Root 2009

www.if.usp.br/suaide

Alexandre Suaide

Aula 3

Programa

- **Aula 1**
  - Introdução ao c++ e ROOT
  - Conceito de classe e objeto
  - Básico de gráficos e funções no ROOT
- **Aula 2**
  - Mais gráficos e funções
  - Histogramas de 1 e 2D
  - Ajustes de funções, legendas, etc.
  - Escrevendo programas simples: Monte Carlo e simulações
- **Aula 3**
  - Referências e ponteiros
  - Nomes e memória
  - Programação mais complexa: mais Monte Carlo
- **Aula 4**
  - I/O no ROOT
  - Mais programação no ROOT
  - Compilando com o ROOT

Ponteiros/referências e outros monstros

- Quando criamos uma variável (ou qualquer outra coisa) esta ocupa um lugar na memória do computador
  - float a = 10;
- Em alguma posição desta memória temos armazenado o valor a e em algum outro lugar da memória temos armazenado que existe a variável a e em que lugar o seu conteúdo está armazenado.
- Como acessar estas informações?
- Como acessar esta memória e modificá-la?
- Como atuar sobre a variável?

Referências (&)

- Referências são apelidos para as variáveis/objetos criados durante um programa
- Exemplos

```
float  a = 10;
float& b = a;
cout <<"a = "<<a<<"  b = "<<b<<endl;
a = 10   b = 10
a = 20;
cout <<"a = "<<a<<"  b = "<<b<<endl;
a = 20   b = 20
b = b-5;
cout <<"a = "<<a<<"  b = "<<b<<endl;
a = 15   b = 15
```

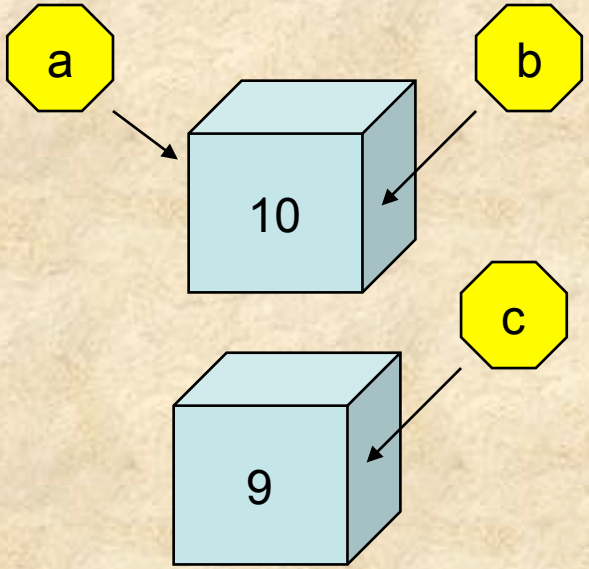
Referências (&)

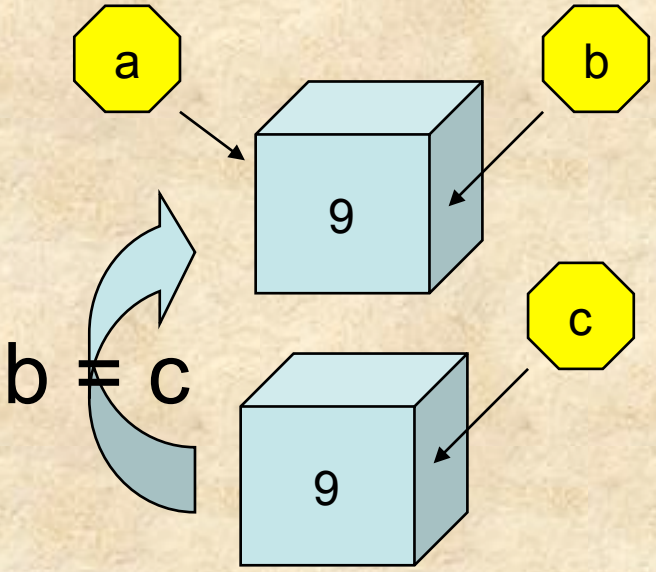
- Referências não podem ser recriadas
- Exemplos

```
float  a = 10;
float& b = a;
float  c = 9;
b = c;
cout <<"a = "<<a<<"  b = "<<b<<"  c = "<<c<<endl;
a = 9   b = 9   c = 9
a = 10;
cout <<"a = "<<a<<"  b = "<<b<<"  c = "<<c<<endl;
a = 10   b = 10   c = 9
```

Referências (&)

- No caso anterior, a e b correspondem à mesma localização de memória, enquanto c corresponde a outra localização
- Quando fazemos b = c, não estamos mudando a referência em b, estamos copiando o valor de c para a posição de memória que b se refere.
- Trabalhar, no seu programa, com a ou b é a mesma coisa





Quando usar referências?

- Quando voce quiser ☺
- Uso mais comum é como parâmetros de funções
- Quando definimos uma função, por exemplo

```
float funcao(float a, int b)
```

  - Os parâmetros a e b são tratados, pela função, como variáveis locais
  - Ex:

```
void func(float a)
{
  a = a*2;
  cout << a << endl;
  return;
}
float var = 1;
func(var);
2
cout << var << endl;
1
```

Note que o valor de x é alterado localmente. A variável original, var, neste caso, permanece com o seu valor original

Quando usar referências?

- Porém, caso passemos para a função uma referência para a variável, podemos manipular o seu conteúdo sem problema

```
float funcao(float& a, int b)
```

  - Neste caso, posso manipular o valor da variável a enquanto b é tratado localmente
  - Ex:

```
void func(float& a)
{
  a = a*2;
  cout << a << endl;
  return;
}
float var = 1;
func(var);
2
cout << var << endl;
2
func(3);
Error: could not convert '3' to float&
```

Note que o valor de var foi alterado pois eu passei uma referência desta variável para a função

Apesar do ROOT aceitar esta sintaxe no prompt de comando, o compilador c++ + retornará uma mensagem de erro

Ponteiros (*)

- Ponteiros são variáveis que contém endereços de memória para alguma coisa
- Ao contrário de referências estes podem mudar de localização
- Operadores
  - * → define um ponteiro
    - Ex: float* a;
      - 'a' contém um endereço de memória para que contém um número float
  - * → também pode ser utilizado para acessar o conteúdo de uma posição de memória
  - & → utilizado para obter o endereço de memória de uma variável/objeto

Ponteiros (*)

- **Exemplos**

```
float x = 10;
float *p = &x;
cout << x <<endl;
10
cout << p <<endl;
(float*)0x45daa58
cout << *p <<endl;
10
*p = 20;
cout << x <<endl;
20
cout << p <<endl;
(float*)0x45daa58
cout << *p <<endl;
20
```

Cria um ponteiro do tipo float cujo valor corresponde ao endereço de memória onde esta armazenada a variável x

O valor de x

O valor de p. Este corresponde ao endereço de memória onde a variável x está armazenada

O conteúdo armazenado nesta posição de memória

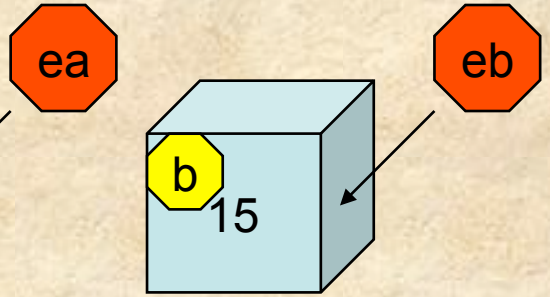
Mudando o conteúdo da memória

Isto reflete automaticamente na variável x

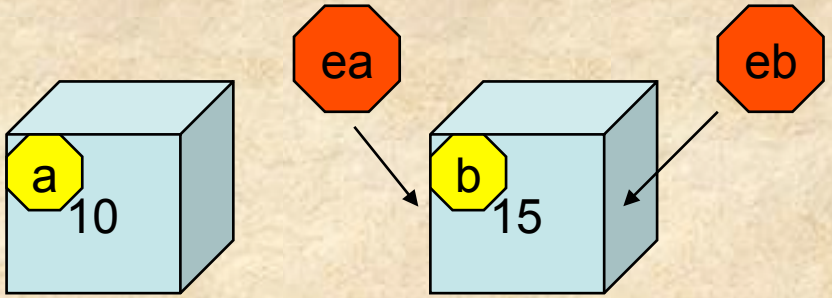
Operações com ponteiros

```
float a = 10;
float b = 15;
float *ea = &a;
float *eb = &b;
cout <<a<<" "<<*ea<<"          <<endl;
10 10 0xb26c3e8
cout <<b<<" "<<          <<" "<<eb<<endl;
15 15 0xb26c408
ea = eb;
cout <<a<<" "<<*ea<<" "<<ea<<endl;
10 15 0xb26c408
eb = eb + 1;
cout <<b<<" "<<          b<
15 1.97565e-36 0x
```

Faz o ponteiro ea ter o mesmo valor do ponteiro eb. Na prática, ea e eb apontam para a mesma posição de memória

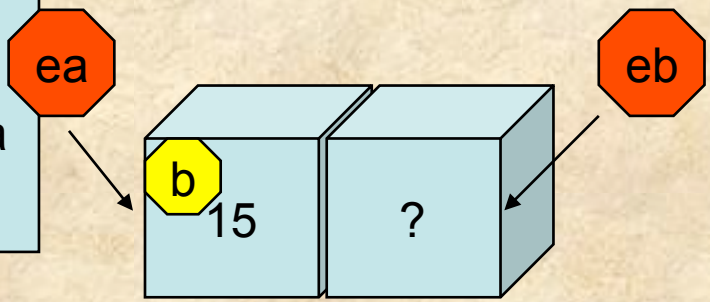


ea = eb



Incrementa o ponteiro eb de uma unidade. Na prática, move o ponteiro para a próxima posição de memória após o seu valor inicial

eb = eb+1



Arrays são, na verdade, ponteiros

- O c++ trata vetores/arrays como sendo ponteiros

```
float a[10] = {1,2,3,4,5,6,7,8,9,10};
cout << a <<endl;
0x95bf5b8
cout <<a[0]<<" "<<*a<<endl;
1  1
cout <<a[5]<<" "<<*(a+5)<<endl;
6  6
```

Isto pode ser útil caso queiramos passar um array como argumento de função

Passando arrays como argumentos de funções

- Caso queiramos passar um array para uma função em c++ utilizamos o seu ponteiro

```
void cinematica(float* M)
{
  cout <<"A_feixe = "<<M[0]<<"   "
       <<"A_alvo = "<<M[1]<<"   "
       <<"A_1 = "<<M[2]<<"   "
       <<"A_2 = "<<M[3]<<endl;
  return;
}
float A[4] = {16,10,20,6};
cinematica(A);
```

```
A_feixe = 16  A_alvo = 10  A_1 = 20  A_2 = 6
```

Referências e ponteiros

- Referências e ponteiros são definições em c++ muito parecidas porém diferentes
- Ambas servem de atalho para objetos/variáveis definidas na memória
- Ambas permitem manipulações dos dados armazenados na memória

- Qual a diferença entre elas?

Ponteiros e referencias

- Referências
  - Define-se com &
  - Atalho para uma variável já definida
  - A referência está sempre atrelada a variável inicialmente associada
  - Mesmo uso de
  - Bom para passar variáveis comuns para funções

- Ponteiros
  - Define-se com *
  - Representa uma posição de memória, tendo ou não informação válida
  - Pode mudar livremente de lugar
  - Precisa-se utiliar operadores * e &
  - Bom para passar arrays para funções
  - Ponteiros são ferramenta sindispensáveis na programação O.O.

Exemplo: cálculo de resíduos

- Porque eu calculo os resíduos em um vetor YM e EY e não uso o próprio y e ey?
  - Porque como os vetores x,y,... São ponteiros uma alteração neles mudaria também o gráfico original

```
TGraphErrors* residuos(TGraphErrors *g,
                       TF1 *fit)
{
 double* x = g->GetX();
 double* y = g->GetY();
 double* ex = g->GetEX();
 double* ey = g->GetEY();

 int N = g->GetN();

 double YM[1000],EY[1000];

 for(int i = 0;i<N;i++)
 {
   double teoria = fit->Eval(x[i]);
   YM[i] = (y[i]-teoria)/ey[i];
   EY[i] = 1;
 }
 return new TGraphErrors(N,x,YM,ex,EY);
}
```

Criando e destruindo objetos (da aula 1)

- **Criando objetos no stack**

```
void exemplo_obj_1()
 {
   TH1F h("hist","histograma",100,0,10);
   h.SetLineColor(1);
   h.Draw();
 }
```

  - O objeto `h` deixa de existir quando a função termina

- **Criando objetos no heap (`new` e `delete`)**

```
void exemplo_obj_2()
 {
   TH1F* h = new TH1F("hist","histograma",100,0,10);
   h->SetLineColor(1);
   h->Draw();
 }
```

  - Objetos no heap são acessados com ponteiros
  - O objeto `h` só deixa de existir com o `delete h;`

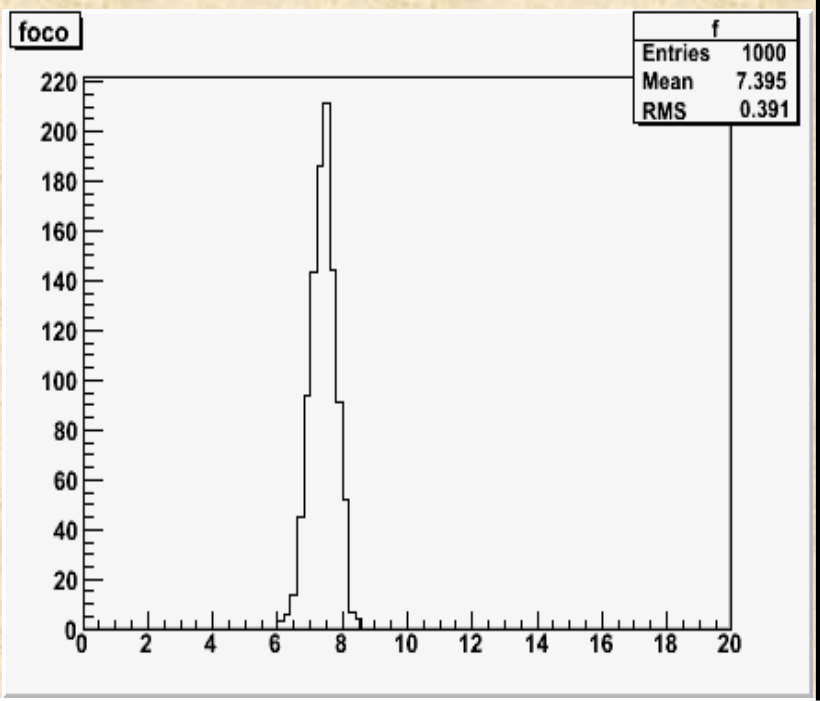
Exemplo: Propagação de incertezas com Monte Carlo

- **No ROOT digite:**

root [0] .L MC_lente.C

root [1] MC_lente(25.3, 0.2, 10.5, 0.8, 1000)
(float)3.910020885171509e-01

root [2] hist->Draw()




```
TH1F *hist;
float MC_lente(float o, float so,
               float i, float si, int N)
{
  hist = new TH1F("f","foco",100,0,20);
  TRandom *r = new TRandom();
  for(int k = 0; k<N; k++)
  {
    float I = r->Gaus(i,si);
    float O = r->Gaus(o,so);
    float F = I*O/(I+O);
    hist->Fill(F);
  }
  float RMS =hist->GetRMS();
  return RMS;
}
```

Exemplo: Propagação de incertezas com Monte Carlo

- Veja a seguinte situação:

```
.L MC_lente.C

float o[] = {10, 20, 30, 40, 50};
float so[] = {0.1, 0.1, 0.1, 0.1};
float i[] = {23, 11, 9.1, 8.5, 8.2};
float si[] = {0.5, 0.7, 0.9, 1.0, 1.1};


for(int j = 1; j<5; j++) {
  float f = i[j]*o[j]/(i[j]+o[j]);
  float e = MC_lente(o[j],so[j],i[j],si[j],100);
  cout <<f<<"  "<<e<<endl;
}
```

O que acontece se eu chamar várias vezes a mesma função?

Exemplo: Propagação de incertezas com Monte Carlo

- **Modificação para levar em conta vazamentos de memória:**

A diferença é que, agora, o histograma velho chamado "f" é apagado da memória toda vez que a função é chamada e cria-se um novo. Isto evita ir preenchendo a memória com vários histogramas

Também apagamos o objeto TRandom antes de sair da função.

```
TH1F *hist = 0;
float MC_lente(float o, float so,
               float i, float si, int N)
{
  if(hist) delete hist;
  hist = new TH1F("f","foco",100,0,20);
  TRandom *r = new TRandom();
  for(int k = 0; k<N; k++)
  {
    float I = r->Gaus(i,si);
    float O = r->Gaus(o,so);
    float F = I*O/(I+O);
    hist->Fill(F);
  }
  float RMS =hist->GetRMS();
  delete r;
  return RMS;
}
```

Exemplo: Propagação de incertezas com Monte Carlo

- **Modificação para levar em conta vazamentos de memória:**

```
TH1F *hist = 0;
float MC_lente(float o, float so,
               float i, float si, int N)
{
  if(hist) delete hist;
  hist = new TH1F("f","foco",100,0,20);
  TRandom r();
  for(int k = 0; k<N; k++)
  {
    float I = r.Gaus(i,si);
    float O = r.Gaus(o,so);
    float F = I*O/(I+O);
    hist->Fill(F);
  }
  float RMS =hist->GetRMS();
  return RMS;
}
```

No caso do TRandom, pode-se criar um objeto no stack (local). Neste caso o próprio computador se encarrega de apagar quando terminar a função (mas note o uso diferente: . ao invés de ->)

Podemos fazer isto para o histograma?

Resumindo

- Ponteiros e referências são amplamente utilizados em várias linguágens de programação, principalmente em c++
  - O entendimento de como eles funcionam permite ter controle sobre o que estamos fazendo
- Muitas funções de vários objetos do ROOT usam ponteiros e referências.
  - Várias retornam ponteiros para outros objetos ou vetores