Aula 3

Programação II

Prof. Sandino Jardim

CC-UFMT-CUA

Agenda

- Alocação padrão de memória
- Alocação dinâmica
- Ponteiros inteligentes

Alocação padrão de memória

- Objetos globais
  - Alocados no início do programa e destruídos ao seu final
- Objetos locais
  - Alocados e destruídos enquanto o bloco em que está contido está em execução
- Objetos `static` locais
  - Alocados antes de sua utilização e destruídos ao final do programa

```cpp
#include <iostream>

// Objeto global
int globalVar = 0;

void demonstrateLocal() {
// Objeto local
int localVar = 0;
std::cout << "Local variable (inside function): " << localVar << std::endl;
localVar++;
}

void demonstrateStaticLocal() {
// Objeto local static
static int staticLocalVar = 0;
std::cout << "Static local variable (inside function): " << staticLocalVar << std::endl;
staticLocalVar++;
}

int main() {
std::cout << "Global variable (initial): " << globalVar << std::endl;
globalVar++;

demonstrateLocal(); // Primeira chamada
demonstrateLocal(); // Segunda chamada (o valor da variável local será reiniciado)

demonstrateStaticLocal(); // Primeira chamada
demonstrateStaticLocal(); // Segunda chamada (o valor da variável estática local será preservado)

std::cout << "Global variable (final): " << globalVar << std::endl;
return 0;
}
```

Exemplos de uso static

```
//Contadores de Função
void countCalls() {
    static int callCount = 0;
    callCount++;
    std::cout << "Function called " << callCount << "
    times" << std::endl;
}

//Inicialização uma só vez
void initializeOnce() {
    static bool initialized = false;
    if (!initialized) {
      // Realiza a inicialização
      initialized = true;
      std::cout << "Initialized once." << std::endl;
    }
    else {
      std::cout << "Already initialized." << std::endl;
    }
}
```

```
//Retenção de estado entre chamadas
void accumulate(int value) {
  static int sum = 0;
  sum += value;
  std::cout << "Accumulated sum: " << sum << std::endl;
}

//Persistência de Dados em Funções Recursivas
void recursiveFunction(int n) {
  static int depth = 0;
  depth++;
  std::cout << "Recursion depth: " << depth << std::endl;
  if (n > 0) {
    recursiveFunction(n - 1);
  }
  depth--;
}
```

Alocação dinâmica de memória

- Assim como em C, C++ permite-nos alocar objetos dinamicamente
- Objetos dinamicamente alocados possuem um tempo de vida independente de onde foram criados
- Existirão até que sejam explicitamente liberados.

Memória dinâmica e *Smart Pointers*

- Operadores de gerenciamento de memória em C++:
  - **`new`**
    - Aloca e inicializa (opcional) um objeto na memória dinâmica e retorna um ponteiro para ele
  - **`delete`**
    - Dado um ponteiro para um objeto dinâmico, destrói o objeto e libera a memória associada a ele.

O operador **new**

- Retorna um ponteiro para um objeto que ele aloca, inicializando quando é um objeto de classe definida.

```
string *ps = new string;   //  initialized to empty string
int *pi = new int;         //  pi points to an uninitialized int
```

O operador **new**

- Outras formas de inicialização

```
int *pi = new int(1024); // object to which pi points has value 1024
string *ps = new string(10, '9');    // *ps is "9999999999"
// vector with ten elements with values from 0 to 9
vector<int> *pv = new vector<int>{0,1,2,3,4,5,6,7,8,9};
```

```
string *ps1 = new string;   // default initialized to the empty string
string *ps = new string();  // value initialized to the empty string
int *pi1 = new int;         // default initialized; *pi1 is undefined
int *pi2 = new int();       // value initialized to 0; *pi2 is 0
```

O operador **`delete`**

- Para prevenir contra exaustão de memória, devemos retornar dinamicamente a memória alocada para o sistema após utilizá-la
- O operador destrói o objeto e libera a memória correspondente.

```
delete p;       //  p must point to a dynamically allocated object or be null

int i, *pi1 = &i, *pi2 = nullptr;
double *pd = new double(33), *pd2 = pd;
delete i;    //  error: i is not a pointer
delete pi1;  //  undefined: pi1 refers to a local
delete pd;   //  ok
delete pd2;  //  undefined: the memory pointed to by pd2 was already freed
delete pi2;  //  ok: it is always ok to delete a null pointer
```

Tempo de vida de objetos dinamicamente alocados

- Objetos dinamicamente alocados a partir do operador **`new`** existem até que sejam explicitamente deletados
- Funções que retornam ponteiros devem se lembrar de liberar memória quando não mais utilizadas

```
//  factory returns a pointer to a dynamically allocated object
Foo* factory(T arg)
{
    //  process arg as appropriate
    return new Foo(arg); //  caller is responsible for deleting this memory
}
```

Tempo de vida de objetos dinamicamente alocados

- Exemplos de uso

```
void use_factory(T arg)
{
    Foo *p = factory(arg);
    // use p but do not delete it
} // p goes out of scope, but the memory to which p points is not freed!
```

```
void use_factory(T arg)
{
    Foo *p = factory(arg);
    // use p
    delete p;   // remember to free the memory now that we no longer need it
}
```

```
Foo* use_factory(T arg)
{
    Foo *p = factory(arg);
    // use p
    return p;   // caller must delete the memory
}
```

Redefinindo valor de ponteiro após **`delete`**

- Mesmo após liberação, o ponteiro pode continuar mantendo o endereço da memória liberada
- O ponteiro em questão se torna um "ponteiro pendurado"
- Problemas são mitigados quando a memória é liberada exatamente antes do ponteiro sair de escopo
- Atribuindo `nullptr` garante que o ponteiro deixará de apontar para a memória já liberada

Solução com proteção limitada

- Se mais de um ponteiro estava apontando para aquele espaço, delete surte efeito nos demais.

```
int *p(new int(42));   //  p points to dynamic memory
auto q = p;            //  p and q point to the same memory
delete p;     //  invalidates both p and q
p = nullptr; //  indicates that p is no longer bound to an object
```

Risco de dupla liberação de memória

```
void doubleFree() {
    int* ptr = new int(10); // Aloca memória na heap
    int* anotherPtr = ptr; // Outro ponteiro aponta para a mesma memória

    delete ptr; // Primeira desalocação
    // ptr ainda aponta para a memória desalocada
    // anotherPtr ainda aponta para a mesma memória desalocada

    delete anotherPtr; // Segunda desalocação - comportamento indefinido
}
```

Problemas usando **`new`** e **`delete`**

- Três problemas comuns:
  - Esquecer de liberar memória
    - *Memory leak* – a memória nunca é retornada para o programa
    - Difícil de detectar
  - Usar ponteiro depois da memória liberada
    - Mitigado quando torna-se o ponteiro nulo
  - Liberando a mesma memória duas vezes
    - Se dois ponteiros apontam a mesma memória
- Solução
  - Smart Pointers!

Smart Pointers

- Ponteiros especiais
  - Atuam como ponteiros normais, exceto que deletam automaticamente o objeto para o qual apontam
  - **`shared_ptr`**
    - Permite que múltiplos ponteiros apontem para o mesmo objeto
  - **`unique_ptr`**
    - Aponta para um único objeto
  - **`weak_ptr`**
    - Referência "fraca" para um objeto apontado por **`shared_ptr`**

A classe **shared_ptr**

- Declaração usando *template*

```
shared_ptr<string> p1;     //  shared_ptr that can point at a string
shared_ptr<list<int>> p2; //  shared_ptr that can point at a list of ints
```

- Exemplo de utilização

```
//  if p1 is not null, check whether it's the empty string
if (p1 && p1->empty())
    *p1 = "hi";   //  if so, dereference p1 to assign a new value to that string
```

A função **`make_shared`**

- Meio mais seguro de alocar e utilizar memória dinamicamente
- Aloca e inicializa um objeto na memória dinâmica e retorna **`shared_ptr`**
- Deve-se especificar o tipo de objeto desejado na inicialização

```
// shared_ptr that points to an int with value 42
shared_ptr<int> p3 = make_shared<int>(42);
// p4 points to a string with value 9999999999
shared_ptr<string> p4 = make_shared<string>(10, '9');
// p5 points to an int that is value initialized (§ 3.3.1 (p. 98)) to 0
shared ptr<int> p5 = make shared<int>();
// p6 points to a dynamically allocated, empty vector<string>
auto p6 = make_shared<vector<string>>();
```

Exemplo

```cpp
#include <iostream>
#include <memory>

struct MyStruct {
  std::string nome;
};

// Função que recebe um shared_ptr por referência
    void processSharedPtr(std::shared_ptr<MyStruct>& ptr) {
    ptr->nome = "Fulano de Tal"; // Usa o shared_ptr como
    normalmente faria
}

int main() {
    // Cria um shared_ptr para um objeto MyClass
    std::shared_ptr<MyStruct> ptr = std::make_shared<MyStruct>();

    // Passa o shared_ptr por referência para a função
    processSharedPtr(ptr);

    // ptr ainda é válido aqui, pois foi passado por referência
    std::cout << ptr->nome;

    // ptr será destruído automaticamente ao sair do escopo
    return 0;
}
```

Copiando e atribuindo **`shared_ptr`**

- Quando copiamos ou atribuimos um ponteiro desta classe, cada **`shared_ptr`** mantém registro de quantos outros ponteiros apontam para o mesmo objeto

```
auto p = make_shared<int>(42); //  object to which p points has one user
auto q(p); //  p and q point to the same object
           //  object to which p and q point has two users
```

- Quando um contador deste ponteiro vai a zero, automaticamente libera o objeto gerenciado (invoca **destructor**).

```
auto r = make_shared<int>(42); //  int to which r points has one user
r = q;   //  assign to r, making it point to a different address
         //  increase the use count for the object to which q points
         //  reduce the use count of the object to which r had pointed
         //  the object r had pointed to has no users; that object is automatically freed
```

```cpp
#include <iostream>
#include <memory>

int main() {
    // Cria um shared_ptr apontando para um objeto MyClass
    auto ptr1 = std::make_shared<int>();
    std::cout << "Reference count after ptr1 creation: " << ptr1.use_count() << std::endl;

    // Cria outro shared_ptr apontando para o mesmo objeto
    auto ptr2 = ptr1;
    std::cout << "Reference count after ptr2 creation: " << ptr1.use_count() << std::endl;

    // Cria mais um shared_ptr apontando para o mesmo objeto
    auto ptr3(ptr1);
    std::cout << "Reference count after ptr3 creation: " << ptr1.use_count() << std::endl;

    // Destroi ptr2 e ptr3
    ptr2.reset();
    std::cout << "Reference count after ptr2 reset: " << ptr1.use_count() << std::endl;
    ptr3.reset();
    std::cout << "Reference count after ptr3 reset: " << ptr1.use_count() << std::endl;

    // Destroi ptr1
    ptr1.reset();
    std::cout << "Reference count after ptr1 reset: " << ptr1.use_count() << std::endl;

}
```

Exemplos

```
// factory returns a shared_ptr pointing to a dynamically allocated object
shared_ptr<Foo> factory(T arg)
{
    // process arg as appropriate
    // shared_ptr will take care of deleting this memory
    return make_shared<Foo>(arg);
}
```

```
void use_factory(T arg)
{
    shared_ptr<Foo> p = factory(arg);
    // use p
} // p goes out of scope; the memory to which p points is automatically freed
```

```
shared_ptr<Foo> use_factory(T arg)
{
    shared_ptr<Foo> p = factory(arg);
    // use p
    return p;   // reference count is incremented when we return p
} // p goes out of scope; the memory to which p points is not freed
```

Operações específicas para **shared_ptr**

Table 12.2: Operations Specific to shared_ptr

`make_shared<T>(args)`	Returns a `shared_ptr` pointing to a dynamically allocated object of type `T`. Uses *args* to initialize that object.
`shared_ptr<T> p(q)`	`p` is a copy of the `shared_ptr` `q`; increments the count in `q`. The pointer in `q` must be convertible to `T*` (§ 4.11.2, p. 161).
`p = q`	`p` and `q` are `shared_ptr`s holding pointers that can be converted to one another. Decrements `p`'s reference count and increments `q`'s count; deletes `p`'s existing memory if `p`'s count goes to 0.
`p.unique()`	Returns `true` if `p.use_count()` is one; `false` otherwise.
`p.use_count()`	Returns the number of objects sharing with `p`; may be a slow operation, intended primarily for debugging purposes.

`shared_ptr` e `new`

- Como visto, se não inicializarmos um ponteiro smart, ele será inicializado com um ponteiro nulo
- Podemos também inicializar um ponteiro smart por um ponteiro retornado por **`new`**, com algumas exceções

```
shared_ptr<double> p1; //  shared_ptr that can point at a double
shared_ptr<int> p2(new int(42)); //  p2 points to an int with value 42

shared_ptr<int> p1 = new int(1024); //  error: must use direct initialization
shared_ptr<int> p2(new int(1024));  //  ok: uses direct initialization
```

shared_ptr e new

- Algumas exceções na construção devem ser consideradas

```
shared_ptr<int> clone(int p) {
    return new int(p); //  error: implicit conversion to shared_ptr<int>
}
```

```
shared_ptr<int> clone(int p) {
    //  ok: explicitly create a shared_ptr<int> from int*
    return shared_ptr<int>(new int(p));
}
```

`shared_ptr` e `new`

- Por padrão, um ponteiro utilizado para inicializar um ponteiro smart deve apontar para memória dinâmica
- Pelo fato de que, por padrão, ponteiros smart usam **`delete`** para liberar o objeto apontado

A classe **`unique_ptr`**

- Ponteiros que são "donos" do objeto para o qual apontam
  - Somente um **`unique_ptr`** pode apontar para um objeto por vez
- O objeto é destruído quando **`unique_ptr`** é destruído
- Deve-se utilizar a forma direta de inicialização:

```
unique_ptr<double> p1; //  unique_ptr that can point at a double
unique_ptr<int> p2(new int(42)); //  p2 points to int with value 42
```

- Não suporta cópia ou atribuições:

```
unique_ptr<string> p1(new string("Stegosaurus"));
unique_ptr<string> p2(p1);   //  error: no copy for unique_ptr
unique_ptr<string> p3;
p3 = p2;                     //  error: no assign for unique_ptr
```

A classe **`unique_ptr`**

- Embora não haja atribuições, pode ser feita transferência de "propriedade" de um ponteiro **`unique_ptr`** (não `const`) para outro
- Utiliza-se `release` / `reset ou move`

```
//  transfers ownership from p1 (which points to the string Stegosaurus) to p2
unique_ptr<string> p2(p1.release()); //  release makes p1 null
unique_ptr<string> p3(new string("Trex"));
//  transfers ownership from p3 to p2
p2.reset(p3.release()); //  reset deletes the memory to which p2 had pointed

p2.release(); //  WRONG: p2 won't free the memory and we've lost the pointer
auto p = p2.release(); //  ok, but we must remember to delete(p)
```

```cpp
// Função que aceita um unique_ptr por valor,
transferindo a propriedade
void processUniquePtr(std::unique_ptr<MyStruct> ptr) {
    ptr->nome = "Beltrano"; // Usa o unique_ptr como
    normalmente faria
    // O unique_ptr será destruído ao sair do escopo,
    liberando a memória
}

int main() {
    // Cria um unique_ptr para um objeto MyClass
    std::unique_ptr<MyStruct> ptr1 =
    std::make_unique<MyStruct>();

    // Acessa o objeto através do unique_ptr
    ptr1->nome = "Fulano";

    // Transfere a propriedade para a função
    processUniquePtr usando std::move
    processUniquePtr(std::move(ptr1));

    // ptr1 não é mais válido aqui, pois a propriedade
    foi transferida
    if (ptr1 == nullptr) {
        std::cout << "ptr1 is now nullptr after
        transfer of ownership." << std::endl;
    }

    // Cria outro unique_ptr para um objeto MyClass
    std::unique_ptr<MyStruct> ptr2 =
    std::make_unique<MyStruct>();

    // Transfere a propriedade para outro unique_ptr
    usando std::release
    std::unique_ptr<MyStruct> ptr3(ptr2.release());

    // ptr2 não é mais válido aqui
    if (ptr2 == nullptr) {
        std::cout << "ptr2 is now nullptr after
        transfer of ownership to ptr3." << std::endl;
    }


    auto ptr4 = std::make_unique<MyStruct>();

    //Transferência de propriedade usando reset e
    release
    ptr4.reset(ptr3.release());

    return 0;
}
```

A classe **`unique_ptr`**

- Existe uma exceção à regra de não cópia ou atribuição:
  - Pode ser copiado ou atribuído unique_ptr prestes a ser destruído
  - Exemplo mais comum: quando retornamos de uma função:

```
unique_ptr<int> clone(int p) {
    //  ok: explicitly create a unique_ptr<int> from int*
    return unique_ptr<int>(new int(p));
}
```

```
unique_ptr<int> clone(int p) {
    unique_ptr<int> ret(new int (p));
    // ...
    return ret;
}
```

Operações comuns para **`shared_ptr`** e **`unique_ptr`**

Table 12.1: Operations Common to `shared_ptr` and `unique_ptr`

`shared_ptr<T> sp` `unique_ptr<T> up`	Null smart pointer that can point to objects of type `T`.
`p`	Use `p` as a condition; `true` if `p` points to an object.
`*p`	Dereference `p` to get the object to which `p` points.
`p->mem`	Synonym for `(*p).mem`.
`p.get()`	Returns the pointer in `p`. Use with caution; the object to which the returned pointer points will disappear when the smart pointer deletes it.
`swap(p, q)` `p.swap(q)`	Swaps the pointers in `p` and `q`.

A classe **`weak_ptr`**

- Ponteiros que não controlam o tempo de vida do objeto para o qual apontam
- Apontam para objetos gerenciados por **`shared_ptr`** sem alterar a contagem de referência
- Até que o último **`shared_ptr`** que aponte para um objeto exista, o objeto será deletedo
  - Mesmo se existirem **`weak_ptr`**'s apontando para ele

```
auto p = make_shared<int>(42);
weak_ptr<int> wp(p);   //   wp weakly shares with p; use count in p is unchanged
```

A classe **`weak_ptr`**

- Não permitem acessar o objeto diretamente
- Requer uso de `lock` para checar se objeto ainda existe

```
if (shared_ptr<int> np = wp.lock()) { //  true if np is not null
     //  inside the if, np shares its object with p
}
```

Resumindo

- `unique_ptr`
  - Quando você quiser que o tempo de vida do objeto dure enquanto exista uma única referência para ele
  - Exemplo:
    - Use-o quando alocar uma memória ao entrar num contexto e que não será necessária ao sair desse contexto
- `shared_ptr`
  - Quando você quiser referenciar seu objeto de múltiplos lugares e não quer que seja desalocado enquanto estas referências existirem
- `weak_ptr`
  - Quando você quer referenciar um objeto já existente de maneira temporária (por exemplo, checar se ele existe)