

IMD0030 – LINGUAGEM DE PROGRAMAÇÃO I

Aula 13 - Gerenciamento de memória.

Objetivo

- Introduzir os conceitos de alocação dinâmica e gerenciamento de memória em **C++**
- Para isso, estudaremos:
 - Alocação estática x alocação dinâmica
 - Ponteiros inteligentes
 - Comandos básicos de gerenciamento de memória
 - Uso da ferramenta **Valgrind**
- Ao final da aula espera-se que o aluno seja capaz de:
 - Distinguir a alocação estática da alocação dinâmica
 - Desenvolver programas capazes de gerenciar dinamicamente a memória do computador
 - Solucionar problemas de memória usando a ferramenta **Valgrind**

Alocação estática x dinâmica

- As linguagens de programação **C** e **C++** permitem dois tipos de alocação de memória:
 - Estática
 - Dinâmica
- Na alocação estática, o espaço de memória para as variáveis é reservado no início da execução, não podendo ser alterado depois

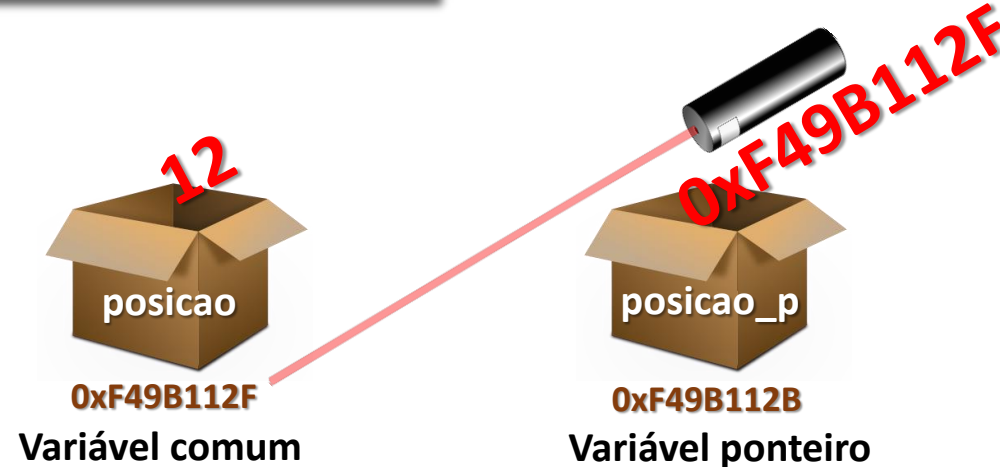
```
1  
2  int numero;  
3  int pontuacao[20];  
4  
5
```

- Na alocação dinâmica, o espaço de memória para as variáveis pode ser alocado dinamicamente durante a execução do programa
 - **Ponteiros se fazem necessários**

Revisão: Ponteiros (1)

- Tipo especial de variável que armazena endereços de memória e permite acessá-los diretamente

```
1  
2  
3     ...  
4     int posicao = 12;  
5     int *posicao_p;  
6     posicao_p = &posicao;  
7     ...
```



Endereço	Conteúdo	Valor
...
0xF49B1134	????	????
0xF49B1133	????	????
0xF49B1132	00001100	12
0xF49B1131	00000000	
0xF49B1130	00000000	
0xF49B112F	00000000	
0xF49B112E	00101111	
0xF49B112D	00010001	0xF49B112F
0xF49B112C	10011011	
0xF49B112B	11110100	
0xF49B112A	???	???
0xF49B1129	???	???
...

Revisão: Ponteiros (2)

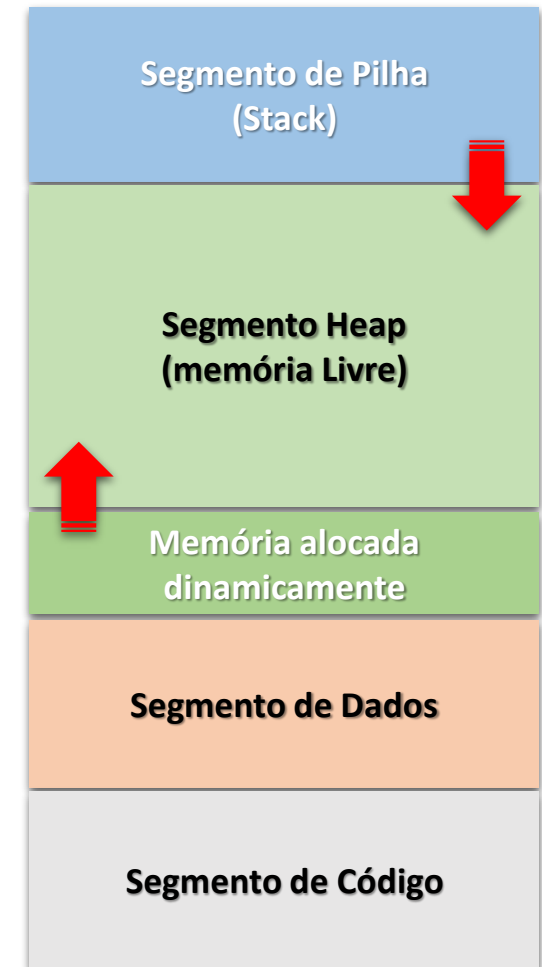
- Operadores utilizados para manipular ponteiros
 - Operador de acesso a memória **&** (**referenciamento**)
 - retorna o endereço de uma variável
 - Operador de indireção ***** (**desreferenciamento**)
 - retorna o conteúdo do endereço de uma variável apontada
 - * de indireção \neq * de multiplicação \neq * do tipo ponteiro

```
1  
2  
3  int posicao = 12;  
4  int *posicao_p = &posicao;  
5  int nova_posicao = *posicao_p;  
6  
7
```



Alocação dinâmica

- A alocação dinâmica de memória é um mecanismo bastante útil na solução de problemas que exigem grandes conjuntos de dados
 - Meio pelo qual um programa pode obter memória enquanto está em execução, sendo gerenciado pelo próprio programador
- Ela pode oferecer grandes benefícios em termos de desempenho e de utilização de recursos
 - A memória alocada dinamicamente é obtida através do segmento **HEAP**, onde apenas o espaço de memória necessário em um dado momento é efetivamente utilizado



Alocação Dinâmica

- As linguagens **C** e **C++** permitem que o programador tenha um alto grau de controle sobre a máquina através da alocação dinâmica
- Elas possuem ambas dois comandos básicos para gerenciamento de memória
 - Comandos da linguagem C++:
 - **new** aloca memória
 - **delete** libera memória alocada
 - Comandos da linguagem C:
 - **malloc** aloca memória
 - **free** libera memória alocada

Alocação dinâmica em C (1)

- Função **malloc** (**cstdlib**)

```
1  
2  
3 void* malloc( unsigned int numero_bytes );  
4  
5
```

- Aloca uma área de memória com **numero_bytes** bytes
- Retorna um ponteiro do tipo **void** para o início da área alocada, ou **NULL** caso não seja possível alocar a memória requisitada
 - O conteúdo deste ponteiro pode ser atribuído a qualquer variável do tipo ponteiro através de um **typecasting**

- Sintaxe para alocação de uma variável ponteiro do tipo **T**

```
1  
2  
3 T *p = (T*) malloc( sizeof( T ) ); // cast para o ponteiro do tipo T  
4  
5
```

Alocação dinâmica em C (2)

- Exemplo:

```
1  #include <iostream>
2  #include <cstdlib>
3
4  int main()
5  {
6      int *p = (int*) malloc( sizeof( int ) );
7      // alocação de variável ponteiro do tipo inteiro
8      // (int*) é o cast para ponteiro do tipo inteiro
9
10     if( p )
11         std::cout << "Memória alocada" << std::endl;
12     else
13         std::cout << "Alocacao impossivel" << std::endl;
14     return 0;
15 }
16
```

Alocação dinâmica em C (3)

- Função **free** (**cstdlib**)

```
1  
2  
3     ...  
4     void free( void *endereco );  
5     ...
```

- Libera a área de memória previamente alocada no sistema utilizando o seu endereço inicial como parâmetro
- O sistema operacional se encarrega de gerenciar lacunas do **heap**

- Exemplo:

```
1  #include <cstdlib>  
2  int main()  
3  {  
4      int *p = (int*) malloc( sizeof( int ) ); // alocação da variável  
5      if( p ) // verifica se a alocação ocorreu corretamente  
6          free( p ); // libera a memória alocada  
7      return 0;  
8  }  
9
```

Alocação dinâmica em C++ (1)

- Operador **new**
 - Aloca uma área de memória do tamanho correspondente à representação do tipo declarado
 - Retorna um ponteiro do tipo declarado apontando para o início da área alocada, ou **NULL** caso não seja possível alocar a memória requisitada
- Sintaxe para alocação de uma variável ponteiro do tipo **T**

```
1  
2  
3   ...  
4   T *p = new T;  
5   ...
```

- Exemplos:

```
1  
2   ...  
3   int *p = new int;  
4   float *q = new float;  
5   Ponto *umPonto = new Ponto;  
6   ...  
7
```

Alocação dinâmica em C++ (2)

- Exemplo:

```
1  #include <iostream>
2
3  int main()
4  {
5      int *p = new int; // alocação de variável ponteiro do tipo inteiro
6
7      /* IMPORTANTE: convém sempre verificar se a alocação ocorreu corretamente,
8       ou seja, se o retorno do operador new é diferente de NULL */
9
10     if( p )
11     {
12         std::cout << "Memoria alocada" << std::endl;
13         std::cout << p << std::endl; // imprime o endereço de p
14         std::cout << *p << std::endl; // imprime o conteúdo de p
15         *p = 10; // inicializa o conteúdo de p com 10
16         std::cout << *p << std::endl; // imprime o conteúdo de p
17     }
18     else
19         std::cout << "Alocacao impossivel" << std::endl;
20     return 0;
21 }
22
```

Alocação dinâmica em C++ (3)

- Operador **delete**
 - Libera a área de memória previamente alocada no sistema utilizando o seu endereço inicial como parâmetro
 - O sistema operacional se encarrega de gerenciar lacunas do **heap**
- Exemplo:

```
1  #include<iostream>
2
3  int main()
4  {
5      int *p = new int; // alocação de variável ponteiro do tipo inteiro
6      if( p )           // verifica se a alocação ocorreu corretamente
7          delete p;     // libera a memória alocada
8      return 0;
9  }
10
```

Erros comuns da alocação dinâmica

- Não alocar memória antes de acessar o conteúdo do ponteiro
 - Para acessar o conteúdo, sempre deve ser verificado se o ponteiro é válido
- Copiar o conteúdo do ponteiro ao invés do conteúdo da variável apontada
- Não liberar memória alocada previamente quando ela passar a ser desnecessária
- Tentar acessar o conteúdo de um ponteiro depois da sua memória já ter sido liberada
- O valor nulo (**0**) deve ser sempre atribuído ao ponteiro após à sua liberação de memória



Exercite-se (1)

- O que está errado neste programa?

```
1  #include <iostream>
2
3  int main()
4  {
5      int a, b, *p;
6      a = 2;
7      *p = 3;
8      b = a + (*p);
9      std::cout << a << std::endl;
10     return 0;
11 }
12
```

Exercite-se (1)

- O que está errado neste programa?

```
1  #include <iostream>
2
3  int main()
4  {
5      int a, b, *p;
6      a = 2;
7      *p = 3;
8      b = a + (*p);
9      std::cout << a << std::endl;
10     return 0;
11 }
12
```

- **Resposta:** A variável **p** (tipo ponteiro para inteiro) foi criada, porém não inicializada, ou seja, não aponta para nenhuma posição de memória válida (pode estar a apontar para “algum lugar” na memória, por causa do “lixo” deixado na posição onde **p** foi alocada). Assim, a instrução da **linha 7** poderá causar um erro de acesso à memória!!!

Exercite-se (2)

- O que será impresso no seguinte programa?

```
1  #include <iostream>
2  int main()
3  {
4      double a, *p, *q;
5      a = 3.14;
6      std::cout << a << std::endl;
7      p = &a;
8      *p = 2.718;
9      std::cout << a << std::endl;
10     a = 5;
11     std::cout << *p << std::endl;
12     p = NULL;
13     p = new double;
14     *p = 20;
15     q = p;
16     std::cout << *p << std::endl;
17     std::cout << a << std::endl;
18     delete p;
19     std::cout << *q << std::endl;
20     return 0;
21 }
22
```

Exercite-se (2)

- O que será impresso no seguinte programa?

```
1  #include <iostream>
2  int main()
3  {
4      double a, *p, *q;
5      a = 3.14;
6      std::cout << a << std::endl;
7      p = &a;
8      *p = 2.718;
9      std::cout << a << std::endl;
10     a = 5;
11     std::cout << *p << std::endl;
12     p = NULL;
13     p = new double;
14     *p = 20;
15     q = p;
16     std::cout << *p << std::endl;
17     std::cout << a << std::endl;
18     delete p;
19     std::cout << *q << std::endl;
20     return 0;
21 }
22
```

Resposta:

#> 3.14

#> 2.718

#> 5

#> 20

#> 5

#> 20

Alocação dinâmica de vetores

- Como **vetores são ponteiros** em linguagem **C/C++**, a alocação dinâmica de vetores segue a lógica utilizada para as variáveis do tipo vetor simples, baseiando-se na indexação dos elementos
 - A liberação de memória de vetores deve ser efetuada com o comando **delete []**

```
1  #include <iostream>
2
3  int main()
4  {
5      int a[10] = {1,2,3,4,5,6,7,8,9,0};
6      int *b;
7      b = a;
8
9      b[5] = 100;
10     std::cout << a[5] << std::endl;
11     std::cout << b[5];
12     return 0;
13 }
14
```

Alocação Estática

```
1  #include <iostream>
2
3  int main()
4  { // b = a não é permitido aqui
5      int a[10] = {1,2,3,4,5,6,7,8,9,0};
6      int *b;
7      b = new int[10];
8
9      b[5] = 100;
10     std::cout << a[5] << std::endl;
11     std::cout << b[5];
12     delete[] b;
13     return 0;
14 }
15
```

Alocação Dinâmica

Alocação dinâmica de matrizes (1)

- Alocação de matrizes se faz da mesma forma que para vetores, incrementada do conceito de indireção múltipla
- A indireção múltipla (ponteiro de ponteiros) se aplica a qualquer dimensão desejada

```
1  #include <iostream>
2  int main()
3  {
4      float **matriz; // ponteiro de ponteiros para a matriz
5      int linhas = 10, colunas = 15;
6      matriz = new float*[linhas]; // aloca as linhas da matriz
7
8      if( matriz != NULL )
9          for( int i = 0; i < linhas; i++ )
10             {
11                 matriz[i] = new float[colunas]; // aloca as colunas da matriz
12                 if( matriz[i] == NULL ) {
13                     std::cout << "Memoria Insuficiente" << std::endl;
14                     break;
15                 }
16             }
17      return 0;
18  }
19
```

Alocação dinâmica de matrizes (2)

- A liberação de memória das matrizes deve ser efetuada para todos os ponteiros da indireção múltipla

```
1  #include <iostream>
2
3  int main()
4  {
5      float **matriz; // ponteiro de ponteiros para a matriz
6      int linhas = 10, colunas = 15;
7
8      ... // Considerando a alocação de memória efetuada
9      if( matriz != NULL )
10     {
11         for( int i = 0; i < linhas; i++ )
12             delete [] matriz[i]; // libera as colunas da matriz
13         delete [] matriz;        // libera as linhas da matriz
14     }
15     return 0;
16 }
17
```

Alocação dinâmica de registros

- Registros são tipos compostos definidos pelo usuário que podem ser alocados dinamicamente da mesma forma que tipos primitivos

```
1  #include <iostream>
2  typedef struct {
3      int idade;
4      double salario;
5  } Registro;
6
7  int main()
8  {
9      Registro *r;
10     r = new Registro;
11     if( r )
12     {
13         r->idade = 30;
14         r->salario = 1000.;
15         delete r;
16         r = 0; // garante que o ponteiro não aponta mais
17               // para o espaço de memória liberado
18     }
19     return 0;
20 }
```

Use *nullptr* ao invés de *NULL*

```
#include <iostream>

int main(int argc, char const *argv[])
{
    int * x = nullptr;

    int a = 89;

    x = &a;

    std::cout << "Endereco de x = " << &x << std::endl;
    std::cout << "Valor de x = " << x << std::endl;
    std::cout << "Endereco de a = " << &a << std::endl;
    std::cout << "Valor apontado por x = " << *x << std::endl;
    return 0;
}
```

- Em C++11, variáveis do tipo ponteiro devem ser inicializados com o valor especial *nullptr*
- A palavra reservada *nullptr* foi introduzida no C++ para representar o endereço 0
- Exemplo de declaração e inicialização de um ponteiro com o valor *nullptr*:

Ponteiros para constantes

- Se quisermos armazenar o endereço de uma constante em um ponteiro, precisamos usar um ponteiro para constante
- Exemplo:

```
const int SIZE = 6;
const double taxas[SIZE] = { 18.55, 17.45, 12.85, 14.97, 10.35, 18.89 };

void mostraTaxas(const double *taxas, int size) {
    for (int count = 0; count < size; ++count) {
        cout << "Taxa " << (count + 1) << " é " << *(taxas + count) << "%." << endl;
    }
}
```

Deduzindo tipos com o uso de **auto**

- Em C++11 uma variável pode ser declarada como sendo “do tipo” **auto**
 - Isso diz ao compilador que o tipo da variável deverá ser deduzido de seus inicializadores (por isso devem ser inicializadas)
- Exemplos de uso:
 - `int x;` // a variável x é do tipo int – declaração explícita (tipada)
 - `auto x = 10;` // variável x é deduzida como sendo inteira, já que 10 é um inteiro

Uso comum do auto

```
using std::string;
using std::vector;

std::vector<std::string> nomes;

//
// Como é hoje
//
for(std::vector<string>::iterator i = nomes.begin() ; i != nomes.end() ; ++i)
{

}

//
// Muito mais fácil de ler.
//
for(auto i = nomes.begin() ; i != nomes.end() ; ++i)
{

}
```

Valgrind

- **Valgrind** um framework (para Linux) de análise de programa em tempo de execução
 - Similar ao Valgrind, no Windows é possível usar o “Dr. Memory”, disponível em: <http://www.drmemory.org/>
- Essa ferramenta permite monitorar como o programa faz uso da memória
 - Para isso, mantém um mapa de bits indicando quais áreas da memória estão alocadas, quais estão livres e quais estão alocadas e iniciadas
- Com isso, é possível encontrar os seguintes tipos de problemas:
 - Leitura/escrita em áreas de memória já desalocadas, não alocadas, que ultrapassam uma área alocada ou que sejam impróprias ou incomuns na pilha de execução
 - Vazamentos de memória (memory leaks)
 - Uso de variáveis ou ponteiros não inicializados
 - Passagem de apontadores para áreas não endereçáveis
 - Uso incorreto das funções malloc, calloc, free, new e delete
 - Sobreposição de ponteiros no uso das funções memcpy, strcpy e semelhantes

Valgrind

- O framework **Valgrind** é composto por múltiplas ferramentas:
 - **Callgrind** – profiler
 - **Massif** – heap profiler
 - **Helgrind** – detector de race-conditions
 - **Memcheck** – ferramenta mais usada para detecção de problemas de memória
- O uso do Valgrind em complemento ao **GDB** (depurador GNU) permite uma depuração completa do código
- Para maiores detalhes, consulte o manual em:
 - <http://valgrind.org/docs/manual/manual.html>

Usando o Valgrind

- Para usar o Valgrind é muito simples!
 - Compile normalmente o seu código usando o compilador g++ (também funciona para C com o gcc)
 - Embora seja possível inspecionar o seu programa com o valgrind após compila-lo com opções de depuração (as diretivas **-g -O0**), **recomenda-se removê-las**, pois os depuradores também fazem uso de memória e inserem código próprio no seu programa
 - Com isso, o resultado que o Valgrind irá apresentar pode diferir da pura execução de seu código
 - Execute o seu programa inserindo a palavra valgrind antes do nome do seu executável
 - Exemplo: **valgrind ./lab01**
 - Será impresso na tela as informações sobre o uso de memória do programa
 - Leia atentamente o relatório do valgrind!!!

Usando o Valgrind

- Por omissão, o valgrind ativa a ferramenta de verificação de memória (o que equivale à opção **--tool=memcheck**)
- Outras opções úteis
 - Para maiores detalhes destes erros deve-se executar o aplicativo novamente inserindo a flag **-v** logo após a palavra valgrind:
 - Exemplo: **valgrind -v ./lab01**
 - Para maiores detalhes dos blocos com problemas de alocação, deve-se executar o aplicativo novamente inserindo as flags **--leak-check=full --show-reachable=yes**
 - Exemplo: **valgrind -v --leak-check=full --show-reachable=yes ./lab01**
- Seguem alguns exemplos...

Alocação dinâmica de memória: Problemas comuns

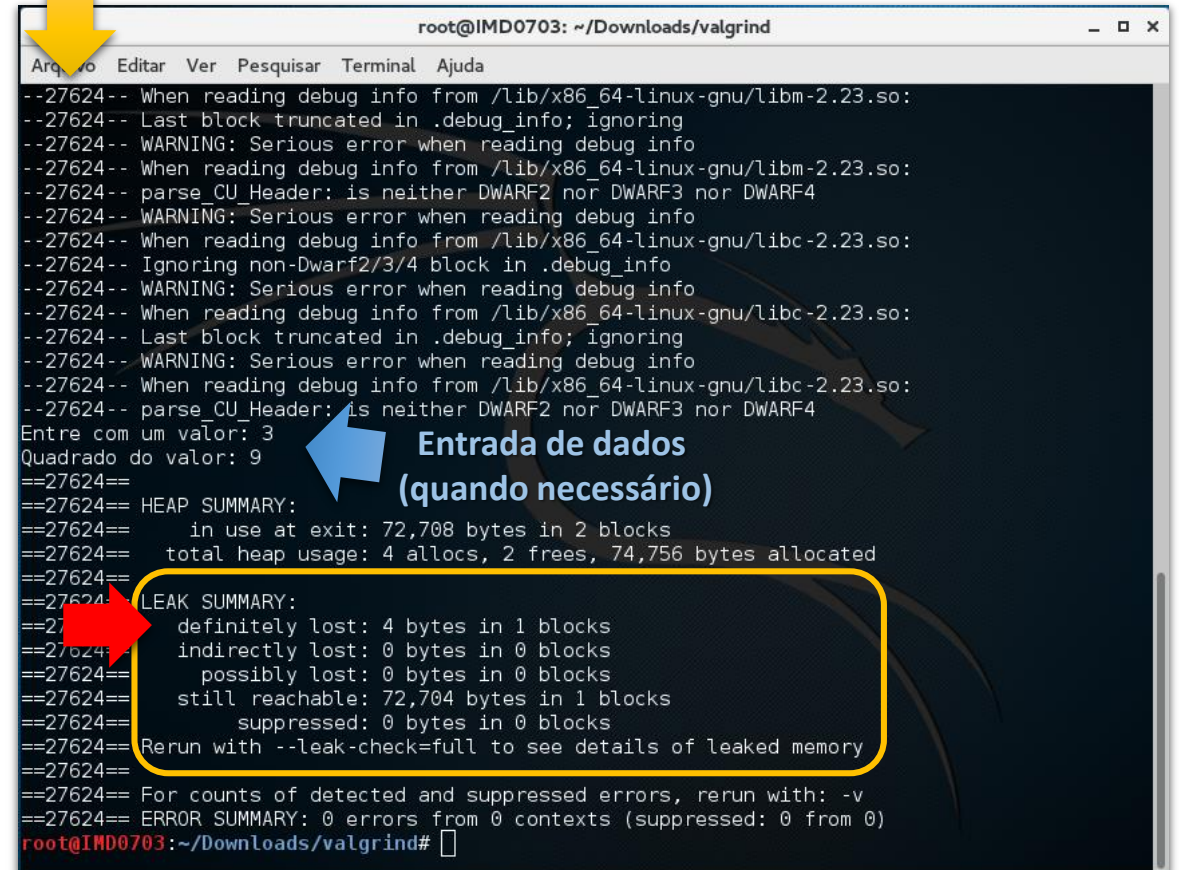
- Memory Leak
 - Memória que é alocada não é devolvida (liberada)
- Buffer Overflow
 - Escrever fora da área alocada
- Memória não inicializada
 - Acesso a memória não inicializada
 - Pode induzir comportamentos “aleatórios”
- Double-free
 - A área de memória de um ponteiro é liberada mais do que uma vez
 - Origina crash na maioria das libcs

Memory leak

- Memória que é alocada não é devolvida (liberada)

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(void)
6  {
7      int valor;
8      int *p;
9      p = new int;
10     cout << "Entre com um valor: ";
11     cin >> valor;
12     (*p) = valor;
13     (*p) *= valor;
14     cout << "Quadrado do valor: " << (*p) << endl;
15 }
16
```

ID do
processo



```
root@IMD0703: ~/Downloads/valgrind
Arquivo Editar Ver Pesquisar Terminal Ajuda
--27624-- When reading debug info from /lib/x86_64-linux-gnu/libm-2.23.so:
--27624-- Last block truncated in .debug_info; ignoring
--27624-- WARNING: Serious error when reading debug info
--27624-- When reading debug info from /lib/x86_64-linux-gnu/libm-2.23.so:
--27624-- parse_CU_Header: is neither DWARF2 nor DWARF3 nor DWARF4
--27624-- WARNING: Serious error when reading debug info
--27624-- When reading debug info from /lib/x86_64-linux-gnu/libc-2.23.so:
--27624-- Ignoring non-Dwarf2/3/4 block in .debug_info
--27624-- WARNING: Serious error when reading debug info
--27624-- When reading debug info from /lib/x86_64-linux-gnu/libc-2.23.so:
--27624-- Last block truncated in .debug_info; ignoring
--27624-- WARNING: Serious error when reading debug info
--27624-- When reading debug info from /lib/x86_64-linux-gnu/libc-2.23.so:
--27624-- parse_CU_Header: is neither DWARF2 nor DWARF3 nor DWARF4
Entre com um valor: 3
Quadrado do valor: 9
==27624==
==27624== HEAP SUMMARY:
==27624==   in use at exit: 72,708 bytes in 2 blocks
==27624==   total heap usage: 4 allocs, 2 frees, 74,756 bytes allocated
==27624==
==27624== LEAK SUMMARY:
==27624==   definitely lost: 4 bytes in 1 blocks
==27624==   indirectly lost: 0 bytes in 0 blocks
==27624==   possibly lost: 0 bytes in 0 blocks
==27624==   still reachable: 72,704 bytes in 1 blocks
==27624==   suppressed: 0 bytes in 0 blocks
==27624== Rerun with --leak-check=full to see details of leaked memory
==27624==
==27624== For counts of detected and suppressed errors, rerun with: -v
==27624== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
root@IMD0703:~/Downloads/valgrind#
```


Memória não inicializada

- Acesso a memória não inicializada
- Pode induzir comportamentos “aleatórios”

```
1  #include <iostream>
2  #include <cstring>
3
4  using namespace std;
5
6  int main(void)
7  {
8      char *buffer = new char[1024];
9      bool condicao = false;
10     if (condicao)
11     {
12         strcpy(buffer, "teste");
13     }
14     cout << "Frase: " << buffer << endl;
15     delete[] buffer;
16 }
17
```



```
root@IMD0703: ~/Downloads/valgrind
Arquivo Editar Ver Pesquisar Terminal Ajuda
--29150-- When reading debug info from /lib/x86_64-linux-gnu/libm-2.23.so:
--29150-- parse_CU_Header: is neither DWARF2 nor DWARF3 nor DWARF4
--29150-- WARNING: Serious error when reading debug info
--29150-- When reading debug info from /lib/x86_64-linux-gnu/libc-2.23.so:
--29150-- Ignoring non-Dwarf2/3/4 block in .debug_info
--29150-- WARNING: Serious error when reading debug info
--29150-- When reading debug info from /lib/x86_64-linux-gnu/libc-2.23.so:
--29150-- Last block truncated in .debug_info; ignoring
--29150-- WARNING: Serious error when reading debug info
--29150-- When reading debug info from /lib/x86_64-linux-gnu/libc-2.23.so:
--29150-- parse_CU_Header: is neither DWARF2 nor DWARF3 nor DWARF4
==29150== Conditional jump or move depends on uninitialised value(s)
==29150==    at 0x4C2EFE9: strlen (vg_replace_strmem.c:454)
==29150==    by 0x4F44A78: std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::
:char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, char const*) (in /usr/lib/
x86_64-linux-gnu/libstdc++.so.6.0.22)
==29150==    by 0x400906: main (valgrind4.cpp:14)
==29150==
Frase:
==29150==
==29150== HEAP SUMMARY:
==29150==    in use at exit: 72,704 bytes in 1 blocks
==29150==    total heap usage: 3 allocs, 2 frees, 74,752 bytes allocated
==29150==
==29150== LEAK SUMMARY:
==29150==    definitely lost: 0 bytes in 0 blocks
==29150==    indirectly lost: 0 bytes in 0 blocks
==29150==    possibly lost: 0 bytes in 0 blocks
==29150==    still reachable: 72,704 bytes in 1 blocks
==29150==    suppressed: 0 bytes in 0 blocks
==29150== Rerun with --leak-check=full to see details of leaked memory
==29150==
==29150== For counts of detected and suppressed errors, rerun with: -v
==29150== Use --track-origins=yes to see where uninitialised values come from
==29150== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
root@IMD0703:~/Downloads/valgrind#
```

Double-free

- A área de memória de um ponteiro é liberada mais do que uma vez
 - Origina crash na maioria das libcs

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(void)
6  {
7      int valor;
8      int *p;
9      p = new int;
10     cout << "Entre com um valor: ";
11     cin >> valor;
12     (*p) = valor;
13     (*p) *= valor;
14     cout << "Quadrado do valor: " << valor << endl;
15     delete p;
16     delete p;
17 }
18
```



```
root@IMD0703: ~/Downloads/valgrind
Arquivo Editar Ver Pesquisar Terminal Ajuda
tor new(unsigned long))
Entre com um valor: 3
--29893-- REDIR: 0x5754ac0 (libc.so.6: __GI_memcpy) redirected to 0x4c33020 (__GI_memcpy)
--29893-- REDIR: 0x57542c0 (libc.so.6: bcmp) redirected to 0x4a266f0 (_vgnU_ifunc_wrapper)
--29893-- REDIR: 0x5819ce0 (libc.so.6: __memcmp_sse4_1) redirected to 0x4c31d50 (__memcmp_sse4_1)
Quadrado do valor: 3
--29893-- REDIR: 0x4ec57a0 (libstdc++.so.6: operator delete(void*)) redirected to 0x4c2d260 (operator d
elete(void*))
==29893== Invalid free() / delete / delete[] / realloc()
==29893==    at 0x4C2D2CB: operator delete(void*) (vg_replace_malloc.c:575)
==29893==    by 0x400A28: main (valgrind2.cpp:16)
==29893== Address 0x5a85c80 is 0 bytes inside a block of size 4 free'd
==29893==    at 0x4C2D2CB: operator delete(void*) (vg_replace_malloc.c:575)
==29893==    by 0x400A1C: main (valgrind2.cpp:15)
==29893== Block was alloc'd at
==29893==    at 0x4C2C16F: operator new(unsigned long) (vg_replace_malloc.c:333)
==29893==    by 0x4009A8: main (valgrind2.cpp:9)
--29893-- REDIR: 0x574b990 (libc.so.6: free) redirected to 0x4c2ce00 (free)
==29893==
==29893== HEAP SUMMARY:
==29893==   in use at exit: 72,704 bytes in 1 blocks
==29893==   total heap usage: 4 allocs, 4 frees, 74,756 bytes allocated
==29893==
==29893== Searching for pointers to 1 not-freed blocks
==29893== Checked 114,288 bytes
==29893==
==29893== 72,704 bytes in 1 blocks are still reachable in loss record 1 of 1
==29893==    at 0x4C2BC0F: malloc (vg_replace_malloc.c:299)
==29893==    by 0x4EC38CF: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.22)
==29893==    by 0x400F3F9: call_init.part.0 (in /lib/x86_64-linux-gnu/ld-2.23.so)
==29893==    by 0x400F50A: _dl_init (in /lib/x86_64-linux-gnu/ld-2.23.so)
==29893==    by 0x4000CF9: ??? (in /lib/x86_64-linux-gnu/ld-2.23.so)
==29893==
==29893== LEAK SUMMARY:
==29893==   definitely lost: 0 bytes in 0 blocks
==29893==   indirectly lost: 0 bytes in 0 blocks
==29893==   possibly lost: 0 bytes in 0 blocks
==29893==   still reachable: 72,704 bytes in 1 blocks
==29893==   suppressed: 0 bytes in 0 blocks
==29893==
```


Buffer Overflow

- Escrever fora da área alocada

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(void)
6  {
7      int *vetor;
8      vetor = new int[5];
9      for (int i=0;i<=5;i++)
10     {
11         vetor[i]=i;
12     }
13     delete[] vetor;
14 }
15
```




```
root@IMD0703: ~/Downloads/valgrind
Arquivo  Editar  Ver  Pesquisar  Terminal  Ajuda
--29148-- When reading debug info from /lib/x86_64-linux-gnu/libm-2.23.so:
--29148-- Last block truncated in .debug_info; ignoring
--29148-- WARNING: Serious error when reading debug info
--29148-- When reading debug info from /lib/x86_64-linux-gnu/libm-2.23.so:
--29148-- parse_CU_Header: is neither DWARF2 nor DWARF3 nor DWARF4
--29148-- WARNING: Serious error when reading debug info
--29148-- When reading debug info from /lib/x86_64-linux-gnu/libc-2.23.so:
--29148-- Ignoring non-Dwarf2/3/4 block in .debug_info
--29148-- WARNING: Serious error when reading debug info
--29148-- When reading debug info from /lib/x86_64-linux-gnu/libc-2.23.so:
--29148-- Last block truncated in .debug_info; ignoring
--29148-- WARNING: Serious error when reading debug info
--29148-- When reading debug info from /lib/x86_64-linux-gnu/libc-2.23.so:
--29148-- parse_CU_Header: is neither DWARF2 nor DWARF3 nor DWARF4
==29148== Invalid write of size 4
==29148==    at 0x400760: main (valgrind3.cpp:11)
==29148==    Address 0x5a85c94 is 0 bytes after a block of size 20 alloc'd
==29148==    at 0x4C2C88F: operator new[](unsigned long) (vg_replace_malloc.c:422)
==29148==    by 0x400737: main (valgrind3.cpp:8)
==29148==
==29148== HEAP SUMMARY:
==29148==    in use at exit: 72,704 bytes in 1 blocks
==29148==    total heap usage: 2 allocs, 1 frees, 72,724 bytes allocated
==29148==
==29148== LEAK SUMMARY:
==29148==    definitely lost: 0 bytes in 0 blocks
==29148==    indirectly lost: 0 bytes in 0 blocks
==29148==    possibly lost: 0 bytes in 0 blocks
==29148==    still reachable: 72,704 bytes in 1 blocks
==29148==    suppressed: 0 bytes in 0 blocks
==29148== Rerun with --leak-check=full to see details of leaked memory
==29148==
==29148== For counts of detected and suppressed errors, rerun with: -v
==29148== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
root@IMD0703:~/Downloads/valgrind#
```

Buffer Overflow

- Outro exemplo

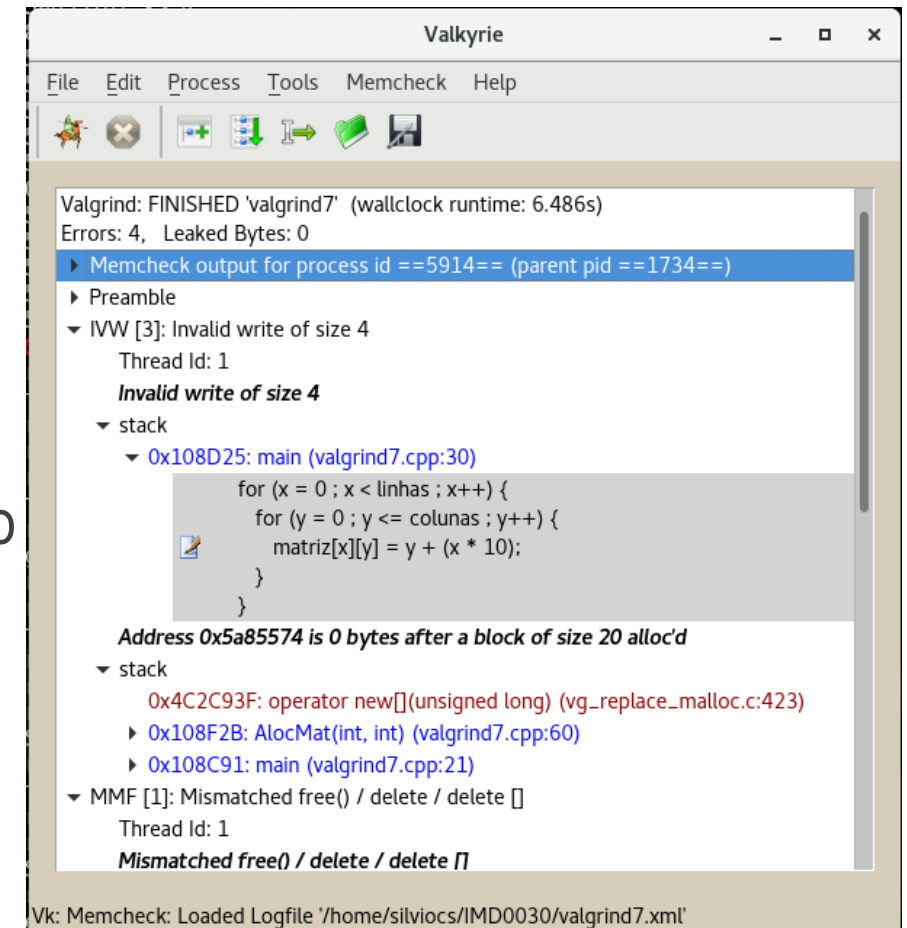
```
1  #include <iostream>
2  #include <cstring>
3
4  using namespace std;
5
6  int main(void)
7  {
8      char *buffer = new char[18];
9      const char *nome = "Testando o programa";
10     int i=0;
11     do
12     {
13         *buffer++ = nome[i];
14         i++;
15     } while (nome[i]);
16     cout << "Frase: " << buffer << endl;
17     delete[] buffer;
18 }
19
```



```
root@IMD0703: ~/Downloads/valgrind
Arquivo  Editar  Ver  Pesquisar  Terminal  Ajuda
--29152-- parse_CU_header: is neither DWARF2 nor DWARF3 nor DWARF4
--29152-- Invalid write of size 1
--29152--    at 0x4008F7: main (valgrind5.cpp:13)
--29152--    Address 0x5a85c92 is 0 bytes after a block of size 18 alloc'd
--29152--    at 0x4C2C88F: operator new[](unsigned long) (vg_replace_malloc.c:422)
--29152--    by 0x4008C7: main (valgrind5.cpp:8)
--29152-- Invalid read of size 1
--29152--    at 0x4C2EFE2: strlen (vg_replace_strmem.c:454)
--29152--    by 0x4F44A78: std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::ch
ar_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, char const*) (in /usr/lib/x86_64
-linux-gnu/libstdc++.so.6.0.22)
--29152--    by 0x400933: main (valgrind5.cpp:16)
--29152--    Address 0x5a85c93 is 1 bytes after a block of size 18 alloc'd
--29152--    at 0x4C2C88F: operator new[](unsigned long) (vg_replace_malloc.c:422)
--29152--    by 0x4008C7: main (valgrind5.cpp:8)
--29152-- Frase:
--29152-- Invalid free() / delete / delete[] / realloc()
--29152--    at 0x4C2D7CB: operator delete[](void*) (vg_replace_malloc.c:620)
--29152--    by 0x400953: main (valgrind5.cpp:17)
--29152--    Address 0x5a85c93 is 1 bytes after a block of size 18 alloc'd
--29152--    at 0x4C2C88F: operator new[](unsigned long) (vg_replace_malloc.c:422)
--29152--    by 0x4008C7: main (valgrind5.cpp:8)
--29152--
--29152-- HEAP SUMMARY:
--29152--    in use at exit: 72,722 bytes in 2 blocks
--29152--    total heap usage: 3 allocs, 2 frees, 73,746 bytes allocated
--29152--
--29152-- LEAK SUMMARY:
--29152--    definitely lost: 18 bytes in 1 blocks
--29152--    indirectly lost: 0 bytes in 0 blocks
--29152--    possibly lost: 0 bytes in 0 blocks
--29152--    still reachable: 72,704 bytes in 1 blocks
--29152--    suppressed: 0 bytes in 0 blocks
--29152-- Rerun with --leak-check=full to see details of leaked memory
--29152--
--29152-- For counts of detected and suppressed errors, rerun with: -v
--29152-- ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 0 from 0)
root@IMD0703:~/Downloads/valgrind#
```

Usando o Valgrind

- Exportando resultados do Valgrind:
valgrind --leak-check=yes --xml=yes --xml-file=prog.xml ./exemplo
- Interface para analisar resultados:
valkyrie -l prog.xml
- O programa vakyrie permite visualizar o relatório gerado pelo Valgrind com a ajuda de uma interface gráfica
- Para instalar o valkyrie:
\$ sudo apt-get install valkyrie



Uso de *smart pointers* para evitar *memory leak*

- O C++ 11 é possível usar ponteiros inteligentes (*smart pointers*) para alocar memória dinamicamente sem ter que se preocupar com sua liberação após acabar o seu uso
- São definidos três tipos de ponteiros inteligentes:
 - **unique_ptr**
 - Substitui o **auto_ptr** (obsoleto)
 - **shared_ptr**
 - **weak_ptr**
- Os ponteiros inteligentes são definidos no namespace std no arquivo de cabeçalho `<memory>`:
 - `#include <memory>`

Na prática...

```
void UsaPonteiroBruto()
{
    // Utiliza um ponteiro bruto -- não recomendado.
    Dado* umDado = new Dado();

    // Usa o umDado...

    // Não se esqueça de liberar da memória!
    delete umDado;
}

void UseSmartPointer()
{
    // Declara como um ponteiro inteligente.
    unique_ptr<Dado> umDado = new Dado();

    // Usa um Dado...

} // umDado é liberado da memória (removido) automaticamente.
```

Atividade para a próxima aula

- Pesquise sobre o uso de ponteiros inteligentes (*smart pointers*) no C++11 ou superior
- Apresente as diferenças no uso de cada tipo suportado:
 - `unique_ptr`
 - `shared_ptr`
 - `weak_ptr`
- Elabore exemplos (funções ou trechos de código) do uso de cada tipo de ponteiro inteligente, usando como base o seu jogo de dados já implementado
 - Justifique bem o uso de cada tipo
- Deverá ser feito **EM DUPLA** e entregue na forma de relatório
 - Entrega via SIGAA até 23:59h de 13/09/2017

Resumo da aula

- A alocação dinâmica é um mecanismo que permite um programa obter memória durante a sua execução, sendo gerenciado pelo próprio programador
 - Ela oferece grandes benefícios em termos de desempenho e de utilização de recursos
- As linguagens **C** e **C++** permitem que o programador tenha um alto grau de controle sobre a máquina através da alocação dinâmica
 - Os comandos básicos para gerenciamento de memória são **new** e **delete** em linguagem **C++** e **malloc** e **free** em linguagem **C**
- Deve-se ter bastante cuidado com a manipulação de ponteiros utilizados para alocação dinâmica
 - Em caso de problemas de memória, ferramentas de apoio como o **Valgrind** permitem verificar como o programa utiliza a memória

Alguma Questão?

