Ponteiros de Variáveis

- C++ permite o armazenamento e a manipulação de valores de endereços de memória.
- Para cada tipo existente, há um tipo ponteiro capaz de armazenar endereços de memória em que existem valores do tipo correspondente armazenados.
- Quando escrevemos

```
int a = 0;
```

a declaração da variável **a**, inteira, significa que automaticamente é reservado um espaço de memória para armazenar valores inteiros (4 bytes).

Ponteiros de Variáveis

- É possível também declarar variáveis que, em vez de armazenar valores inteiros, armazenam valores de endereços de memória onde existem valores inteiros.
- C++ não reserva uma palavra especial para a declaração de ponteiros, mas sim o operador *, ou seja:

```
int *p;
```

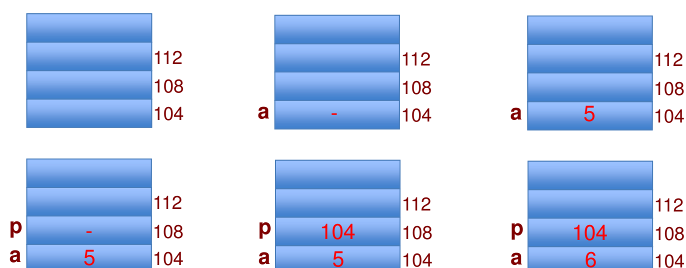
declara uma variável de nome **p** que pode armazenar endereços de memória em que existe um inteiro armazenado.

Ponteiros de Variáveis

- Para atribuir e acessar endereços de memória, a linguagem oferece dois operadores unários:
- O operador &, ou endereço de, aplicado a variáveis resulta no endereço da posição de memória reservada para a variável.
- O operador unário *, ou conteudo de, aplicado a variáveis do tipo ponteiro, acessa o conteudo do endereço de memória armazenado pela variável ponteiro.

Ponteiros de Variáveis

```
int a;  // aloca-se memória para armazenar valores inteiros.
a = 5;  // a recebe o valor 5.
int *p; // aloca-se memória para armazenar endereços de memória.
p = &a; // p recebe o endereço de a, ou p aponta para a.
*p = 6; // conteudo de p recebe o valor 6.
```



Ponteiros de Variáveis

```
#include <iostream>

using namespace std;

int main( void )
{

  int a;
  int *p;

  p = &a;

  *p = 2;

  cout << a << endl;

  return 1;
}
```

```
#include <iostream>

using namespace std;

int main( void )
{

  int a;
  int *p;

  a = 2;

  *p = 3;

  b = a + (*p);

  cout << b << endl;

  return 1;
}
```

Ponteiros de Variáveis

- Da mesma forma que declaramos ponteiros de variáveis inteiras, é também possível declarar ponteiros para os demais tipos básicos oferecidos pela linguagem:

  float *m;

  char *s;

  double *d;

Vetores e Alocação Dinâmica

```
// Cálculo da média de n numeros reais.

#include <iostream>

using namespace std;

int main( void )
{

  int   n;
  float med = 0.0f;

  cin >> n; // leitura do número de valores.

  // leitura do conjunto de valores e cálculo do somatório:
  for( int i=0; i<n; i++ )
  {
    float v;        // variável para armazenar valor lido;
    cin >> v;       // lê cada valor;
    med = med + v; // acumula soma dos valores;
  }

  // calculo da média:
  med = med / n;

  // exibição do resultado:
  cout << "Valor da Média = " << med << endl;

  return 0;
}
```

Vetores e Alocação Dinâmica

- A forma mais simples de se estruturar um conjunto de dados é por meio de vetores:

  int v[10];

- Essa declaração diz que **v** é um vetor de inteiros dimensionado com 10 elementos, ou seja, é reservado um espaço de memória contínuo para armazenar 10 valores inteiros.
- O acesso a cada elemento do vetor é feito através de indexação da variável v.
- Em C++ a indexação de um vetor varia de *0 a n-1*, onde n é a dimensão do vetor.

Vetores e Alocação Dinâmica

- Para a declaração do vetor v:

  v[0] acessa o primeiro elemento de **v**;

  v[1] acessa o segundo elemento de **v**;

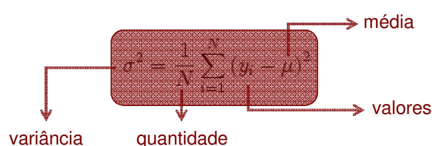
  …

  v[9] acessa o último elemento de v;

  v[10] ERRADO ! **Invasão de memória !!**

Vetores e Alocação Dinâmica

- Para calcular-se a média aritmética de números lidos do teclado não é necessário armazenar esses números na memória, mas se por algum motivo precisarmos, mais a frente, novamente manipular esses valores, vamos precisar armazená-los.
- Por exemplo para o caso de cálculo da variância desses números:



$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(y_i - \mu)^2$$

Vetores e Alocação Dinâmica

```
// Cálculo da média e da variância de até 10 numeros reais.

#include <iostream>

using namespace std;

int main( void )
{
  // leitura da quantidade de números.
  int n;
  cout << "Entre com a quantidade de valores (máximo 10)\t";
  cin >> n;

  // leitura dos valores.
  float  v[10];

  cout << "Entre com os valores:" << endl;
  for( int i=0; i<n; i++ )
    cin >> v[i];

  // cálculo da média.
  float med = 0.0f;

  for( int i=0; i<n; i++ )
    med = med + v[i];

  med = med / n;

  // cálculo da variância.
  float var = 0.0f;

  for( int i=0; i<n; i++ )
    var = var + ((v[i]-med)*(v[i]-med));

  var = var / n;

  // exibição do resultado:
  cout << "Média     = " << med << endl;
  cout << "Variância = " << var << endl;

  return 0;
}
```

Vetores e Alocação Dinâmica

- No exemplo anterior o vetor ***v*** foi definido com um tamanho fixo de 10 campos. Isto significa que não poderíamos calcular a média e a variância de mais de 10 valores lidos.
- Para garantir que o programa funcione “sempre”, seria necessário dimensionar o vetor **v** com um tamanho absurdamente alto, o que representa um disperdicio de memória.
- C++ oferece meios de requisitar espaços de memória em tempo de execução, ou seja, fazer alocação dinâmica.

Alocação Dinâmica

- Para se alocar dinâmicamente um vetor que armazene, por exemplo, 10 inteiros, são necessários os seguintes comandos:

  *int *v;*

  v = ***new*** *int[10];*

- Após isso o vetor pode ser manipulado da mesma forma que foi apresentado anteriormente:

  v[0] acessa o primeiro campo, etc ...

- Ao final, o programador deverá liberar a memória alocada dinamicamente através do seguinte comando:

  delete *[] v;*

- Os operadores new e delete serão revisitados posteriormente.

Alocação Dinâmica

- O programa que calcula média e variância pode ser implementado agora com alocação dinâmica:

```
// Cálculo da média e da variância de n numeros reais.

#include <iostream>

using namespace std;

int main( void )
{
  // leitura da quantidade de números.
  int n;
  cout << "Entre com a quantidade de valores (sem limite)\t";
  cin >> n;

  // Alocação Dinâmica baseada no número digitado.
  float *v = new float[n];

  // leitura dos valores.
  cout << "Entre com os valores:" << endl;
  for( int i=0; i<n; i++ )
    cin >> v[i];

  // cálculo da média.
  float med = 0.0f;

  for( int i=0; i<n; i++ )
    med = med + v[i];

  med = med / n;

  // cálculo da variância.
  float var = 0.0f;

  for( int i=0; i<n; i++ )
    var = var + ((v[i]-med)*(v[i]-med));

  var = var / n;

  // exibição do resultado:
  cout << "Média     = " << med << endl;
  cout << "Variância = " << var << endl;

  delete [] v;

  return 0;
}
```

Funções

- Para a construção de programas estruturados, é sempre preferível dividir as grandes tarefas de computação em tarefas menores.
- Para a linguagem de programação C faz-se o uso de funções para estruturar um código de programação. As principais vantagens são:
  - Facilita a codificação
  - Reuso do código
- Em C++ não se utiliza funções da forma que será apresentado aqui, mas boa parte dos conceitos aqui envolvidos serão aproveitados.

Funções

- A forma geral para se definir uma função é:

```
tipo_retornado nome_da_função (lista de parâmetros)
{
 corpo da função
}
```

Funções

- Para o exemplo do cálculo do fatorial de um número, poderíamos re-escreve-lo da seguinte forma:

```
// Fatorial versão 2
#include <iostream>
using namespace std;
int main( void )
{
  cout << "Digite um número inteiro não negativo: ";
  int n; cin >> n;
  // calcula fatorial
  int f = 1;
  for( int i=1; i <= n; i++ )
    f *= i;
  cout << "Fatorial = " << f << endl;
  return 0;
}
```

```
// Fatorial versão 3 (função)
#include <iostream>
using namespace std;
void fat( int n );
int main( void )
{
  cout << "Digite um número inteiro não negativo: ";
  int n; cin >> n;
  // calcula fatorial
  fat( n );
  return 0;
}
void fat( int n )
{
  int f = 1;
  for( int i=1; i<=n; i++ )
    f *= i;
  cout << "Fatorial = " << f << endl;
}
```

Funções

- Nesse exemplo a função *fat* recebe como parâmetro o número cujo fatorial tem que ser impresso.
- Os parâmetros devem ser listados com os respactivos tipos entre os parênteses que seguem ao nome da função.
- Quando a função não tem parâmetros, colocamos a palavra reservada *void* entre os parênteses.
- *main* também é uma função. Sua única particularidade consiste em ser a função automaticamente executada após o programa ser carregado.
- Como a função *main* não está recebendo parâmetros, utiliza-se a palavra *void* na lista de parâmetros.

Funções

- Além de receber parâmetros, uma função pode ter um valor de retorno associado.
- Na versão 3 do cálculo do fatorial, como a função fat não retorna parâmetros, por isso a função foi definida com a palavra *void* antes do seu nome:

```
void fat( int n )
{
...
}
```

- A função *main* deve obrigatoriamente ter um valor inteiro como retorno. Esse valor pode ser usado pelo sistema operacional para testar a execução do programa.
- No corpo da função é necessário utilizar a palavra reservada ***return*** para encerrar a execução da função retornando o valor da expressão que vem imediatamente a seguir.

Funções

- C/C++ exige que se coloque o protótipo da função antes dela ser chamada. O protótipo consiste na repetição da sua linha de sua definição seguida do caractere (**;**):

```
void fat (int n);
```

- O protótipo da função é necessário para que o compilador verifique os tipos dos parâmetros na chamada da função.
- O que aconteceria caso a função a chamada da função fat fosse feita da forma abaixo ?

```
fat(4.5);        // warning
fat(“abcd”);     // error
```

Funções

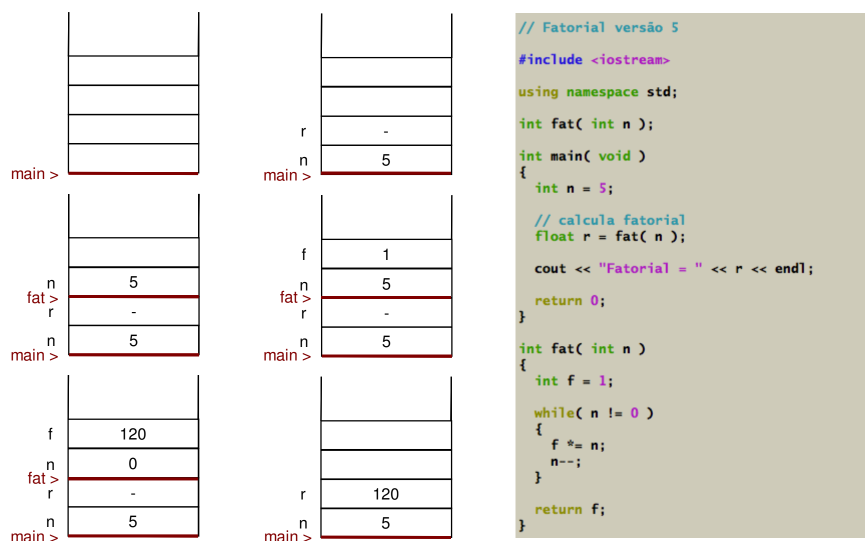
```
// Fatorial versão 3 (função)
#include <iostream>
using namespace std;
void fat( int n );
int main( void )
{
  cout << "Digite um número inteiro não negativo: ";
  int n; cin >> n;
  // calcula fatorial
  fat( n );
  return 0;
}
void fat( int n )
{
  int f = 1;
  for( int i=1; i<=n; i++ )
    f *= i;
  cout << "Fatorial = " << f << endl;
}
```

```
// Fatorial versão 4 (função com valor de retorno)
#include <iostream>
using namespace std;
int fat( int n );
int main( void )
{
  cout << "Digite um número inteiro não negativo: ";
  int n; cin >> n;
  // calcula fatorial
  float r = fat( n );
  cout << "Fatorial = " << r << endl;
  return 0;
}
int fat( int n )
{
  int f = 1;
  for( int i=1; i<=n; i++ )
    f *= i;
  return f;
}
```

Pilha de Execução

- A função *fat* foi chamada da função *main* no exemplo do programa que calcula o fatorial, mas como funciona a comunicação entre a função que chama e a que é chamada?
- As funções são independentes entre si.
- As variáveis locais definidas dentro do corpo de uma função, incluidos os parâmetros, não existem fora dela.
- Cada vez que a função é executada, as variáveis locais são criadas e quando a execução termina, as variáveis deixam de existir.
- A transferência dos dados é feita com o uso de parâmetros e com o valor de retorno da função chamada, através do comando *return*.

Pilha de Execução



Funções: Passagem de Parâmetros

- Nem sempre é suficiente apenas um valor de retorno para uma determinada função.
- No caso da função que calcula o fatorial, um valor de retorno foi suficiente, mas e se fosse necessário retornar mais de um valor? Como fazer?
- No caso de uma função, por exemplo, que receba dois números e retorne a soma e o produto desses valores, o que fazer?
- Poderíamos tentar com o código a seguir:

Funções: Passagem de Parâmetros

```
#include <iostream>

using namespace std;

void somaprod( int a, int b, int sum, int prod );

int main( void )
{
  int s, p;

  somaprod( 3, 5, s, p );

  cout << "Soma    = " << s << endl;
  cout << "Produto = " << p << endl;

  return 0;
}

void somaprod( int a, int b, int sum, int prod )
{
  sum  = a + b;
  prod = a * b;
}
```

- Como seria a pilha de execução desse programa e o que ele vai imprimir?

Passando Ponteiros para Funções

- Como na passagem de parâmetros o que é passado é uma cópia para os parâmetros da função, a solução anterior não vai funcionar.
- A solução para esse problema é, ao invés de passar valores inteiros para os dois últimos, argumentos da função, passar ponteiros para inteiros. O protótipo da função somaprod passaria a ser então o seguinte:

void somaprod(int a, int b, int *sum, int *prod);

Passando Ponteiros para Funções

```
#include <iostream>

using namespace std;

void somaprod( int a, int b, int *sum, int *prod );


int main( void )
{
  int s, p;

  somaprod( 3, 5, &s, &p );

  cout << "Soma    = " << s << endl;
  cout << "Produto = " << p << endl;

  return 0;
}

void somaprod( int a, int b, int *sum, int *prod )
{
  *sum  = a + b;
  *prod = a * b;
}
```

Passando Ponteiros para Funções

- A função *somaprod* não retorna explicitamente nenhum valor (através do comando ***return***).
- A função recebe o endereço de memória de duas variáveis e armazena a soma e o produto no endereço das duas variáveis passadas.
- A seguir mostra-se a execução da pilha para esse programa.
- É possível, na execução da pilha, observar que na realidade continua-se fazendo cópia para os parâmetros *sum* e *prod*. A diferença é que agora é copiado o endereço de memória das variáveis existentes no escopo da função ***main***.

Passando Ponteiros para Funções

Variável	Valor	Endereço
		112
p	-	108
main > s	-	104

Variável	Valor	Endereço
prod	108	124
sum	104	120
b	5	116
somaprod > a	3	112
p	-	108
main > s	-	104

Variável	Valor	Endereço
prod	108	124
sum	104	120
b	5	116
somaprod > a	3	112
p	15	108
main > s	8	104

Variável	Valor	Endereço
		112
p	15	108
main > s	8	104

```
#include <iostream>

using namespace std;

void somaprod( int a, int b, int *sum, int *prod );

int main( void )
{
  int s, p;

  somaprod( 3, 5, &s, &p );

  cout << "Soma    = " << s << endl;
  cout << "Produto = " << p << endl;

  return 0;
}

void somaprod( int a, int b, int *sum, int *prod )
{
  *sum  = a + b;
  *prod = a * b;
}
```

Passando Ponteiros para Funções

- Para o exemplo abaixo, como ficaria a pilha de execução?

```
// Exemplo: Função Troca

#include <iostream>

using namespace std;

void troca( int *px, int *py );

int main( void )
{
  int a = 5, b = 7;

  // chama função troca passando os endereços das variáveis.
  troca( &a, &b );

  cout << "a = " << a << endl;
  cout << "b = " << b << endl;

  return 0;
}

void troca( int *px, int *py )
{
  int temp;

  temp = *px;
  *px  = *py;
  *py  = temp;
}
```

Referências

- É uma forma alternativa para criar variáveis de tipos básicos da linguagem, ou mesmo tipos agregados.
- Esse recurso só existe em C++ (a liguagem C não faz uso de referências).
- O principal uso de referências é para passagem de parâmetros e retorno de funções.
- A notação ***X&*** representa referência para ***X***, onde ***X*** é um tipo (básico ou não).

Referências

- Internamente, uma referência é um ponteiro;
- Uma referência tem que ser obrigatoriamente inicializada:

```
int i = 1;
int& r = i;  // r e i referem-se ao mesmo espaço
int x = r;   // x = 1
r = 2;       // i = 2;
```

Referências como Variáveis Locais

```
{
   int  a;          // ok, variável normal
   int& b = a;   // ok, b é uma referencia para a
   int& c;         // erro! não foi inicializada
   int& d = 12; // erro! inicialização inválida
}
```

```
{
   int  a = 10;
   int& b = a;
   printf("a=%d, b=%d\n", a, b); // produz a=10, b=10
   a = 3;
   printf("a=%d, b=%d\n", a, b); // produz a=3, b=3
   b = 7;
   printf("a=%d, b=%d\n", a, b); // produz a=7, b=7
}
```

Referências como Tipos de Parâmetros

```
void f(int a1, int &a2, int *a3)
{
   a1 = 1;  // altera cópia local
   a2 = 2;  // altera a variável passada (b2 de main)
  *a3 = 3;  // altera o conteúdo do endereço de b3
}
```

```
void main()
{
   int b1 = 10, b2 = 20, b3 = 30;
   f(b1, b2, &b3);
   printf("b1=%d, b2=%d, b3=%d\n", b1, b2, b3);
   // imprime b1=10, b2=2, b3=3
}
```

Referências

```
// Exemplo: Função Troca
#include <iostream>
using namespace std;
void troca( int *px, int *py );
int main( void )
{
  int a = 5, b = 7;
  // chama função troca passando os endereços das variáveis.
  troca( &a, &b );
  cout << "a = " << a << endl;
  cout << "b = " << b << endl;
  return 0;
}
void troca( int *px, int *py )
{
  int temp;
  temp = *px;
  *px  = *py;
  *py  = temp;
}
```

```
// Exemplo: Função Troca com Referência
#include <iostream>
using namespace std;
void troca( int &rx, int &ry );
int main( void )
{
  int a = 5, b = 7;
  // chama função troca passando os endereços das variáveis.
  troca( a, b );
  cout << "a = " << a << endl;
  cout << "b = " << b << endl;
  return 0;
}
void troca( int &rx, int &ry )
{
  int temp;
  temp = rx;
  rx   = ry;
  ry   = temp;
}
```

Passagem de Vetores para Funções

- Passar um vetor para uma função consiste em passar o endereço do primeiro campo do vetor.
- Se passamos um valor de endereço, então o parâmetro da função que vai receber esse valor, deve ser um ponteiro.
- Os elementos do vetor não são copiados, apenas o que é copiado é o endereço do primeiro campo do vetor.

Passando de Vetores para Funções

```cpp
// Cálculo da média e da variância de n numeros reais.

#include <iostream>

using namespace std;

float media     ( int n, float *v );
float variancia ( int n, float m, float *v );

int main( void )
{
  // leitura da quantidade de números.
  int n;
  cout << "Entre com a quantidade de valores (sem limite)\t";
  cin >> n;

  // Alocação Dinâmica baseada no número digitado.
  float *v = new float[n];

  // leitura dos valores.
  cout << "Entre com os valores:" << endl;
  for( int i=0; i<n; i++ )
    cin >> v[i];

  float med = media( n, v );
  float var = variancia( n, med, v );

  // exibição do resultado:
  cout << "Média     = " << med << endl;
  cout << "Variância = " << var << endl;

  delete [] v;

  return 0;
}
```

```cpp
float media( int n, float *v )
{
  float sum = 0.0f;

  for( int i=0; i<n; i++ )
    sum = sum + v[i];

  return sum / n;
}

float variancia ( int n, float m, float *v )
{
  float var = 0.0f;

  for( int i=0; i<n; i++ )
    var = var + ((v[i]-m)*(v[i]-m));

  var = var / n;

  return var;
}
```

•

Passando de Vetores para Funções

- Como passamos o endereço do primeiro campo do vetor, seria possível alterar o valor do vetor passado;
- No exemplo ao lado o que o programa imprime ao ser executado é: 2   4   6;

```cpp
// Incrementa elementos de um vetor.

#include <iostream>

using namespace std;

void incr_vetor( int n, int *v );

int main( void )
{
  int v[] = {1, 3, 5};

  incr_vetor( 3, v );

  cout << v[0] << "\t";
  cout << v[1] << "\t";
  cout << v[2] << endl;

  return 0;
}

void incr_vetor( int n, int *v )
{
  for( int i=0; i<n; i++ )
    v[i]++;
}
```

Funções: Retorno Explícito de Ponteiros

- Deve-se tomar cuidado com o uso de vetores locais de uma determinada função:

```
float* produto_vetorial( float *u, float *v )
{
  float p[3];

  p[0] = u[1]*v[2] - v[1]*u[2];
  p[1] = u[2]*v[0] - v[2]*u[0];
  p[2] = u[0]*v[1] - v[0]*u[1];

  return p; // ERRO ! Endereço de área local !
}
```

```
float* produto_vetorial( float *u, float *v )
{
  float *p = new float[3];

  p[0] = u[1]*v[2] - v[1]*u[2];
  p[1] = u[2]*v[0] - v[2]*u[0];
  p[2] = u[0]*v[1] - v[0]*u[1];

  return p;
}
```

- Seria possível alguma outra solução?

Variáveis Globais

- Também é possível através de variáveis globais fazer a comunicação entre funções.
- Se uma variável é declarada for a do corpo das funções, ela é dita global.
- Uma variável global é visível a todas as funções subsequentes.
- As variáveis globais não são armazenadas na pilha de execução, portanto, não deixam de existir quando a execução da função termina.
- Elas existem enquanto o programa está sendo executado.
- Se uma variável global é visível em duas funções, ambas podem acessar e/ou alterar o valor da variável diretamente.

Variáveis Globais

```
#include <iostream>

using namespace std;

// Variáveis Globais
int Sum;
int Prod;

void somaprod( int a, int b );

int main( void )
{
  int s, p;

  somaprod( 3, 5 );

  cout << "Soma    = " << Sum  << endl;
  cout << "Produto = " << Prod << endl;

  return 0;
}

void somaprod( int a, int b )
{
  Sum  = a + b;
  Prod = a * b;
}
```

Não é boa prática de programação, mesmo em C Puro !!!

Uso da Memória

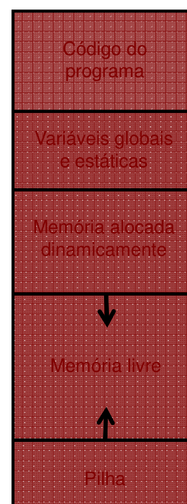
- Informalmente pode-se dizer que existem 3 maneiras de reservar espaços de memória para o armazenamento de informações:
  - Variáveis globais:
    - O espaço existe enquanto o programa estiver sendo executado.
  - Variáveis locais:
    - O espaço existe apenas enquanto a função que declarou a variável estiver sendo executada.
  - Alocação dinâmica:
    - O espaço permanece reservado até que o programa explicitamente libere a sua memória.
    - Por isso podemos alocar dinamicamente espaço em uma função e acessá-lo em uma outra função.

Uso da Memória

- A partir do momento em que um espaço de memória alocado dinâmicamente é liberado por um programa, esse espaço automaticamente deixa de estar reservado para o programa e não poderá mais ser acessado pelo programa.
- Se o programa não liberar um espaço alocado, ele será automaticamente liberado quando a execução do programa terminar.

Uso da Memória

- Na execução do programa, o código binário é carregago pelo SO em um espaço de memória.
- O SO também reserva espaços necessários para as variáveis globais e estáticas.
- O restante da memória livre é utilizado pelas variáveis locais e pelas variáveis alocadas dinamicamente.
- Cada vez que uma função é chamada, o sistema reserva oespaço necessário para as variáveis locais da função. Esse espaço pertence a pilha de execução e quando a função termina, é desempilhado.



Uso da Memória

- A parte de memória não ocupada pela pilha de execução pode ser requisitada dinamicamente.
- Se a pilha tentar crescer além do espaço disponível existente, diz-se que ela estourou e o programa é abortado com erro.
- O mesmo acontece com a memória alocada dinamicamente.