)*

```
for(int i=0; i<5; i++) {
  if(i==3) break;
  std::cout << i << std::endl;
}
```

Thông qua 2 ví dụ ở trên, chúng ta có thể thấy được là khi giá trị i = 3, đối với lệnh `continue` chương trình sẽ bỏ qua không xuất ra giá trị của i và chuyển tới lần lặp tiếp theo. Trong khi đó, đối với lệnh `break`, chương trình sẽ thoát ra khỏi vòng lặp, do đó các giá trị i từ 3 trở đi sẽ không được xuất ra màn hình.

Cấu trúc lựa chọn *switch*

Cú pháp:

switch (biểu thức) {
case <giá trị 1>:
các lệnh;
break;
case <giá trị 2>:
các lệnh;
break;
...
default:
các lệnh;
}

Ví dụ: *kiểm tra xem x la số chẵn hay số lẻ*

```
int x=5;
switch (x%2) {
  case 0:
    std::cout << "So chan" << std::endl;
    break;
  case 1:
    std::cout << "So le" << std::endl;
    break;
  default:
    std::cout << "Undefined" << std::endl;
}
```

2.2.11 Hàm

Hàm (*function*) là một khối lệnh dùng để thực thi một tác vụ nào đó và trả kết quả về khi được gọi. Cú pháp khai báo một hàm như sau

<kiểu dữ liệu trả về> <tên hàm> (<đối số 1>, <đối số 2>,...)
{ <nội dung của hàm> }

Dữ liệu được trả về thông qua lệnh `return`. Nếu không trả về dữ liệu thì khai báo kiểu dữ liệu trả về là `void`.

Ví dụ:

```
int tong(int x, int y) {
  return (x+y);
}

void main () {
  int x=5, y=6;
  std::cout << "Tong cua hai so x va y la: " << tong(x,y) << std::endl;
}
```

Ta có thể khai báo hai hay nhiều hàm có cùng tên nếu chúng có số lượng đối số, kiểu dữ liệu của đối số hay kiểu dữ liệu trả về khác nhau. Trình biên dịch sẽ lựa chọn gọi hàm nào bằng cách phân tích kiểu đối số khi hàm được gọi.

```
int tong(int x, int y) {
  return (x+y);
}
```

```
void tong(float x, float y) {
  std::cout << "Tong cua hai so la" << x+y << std::endl;
}
```

Trên nguyên tắc, một hàm phải được khai báo trước khi nó được gọi, điều này cũng đồng nghĩa với việc hàm `main()` phải được đặt cuối chương trình để có thể gọi được tất cả các hàm đã khai báo. Tuy nhiên, để tránh việc phải viết tất cả mã chương trình trước khi chúng có thể được dùng trong hàm `main()` hay bất kì một hàm nào khác, ta có thể sử dụng cách khai báo nguyên mẫu (*prototype*) cho hàm. Cách khai báo này đủ để cho trình biên dịch có thể biết các tham số và kiểu dữ liệu trả về của hàm. Khai báo prototype không bao gồm phần thân hàm và được kết thúc bằng dấu ';'.

Ví dụ:

```
int tong(int x, int y);

void main () {
  int x=5, y=6;
  std::cout << "Tong cua hai so x va y la: " << tong(x,y) << std::endl;
}

int tong(int x, int y) {
  return (x+y);
}
```

Đối số của hàm

Đối số (*argument*) là các tham số mà ta sẽ truyền cho hàm khi được gọi. Khi định nghĩa một hàm chúng ta có thể chỉ định những giá trị mặc định sẽ được truyền cho các đối số trong trường hợp chúng bị bỏ qua khi hàm được gọi.

```
int tong(int x=0, int y=1)
```

Khi gọi hàm, ta không nhất thiết phải khai báo đầy đủ tất cả các đối số. Ví dụ như hàm `tong(x,y)` có thể được gọi chỉ với 1 đối số `tong(2)`, khi đó giá trị trả về sẽ là 3 (x = 2 được truyền cho hàm và y = 1 được thiết lập mặc định).

Ngoài ra trong phần lớn các trường hợp gọi hàm, các đối số truyền cho hàm đều là các giá trị (chứ không phải là bản thân các biến), cách truyền này được gọi là cách truyền đối số theo dạng tham trị (*by value*). Do đó khi thay đổi giá trị của các đối số này bên trong hàm thì các biến đó vẫn không bị thay đổi. Để thay đổi được giá trị của biến bên ngoài hàm ta cần phải truyền bản thân biến đó. Việc này có thể được thực hiện bằng cách truyền đối số dưới dạng tham chiếu (*by reference*), khi đó bất kì sự thay đổi nào của đối số đó bên trong hàm sẽ ảnh hưởng trực tiếp lên biến. Để khai báo đối số theo dạng tham chiếu, ta sử dụng kí tự '&' đặt trước tên biến được truyền.

```
void hoandoi(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}

int main()
{
  int x = 1, y = 4;
  hoandoi(x,y);
  std::cout << "x = " << x << ", y = " << y << std::endl;
```

```
  return 0;
}
```

Ngoài ra còn một cách truyền đối số khác đó là truyền dưới dạng tham biến (*by variable*) hay còn gọi là truyền bằng con trỏ (*by pointer*). Cách truyền này được thực hiện bằng cách sử dụng kí tự '*' đặt trước tên biến (sẽ bàn kĩ hơn trong phần 2.2.13).

```
int hoandoi(int *x, int *y) {
  int tmp = *x;
  *x = *y;
  *y = tmp;
}
```

Hàm đệ quy

Hàm đệ quy (*recursive function*) là hàm gọi chính nó trong quá trình thực thi. Các hàm này thích hợp cho một số mục đích cụ thể chẳng hạn như sắp xếp dãy số, tính giai thừa của một số,...

Ví dụ:

```
int factorial (int n) {
  if (n > 1)
    return (n * factorial (n-1));
  else
    return (1);
}
int main () {
  int n;
  std::cout << "Nhap mot so nguyen: ";
  std::cin >> n;
  std::cout << "!" << n << " = " << factorial (n) << std::endl;
  return 0;
}
```

Hàm nội tuyến

Hàm nội tuyến (*inline function*) là các hàm mà trình biên dịch sẽ thay thế lời gọi hàm bằng mã lệnh của hàm khi chương trình được dịch, hay nói một cách khác là trình biên dịch sẽ chèn phần thân hàm vào vị trí mà nó được sử dụng.

Thông thường một hàm có được là nội tuyến hay không sẽ do trình biên dịch quyết định. Tuy nhiên khi đặt từ khóa `inline` trước khai báo của một hàm, trình biên dịch sẽ xem hàm này như là nội tuyến. Cách này thường được dùng cho các hàm thực thi thường xuyên nhằm loại bỏ thời gian quá dụng (*overhead*) khi gọi hàm, tuy nhiên đôi khi cũng gây ra vấn đề khi biên dịch nếu số lượng hàm nội tuyến quá lớn.

```
inline int tong(int x=0, int y=1)
```

Nạp chồng hàm

Trong C++ ta có thể xây dựng các hàm có cùng tên nhưng khác nhau về nội dung hoặc chức năng thực hiện nhiệm vụ, cách thức này được gọi là nạp chồng hay tái định nghĩa (*overloading*).

Chúng ta có thể định nghĩa nhiều hàm có cùng tên với nhau, nhưng phải có các đối số khác nhau (về kiểu dữ liệu hay số lượng đối số). Khi chúng ta gọi hàm, trình biên dịch sẽ xác định trong số

những hàm đó, hàm nào là phù hợp nhất bằng cách so sánh các đối số được truyền cho hàm (toán tử) với các kiểu đối số được khai báo trong phần định nghĩa hàm[5].

```
int tong(int a, int b) {
  return (a+b);
}

int tong(int a, int b, int c) {
  return (a+b+c);
}

float tong(float a, float b) {
  return (a+b);
}

int main()
{
  int a=2, b=3, c=-1;
  float d=0.5, e=-1.2;
  std::cout << "Tong cua hai so nguyen: " << tong(a,b) << std::endl;
  std::cout << "Tong cua ba so nguyen: " << tong(a,b,c) << std::endl;
  std::cout << "Tong cua hai so thuc: " << tong(d,e) << std::endl;
  return 0;
}
```

2.2.12 Mảng và chuỗi

Mảng

Mảng (*array*) là một dãy các phần tử có cùng kiểu được đặt liên tiếp trong bộ nhớ và có thể truy xuất đến từng phần tử bằng cách thêm một chỉ số vào sau tên của mảng. Cách thức khai báo mảng như sau

<kiểu biến> <tên mảng> [kích thước]

hoặc

*<kiểu biến> *<tên_mảng>*

Ví dụ:

```
// Khai bao hai ma tran (mang) A & B co kich thuoc 2x2
int A[2][2] = {1,3,6,4}, B[2][2] = {0,5,2,1};
```

Để gán và truy xuất giá trị của các phần tử trong mảng, ta chỉ cần gọi chỉ số của phần tử đó

```
int A[2][2];
A[0][0] = 1;
std::cout << A[0][0] << std::endl;
A[A[0][0]][2] = 3;  // tuong duong khai bao A[1][2] = 3, vi A[0][0] = 1
```

Lưu ý: trong C++ chỉ số của mảng bắt đầu từ 0.

Trong trường hợp muốn truyền tham số là mảng cho hàm thì chúng ta cần phải chỉ định trong phần đối số kiểu dữ liệu cơ bản của mảng, tên mảng và cặp ngoặc vuông trống khi khai báo đối số của hàm.

[5]Còn một khái niệm tái định nghĩa (hay nạp chồng) hàm khác nữa gọi là *overriding*, điểm khác biệt giữa hai khái niệm này là *overloading* sử dụng cùng một tên hàm nhưng khác nhau các đối số, còn *overriding* có cùng tên hàm lẫn đối số tuy nhiên nội dung hàm lại khác. Do đó, *overriding* thường được sử dụng trong việc tái định nghĩa các phương thức của lớp dẫn xuất so với lớp cơ sở (xem phần 2.3)

```
void function (int arg[]) // mang 1 chieu
void function (int arg[][3][4]) // mang nhieu chieu
```

Chuỗi kí tự

Trong ngôn ngữ C truyền thống không có kiểu dữ liệu cơ bản để lưu các chuỗi kí tự, do đó để người ta thường sử dụng mảng có kiểu `char`.

Ví dụ: *khai báo chuỗi kí tự tên có độ dài cực đại là 20 kí tự*

```
char ten[20];
```

Nội dung của chuỗi kí tự luôn được kết thúc với kí tự *null* (`\0`).

Để khởi tạo nội dung cho chuỗi kí tự, ta có thể thực hiện theo cách sau

```
char ten[] = {'P', 'h', 'u', 'o', 'n', 'g', '\0'};
```

hoặc

```
char ten[] = "Phuong";
```

Lưu ý rằng việc gán nhiều hằng cho các phần tử trong mảng chỉ hợp lệ khi khởi tạo mảng. Thao tác gán chỉ có thể được thực hiện với từng phần tử một trong khi thực thi chương trình. Để có thể gán giá trị cho một chuỗi kí tự, chúng ta có thể sử dụng hàm `strcpy()`

```
strcpy(ten, "Phuong");
```

Khi ta có chuỗi kí tự là một số (vd: "1234"), ta có thể chuyển đổi chuỗi này sang số bằng cách sử dụng các hàm có trong thư viện *cstdlib* (*stdlib.h*)

- `atoi`: chuyển chuỗi thành kiểu `int`
- `atol`: chuyển chuỗi thành kiểu `long`
- `atof`: chuyển chuỗi thành kiểu `float`

Ví dụ:

```
char mybuffer [100];
float price;
std::cout << "Enter price: ";
std::cin.getline(mybuffer,100);
price = std::atof(mybuffer);
```

Một số hàm khác trong thư viện *cstring* (*string.h*) thường hay được sử dụng để thao tác trên chuỗi bên cạnh hàm `strcpy()` gồm có

- `strcat()`: gắn thêm chuỗi *src* vào phía cuối của *dest* và trả về *dest*
  char strcat (char* dest, const char* src)*
- `strcmp()`: so sánh hai chuỗi *string1* và *string2*, trả về 0 nếu hai chuỗi là bằng nhau
  int strcmp (const char string1, const char* string2)*
- `strlen()`: Trả về độ dài của chuỗi
  size_t strlen (const char string)*

Ngoài ra, thư viện chuẩn của C++ có cung cấp lớp `string` để định nghĩa các chuỗi thông qua thư viện *string* (lưu ý rằng thư viện này khác với thư viện *string.h*)

```
std::string str1 = "Hello";
std::string str2 = "World";
```

Cách chuyển đổi giữa `string` và `char*`

```
std::string str = "Hello";
const char *c = str.c_str();  // chuyen tu string thanh char*
std::string str1(c);  // chuyen tu char* thanh string
```

Một số thao tác trên chuỗi `string`

- Nối chuỗi

```
std::string str3 = str1 + str2;  // cong hai chuoi
std::cout << "str1 + str2 : " << str3 << std::endl;
```

- Độ dài chuỗi

```
int len = str3.size();  // kich thuoc chuoi
std::cout << "Do dai duoi str3 :  " << len << std::endl;
```

- So sánh chuỗi

```
std::string str1 ("green apple");
std::string str2 ("red apple");
if(str1.compare(str2)!=0)
  std::cout<<"it is not the same"<< std::endl;
```

2.2.13 Con trỏ

Như chúng ta đã biết, bộ nhớ máy tính có thể xem như là một dãy gồm các ô nhớ (có kích thước 1 byte), mỗi ô có một địa chỉ xác định. Các biến chính là các ô nhớ mà chúng ta có thể truy xuất dưới các tên. Khi chúng ta khai báo một biến thì nó phải được lưu trữ trong một vị trí cụ thể trong bộ nhớ, vị trí của biến được lưu trữ sẽ do trình biên dịch và hệ điều hành quyết định. Để có thể biết được chính xác vị trí (hay địa chỉ) của biến đó ở đầu trong bộ nhớ, chúng ta cần sử dụng một công cụ hỗ trợ, đó chính là con trỏ.

Con trỏ (*pointer*) là một biến lưu trữ địa chỉ của một biến hoặc một vùng biến khác. Để lấy địa chỉ của một biến, người ta sử dụng toán tử &. Cách thức khai báo con trỏ như sau

*<kiểu dữ liệu> *<tên con trỏ>*

Trong đây chúng ta sẽ làm quen với hai toán tử

- Toán tử lấy địa chỉ (&): được dùng như là một tiền tố của biến và có thể được hiểu là "địa chỉ của", vd: `&a` có thể được hiểu là "địa chỉ của biến `a`".
- Toán tử tham chiếu (*): truy xuất trực tiếp đến giá trị được lưu trữ trong biến được trỏ bởi nó và có thể được hiểu là "giá trị được trỏ bởi", vd: `*pointer` có thể hiểu là "giá trị được trỏ bởi `pointer`".

Ví dụ: *thay đổi giá trị của biến a từ 3 thành 5*

```
int a=3;
int *pointer;  // Khai bao bien con tro pointer
pointer = &a;  // pointer chua dia chi cua bien a
*pointer = 5;  // gan gia tri cua bien ung voi dia chi ma pointer tro toi
   la 5 (a = 5)
```

Ta cũng có thể khởi tạo con trỏ ngay khi định nghĩa

```
int a=3;
int *pointer = &a;
```

Việc thực hiện các phép tính số học với con trỏ hơi khác so với các kiểu dữ liệu số khác. Chúng ta chỉ được sử dụng phép cộng và trừ, và kết quả của phép tính phụ thuộc vào kích thước của kiểu dữ liệu mà biến con trỏ trỏ tới. Điều này có nghĩa là kích thước tính bằng byte của kiểu dữ liệu nó trỏ tới sẽ được cộng (hoặc trừ) thêm vào biến con trỏ.

Ví dụ:

```
int *p1 = &a1;  // gia su gia tri cua p1 la 1000 (dia chi cua a1)
long *p2 = &a2;  // gia su gia tri cua p2 la 2000 (dia chi cua a2)
p1++;  // gia tri cua p1 luc nay la 1001 (kieu int co kich thuoc 1 byte)
p2++; // gia tri cua p2 luc nay la 2002 (kieu long co kich thuoc 2 byte)
```

Con trỏ và mảng

Trong thực tế, tên của một mảng tương đương với địa chỉ phần tử đầu tiên của nó, giống như một con trỏ tương đương với địa chỉ của phần tử đầu tiên mà nó trỏ tới.

Ví dụ: *khai báo một mảng chứa biến kiểu số nguyên, địa chỉ của con trỏ trỏ tới cũng chính là địa chỉ của phần tử đầu tiên trong mảng*

```
int *p;
```

Ví dụ: *gán mảng*

```
int a[10];
int *p;
p = a;  // lenh gan nguoc lai a = p la sai vi a[10] cho con tro hang
a[5] = 0;  // phan tu thu 5 cua mang a co gia tri la 0
*(a+5) = 0;  // tuong duong a[5] = 0 (a cung la con tro)
```

2.2.14 Dữ liệu có cấu trúc

Dữ liệu tự định nghĩa

C++ cho phép chúng ta định nghĩa các kiểu dữ liệu của riêng mình dựa trên các kiểu dữ liệu đã có. Để có thể làm việc đó chúng ta sẽ sử dụng từ khoá `typedef`, dạng thức như sau

typedef <kiểu dữ liệu đã có> <kiểu dữ liệu mới>

Ví dụ:

```
typedef char* chuoi;
chuoi loichao;
```

Dữ liệu có cấu trúc

Một cấu trúc dữ liệu là một tập hợp của những kiểu dữ liệu khác nhau được gộp lại với một cái tên duy nhất. Để định nghĩa các kiểu dữ liệu có cấu trúc ta có thể sử dụng các từ khóa `struct`, `union` hay `enum`,...

Struct: dạng thức của nó như sau

struct <tên kiểu cấu trúc> {
<kiểu dữ liệu> phần tử 1;
<kiểu dữ liệu> phần tử 2;
. . .
} <tên biến>;

Ví dụ:

```
struct sinhvien {  // dinh nghia cau truc
  char ten[100];
  int tuoi;
}
sinhvien A;  // khai bao bien
sinhvien B, C;
```

Để truy xuất các phần tử của biến, ta sử dụng dấu '.', ví dụ:

```
A.tuoi = 20;
```

Ta cũng có thể định nghĩa các cấu trúc lồng nhau, chẳng hạn như

```
struct diem {
  float toan;
  float anhvan;
}
struct sinhvien {
  char ten[100];
  int tuoi;
  diem ketqua;  // bien ketqua thuoc kieu "diem"
}
sinhvien A;
```

Sau khai báo trên chúng ta có thể truy xuất

```
A.ketqua.toan
A.ketqua.anhvan
```

Ngoài ra, chúng ta cũng có thể sử dụng con trỏ để khai báo biến có cấu trúc

```
sinhvien *pA = &A;
```

Để truy cập phần tử của biến con trỏ ta sử dụng dấu `->`

```
pA->ketqua.toan
```

Union: có cấu trúc tương tự `struct`

union <tên kiểu cấu trúc> {
<kiểu dữ liệu> phần tử 1;
<kiểu dữ liệu> phần tử 2;
. . .
} <tên biến>;

Tuy nhiên điểm khác biệt của `union` so với `struct` là tất cả các phần tử của union đều chiếm cùng một chỗ trong bộ nhớ, kích thước của nó là kích thước của phần tử lớn nhất. Do đó bất kì một sự thay đổi nào đối với một phần tử sẽ ảnh hưởng tới tất cả các phần tử còn lại.

Enum: kiểu dữ liệu liệt kê dùng để tạo ra các kiểu dữ liệu chứa một cái gì đó hơi đặc biệt một chút (không phải kiểu số hay kiểu kí tự), cấu trúc của nó như sau

enum <tên kiểu cấu trúc> {
giá trị 1,
giá trị 2,
. . .
} <tên biến>;

Ví dụ:

```
enum colors {black, blue, green, cyan, red, purple, yellow, white};
colors mycolor;
mycolor = blue;
if (mycolor == green) mycolor = red;
```

Trên thực tế kiểu dữ liệu liệt kê được dịch là một số nguyên, giá trị tương ứng với phần tử đầu tiên là 0 và các giá trị tiếp theo cứ thế tăng lên 1. Ví dụ trong kiểu dữ liệu mà chúng ta định nghĩa ở trên, black tương đương với 0 , blue tương đương với 1 , green tương đương với 2 và cứ tiếp tục như thế. Ngoài ra, chúng ta có thể chỉ định giá trị tương ứng bằng cách gán

```
enum colors {black=2, blue, green, cyan, red, purple, yellow, white};
```

2.2.15 File

File là một trong những hình thức lưu trữ dữ liệu phổ biến, có hai loại file chính

- File văn bản (*text file*): là loại file chỉ lưu trữ thuần túy văn bản, các kí tự được biểu diễn bằng mã ASCII của nó, và người dùng có thể dễ dàng đọc được nội dung của file này.
- File nhị phân (*binary file*): là loại file chứa các đoạn mã nhị phân, người dùng không thể đọc được nội dung các file này.

Có hai cách để thực hiện cách thao tác trên file: sử dụng các hàm trong thư viện C *cstdio* (*stdio.h*) hay các hàm trong thư viện dòng xuất nhập *iostream*

Sử dụng thư viện cstdio

Thư viện *stdio* bao gồm các hàm

Tên hàm	Chức năng
Các hàm chung	
`fopen`	Mở file
`fclose(all)`	Đóng (tất cả các) file
`fflush(all)`	Làm sạch vùng đệm của (tất cả các) file đang mở
`remove`	Xóa file
`feof`	Kiểm tra xem tới cuối file chưa
Xử lý file văn bản	
`fprintf`	Ghi ra file
`fscanf`	Đọc từ file
`fputc/fgetc`	Ghi (đọc) 1 kí tự ra (từ) file
`fputs/fgets`	Ghi (đọc) 1 chuỗi ra (từ) file
Xử lý file nhị phân	
`fwrite`	Ghi ra file
`fread`	Đọc từ file
`fseek`	Di chuyển con trỏ tới vị trí trong file
`ftell`	Cho biết vị trí hiện tại của con trỏ trong file

Ví dụ:

```
FILE *f;  // Khai bao con tro file
char name [100];
f = fopen ("myfile.txt","w");  // Mo mot file co ten la file.txt de ghi du
    lieu
puts ("Nhap ten cua ban: ");
gets (name);
fprintf (f, "%-10.10s \n", name);  // Ghi ra file.txt
fclose (f);  // Dong file.txt
```

Sử dụng dòng xuất nhập

Các phương thức dòng xuất nhập cho file gồm có

- *ifstream*: lớp gồm các phương thức đọc file
- *ofstream*: lớp gồm các phương thức ghi file
- *fstream*: lớp gồm các phương thức đọc/ghi file

Tên phương thức	Chức năng
Các hàm chung	
open	Mở file
close	Đóng (tất cả các) file
is_open	Kiểm tra xem file đã mở hay chưa
flush	Làm sạch vùng đệm của file đang mở
good	Kiểm tra xem tình trạng file có tốt để xử lý hay không
eof	Kiểm tra xem tới cuối file chưa
Xử lý file văn bản	
<<	Ghi ra file
>>	Đọc từ file
getline	Đọc 1 dòng dữ liệu từ file
get	Đọc 1 kí tự hoặc chuỗi từ file
put	Ghi 1 kí tự ra file
Xử lý file nhị phân	
fwrite	Ghi ra file
fread	Đọc từ file
seekg	Di chuyển con trỏ tới vị trí trong file
tellp	Cho biết vị trí hiện tại của con trỏ trong file

Ví dụ:

```
char name [100];
std::ofstream f("file.txt");  // Mo mot file co ten la file.txt de ghi du
    lieu
std::cout << "Nhap ten cua ban: ";
std::cin >> name;
f << name << std::endl;  // Ghi ra file
f.close();  // Dong file.txt
```

2.3 Lập trình hướng đối tượng

Lập trình hướng đối tượng (*object-oriented programming* – OOP) là kĩ thuật lập trình hỗ trợ công nghệ đối tượng. Kĩ thuật lập trình này giúp tăng năng suất và giảm nhẹ các thao tác viết mã cho người lập trình. Nó cũng giúp cho người lập trình tạo ra các ứng dụng mà các yếu tố bên ngoài

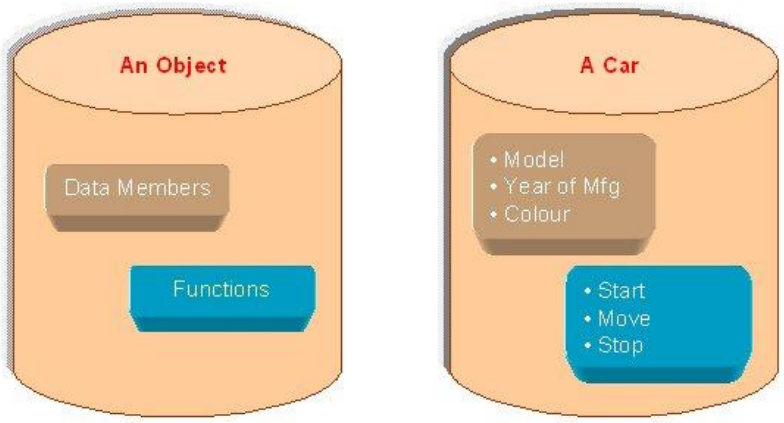
có thể tương tác với các chương trình đó giống như là tương tác với các đối tượng vật lý, điều này giúp đơn giản hóa độ phức tạp khi bảo trì cũng như mở rộng phần mềm bằng cách cho phép lập trình viên tập trung vào các đối tượng phần mềm ở bậc cao hơn.

2.3.1 Đối tượng

Đối tượng (*object*) là sự kết hợp giữa mã và dữ liệu hình thành nên một đơn vị duy nhất, đơn vị này tương đương với một chương trình con. Các đối tượng bao gồm hai thành phần chính:

- Các phương thức (*method*): là phương tiện để sử dụng một đối tượng, các phương thức thường là các hàm.
- Các thuộc tính (*attribute*): dùng để mô tả các tính chất của đối tượng, thường là các biến, tham số hay hằng nội tại (các dữ liệu nội tại).

Mỗi phương thức hay mỗi dữ liệu nội tại cùng với các tính chất được định nghĩa (bởi người lập trình) được xem là một đặc tính của đối tượng (xem Hình 2.1). Trong thực tế, các đối tượng thường được trừu tượng hóa qua việc định nghĩa các **lớp** (*class*).



Hình 2.1: Minh họa đối tượng; một đối tượng, chẳng hạn như một chiếc xe hơi, sẽ có các thông tin (*thuộc tính*) như mẫu xe, năm sản xuất, màu sắc và các hành động (*phương thức*) như khởi động, chạy và dừng.

Mỗi đối tượng có một tên riêng biệt và tất cả các tham chiếu đến đối tượng đó được tiến hành qua tên của nó. Như vậy, mỗi đối tượng có khả năng nhận vào các thông báo, xử lý dữ liệu (bên trong của nó), và gửi ra hay trả lời đến các đối tượng khác hay đến môi trường.

Một số tính chất chính của lập trình hướng đối tượng:

- Tính trừu tượng (*abstraction*): khả năng của chương trình bỏ qua hay không chú ý đến một số khía cạnh của thông tin mà nó đang trực tiếp làm việc lên. Mỗi đối tượng có thể hoàn tất các công việc một cách nội bộ và liên lạc với các đối tượng khác mà không cần cho biết làm cách nào đối tượng tiến hành được các thao tác. Tính trừu tượng còn thể hiện qua việc một đối tượng ban đầu có thể có một số đặc điểm chung cho nhiều đối tượng khác như là sự mở rộng của nó nhưng bản thân đối tượng ban đầu này có thể không có các biện pháp thi hành. Tính trừu tượng này thường được xác định trong khái niệm gọi là lớp trừu tượng (*abstract class*) hay lớp cơ sở trừu tượng (*abstract base class*).
- Tính đóng gói (*encapsulation*): không cho phép người sử dụng các đối tượng thay đổi trạng thái nội tại của một đối tượng, chỉ có các phương thức nội tại của đối tượng cho phép thay đổi trạng thái của nó. Việc cho phép môi trường bên ngoài tác động lên các dữ liệu nội tại của một đối tượng theo cách nào là hoàn toàn tùy thuộc vào người lập trình. Đây là tính chất đảm bảo sự toàn vẹn của đối tượng.

- Tính kế thừa (*inheritance*): cho phép một đối tượng có thể có sẵn các đặc tính mà đối tượng khác đã có thông qua kế thừa. Điều này cho phép các đối tượng chia sẻ hay mở rộng các đặc tính sẵn có mà không cần phải tiến hành định nghĩa lại. Nếu ta có một lớp B kế thừa từ lớp A thì A được gọi là lớp cơ sở (*base class*) và B được gọi là lớp dẫn xuất (*derived class*).
- Tính đa hình (*polymorphism*): thể hiện thông qua việc gửi các thông điệp (*message*), các phương thức dùng trả lời cho một thông điệp sẽ tùy theo đối tượng mà thông điệp đó được gửi tới sẽ có các phản ứng khác nhau. Người lập trình có thể định nghĩa một đặc tính (chẳng hạn như một phương thức) cho một loạt các đối tượng gần nhau, khi thi hành chương trình sẽ tự động kiểm tra xem đối tượng là thuộc kiểu lớp nào sau đó sẽ gọi phương thức tương ứng với lớp đó.

2.3.2 Lớp

Lớp (*class*) là một kiểu dữ liệu do người dùng tự định nghĩa, có thể bao gồm trong nó các dữ liệu và hàm. Cú pháp khai báo một lớp như sau

class <tên của lớp> {
<kiểu thuộc tính>:
dữ liệu 1;
dữ liệu 2;
hàm 1;
....
} <biến đối tượng>;

Các thành viên (*member*) của lớp được chia làm 2 loại: các thành phần chứa dữ liệu của lớp được gọi là các thuộc tính (*attribute*) của lớp, các thành phần này thường là các biến; các thành phần chỉ hành động của lớp được gọi là các phương thức (*method*) của lớp, các thành phần này thường là các hàm.

Thuộc tính của dữ liệu và hàm có các dạng: `private` (chỉ được truy xuất bởi các thành viên trong cùng lớp), `protected` (có thể được truy xuất bởi các thành viên trong cùng lớp hoặc từ các lớp kế thừa), hoặc `public` (có thể được truy xuất từ bên ngoài lớp).

Ví dụ:

```
class Sinhvien {       // Tao mot lop co ten la Sinhvien
private:
  char ten[100];
  int tuoi;
  float diemthi;
public:
  Sinhvien()  {};     // Constructor
  ~Sinhvien()  {};    // Destructor
  void ketqua() {
    if(diemthi >= 5)
      cout << "Dau" << endl;
    else
      cout << "Rot" << endl;
    }
};
```

Hàm khởi tạo (*constructor*) của một lớp sẽ được gọi tự động khi khai báo một đối tượng mới thuộc lớp đó, ngược lại hàm hủy (*destructor*) sẽ được gọi tự động khi đối tượng đó kết thúc hoạt động hoặc được giải phóng khỏi bộ nhớ. Một số đặc tính của hàm khởi tạo và hàm hủy

- Hàm khởi tạo phải có tên trùng với tên của lớp; hàm hủy có tên bắt đầu bằng $\sim$ và theo sau là tên của lớp tương ứng.

- Cả hai hàm không có giá trị trả về và đều có thuộc tính `public`.
- Một lớp có thể có nhiều hàm khởi tạo nhưng chỉ có duy nhất một hàm hủy.

Nếu định nghĩa thuộc tính hoặc phương thức bên ngoài phạm vi định nghĩa lớp, ta phải dùng chỉ thị phạm vi được thể hiện qua dấu `::`

```
class Sinhvien {       // Tao mot lop co ten la Sinhvien
private:
  char ten[100];
  int tuoi;
  float diemthi;
public:
  Sinhvien()  {};     // Constructor
  ~Sinhvien()  {};    // Destructor
  void ketqua();
};

void Sinhvien::ketqua() {
  if(diemthi >= 5)
    std::cout << "Dau" << std::endl;
  else
    std::cout << "Rot" << std::endl;
}
```

Muốn truy xuất các thành viên của đối tượng thuộc một lớp nào đó, ta có thể sử dụng toán tử dấu `.` đặt ngay sau đối tượng đó

```
void main()
{
  Sinhvien A;          // Doi tuong A thuoc lop Sinhvien
  A.ten = "Nguyen Van A";
  A.diemthi = 6.5;
  A.ketqua();
}
```

Chú ý: cần phân biệt giữa toán tử dấu chấm (`.`) dùng để truy cập các thành phần của đối tượng và toán tử mũi tên (`->`) dùng để truy cập các thành phần của con trỏ tới đối tượng. Ta có mối quan hệ như sau

```
A->ten = (*A).ten
```

2.3.3 Lớp dẫn xuất

Lớp dẫn xuất (*derived class*) hay còn gọi là lớp con (*subclass*) là một lớp được kế thừa từ một lớp khác, cú pháp khai báo lớp này như sau

class <tên lớp dẫn xuất>: <từ khóa dẫn xuất> <tên lớp cơ sở> {
....
};

Trong đó từ khóa dẫn xuất quy định tính chất kế thừa của lớp dẫn xuất từ lớp cơ sở (*base class*)

- **private**: các thành viên `private` của lớp cơ sở không thể truy cập từ lớp dẫn xuất, các thành viên `protected` và `public` của lớp cơ sở trở thành các thành viên `private` của lớp dẫn xuất.
- **protected**: các thành viên `private` của lớp cơ sở không thể truy cập từ lớp dẫn xuất, các thành viên `protected` và `public` của lớp cơ sở trở thành các thành viên `protected` của lớp dẫn xuất.

- **public**: các thành viên `private` của lớp cơ sở không thể truy cập từ lớp dẫn xuất, các thành viên `protected` và `public` của lớp cơ sở trở thành các thành viên của lớp dẫn xuất với thuộc tính không thay đổi.

Ví dụ:

```
// Lop Sinhvien la dan xuat cua lop Nghenghiep
class Sinhvien: public Nghenghiep {
    . . .
}
```

Khi một đối tượng của lớp dẫn xuất được tạo ra thì hàm khởi tạo của lớp cơ sở được áp dụng trước rồi mới tới hàm khởi tạo của lớp dẫn xuất, còn khi đối tượng kết thúc thì hàm hủy của lớp dẫn xuất được áp dụng trước rồi mới tới hàm hủy của lớp cơ sở.

Về mặt dữ liệu, một lớp dẫn xuất bao giờ cũng chứa toàn bộ dữ liệu của lớp cơ sở, do đó ta có thể thực hiện phép gán một đối tượng thuộc lớp dẫn xuất cho đối tượng thuộc lớp cơ sở (nhưng không thể theo chiều ngược lại)

```
Sinhvien A;
Nghenghiep B;
B = A;  // dung
A = B;  // sai
```

2.3.4 Đa kế thừa

Một lớp có thể được dẫn xuất từ những lớp cơ sở khác nhau với những kiểu dẫn dẫn xuất khác nhau, cú pháp như sau

class <tên lớp dẫn xuất>: <từ khóa dẫn xuất 1> <tên lớp cơ sở 1>, ... ,
<từ khóa dẫn xuất N> <tên lớp cơ sở N> {
..... nội dung .....
};

Ví dụ:

```
// Lop Sinhvien la dan xuat cua hai lop Nghenghiep va Truong
class Sinhvien: public Nghenghiep, public Truong {
    . . .
}
```

Hàm khởi tạo và hủy trong đa kế thừa được khai báo tương tự như trong đơn kế thừa, ngoại trừ việc phải sắp xếp thứ tự gọi hàm của các lớp cơ sở. Thông thường, thứ tự gọi hàm của các lớp cơ sở nên tuân theo thứ tự dẫn xuất từ các lớp cơ sở trong đa kế thừa.

2.3.5 Bộ nhớ động

Khi thực thi chương trình, máy tính sẽ sử dụng bộ nhớ để lưu trữ các dữ liệu như các biến, mảng và các đối tượng mà chúng ta khai báo, kích cỡ của chúng là cố định và không thể thay đổi trong thời gian thực thị chương trình. Đôi khi cách làm này gây lãng phí hoặc thiếu bộ nhớ, do đó chúng ta cần sử dụng bộ nhớ theo cách mà kích cỡ của nó chỉ có thể được xác định khi chương trình chạy. Để làm được điều này chúng ta sẽ sử dụng các bộ nhớ động (*dynamic memory*).

Việc cấp phát bộ nhớ động (*dynamic memory allocation*) trong C++ được thực hiện qua hai toán tử `new` và `delete`

- Toán tử `new` được dùng để cấp phát bộ nhớ, có cú pháp như sau

<con trỏ> = new <kiểu dữ liệu>
hoặc *<con trỏ> = new <kiểu dữ liệu>[kích thước]*

```
int *p1 = new int;
int *p2 = new int[10];  // danh vung nho cho 10 phan tu va tra ve 1
   con tro tro den dau khoi du lieu
int *p3 = new int(10);  // khoi tao gia tri cua p3 bang 10
```

Tất cả các biến cấp phát động mới đều được đặt trong vùng nhớ *free store*. Trong trường hợp hệ điều hành hết bộ nhớ để cấp phát, một con trỏ *null* (0) sẽ được trả về

```
int *p = new int;
if (p == NULL) {
  cout << "Khong du bo nho" << endl;
}
```

- Toán tử `delete` được dùng để giải phóng bộ nhớ được cấp phát khi nó không cần dùng đến nữa, có cú pháp như sau

  delete <con trỏ>
  hoặc *delete [] <con trỏ>*

```
delete p;
```

Lưu ý: khi sử dụng, các toán tử `new`/`delete` sẽ gọi các hàm khởi tạo và hủy của kiểu dữ liệu.

Các lỗi thường gặp khi sử dụng bộ nhớ động

- *Memory leak*: xảy ra khi chương trình không thể thu hồi (hay truy cập) một vùng nhớ đã cấp phát mặc dù không còn sử dụng. Nó xảy ra khi tất cả con trỏ trỏ đến vùng nhớ đó bị thay đổi giá trị trước khi vùng nhớ được giải phóng.
- *Dangling pointer*: xảy ra khi tồn tại con trỏ trỏ tới một vùng nhớ chưa được cấp phát, hoặc đã bị thu hồi.

CHƯƠNG 3

HISTOGRAM

Khái niệm **histogram** được đưa ra lần đầu tiên bởi Karl Pearson[1]. Nói một cách đơn giản, histogram là một dạng biểu đồ phân bố xác suất (tần suất) của một đại lượng biến thiên liên tục (*continous variable*)[2] bằng cách vẽ các tần số xuất hiện đại lượng đó trong các khoảng chia giá trị xác định (các khoảng chia này được gọi là các **bin**). Histogram thông thường có dạng biểu đồ cột đơn giản.

Ích lợi của histogram là nó có thể mô tả phân bố của một lượng dữ liệu lớn ở dạng đơn giản mà không làm mất bất cứ thông tin thống kê nào. Chúng ta vẫn có thể xác định được các đặc trưng thống kê của dữ liệu như giá trị trung bình (*mean*), độ lệch chuẩn (*standard deviation*),... từ histogram mà không cần phải xem lại dữ liệu gốc.

3.1 Histogram

3.1.1 Khai báo histogram

Trong ROOT, lớp histogram được kí hiệu là `THdp`, với `d` là số chiều của histogram và `p` là kiểu của các phần tử chứa trong histogram[3]. Các lại histogram chính trong ROOT gồm có 1D, 2D, 3D và profile. Mối quan hệ giữa các lớp histogram được minh họa trong Hình 3.1.

- `TH1` là lớp histogram cơ bản (*base class*) để xây dựng các lớp khác
- `TH1C`, `TH2C` và `TH3C` chứa 1 byte/bin (giá trị lớn nhất của bin = 255)
- `TH1S`, `TH2S` và `TH3S` chứa 1 số kiểu short/bin (giá trị lớn nhất của bin = 65 535).
- `TH1I`, `TH2I` và `TH3I` chứa 1 số kiểu nguyên/bin (giá trị lớn nhất của bin = 2 147 483 647).
- `TH1F`, `TH2F` và `TH3F` chứa 1 số kiểu thực/bin (giá trị lớn nhất của bin = 7 digits).
- `TH1D`, `TH2D` và `TH3D` chứa 1 số kiểu thực kép/bin (giá trị lớn nhất của bin = 14 digits).

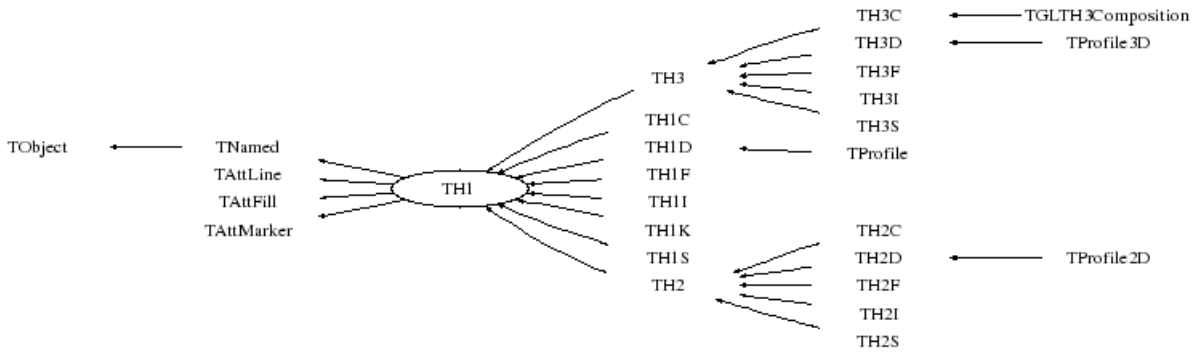
Ta có thể khai báo một histogram theo cú pháp như sau:

*<lớp histogram> *<tên histogram> = new <lớp histogram>("<tên histogram>", "<tiêu đề>", <số khoảng chia>, <biên trái>, <biên phải>);*

[1] Karl Pearson (1857 - 1936) là một nhà toán học và sinh trắc học người Anh, được ghi nhận là người đã thành lập nên bộ môn thống kê toán học *mathematical statistics*.

[2] Histogram cũng có thể sử dụng cho các đại lượng rời rạc.

[3] Tham khảo thêm tại `ftp://root.cern.ch/root/doc/3Histograms.pdf`



Hình 3.1: Mối quan hệ giữa các lớp histogram

Ví dụ: *khai báo histogram 1 chiều*

```
TH1F *h = new TH1F("h", "Pho gamma", 8192, 0, 8192);
```

Ví dụ trên khai báo 1 histogram tên là `h` có tiêu đề là `"Pho gamma"` với số khoảng chia là 8192 khoảng (*bin*), độ rộng mỗi khoảng chia là

$$\frac{\text{biên phải} - \text{biên trái}}{\text{số khoảng chia}} = \frac{8192 - 0}{8192} = 1$$

Nếu muốn phân chia lại khoảng giá trị của histogram, ta sử dụng phương thức `TH1::Rebin()`, phương thức này sẽ nhóm các khoảng chia kề nhau lại (mặc định là 2 khoảng chia)

```
h->Rebin(4); // so khoang chia hien gio la 8192/4 = 2048, do rong moi
    khoang la 4
```

Với cách thức khai báo như trên, độ rộng mỗi khoảng chia là không đổi. Trong trường hợp chúng ta muốn tạo một histogram với độ rộng khoảng chia khác nhau, ta có thể làm theo cách sau

```
const Int_t NBINS = 5;  // so khoang chia
Double_t edges[NBINS + 1] = {0.0, 0.2, 0.3, 0.6, 0.8, 1.0};
// Xac dinh bien cua cac khoang chia [0.0, 0.2] [0.2, 0.3] ...
TH1* h = new TH1D("h", "Tieu de", NBINS, edges);
```

Một số ví dụ khác

```
TH1I* h1 = new TH1I("h1", "Hisogram 1-chieu", 100, 0.0, 4.0);
TH2F* h2 = new TH2F("h2", "Hisogram 2-chieu", 40, 0.0, 2.0, 30, -1.5, 3.5)
    ;
TH3D* h3 = new TH3D("h3", "Hisogram 3-chieu", 80, 0.0, 1.0, 100, -2.0,
    2.0, 50, 0.0, 3.0);
```

Lưu ý: khi khai báo histogram như trên đồng nghĩa với việc ta đang khai báo con trỏ thuộc lớp histogram, và bộ nhớ để chứa histogram được cấp phát động thông qua toán tử `new`. Với khai báo dạng này, để truy xuất các phương thức histogram ta cần sử dụng dấu mũi tên (`->`)

```
h->Method();
```

Trong trường hợp ta muốn khai báo đối tượng (*object*) thuộc lớp histogram

```
TH1F h1("h1", "Hisogram 1-chieu", 100, 0.0, 4.0);
```

Để truy xuất các phương thức histogram lúc này ta sử dụng dấu chấm (`.`)

```
h.Method();
```

Tuy nhiên, trong thực tế với trình thông dịch CINT, cả hai cách truy xuất trên đều được chấp nhận.

3.1.2 Điền giá trị vào histogram

Để điền giá trị vào histogram, ta sử dụng phương thức `TH1::Fill()`

```
h1->Fill(x);
h2->Fill(x,y);
h3->Fill(x,y,z);
```

Phương thức này sẽ tăng 1 số đếm tại vị trí bin tương ứng với giá trị đưa vào. Trong trường hợp ta muốn điền vào một giá trị khác 1 (trọng số `w`), ta làm như sau

```
h1->Fill(x,w);
h2->Fill(x,y,w);
h3->Fill(x,y,z,w);
```

hoặc cộng trực tiếp vào bằng phương thức `TH1::AddBinContent()`

```
h1->AddBinContent(i);  // i la so thu tu cua bin
h1->AddBinContent(i,w);
```

Lưu ý:

- Bin 0 (i = 0) sẽ chứa các giá trị nhỏ hơn khoảng giá trị từ biên trái đến biên phải của histogram (*underflow bin*).
- Bin 1 (i = 1) ứng với khoảng chia đầu tiên cùng với giá trị biên trái.
- Bin kế cuối (i = nbins) ứng với khoảng chia cuối cùng với giá trị biên phải.
- Bin cuối cùng (i = nbins+1) chứa các giá trị lớn hơn khoảng giá trị từ biên trái đến biên phải của histogram (*overflow bin*).

Phương thức `TH1::Sumw2()` thực hiện lưu trữ tổng bình phương các trọng số đi kèm với các giá trị của histogram trong quá trình ghi nhận (nên gọi ngay sau khi khai báo histogram)

```
h1->Sumw2();
```

Ngoài ra, thay vì điền lần lượt từng giá trị vào tring bin tương ứng, ta có thể gán thẳng giá trị vào trong bin bằng phương thức `TH1::SetBinContent()`

```
h1->SetBinContent(i,value);
```

Quá trình ngược lại, lấy ra giá trị tương ứng với vị trí bin, được thực hiện thông qua phương thức `TH1::GetBinContent()`

```
Float_t value = h1->GetBinContent(i);
```

Điền giá trị ngẫu nhiên

Phương thức `TH1::FillRandom()` được sử dụng để điền ngẫu nhiên vào histogram theo một phân bố cho trước

```
TH1F* h1 = new TH1F ("h1","Histo from a Gaussian",100,-3,3);
h1->FillRandom("gaus",10000);
```

Hoặc theo một phân bố histogram

```
TH1F h2 = new TH1F ("h2","Histo from existing histo",100,-3,3);
h2->FillRandom(&h1, 1000);
```

Trong ví dụ trên, chương trình sẽ tính tích phân trên toàn khoảng chia của `h1` và chuẩn về 1. Sau đó, một số ngẫu nhiên `r` sẽ được gieo trong khoảng từ 0 đến 1, vị trí bin của histogram `h2` tương ứng với giá trị của `r` sẽ được tăng số đếm. Ngoài ra, ta cũng có thể sử dụng phương thức `TH1::GetRandom()` để làm điều tương tự

```
h2->Fill(h1->GetRandom());
```

3.1.3 Một số phương thức thông dụng cho histogram

Để tạo một histogram giống với histogram đang có, ta làm như sau

```
TH1F* h2  =  (TH1F*)h1->Clone();
```

Ta cũng có thể reset lại các giá trị trong mỗi bin của histogram bằng cách

```
h1->Reset();
```

Tính tích phân histogram

```
h1->Integral(); // tinh tich phan tu tren toan khoang chia (1->nbins)
h1->Integral(i,j); // tinh tich phan tu bin i den bin j
h1->IntegralAndError(i,j,err); // tinh tich phan tu bin i den bin j, sai
   so thong ke cua tich phan duoc luu vao trong err
```

Nhân histogram với một hệ số

```
h1->Scale(scale);
```

Ta cũng có thể sử dụng cách này để chuẩn hóa histogram về 1

```
Float_t scale  =  1./h1->Integral(); // khai bao bien scale bang nghich
   dao cua tich phan cua h
h1->Scale(scale); // nhan h1 voi he so scale, dong nghia voi viec chuan
   hoa h ve 1
```

Làm trơn histogram

```
h1->Smooth(2); // so lan lam tron la 2
```

Các phương thức tính cho histogram

- `Add(h1,c1)`: cộng thêm histogram *h1* với trọng số *c1* ($h = h + c1 * h1$).
- `Multiply(h1,c1)`: nhân với histogram *h1* với trọng số *c1* ($h = h * c1 * h1$).
- `Divide(h1,c1)`: chia cho histogram *h1* với trọng số *c1* ($h = h/(c1 * h1)$).
- `Divide(h1,h2,c1,c2)`: lấy giá trị thương của hai histogram *h1* và *h2* ($h = c1 * h1/(c2 * h2)$).
- `GetAsymmetry(h2,c2,dc2)`: trả về giá trị asymmetry của hai histogram *h1* và *h2* ($Asym = (h1 - h2)/(h1 + h2)$), `c2` là trọng số tương đối giữa 2 histogram và `dc2` là sai số tương ứng.
- `Interpolate(x)`: nội suy tuyến tính giá trị tại x.

Ví dụ:

```
{
  TH1F *h1 = new TH1F("h1", "", 5, 0, 5);
   h1->SetBinContent(1,5);
   h1->SetBinContent(2,2.3);
   h1->SetBinContent(3,0);
   h1->SetBinContent(4,1);
   h1->SetBinContent(5,-3.1);

   TH1F *h2 = h1->Clone("h2");  // h2 = h1
   h2->Add(h1,-1);  // h2 = h2 - h1
   h2->Divide(h1);  // h2 = h2 / h1
   h2->Divide(h,h1);  // h2 = h / h1
   TH1F *h3 = h1->GetAsymmetry(h2);
}
```

Truy xuất các thông số của histogram

- `GetEntries()`: số lượng các giá trị được ghi nhận trong histogram
- `GetMean()`, `GetMeanError()`: giá trị trung bình và sai số tương ứng
- `GetRMS()`, `GetRMSError()`: căn của phương sai và sai số tương ứng
- `GetMaximum()`, `GetMinimum()`: giá trị lớn nhất và nhỏ nhất trong histogram
- `GetMaximumBin()`, `GetMinimumBin()`: lấy vị trí bin tương ứng với giá trị lớn nhất và nhỏ nhất
- `GetBinCenter(i)`: giá trị trung tâm của bin thứ i

Ví dụ:

```
root [0] cout << "So gia tri duoc ghi nhan: " << h1->GetEntries() << endl;
So gia tri duoc ghi nhan: 4
root [1] cout << "Gia tri trung binh: " << h1->GetMean() << endl;
Gia tri trung binh: 2.825
root [2] cout << "Can cua phuong sai: " << h1->GetRMS() << endl;
Can cua phuong sai: 1.17978
root [3] cout << "So dem cao nhat: " << h1->GetMaximum() << endl;
So dem cao nhat: 2
root [4] cout << "Vi tri bin co so dem cao nhat: " << h1->GetMaximumBin()
    << endl;
Vi tri bin co so dem cao nhat: 4
```

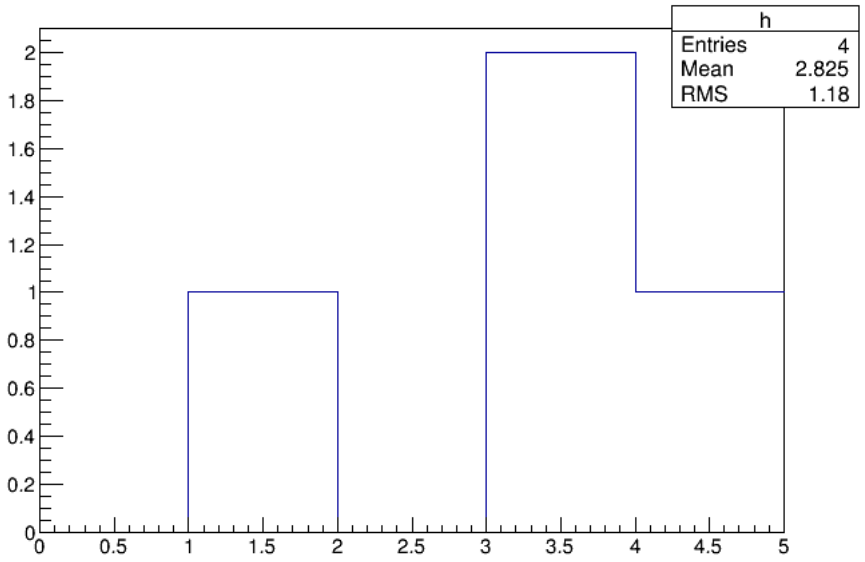
3.2 Vẽ histogram

Để vẽ histogram trong ROOT, ta sử dụng phương thức `TH1::Draw()`, phương thức này sẽ tạo ra một đối tượng thuộc lớp `THistPainter` và lưu con trỏ của đối tượng này như là một thành viên của histogram. Nếu muốn tạo ra một bản sao của histogram khi vẽ, ta sử dụng phương thức `TH1::DrawClone()`.

Ví dụ: *(xem Hình 3.2)*

```
{
   // Tao histogram co 5 khoang chia tu 0 den 5
   TH1F *h1 = new TH1F("h", "", 5, 0, 5);
   h1->Fill(1); // Tang 1 so dem tai khoang chia tu 1 den 2
   h1->Fill(3); // Tang 1 so dem tai khoang chia tu 3 den 4
   h1->Fill(3); // Tang 1 so dem tai khoang chia tu 3 den 4
```

```
    h1->Fill(4.3); // Tang 1 so dem tai khoang chia tu 4 den 5
    h1->Draw(); // Ve histogram
}
```



Hình 3.2: Biểu đồ của ví dụ

Thông thường phương thức `TH1::Draw()` sẽ xóa hình vẽ cũ và tạo ra hình vẽ mới mỗ khi được gọi. Trong trường hợp ta muốn vẽ nhiều histogram cùng lúc thì với các histogram được vẽ sau histogram đầu tiên ta thêm tùy chỉnh `"same"`

```
h1->Draw("same");
```

Ngoài ra, ta cũng có thể vẽ bản sao phân bố đã được chuẩn hóa của histogram bằng phương thức `TH1::DrawNormalized()`

```
h1->DrawNormalized("same", 2);  // chuan hoa h1 ve 2
```

3.2.1 Thiết lập các tùy chỉnh cho đồ thị

Tùy chỉnh cho dạng đồ thị

Khi vẽ đồ thị bằng phương thức `TH1::Draw()`, ta có thể sử dụng các tùy chỉnh để vẽ các dạng đồ thị khác nhau của cùng một histogram. Bảng 3.1 trình bày một số tùy chỉnh thường sử dụng cho các histogram 1D và 2D.

Tùy chỉnh kiểu vẽ

Các tùy chỉnh cho kiểu vẽ bao gồm cho điểm đánh dấu (*marker*), đường (*line*) và chế độ tô màu (*fill*)

- **Chọn màu:** `SetMarkerColor()`, `SetLineColor()`, `SetFillColor()`
- **Chọn kiểu vẽ:** `SetMarkerStyle()`, `SetLineStyle()`, `SetFillStyle()`
- **Chọn bề dày:** `SetMarkerWidth()`, `SetLineWidth()`

Bảng 3.1: Các tùy chỉnh của phương thức `TH1::Draw()` cho các histogram 1D và 2D (Xem thêm tại http://root.cern.ch/root/html/THistPainter.html)

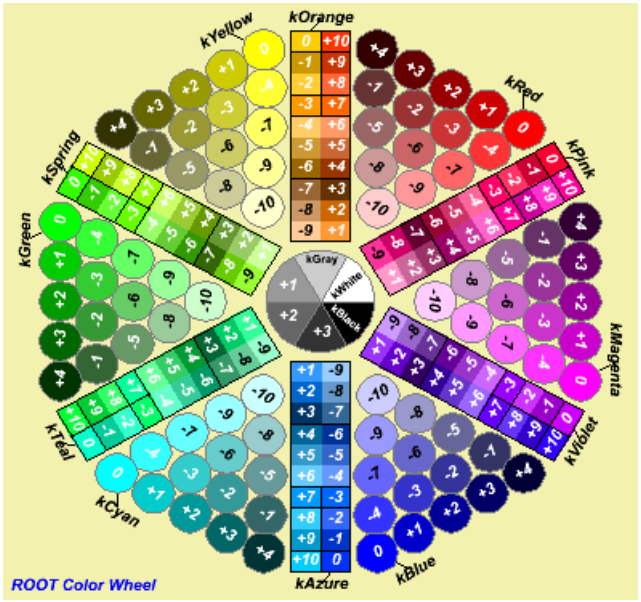
1D & 2D	SAME	Vẽ trên cùng 1 đồ thị
	E	Vẽ các thanh sai số
	HIST	Vẽ dạng histogram
	TEXT	Ghi giá trị của điểm trên đồ thị
	LEGO	Vẽ dạng lego có loại bỏ các đường bị ẩn
	LEGO1	Vẽ dạng lego có loại bỏ các mặt bị ẩn
	LEGO2	Vẽ dạng lego có sử dụng màu để hiển thị giá trị
1D	B	Vẽ biểu đồ cột
	C	Vẽ đường làm trơn qua các điểm
	L	Vẽ đường nối các điểm
	P	Vẽ các điểm đánh dấu
	P0	Vẽ các điểm đánh dấu kể cả cho các bin không có giá trị
	E0	Vẽ thanh sai số
	E1	Vẽ thanh sai số có vạch ngang vuông góc với thanh sai số
	E2	Vẽ thanh sai số có khung chữ nhật
	E3	Vẽ thanh sai số và tô phần diện tích thanh theo trục tung
	E4	Vẽ thanh sai số và tô diện tích đã được làm trơn
	E5	Tương tự E3 nhưng bỏ qua các giá trị 0
	E6	Tương tự E4 nhưng bỏ qua các giá trị 0
2D	CYL	Vẽ trên trục tọa độ trụ
	POL	Vẽ trên trục tọa độ cực
	SPH	Vẽ trên trục tọa độ cầu
	PSR	Vẽ trên trục tọa độ pseudorapidity/phi
	BOX	Vẽ dạng hộp
	COL	Vẽ dạng hộp có tô màu
	COLZ	Tương tự như COL nhưng có thêm thang màu
	CONT	Vẽ đường contour (tương tự CONT0)
	CONT0	Vẽ đường contour có màu
	CONT1	Vẽ đường contour có phân biệt kiểu đường
	CONT2	Vẽ đường contour với cùng kiểu đường
	CONT3	Vẽ đường contour có tô màu
	CONT4	Vẽ đường contour có sử dụng màu của mặt
	CONT5	Vẽ đường contour có sử dụng tam giác Delaunay
	SURF	Vẽ dạng mặt
	SURF1	Vẽ dạng mặt
	SURF2	Vẽ dạng mặt có sử dụng màu để hiển thị giá trị
	SURF3	Tương tự SURF, có thêm đường contour phía trên
	SURF4	Vẽ dạng mặt được làm trơn (Gouraud shading)
	SURF5	Tương tự SURF3, có đường contour màu
	SCAT	Vẽ dạng chấm điểm

Hình 3.3 đến Hình 3.6 trình bày các bảng kí hiệu màu, kiểu đường và kiểu marker có trong ROOT.

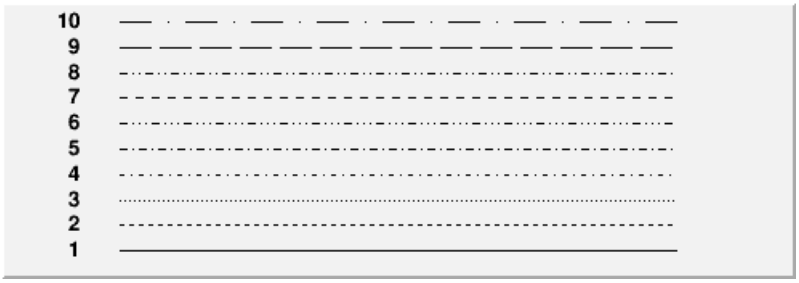
Ví dụ: *(xem Hình 3.7)*

```
{
  TH1F *h1 = new TH1F("h1","1D Gaussian", 100, -100, 100);
  Float_t mean = 0, sigma = 20;
  for (Int_t i = 0; i < 10000; i++) {
      h1->Fill(gRandom->Gaus(mean,sigma));
```

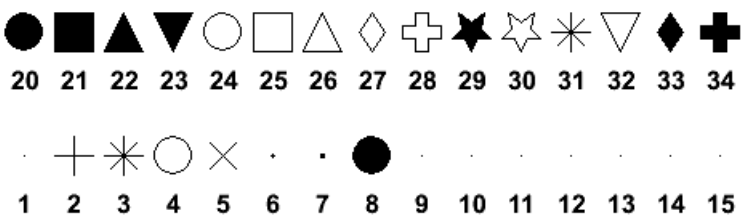
Hình 3.3: Bảng kí hiệu màu



Hình 3.4: Bảng kí hiệu màu



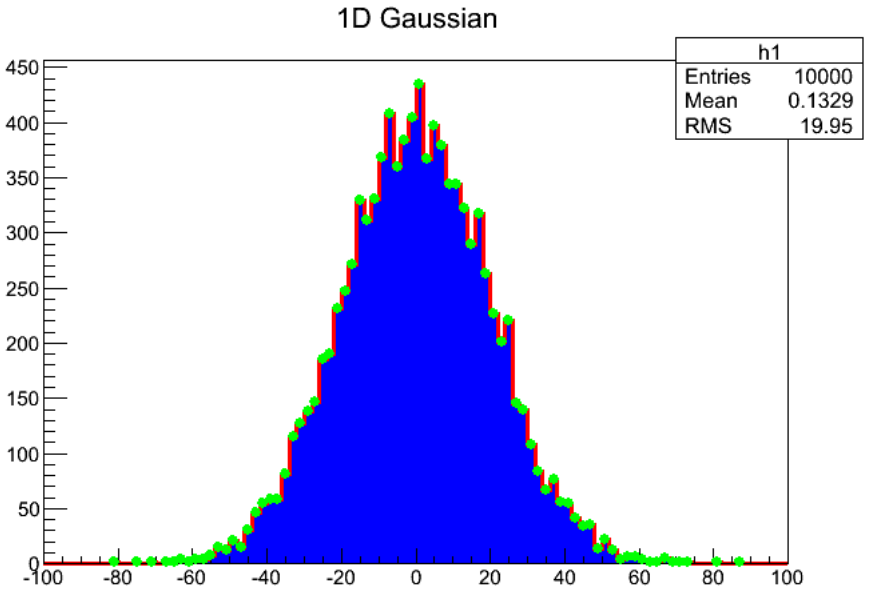
Hình 3.5: Bảng kiểu đường



Hình 3.6: Bảng kiểu marker

```
}
h1->SetFillColor(kBlue);  // chon mau to mau xanh
```

```
  h1->SetLineColor(kRed);  // chon mau line mau do
  h1->SetLineWidth(3);  // chon be day line bang 3
  h1->Draw("histo");
  h1->SetMarkerStyle(20);  // chon kieu marker 20
  h1->SetMarkerColor(kGreen);  // chon mau marker xanh la
  h1->SetMarkerSize(1);  // chon kich thuoc marker bang 1
  h1->Draw("same p");
}
```



Hình 3.7: Đồ thị phân bố ngẫu nhiên dạng Gauss 1 chiều

Tùy chỉnh cho trục đồ thị

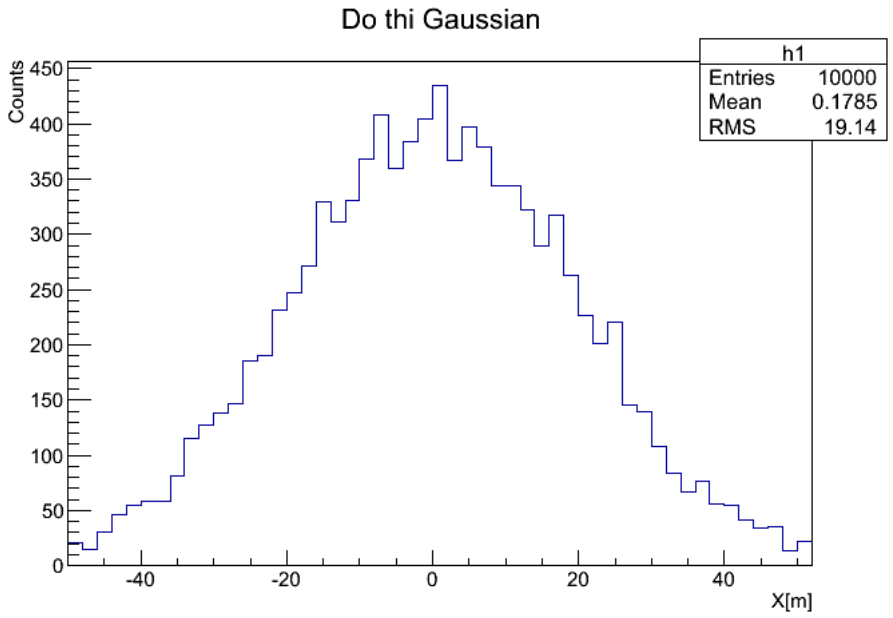
Để tiến hành hiệu chỉnh các trục đồ thị, ta sử dụng các phương thức `TH1::GetXaxis()`, `TH1::GetYaxis()`, `TH1::GetZaxis()` để chọn các trục x,y,z tương ứng và sau đó sử dụng các phương thức của lớp `TAxis`

- `SetTitle(" ")`: đặt tiêu đề cho trục đồ thị
- `SetRangeUser(a,b)`: chọn khoảng vẽ từ a đến b
- `SetTitleSize()`, `SetTitleFont()`, `SetTitleOffset()`: chọn cỡ chữ và font chữ và offset cho tiêu đề trên trục
- `SetLabelSize()`, `SetLabelFont()`: chọn cỡ chữ và font chữ hiển thị trên trục

Ví dụ: *(xem Hình 3.8)*

```
{
  TH1F *h1 = new TH1F("h1","1D Gaussian", 100, -100, 100);
  Float_t mean = 0, sigma = 20;
  for (Int_t i = 0; i < 10000; i++) {
      h1->Fill(gRandom->Gaus(mean,sigma));
  }
  h1->SetTitle("Do thi Gaussian"); // dat tieu de
  h1->GetXaxis()->SetTitle("X[m]"); // gia tri cua truc x
  h1->GetYaxis()->SetTitle("Counts"); // gia tri cua truc y
  h1->GetXaxis()->SetRangeUser(-50,50); // khoang ve
  h1->Draw();
```

```
}
```



Hình 3.8: Ví dụ chỉnh các trục cho đồ thị

3.2.2 Hiển thị bảng thống kê

Trên các đồ thị được vẽ bằng ROOT, ta thấy góc trên cùng bên phải luôn xuất hiện một bảng liệt kê tên của histogram cùng các thông số như giá trị trung bình, RMS (*root mean square*)[4],... đó là các bảng thống kê (*statistics box*). Để tắt các bảng này, ta có thể thiết lập `TH1::SetStats(kFALSE)`

Để thay đổi thông số hiển thị trong các bảng thống kê của histogram, ta có thể sử dụng phương thức

```
gStyle->SetOptStat(mode)
```

Có tất cả 9 thông số cho `mode` có thể được bật (1, 2) hoặc tắt (0) theo thứ tự `ksiourmen` (giá trị mặc định là `000001111`).

- `n = 1`: tên của histogram
- `e = 1`: số entry (dữ liệu vào)
- `m = 1`: giá trị trung bình
- `m = 2`: giá trị trung bình và sai số của nó
- `r = 1`: RMS
- `r = 2`: RMS và sai số của nó
- `s = 1`: skewness[5]
- `s = 2`: skewness và sai số của nó

[4] Mặc dù mang tên là *root mean square* (căn của trị trung bình của bình phương) nhưng đây lại là giá trị ước lượng độ lệch chuẩn (*standard deviation*) của histogram, nếu muốn tính giá trị RMS thực sự của histogram ta có thể làm như sau

```
double rootMeanSquare = std::sqrt(h1->GetMean()*h1->GetMean()+h1->GetRMS()*h1->GetRMS())
```

[5] Skewness là moment bậc 3 được dùng để đánh giá độ đối xứng (*symmetry*) của phân bố, xem thêm trong phần 6.3 của tài liệu ***"Các kiến thức cơ sở của phương pháp Monte Carlo"*** (http://goo.gl/BsMUqv)

- `s = 1`: kurtosis[6]
- `s = 2`: kurtosis và sai số của nó
- `u = 1`: underflow
- `o = 1`: overflow
- `i = 1`: tích phân (tổng) giá trị của tất cả các bin

Ví dụ:

```
gStyle->SetOptStat(1111);
gStyle->SetOptStat("ne"); // xuat ra ten histogram va so entry
gStyle->SetOptStat("nemr");  // xuat thong so mac dinh
```

3.3 Thay đổi nhãn cho histogram

Thông thường khi vẽ đồ thị, các bin của histogram thường được hiển thị dưới dạng số, tuy nhiên ta có thể gán nhãn hiển thị là kiểu chữ số (*alphanumeric*) cho các bin này thông qua phương thức `TAxis::SetBinLabel()`

```
h1->GetXaxis()->SetBinLabel(1, "January");
h1->GetXaxis()->SetBinLabel(2, "February");
```

Ngoài ra phương thức `TH1::Fill()` cũng có thể điền dữ liệu dưới dạng các chuỗi vào trong histogram

```
{
  const Int_t nx = 20;
  char *people[nx] = {"Jean","Pierre","Marie","Odile","Sebastien",
    "Fons","Rene","Nicolas","Xavier","Greg","Bjarne","Anton","Otto",
    "Eddy","Peter","Pasha","Philippe","Suzanne","Jeff","Valery"};
  TH1F *h = new TH1F("h","test",3,0,3);
  h->SetStats(0);
  h->SetFillColor(38);
  h->SetBit(TH1::kCanRebin);
  for (Int_t i=0; i<5000; i++) {
    Int_t r = gRandom->Rndm()*20;
    h->Fill(people[r],1);
  }
  h->LabelsDeflate();
  h->Draw();
}
```

Sau khi kết thúc việc điền dữ liệu, đôi khi ta cần hiệu chỉnh số lượng bin cho khớp với số lượng nhãn, điều này được thực hiện với phương thức `TH1::LabelsDeflate()`.

Với cách khai báo như trên, trục tọa độ sẽ được vẽ với thứ tự bin tương ứng với thứ tự chuỗi dữ liệu được điền vào. Ta có thể sắp xếp lại thứ tự này với phương thức `TH1::LabelsOption(option,axis)` trong đó `axis` là trục tọa độ còn `option` là các tùy chỉnh

- `"a"` sắp xếp theo thứ tự alphabetic
- `"<"` sắp xếp theo giá trị giảm dần
- `">"` sắp xếp theo giá trị tăng dần
- `"h"` vẽ các nhãn nằm ngang

[6] Kurtosis là moment bậc 4 được dùng để đánh giá độ phẳng (*flatness*) của phân bố, xem thêm trong cùng tài liệu với skewness

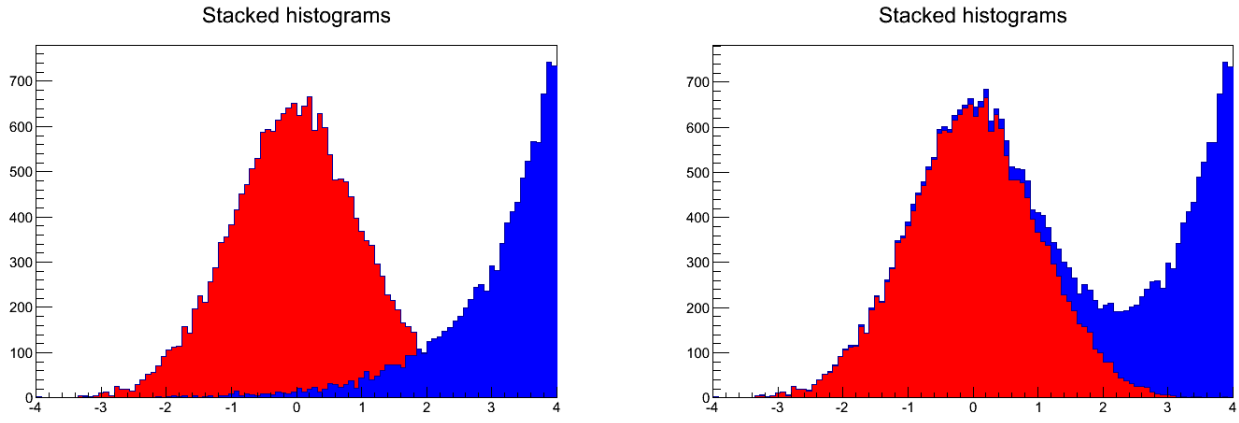
- "v" vẽ các nhãn nằm dọc
- "u" vẽ các nhãn hướng lên (căn phải)
- "d" vẽ các nhãn hướng xuống (căn trái)

3.4 Xếp chồng histogram

Chồng histogram (histogram stack) là một tập hợp các histogram 1 chiều (TH1) được vẽ xếp chồng lên nhau, được tạo thông qua lớp THStack. Để thêm các histogram vào trong chồng, ta có thể sử dụng phương thức THStack::Add(TH1 *h). Lớp THStack không sở hữu các đối tượng có trong chồng histogram, theo mặc định phương thức THStack::Draw() sẽ vẽ các histogram chồng lên nhau theo thứ tự như trong ví dụ bên dưới. Trong trường hợp tùy chỉnh "nostack" được sử dụng, các histogram sẽ được vẽ đè lên nhau tương tự như khi vẽ nhiều histogram cùng lúc với tùy chỉnh "same".

Ví dụ:

```
{
  THStack hs("hs","Stacked histograms");
  //Tao histogram h1 bang cach gieo ngau nhien ham Gauss
  TH1F *h1 = new TH1F("h1","Histogram 1",100,-4,4);
  h1->FillRandom("gaus",20000);
  h1->SetFillColor(kRed);
  //Tao histogram h2 bang cach gieo ngau nhien ham exponential
  TH1F *h2 = new TH1F("h2","Histogram 2",100,-4,4);
  h2->FillRandom("expo",10000);
  h2->SetFillColor(kBlue);
  // Them hai ham vao stack
  hs->Add(h1);
  hs->Add(h2);
  // Ve histogram
  // hs->Draw("nostack");   // ve khong xep chong
  hs->Draw();   // ve xep chong
}
```



Hình 3.9: Ví dụ về xếp chồng histogram: hình bên trái là hai histogram vẽ trên cùng đồ thị, hình bên phải là hai histogram được xếp chồng lên nhau.

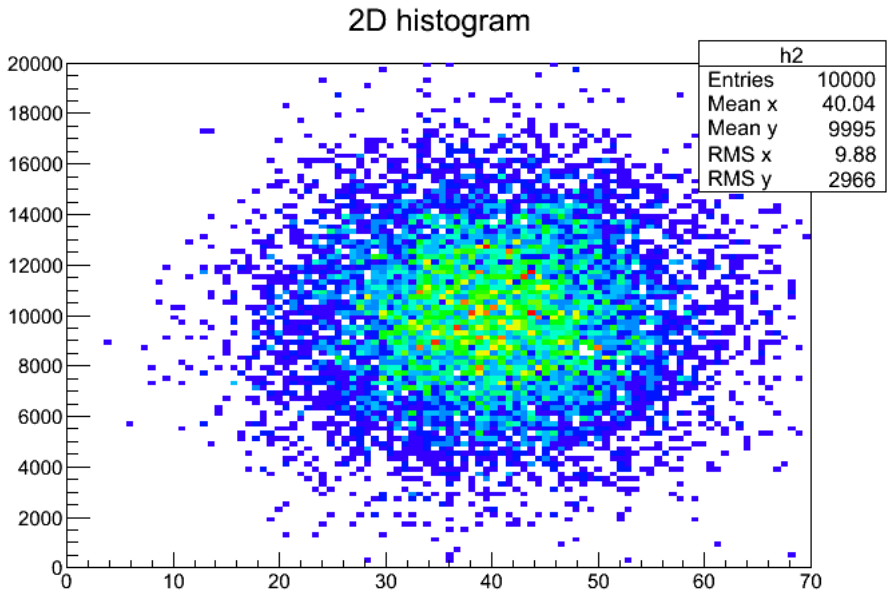
3.5 Histogram 2 chiều và 3 chiều

3.5.1 Khai báo histogram

Cách thức khai báo histogram 2 chiều và 3 chiều cũng tương tự như với histogram 1 chiều

Ví dụ: *Histogram 2 chiều (xem Hình 3.10)*

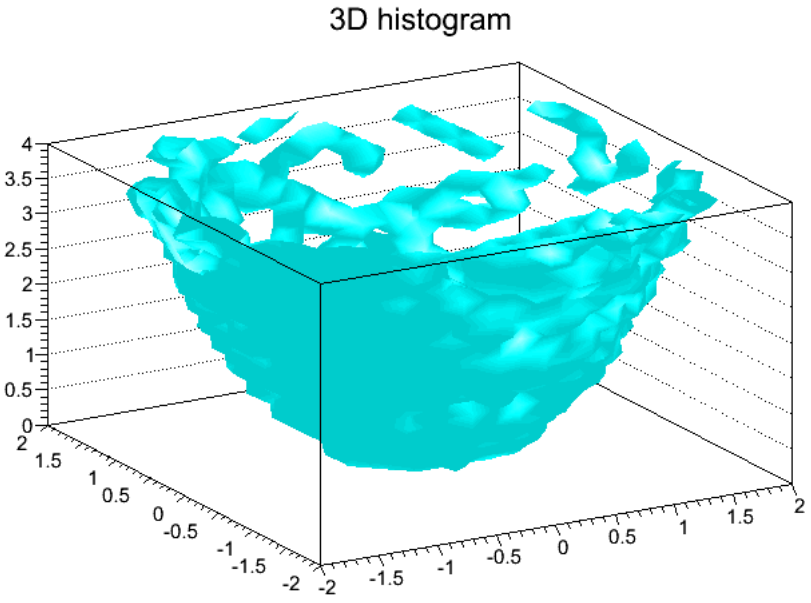
```
{
  TH2D *h2 = new TH2D("h2","2D histogram",100,0,70,100,0,20000);
  Double_t x,y;
  for (Int_t i = 0; i < 10000; i++) {
    // Gieo ngau nhien cac gia tri x,y theo ham Gauss
    x=gRandom->Gaus(40,10);
    y=gRandom->Gaus(10000,3000);
    h2->Fill(x,y);
  }
  h2->Draw("col");
}
```



Hình 3.10: Ví dụ histogram 2 chiều

Ví dụ: *Histogram 3 chiều (xem Hình 3.11)*

```
{
  TH3D *h3=new TH3D("h3","3D histogram",20,-2,2,20,-2,2,20,0,4);
  Double_t x,y,z;
  for (Int_t i=0; i<10000; i++) {
    // Gieo ngau nhien cac gia tri x,y theo ham Gauss
    x=gRandom->Gaus(-2,2);
    y=gRandom->Gaus(-2,2);
    z=x*x+y*y;
    h3->Fill(x,y,z);
  }
  h3->SetFillColor(kCyan);
  h3->Draw("iso");
}
```



Hình 3.11: Ví dụ histogram 3 chiều

3.5.2 Hình chiếu của histogram

Với các histogram 2D và 3D, ta có thể thu được hình chiếu của chúng trên các trục đồ thị thông qua các phương thức `THd::ProjectionX()`, `THd::ProjectionY()` và `THd::ProjectionZ()`.

Ví dụ:

```
{
  TH2F *h2 = new TH2F("h2","",40,-4,4,40,-20,20);
  Float_t px, py;
  for (Int_t i = 0; i < 25000; i++) {
    gRandom->Rannor(px,py);
    h2->Fill(px,5*py);
  }
  TH1D * projh2X = h2->ProjectionX();
  TH1D * projh2Y = h2->ProjectionY();
}
```

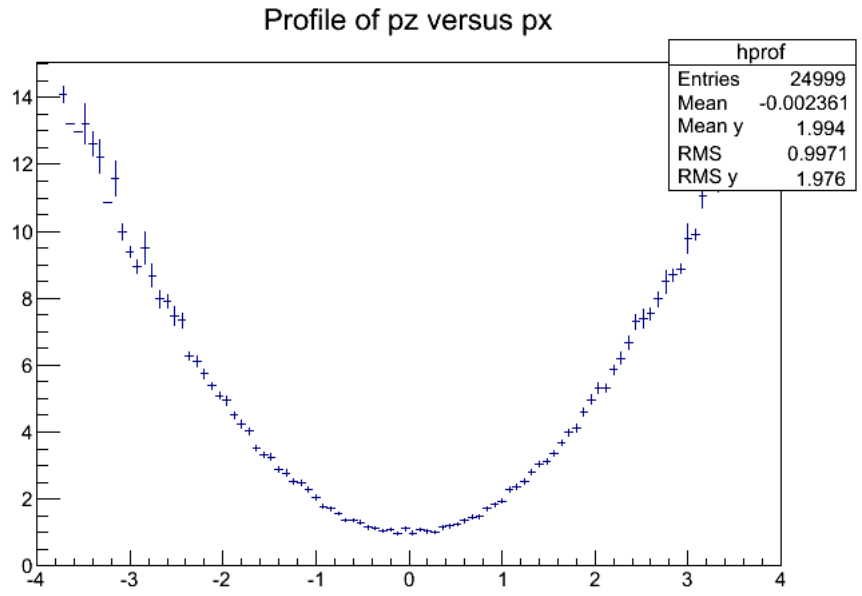
3.5.3 Profile histogram

Profile histogram nhằm giúp cho người dùng thấy được sự tương quan giữa các biến với nhau trong cùng một histogram (2 chiều hoặc 3 chiều). Trong một số trường hợp, các profile có thể được xem là sự biểu diễn thay thế cho các đồ thị histogram nhiều chiều. Nhiệm vụ của profile là biểu diễn giá trị trung bình và sai số của một biến theo từng bin của biến kia. Để xây dựng các profile histogram ta sử dụng lớp `TProfile` cho 1 chiều và `TProfile2D` cho 2 chiều.

Ví dụ: *Profile 1 chiều (xem Hình 3.12)*

```
{
  TProfile *hprof = new TProfile("hprof","Profile of pz versus px"
     ,100,-4,4,0,20);
  Float_t px, py, pz;
  for ( Int_t i=0; i<25000; i++) {
  gRandom->Rannor(px,py);
    pz = px*px + py*py;
      hprof->Fill(px,pz,1);
```

```
  }
  hprof->Draw();
}
```



Hình 3.12: Ví dụ profile 1 chiều

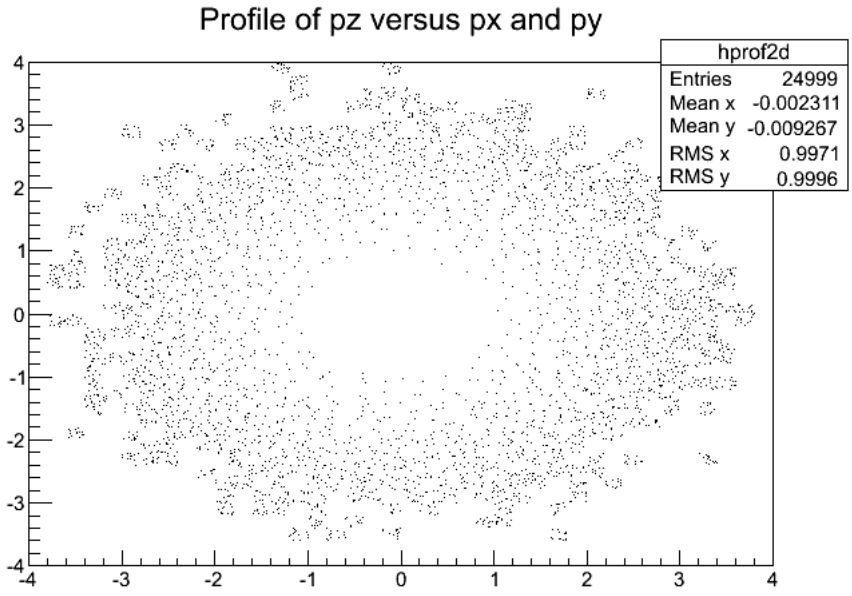
Ví dụ: *Profile 2 chiều (xem Hình 3.13)*

```
{
  TProfile2D *hprof2d  = new TProfile2D("hprof2d","Profile of pz versus px
      and py",40,-4,4,40,-4,4,0,20);
  Float_t px, py, pz;
  for ( Int_t i=0; i<25000; i++) {
  gRandom->Rannor(px,py);
    pz = px*px + py*py;
      hprof2d->Fill(px,py,pz,1);
  }
  hprof2d->Draw();
}
```

Các profile histogram có thể thu được từ các histogram có số chiều cao hơn thông qua các phương thức `TH2::ProfileX()`, `TH2::ProfileY()`, `TH3::Project3DProfile()`

Ví dụ:

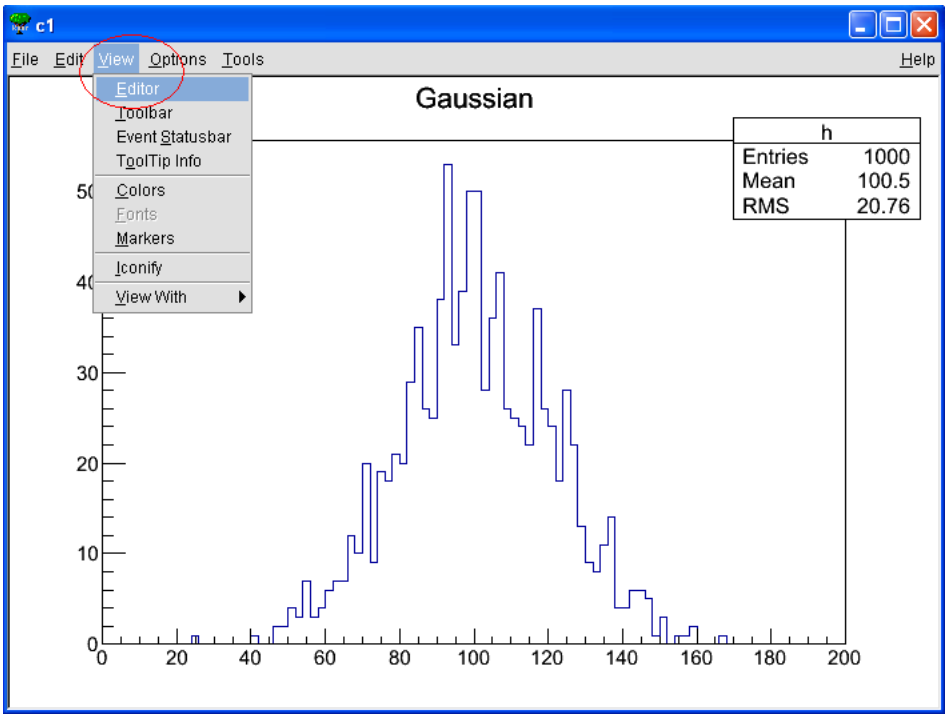
```
{
  TH2F *h2 = new TH2F("h2","h2",40,0,1,40,0,10);
  Float_t u,v;
  for (Int_t i=0;i<1000;i++) {
    u = gRandom->Rndm();
    v = 10*gRandom->Rndm();
    h2->Fill(u,v);
  }
  // dung TProfile de chuyen bai toan 2 chieu thanh 1 chieu
  TProfile *prof = h2->ProfileX();
```



Hình 3.13: Ví dụ profile 2 chiều

3.6 Graphics Editor

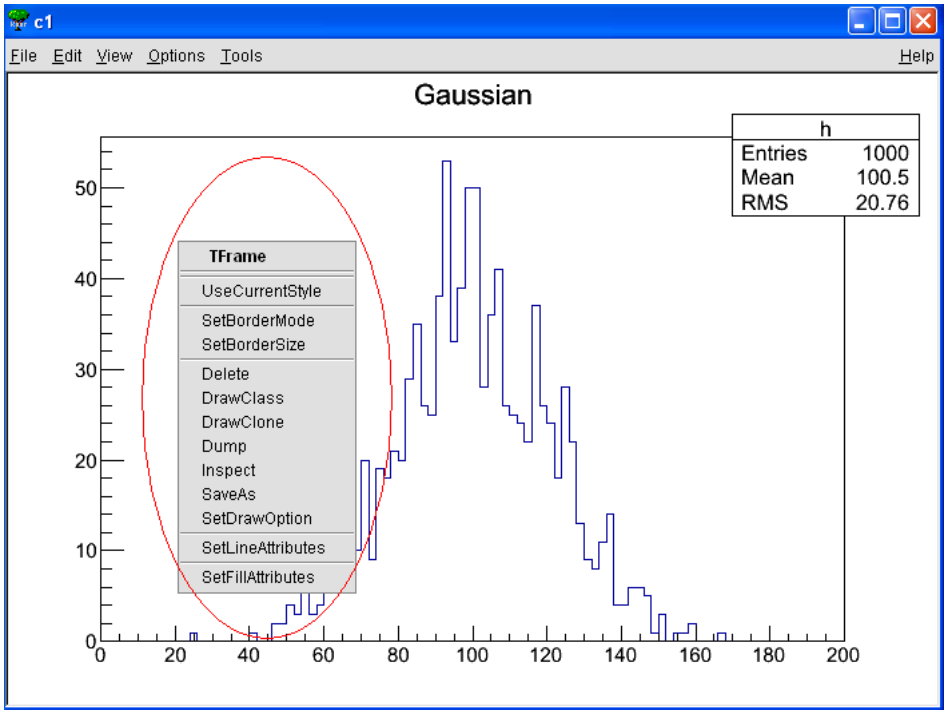
Bên cạnh việc hiệu chỉnh các hình vẽ bằng câu lệnh, ROOT cũng xây dựng các lớp GUI (*Graphics User Interface*) chẳng hạn như `TH1Editor` hay `TH2Editor` nhằm giúp cho việc hiệu chỉnh hình vẽ được thuận tiện hơn. Để khởi động phần Graphics Editor ta có thể nhấp vào mục *View* → *Editor* như trong Hình 3.14. Để hiệu chỉnh cho đối tượng nào trong hình vẽ (canvas, pad, axis,...) ta chỉ cần nhấp vào đối tượng đó trên hình. Các hiệu chỉnh bằng Graphics Editor cũng tương đương với các hiệu chỉnh bằng câu lệnh.



Hình 3.14: Cách chọn Graphics Editor

Ngoài ra, thay vì mở Graph Editor trực tiếp, ta có thể hiệu chỉnh hình vẽ bằng cách nhấp chuột

phải vào các đối tượng cần chỉnh và chọn các lệnh tương ứng như trong Hình 3.15.



Hình 3.15: Cách chỉnh sửa thứ 2

CHƯƠNG 4

HÀM

Trong toán học, hàm số hay gọi tắt là hàm (*function*) được dùng để biểu diễn mối quan hệ giữa một tập hợp các số liệu đầu vào với một tập hợp các số liệu đầu ra mà trong đó mỗi số liệu đầu vào chỉ tương ứng với duy nhất một số liệu đầu ra.

Việc làm khớp histogram trong ROOT có thể được thực hiện theo hai phương pháp chính là sử dụng phương thức `TH1::Fit()` và sử dụng bảng làm khớp (*Fit Panel*, trong đó phương thức `TH1::Fit()` được sử dụng chủ yếu.

4.1 Khai báo hàm

Trong ROOT, lớp hàm (*function class*) được kí hiệu là `TFd`[1], với `d` là số chiều của của hàm, ví dụ `TF1` là lớp hàm 1 chiều, `TF2` là lớp hàm 2 chiều,...

Cú pháp chung cho việc khai báo một hàm là

*<lớp hàm> *<tên hàm> = new <lớp hàm>("<tên hàm>", "<công thức>", <giá trị nhỏ nhất>, <giá trị lớn nhất>);*

4.1.1 Khai báo hàm không chứa tham số

Ta có thể khai báo công thức của một hàm theo 1 trong các cách dưới đây

Cách 1: tạo trực tiếp một hàm dựa vào các công thức C++ có sẵn

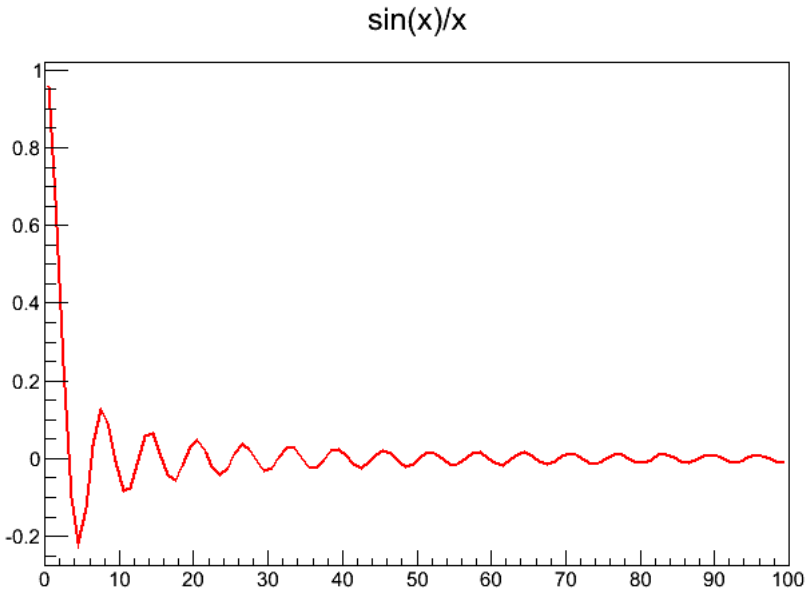
Ví dụ: *Khai báo hàm sin(x)/x (xem Hình 4.1)*

```
root [0] TF1 *f = new TF1("f", "sin(x)/x", 0, 100)
root [1] f->Draw()
```

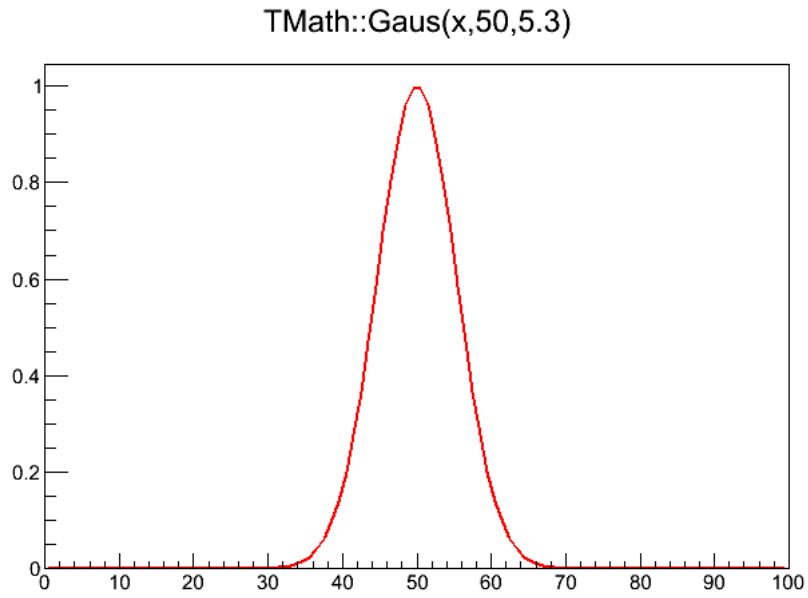
Cách 2: tạo trực tiếp một hàm dựa vào lớp `TMath`

Ví dụ: *Khai báo hàm Gaussian có* $\bar{x} = 50$ *và* $\sigma = 5.3$ *(xem Hình 4.2)*

```
root [0] TF1 *f = new TF1("f", "TMath::Gaus(x,50,5.3)", 0, 200)
root [1] f->Draw()
```



Hình 4.1: Đồ thị của hàm sin(x)/x



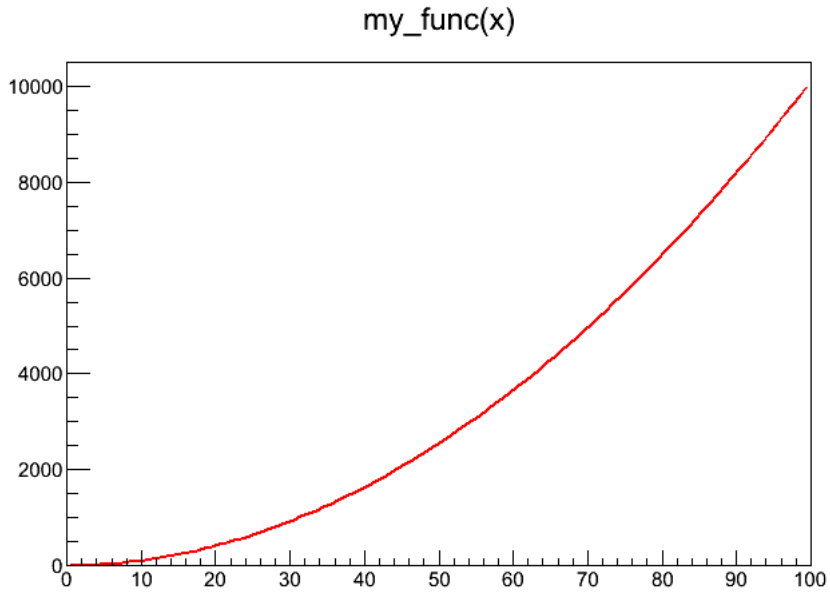
Hình 4.2: Đồ thị của hàm Gauss(50,5.3)

Cách 3: tự định nghĩa hàm riêng

Ví dụ: *Khai báo hàm* $x^2 + x + 1 = 0$ *(xem Hình 4.3)*

```
Double_t my_func (Double_t x) {
      return (x*x+x+1);
}
root [0] TF1 *f = new TF1("f", "my_func(x)", 0, 100)
root [1] f->Draw()
```

[1] Xem thêm tại http://root.cern.ch/root/html/TF1.html



Hình 4.3: Đồ thị của hàm my_func(x)

4.1.2 Khai báo hàm có chứa tham số

Trong trường hợp khai báo các hàm có tham số, kí hiệu của các tham số được đặt trong cặp ngoặc vuông []. Giá trị của các tham số có thể được gán thông qua phương thức `TF1::SetParameter()` cho từng tham số hoặc `TF1::SetParameters()` cho nhiều tham số cùng lúc.

Ta có thể khai báo công thức của một hàm có tham số theo 1 trong các cách

Cách 1: Sử dụng các hàm của C++

Ví dụ: *Khai báo hàm a cos(x) + b (xem Hình 4.4)*

```
root [0] TF1 *f = new TF1("f", "[0]*cos(x) + [1]", -10, 10)
root [1] f->SetParameter(0,5)    // a = 5
root [2] f->SetParameter(1,3)    // b = 3
root [3] f->Draw()
root [3] f->SetParameters(10,2)  // a = 10, b = 2
```

Cách 2: Sử dụng các hàm của thư viện `TMath`

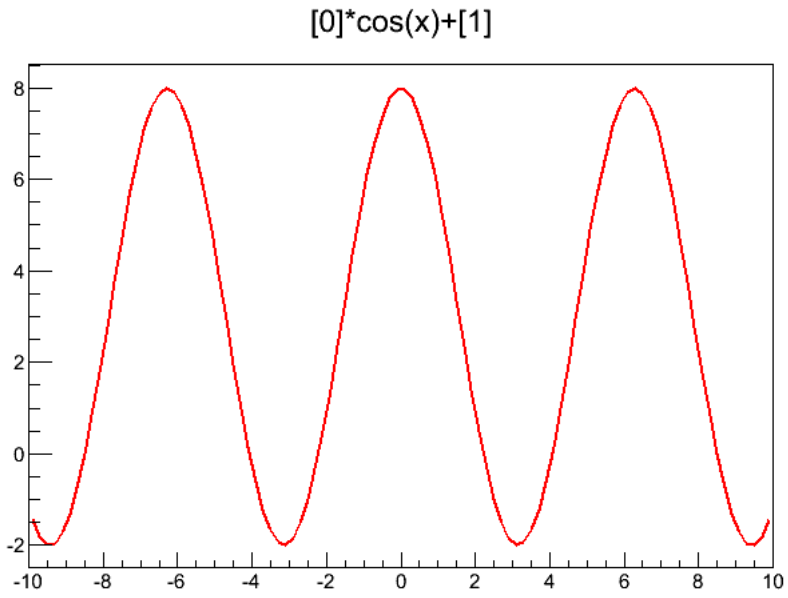
Ví dụ: *Khai báo hàm a cos(x) + b (xem Hình 4.5)*

```
root [0] TF1 *fb2 = new TF1("fa3","TMath::Landau(x,[0],[1],0)",-5,10)
root [1] fb2->SetParameters(0.2,1.3)
root [2] fb2->Draw()
```

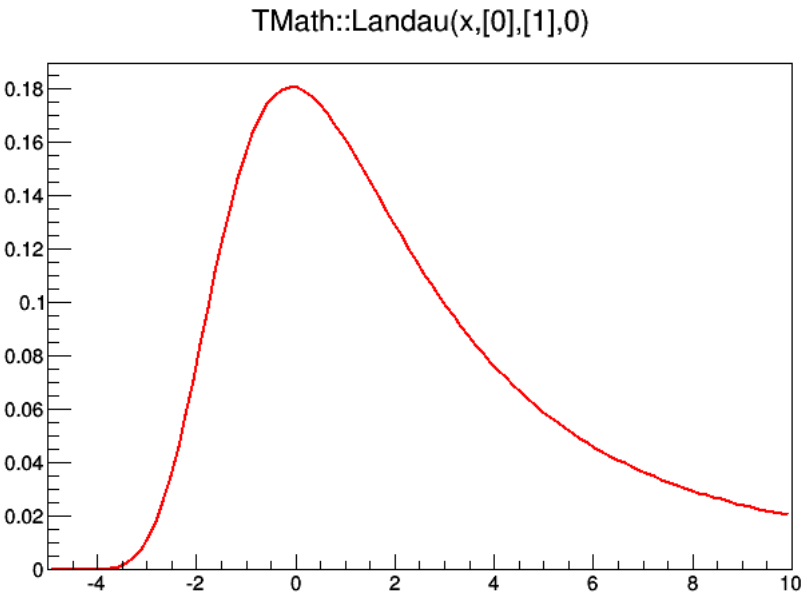
Cách 3: Sử dụng các hàm tự xây dựng

Ví dụ: *Khai báo hàm a cos(x) + b (xem Hình 4.6)*

```
Double_t myfunction(Double_t *x, Double_t *par) {
  Float_t xx =x[0];
  Double_t f = TMath::Abs(par[0]*sin(par[1]*xx)/xx);
  return f;
}
void myfunc() {
  TF1 *f1 = new TF1("myfunc",myfunction,0,10,2);
  f1->SetParameters(2,1);
```



Hình 4.4: Đồ thị của hàm a cos(x) + b

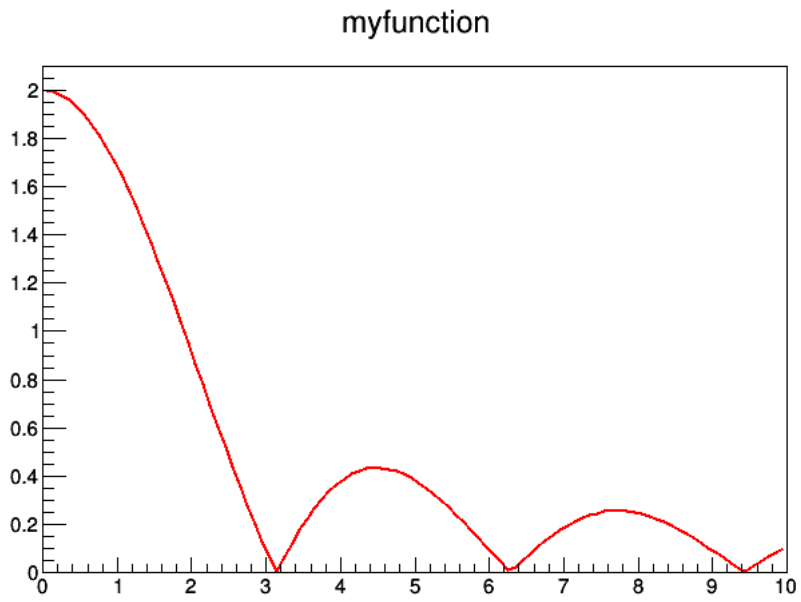


Hình 4.5: Đồ thị của hàm Landau

```
  f1->SetParNames("constant","coefficient");
  f1->Draw();
}
```

Trong cách này, hàm `myfunction()` có hai tham số:

- `x` là một con trỏ mảng, mỗi phần tử sử dụng cho 1 chiều, ví dụ ta làm khớp với histogram 1D thì chỉ cần sử dụng `x[0]`, 2D thì cần `x[0]` và `x[1]`. Mặc dù chỉ có tối đa 3 chiều cho histogram nhưng phương pháp này vẫn có thể áp dụng cho các đối tượng có số chiều lớn hơn 3.
- `par` cũng là một con trỏ mảng, mảng này sẽ chứa các giá trị tham số khi được gọi bởi hàm cần làm khớp.



Hình 4.6: Đồ thị của hàm myfunction

4.1.3 Các cách khai báo khác

Sử dụng các phương thức của lớp C++

```
class  MyFunction {
public:
   double Evaluate() (double *x, double *p) { /* noi dung */ }
};

void main()
{
    ....
   MyFunction * fptr = new MyFunction(....);  // tao doi tuong thuoc lop
      MyFunction
   TF1 * f = new TF1("f", fptr, &MyFunction::Evaluate, 0, 1, npar, "
      MyFunction", "Evaluate"); // tao doi tuong thuoc lop TF1
}
```

Sử dụng các functor (đối tượng hàm)[2]

```
class  MyFunctionObject {
public:
  double operator() (double *x, double *p) { /* noi dung */ }
};

void main()
{
  .....
  MyFunctionObject * fobj = new MyFunctionObject(....); // tao doi tuong
     ham
  TF1 * f = new TF1("f", fobj, 0, 1, npar, "MyFunctionObject"); // tao doi
      tuong thuoc lop TF1
}
```

[2]Xem thêm trong phần 4.6.3 của tài liệu *"Ngôn ngữ lập trình C++ (chuẩn 2011)"* (http://goo.gl/ng8KyS)

4.2 Làm khớp histogram theo hàm

4.2.1 Phương thức làm khớp

Để làm khớp histogram, ta có thể dùng phương thức `TH1::Fit()`[3] với cú pháp sau

<tên histogram>->Fit(<tên hàm>, <các option>,...)

Đối với một số hàm xây dựng sẵn, ta chỉ cần khai báo từ khóa mà không cần phải khai báo hàm:

- `"gaus"`: hàm Gaussian với 3 tham số, f(x) = [0]*exp(-0.5*((x-[1])/[2])**2)
- `"landau"`: hàm Landau
- `"expo"`: hàm e mũ với 2 tham số, f(x) = exp([0]+[1]*x)
- `"polN"`: hàm đa thức bậc N, f(x) = [0] + [1]*x + [2]*x**2 + ...

Một số tùy chỉnh thường được sử dụng khi làm khớp:

- `"W"`: chuyển tất cả trọng số của các bin khác 0 về 1.
- `"WW"`: chuyển tất cả trọng số của các bin về 1.
- `"L"`: sử dụng phương pháp log likelihood.
- `"U"`: sử dụng phương pháp làm khớp do người dùng tự định nghĩa (thông qua `SetFCN`).
- `"V"`: sử dụng verbose mode.
- `"B"`: sử dụng tham số thiết lập bởi người dùng đối với các hàm xây dựng sẵn.
- `"R"`: sử dụng khoảng giá trị được xác định bởi hàm làm khớp.
- `"E"`: sử dụng kĩ thuật Minos trong ước lượng sai số.
- `"M"`: cải tiến kết quả làm khớp (gọi lệnh `IMPROVE` trong `TMinuit`).
- `"C"`: không tính chisquare trong trường hợp làm khớp tuyến tính.
- `"0"`: không vẽ kết quả làm khớp.
- `"+"`: thêm hàm vừa mới làm khớp vào trong danh sách các hàm được làm khớp (theo mặc định các hàm làm khớp cũ sẽ bị xóa khi có hàm làm khớp mới).

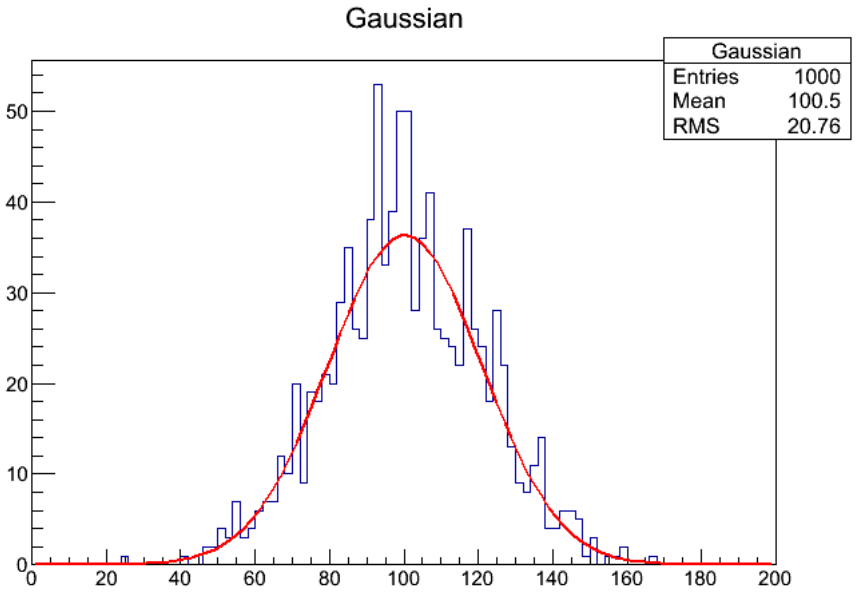
Ví dụ: *Gieo số ngẫu nhiên phân bố dạng Gauss và làm khớp lại theo dạng này (xem Hình 4.7)*

```
{
  TH1F *h = new TH1F("h","Gaussian",100,0,200);
  Float_t mean = 100, sigma = 20;
  for (Int_t i = 0; i < 1000; i++) {  // gieo ngau nhien 1000 lan
      h->Fill(gRandom->Gaus(mean,sigma));  // gieo ngau nhien theo phan bo
          Gauss
  }

  TF1 *f = new TF1("f", "[0]*TMath::Gaus(x,[1],[2])", 0, 200);
  f->SetParameters(10,10,10);
  h->Fit("f");  // ta co the su dung f->Fit("gaus") thay the
  h->Draw();  // ve histogram h
  f->Draw("same");  // ve ham lam khop f tren cung do thi
}
```

Kết quả làm khớp như sau:

[3] Xem thêm tại http://root.cern.ch/root/HowtoFit.html và http://root.cern.ch/root/html/TH1.html

```
FCN=65.1586 FROM MIGRAD    STATUS=CONVERGED      358 CALLS          359 TOTAL
                    EDM=4.68058e-009    STRATEGY= 1      ERROR MATRIX ACCURATE

 EXT PARAMETER                                          STEP         FIRST
 NO.   NAME      VALUE            ERROR          SIZE      DERIVATIVE
  1   p0           3.63132e+001   1.52623e+000   4.71709e-003   6.14789e-005
  2   p1           1.00262e+002   6.94998e-001   2.76013e-003  -9.82112e-007
  3   p2           2.06826e+001   5.76078e-001   1.78036e-003   2.12470e-004
```



Hình 4.7: Đồ thị làm khớp hàm dạng Gauss

4.2.2 Các thiết lập cho tham số

Như đã được đề cập đến trong phần trên, để thiết lập giá trị ban đầu cho tham số ta có thể dùng các phương thức `TF1::SetParameter()` cho từng tham số hoặc `TF1::SetParameters()` cho tất cả các tham số. Đối với các hàm dựng sẵn trong ROOT như `gaus`, `landau`, `polN` hay `exp` việc thiết lập giá trị ban đầu cho tham số là tự động.

Trong trường hợp muốn cố định 1 tham số trong quá trình làm khớp, ta có thể sử dụng phương thức `TF1::FixParameter`, ví dụ như

```
f->FixParameter(0,1.);
```

Đối với các hàm dựng sẵn ta có thể cố định tham số bằng cách sử dụng tùy chọn `"B"` khi làm khớp.

Ngoài ra ta cũng có thể xác định khoảng giới hạn cho tham số thông qua phương thức `TF1::SetParLimits()`

```
f->SetParLimits(0,-1.,1.);
```

Để xác định khoảng làm khớp

```
h->Fit("f","","",40,160);
```

Các tham số làm khớp có thể được truy xuất qua các phương thức sau

- `GetFunction("f")`: lấy hàm đã được làm khớp

- `GetChisquare()`: giá trị χ^2 của việc làm khớp
- `GetParameter(i)`: giá trị của tham số thứ i
- `GetParError(i)`: sai số của tham số thứ i

Ví dụ:

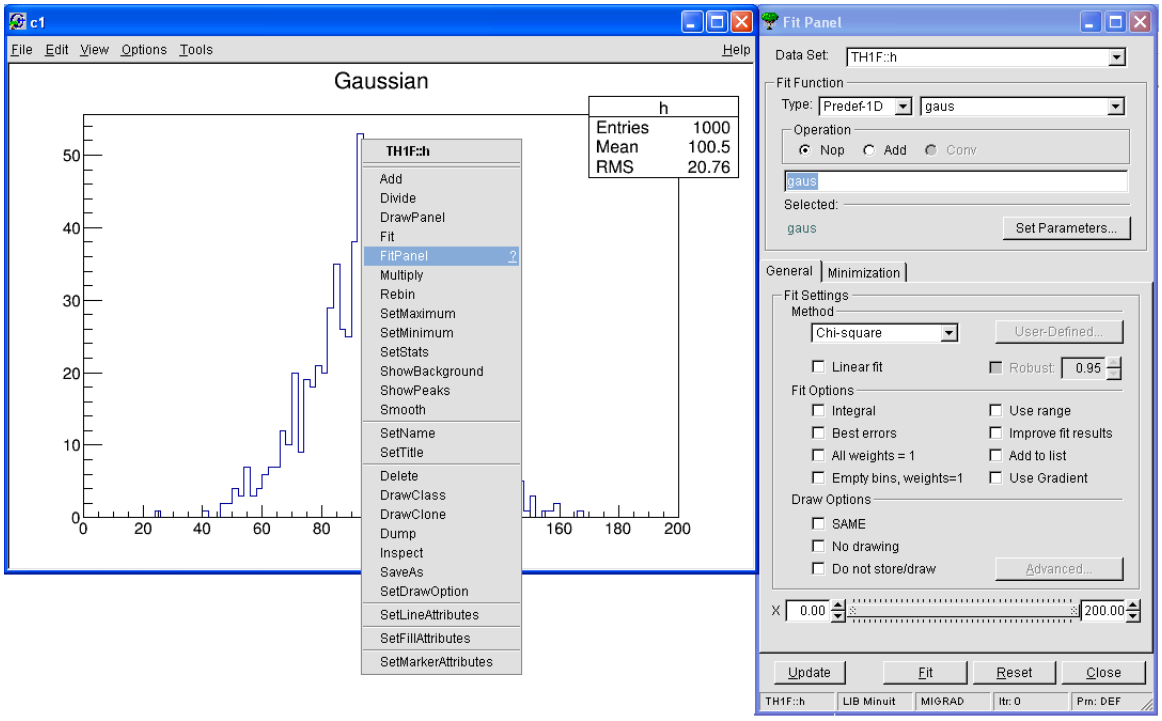
```
root [1] f->GetChisquare()
(const Double_t)6.51585976928294170e+001
root [2] f->GetParameter(1)
(const Double_t)1.00261634283092450e+002
root [3] f->GetParError(1)
(const Double_t)6.94997641719103610e-001
```

Hàm làm khớp cũng có thể được truy cập qua phương thức `TH1::GetFunction()`

```
TF1 *myfunc = h->GetFunction("f");
```

4.2.3 Làm khớp với Fit Panel

Để làm khớp với Fit Panel, ta nhấp chuột phải vào bên trong hình vẽ histogram và lựa chọn dòng FitPanel trong menu xuất hiện (Hình 4.8).



Hình 4.8: Fit Panel làm khớp hàm dạng Gauss

4.2.4 Làm khớp với nhiều hàm

Để thuận tiện cho việc tìm hiểu cách thức làm khớp với nhiều hàm cùng lúc, ta sẽ xem xét một ví dụ cụ thể. Giả sử ta muốn tiến hành việc làm khớp bộ số liệu dưới đây với hai hàm, một hàm mô tả đỉnh Gaussian và hàm còn lại mô tả phông nền đa thức bậc 2

```
  const int nBins = 50;
  Double_t data[nBins] = {15,22,25,18,22,22,16,21,19,
                          20,34,41,45,49,72,77,91,89,
```

```
                                90,94,96,94,84,80,64,61,52,
                                38,39,31,32,24,25,25,30,20,
                                27,20,20,14,16,18,12,23,
                                8,22,17,12,23,12};
```

Đầu tiên ta sẽ tạo một histogram để chứa bộ số liệu này

```
  TH1F *h = new TH1F("h","Gaussian + background",50,0,nBins);

  for(int i=0; i < nBins;  i++) {
    h->SetBinContent(i+1,data[i]);
  }
  h->Sumw2();
```

Sau đó là tiến hành xây dựng hai hàm phông nền và đỉnh để phục vụ cho việc làm khớp, các bạn có thể sử dụng các hàm dựng sẵn "gaus" hay "plo2" nhưng trong ví dụ này tôi sử dụng các hàm định nghĩa bởi người dùng

```
Double_t background(Double_t *x, Double_t *par) {
  return par[0] + par[1]*x[0] + par[2]*x[0]*x[0];
}

Double_t Gaussian(Double_t *x, Double_t *par) {
  return (par[0]*TMath::Gaus(x[0],par[1],par[2]));
}
```

Hàm làm khớp sẽ là tổng của hai hàm phông nền và đỉnh

```
Double_t function(Double_t *x, Double_t *par) {
  return background(x,par) + Gaussian(x,&par[3]);
}
```

Lưu ý rằng 3 phần tử đầu trong mảng `par` (`par[0]`, `par[1]`, `par[2]`) sẽ được dùng để mô tả hàm phông nền, do đó hàm đỉnh Gauss sẽ sử dụng từ phần tử thứ 4 trở đi (`par[3]`, `par[4]`, `par[5]`). Đó là lý do vì sao ta truyền tham chiếu cho hàm `Gaussian()` với chỉ số là 3 (`Gaussian(x,&par[3])`).

Để thực hiện việc làm khớp, ta cần các đối tượng thuộc lớp `TF1` tương ứng

```
  TF1 *f = new TF1("f",function,0,nBins,6);
```

Ta sẽ tiến hành việc làm khớp như sau

```
  f->SetParameter(4,20);  // uoc luong vi tri dinh
  f->SetParameter(5,10);  // uoc luong be rong dinh
  h->Fit("f");
```

Trong trường hợp muốn vẽ riêng các hàm phông nền và đỉnh sau khi đã làm khớp, ta cần khai báo các đối tượng `TF1` cho từng thành phần

```
  TF1 *bkg = new TF1("bkg",background,0,50,3);
  TF1 *peak = new TF1("peak",Gaussian,0,50,3);
```

Và đưa các tham số sau khi làm khớp từ `f` (thông qua phương thức `TH1::GetParameters()`) vào trong các đối tượng này (thông qua phương thức `TH1::SetParameters()`). Lưu ý rằng các tham số cho hàm đỉnh bắt đầu từ chỉ số 3.

```
  Double_t par[6];
  f->GetParameters(par);
  bkg->SetParameters(par);
  peak->SetParameters(&par[3]);
```

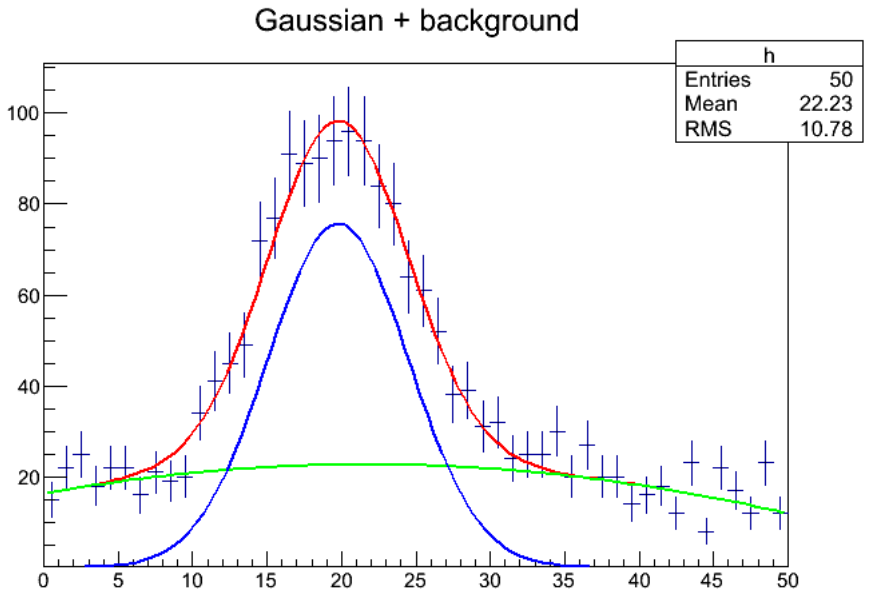
Ta có thể dùng tùy chỉnh "same" để vẽ trên cùng 1 đồ thị

```
  bkg->Draw("same");
  peak->Draw("same");
```

Đoạn mã đầy đủ của ví dụ được cho ở dưới đây, kết quả làm khớp được trình bày trong Hình 4.9.

```
// Ham mo ta phong nen dang da thuc bac 2
Double_t background(Double_t *x, Double_t *par) {
  return par[0] + par[1]*x[0] + par[2]*x[0]*x[0];
}

// Ham mo ta dinh dang Gaussian
Double_t Gaussian(Double_t *x, Double_t *par) {
  return (par[0]*TMath::Gaus(x[0],par[1],par[2]));
}

// Ham lam khop bang tong cua ham dinh Gaussian va phong nen
Double_t function(Double_t *x, Double_t *par) {
  return background(x,par) + Gaussian(x,&par[3]);
}

// Thuc hien viec lam khop
void fitting() {
  const int nBins = 50;
  Double_t data[nBins] = {15,22,25,18,22,22,16,21,19,
                          20,34,41,45,49,72,77,91,89,
                          90,94,96,94,84,80,64,61,52,
                          38,39,31,32,24,25,25,30,20,
                          27,20,20,14,16,18,12,23,
                          8,22,17,12,23,12};
  TH1F *h = new TH1F("h","Gaussian + background",50,0,nBins);

  for(int i=0; i < nBins;  i++) {
    h->SetBinContent(i+1,data[i]);
  }
  h->Sumw2();

  // tao ham TF1 tu 0 den nBins, co 6 tham so (3 cho phong nen, 3 cho dinh
     )
  TF1 *f = new TF1("f",function,0,nBins,6);

  // dua tham so dau vao
  f->SetParameter(4,20);  // uoc luong vi tri dinh
  f->SetParameter(5,10);  // uoc luong be rong dinh
  h->Fit("f");

  // khao bao ham rieng cho phong nen va dinh
  TF1 *bkg = new TF1("bkg",background,0,50,3);
  bkg->SetLineColor(3);
  TF1 *peak = new TF1("peak",Gaussian,0,50,3);
  peak->SetLineColor(4);
  Double_t par[6];

  // dua cac tham so vao ham phong nen va dinh sau do ve cac ham nay
  f->GetParameters(par);
  bkg->SetParameters(par);
  bkg->Draw("same");
  peak->SetParameters(&par[3]);
  peak->Draw("same");
```

```
}
```



Hình 4.9: Làm khớp số liệu với dạng đỉnh Gaussian và phông nền đa thức bậc 2

4.2.5 Làm khớp với nhiều khoảng giá trị

Việc làm khớp số liệu trên nhiều khoảng giá trị cũng tương tự như việc làm khớp với nhiều hàm nhưng thay vì tất cả các hàm đều được xác định trên cùng một khoảng thì ta sẽ phân chia các khoảng giá trị khác nhau cho mỗi hàm.

Giả sử ta cần làm khớp bộ số liệu sau

```
const Int_t np = 49;
Float_t x[np] = {1.913521, 1.953769, 2.347435, 2.883654, 3.493567,
                 4.047560, 4.337210, 4.364347, 4.563004, 5.054247,
                 5.194183, 5.380521, 5.303213, 5.384578, 5.563983,
                 5.728500, 5.685752, 5.080029, 4.251809, 3.372246,
                 2.207432, 1.227541, 0.8597788,0.8220503,0.8046592,
                 0.7684097,0.7469761,0.8019787,0.8362375,0.8744895,
                 0.9143721,0.9462768,0.9285364,0.8954604,0.8410891,
                 0.7853871,0.7100883,0.6938808,0.7363682,0.7032954,
                 0.6029015,0.5600163,0.7477068,1.188785, 1.938228,
                 2.602717, 3.472962, 4.465014, 5.177035};

TH1F *h = new TH1F("h","Example of several fits in subranges",np,85,134)
   ;
h->SetMaximum(7);
```

Giả sử như ta muốn làm khớp trên 3 khoảng giá trị, ta sẽ xây dựng 3 hàm tương ứng với mỗi khoảng

```
Double_t par[9];
TF1 *g1    = new TF1("g1","gaus",85,95);
TF1 *g2    = new TF1("g2","gaus",98,108);
TF1 *g3    = new TF1("g3","gaus",110,121);
TF1 *total = new TF1("total","gaus(0)+gaus(3)+gaus(6)",85,125);
```

Có tất cả 9 tham số và chúng ta cũng tạo một hàm `total` là tổng của tất cả 3 hàm trên.

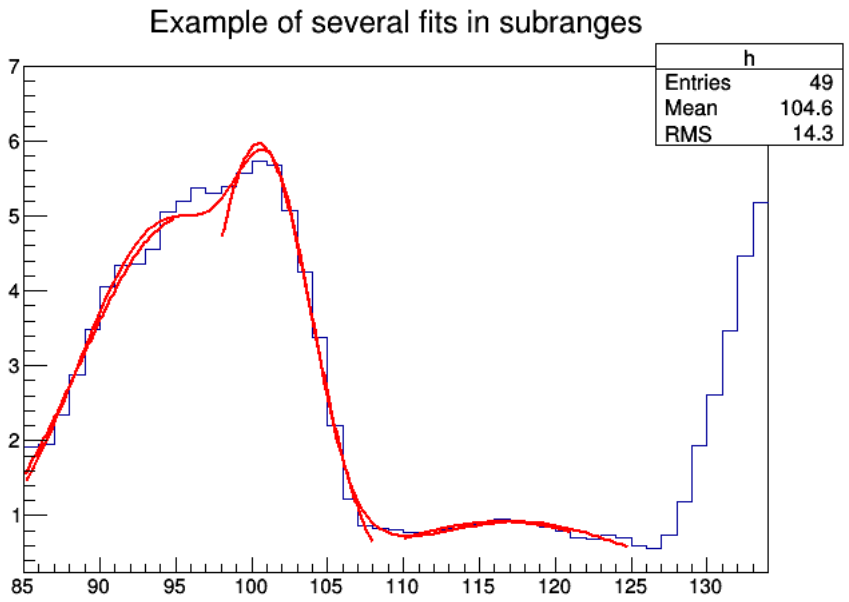
Ta sẽ bắt đầu tiến hành làm khớp trên từng khoảng giá trị

```
h->Fit(g1,"R");
h->Fit(g2,"R+");
h->Fit(g3,"R+");
```

Tùy chỉnh "R" sẽ báo cho chương trình biết là ta muốn làm khớp trên các khoảng giá trị tương ứng với khoảng giá trị được khai báo trong các hàm. Tùy chỉnh `"+"` sẽ thêm hàm mới được làm khớp mà không bỏ đi hàm cũ vừa được làm khớp.

Sau khi thu được các tham số từ việc làm khớp trên mỗi khoảng, ta sẽ đưa các tham số này vào hàm `total` như là các giá trị ước lượng ban đầu và làm khớp lại một lần nữa trên toàn khoảng giá trị, kết quả làm khớp được cho như trong Hình 4.10

```
g1->GetParameters(&par[0]);
g2->GetParameters(&par[3]);
g3->GetParameters(&par[6]);
total->SetParameters(par);
h->Fit(total,"R+");
```



Hình 4.10: Làm khớp số liệu với dạng Gaussian trên nhiều khoảng giá trị

4.3 Làm khớp với Minuit

4.3.1 Minuit

`Minuit`[4] là một gói làm khớp bằng phương pháp cực tiểu hóa hàm mục tiêu (*objective function*) nằm trong PACKLIB, được viết đầu tiên bằng ngôn ngữ lập trình Fortran bởi Fred James và được chuyển sang lớp `TMinuit`[5] bằng ngôn ngữ C++ bởi Réne Brun.

Để làm khớp bằng `TMinuit`, người dùng cần xây dựng hàm mục tiêu theo các tham số cần làm khớp và tiến hành cực tiểu hóa hàm này, từ đó thu được các tham số làm khớp. Các bước tiến hành làm khớp sử dụng `TMinuit` như sau

[4] Xem thêm trong tài liệu ***"Hướng dẫn sử dụng Minuit"*** (http://goo.gl/T7TOvy)
[5] Chi tiết về lớp này có thể được xem tại http://root.cern.ch/root/html/TMinuit.html

- Xây dựng hàm mục tiêu để cực tiểu hóa `FCN` (hàm này có thể là *chi square*, *likelihood*,...).
- Khai báo bộ số liệu được sử dụng để làm khớp.
- Khai báo các tham số và khoảng làm khớp, cùng với các kiểu làm khớp
  - `SIMPLEX`: tìm cực tiểu địa phương, ưu điểm của phương pháp này là tốc độ nhanh, tuy nhiên độ chính xác không cao lắm.
  - `MIGRAD`: làm khớp hàm địa phương sử dụng phương pháp Davidson-Fletcher-Powell cải tiến.
  - `HESSE`: tính ma trận đạo hàm bậc hai của hàm `FCN` sử dụng phương pháp vi phân hữu hạn (*finite difference method*), thường được sử dụng để tối ưu kết quả thu được từ `MIGRAD`.
  - `MINOS`: phân tích sai số (bất đối xứng), thông thường ta hay sử dụng ít nhất hai phương pháp (`HESSE` được sử dụng ngay sau `MIGRAD`). Còn `MINOS` được sử dụng để tối ưu sai số của các giá trị làm khớp.
- Trả về kết quả làm khớp (cùng với các kiểm tra thống kê).

Một số lệnh thông dụng trong `Minuit` gồm có

- **CALl fcn <iflag>** gọi hàm `FCN` với `IFLAG = <iflag>`
- **CLEar** xóa hết tất cả các tên và giá trị của tham số
- **FIX <parno> [parno] . . .** giá trị của tham số `parno` sẽ được giữ nguyên (hằng số) trong quá trình tiến hành làm khớp
- **SIMplex [maxcalls] [tolerance]** tìm cực tiểu bằng phương pháp Simplex, đối số `maxcalls` khai báo số lần tối đa gọi hàm trong quá trình tính và đối số `tolerance` giới hạn sai lệch (dung sai) của cực tiểu tìm được
- **MIGrad [maxcalls] [tolerance]** tìm cực tiểu bằng phương pháp Migrad
- **MINOs [maxcalls] [parno] . . .** phân tích sai số bằng phương pháp Minos
- **HESse [maxcalls]** tính ma trận sai số (ma trận Hessian)
- **IMProve [maxcalls]** được gọi sau các lệnh làm khớp bằng cách tìm cực tiểu, nhiệm vụ của hàm này là tìm xem còn điểm cực tiểu địa phương nào nữa không
- **MNContour <par1> <par2> [npts]** tính các giá trị đường contour cho cùa hàm `FCN` tương ứng với 2 tham số `par1` và `par2`, các tham số khác của `FCN` được giữ nguyên tại vị trí hàm `FCN` đạt cực tiểu, `npts` là số điểm của contour được tính (mặc định là 20).
- **CONtour <par1> <par2> [devs] [ngrid]** vẽ đường contour ứng với 2 tham số `par1` và `par2`, `devs` (mặc định là 2) cho số lần độ lệch chuẩn của mỗi tham số, `ngrid` (mặc định là 25) cho độ phân giải của hình vẽ (số điểm theo cột và dòng)
- **SET PARameter <parno> <value>** gán giá trị cho tham số
- **SET LIMits [parno] [lolim] [uplim]** thiết lập giới hạn dưới (`lolim`) và trên (`uplim`) cho tham số
- **SET PRIntout <level>** thiết lập chế độ in kết quả, tùy theo giá trị của đối số `level`
  −1: không xuất kết quả
  0: xuất kết quả ít nhất (không có giá trị đầu vào, các kết quả trung gian)
  1: xuất kết quả mặc định
  2: xuất thêm các kết quả trung gian
  3: xuất tối đa các kết quả (trong quá trình tìm cực tiểu)

Để tìm hiểu cách thức làm khớp với TMinuit, ta xem xét một ví dụ làm khớp số liệu (5 điểm) theo hàm $z = f(x, y)$ dưới đây

```
Double_t func(float x,float y,Double_t *par) {
  Double_t value=( (par[0]*par[0])/(x*x)-1)/ ( par[1]+par[2]*y-par[3]*y*y)
      ;
  return value;
}
```

Việc đầu tiên cần làm là xây dựng hàm FCN (trong ví dụ này ta sử dụng *chi square*). ROOT sẽ tiến hành làm khớp số liệu thông qua việc cực tiểu hóa hàm FCN này, để thực hiện điều đó ta cần tính giá trị chisq để TMinuit tìm giá trị cực tiểu của biến này

```
Float_t z[5],x[5],y[5],errorz[5];

void fcn(Int_t &npar, Double_t *gin, Double_t &f, Double_t *par, Int_t
    iflag) {
  const Int_t nbins = 5; // so diem du lieu can lam khop
  Int_t i;
  // Tinh chisquare
  Double_t chisq = 0;
  Double_t delta;
  for (i=0;i<nbins; i++) {
    delta  = (z[i]-func(x[i],y[i],par))/errorz[i];
    chisq += delta*delta;
  }
  f = chisq;
}
```

Bước kế tiếp ta sẽ tiến hành xây dựng hàm chính để làm khớp bằng Minuit và khai báo đối tượng thuộc lớp TMinuit

```
void Ifit()
{
  TMinuit *gMinuit = new TMinuit(5); // toi da 5 tham so
  gMinuit->SetFCN(fcn);
}
```

Ta cũng khai báo một mảng arglist chứa đối số cho các phương thức của TMinuit và biến ierflg để chứa mã lỗi

```
  Double_t arglist[10];
  Int_t ierflg = 0;
```

Các lệnh trong Minuit được thực hiện thông qua phương thức TMinuit::mnexcm() chẳng hạn như

```
  gMinuit->mnexcm("SET ERR", arglist ,1,ierflg);
```

Để bắt đầu quá trình làm khớp, trước tiên ta cần thiết lập các tham số ban đầu cho Minuit

```
  static Double_t vstart[4] = {3, 1 , 0.1 , 0.01};
  static Double_t step[4] = {0.1 , 0.1 , 0.01 , 0.001};
  gMinuit->mnparm(0, "a1", vstart[0], step[0], 0,0,ierflg);
  gMinuit->mnparm(1, "a2", vstart[1], step[1], 0,0,ierflg);
  gMinuit->mnparm(2, "a3", vstart[2], step[2], 0,0,ierflg);
  gMinuit->mnparm(3, "a4", vstart[3], step[3], 0,0,ierflg);
```

Sau đó thực hiện bước tìm cực tiểu với MIGRAD

```
  arglist[0] = 500;
  arglist[1] = 1.;
  gMinuit->mnexcm("MIGRAD", arglist ,2,ierflg);
```

Và cuối cùng là xuất ra kết quả

```
  Double_t amin,edm,errdef;
  Int_t nvpar,nparx,icstat;
  gMinuit->mnstat(amin,edm,errdef,nvpar,nparx,icstat);
  gMinuit->mnprin(3,amin);
```

Dưới đây là đoạn mã đầy đủ của ví dụ làm khớp số liệu với lớp TMinuit

```
Float_t z[5],x[5],y[5],errorz[5];
//______________________________________________________________________
void fcn(Int_t &npar, Double_t *gin, Double_t &f, Double_t *par, Int_t
    iflag) {
  const Int_t nbins = 5;
  Int_t i;
  // Tinh chisquare
  Double_t chisq = 0;
  Double_t delta;
  for (i=0;i<nbins; i++) {
    delta  = (z[i]-func(x[i],y[i],par))/errorz[i];
    chisq += delta*delta;
  }
  f = chisq;
}
//______________________________________________________________________
Double_t func(float x,float y,Double_t *par) {
  Double_t value=( (par[0]*par[0])/(x*x)-1)/ ( par[1]+par[2]*y-par[3]*y*y)
      ;
  return value;
}
//______________________________________________________________________
void Ifit()
{
  // Gia tri z
        z[0]=1;
        z[1]=0.96;
        z[2]=0.89;
        z[3]=0.85;
        z[4]=0.78;
  // Sai so cua z
        Float_t error = 0.01;
        errorz[0]=error;
        errorz[1]=error;
        errorz[2]=error;
        errorz[3]=error;
        errorz[4]=error;
  // Gia tri x
        x[0]=1.5751;
        x[1]=1.5825;
        x[2]=1.6069;
        x[3]=1.6339;
        x[4]=1.6706;
  // Gia tri y
        y[0]=1.0642;
        y[1]=0.97685;
```

```
        y[2]=1.13168;
        y[3]=1.128654;
        y[4]=1.44016;

  // Khoi tao doi tuong thuoc TMinuit voi toi da 5 tham so
  TMinuit *gMinuit = new TMinuit(5);
  gMinuit->SetFCN(fcn);

  Double_t arglist[10];
  Int_t ierflg = 0;

  arglist[0] = 1;
  gMinuit->mnexcm("SET ERR", arglist ,1,ierflg);

  // Khai bao cac gia tri ban dau va buoc nhay cua cac tham so
  static Double_t vstart[4] = {3, 1 , 0.1 , 0.01};
  static Double_t step[4] = {0.1 , 0.1 , 0.01 , 0.001};
  gMinuit->mnparm(0, "a1", vstart[0], step[0], 0,0,ierflg);
  gMinuit->mnparm(1, "a2", vstart[1], step[1], 0,0,ierflg);
  gMinuit->mnparm(2, "a3", vstart[2], step[2], 0,0,ierflg);
  gMinuit->mnparm(3, "a4", vstart[3], step[3], 0,0,ierflg);

  // Thuc hien buoc tim cuc tieu
  arglist[0] = 500;
  arglist[1] = 1.;
  gMinuit->mnexcm("MIGRAD", arglist ,2,ierflg);

  // In ket qua
  Double_t amin,edm,errdef;
  Int_t nvpar,nparx,icstat;
  gMinuit->mnstat(amin,edm,errdef,nvpar,nparx,icstat);
  gMinuit->mnprin(3,amin);
}
```

4.3.2 Minuit2

`Minuit2` là phiên bản hướng đối tượng (*object oriented*) của `Minuit`, được viết bằng ngôn ngữ C++. Phiên bản này bao gồm tất cả các chức năng đã có của phiên bản trước, đồng thời có thêm các tính năng mới như khả năng thiết lập giới hạn một bên cho tham số. Ngoài ra, trong ROOT còn có các lớp như `TFitterMinuit` và `TFitterFumili` cũng sử dụng `Minuit2`[6].

Giao diện lập trình ứng dụng (*Application Programming Interface – API*) của `Minuit2` cũng tuân theo các quy ước của ROOT, với không gian tên là `ROOT::Minuit2`.

Kể từ phiên bản ROOT 5.17, một lớp mới đã được xây dựng `ROOT::Minuit2::Minuit2Minimizer` thông qua việc tích hợp thêm giao diện từ `ROOT::Math::Minimizer`.

Một số lớp thông dụng trong `Minuit2`:

`FCNBase`	lớp trừu tượng (*abstract class*) khai báo hàm mục tiêu
`MnUserParameters`	lớp API khai báo tham số
`MnSimplex`	lớp API tìm cực tiểu bằng phương pháp Simplex
`MnMigrad`	lớp API tìm cực tiểu bằng phương pháp Migrad
`MnMinos`	lớp API phân tích sai số Minos
`MnHesse`	lớp API tính ma trận sai số
`MnContours`	lớp API phân tích sai số 2 chiều
`FunctionMinimum`	chứa toàn bộ kết quả của việc cực tiểu hóa

[6] Thông tin chi tiết về `Minuit2` có thể được xem tại http://root.cern.ch/root/html/MATH_MINUIT2_Index.html

`MnMinosError`	chứa các giá trị sai số Minos
`ContoursError`	chứa các giá trị sai số 2 chiều

Dưới đây là một ví dụ cho việc làm khớp hàm Gaussian thông qua việc sử dụng `Minuit2`.

Đầu tiên ta sẽ đi xây dựng một lớp chứa các tham số và hàm cần làm khớp, trong trường hợp này là hàm Gaussian với 3 tham số *mean*, *sigma* và *constant*. Lớp được xây dựng có tên là *GaussFunction* và được khai báo trong file *GaussFunction.h*

```
#ifndef MN_GaussFunction_H_
#define MN_GaussFunction_H_

#define _USE_MATH_DEFINES
#include <math.h>

class GaussFunction {

public:

  GaussFunction(double mean, double sig, double constant) :
    theMean(mean), theSigma(sig), theConstant(constant) {}

  ~GaussFunction() {}

  double m() const {return theMean;}
  double s() const {return theSigma;}
  double c() const {return theConstant;}

  double operator()(double x) const {

    return c()*exp(-0.5*(x-m())*(x-m())/(s()*s()))/(sqrt(2.*M_PI)*s());
  }

private:

  double theMean;
  double theSigma;
  double theConstant;
};

#endif // MN_GaussFunction_H_
```

Tiếp theo ta cần xây dựng hàm mục tiêu `FCN`, hàm này được xây dựng thông qua lớp *GaussFcn* (là lớp dẫn xuất của `FCNBase`). Lớp này được khai báo thông qua hai file, file thứ nhất là *GaussFcn.h*

```
#ifndef MN_GaussFcn_H_
#define MN_GaussFcn_H_

#include "Minuit/FCNBase.h"

#include <vector>

class GaussFcn : public FCNBase {

public:

  GaussFcn(const std::vector<double>& meas,
         const std::vector<double>& pos,
         const std::vector<double>& mvar) : theMeasurements(meas),
                                            thePositions(pos),
```

```
                                        theMVariances(mvar),
                                        theErrorDef(1.) {}

  ~GaussFcn() {}

  virtual double up() const {return theErrorDef;}
  virtual double operator()(const std::vector<double>&) const;

  std::vector<double> measurements() const {return theMeasurements;}
  std::vector<double> positions() const {return thePositions;}
  std::vector<double> variances() const {return theMVariances;}

  void setErrorDef(double def) {theErrorDef = def;}

private:


  std::vector<double> theMeasurements;
  std::vector<double> thePositions;
  std::vector<double> theMVariances;
  double theErrorDef;
};

#endif //MN_GaussFcn_H_
```

Và file thứ hai là *GaussFcn.cpp*, hàm `FCN` ở đây là *chi squared*

```
#include "GaussFcn.h"
#include "GaussFunction.h"

#include <cassert>

double GaussFcn::operator()(const std::vector<double>& par) const {

  assert(par.size() == 3);
  GaussFunction gauss(par[0], par[1], par[2]);

  double chi2 = 0.;
  for(unsigned int n = 0; n < theMeasurements.size(); n++) {
    chi2 += ((gauss(thePositions[n]) - theMeasurements[n])*(gauss(
        thePositions[n]) - theMeasurements[n])/theMVariances[n]);
  }

  return chi2;
}
```

Cuối cùng là hàm `main()` được khai báo trong file *DemoGaussSim.cpp*

```
#include "GaussFcn.h"
#include "GaussDataGen.h"
#include "Minuit/FunctionMinimum.h"
#include "Minuit/MnUserParameterState.h"
#include "Minuit/MnPrint.h"
#include "Minuit/MnMigrad.h"
#include "Minuit/MnMinos.h"
#include "Minuit/MnContours.h"
#include "Minuit/MnPlot.h"
#include "Minuit/MinosError.h"
#include "Minuit/ContoursError.h"
```

```
#include <iostream>

int main() {

  // generate the data (100 data points)
  GaussDataGen gdg(100);

  std::vector<double> pos = gdg.positions();
  std::vector<double> meas = gdg.measurements();
  std::vector<double> var = gdg.variances();

  // create FCN function
  GaussFcn theFCN(meas, pos, var);

  // create initial starting values for parameters
  double x = 0.;
  double x2 = 0.;
  double norm = 0.;
  double dx = pos[1]-pos[0];
  double area = 0.;
  for(unsigned int i = 0; i < meas.size(); i++) {
    norm += meas[i];
    x += (meas[i]*pos[i]);
    x2 += (meas[i]*pos[i]*pos[i]);
    area += dx*meas[i];
  }
  double mean = x/norm;
  double rms2 = x2/norm - mean*mean;
  double rms = rms2 > 0. ? sqrt(rms2) : 1.;

  {
    // demonstrate minimal required interface for minimization
    // create Minuit parameters without names

    // starting values for parameters
    std::vector<double> init_par;
    init_par.push_back(mean);
    init_par.push_back(rms);
    init_par.push_back(area);

    // starting values for initial uncertainties
    std::vector<double> init_err;
    init_err.push_back(0.1);
    init_err.push_back(0.1);
    init_err.push_back(0.1);

    // create minimizer (default constructor)
    VariableMetricMinimizer theMinimizer;

    // minimize
    FunctionMinimum min = theMinimizer.minimize(theFCN, init_par, init_err
       );

    // output
    std::cout<<"minimum: "<<min<<std::endl;
  }

  {
    // demonstrate standard minimization using MIGRAD
```

```
  // create Minuit parameters with names
  MnUserParameters upar;
  upar.add("mean", mean, 0.1);
  upar.add("sigma", rms, 0.1);
  upar.add("area", area, 0.1);

  // create MIGRAD minimizer
  MnMigrad migrad(theFCN, upar);

  // minimize
  FunctionMinimum min = migrad();

  // output
  std::cout<<"minimum: "<<min<<std::endl;
}

{
  // demonstrate full interaction with parameters over subsequent
  // minimizations

  // create Minuit parameters with names
  MnUserParameters upar;
  upar.add("mean", mean, 0.1);
  upar.add("sigma", rms, 0.1);
  upar.add("area", area, 0.1);

  // access parameter by name to set limits...
  upar.setLimits("mean", mean-0.01, mean+0.01);

  // ... or access parameter by index
  upar.setLimits(1, rms-0.1, rms+0.1);

  // create Migrad minimizer
  MnMigrad migrad(theFCN, upar);

  // fix a parameter...
  migrad.fix("mean");

  // ... and minimize
  FunctionMinimum min = migrad();

  // output
  std::cout<<"minimum: "<<min<<std::endl;

  // release a parameter...
  migrad.release("mean");

  // ... and fix another one
  migrad.fix(1);

  // and minimize again
  FunctionMinimum min1 = migrad();

  // output
  std::cout<<"minimum1: "<<min1<<std::endl;

  // release the parameter...
  migrad.release(1);
```

```
  // ... and minimize with all three parameters (still with limits!)
  FunctionMinimum min2 = migrad();

  // output
  std::cout<<"minimum2: "<<min2<<std::endl;

  // remove all limits on parameters...
  migrad.removeLimits("mean");
  migrad.removeLimits("sigma");

  // ... and minimize again with all three parameters (now without
     limits!)
  FunctionMinimum min3 = migrad();

  // output
  std::cout<<"minimum3: "<<min3<<std::endl;
}

{
  // test single sided limits
  MnUserParameters upar;
  upar.add("mean", mean, 0.1);
  upar.add("sigma", rms-1., 0.1);
  upar.add("area", area, 0.1);

  // test lower limits
  upar.setLowerLimit("mean", mean-0.01);

  // test upper limits
  upar.setUpperLimit("sigma", rms-0.5);

  // create MIGRAD minimizer
  MnMigrad migrad(theFCN, upar);

  // ... and minimize
  FunctionMinimum min = migrad();
  std::cout<<"test lower limit minimim= "<<min<<std::endl;
}

{
  // demonstrate MINOS error analysis

  // create Minuit parameters with names
  MnUserParameters upar;
  upar.add("mean", mean, 0.1);
  upar.add("sigma", rms, 0.1);
  upar.add("area", area, 0.1);

  // create Migrad minimizer
  MnMigrad migrad(theFCN, upar);

  // minimize
  FunctionMinimum min = migrad();

  // create MINOS error factory
  MnMinos minos(theFCN, min);

  {
    // 1-sigma MINOS errors (minimal interface)
```

```
    std::pair<double,double> e0 = minos(0);
    std::pair<double,double> e1 = minos(1);
    std::pair<double,double> e2 = minos(2);

    // output
    std::cout<<"1-sigma minos errors: "<<std::endl;
    std::cout<<"par0: "<<min.userState().value("mean")<<" "<<e0.first<<"
        "<<e0.second<<std::endl;
    std::cout<<"par1: "<<min.userState().value(1)<<" "<<e1.first<<" "<<
       e1.second<<std::endl;
    std::cout<<"par2: "<<min.userState().value("area")<<" "<<e2.first<<"
        "<<e2.second<<std::endl;
  }

  {
    // 2-sigma MINOS errors (rich interface)
    theFCN.setErrorDef(4.);
    MinosError e0 = minos.minos(0);
    MinosError e1 = minos.minos(1);
    MinosError e2 = minos.minos(2);

    // output
    std::cout<<"2-sigma minos errors: "<<std::endl;
    std::cout<<e0<<std::endl;
    std::cout<<e1<<std::endl;
    std::cout<<e2<<std::endl;
  }
}

{
  // demonstrate how to use the CONTOURs

  // create Minuit parameters with names
  MnUserParameters upar;
  upar.add("mean", mean, 0.1);
  upar.add("sigma", rms, 0.1);
  upar.add("area", area, 0.1);

  // create Migrad minimizer
  MnMigrad migrad(theFCN, upar);

  // minimize
  FunctionMinimum min = migrad();

  // create contours factory with FCN and minimum
  MnContours contours(theFCN, min);

  //70% confidence level for 2 parameters contour around the minimum
  // (minimal interface)
  theFCN.setErrorDef(2.41);
  std::vector<std::pair<double,double> > cont = contours(0, 1, 20);

  //95% confidence level for 2 parameters contour
  // (rich interface)
  theFCN.setErrorDef(5.99);
  ContoursError cont4 = contours.contour(0, 1, 20);

  // plot the contours
  MnPlot plot;
```

```
    cont.insert(cont.end(), cont4().begin(), cont4().end());
    plot(min.userState().value("mean"), min.userState().value("sigma"),
        cont);

    // print out one contour
    std::cout<<cont4<<std::endl;
  }

  return 0;
}
```

CHƯƠNG 5

ĐỒ THỊ

Trong ROOT, tất cả các đối tượng đều được xây dựng từ các lớp dẫn xuất của `TObject`, lớp này có chứa 1 phương thức ảo là `Draw()`. Điều này có nghĩa là tất cả mọi đối tượng trong ROOT đều có thể được "vẽ".

Trong hai chương histogram và hàm trước đó, các bạn đã được làm quen với các cách thức để vẽ một đối tượng histogram hay hàm. Trong chương này, chúng ta sẽ tiếp tục đi sâu vào tìm hiểu những khái niệm liên quan tới việc vẽ đó, cũng như làm quen với một số lớp đồ thị thông dụng của ROOT.

5.1 Canvas và pad

Canvas là một cửa sổ mà hình ảnh có thể được hiển thị trên đó. Một canvas có thể được chia thành 1 hay nhiều khu vực vẽ, các khu vực vẽ này được gọi là **pad**. Các hình vẽ của đối tượng (vd: histogram hay hàm) được thực hiện trên các pad.

Trong ROOT, khi một đối tượng sử dụng phương thức `Draw()`, một canvas sẽ được kích hoạt để hiển thị hình vẽ của đối tượng đó. Nếu canvas không được khai báo trước khi vẽ hình, chương trình sẽ mặc định tạo ra canvas có tên là **c1**, và pad tương ứng cũng có cùng tên với canvas này. Một pad có thể nằm trong 1 canvas hoặc nằm trong 1 pad khác. Trong trường hợp có nhiều canvas (pad) được định nghĩa, chỉ có duy nhất 1 canvas (pad) hoạt động tại mỗi thời điểm.

5.1.1 TCanvas

`TCanvas` là lớp định nghĩa cho các canvas, cú pháp khai báo một canvas như sau

*TCanvas *<tên canvas> = new TCanvas("<tên canvas>", "<tiêu đề>", <kích thước trục X>, <kích thước trục Y>)*

hoặc

*TCanvas *<tên canvas> = new TCanvas("<tên canvas>", "<tiêu đề>", form)*

Trong đó `form` là tùy chọn cho kích thước của canvas

với `form` = 1 kích thước 700×500 tại (10,10)
`form` = 2 kích thước 500×500 tại (20,20)
`form` = 3 kích thước 500×500 tại (30,30)

`form` = 4	kích thước 500 × 500 tại (40,40)
`form` = 5	kích thước 500 × 500 tại (50,50)

5.1.2 TPad

`TPad` là lớp định nghĩa cho các pad, cú pháp khai báo một pad như sau

*TPad *<tên pad> = new TPad("<tên pad>", "<tiêu đề>", xlow, ylow, xup, yup, color, bordersize, bordermode)*

Trong đó:

`xlow`	vị trí của điểm dưới bên trái của pad đang khai báo
`ylow`	tọa độ y của điểm đó
`xup`	vị trí của điểm trên bên phải của pad đang khai báo
`yup`	tọa độ y của điểm đó
`color`	kí hiệu màu của pad
`bordersize`	kích thước biên của pad (tính theo pixel)
`bordermode`	hiệu ứng biên chìm xuống (= −1)
`bordermode`	không có hiệu ứng (= 0)
`bordermode`	hiệu ứng biên nổi lên (= 1)

5.1.3 Hiệu chỉnh canvas và pad

Một số phương thức thông dụng cho canvas và pad [1]

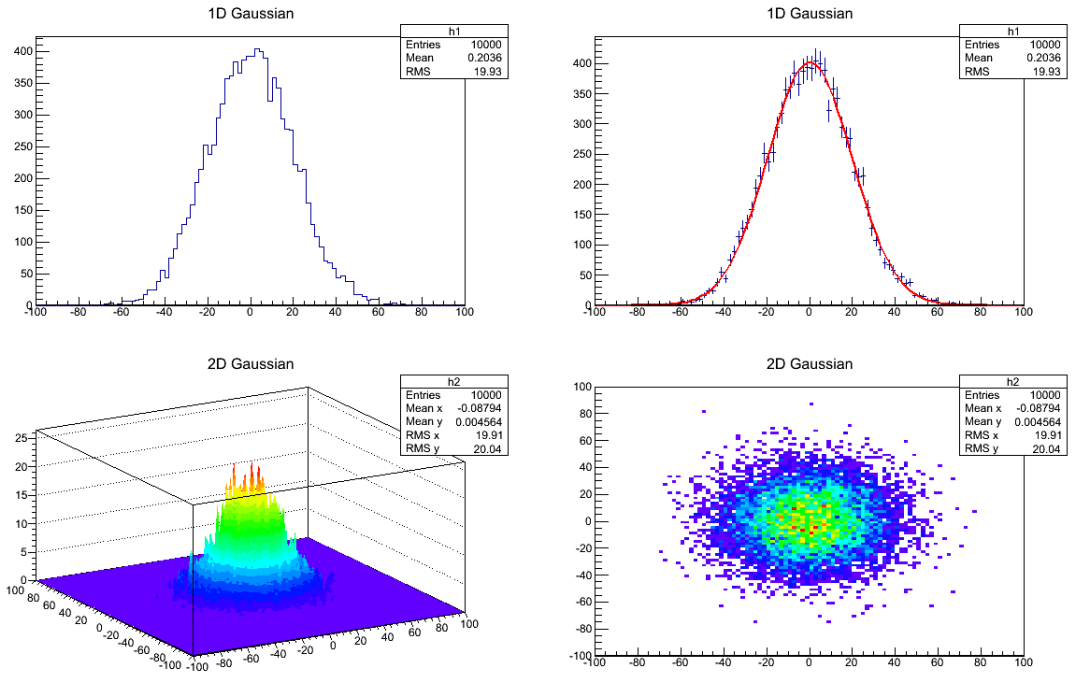
- `Divide(nx,ny)`: chia canvas/pad ra làm nhiều phần theo trục x và y, các pad nhỏ bên trong sẽ được đánh số từ 1 đến nx*ny
- `cd(i)`: chọn pad nhỏ thứ i, mặc định `cd() = cd(0)` tức là chọn chính canvas/pad đó
- `GetPad(i)`: chọn pad thứ i (trường hợp nằm trong một pad khác)
- `SetCanvasSize(w,h)`: chỉnh kích thước cho canvas
- `SetBorderSize()`: chỉnh kích thước cho biên
- `SetBorderMode()`: chỉnh kiểu cho biên
- `SetLogx()`, `SetLogy()`, `SetLogz()`: vẽ trục x,y,z theo thang log
- `SetGrid()`, `SetGridx()`, `SetGridy()`: vẽ lưới cho trục tọa độ
- `SetTicks()`, `SetTickx()`, `SetTicky()`: đánh dấu các khoảng chia cho trục tọa độ
- `Print("file")`: lưu nội dung trong hình vẽ ra file
- `SaveAs("file")`: tương tự `Print()`

Ví dụ 1: *(xem Hình 5.1)*

```
{
  TH1F *h1 = new TH1F("h1","1D Gaussian", 100, -100, 100);
  TH2F *h2 = new TH2F("h2","2D Gaussian", 100,-100, 100, 100, -100, 100);
  h1->Sumw2();
  h2->Sumw2();
  Float_t mean = 0, sigma = 20;
  for (Int_t i = 0; i < 10000; i++) {
      h1->Fill(gRandom->Gaus(mean,sigma));
      h2->Fill(gRandom->Gaus(mean,sigma),gRandom->Gaus(mean,sigma));
```

[1] Xem thêm tại http://root.cern.ch/root/html/TPad.html và http://root.cern.ch/root/html/TCanvas.html

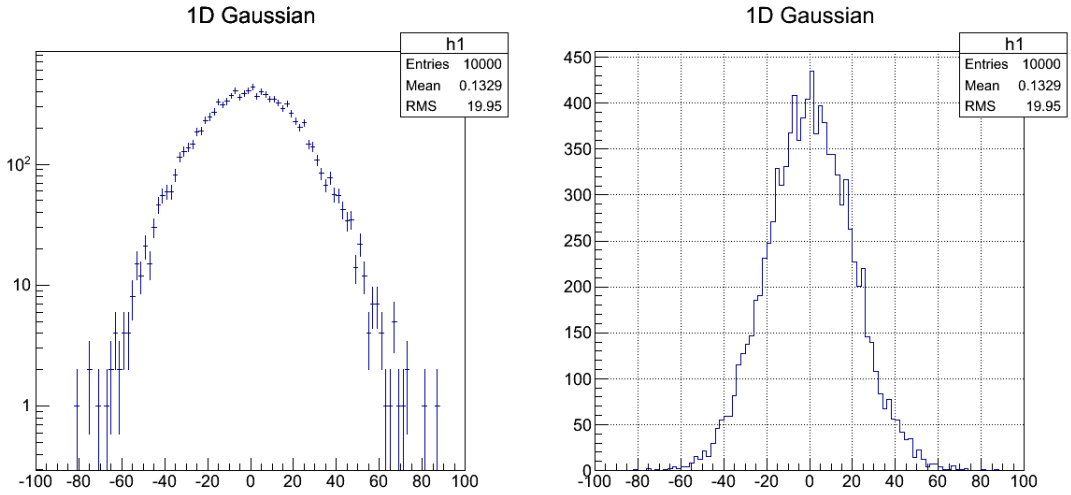
```
  }

  TCanvas *c = new TCanvas("c", "Gaussian", 1200, 800);  // tao canvas co
     kich thuoc 1200x800
  c->Divide(2,2);  // chia canvas thanh 4 pad nho
  c->cd(1);  // chon pad thu 1
  h1->Draw("histo");  // ve h1 theo dang histogram
  c->cd(2);  // chon pad thu 2
  h1->Fit("gaus");  // lam khop va ve h1
  c->cd(3);  // chon pad thu 3
  h2->Draw("surf2");  // ve h2 theo dang surface
  c->cd(4);  // chon pad thu 4
  h2->Draw("col");  // ve h2 theo dang color
}
```



Hình 5.1: Đồ thị phân bố ngẫu nhiên dạng Gauss 1 chiều và 2 chiều

Ví dụ 2: *(xem Hình 5.2)*

```
{
  TH1F *h1 = new TH1F("h1","1D Gaussian", 100, -100, 100);
  h1->Sumw2();
  Float_t mean = 0, sigma = 20;
  for (Int_t i = 0; i < 10000; i++) {
      h1->Fill(gRandom->Gaus(mean,sigma));
  }

  TCanvas *c = new TCanvas("c", "Gaussian", 600, 300);
  c->Divide(2,1);
  c->cd(1)->SetLogy();  // chon thang log cho truc y
  h1->Draw();
  c->cd(2)->SetGrid(); // ve luoi cho x va y
  h1->Draw("histo");
}
```



Hình 5.2: Đồ thị phân bố ngẫu nhiên dạng Gauss 1 chiều

Khi một đối tượng được vẽ, nó luôn kích hoạt một pad. Để thuận tiện cho việc xử lý trên pad đang được kích hoạt, ta sẽ sử dụng một con trỏ toàn cục (*global pointer*) `gPad`, đặc điểm của `gPad` là nó luôn trỏ tới pad đang hoạt động (*active pad*), ví dụ như

```
gPad->SetFillColor(38)  // thiet lap mau cho pad
gPad->SetLogx()  // chuyen thang do truc X thanh log
```

Do pad luôn chứa các đối tượng mà nó vẽ, nên ta có thể truy cập các đối tượng này thông qua phương thức `TPad::GetPrimitive()` (lưu ý con trỏ trả về luôn thuộc lớp `TObject`)

```
root[0] obj = gPad->GetPrimitive("myobjectname")
(class TObject*)0x1063cba8
```

Để chuyển đổi con trỏ sang đúng kiểu ta làm như sau

```
root[0] obj = (TH1*) gPad->GetPrimitive("myobjectname")
(class TH1*)0x1063cba8
```

Trong trường hợp chúng ta muốn liệt kê tất cả đối tượng có trong pad, chúng ta sử dụng phương thức `TPad::GetListOfPrimitives()`.

Trong nhiều trường hợp, một pad sẽ không thực hiện việc cập nhật những thay đổi từ người dùng, khi đó ta có thể dùng các phương thức sau để bắt pad cập nhật

```
pad1->Modified();  // cap nhat pad
c1->Update();  // cap nhat canvas sau khi cap nhat pad
```

5.2 Đồ thị

Đồ thị (*graph*) là một hình vẽ được tạo thành bởi hai mảng X và Y tương ứng nhau, tạo nên các điểm theo hai trục x và y. Trong ROOT có các lớp đồ thị như `TGraph`, `TGraphErrors`, `TGraphAsymmErrors` và `TMultiGraph`.

5.2.1 TGraph

`TGraph` là lớp hỗ trợ các đồ thị dạng tổng quát (các điểm không cách đều nhau), cú pháp khai báo đối tượng thuộc lớp `TGraph` như sau

<tên graph> = new TGraph(n, x, y)

Trong đó n là số điểm, x và y la hai mảng có n điểm tương ứng, hoặc

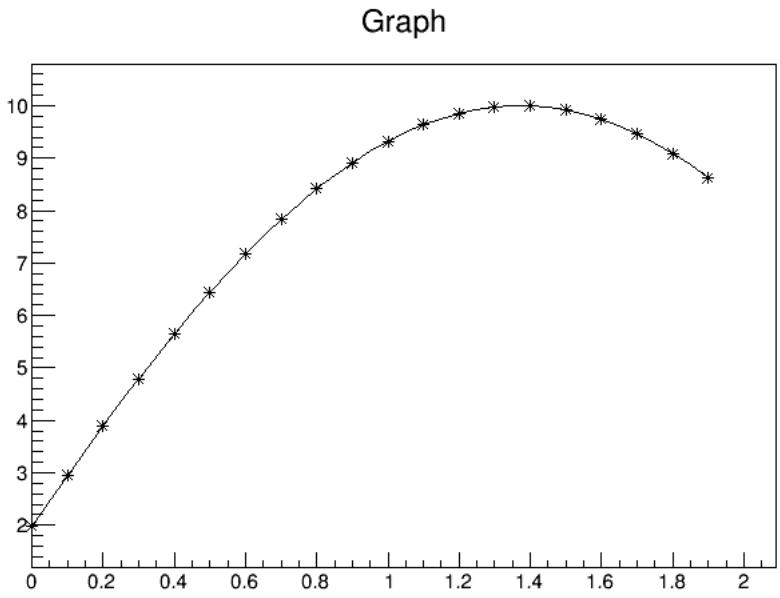
<tên graph> = new TGraph(h) với *h* là tên histogram

Để vẽ đồ thị ta sử dụng phương thức `TGraph::Draw()`, một số tùy chỉnh cho phương thức này

- `"L"`: vẽ đường nối giữa các điểm
- `"F"`: tô màu diện tích dưới đồ thị
- `"A"`: trục tọa độ được vẽ quanh đồ thị
- `"C"`: vẽ đường trơn qua các điểm
- `"*"`: đánh dấu * tại các điểm
- `"P"`: sử dụng loại marker hiện tại để đánh dấu tại các điểm
- `"B"`: vẽ biểu đồ dạng cột
- `"[]"`: chỉ vẽ đường sai số (áp dụng cho lớp lstTGraphAsymmErrors)
- `"1"`: ylow = rwymin

Ví dụ: *(xem Hình 5.3)*

```
{
   Int_t n = 20;
   Double_t x[n], y[n];
   for (Int_t i=0;i<n;i++) {
      x[i] = i*0.1;
      y[i] = 10*sin(x[i]+0.2);
   }
   TGraph *gr = new TGraph(n,x,y);
   gr->Draw("AC*");
}
```

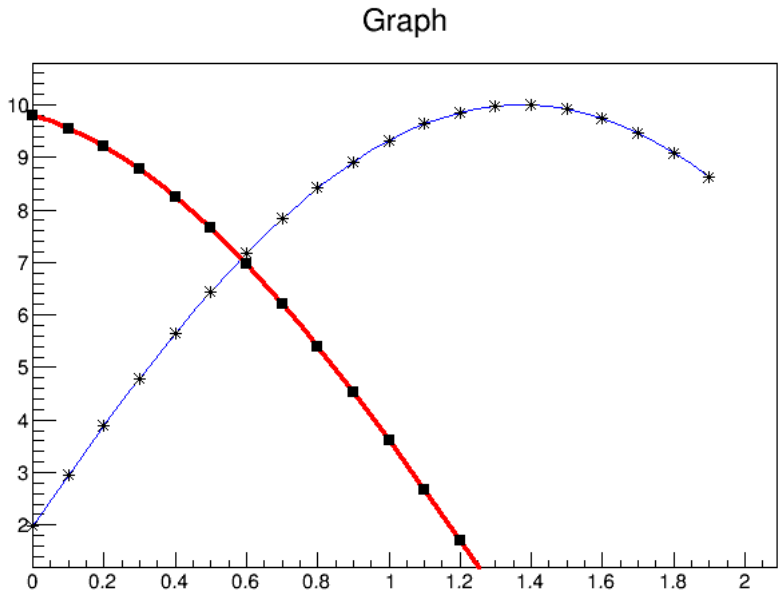


Hình 5.3: Đồ thị ví dụ cho `TGraph`

Trong trường hợp ta muốn vẽ hai hay nhiều đồ thị cùng lúc, ta chỉ cần vẽ trục đồ thị duy nhất một lần đầu tiên, tùy chọn `"A"` không cần sử dụng cho các đồ thị từ thứ hai trở đi.

Ví dụ: *(xem Hình 5.4)*

```
{
  Int_t n = 20;
  Double_t x[n], y[n], x1[n], y1[n];
  for (Int_t i=0;i<n;i++) {
      x[i] = i*0.1;
      y[i] = 10*sin(x[i]+0.2);
      x1[i] = i*0.1;
      y1[i] = 10*cos(x[i]+0.2);
   }

  // Do thi thu nhat
  TGraph *gr1 = new TGraph(n,x,y);
  gr1->SetLineColor(4);
  gr1->Draw("AC*");

  // Do thi thu 2 ve chong len do thi 1
  TGraph *gr2 = new TGraph(n,x1,y1);
  gr2->SetLineWidth(3);
  gr2->SetMarkerStyle(21);
  gr2->SetLineColor(2);
  gr2->Draw("CP");
}
```



Hình 5.4: Đồ thị ví dụ vẽ hai `TGraph` cùng lúc

Một số phương thức của lớp `TGraph`[2]

- `RemovePoint(i)`: xóa điểm thứ i
- `CompareX(g,i,j)`, `CompareY(g,i,j)`: so sánh giá trị của hai điểm thứ `i` và `j` trong đồ thị `g`
- `Sort()`: sắp xếp lại các điểm

[2] Xem thêm tại http://root.cern.ch/root/html/TGraph.html

5.2.2 TGraphErrors

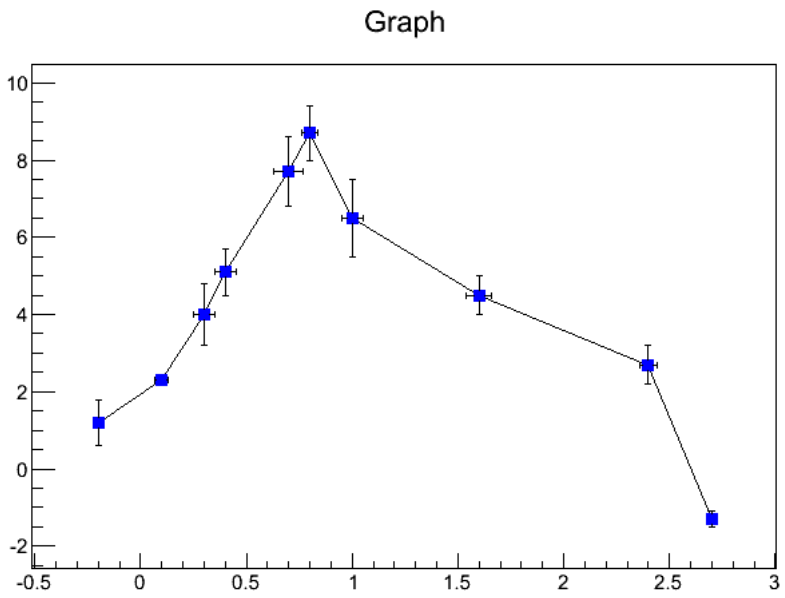
TGraphErrors cũng tương tự như TGraph nhưng được tạo bởi hai mảng X và Y tương ứng có sai số đi kèm. Cú pháp khai báo đối tượng thuộc lớp TGraphErrors như sau

<tên graph> = new TGraphErrors(n, x, y, ex, ey)

Trong đó `n` là số điểm, `x` và `y` là hai mảng có `n` điểm tương ứng, `ex` và `ey` là hai mảng chứa sai số của `x` và `y`

Ví dụ: *(xem Hình 5.5)*

```
{
   Int_t n = 10;
   Double_t x[n] = {-0.2, 0.1, 0.3, 0.4, 0.7, 0.8, 1.0, 1.6, 2.4, 2.7};
   Double_t y[n] = {1.2, 2.3, 4.0, 5.1, 7.7, 8.7, 6.5, 4.5, 2.7, -1.3};
   Double_t ex[n] = {0.02, 0.03, 0.05, 0.05, 0.07, 0.04, 0.05, 0.06, 0.04,
       0.01};
   Double_t ey[n] = {0.6, 0.1, 0.8, 0.6, 0.9, 0.7, 1.0, 0.5, 0.5, 0.2};
   gr = new TGraphErrors(n,x,y,ex,ey);
   gr->SetMarkerStyle(21);
   gr->SetMarkerColor(4);
   gr->Draw("ALP");
}
```



Hình 5.5: Đồ thị phần ví dụ cho TGraphErrors

Các tùy chọn cho TGraphErrors cũng tương tự như TGraph nhưng có thêm hai tùy chọn là `"Z"` và `">"` để vẽ các đường nhỏ và mũi tên ở cuối thanh sai số. Nếu tùy chọn `"|>"` được chọn, các mũi tên lớn sẽ được vẽ ở cuối thanh sai số, kích thước của các mũi tên này bằng 2/3 so với kích thước của marker.

Một số phương thức thông dụng trong TGraphErrors[3]

- `GetErrorX(i)`, `GetErrorY(i)`: sai số theo trục `x` và `y` tại bin thứ `i`
- `Apply(f)`: áp dụng hàm `f(x,y)` cho tất cả các điểm

[3] Xem thêm tại http://root.cern.ch/root/html/TGraphErrors.html

5.2.3 TGraphAsymmErrors

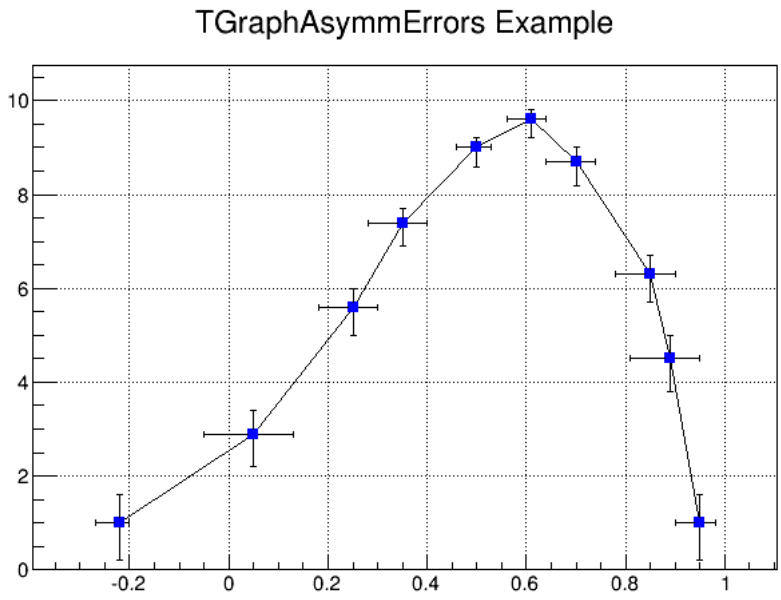
TGraphErrors cũng tương tự như TGraphErrors nhưng dành cho các dữ liệu có sai số bất đối xứng. Cú pháp khai báo đối tượng thuộc lớp TGraphAsymmErrors như sau

<tên graph> = new TGraphAsymmErrors(n, x, y, exl, exh, eyl, eyh)

Trong đó n là số điểm, x và y la hai mảng có n điểm tương ứng, exl (eyl), exh (eyh) là hai mảng chứa sai số về bên trái và bên phải của x (y)

Ví dụ: *(xem Hình 5.6)*

```
{
  c1 = new TCanvas("c1","A Simple Graph with error bars",200,10,700,500);
  c1->SetGrid();

  Int_t n = 10;
  Double_t x[n] = {-.22,.05,.25,.35,.5, .61,.7,.85,.89,.95};
  Double_t y[n] = {1,2.9,5.6,7.4,9,9.6,8.7,6.3,4.5,1};

  Double_t exl[n] = {.05,.1,.07,.07,.04,.05,.06,.07,.08,.05};
  Double_t eyl[n] = {.8,.7,.6,.5,.4,.4,.5,.6,.7,.8};
  Double_t exh[n] = {.02,.08,.05,.05,.03,.03,.04,.05,.06,.03};
  Double_t eyh[n] = {.6,.5,.4,.3,.2,.2,.3,.4,.5,.6};

  gr = new TGraphAsymmErrors(n,x,y,exl,exh,eyl,eyh);
  gr->SetTitle("TGraphAsymmErrors Example");
  gr->SetMarkerColor(4);
  gr->SetMarkerStyle(21);
  gr->Draw("ALP");
}
```



Hình 5.6: Đồ thị phần ví dụ cho TGraphAsymmErrors

5.2.4 TGraphPolar

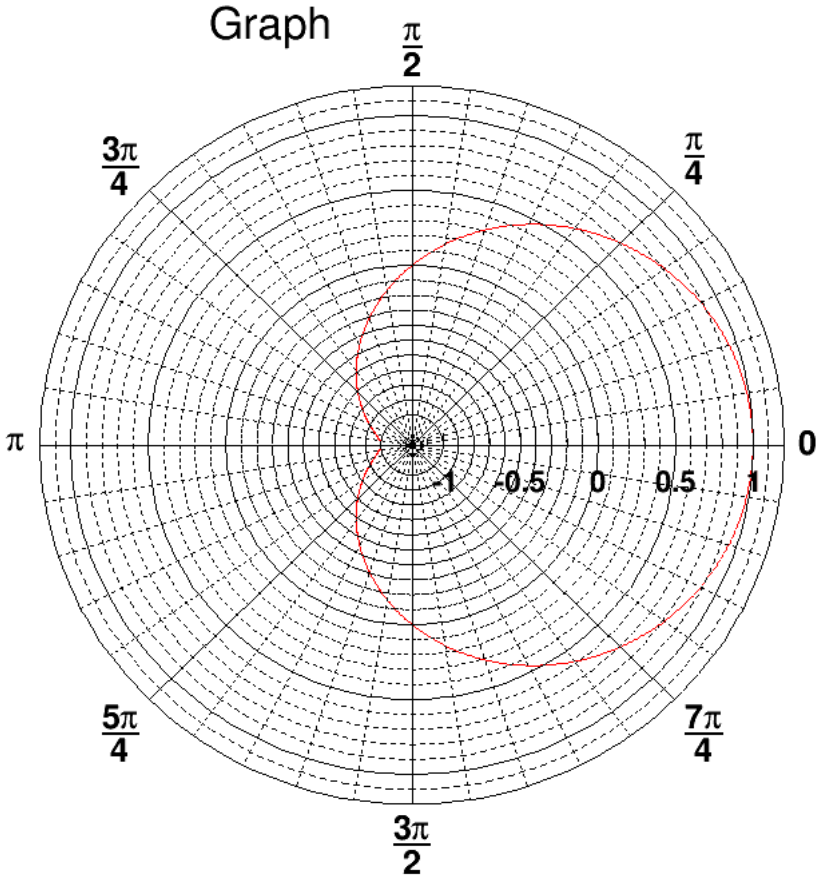
Lớp TGraphPolar dùng để tạo các đồ thị trên hệ trục tọa độ cực (bao gồm cả sai số). Cú pháp khai báo đối tượng thuộc lớp TGraphPolar như sau

<tên graph> = new TGraphPolar(n, theta, r, etheta, er)

Trong đó `n` là số điểm, `theta` và `r` là các giá trị của `n` điểm tương ứng, `etheta` và `er` là hai mảng chứa sai số của `theta` và `r`

Ví dụ: *(xem Hình 5.7)*

```
{
  TCanvas *CPol = new TCanvas("CPol","TGraphPolar Examples",700,700);
  Double_t rmin=0;
  Double_t rmax=TMath::Pi()*2;
  Double_t r[1000];
  Double_t theta[1000];
  TF1 * fp1 = new TF1("fplot","cos(x)",rmin,rmax);
  for (Int_t ipt = 0; ipt < 1000; ipt++) {
    r[ipt] = ipt*(rmax-rmin)/1000+rmin;
  theta[ipt] = fp1->Eval(r[ipt]);
  }
  TGraphPolar * grP1 = new TGraphPolar(1000,r,theta);
  grP1->SetLineColor(2);
  grP1->Draw("AOL");
}
```



Hình 5.7: Đồ thị phần ví dụ cho `TGraphPolar`

5.2.5 TMultiGraph

Lớp `TMultiGraph` tạo một tập hợp các đồ thị. Cú pháp khai báo đối tượng thuộc lớp `TMultiGraph` như sau

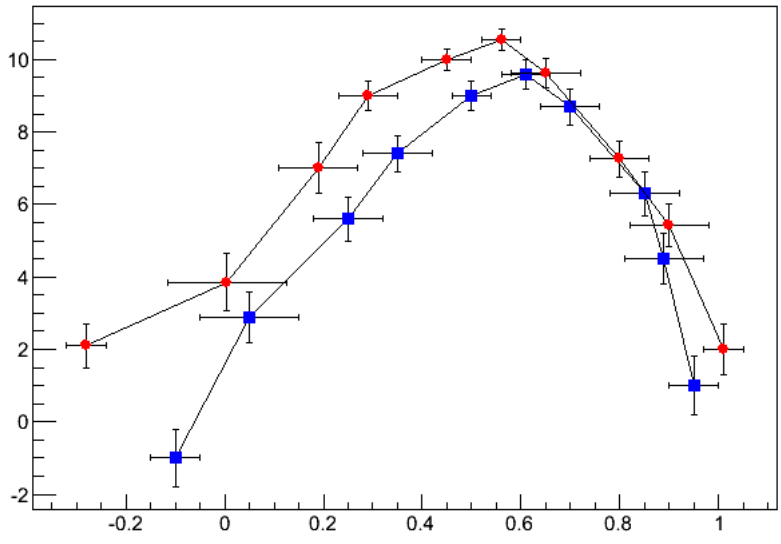
<tên multigraph> = new TMultiGraph("<tên multigraph>", "<tiêu đề>")

Một số phương thức thông dụng của TMultiGraph[4]

- `Add()`: thêm đồ thị vào multigraph
- `Draw()`: vẽ các đồ thị

Ví dụ: *(xem Hình 5.8)*

```
{
   TMultiGraph *mg = new TMultiGraph();

   Int_t n1 = 10, n2 = 10;
   Double_t x1[]  = {-0.1, 0.05, 0.25, 0.35, 0.5,
      0.61,0.7,0.85,0.89,0.95};
   Double_t y1[]  = {-1,2.9,5.6,7.4,9,9.6,8.7,6.3,4.5,1};
   Double_t ex1[] = {.05,.1,.07,.07,.04,.05,.06,.07,.08,.05};
   Double_t ey1[] = {.8,.7,.6,.5,.4,.4,.5,.6,.7,.8};
   TGraphErrors *gr1 = new TGraphErrors(n1,x1,y1,ex1,ey1);
   gr1->SetMarkerColor(kBlue);
   gr1->SetMarkerStyle(21);
   mg->Add(gr1);

   Float_t x2[]  = {-0.28, 0.005, 0.19, 0.29, 0.45,
      0.56,0.65,0.80,0.90,1.01};
   Float_t y2[]  = {2.1,3.86,7,9,10,10.55,9.64,7.26,5.42,2};
   Float_t ex2[] = {.04,.12,.08,.06,.05,.04,.07,.06,.08,.04};
   Float_t ey2[] = {.6,.8,.7,.4,.3,.3,.4,.5,.6,.7};
   TGraphErrors *gr2 = new TGraphErrors(n2,x2,y2,ex2,ey2);
   gr2->SetMarkerColor(kRed);
   gr2->SetMarkerStyle(20);
   mg->Add(gr2);
   mg->Draw("APL");
}
```



Hình 5.8: Đồ thị ví dụ cho MultiGraph

[4]Xem thêm tại http://root.cern.ch/root/html/TMultiGraph.html

5.2.6 TGraph2D và TGraph2DErrors

Hai lớp TGraph2D và TGraph2DErrors là các lớp được dùng để vẽ đồ thị được tạo bởi hai mảng X,Y và Z tương ứng. Cú pháp khai báo đối tượng thuộc hai lớp này như sau

<tên graph> = new TGraph(n, x, y, z)

<tên graph> = new TGraphErrors(n, x, y, z, ex, ey, ez)

Trong đó `n` là số điểm, `x`, `y` và `z` là các mảng có `n` điểm tương ứng, `ex`, `ey` và `ez` là các mảng chứa sai số của `x`, `y` và `z`.

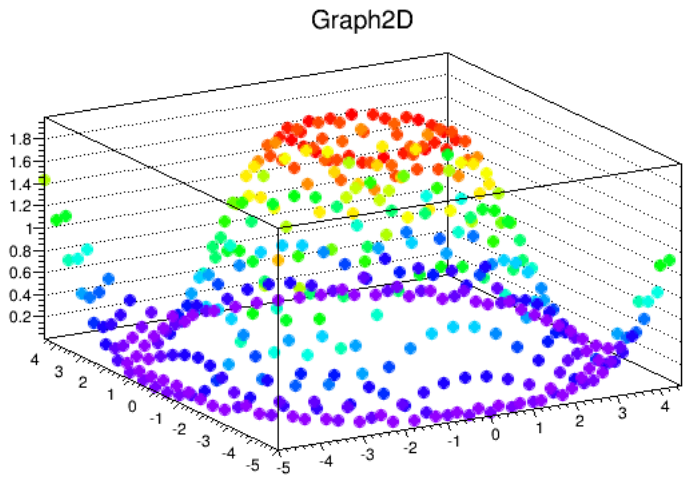
Các đối tượng TGraph2D có thể được vẽ với các tùy chỉnh dành cho histogram tương ứng, bên cạnh đó còn có thêm một số tùy chỉnh khác như

- "TRI": vẽ lưới tam giác Delaunay (*Delaunay triangle*) có sử dụng đổ màu
- "TRIW": vẽ lưới tam giác Delaunay dưới dạng lưới (*wireframe*)
- "TRI1": vẽ lưới tam giác Delaunay với các mức độ màu, cạnh của các tam giác được tô cùng màu với các đường
- "TRI2": vẽ lưới tam giác Delaunay với các mức độ màu
- "P": vẽ marker tại các điểm giao
- "P0": vẽ đường tròn tại các điểm giao

Ví dụ: *(xem Hình 5.9)*

```
{
   TCanvas *c1 = new TCanvas("c1","Graph2D example",0,0,600,400);
   Double_t P = 5.;
   Int_t npx  = 20 ;
   Int_t npy  = 20 ;
   Double_t x = -P;
   Double_t y = -P;
   Double_t z;
   Int_t k = 0;
   Double_t dx = (2*P)/npx;
   Double_t dy = (2*P)/npy;
   TGraph2D *dt = new TGraph2D(npx*npy);
   dt->SetNpy(41);
   dt->SetNpx(40);
   for (Int_t i=0; i<npx; i++) {
      for (Int_t j=0; j<npy; j++) {
         z = sin(sqrt(x*x+y*y))+1;
         dt->SetPoint(k,x,y,z);
         k++;
         y = y+dy;
      }
      x = x+dx;
      y = -P;
   }
   gStyle->SetPalette(1);
   dt->SetMarkerStyle(20);
   dt->Draw("pcol");
   return c1;
}
```

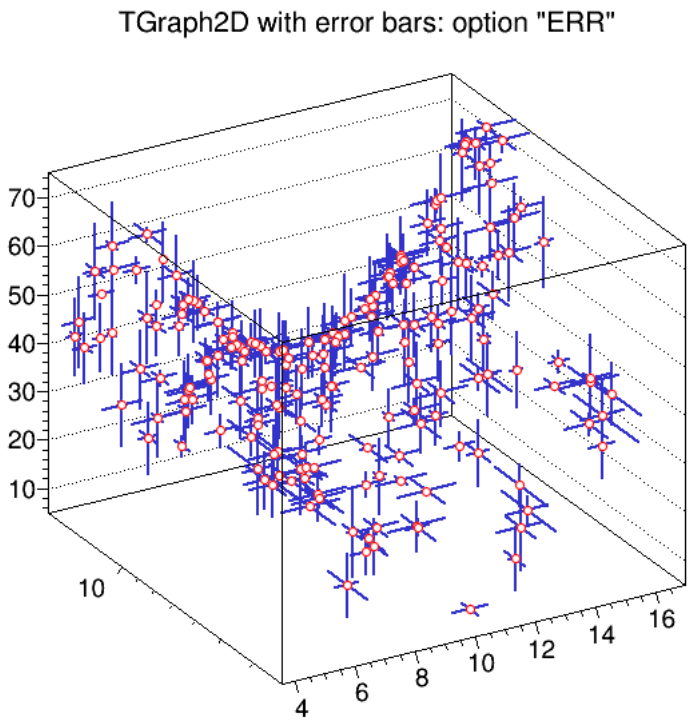
Ví dụ: *(xem Hình 5.10)*



Hình 5.9: Đồ thị phần ví dụ cho TGraph2D

```
{
   TCanvas *c = new TCanvas("c","Graph2DErrors example",0,0,600,600);
   Double_t P = 6.;
   Int_t np   = 200;

   Double_t *rx=0, *ry=0, *rz=0;
   Double_t *ex=0, *ey=0, *ez=0;

   rx = new Double_t[np];
   ry = new Double_t[np];
   rz = new Double_t[np];
   ex = new Double_t[np];
   ey = new Double_t[np];
   ez = new Double_t[np];

   TRandom *r = new TRandom();

   for (Int_t N=0; N<np;N++) {
      rx[N] = 2*P*(r->Rndm(N))-P;
      ry[N] = 2*P*(r->Rndm(N))-P;
      rz[N] = rx[N]*rx[N]-ry[N]*ry[N];
      rx[N] = 10.+rx[N];
      ry[N] = 10.+ry[N];
      rz[N] = 40.+rz[N];
      ex[N] = r->Rndm(N);
      ey[N] = r->Rndm(N);
      ez[N] = 10*r->Rndm(N);
   }

   TGraph2DErrors *dte = new TGraph2DErrors(np, rx, ry, rz, ex, ey, ez);
   dte->SetTitle("TGraph2D with error bars: option \"ERR\"");
   dte->SetFillColor(29);
   dte->SetMarkerSize(0.8);
   dte->SetMarkerStyle(20);
   dte->SetMarkerColor(kRed);
   dte->SetLineColor(kBlue-3);
   dte->SetLineWidth(2);
   dte->Draw("err p0");
   gPad->SetLogy(1);
```

```
    return c;
}
```



Hình 5.10: Đồ thị phần ví dụ cho TGraph2DErrors

5.3 Một số hiệu chỉnh cho đồ thị

5.3.1 Trục tọa độ

Các đối tượng trục tọa độ (thuộc lớp TAxis) được xây dựng một cách tự động thông qua các đối tượng khác như histogram hoặc đồ thị.

Với histogram, đối tượng trục tọa độ có thể được gọi thông qua phương thức TH1::GetAxis()

```
TAxis *axis = histo->GetXaxis()
```

Để thiết lập tên (*title*) của đối tượng này ta có thể làm như sau

```
axis->SetTitle("Ten cua truc")
h->GetXaxis()->SetTitle("Ten cua truc")
```

Một số tùy chỉnh cho đối tượng trục tọa độ

- Chọn màu trục

  ```
  axis->SetAxisColor(Color_t color = 1);
  ```

- Chọn số khoảng chia (*number of divisions*) trên trục

  ```
  axis->SetNdivisions(Int_t ndiv = 510, Bool_t optim = kTRUE);
  ```

  Giá trị của ndiv và optim được tính như sau

  – ndiv = N1 + 100*N2 + 10000*N3

  - `N1`: số khoảng chia chính
  - `N2`: số khoảng chia phụ
  - `N3`: số khoảng chia nhỏ
  - `optim = kTRUE` (mặc định) số khoảng chia được tối ưu dựa vào những giá trị đã cho
  - `optim = kFALSE` hay `ndiv < 0`, số khoảng chia bắt buộc giống với những gì được thết lập

  Ví dụ:
  `ndiv = 0`: không vẽ khoảng chia trên trục tọa độ
  `ndiv = 510`: trục tọa độ được đánh dấu 5 khoảng chia chính, trong mỗi khoảng chính có 10 khoảng chia phụ
  `ndiv = -10`: trục tọa độ được đánh dấu chính xác 10 khoảng chia chính

- Phóng to trục tọa độ bằng cách thiết lập khoảng giá trị để vẽ

```
axis->SetRange(Int_t binfirst,Int_t binlast)
axis->SetRangeUser(Axis_t ufirst,Axis_t ulast)
```

  Điểm khác biệt giữa hai phương thức `SetRange()` và `SetRangeUser()` là phương thức `SetRange()` có đối số là giá trị của bin (vd: `SetRange(1, 10)` sẽ vẽ từ bin 1 đến bin 10) còn `SetRangeUser()` có đối số là giá trị của trục (vd: `SetRangeUser(0., 3.)` sẽ vẽ từ giá trị 0 đến 3).

- Tùy chỉnh cho nhãn (*label*) của trục

```
axis->SetLabelColor(Color_t color = 1);  // mau
axis->SetLabelFont(Style_t font = 62);  // font
axis->SetLabelOffset(Float_t offset = 0.005);  // khoang cach toi truc
axis->SetLabelSize(Float_t size = 0.04);  // kich thuoc
```

- Tùy chỉnh cho kích thước đường đánh dấu các khoảng chia của trục

```
axis->SetTickLength(Float_t length = 0.03);
```

- Tùy chỉnh cho tên của trục

```
axis->SetTitleOffset(Float_t offset = 1);  // khoang cach toi truc
axis->SetTitleSize(Float_t size = 0.02);  // kich thuoc
```

Trong trường hợp ta muốn vẽ trục một cách độc lập mà không cần thông qua histogram hay đồ thị, ta có thể sử dụng lớp `TGaxis`, cú pháp khai báo đối tượng thuộc lớp này như sau

```
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 gridlength = 0)
```

Trong đó `xmin`, `ymin` là tọa độ mà trục bắt đầu trong hệ trục tọa độ của người dùng, và `xmax`, `ymax` là tọa độ kết thúc. Các đối số `wmin` và `wmax` là các giá trị nhỏ nhất (khi bắt đầu) và lớn nhất (khi kết thúc) được thể hiện trên trục, `ndiv` là số các khoảng chia trên trục. Các tùy chỉnh được cho bởi chuỗi `"chopt"` gồm có

- `chopt = 'G'`: thang đo logarithmic (mặc định là thang đo tuyến tính)
- `chopt = 'B'`: trục tọa độ trắng (*blank*), thường được sử dụng trong trường hợp vẽ chồng trục tọa độ

Ví dụ:

```
TGaxis *axis1 = new TGaxis(-4.5,-0.2,5.5,-0.2,-6,8,510,"");
axis1->SetName("axis1");
axis1->Draw();
```

5.3.2 Bảng chú giải

Cú pháp khai báo bảng chú giải như sau:

<tên bảng> = new TLegend(x1, y1, x2, y2, <tiêu đề>, option)

Trong đó x1, y1, x2, y2 là tọa độ của bảng chú giải trong pad.

Một số lệnh liên quan đến bảng chú giải:
(xem thêm tại http://root.cern.ch/root/html/TLegend.html)

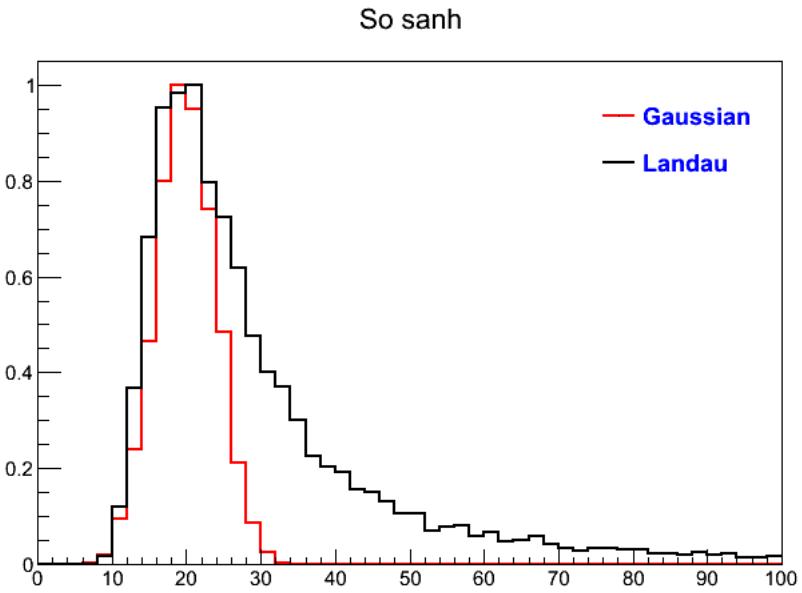
- `AddEntry(h," ")`: gán nội dung cho histogram h
- `SetBorderMode()`, `SetBorderSize()`: chọn kiểu và bề dày cho khung của bảng chú giải
- `SetTextFont()`, `SetTextSize()`, `SetTextColor()`: chọn font chữ và cỡ chữ, màu chữ hển thị
- `SetFillColor()`: chọn màu cho bảng chú giải
- `Draw()`: vẽ bảng chú giải

Lưu ý: trong một số trường hợp sau khi sử dụng lệnh `Draw()`, nếu bảng chú giải (thậm chí là trục tọa độ) không hiển thị như được chỉnh sửa thì nên gõ thêm lệnh `gPad->RedrawAxis()` sau khi chỉnh sửa.

Ví dụ: *So sánh 2 dạng phân bố Gauss và Landau (xem Hình 5.11)*

```
{
  TH1F *h1 = new TH1F("h1","So sanh", 50, 0, 100);
  TH1F *h2 = new TH1F("h2","So sanh", 50, 0, 100);
  h1->Sumw2();
  h2->Sumw2();
  Float_t mean = 20, sigma = 4;
  for (Int_t i = 0; i < 10000; i++) {
      h1->Fill(gRandom->Gaus(mean,sigma));  // phan bo Gauss
      h2->Fill(gRandom->Landau(mean,sigma));  // phan bo Landau
  }

  // Chuan dinh cua hai phan bo ve 1
  h1->Scale(1./h1->GetMaximum());
  h2->Scale(1./h2->GetMaximum());

  // Ve cac phan bo
  h1->SetLineColor(kRed);
  h2->SetLineColor(kBlack);
  h1->SetLineWidth(2);
  h2->SetLineWidth(2);
  h1->Draw("histo");
  h2->Draw("same histo");

  // Chon thong so cho bang chu giai
  gStyle->SetOptStat(0);  // an bang thong tin histogram
  leg = new TLegend(0.7,0.7,0.9,0.85);  // khai bao bang chu giai
  leg->SetTextSize(0.04);  // chon co chu
  leg->SetTextColor(kBlue);  // chon mau chu
  leg->SetFillColor(0);  // chon mau to
  leg->SetBorderSize(0);  // khong ve khung bang chu giai
  leg->AddEntry(h1,"Gaussian");  // dat chu giai cho histogram h1
```

```
  leg->AddEntry(h2,"Landau");  // dat chu giai cho histogram h2
  leg->Draw();  // ve bang chu giai
}
```



Hình 5.11: So sánh dạng phân bố Gauss và Landau

5.3.3 Cách tạo văn bản và biểu thức toán học

Các đoạn văn bản (*text*) được đưa vào trong pad có thể được nằm trong các khung được gọi là *pave* (lớp `TPaveLabel`) hoặc là nằm tự do. Các đoạn văn bản hiển thị tiêu đề đồ thị hay các trục có thể được khai báo một cách trực tiếp. Tất cả các văn bản, công thức toán học hiển thị trong hình vẽ đều là các đối tượng thuộc lớp `TText` và tuân theo cú pháp viết văn bản latex (lớp `TLatex`). Hình 5.12 trình bày cách viết một số kí hiệu latex đơn giản được sử dụng trong ROOT.

Ví dụ: *(xem Hình 5.13)*

```
{
  TLatex l;
  l->SetTextAlign(12);
  l->SetTextSize(0.04);
  l->DrawLatex(0.1,0.8,"1) C(x) = d #sqrt{#frac{2}{#lambdaD}}
    #int^{x}_{0}cos(#frac{#pi}{2}t^{2})dt");
  l->DrawLatex(0.1,0.6,"2) C(x) = d #sqrt{#frac{2}{#lambdaD}}
    #int^{x}cos(#frac{#pi}{2}t^{2})dt");
  l->DrawLatex(0.1,0.4,"3) R = |A|^{2} =
    #frac{1}{2}(#[]{#frac{1}{2}+C(V)}^{2}+
    #[]{#frac{1}{2}+S(V)}^{2})");
  l->DrawLatex(0.1,0.2,"4) F(t) = #sum_{i=
    -#infty}^{#infty}A(i)cos#[]{#frac{i}{t+i}}");
}
```

Một số phương thức thông dụng tùy chỉnh cho văn bản chẳng hạn như

- Chỉnh căn lề

```
l->SetTextAlign(align)
```

Lower case		Upper case		Variations	
alpha :	α	Alpha :	A		
beta :	β	Beta :	B		
gamma :	γ	Gamma :	Γ		
delta :	δ	Delta :	Δ		
epsilon :	ϵ	Epsilon :	E	varepsilon :	ε
zeta :	ζ	Zeta :	Z		
eta :	η	Eta :	H		
theta :	θ	Theta :	Θ	vartheta :	ϑ
iota :	ι	Iota :	I		
kappa :	κ	Kappa :	K		
lambda :	λ	Lambda :	Λ		
mu :	μ	Mu :	M		
nu :	ν	Nu :	N		
xi :	ξ	Xi :	Ξ		
omicron :	o	Omicron :	O		
pi :	π	Pi :	Π		
rho :	ρ	Rho :	P		
sigma :	σ	Sigma :	Σ	varsigma :	ς
tau :	τ	Tau :	T		
upsilon :	υ	Upsilon :	Y	varUpsilon :	$\varUpsilon$
phi :	ϕ	Phi :	Φ	varphi :	φ
chi :	χ	Chi :	X		
psi :	ψ	Psi :	Ψ		
omega :	ω	Omega :	Ω	varomega :	$\varomega$

Hình 5.12: Bảng chữ cái Latin

$$1)\ C(x) = d\sqrt{\frac{2}{\lambda D}}\int_0^x \cos\left(\frac{\pi}{2}t^2\right)dt$$

$$2)\ C(x) = d\sqrt{\frac{2}{\lambda D}}\int^x \cos\left(\frac{\pi}{2}t^2\right)dt$$

$$3)\ R = |A|^2 = \frac{1}{2}\left(\left[\frac{1}{2}+C(V)\right]^2+\left[\frac{1}{2}+S(V)\right]^2\right)$$

$$4)\ F(t) = \sum_{i=-\infty}^{\infty} A(i)\cos\left[\frac{i}{t+i}\right]$$

Hình 5.13: Ví dụ công thức latex

Tham số `align` là một số có 2 chữ số, mỗi chữ số mang 1 trong 3 giá trị (1-3) tương ứng

với căn trái, giữa và phải (chữ số đầu tiên) và căn trên, giữa, dưới (chữ số số thứ 2), ví dụ: `l->SetTextAlign(13)` có nghĩa là căn trái và dưới.

- Chỉnh góc quay

```
l->SetTextAngle(angle)
```

Với `angle` là góc quay (tính theo độ) so với phương ngang.

- Chỉnh màu

```
l->SetTextColor(color)
```

- Chỉnh kích cỡ chữ

```
l->SetTextSize(size)
```

Trong đó `size` là giá trị thể hiện tỉ lệ (theo %) so với kích thước của pad mà chữ được viết lên trên. Kích cỡ chữ hiển thị tính theo pixel sẽ bằng tích của `size` nhân với chiều cao và chiều rộng của pad.

- Chỉnh font chữ

```
l->SetTextFont(font)
```

Trong đó `font = 10*fontID + precision` với `fontID` là giá trị thể hiện loại font chữ của văn bản, có tất cả 14 loại font chữ trong ROOT

`fontID`	Font chữ	In nghiêng	Độ đậm
1	Times New Roman	Có	4
2	Times New Roman	Không	7
3	Times New Roman	Có	7
4	Arial	Không	4
5	Arial	Có	4
6	Arial	Không	7
7	Arial	Có	7
8	Courier New	Không	4
9	Courier New	Có	4
10	Courier New	Không	7
11	Courier New	Có	7
12	Symbol	Có	6
13	Times New Roman	Không	4
14	Wingdings	Không	4

Và `precision` mang một trong ba giá trị 0 (font chữ gốc của hệ thống), 1 (có thể thay đổi kích cỡ và quay góc) và 2 (tương tự 1 nhưng khác ở một số kí tự đặc biệt).

5.4 Một số đối tượng hình học khác

Đường thẳng

Đường thẳng trong ROOT được xây dựng qua lớp `TLine`

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

Trong đó `x1`, `y1` là tọa độ của điểm đầu; `x2`, `y2` là tọa độ của điểm cuối.

Ví dụ:

```
root[0] l = new TLine(0.2,0.2,0.8,0.3)
root[1] l->Draw()
```

Xem thêm tại https://root.cern.ch/root/html/TLine.html.

Mũi tên

Đường thẳng trong ROOT được xây dựng qua lớp TArrow

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

Trong đó x1, y1 , x2, y2 tương tự như đường thẳng, arrowsize là kích thước của đầu mũi tên, và option là tùy chọn cho hình thức của mũi tên:

Mũi tên hướng sang phải: ">" (không tô màu đầu mũi tên), "|>" (tô màu đầu mũi tên)
Mũi tên hướng sang trái: "<" (không tô màu đầu mũi tên), "<|" (tô màu đầu mũi tên)
Mũi tên ở hai đầu: "<>" (không tô màu đầu mũi tên), "<|>" (tô màu đầu mũi tên)

Để tô màu cho đầu mũi tên ta sử dụng phương thức TArrow::SetFillColor().

Xem thêm tại https://root.cern.ch/root/html/TArrow.html.

Đường tròn và ellipse

Đường tròn và ellipse trong ROOT được xây dựng qua lớp TEllipse

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

Ví dụ:

```
root[0] e = new TEllipse(0.2,0.2,0.8,0.3)
root[1] e->Draw()
```

Xem thêm tại https://root.cern.ch/root/html/TEllipse.html.

Hình vuông và chữ nhật

Hình vuông và chữ nhật trong ROOT được xây dựng qua lớp TBox

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

Ví dụ:

```
root[0] b = new TBox(0.2,0.2,0.8,0.3)
root[1] b->SetFillColor(5)
root[2] b->Draw()
```

Xem thêm tại https://root.cern.ch/root/html/TBox.html.

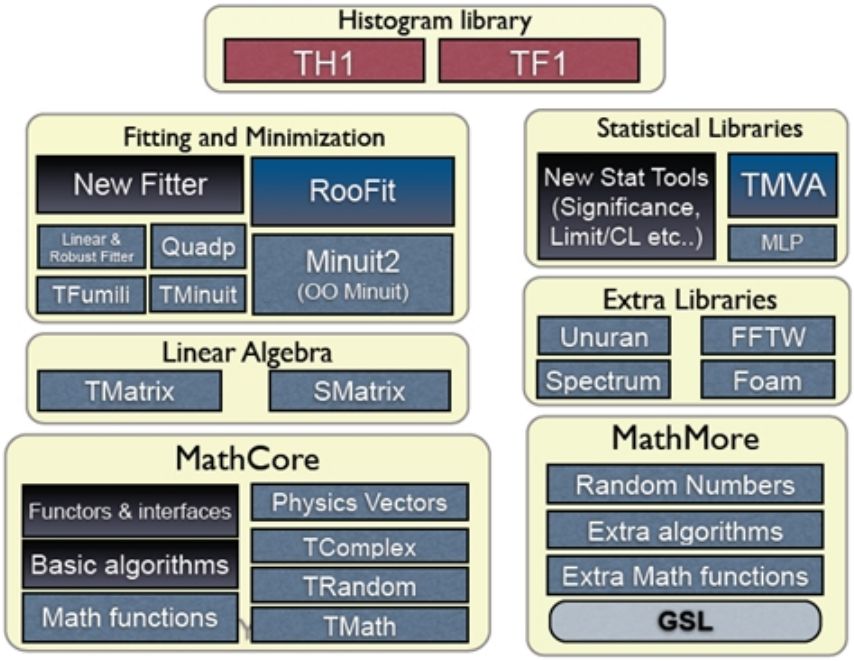
CHƯƠNG 6

CÁC THƯ VIỆN TOÁN HỌC

Các thư viện toán học trong ROOT cung cấp một lượng lớn các hàm toán học và thống kê (xem thêm tại https://root.cern.ch/root/htmldoc/guides/users-guide/MathLibraries.html). Hai thư viện toán học chính trong ROOT là

- Thư viện `MathCore`: cung cấp các lớp và hàm cho các tính toán số, thư viện này bao gồm các hàm toán học, thống kê và một số thuật toán cơ bản.
- Thư viện `MathMore`: cung cấp các hàm và thuật toán nâng cao dựa trên nền tảng thư viện GNU Scientific Library (GSL).

Ngoài ra còn có một số các gói thư viện khác như TMVA hay RooFit... như được trình bày trong Hình 6.1.



Hình 6.1: Cấu trúc các thư viện toán học trong ROOT

6.1 Các hàm toán học

Tất cả các hàm toán học đều được xây dựng như là các hàm tự do với không gian tên là `TMath` hay `ROOT::Math`. Các hàm đặc biệt trong không gian tên `ROOT::Math` được định nghĩa giống như trong phần mở rộng Thư viện Chuẩn (*Standard Library extensions*) của tài liệu Báo cáo Kĩ thuật C++ (*C++ Technical Report*).

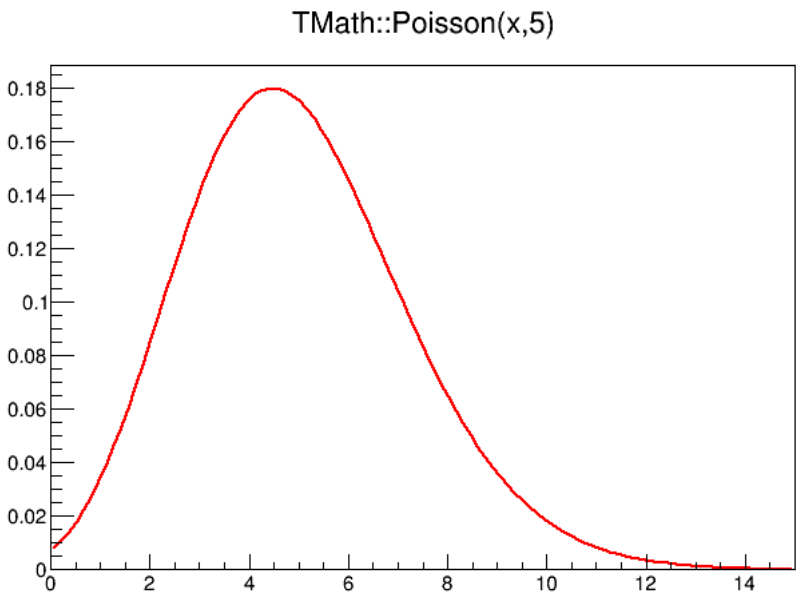
6.1.1 TMath

`TMath` là lớp chứa các hàm toán học được định nghĩa sẵn trong ROOT (xem thêm tại `http://root.cern.ch/root/html/TMath.html`). Một số hàm thông dụng:

- `Abs()`: hàm trị tuyệt đối
- `Sqrt()`: hàm khai căn
- `ASin()`, `ACos()`: các hàm *arcsin()*, *arccos()*,...
- `Gaus()`: hàm Gaussian
- `E()`, `Pi()`: các giá trị e, π

Ví dụ: *(xem Hình 6.2)*

```
{
   TF1 *f1 = new TF1("f1","TMath::Poisson(x,5)",0,15);
   f1->Draw();
}
```



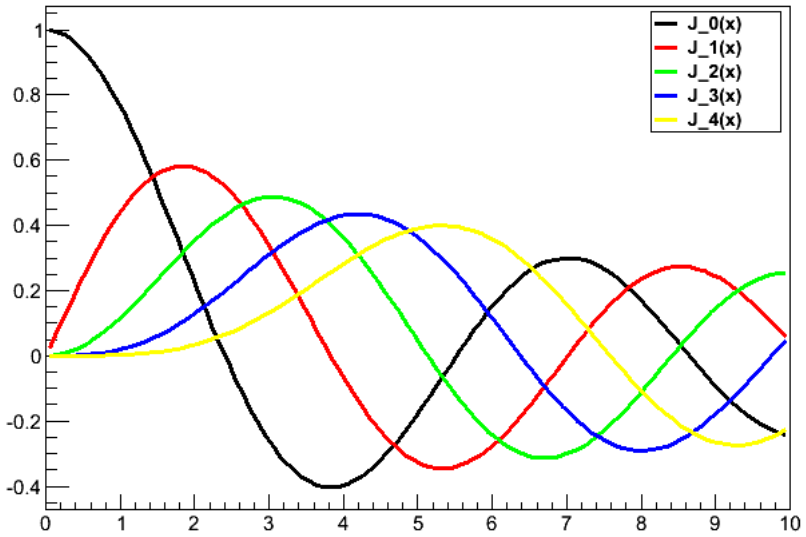
Hình 6.2: Ví dụ hàm Poisson(x,5)

6.1.2 Các hàm đặc biệt

Các hàm đặc biệt trong ROOT được định nghĩa trong file header `Math/SpecFunc.h` (xem thêm tại `http://project-mathlibs.web.cern.ch/project-mathlibs/sw/html/group__SpecFunc.html`).

Ví dụ: *(xem Hình 6.3)*

```
{
  gSystem->Load("libMathMore");

  TLegend *leg = new TLegend(0.75, 0.7, 0.89, 0.89);

  int n = 5;
  TF1* JBessel[5];
  for(int nu = 0; nu < n; nu++)
  {
       JBessel[nu]= new TF1("J_0", "ROOT::Math::cyl_bessel_j([0],x)", 0,
          10);
       JBessel[nu]->SetParameters(nu, 0.);
       JBessel[nu]->SetTitle("");
       JBessel[nu]->SetLineStyle(1);
       JBessel[nu]->SetLineWidth(3);
       JBessel[nu]->SetLineColor(nu+1);
  }

  leg->SetFillColor(0);
  leg->AddEntry(JBessel[0]->DrawCopy(), " J_0(x)", "l");
  leg->AddEntry(JBessel[1]->DrawCopy("same"), " J_1(x)", "l");
  leg->AddEntry(JBessel[2]->DrawCopy("same"), " J_2(x)", "l");
  leg->AddEntry(JBessel[3]->DrawCopy("same"), " J_3(x)", "l");
  leg->AddEntry(JBessel[4]->DrawCopy("same"), " J_4(x)", "l");
  leg->Draw();
}
```



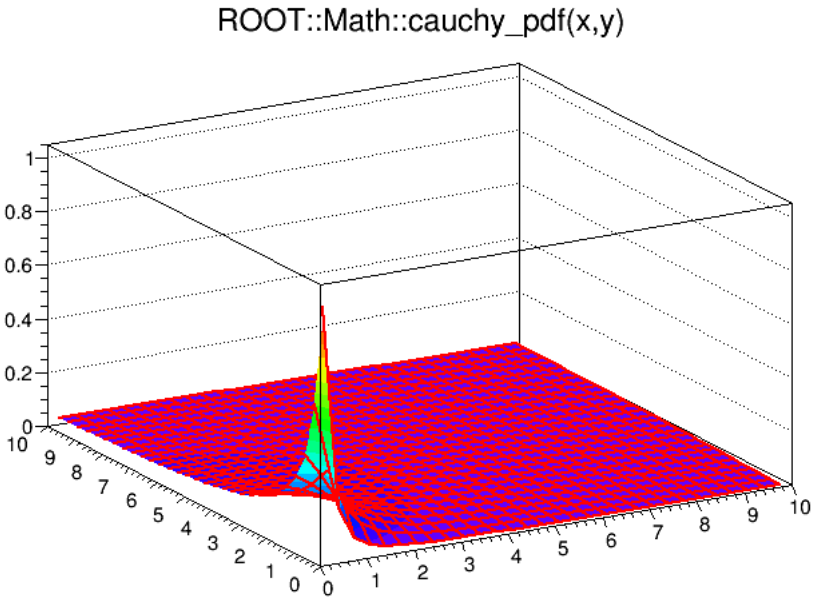
Hình 6.3: Các hàm Bessel J

6.1.3 Các hàm thống kê

Các hàm thống kê trong ROOT bao gồm các hàm mật độ xác suất (*probability density function*), xác suất tích lũy (*cumulative distribution function*) và nghịch đảo của chúng (*quantile*), được định nghĩa trong file header `Math/DistFunc.h` (xem thêm tại http://project-mathlibs.web.cern.ch/project-mathlibs/sw/html/group__StatFunc.html).

Ví dụ: *(xem Hình 6.4)*

```
{
    gSystem->Load("libMathCore");
    TF2 *f1 = new TF2("f1","ROOT::Math::cauchy_pdf(x, y)",0,10,0,10);
    f1->Draw("surf1");
}
```



Hình 6.4: Hàm mật độ xác suất Cauchy

6.2 Số ngẫu nhiên

TRandom là lớp tạo số ngẫu nhiên cơ bản (chu kì 10^9) (xem thêm tại http://root.cern.ch/root/html/TRandom.html). Ngoài ra còn có các lớp TRandom1 tạo số ngẫu nhiên dựa trên thuật toán RANLUX (chu kì 10^{171}), TRandom2 dựa trên thuật toán Tausworthe (chu kì 10^{26}) và TRandom3 dựa trên thuật toán Mersenne Twister (chu kì 10^{6000}).

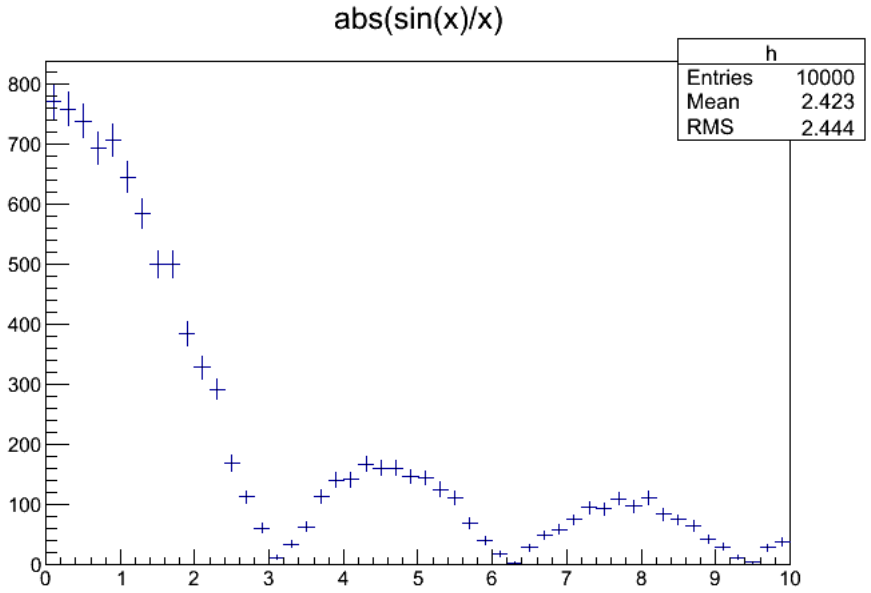
Một số phân bố ngẫu nhiên được định nghĩa sẵn trong ROOT:

- `Exp(X)`: phân bố exp(−x/X)
- `Integer(max)`: số nguyên ngẫu nhiên từ 0 đến max−1
- `Gaus(mean,sigma)`: phân bố Gauss
- `Rndm()`: phân bố ngẫu nhiên từ 0 đến 1
- `Uniform(x)`: phân bố ngẫu nhiên trong khoảng từ 0 đến x
- `Landau(mean,sigma)`: phân bố Landau
- `Poisson(mean)`: phân bố Poisson
- `Binomial(n,p)`: phân bố nhị thức
- `BreitWigner(mean,gamma)`: phân bố Breit − Wigner

Ngoài ra ta còn có thể gieo số ngẫu nhiêu theo một phân bố tự định nghĩa thông qua lệnh `GetRandom()`.

Ví dụ: *(xem Hình 6.5)*

```
{
  TH1F *h = new TH1F("h","abs(sin(x)/x)",50,0,10);
  h->Sumw2();
  TF1 *f = new TF1("f","abs(sin(x)/x)",0,10);  // dinh nghia ham f
  for (Int_t i = 0; i < 10000; i++) {
      h->Fill(f->GetRandom());  // gieo ngau nhien theo phan bo ham f
  }
  h->Draw();
}
```



Hình 6.5: Phân bố ngẫu nhiên theo hàm |sin(x)/x|

6.3 Ma trận và vector

6.3.1 Ma trận

Để xây dựng vào thực hiện tính toán ma trận trong ROOT ta có thể sử dụng các lớp ma trận cơ bản như TMatrixF, TMatrixD, TMatrixT, hoặc các ma trận đối xứng (TMatrixFSym, TMatrixDSym, TMatrixTSym),... (xem thêm tại http://root.cern.ch/root/html/MATH_MATRIX_Index.html).

Một số lệnh dành cho ma trận:

- `Print()`: xuất ra giá trị của ma trận
- `Determinant()`: tính định thức
- `Invert()`: tính định thức
- `Transpose()`: tính định thức
- `Plus(a,b)`: tổng hai ma trận
- `Minus(a,b)`: hiệu hai ma trận
- `Mult(a,b)`: tích hai ma trận

Ví dụ: *Tính định thức và ma trận nghịch đảo của một ma trận cho trước*

```
{
  #include <TMatrixD.h>

  Double_t x[]={3,4,0,1};
  // Tao ma tran n kich thuoc 2x2 tu mang x
  TMatrixD n(2,2,x);
  // Tinh dinh thuc cua ma tran n
  cout << "Dinh thuc = " << n.Determinant() << endl;
  // Tinh ma tran nghich dao
  TMatrixD m = n.Invert();
  m.Print();
}
```

Kết quả

```
Dinh thuc = 3

2x2 matrix is as follows

     |      0    |      1    |
---------------------------------
   0 |      0.3333       -1.333
   1 |           0            1
```

Ví dụ: *Tính trị riêng và vector riêng của ma trận đối xứng*

```
{
  #include <TMatrixDSym.h>
  #include <TMatrixDSymEigen.h>
  #include <TVectorD.h>

  const int N = 3;
  // Tao mang voi kich thuoc N*N
  double e[N*N] = {1, 4, 5,
                   4, 2, 6,
                   5, 6, 3};
  // Tao ma tran doi xung voi mang vua tao
  TMatrixDSym m(N, e);
  TMatrixDSymEigen me(m);
  // Tinh tri rieng va vector rieng
  TVectorD eigenval = me.GetEigenValues();
  TMatrixD eigenvec = me.GetEigenVectors();
  // Xuat ket qua
  m.Print();
  eigenval.Print();
  eigenvec.Print();
}
```

Kết quả

```
3x3 matrix is as follows

     |      0    |      1    |      2    |
--------------------------------------------
   0 |           1           4           5
   1 |           4           2           6
   2 |           5           6           3


Vector (3)  is as follows
```

```
     |         1   |
------------------
   0 |12.176
   1 |-2.50729
   2 |-3.66868


3x3 matrix is as follows

     |       0     |       1     |       2     |
------------------------------------------------
   0 |      0.4966       0.8096        0.313
   1 |      0.5774      -0.5774       0.5774
   2 |      0.6481       -0.106      -0.7541
```

6.3.2 Vector

Tương tự với các lớp ma trận , ta cũng có thể sử dụng các lớp vector như TVectorD, TVectorT, TVector1, TVector2, TVector3,... (xem thêm tại http://root.cern.ch/root/html/MATH_MATRIX_Index.html).

Một số lệnh dành cho vector:

- `Print()`: xuất ra giá trị của vector
- `Abs()`: tính trị tuyệt đối của vector
- `Sqr()`: bình phương các phần tử
- `Sqrt()`: khai căn các phần tử
- `Invert()`: nghịch đảo từng phần tử
- `Norm1()`: tính tổng của trị tuyệt đối các phần tử
- `Norm2Sqr()`: tính tổng bình phương các phần tử
- `Sum()`: tính tổng các phần tử
- `Add(v)`: cộng thêm vector v
- `Dot(v)`: tích vô hướng với vector v
- `Cross(v)`: tích hữu hướng với vector v
- `Unit(v)`: lấy vector đơn vị
- `Angle(v)`: góc giữa hai vector
- `RotateX(phi)`, `RotateY(phi)`,`RotateZ(phi)`: quay quanh trục một góc `phi`
- `Rotate(phi,v)`: quay quanh vector v một góc `phi`

Ví dụ: *Tính tổng hai vector với lớp TVector3*

```
{
  TVector3 v1(1,3,2);
  TVector3 v2(-2,2,0);
  v1 += v2;
  v1.Print();
}
```

Kết quả

```
TVector3 A 3D physics vector (x,y,z)=(-1.000000,5.000000,2.000000) (rho,
   theta,phi)=(5.477226,68.583286,101.309932)
```

Ví dụ: *Tính tổng hai vector với lớp TVectorD*

```
{
  #include <TVectorD.h>

  TVectorD v1(2);
  TVectorD v2(2);
  v1(0)=1;   v1(1)=3;
  v2(0)=-2;  v2(1)=2;
  v1.Add(v2);
  v1.Print();
}
```

Kết quả

```
Vector (2)  is as follows

     |         1   |
------------------
   0 |-1
   1 |5
```

6.3.3 Lorentz Vector

TLorentzVector là vector 4 chiều được xây dựng bằng cách thêm 1 phần tử vào trong TVector3. Đây là lớp vector thường hay được sử dụng trong các bài toán hạt cơ bản đòi hỏi sử dụng các phép biến đổi Lorentz, tính khối lượng bất biến,... (xem thêm tại http://root.cern.ch/root/html/TLorentzVector.html).

Một số lệnh dành cho Lorentz vector:

- `SetXYZT()`: đặt các giá trị tọa độ (x,y,z) và thời gian t
- `SetPxPyPzE()`: đặt các giá trị xung lượng (p_x,p_y,p_z) và năng lượng E
- `SetPtEtaPhiE()`: đặt các giá trị xung lượng ngang p_T, góc η, ϕ và năng lượng E
- `SetPtEtaPhiM()`: đặt các giá trị xung lượng ngang p_T, góc η, ϕ và khối lượng M
- `Rho()`, `CosTheta()`, `Phi()`, `E()`, `M()`,...: truy xuất các giá trị ρ, $\cos(\theta)$, E, M,...
- `Boost(v)`: chuyển trục tọa độ theo vector v

Ví dụ: *Tính khối lượng bất biến của một hạt có năng lượng E và xung lượng (px,py,pz)*

```
{
  Double_t E=0.57, px=0.1, py=0.1, pz=0.2;
  // Khai bao mot hat co nang luong E va xung luong px,py,pz
  TLorentzVector particle(px,py,pz,E);
  // Tinh khoi luong bat bien cua hat
  cout << particle.M() << endl;
}
```

Kết quả

```
0.514684
```

Phép biến đổi Lorentz theo vector bất kì được thực hiện với các đối tượng thuộc lớp `TLorentzRotation` , thông qua toán tử `*`, lệnh `MatrixMultiplication(v)` hoặc `Transform(v)` với v là đối tượng thuộc lớp `TLorentzRotation`.

Ví dụ: *Tính xung lượng, năng lượng của hạt tới và hạt bia trong hệ tọa độ khối tâm*

```
{
  #include <TLorentzVector.h>

  Double_t E = 3.2;
  Double_t m = 0.94;

  // Hat toi khong khoi luong co px = E
  TLorentzVector   bombard( E, 0, 0, E );
  // Hat bia co khoi luong m
  TLorentzVector   target( 0, 0, 0, m );

  // Vector khoi tam
  TLorentzVector   cms = bombard+target;
  TVector3 boost = -cms.BoostVector();

  // Hat toi trong he toa do khoi tam
  bombard.Boost(boost);
  bombard.Print();
  // Hat bia trong he toa do khoi tam
  target.Boost(boost);
  target.Print();
}
```

Kết quả

```
(x,y,z,t)=(1.145159,0.000000,0.000000,1.145159) (P,eta,phi,E)
    =(1.145159,-0.000000,0.000000,1.145159)
(x,y,z,t)=(-1.145159,0.000000,0.000000,1.481550) (P,eta,phi,E)
    =(1.145159,-0.000000,3.141593,1.481550)
```

CHƯƠNG 7

FILE

File một tập hợp các thông tin (dữ liệu) mà được người dùng tạo ra từ máy tính. Đối với lượng thông tin nhỏ, người dùng có thể dễ dàng lưu chúng lại dưới dạng các văn bản (*text format*). Tuy nhiên đối với những lượng thông tin lớn, để có thể đạt được sự tối ưu trong lưu trữ dữ liệu, chúng thường được lưu lại dưới dạng nhị phân (*binary format*). Để giải quyết vấn đề này, ROOT đã cung cấp một định dạng file nhị phân nén (*compressed binary format*) có thể chứa cả thông tin của dữ liệu và mô tả của chúng. ROOT cũng cung cấp một công cụ tự động mã nguồn mở để tạo ra mô tả (*description* hay *dictionary*) khi chúng ta lưu trữ dữ liệu cũng như tạo ra một lớp C++ tương ứng với mô tả này khi đọc lại dữ liệu.

7.1 File

7.1.1 Khai báo file

Cú pháp khai báo file như sau:

TFile <tên file> = new TFile("<tên file>", option)

hoặc

TFile <tên file>("<tên file>", option)

Một số tùy chỉnh cho file

- `"NEW"`: tạo file mới, trong trường hợp file cùng tên có tồn tại, file mới sẽ không được tạo ra.
- `"CREATE"`: tương tự `"NEW"`.
- `"RECREATE"`: tạo file mới trong ngay cả khi hiện dang có file cùng tên với file định tạo.
- `"UPDATE"`: mở file đã có sẵn, nếu chưa có thì tạo mới.
- `"READ"`: mở file và đọc (mặc định).

Một số lệnh cho file (xem thêm tại http://root.cern.ch/root/html/TFile.html)

- `Close()`: đóng file
- `Draw()`: duyệt các cấu trúc của file
- `GetSize()`: trả về kích thước của file
- `ls()`: liệt kê nội dung của file

- `Print()`: in tất cả đối tượng có trong file
- `Recover()`: phục hồi file
- `ReOpen()`: mở lại file
- `Seek()`: tìm đến vị trí cụ thể trong file

Ví dụ:

```
root[0] TFile *f = new TFile("histogram.root");
root[1] f->ls();
TFile**    histogram.root Histograms for ROOT class
 TFile*    histogram.root Histograms for ROOT class
  KEY: TH1F hist1;1 Function to be fit
  KEY: TH1F hist2;1 Another function to be fit
root[2] .q
```

Một số đặc điểm của ROOT file

- Khi ROOT file được mở, nó trở thành thư mục hiện hành.
- Tất cả các histogram và tree sẽ được tự động lưu vào file.
- Khi file đóng, tất cả các histogram, tree, đối tượng liên quan đều sẽ bị xóa bỏ.
- Tất cả các đối tượng có xuất xứ từ `TObject` đều có thể được ghi trên file.

7.1.2 Cách lưu và đọc histogram từ file

Lưu histogram vào file để lưu histogram vào file chúng ta sử dụng phương thức `TH1::Write()`
Ví dụ:

```
   TFile f("histos.root","new");
   TH1F h1("hgaus","histo from a gaussian",100,-3,3);
   h1.FillRandom("gaus",10000);
   h1.Write();
```

Ngoài ra chúng ta cũng có thể lưu tất cả các histogram (hay rộng hơn là các đối tượng) đang lưu trong bộ nhớ vào file hiện hành bằng cách sử dụng phương thức `TFile::Write()`

```
file->Write();
```

Đọc histogram từ file để đọc histogram từ file chúng ta sử dụng phương thức `TFile::Get()`
Ví dụ:

```
   TFile f("histos.root");
   TH1F *h = (TH1F*)f.Get("hgaus");
```

7.1.3 Thư mục

Khi chúng ta tạo ra một đối tượng `TFile` thì nó sẽ trở thành thư mục hiện hành (*current directory*) của chúng ta. Để kiểm tra thư mục hiện hành chúng ta có thể gõ

```
gDirectory->pwd()
```

Nếu muốn tạo ra một thư mục con bên trong file, ta có thể sử dụng phương thức `TDirectory::mkdir()`

```
   TFile *f = new TFile("file.root","RECREATE");
   f->mkdir("folder");
   f->ls();
```

Phương thức `TDirectory::ls()` sẽ liệt kê tất cả các thư mục có trong thư mục hiện hành, kết quả sẽ là

```
TFile**         file.root
 TFile*         file.root
  TDirectoryFile*                folder  folder
  KEY: TDirectoryFile   folder;1         folder
```

Để di chuyển tới thư mục `folder` ta gõ

```
   f->cd("folder");
```

Thông tin thêm về `TDirectory` có thể xem được tại https://root.cern.ch/root/html/TDirectory.html.

Cách tạo một histogram bên trong một thư mục như sau

```
root[0] f->cd("folder")
root[1] TH1F *histo = new TH1F("histo","histo",10,0,10)
root[2] gDirectory->ls()
TDirectory* folder folder
OBJ: TH1F histo histo : 0
```

Để truy xuất một histogram nằm bên trong 1 thư mục ta cần cung cấp đường dẫn đến thư mục đó từ file

```
root[0] TH1 *h
root[1] f->GetObject("folder/histo;1",h)
```

Số `1` nằm sau dấu `;` được gọi là *cycle number*, dùng để phân biệt các histogram có cùng tên với nhau. Trong trường hợp các tên của histogram là khác nhau thì ta có thể không cần khai báo số này khi truy xuất histogram.

7.2 Tree

Trong quá trình xử lý, dữ liệu có thể được tổ chức theo nhiều cách khác nhau. Tuy nhiên, trong các trường hợp thông thường, dữ liệu được lưu dưới dạng tuyến tính (*record*) và cuối cùng được tổ chức dưới dạng các bảng (*table*). Tất nhiên cách tổ chức dữ liệu này cũng được chấp nhận trong ROOT. Các bảng này được gọi là các *n-tuple*, các *record* được gọi là *event* và các tiêu đề cột được gọi là các biến (*variable*).

Chúng ta có thể lưu các *n-tuple* trong ROOT theo hai cách. Các thứ nhất là lưu theo một chuỗi các event, đây cũng là cách lưu dữ liệu thông thường. Tuy nhiên, khi người sử dụng chỉ quan tâm đến một bộ số liệu nhỏ hơn nhiều so với cấu trúc dữ liệu được lưu thì hiệu quả xử lý sẽ không được cao do phải di chuyển từ bộ số liệu của event này đến bộ số liệu của event kế tiếp. Do đó, thay vì lưu trữ theo chuỗi các event, ROOT sẽ chia tách mỗi event theo các biến và xây dựng một tập tin bằng cách đặt cùng tất cả các cột. Điều này tạo ra hai ưu điểm: thứ nhất, mỗi cột là một chuỗi đồng nhất của cùng một biến, do đó sẽ cho kích thước file nhỏ hơn nhiều so với trường hợp thông thường. Thứ hai, khi người sử dụng chỉ quan tâm đến một vài biến cụ thể của mỗi event, thời gian truy xuất dữ liệu nhỏ hơn nhiều do không phải quét toàn bộ dữ liệu trong file.

Trong ROOT, lớp `TTree` được thiết kế để lưu trữ dữ liệu theo cách thứ hai, làm giảm không gian lưu trữ và tăng tốc độ truy cập dữ liệu.

7.2.1 TNtuple

Lớp TNtuple là một phiên bản đơn giản của lớp TTree, chỉ chứa các dòng số liệu. Cú pháp khai báo Ntuple như sau:
TNtuple <tên ntuple> = new TNtuple("<tên ntuple>", "<tiêu đề>", các biến, kích thước)

Một số lệnh dành cho Ntuple

- `Fill(*x)`: lưu giá trị của mảng vào ntuple
- `Fill(x0, x1, x2,...)`: lưu các giá trị vào ntuple
- `GetArgs()`: số lượng các biến
- `GetNvar()`: số các cột
- `Scan()`: xuất ra bảng giá trị
- `Print()`: in ra bảng thông tin

Ví dụ:

```
{
  TNtuple *ntuple = new TNtuple("ntuple","ntuple","x:y");
  for (Int_t i = 0; i < 100; i++) {
    x=gRandom->Gaus(40,10);
    y=gRandom->Gaus(10,30);
    ntuple->Fill(x,y);
  }
  ntuple->Scan();
}
```

Kết quả in ra như sau:

```
***********************************
*    Row   *         x *         y *
***********************************
*        0 * 49.989326 * -3.042931 *
*        1 * 47.817962 * 9.0984172 *
*        2 * 48.242637 * 8.2984800 *
*        3 * 30.991239 * 7.7588658 *
*        4 * 40.079120 * -2.322895 *
*        5 * 53.911941 * -19.55198 *
*        6 * 39.510593 * -33.30006 *
*        7 * 29.393295 * -31.64893 *
*        8 * 47.673965 * -12.08088 *
*        9 * 45.797214 * -1.464032 *
*       10 * 60.609020 * -27.04453 *
*       11 * 51.653369 * -3.626074 *
*       12 * 38.652286 * -4.989183 *
*       13 * 38.175640 * 65.329391 *
*       14 * 37.571521 * 69.905838 *
*       15 * 40.048061 * -2.665381 *
*       16 * 55.405315 * 12.842284 *
*       17 * 55.246921 * 46.502574 *
*       18 * 38.636909 * 4.0235281 *
*       19 * 37.062355 * 6.4492521 *
*       20 * 28.397064 * 10.370863 *
*       21 * 40.220642 * 7.8624186 *
*       22 * 26.783823 * 62.329811 *
*       23 * 36.676227 * 2.5818193 *
*       24 * 28.197847 * -25.71282 *
```

```
Type <CR> to continue or q to quit ==>
```

7.2.2 TTree

Lớp `TTree` khác với lớp `TNtuple` ở chỗ là lớp `TNtuple` chỉ giới hạn trong việc chứa dữ liệu dạng số trong khi lớp `TTree` có thể chứa mọi loại dữ liệu chẳng hạn như các đối tượng (*object*) hoặc các mảng. Cú pháp khai báo đối tượng thuộc lớp `TTree` như sau:

TTree <tên tree> = new TTree("<tên file>", splitlevel)

Một số lệnh dành cho Tree

- `CloneTree()`: copy tree
- `Fill()`: lưu tất cả các *branch*
- `AddBranchToCache()`, `DropBranchFromCache()`: thêm, loại bỏ *branch* trong tree
- `GetEntries()`: trả về tổng số entry
- `Merge(list)`: hợp nhất tất cả các tree có trong list
- `Show(entry)`: liệt kê tất cả các giá trị của *leaf* trong entry
- `Process()`: thực thi code đối với mỗi event
- `Print()`: in ra bảng tóm tắt nội dung của tree
- `Scan()`: in ra tất cả các entry vượt qua selection
- `StartViewer()`: khởi động giao diện *TreeViewer*

Ví dụ: *(xem Hình 7.1)*

```
{
    TFile f("116944.root");
    TTree *t = (TTree*) t->Get("HWWTree_e");
    t->StartViewer();
    f->Close();
}
```

7.2.3 Cấu trúc của tree

Một tree có thể chứa các *branch* (nhánh), các *branch* này có thể là các biến, các *object* hoặc bao gồm cả tree khác. Cuối mỗi *branch* luôn là một biến, được gọi là *leaf* (lá). Thông thường, các tree đều được chia thành các *branch* khi được lưu vào một file ROOT. Bởi vì cấu trúc tree phức tạp hơn so với cấu trúc bảng, nên mức độ chia tách có thể được điều chỉnh để phù hợp với nhu cầu của người sử dụng. Hình 7.2 trình bày cấu trúc của một tree.

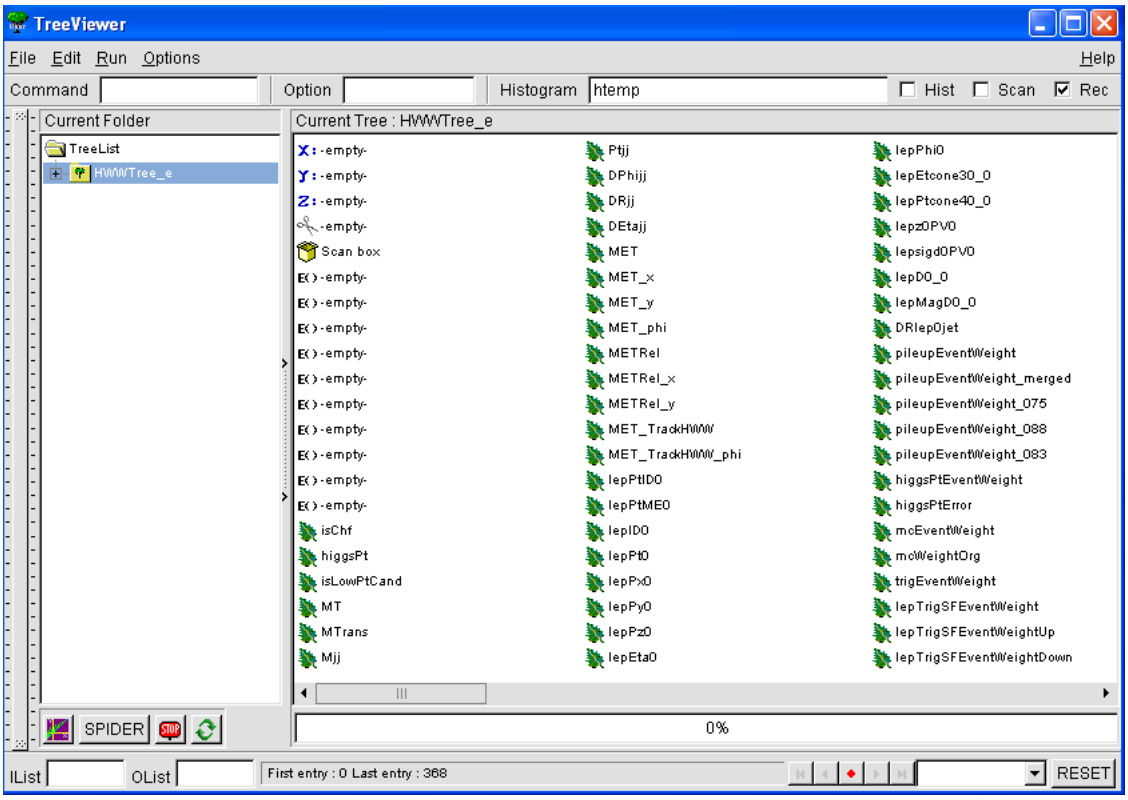
Các tree có thể được lưu trữ trên nhiều file khác nhau và có thể được nối tiếp với nhau trong quá trình xử lý thông qua một đối tượng duy nhất gọi là *chain*.

7.3 Cách tạo và truy xuất dữ liệu từ file root

7.3.1 Cách tạo file root

Các bước để tạo một file root như sau:

1. Tạo một file
   VD: `TFile *file = new TFile("file.root","RECREATE");`



Hình 7.1: Cấu trúc của tree

2. Tạo một tree
   VD: `TTree *tree = new TTree("Tree","Tree");`
   Nếu muốn tạo tree với thư mục bên trong ta có thể làm như sau: `TTree tree("Tree", "/folder")`

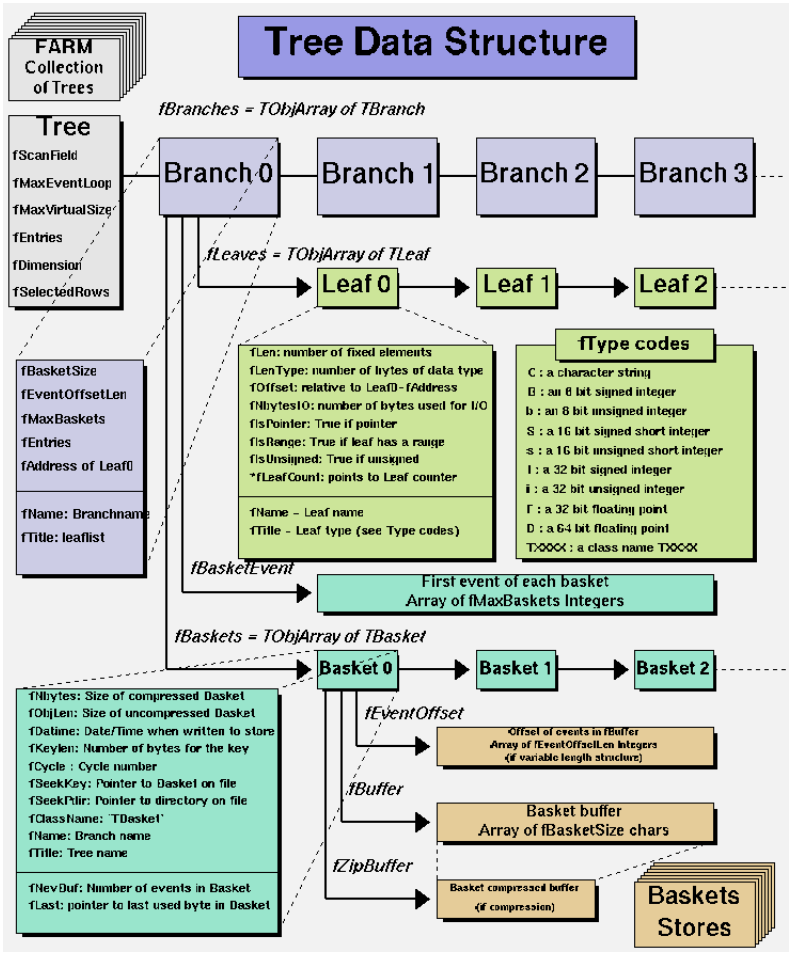
3. Thêm *branch* vào tree
   VD: `Event *event = new Event();`
   `tree->Branch("EventBranch","Event",&event);`

4. Lưu dữ liệu vào tree
   VD: `tree->Fill();`

5. Lưu vào file
   VD: `file->Write();`

Ví dụ:

```
{
  TFile f("example.root","recreate");
  TTree *t = new TTree("tree","tree");

  Float_t px, py, pz;
  t->Branch("px",&px,"px/F");
  t->Branch("py",&py,"py/F");
  t->Branch("pz",&pz,"pz/F");

  for (Int_t i=0; i<10000; i++) {
     gRandom->Rannor(px,py);
     pz = px*px + py*py;
     t->Fill();
  }
  t->Write();
```



Hình 7.2: Cấu trúc của tree

```
}
```

7.3.2 Cách truy xuất file root

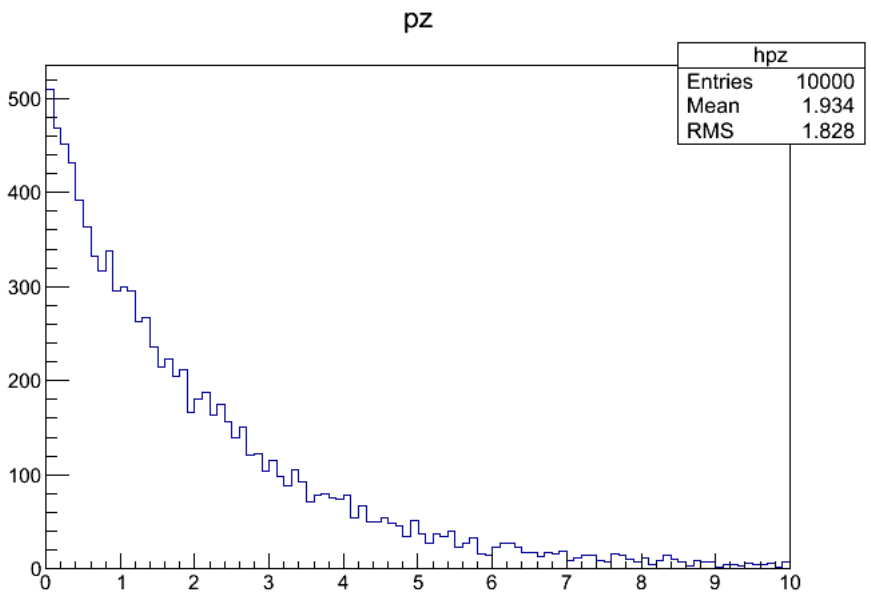
Các bước để truy xuất một file root như sau:

1. Mở một file
   VD: `TFile file("file.root");`
2. Đọc tree
   VD: `TTree * tree = (TTree*)file->FindObject("tree")`
3. Tạo các biến để lưu giữ liệu
   VD: `Float_t a, b, c;`
4. Liên kết *branch* với các biến
   VD: `tree->SetBranchAddress("a", &a)`
5. Đọc entry và lấy giá trị của biến
   VD: `tree->GetEntry(i); cout << a << endl;`

Ví dụ: *(xem Hình 7.3)*

```
{
  TFile *f = new TFile("example.root");
  TTree *t = (TTree*)f->Get("tree");
```

```
  Float_t px, py, pz;
  t->SetBranchAddress("pz",&pz);

  TH1F *hpz   = new TH1F("hpz","pz",100,0,10);
  Long64_t nentries = t->GetEntries();
  for (Long64_t i=0; i<nentries; i++) {
     t->GetEntry(i);
     hpz->Fill(pz);
  }
  hpz->Draw();
}
```



Hình 7.3: Ví dụ file root

Trong trường hợp muốn xem nhanh histogram của một vài biến, ta có thể sử dụng lệnh `Draw()`.

Ví dụ: *Câu lệnh vẽ histogram của sqrt(x) với các giá trị x > 0, histogram được đặt tên là histo*

```
tree->Draw("sqrt(x) >> hist", "x>0");
```

Ngoài ra, trong trường hợp tree được chia thành nhiều file, ta có thể dùng `TChain` để nối các file lại. Ví dụ:

```
TChain chain("tree");
chain->Add("file1.root");
chain->Add("file2.root");
...
```

7.4 Xử lý số liệu có cấu trúc dạng tree

Quy trình xử lý số liệu có cấu trúc dạng *tree* hoặc *ntuple* thông thường bao gồm ba bước sau

- Mở file số liệu, khai báo các biến, hitogram,...
- Tạo vòng lặp trên tất cả các dữ liệu (*entry*) chứa trong file, tiến hành tính toán giá trị, áp dụng các ngưỡng cắt, lưu giá trị vào histogram,...

- Lưu kết quả, biểu diễn kết quả,...

Để tiến hành xử lý dữ liệu có cấu trúc dạng *tree* hoặc *ntuple* chứa trong các file ROOT, việc đầu tiên ta cần làm là tạo ra một lớp xử lý (*analysis class*). Bộ khung sườn (*skeleton*) của lớp này được tạo ra thông qua lệnh `MakeClass()`.

```
root [0] TFile myFile("data.root")
root [1] tree->MakeClass("MyAnalysis")    // gia su doi tuong thuoc lop
   TTree co ten la tree
```

Lệnh `MakeClass()` sẽ tạo ra hai file *MyAnalysis.h* và *MyAnalysis.C* tương ứng với tên của lớp mà ta muốn tạo ra. File *MyAnalysis.h* là file header chứa các khai báo cho lớp mà ta tạo, trong file này cũng chứa danh sách các biến có trong file ROOT mà ta muốn xử lý.

Ví dụ: *MyAnalysis.h*

```
class MyAnalysis {
public :
  // Con tro doi tuong thuoc lop TTree hay TChain
  TTree          *fChain;
  // Tree hien tai trong TChain
  Int_t           fCurrent;
  // Cac bien
  UInt_t          fUniqueID;
  UInt_t          fBits;
  Char_t          fType[20];
  Int_t           fNtrack;
  Int_t           fNseg;
  Int_t           fNvertex;
  UInt_t          fFlag;
  // Cac nhanh (branch)
  TBranch        *b_fUniqueID;
  TBranch        *b_fBits;
  TBranch        *b_fType;
  TBranch        *b_fNtrack;
  TBranch        *b_fNseg;
  TBranch        *b_fNvertex;
  TBranch        *b_fFlag;
  ...
  MyClass(TTree *tree=0);
  ~MyClass();
  Int_t  Cut(Int_t entry);
  Int_t  GetEntry(Int_t entry);
  Int_t  LoadTree(Int_t entry);
  void   Init(TTree *tree);
  void   Loop();
  Bool_t Notify();
  void   Show(Int_t entry = -1);
};
```

Một số thành phần quan trọng trong lớp này gồm có:

- `fChain`: trỏ tới *tree* hay *chain* gốc (là *tree* hay *chain* mà lớp này được tạo ra từ đó).
- `fCurrent`: trỏ tới *tree* hay *chain* hiện tại đang xử lý, đặc biệt trong trường hợp xử lý nhiều *tree* liên kết lại với nhau qua `TChain`.
- `void Init(TTree *tree)`: khởi tạo *tree* để chuẩn bị đọc dữ liệu.
- `Int_t GetEntry(Int_t entry)`: đọc *entry* tương ứng, ví dụ `GetEntry(9)` sẽ đọc sự kiện thứ 10 ở trong *tree* (sự kiện đầu tiên có chỉ số là 0).

- `void Loop()`: hàm chứa các lệnh xử lý, hàm này sẽ lặp trên tất cả dữ liệu trong *tree* hay *chain* và áp các lệnh xử lý vào từng dữ liệu.

Ví dụ: *MyAnalysis.C (trong day quan trong nhat la ham Loop())*

```
void MyAnalysis::Loop()         {
//   In a ROOT session, you can do:
//      Root > .L MyClass.C
//      Root > MyClass t
//      Root > t.GetEntry(12); // Fill t data members with entry number 12
//      Root > t.Show();       // Show values of entry 12
//      Root > t.Show(16);     // Read and show values of entry 16
//      Root > t.Loop();       // Loop on all entries
//

//     This is the loop skeleton where:
//    jentry is the global entry number in the chain
//    ientry is the entry number in the current Tree
//  Note that the argument to GetEntry must be:
//    jentry for TChain::GetEntry
//    ientry for TTree::GetEntry and TBranch::GetEntry
//
//       To read only selected branches, Insert statements like:
// METHOD1:
//    fChain->SetBranchStatus("*",0);  // disable all branches
//    fChain->SetBranchStatus("branchname",1);  // activate branchname
// METHOD2: replace line
//    fChain->GetEntry(jentry);       //read all branches
//by  b_branchname->GetEntry(ientry); //read only this branch

 //  gROOT->Reset();

  if (fChain == 0) return;

  /* Cac khai bao viet o day */

  // Tong so entry trong chain
  Int_t nentries = Int_t(fChain->GetEntries());

  // Lap tren tat ca entry
  Int_t nbytes = 0, nb = 0;
  for (Int_t jentry=0; jentry<nentries;jentry++) {
    Int_t ientry = LoadTree(jentry);
    // trong truong hop TChain, 'ientry' la so entry trong file dang doc
    nb = fChain->GetEntry(jentry);   nbytes += nb;
    // if (Cut(ientry) < 0) continue;

    /* Doan code xu ly viet o day */

  }

  /* Doan code luu va trinh bay ket qua xu ly */

}
```

File *MyAnalysis.C* chứa các nội dung tương ứng với các hàm được khai báo trong *MyAnalysis.h*. Các xử lý số liệu đều được thực hiện trong hàm `Loop()`, các khai báo và lệnh tương ứng được đặt tại các vị trí như được chỉ ra trong ví dụ bên trên.

Ví dụ:

```
void Ana::Loop()          {
  if (fChain == 0) return;
  Int_t nentries = Int_t(fChain->GetEntries());

  TH1F *hist  = new TH1F("","", 100, 0, 10000);

  Int_t nbytes = 0, nb = 0;
  for (Int_t jentry=0; jentry<nentries;jentry++) {
    Int_t ientry = LoadTree(jentry);
    nb = fChain->GetEntry(jentry);   nbytes += nb;
    for(Int_t itrack=0;itrack<TrackP->size();++itrack) {
      if(TrackP->at(itrack) > 1000)
        hist->Fill(TrackP->at(itrack));
    }
  }
  hist->Draw();
}
```

Để thực thi file xử lý, ta thực hiện như sau

```
root [0] .L MyAnalysis.C
root [1] MyAnalysis a
root [2] a.Loop()
```

CHƯƠNG 8

BIÊN DỊCH THỰC THI ROOT

Bên cạnh việc thực thi các lệnh của ROOT thông qua giao diện tương tác dòng lệnh (*interactive session*) hay sử dụng các macro, chúng ta còn có thể biên dịch chúng thành các chương trình độc lập để có thể rút ngắn thời gian chạy chương trình. Trong chương này, các bạn sẽ được hướng dẫn cách thức biên dịch các mã lệnh của ROOT thành một chương trình độc lập, cũng như cách liên kết với các module ngoài được xây dựng bằng các ngôn ngữ lập trình C++ và Python.

8.1 Biên dịch chương trình C++ với ROOT

8.1.1 Trình biên dịch

Trong hầu hết các phiên bản Linux, trình biên dịch được mặc định cài đặt sẵn là GCC (*GNU Compiler Collection*), đây là một bộ các trình biên dịch có khả năng biên dịch nhiều ngôn ngữ khác nhau như C (`gcc`), C++ (`g++`), Fortran (`gfortran`),...

Tên gốc của GCC là *GNU C Compiler* do ban đầu nó chỉ hỗ trợ dịch ngôn ngữ lập trình C. Phiên bản đầu tiên GCC 1.0 được phát hành vào năm 1987, sau đó được mở rộng hỗ trợ dịch C++ vào tháng 12 cùng năm đó. Sau đó, GCC được phát triển cho các ngôn ngữ lập trình Fortran, Pascal, Objective C, Java, and Ada,... Bảng dưới trình bày một số trình biên dịch thông dụng nhất trong GCC.

Ngôn ngữ	Trình biên dịch
C	`gcc`
C++	`g++`
Fortran	`gfortran`
Pascal	`gpc`
Java	`gcj`
Ada	`gnat`
D	`gdc`
VHDL	`ghdl`

Cú pháp để biên dịch một chương trình như sau

```
compiler [options] <source codes>
```

Các tùy chọn để điều khiển quá trình biên dịch

`-c`	tạo ra tập tin đối tượng (.o)
`-o filename`	tạo ra tập tin output có tên là filename
`-g`	biên dịch ở chế độ debug (báo lỗi khi có lỗi xảy ra)
`-Wall`	hiển thị thông điệp cảnh báo (*warning*)
`-l directory`	thêm thư viện trong quá trình biên dịch
`-I directory`	thêm thư mục chứa các header cần thiết trong quá trình biên dịch
`-L directory`	thêm thư mục chứa các thư viện cần thiết trong quá trình biên dịch
`-On`	biên dịch với chế độ tối ưu, n = 1,2,3 (thông thường là 2)

Các trình biên dịch thường được đặt trong thư mục `/usr/bin` hay `/usr/local/bin`. Trong quá trình biên dịch, nếu không có yêu cầu cụ thể, trình biên dịch sẽ mặc định tìm kiếm các tập tin header và thư viện trong cùng thư mục với tập tin mã nguồn và các thư mục như `/usr/include` hay `/usr/lib`.

Ví dụ: đối với mã nguồn được viết bằng ngôn ngữ C++, ta sẽ sử dụng trình biên dịch `g++`

```
$ g++ -o helloworld helloworld.cpp
```

8.1.2 Biên dịch macro với ROOT

Để biên dịch một macro ROOT thành một chương trình độc lập (*stand alone*) chúng ta cần phải biên dịch với một trình biên dịch C/C++, trong trường hợp này chúng ta sẽ sử dụng `g++`. Trước khi biên dịch, cần phải chắc chắn là tất cả các file header cần thiết đều đã được include. Để làm được điều đó, chúng ta sử dụng công cụ *root-config*, đây là một *script* của ROOT cung cấp tất cả các flag và thư viện cần thiết để có thể biên dịch được chương trình và liên kết chúng tới các thư viện của ROOT. Lưu ý rằng để có thể tạo ra một chương trình độc lập, chúng ta cần khai báo hàm *main()* trong code. Hàm *main()* có thể được tạo ra chỉ với một nhiệm vụ duy nhất là gọi hàm macro.

Ví dụ:

```
#ifndef __CINT__
int main() {
    Macro();
    return 0;
}
#endif
```

Khi chạy chương trình trong ROOT, `__CINT__` được xác định thì đoạn code trên sẽ không được thực thi, ngược lại khi biên dịch với `g++` CINT không được xác định thì đoạn code trên sẽ được thực thi.

Để biên dịch một chương trình độc lập từ file *Macro.C* ta chỉ cần gõ đoạn lệnh:

```
g++ -o Macro.exe Macro.C `root-config --cflags --libs`
```

8.1.3 Biên dịch với Makefile

Makefile là một tập tin đặc biệt dùng để mô tả và quản lý quá trình biên dịch các tập tin trong một dự án (*project*). Tập tin này chứa các quy tắc biên dịch và xây dựng một đồ thị phụ thuộc giữa các tập tin, thư viện trong cùng dự án.

Để biên dịch với Makefile ta sử dụng lệnh `make`, lệnh này sẽ đọc các bước biên dịch trong Makefile để dịch và sinh ra chương trình thực thi.

```
$ make
```

Cấu trúc tập tin Makefile gồm nhiều khối thực hiện, mỗi khối gồm có các thông tin về các thành phần phụ thuộc (*dependencies*) và cách thức hay quy tắc thực hiện biên dịch (*rule*).

Quy tắc tạo một khối trong Makefile như sau

```
target : dependencies
<tab>commands
```

- **Target**: tên của tập tin được tạo ra bởi chương trình hoặc là các chỉ định để thực thi một hoặc một loạt tác vụ nào đó, các *target* không được bắt đầu bằng dấu '.'
- **Dependencies**: các tập tin đầu vào hoặc phụ thuộc để tạo ra *target*, khi các tập tin này thay đổi (do chỉnh sửa mã nguồn hoặc thời gian lưu bị thay đổi) thì *target* cần được biên dịch lại.
- **Commands**: các lệnh mà trình `make` sẽ thực thi. Một quy tắc (*rule*) có thể có nhiều lệnh, mỗi lệnh thường được viết trên một dòng, trước mỗi dòng cần phải có dấu *tab*.

Ví dụ Makefile Giả sử ta có một lớp `MyClass` đã được xây dựng sẵn bằng ngôn ngữ lập trình C++, lớp này được định nghĩa trong hai tập tin `MyClass.h` và `MyClass.cpp`. Ta viết một chương trình có sử dụng lớp `MyClass` (tập tin *main.cpp*), để tạo tập tin thực thi cho chương trình này ta có thể thực hiện việc biên dịch lần lượt như sau

```
$ g++ -c MyClass.cpp
$ g++ -c main.cpp
$ g++ main.o MyClass.o -o myprogram
```

Dòng cuối cùng liên kết hai tập tin đã được biên dịch lại tạo thành tập tin thực thi `myprogram` mà ta có thể chạy được qua lệnh

```
$ ./myprogram
```

Tuy nhiên, trong ví dụ này thay vì thực hiện biên dịch theo cách trên, ta sẽ viết một Makefile để biên dịch tự động chương trình này.

Đầu tiên, ta sử dụng trình soạn thảo văn bản tạo một tập tin có tên là *Makefile*

```
$ gedit Makefile
```

Trong tập tin này, ta khai báo trình biên dịch và các tùy chỉnh cần thiết

```
CC = g++
FLAGS = -c -g -Wall
```

Khai báo các đối tượng (*object*) cần biên dịch, trong trường hợp này ta cần hai đối tượng là `main.o` và `MyClass.o`

```
main.o : main.cpp MyClass.h
   $(CC) $(FLAGS) main.cpp
MyClass.o : MyClass.cpp MyClass.h
   $(CC) $(FLAGS) MyClass.cpp
```

Khai báo tập tin thực thi và phương thức liên kết

```
myprogram : main.o MyClass.o
   $(CC) main.o MyClass.o -o myprogram
```

Khai báo thêm tùy chỉnh `clean` cho lệnh `make` để xóa các tập tin đã được biên dịch

```
clean:
   rm -f *.o myprogram
```

Makefile hoàn chỉnh có nội dung như sau

```
CC = g++
FLAGS = -c -g -Wall

main.o : main.cpp MyClass.h
   $(CC) $(FLAGS) main.cpp
MyClass.o : MyClass.cpp MyClass.h
   $(CC) $(FLAGS) MyClass.cpp
myprogram : main.o MyClass.o
   $(CC) main.o MyClass.o -o myprogram
clean:
   rm -f *.o myprogram
```

Trong trường hợp biên dịch với ROOT, ta thêm các flag sau

```
CFLAGS=-c -g -Wall `root-config --cflags`
LDFLAGS=`root-config --libs`
```

Để thực thi việc biên dịch với *Makefile*, tại dấu nhắc ta gõ

```
$ make
```

Để xóa tất cả các tập tin vừa được biên dịch, ta gõ

```
$ make clean
```

8.2 Cách tổ chức mã nguồn của C++ và ROOT

Đối với một chương trình C++ đơn giản thì toàn bộ nội dung của chương trình có thể được chứa trong một file duy nhất. Tuy nhiên đối với các chương trình lớn có cấu trúc phức tạp, thường là các dự án (*project*), ta cần phải chia nhỏ mã nguồn ra thành nhiều file để quản lý. Việc chia nhỏ này có nhiều lợi ích

- *Tăng tốc độ biên dịch*: hầu hết các trình biên dịch làm việc trên 1 file trong 1 lúc, do đó cho dù ta chỉ thực hiện một thay đổi nhỏ trong file, ta cũng phải biên dịch lại toàn bộ file. Mặt khác, nếu ta chia mã nguồn ra thành nhiều file, thì chỉ yêu cầu file có sự thay đổi được biên dịch lại.
- *Tăng cường sự tổ chức*: việc phân chia mã nguồn một cách hợp lý sẽ giúp ta dễ dàng hơn trong việc tìm kiếm những hàm, biến, định nghĩa các cấu trúc/lớp,... đặc biệt khi ta cần xem lại 1 đoạn mã để tìm kiếm cái gì đó.
- *Giảm bớt sự viết lại*: nếu mã nguồn được phân chia cẩn thận thành những đoạn độc lập với nhau, ta có thể sử dụng lại chúng trong các dự án khác, tiết kiệm thời gian phải viết lại lần sau.
- *Chia sẻ mã nguồn giữa các dự án*: tương tự như việc giảm bớt sự viết lại, lợi ích của việc chia sẽ file này so với sử dụng copy-paste là bất cứ lỗi nào được sửa trong 1 hay nhiều file trong 1 dự án sẽ có tác dụng với những dự án khác, và tất cả các dự án có thể chắc chắn được cập nhật bản mới nhất.
- *Phân chia trách nhiệm giữa các lập trình viên*: trong những dự án lớn thật sự thường có nhiều hơn 1 lập trình viên cùng viết chung mã nguồn. Do đó, ta sẽ phải chia làm nhiều file để mỗi lập trình viên có thể làm việc trên từng phần riêng biệt mà không ảnh hưởng đến các lập trình viên khác.

File header Thông thường mã nguồn được phân chia theo dạng các 'module' (đôi khi 1 module có thể chia ra thành 2 hay nhiều file), tạo những file mới với những tên có nghĩa sẽ giúp người dùng biết nội dung bên trong khi nhìn lướt qua. Thông thường trong C++, người ta chia mỗi lớp vào một file riêng, như vậy sẽ tiện hơn thay vì cho tất cả vào một chỗ, và các file này thường có tên trùng với tên của lớp. Để tách theo cách này, ta cần sử dụng các file *header* (thường có đuôi *.h*, *.hpp*, *.hxx*), đây là các file cho phép định nghĩa các thành phần của chương trình ở các file riêng, và khi cần sử dụng lại thì có thể gọi dễ dàng bằng cách đưa vào chương trình qua lệnh `#include`.

Nội dung của file header thường gồm có

- Các chỉ thị tiền xử lý
- Định nghĩa lớp hoặc cấu trúc, template
- Định nghĩa kiểu `typedef`
- Định nghĩa hàm
- Biến toàn cục
- Hằng số

Giả sử chúng ta muốn định nghĩa một lớp `MyClass`, ta sẽ tạo một file header *MyClass.h* có nội dung như sau

```
class MyClass {
public:
  void foo();
  int bar;
};
```

File mã nguồn *MyClass.cpp* có nội dung

```
#include "MyClass.h"

void MyClass::foo() {
  std::cout << "Hello" << std::endl;
}
```

Mỗi file mã nguồn của lớp `MyClass` cần phải `#include "MyClass.h"`. Lưu ý rằng ta nên dùng ngoặc kép " " hơn là ngoặc nhọn <> khi include những file header, dấu ngoặc kép sẽ cho trình biên dịch biết phải tìm kiếm header trong thư mục chương trình trước, rồi mới đến các header chuẩn của trình biên dịch.

File mã nguồn chính *main.cpp* có nội dung

```
#include "MyClass.h"  // dinh nghia MyClass

int main()
{
  MyClass a;
  a.foo();
  return 0;
}
```

Những file header sẽ trở thành phần chung giữa các module hay lớp con. Bằng cách `#include` một header, ta có thể truy xuất đến toàn bộ những định nghĩa cấu trúc, tên hàm, hàng số... của module hoặc lớp tương ứng.

Một số lỗi thường gặp khi biên dịch với header[1]

- *Không tìm thấy định nghĩa cần thiết*: điều này sẽ làm cho file mã nguồn không biên dịch được bởi vì có một số những định nghĩa chưa được khai báo do không được include từ các file header. Giả sử ta có 3 file (*Header1.h*, *Header2.h* và *File.cpp*) như sau

```
/* Header1.h */
class ClassOne { ... };

/* Header2.h */
#include "Header1.h"
class ClassTwo { ... };

/* File.cpp */
#include "Header2.h"
ClassOne myClassOne;
ClassTwo myClassTwo;
```

  Trong trường hợp này, *File.cpp* sẽ biên dịch tốt, vì đã include *Header2.h* và gián tiếp include *Header1.h*, có nghĩa là *File.cpp* có thể truy xuất đến lớp `ClassOne`. Nhưng nếu một thời gian sau, ai đó cho rằng *Header2.h* không cần thiết phải include *Header1.h* và xoá dòng include đó đi, hậu quả là *File.cpp* sẽ không thể biên dịch được. Do đó, ta cần phải dứt khoát khi include bất kì header nào cần cho file mã nguồn để biên dịch, không nên chỉ dựa vào những file header include gián tiếp mà có thể sẽ thay đổi.

  Tuy nhiên việc này đôi khi lại dẫn tới một số lỗi sẽ được trình bày tiếp theo sau.

- *Phụ thuộc vòng tròn*: khi những header xuất hiện khi cần include lẫn nhau để làm việc một cách có dụng ý. Chẳng hạn như ta có hai lớp `ClassOne` và `ClassTwo` phụ thuộc lẫn nhau

```
/* Header1.h */
#include "Header2.h"
class ClassOne { ClassTwo two; };

/* Header2.h */
#include "Header1.h"
class ClassTwo { ClassOne one; };
```

  Thật ra lớp `ClassOne` không cần phải biết chi tiết của lớp `ClassTwo`, do nó chỉ sử dụng con trỏ của lớp `ClassTwo` chứ không sử dụng toàn bộ đối tượng của lớp này. Con trỏ không cần biết nó chỉ đến đâu, do đó ta không cần phải định nghĩa cấu trúc hoặc lớp để chứa con trỏ. Điều này có nghĩa là dòng `#include` ở đây là không cần thiết.

  Tuy nhiên, khi trình biên dịch hoạt động, khi biên dịch lớp `ClassOne` nó sẽ cần tới `ClassTwo` mà lúc này chưa được biên dịch do nó cũng cần tới `ClassOne` được biên dịch (nên nhớ là trình biên dịch mỗi lúc chỉ hoạt động với 1 file), do đó sẽ gây ra lỗi *"Undeclared identifier"*. Để khắc phục lỗi này, thay vì sử dụng include, ta sẽ sử dụng khai báo trước (*forward declaration*)

```
/* Header1.h */
class ClassTwo;  // khai bao truoc ClassTwo
class ClassOne { ClassTwo two; };

/* Header2.h */
class ClassOne;  // khai bao truoc ClassOne
class ClassTwo { ClassOne one; };
```

[1] Cách thức biên dịch chương trình với file header được hướng dẫn ở Phần 6 trong tài liệu ***"Nhập môn hệ điều hành Linux"*** (http://goo.gl/11gQjF)

- *Định nghĩa chồng*: khi 1 lớp hoặc cấu trúc được gọi 2 lần trong 1 file nguồn. Điều này sẽ gây ra lỗi trong thời gian biên dịch (*compile-time error*) và thường xuất hiện khi gọi nhiều file header trong 1 file header khác, làm cho header được gọi 2 lần khi bạn biên dịch file nguồn. Để khắc phục được lỗi này ta sử dụng các *include guard* để đảm bảo rằng một header file chỉ được include một lần trong chương trình. Chẳng hạn như file *MyClass.h* lúc này sẽ có nội dung

```
#ifndef MYCLASS_H  // neu MyClass chua duoc dinh nghia
#define MYCLASS_H  // thi dinh nghia MyClass

class MyClass {
public:
  void foo();
  int bar;
};

#endif
```

8.3 ROOT và Python

Tương tự như C++, ROOT cũng cần liên kết với các module Python để xây dựng nên các gói xử lý số liệu. Phần này sẽ trình bày một cách sơ lược cách sử dụng Python trong ROOT và ngược lại.

8.3.1 PyROOT

PyROOT là một module Python cho phép người dùng tương tác với các lớp của ROOT từ trình thông dịch Python. Việc này cũng tương tự như các *ROOT dictionary* nên không cần phải tạo các *Python wrapper* để đưa các lớp của ROOT vào. Đồng thời PyROOT cũng cho phép người dùng thực thi các lệnh của Python bên trong trình thông dịch ROOT/CINT.

Để cài đặt PyROOT, ta làm theo các hướng dẫn ở đây http://wlav.web.cern.ch/wlav/pyroot/installation.html.

Sau khi đã cài đặt PyROOT, ta có thể sử dụng lệnh `import` để nhập các lớp của ROOT mà ta muốn dùng vào trong Python, chẳng hạn như đoạn code sau

```
from ROOT import gROOT, TCanvas, TF1

gROOT.Reset()
c1 = TCanvas( 'c1', 'Example with Formula', 200, 10, 700, 500 )

# Tao mot ham 1-chieu va ve no
fun1 = TF1( 'fun1', 'abs(sin(x)/x)', 0, 10 )
c1.SetGridx()
c1.SetGridy()
fun1.Draw()
c1.Update()
```

Trong trường hợp ta muốn đưa một lớp tự tạo vào trong Python, trước tiên ta cần tạo thư viện từ lớp đó thông qua ACLIC. Ví dụ: giả sử ta có file *MyClass.C* có nội dung

```
class MyClass {
public:
MyClass(int value = 0) {
  m_value = value;
}
void SetValue(int value) {
```

```
  m_value = value;
}
int GetValue() {
  return m_value;
}
private:
  int m_value;
};
```

Thư viện được tạo ra bằng lệnh

```
$ echo .L MyClass.C+ | root.exe -b
```

Sau đó load từ Python

```
from ROOT import gSystem
# load thu vien MyClass
gSystem.Load('MyClass_C')
# import MyClass tu ROOT
from ROOT import MyClass
# su dung lop MyClass
m = MyClass(42)
print m.GetValue()
```

Ngoài ra, ta cũng có thể load trực tiếp file macro mà không cần phải tạo thư viện, nhưng điều này sẽ giới hạn chúng ta chỉ được sử dụng hàm khởi tạo mặc định của lớp

```
from ROOT import gROOT
gROOT.LoadMacro('MyClass.C')
```

Chi tiết cách sử dụng PyROOT có thể được tham khảo tại http://wlav.web.cern.ch/wlav/pyroot/.

8.3.2 Sử dụng Python trong ROOT

Ta có thể sử dụng Python trong ROOT bằng cách sử dụng thư viện của PyROOT (lớp `TPython`[2]), chẳng hạn như

```
root [0] gSystem->Load("libPyROOT")
root [1] TPython::Exec( "print 1 + 1" )
2
root [2] TBrowser *b = (void*)TPython::Eval( "ROOT.TBrowser()" )
(class TObject*)0x8d1daa0
root [3] TPython::Prompt()  // su dung shell Python
>>> i = 2
root [4] TPython::Prompt()  // su dung shell Python
>>> print i
2
```

[2] Xem thêm tại http://root.cern.ch/root/html/TPython.html

CHƯƠNG 9

XỬ LÝ PHỔ GAMMA VỚI ROOT

Chương này sẽ hướng dẫn các bạn một số cách thức cơ bản đọc một file phổ dạng text và xử lý phổ với công cụ ROOT. Mặc dù chỉ chủ yếu tập trung vào cách thức xử lý phổ gamma nhưng các bạn hoàn toàn có thể áp dụng các kiến thức đọc được vào trong việc xử lý các loại phổ khác một khi đã nắm được cách thức hoạt động của chương trình.

9.1 Đọc file phổ

Thông thường các phổ sẽ được lưu lại dưới dạng các file số liệu (phần lớn là dạng text nếu kích thước dữ liệu không quá lớn) để tiện cho việc xử lý offline. Trong phần này chúng ta sẽ học cách đọc phổ từ một file dạng text và lưu vào trong histogram để phục vụ cho các bước xử lý tiếp theo.

Để tiện cho việc minh họa, ta sẽ sử dụng một file phổ gamma của nguồn ^{60}Co 1024 kênh (download tại http://goo.gl/wQeh2m). Phổ gamma này có hai đỉnh tương ứng với năng lượng 1173 và $1332keV$.

Đầu tiên, chúng ta sẽ tạo một file macro để xử lý phổ có tên là *pho_gamma.c*, Trong file macro này, ta tạo một hàm có tên `pho_gamma()` (lưu ý rằng tên hàm phải trùng với tên của file)

```
void pho_gamma()
{

}
```

Kế tiếp chúng ta sẽ khai báo một histogram 1-chiều để chứa phổ

```
void pho_gamma()
{
  // Khai bao histogram
  TH1F *h1 = new TH1F("h1", "Pho Co-60 1024 kenh", 1024, 1, 1024);
}
```

Bộ ba tham số cuối cùng `(1024,1,1024)` dùng để khai báo lần lượt số kênh (1024), giá trị ứng với kênh đầu tiên (1) và kênh cuối cùng (1024).

Trong trường hợp ta muốn vẽ phổ theo năng lượng ta có thể khai báo như sau

```
double nang_luong(int kenh)
{
  const double A = 1.0;  // gia tri vi du
  const double B = 0.0;  // gia tri vi du
  return (A*kenh+B);
}

void pho_gamma()
{
  // Khai bao histogram
  TH1F *h1 = new TH1F("h1", "Pho Co-60 1024 kenh", 1024, nang_luong(1),
      nang_luong(1024));
}
```

với `nang_luong(kenh)` là hàm trả về giá trị năng lượng tương ứng với kênh dựa vào đường chuẩn năng lượng ta đã chuẩn trước đó.

Sau khi đã hoàn thành việc khai báo histogram, công việc tiếp theo chúng ta sẽ là tiến hành mở file để đọc số liệu, có 2 cách để thực hiện việc này

- Cách 1: sử dụng thư viện *stdio*

```
// Mo file pho Co60.dat
FILE *file = fopen("Co60.dat","r");

// Doc file pho
int i = 0;
double sodem;
while (!feof(file)) {
  fscanf(file,"%lf",&sodem);
  h1->SetBinContent(++i,sodem);
}

// Dong file pho
fclose(file);
```

- Cách 2: sử dụng thư viện *iostream*

```
// Mo file pho Co60.dat
ifstream file("Co60.dat");

// Doc file pho
int i = 0;
double sodem;
while (!file.eof()) {
  file >> sodem;
  h1->SetBinContent(++i,sodem);
}

// Dong file pho
file.close();
```

Lưu ý:

- Trong một số trường hợp, dữ liệu phổ có chứa cả những thông tin về detector, nguồn (mẫu) đo, thời gian đo,... (thường nằm ở đầu file), do đó chúng ta có thể sẽ phải thay đổi cách đọc file cho phù hợp với từng trường hợp cụ thể.
- Do số liệu phổ gamma đã ghi nhận sẵn số đếm theo kênh nên ta sử dụng `SetBinContent()` để gán thẳng giá trị cho kênh tương ứng. Trong trường hợp số liệu thô (chỉ có giá trị năng

lượng hoặc điện thế ghi nhận, chưa sắp xếp số đếm theo kênh), ta có thể sử dụng phương thức `Fill()` để ghi nhận giá trị cho kênh tương ứng, vd:

```
h1->Fill(nangluong);
```

Trong trường hợp cần vẽ phổ, ta gõ

```
h1->Draw();
```

Nếu muốn thêm thông tin của hai trục x và y khi vẽ phổ, ta thêm các dòng lệnh sau trước dòng lệnh `h1->Draw()`

```
h1->GetXaxis()->SetTitle("Channel");
h1->GetYaxis()->SetTitle("Number of entries");
```

Sau khi thực hiện xong tất cả các bước liệt kê, file macro *pho_gamma.c* điển hình có nội dung như sau

```
void pho_gamma()
{
  // Khai bao histogram
  TH1F *h1 = new TH1F("h1", "Pho Co-60 1024 kenh",1024, 1, 1024);

  // Mo file pho Co60.dat
  ifstream file("Co60.dat");

  // Doc file pho
  int i = 0;
  double sodem;
  while (!file.eof()) {
    file >> sodem;
    h1->SetBinContent(++i,sodem);
  }

  // Dong file pho
  file.close();

  // Ve pho
  h1->GetXaxis()->SetTitle("Channel");
  h1->GetYaxis()->SetTitle("Number of entries");
  h1->Draw();
}
```

Để thực thi file macro trong ROOT, tại dấu nhắc ta gõ lệnh

```
> root -l -q pho_gamma.c
```

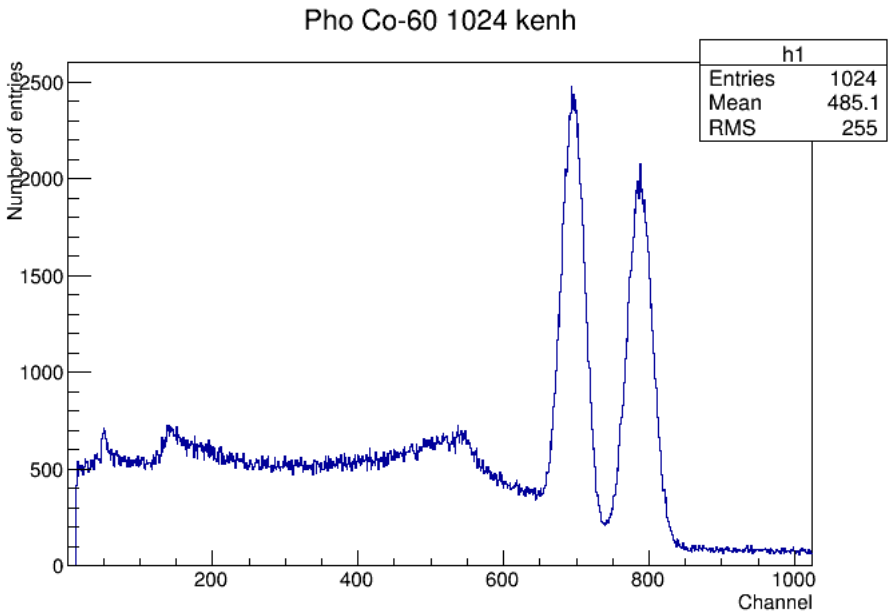
Kết quả vẽ phổ được cho như trong Hình 9.1.

9.2 Làm khớp đỉnh phổ và phông nền

9.2.1 Làm khớp với 1 đỉnh

Giả sử ta muốn làm khớp đỉnh năng lượng $1173keV$ theo dạng Gaussian

$$f(x) = Aexp\left[-\frac{(x-x_0)^2}{2\sigma^2}\right] \tag{9.1}$$



Hình 9.1: Phổ của nguồn ^{60}Co 1024 kênh

trong đó A là độ cao đỉnh, x_0 là vị trí của đỉnh và σ là độ lệch chuẩn của phân bố đỉnh. Và phông nền có dạng tuyến tính (đa thức bậc nhất)

$$g(x) = ax + b \tag{9.2}$$

Ta khai báo hai hàm đỉnh và phông nền trong ROOT như sau

```
// Ham mo ta dinh dang Gaussian
TF1 *dinh = new TF1("dinh", "[0]*TMath::Gaus(x,[1],[2])", min, max);
// Ham mo ta phong nen dang da thuc bac 1
TF1 *phongnen = new TF1("phongnen", "[0]+[1]*x", min, max);
```

Chúng ta định nghĩa hai hàm `dinh` (3 tham số)và `phongnen` (2 tham số) tương ứng với dạng của đỉnh và phông nền mà ta cần làm khớp. Hai giá trị `min` và `max` tương ứng với khoảng giá trị (khoảng kênh) mà ta cần làm khớp. Ngoài ra ta cũng có thể sử dụng các hàm tích hợp có sẵn trong `TF1` để khai báo

```
// Ham mo ta dinh dang Gaussian
TF1 *dinh = new TF1("dinh", "gaus", min, max);
// Ham mo ta phong nen dang da thuc bac 1
TF1 *phongnen = new TF1("phongnen", "pol1", min, max);
```

Kế đó, ta sẽ định nghĩa hàm cần làm khớp là tổng của hai hàm này

```
// Ham lam khop bang tong cua ham dinh va phong nen
TF1 *f = new TF1("f", "dinh+phongnen", min, max);
```

Hàm `f` sẽ có tất cả 5 tham số, trong đó ba tham số đầu tiên (có chỉ số từ 0 đến 2) sẽ tương ứng với các tham số của đỉnh và 2 tham số cuối sẽ tương ứng với phông nền.

Trong trường hợp muốn cung cấp giá trị ước lượng (vị trí đỉnh, độ rộng đỉnh,...) cho các tham số trong quá trình làm khớp, ta có thể sử dụng hàm `SetParameter()`

```
// Tham so dau vao
f->SetParameter(1,680);  // uoc luong vi tri dinh (tham so co chi so 1)
f->SetParameter(2,10);  // uoc luong be rong dinh (tham so co chi so 2)
```

Để làm khớp histogram, ta gõ

```
  h1->Fit("f");
```

Sau khi thêm phần làm khớp, file *pho_gamma.c* có nội dung như sau

```
void pho_gamma()
{
  TH1F *h1 = new TH1F("h1", "Pho Co-60 1024 kenh",1024, 1, 1024);
  ifstream file("Co60.dat");

  int i = 0;
  double sodem;
  while (!file.eof()) {
    file >> sodem;
    h1->SetBinContent(++i,sodem);
  }
  file.close();

  // Khai bao ham can lam khop
  double min = 650, max = 730;
  TF1 *dinh = new TF1("dinh", "[0]*TMath::Gaus(x,[1],[2])",min, max);
  TF1 *phongnen = new TF1("phongnen", "[0]+[1]*x", min, max);
  TF1 *f = new TF1("f", "dinh+phongnen", min, max);

  // Tien hanh lam khop
  f->SetParameter(1,680);  // uoc luong vi tri dinh
  f->SetParameter(2,10);  // uoc luong do rong dinh
  h1->Fit("f");
}
```

Kết quả làm khớp[1] được cho dưới đây và trong Hình 9.2

```
 FCN=190921 FROM MIGRAD    STATUS=CONVERGED     388 CALLS         389 TOTAL
                     EDM=6.14761e-09    STRATEGY= 1   ERROR MATRIX UNCERTAINTY
                         1.9 per cent
  EXT PARAMETER                                   STEP         FIRST
  NO.   NAME      VALUE            ERROR          SIZE      DERIVATIVE
   1  p0           2.16985e+03   1.02833e+01  -7.58175e-03   1.92097e-06
   2  p1           6.94718e+02   6.69816e-02  -5.47668e-07  -6.69323e-04
   3  p2           1.58867e+01   5.90663e-02   8.79692e-05   1.05771e-03
   4  p3           4.55614e+02   1.09715e+00   7.77381e-04   7.57560e-06
   5  p4          -3.34368e-01   1.45069e-03  -9.55275e-07  -8.72390e-02
```

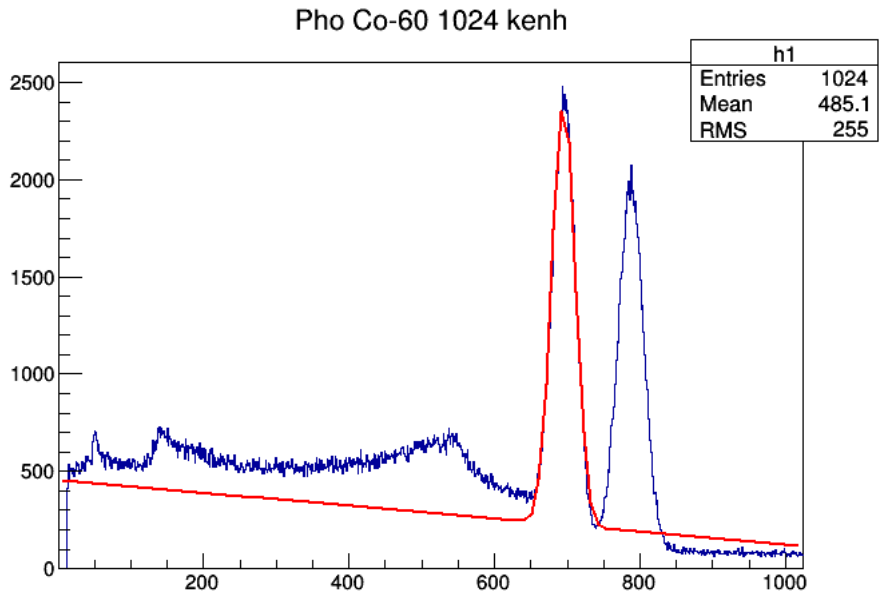
Trong đó, p0,...,p4 là các tham số được làm khớp. Để truy xuất các tham số được làm khớp, ta sử dụng hàm `GetParameters()`

```
  float *par;
  f->GetParameters(par);
```

Làm khớp với hàm do người dùng tự định nghĩa Trong trường hợp sử dụng hàm do người dùng tự định nghĩa, trước tiên ta cần phải khai báo các dạng của hàm cần làm khớp

```
// Ham mo ta dinh dang Gaussian
double dinh(double *x, double *par) {
  return (par[0]*TMath::Gaus(x[0],par[1],par[2]));
}
```

[1] Công cụ được sử dụng để làm khớp ở đây là lớp `TMinuit`



Hình 9.2: Kết quả làm khớp đỉnh 1173keV

```
// Ham mo ta phong nen dang da thuc bac 1
double phongnen(double *x, double *par) {
  return par[0] + par[1]*x[0];
}
```

Hàm cần làm khớp là tổng của hai hàm `dinh()` và `phongnen()`

```
// Ham lam khop bang tong cua ham dinh Gaussian va phong nen
double tong(double *x, double *par) {
  return (dinh(x,par)+phongnen(x,&par[3]));
}
```

Hàm `tong()` có tất cả 5 tham số, trong đó 3 tham số đầu (có chỉ số từ 0 đến 2) là các tham số mô tả đỉnh Gaussian và 2 tham số cuối (có chỉ số 3 và 4) mô tả phông nền. Do đó trong ta chỉ cần khai báo `dinh(x,par)` (tương đương với `dinh(x,&par[0])`, còn đối với hàm phông nền ta cần khai báo `phongnen(x,&par[3])` để gán tham số thứ tư (có chỉ số 3) cho tham số đầu tiên của hàm phông nền.

Trong hàm `pho_gamma()` ta khai báo một đối tượng thuộc lớp `TF1` tương ứng với hàm cần làm khớp

```
  // Tao ham f trong khoang (min,max) co 5 tham so (3 cho dinh, 2 cho
     phong nen)
  TF1 *f = new TF1("f", tong, min, max, 5);
```

Sau khi định nghĩa hàm `f`, cách thức làm khớp cũng tương tự như hướng dẫn ở trên.

9.2.2 Làm khớp với nhiều đỉnh

Tương tự như với cách thức làm khớp một đỉnh nhưng thay vì chỉ khai báo 1 hàm cho 1 đỉnh duy nhất, ta sẽ khai báo thêm các hàm đỉnh nữa tương ứng với số lượng đỉnh cần làm khớp.

Trong ví dụ này, ta sẽ tiến hành làm khớp hai đỉnh 1173 và 1332keV cùng lúc.

```
void pho_gamma()
{
```

```
  TH1F *h1 = new TH1F("h1", "Pho Co-60 1024 kenh",1024, 1, 1024);
  ifstream file("Co60.dat");

  int i = 0;
  double sodem;
  while (!file.eof()) {
    file >> sodem;
    h1->SetBinContent(++i,sodem);
  }
  file.close();

  // Khai bao ham can lam khop
  double min = 650, max = 840;
  TF1 *dinh1 = new TF1("dinh1", "[0]*TMath::Gaus(x,[1],[2])",min, max);
  TF1 *dinh2 = new TF1("dinh2", "[0]*TMath::Gaus(x,[1],[2])",min, max);
  TF1 *phongnen = new TF1("phongnen", "[0]+[1]*x", min, max);
  TF1 *f = new TF1("f", "dinh1+dinh2+phongnen", min, max);

  // Tien hanh lam khop
  f->SetParameter(1,680);  // uoc luong vi tri dinh 1
  f->SetParameter(2,10);  // uoc luong do rong dinh 1
  f->SetParameter(4,800);  // uoc luong vi tri dinh 2
  f->SetParameter(5,10);  // uoc luong do rong dinh 2
  h1->Fit("f");
}
```

Kết quả thu được như sau

```
 FCN=129525 FROM MIGRAD    STATUS=CONVERGED     778 CALLS          779 TOTAL
                     EDM=1.28668e-07    STRATEGY= 1  ERROR MATRIX UNCERTAINTY
                         1.7 per cent
  EXT PARAMETER                                   STEP         FIRST
  NO.   NAME      VALUE            ERROR          SIZE      DERIVATIVE
   1  p0           2.18317e+03   1.01023e+01   1.19452e-02   4.64791e-05
   2  p1           6.94656e+02   6.54987e-02   4.24025e-05  -4.75500e-03
   3  p2           1.60211e+01   5.71597e-02  -7.22233e-05   1.06656e-02
   4  p3           1.83757e+03   8.86722e+00  -6.74572e-03   9.54455e-06
   5  p4           7.87329e+02   7.05984e-02   7.15374e-05  -1.11855e-03
   6  p5           1.60234e+01   5.54844e-02  -3.39193e-05  -6.49520e-03
   7  p6           4.49787e+02   1.08471e+00  -4.24501e-04   1.32724e-04
   8  p7          -3.49310e-01   1.42140e-03   4.71660e-07   2.05778e-01
```

9.3 TSpectrum

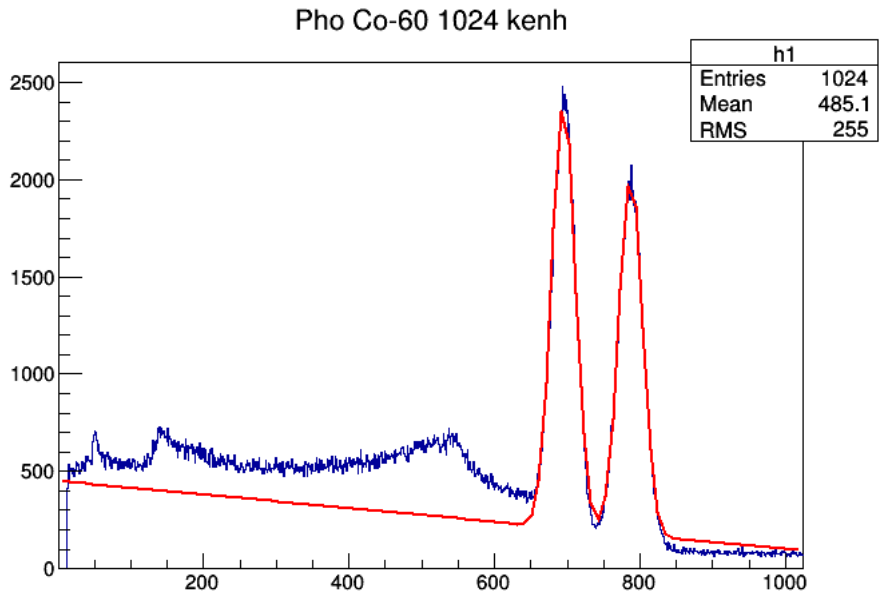
TSpectrum là một lớp được xây dựng nhằm phục vụ cho nhu cầu xử lý phổ, đặc biệt là phổ gamma. Mã nguồn gốc của lớp này được viết bởi Miroslav Morhac bằng ngôn ngữ lập trình C và viết lại bằng C++ bởi Rene Brun. Các bạn có thể tham khảo thêm thông tin về lớp này tại http://root.cern.ch/root/html530/TSpectrum.html.

Khai báo để khai báo một đối tượng thuộc lớp TSpectrum ta gõ

```
  TSpectrum *s = new TSpectrum();
```

Ước lượng phông nền thông qua hàm `Background()`, hàm này sẽ tính toán phổ phông nền thông qua thuật toán SNIP (*Sensitive Nonlinear Iterative Peak*) có công thức như sau

$$v_p(i) = \min\left\{v_{p-1}(i), \frac{v_{p-1}(i+p) + v_{p-1}(i-p)}{2}\right\} \tag{9.3}$$



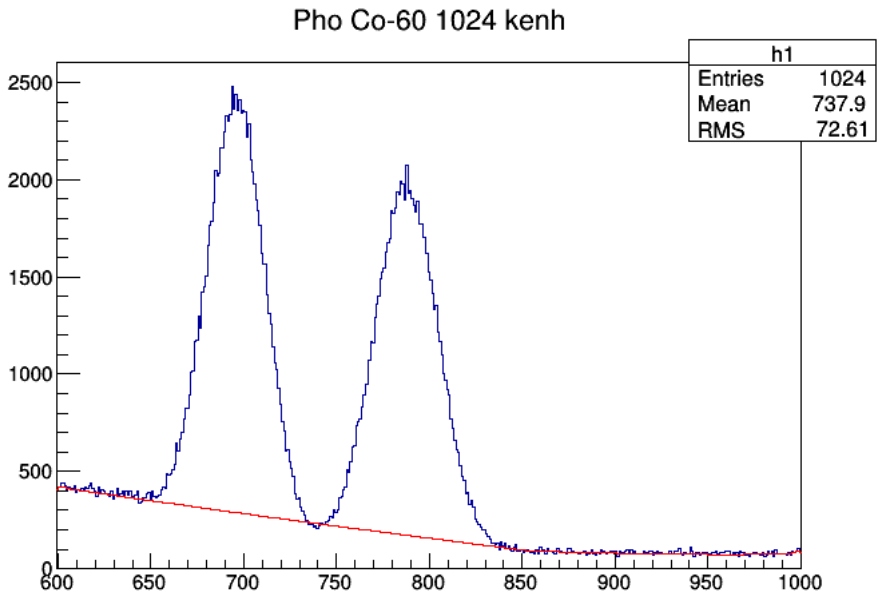
Hình 9.3: Kết quả làm khớp hai đỉnh 1173 và 1332keV

với p là số lần lặp và i là vị trí kênh.
Để ước lượng phông nền dưới chân hai đỉnh phổ ^{60}Co ta làm như sau

```
void pho_gamma()
{
  TH1F *h1 = new TH1F("h1", "Pho Co-60 1024 kenh",1024, 1, 1024);
  ifstream file("Co60.dat");

  int i = 0;
  double sodem;
  while (!file.eof()) {
    file >> sodem;
    h1->SetBinContent(++i,sodem);
  }
  file.close();

  h1->GetXaxis()->SetRange(600,1000);  // khoang gia tri (khoang kenh) can
      uoc luong phong nen
  h1->Draw();  // ve pho

  // Khai bao TSpectrum
  TSpectrum *s = new TSpectrum();

  // Uoc luong phong nen voi so lan lap la 50
  // su dung option "same" de ve phong nen tren cung do thi voi pho
  TH1F *background = s->Background(h1,50,"same");
}
```

Một số option có thể được sử dụng trong hàm `Background()` gồm có:

- `BackIncreasingWindow`: đây là option mặc định cho hàm
- `BackOrderN`: sử dụng phương pháp *clipping filter*, với `N` = 2, 4, 6, 8
- `nosmoothing`: không làm trơn (mặc định là có làm trơn)
- `BackSmoothingN`: độ rộng cửa sổ làm trơn, với `N` = 3, 5, 7, 9, 11, 13, 15



Hình 9.4: Kết quả ước lượng phông nền dưới chân hai đỉnh 1173 và 1332keV

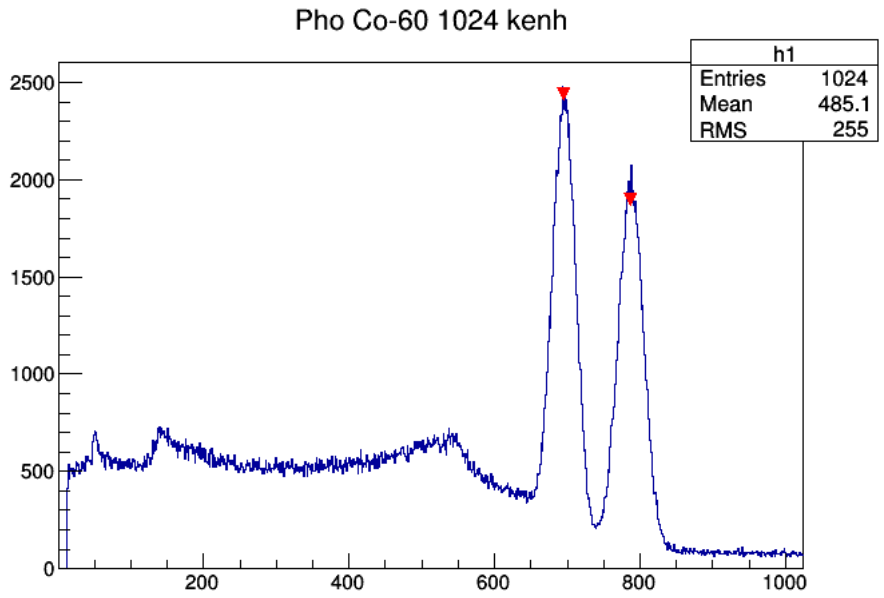
- `Compton`: sử dụng nếu có cạnh Compton (*Compton edge*) trong vùng làm khớp
- `same`: hiển thị kết quả trên cùng đồ thị với phổ

Tìm đỉnh thông qua hàm `Search()`. Phương pháp tìm đỉnh ở đây được dựa trên phương pháp vi phân bậc hai có làm trơn. Cách thức thực hiện như sau

```
void pho_gamma()
{
  TH1F *h1 = new TH1F("h1", "Pho Co-60 1024 kenh",1024, 1, 1024);
  ifstream file("Co60.dat");

  int i = 0;
  double sodem;
  while (!file.eof()) {
    file >> sodem;
    h1->SetBinContent(++i,sodem);
  }
  file.close();

  // Khai bao TSpectrum
  TSpectrum *s = new TSpectrum();

  // Tim dinh voi cac tham so lan luot la:
  // (histogram, sigma, "option", threshold)
  int sodinh = s->Search(h1, 2, "", 0.20);
}
```

Chức năng của các tham số trong hàm `Search()` như sau:

- *histogram*: phổ cần tìm đỉnh.
- *sigma*: độ rộng đỉnh. Trong trường hợp đỉnh tại kênh i có hiệu giữa số đếm đỉnh và số đếm trung bình của hai kênh (i − 3*sigma, i + 3*sigma) nhỏ hơn *threshold*, đỉnh này sẽ bị loại bỏ.
- *threshold*: các đỉnh có số đếm nhỏ hơn tích của *threshold* và số đếm của đỉnh cao nhất sẽ bị loại bỏ (giá trị mặc định là 0.05).



Hình 9.5: Kết quả tìm đỉnh của phổ ^{60}Co, vị trí của các đỉnh được đánh dấu bởi tam giác màu đỏ

- *option*:
  - `noMarkov`: không sử dụng thuật toán chuỗi Markov (mặc định là có sử dụng)
  - `goff`: không lưu dữ liệu và vẽ kết quả tìm đỉnh
  - `nodraw`: không vẽ phổ với kết quả tìm đỉnh

Để thu được thông tin về số lượng đỉnh tìm được và vị trí các đỉnh ta có thể truy xuất các thành phần `fNPeaks` và `fPositionX` thông qua các hàm

```
int sodinh = s->GetNPeaks();   // so luong dinh tim duoc
float *vitri = s->GetPositionX();  // mang (array) chua vi tri cac dinh
```

TÀI LIỆU THAM KHẢO

[1] René Brun, Fons Rademakers, *ROOT – An Object Oriented Data Analysis Framework*, Nuclear Instruments and Methods in Physics Research A389, p.81-86, 1997.

[2] I. Antcheva et al, *ROOT − A C++ framework for petabyte data storage, statistical analysis and visualization*, Computer Physics Communications, Vol.180, p.2499-2512, 2009.

[3] René Brun et al, *ROOT Overview*, CERN, 1997.

[4] Danilo Piparo, Günter Quast, *A ROOT Guide for Student "Diving Into ROOT"*, Karlsruhe Institute of Technology, 2011.

[5] W.G. Seligman, *Basic Data Analysis Using ROOT*, 2013.
`http://www.nevis.columbia.edu/~seligman/root-class/RootClass2013.pdf`

[6] `http://root.cern.ch/drupal/content/users-guide`

[7] `http://root.cern.ch/drupal/content/reference-guide`

[8] `http://root.cern.ch/root/html/TF1.html`

[9] `http://root.cern.ch/root/html530/TSpectrum.html`

[10] `http://atlas.fis.utfsm.cl/atlas/tutorial.root.utfsm.html`

[11] `http://en.wikipedia.org/wiki/ROOT`