HEAP AND COPY CONSTRUCTOR

Lecturer：陈笑沙

TABLE OF CONTENTS

6.1 ABOUT THE HEAP

The program resides in memory to run, with its memory layout consisting of four regions:

6.1 ABOUT THE HEAP

The program resides in memory to run, with its memory layout consisting of four regions:

- Code region

6.1 ABOUT THE HEAP

The program resides in memory to run, with its memory layout consisting of four regions:

- Code region
- Data region (global data area)

6.1 ABOUT THE HEAP

The program resides in memory to run, with its memory layout consisting of four regions:

- Code region
- Data region (global data area)
- Heap region

6.1 ABOUT THE HEAP

The program resides in memory to run, with its memory layout consisting of four regions:

- Code region
- Data region (global data area)
- Heap region
- Stack region

6.1 ABOUT THE HEAP

6.1 ABOUT THE HEAP

- **Code area** stores executable instructions

6.1 ABOUT THE HEAP

- **Code area** stores executable instructions
  - Read-only, possibly shared by multiple processes (such as dynamic libraries), fixed size, loaded at runtime

6.1 ABOUT THE HEAP

- **Code area** stores executable instructions
  - Read-only, possibly shared by multiple processes (such as dynamic libraries), fixed size, loaded at runtime
- **Global data area** saves global variables and static variables, constant strings

6.1 ABOUT THE HEAP

- **Code area** stores executable instructions
  - Read-only, possibly shared by multiple processes (such as dynamic libraries), fixed size, loaded at runtime
- **Global data area** saves global variables and static variables, constant strings
  - Read and write, lifecycle consistent with the program

6.1 ABOUT THE HEAP

- **Code area** stores executable instructions
  - Read-only, possibly shared by multiple processes (such as dynamic libraries), fixed size, loaded at runtime
- **Global data area** saves global variables and static variables, constant strings
  - Read and write, lifecycle consistent with the program
- **Heap area** dynamic memory allocation and storage area

6.1 ABOUT THE HEAP

- **Code area** stores executable instructions
  - Read-only, possibly shared by multiple processes (such as dynamic libraries), fixed size, loaded at runtime
- **Global data area** saves global variables and static variables, constant strings
  - Read and write, lifecycle consistent with the program
- **Heap area** dynamic memory allocation and storage area
  - Manual management, address from low to high, variable size, dynamically allocated at runtime, fragmentation problems may be relatively severe

6.1 ABOUT THE HEAP

- **Code area** stores executable instructions
  - Read-only, possibly shared by multiple processes (such as dynamic libraries), fixed size, loaded at runtime
- **Global data area** saves global variables and static variables, constant strings
  - Read and write, lifecycle consistent with the program
- **Heap area** dynamic memory allocation and storage area
  - Manual management, address from low to high, variable size, dynamically allocated at runtime, fragmentation problems may be relatively severe
- **Stack area** stores local variables, function parameters, return addresses, and saves function call context (such as register status)

6.1 ABOUT THE HEAP

- **Code area** stores executable instructions
  - Read-only, possibly shared by multiple processes (such as dynamic libraries), fixed size, loaded at runtime
- **Global data area** saves global variables and static variables, constant strings
  - Read and write, lifecycle consistent with the program
- **Heap area** dynamic memory allocation and storage area
  - Manual management, address from low to high, variable size, dynamically allocated at runtime, fragmentation problems may be relatively severe
- **Stack area** stores local variables, function parameters, return addresses, and saves function call context (such as register status)
  - Automatic management, address from high to low, limited size, fast access

6.1 ABOUT THE HEAP

C language heap allocation and release

```c
#include <stdlib.h>
#include <stdio.h>

int num = 10; // Data area

int main()
{
    int* arr = (int*)malloc(sizeof(int) * num); // Heap area application
    for (int i = 0; i < num; i++) {
        arr[i] = i; // arr and i are stack variables
        printf("%d\n", arr[i]);
    }

    free(arr); // Release
    return 0;
```

You can view the assembly code on https://godbolt.org to intuitively feel the underlying principles (it is best to clear the Compiler options option)

6.1 ABOUT THE HEAP

C language heap allocation and release

```c
#include <stdlib.h>
#include <stdio.h>

int num = 10; // Data area

int main()
{
    int* arr = (int*)malloc(sizeof(int) * num); // Heap area application
    for (int i = 0; i < num; i++) {
        arr[i] = i; // arr and i are stack variables
        printf("%d\n", arr[i]);
    }

    free(arr); // Release
    return 0;
```

You can view the assembly code on https://godbolt.org to intuitively feel the underlying principles (it is best to clear the Compiler options option)

6.1 ABOUT THE HEAP

C language heap allocation and release

```c
#include <stdlib.h>
#include <stdio.h>

int num = 10; // Data area

int main()
{
    int* arr = (int*)malloc(sizeof(int) * num); // Heap area application
    for (int i = 0; i < num; i++) {
        arr[i] = i; // arr and i are stack variables
        printf("%d\n", arr[i]);
    }

    free(arr); // Release
    return 0;
```

You can view the assembly code on https://godbolt.org to intuitively feel the underlying principles (it is best to clear the Compiler options option)

6.1 ABOUT THE HEAP

C language heap allocation and release

```c
#include <stdio.h>

int num = 10; // Data area

int main()
{
    int* arr = (int*)malloc(sizeof(int) * num); // Heap area application
    for (int i = 0; i < num; i++) {
        arr[i] = i; // arr and i are stack variables
        printf("%d\n", arr[i]);
    }

    free(arr); // Release
    return 0;
}
```

You can view the assembly code on https://godbolt.org to intuitively feel the underlying principles (it is best to clear the Compiler options option)

6.1 ABOUT THE HEAP

C language heap allocation and release

```c
#include <stdio.h>

int num = 10; // Data area

int main()
{
    int* arr = (int*)malloc(sizeof(int) * num); // Heap area application
    for (int i = 0; i < num; i++) {
        arr[i] = i; // arr and i are stack variables
        printf("%d\n", arr[i]);
    }

    free(arr); // Release
    return 0;
}
```

You can view the assembly code on https://godbolt.org to intuitively feel the underlying principles (it is best to clear the Compiler options option)

6.2 THE NEED FOR new AND delete

C++ contains many object data, and when applying for the heap, you also need to initialize the memory (constructor); when releasing memory, you need to call the destructor.

```
1  class Point {
2  public:
3      Point() { x = 1; y = 1; } // No parameter constructor
4      void print() { cout << x << ", " << y; }
5      void set(int x, int y) {
6          this->x = x;
7          this->y = y;
8      }
9      ~Point() { // do something ... }
10 private:
11     int x, y;
12 };
```

6.2 THE NEED FOR new AND delete

C++ contains many object data, and when applying for the heap, you also need to initialize the memory (constructor); when releasing memory, you need to call the destructor.

```
1  class Point {
2  public:
3      Point() { x = 1; y = 1; } // No parameter constructor
4      void print() { cout << x << ", " << y; }
5      void set(int x, int y) {
6          this->x = x;
7          this->y = y;
8      }
9      ~Point() { // do something ... }
10 private:
11     int x, y;
12 };
```

6.2 THE NEED FOR new AND delete

The shortcomings of malloc and free.

```
1  int main() {
2      // No constructor called
3      Point* pd = (Point*)malloc(10 * sizeof(Point));
4      for (int i = 0; i < 10; i++)
5          pd[i].set(1, 1); // Need to initialize like this
6      free(pd); // No destructor called
7      return 0;
8  }
```

6.2 THE NEED FOR new AND delete

The shortcomings of malloc and free.

```
1  int main() {
2      // No constructor called
3      Point* pd = (Point*)malloc(10 * sizeof(Point));
4      for (int i = 0; i < 10; i++)
5          pd[i].set(1, 1); // Need to initialize like this
6      free(pd); // No destructor called
7      return 0;
8  }
```

6.2 THE NEED FOR new AND delete

The shortcomings of malloc and free.

```
1  int main() {
2      // No constructor called
3      Point* pd = (Point*)malloc(10 * sizeof(Point));
4      for (int i = 0; i < 10; i++)
5          pd[i].set(1, 1); // Need to initialize like this
6      free(pd); // No destructor called
7      return 0;
8  }
```

6.2 THE NEED FOR new AND delete

The shortcomings of malloc and free.

```
1  int main() {
2      // No constructor called
3      Point* pd = (Point*)malloc(10 * sizeof(Point));
4      for (int i = 0; i < 10; i++)
5          pd[i].set(1, 1); // Need to initialize like this
6      free(pd); // No destructor called
7      return 0;
8  }
```

6.3 ALLOCATING HEAP OBJECTS

```
1  int main() {
2      Point* pd = new Point; // Allocate a single object
3      Point* pds = new Point[10]; // Allocate an array
4      pd->print(); // Already initialized, no need to set again
5      pds[2].print();
6
7      delete pd; // Release a single object
8      delete[] pds; // Release multiple objects
9
10     return 0;
11 }
```

6.3 ALLOCATING HEAP OBJECTS

```
1  int main() {
2      Point* pd = new Point; // Allocate a single object
3      Point* pds = new Point[10]; // Allocate an array
4      pd->print(); // Already initialized, no need to set again
5      pds[2].print();
6
7      delete pd; // Release a single object
8      delete[] pds; // Release multiple objects
9
10     return 0;
11 }
```

6.3 ALLOCATING HEAP OBJECTS

```
1  int main() {
2      Point* pd = new Point; // Allocate a single object
3      Point* pds = new Point[10]; // Allocate an array
4      pd->print(); // Already initialized, no need to set again
5      pds[2].print();
6
7      delete pd; // Release a single object
8      delete[] pds; // Release multiple objects
9
10     return 0;
11 }
```

6.3 ALLOCATING HEAP OBJECTS

```
1  int main() {
2      Point* pd = new Point; // Allocate a single object
3      Point* pds = new Point[10]; // Allocate an array
4      pd->print(); // Already initialized, no need to set again
5      pds[2].print();
6
7      delete pd; // Release a single object
8      delete[] pds; // Release multiple objects
9
10     return 0;
11 }
```

6.3 ALLOCATING HEAP OBJECTS

```
1  int main() {
2      Point* pd = new Point; // Allocate a single object
3      Point* pds = new Point[10]; // Allocate an array
4      pd->print(); // Already initialized, no need to set again
5      pds[2].print();
6
7      delete pd; // Release a single object
8      delete[] pds; // Release multiple objects
9
10     return 0;
11 }
```

6.3 ALLOCATING HEAP OBJECTS

```
1  int main() {
2      Point* pd = new Point; // Allocate a single object
3      Point* pds = new Point[10]; // Allocate an array
4      pd->print(); // Already initialized, no need to set again
5      pds[2].print();
6
7      delete pd; // Release a single object
8      delete[] pds; // Release multiple objects
9
10     return 0;
11 }
```

6.3 ALLOCATING HEAP OBJECTS

```
1  int main() {
2      Point* pd = new Point; // Allocate a single object
3      Point* pds = new Point[10]; // Allocate an array
4      pd->print(); // Already initialized, no need to set again
5      pds[2].print();
6
7      delete pd; // Release a single object
8      delete[] pds; // Release multiple objects
9
10     return 0;
11 }
```

6.3 ALLOCATING HEAP OBJECTS

```
1  int main() {
2      Point* pd = new Point; // Allocate a single object
3      Point* pds = new Point[10]; // Allocate an array
4      pd->print(); // Already initialized, no need to set again
5      pds[2].print();
6
7      delete pd; // Release a single object
8      delete[] pds; // Release multiple objects
9
10     return 0;
11 }
```

6.3 ALLOCATING HEAP OBJECTS

6.3 ALLOCATING HEAP OBJECTS

- new and delete are C++ operators, no need to introduce new header files

6.3 ALLOCATING HEAP OBJECTS

- new and delete are C++ operators, no need to introduce new header files
- Its functionality covers C language, can apply for basic data types

6.3 ALLOCATING HEAP OBJECTS

- new and delete are C++ operators, no need to introduce new header files
- Its functionality covers C language, can apply for basic data types
- Performance is the same as C language, automatic type matching, no need to convert

6.3 ALLOCATING HEAP OBJECTS

You can pass constructor parameters when using new:

```
1  class Point {
2  public:
3      Point(int x, int y): x(x), y(y){}
4      void print() { cout << x << ", " << y; }
5      void set(int x, int y) {
6          this->x = x;
7          this->y = y;
8      }
9  };
10
11 int main() {
12     Point* p = new Point(3, 4);
13     p->print();
14     delete p; // cannot have parameters
15     return 0;
```

6.3 ALLOCATING HEAP OBJECTS

You can pass constructor parameters when using new:

```
1  class Point {
2  public:
3      Point(int x, int y): x(x), y(y){}
4      void print() { cout << x << ", " << y; }
5      void set(int x, int y) {
6          this->x = x;
7          this->y = y;
8      }
9  };
10
11 int main() {
12     Point* p = new Point(3, 4);
13     p->print();
14     delete p; // cannot have parameters
15     return 0;
```

6.3 ALLOCATING HEAP OBJECTS

You can pass constructor parameters when using new:

```
2  public:
3      Point(int x, int y): x(x), y(y){}
4      void print() { cout << x << ", " << y; }
5      void set(int x, int y) {
6          this->x = x;
7          this->y = y;
8      }
9  };
10
11 int main() {
12     Point* p = new Point(3, 4);
13     p->print();
14     delete p; // cannot have parameters
15     return 0;
16 }
```

6.3 ALLOCATING HEAP OBJECTS

You can pass constructor parameters when using new:

```
2  public:
3      Point(int x, int y): x(x), y(y){}
4      void print() { cout << x << ", " << y; }
5      void set(int x, int y) {
6          this->x = x;
7          this->y = y;
8      }
9  };
10
11 int main() {
12     Point* p = new Point(3, 4);
13     p->print();
14     delete p; // cannot have parameters
15     return 0;
16 }
```

6.3 ALLOCATING HEAP OBJECTS

You can pass constructor parameters when using new:

```
2  public:
3      Point(int x, int y): x(x), y(y){}
4      void print() { cout << x << ", " << y; }
5      void set(int x, int y) {
6          this->x = x;
7          this->y = y;
8      }
9  };
10
11 int main() {
12     Point* p = new Point(3, 4);
13     p->print();
14     delete p; // cannot have parameters
15     return 0;
16 }
```

6.3 ALLOCATING HEAP OBJECTS

You can pass constructor parameters when using new:

```
1  class Point {
2  public:
3      Point(int x, int y): x(x), y(y){}
4      void print() { cout << x << ", " << y; }
5      void set(int x, int y) {
6          this->x = x;
7          this->y = y;
8      }
9  };
10
11 int main() {
12     Point* p = new Point(3, 4);
13     p->print();
14     delete p; // cannot have parameters
15     return 0;
```

6.4 COPY CONSTRUCTOR

Common operations on objects:

6.4 COPY CONSTRUCTOR

Common operations on objects:

- Creation (constructor)

6.4 COPY CONSTRUCTOR

Common operations on objects:

- Creation (constructor)
- Comparison

6.4 COPY CONSTRUCTOR

Common operations on objects:

- Creation (constructor)
- Comparison
- Assignment

6.4 COPY CONSTRUCTOR

Common operations on objects:

- Creation (constructor)
- Comparison
- Assignment
- Copy constructor

6.4 COPY CONSTRUCTOR

Common operations on objects:

- Creation (constructor)
- Comparison
- Assignment
- Copy constructor
- Destruction (destructor)

6.4 COPY CONSTRUCTOR

BEYOND THE TOPIC

In essence, not all objects support these operations, and there is no strict limitation in C++ on which objects need to support which operations. After C++ 20, the concept of concept has been introduced to address this issue.

6.4 COPY CONSTRUCTOR

When a class does not customize these operations, the compiler will automatically add a default operation.

```
class Student {
public:
    Student(); // Default constructor
    Student(const Student& other); // Default copy constructor
    ~Student(); // Default destructor
    Student& operator=(const Student& other); // Default assignment function
    bool operator==(const Student& other) const; // Default comparison function
};
```

6.4 COPY CONSTRUCTOR

We can customize copy constructors:

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Student {
6  public:
7    Student(const string &name, int age)
8      : name(name), age(age) {
9      cout << "Constructing Student " << name << endl;
10   }
11   Student(const Student &other) {
12     cout << "Constructing copy of Student "
13          << other.name << endl;
14     name = other.name;
15     age = other.age;
```

6.4 COPY CONSTRUCTOR

We can customize copy constructors:

```
#include <iostream>

using namespace std;

class Student {
public:
  Student(const string &name, int age)
    : name(name), age(age) {
    cout << "Constructing Student " << name << endl;
  }
  Student(const Student &other) {
    cout << "Constructing copy of Student "
         << other.name << endl;
    name = other.name;
    age = other.age;
    age = other.age;
```

6.4 COPY CONSTRUCTOR

We can customize copy constructors:

```
 7   Student(const string &name, int age)
 8     : name(name), age(age) {
 9     cout << "Constructing Student " << name << endl;
10   }
11   Student(const Student &other) {
12     cout << "Constructing copy of Student "
13          << other.name << endl;
14     name = other.name;
15     age = other.age;
16   }
17   ~Student() {
18     cout << "Destructing Student " << name << endl;
19   }
20   void setName(const string &name) { this->name = name; }
```

6.4 COPY CONSTRUCTOR

We can customize copy constructors:

```
11    Student(const Student &other) {
12      cout << "Constructing copy of Student "
13           << other.name << endl;
14      name = other.name;
15      age = other.age;
16    }
17    ~Student() {
18      cout << "Destructing Student " << name << endl;
19    }
20    void setName(const string &name) { this->name = name; }
21
22  private:
23    int age;
24    string name;
25  };
```

6.4 COPY CONSTRUCTOR

We can customize copy constructors:

```
20    void setName(const string &name) { this->name = name; }
21
22  private:
23    int age;
24    string name;
25  };
26
27  int main() {
28    Student s{"Eric", 12};
29    Student s2 = s;
30    Student s3{"Alice", 20};
31    s3 = s; // this will not call copy constructor
32    s2.setName("Bob");
33    return 0;
34  }
19      age = other.age;
```

6.4 COPY CONSTRUCTOR

We can customize copy constructors:

```
20    void setName(const string &name) { this->name = name; }
21
22  private:
23    int age;
24    string name;
25  };
26
27  int main() {
28    Student s{"Eric", 12};
29    Student s2 = s;
30    Student s3{"Alice", 20};
31    s3 = s; // this will not call copy constructor
32    s2.setName("Bob");
33    return 0;
34  }
19      age = other.age;
```

6.4 COPY CONSTRUCTOR

6.4 COPY CONSTRUCTOR

- If the same class object's reference is used as a parameter, then this constructor is called a copy constructor.

6.4 COPY CONSTRUCTOR

- If the same class object's reference is used as a parameter, then this constructor is called a copy constructor.
- Features of the copy constructor

6.4 COPY CONSTRUCTOR

- If the same class object's reference is used as a parameter, then this constructor is called a copy constructor.
- Features of the copy constructor
  - It is a constructor, which will be automatically called by the system when creating an object.

6.4 COPY CONSTRUCTOR

- If the same class object's reference is used as a parameter, then this constructor is called a copy constructor.
- Features of the copy constructor
  - It is a constructor, which will be automatically called by the system when creating an object.
  - It takes an already created object as a parameter and assigns the data members of the object to the new object as needed.

6.4 COPY CONSTRUCTOR

6.4 COPY CONSTRUCTOR

- If no copy constructor is defined, the compiler will generate a default copy constructor for the class.

6.4 COPY CONSTRUCTOR

- If no copy constructor is defined, the compiler will generate a default copy constructor for the class.
- The default copy constructor uses bitwise copying to complete the copying of objects to objects.

6.5 SHALLOW COPY VS. DEEP COPY

Consider the following situation, what is the problem?

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Array {
6  public:
7    explicit Array(int size) {
8      this->size = size;
9      arr = new int[size];
10   }
11   Array(const Array &other) {
12     this->size = other.size;
13     this->arr = other.arr;
14   }
15   void set(int i, int value) { arr[i] = value; }
```

6.5 SHALLOW COPY VS. DEEP COPY

Consider the following situation, what is the problem?

```
19      for (int i = 0; i < size - 1; i++)
20        cout << arr[i] << ", ";
21      (size>0 ? cout<<arr[size - 1] : cout) << "]" << endl;
22    }
23    ~Array() { delete[] arr; }
24
25  private:
26    int *arr;
27    int size;
28  };
29
30  int main() {
31    Array a{10};
32    Array b = a;
33    a.set(2, 100);
```

6.5 SHALLOW COPY VS. DEEP COPY

Consider the following situation, what is the problem?

```
#include <iostream>

using namespace std;

class Array {
public:
  explicit Array(int size) {
    this->size = size;
    arr = new int[size];
  }
  Array(const Array &other) {
    this->size = other.size;
    this->arr = other.arr;
  }
  void set(int i, int value) { arr[i] = value; }
  void set(int i, int value) { arr[i][] = value; }
```

6.5 SHALLOW COPY VS. DEEP COPY

Consider the following situation, what is the problem?

```
 6  public:
 7   explicit Array(int size) {
 8     this->size = size;
 9     arr = new int[size];
10   }
11   Array(const Array &other) {
12     this->size = other.size;
13     this->arr = other.arr;
14   }
15   void set(int i, int value) { arr[i] = value; }
16   int get(int i) const { return arr[i]; }
17   void print() {
18     cout << "[";
19     for (int i = 0; i < size - 1; i++)
```

6.5 SHALLOW COPY VS. DEEP COPY

Consider the following situation, what is the problem?

```
 9     arr = new int[size];
10   }
11   Array(const Array &other) {
12     this->size = other.size;
13     this->arr = other.arr;
14   }
15   void set(int i, int value) { arr[i] = value; }
16   int get(int i) const { return arr[i]; }
17   void print() {
18     cout << "[";
19     for (int i = 0; i < size - 1; i++)
20       cout << arr[i] << ", ";
21     (size>0 ? cout<<arr[size - 1] : cout) << "]" << endl;
22   }
```

6.5 SHALLOW COPY VS. DEEP COPY

Consider the following situation, what is the problem?

```
13      this->arr = other.arr;
14    }
15    void set(int i, int value) { arr[i] = value; }
16    int get(int i) const { return arr[i]; }
17    void print() {
18      cout << "[";
19      for (int i = 0; i < size - 1; i++)
20        cout << arr[i] << ", ";
21      (size>0 ? cout<<arr[size - 1] : cout) << "]" << endl;
22    }
23    ~Array() { delete[] arr; }
24
25  private:
26    int *arr;
```

6.5 SHALLOW COPY VS. DEEP COPY

Consider the following situation, what is the problem?

```
23    ~Array() { delete[] arr; }
24
25  private:
26    int *arr;
27    int size;
28  };
29
30  int main() {
31    Array a{10};
32    Array b = a;
33    a.set(2, 100);
34    a.print();
35    b.print();
36    return 0;
37  }
    void set(int i, int value) { arr[i] = value; }
```

6.5 SHALLOW COPY VS. DEEP COPY

A change in one object also affects another object because they share the same memory space.

```
#include <iostream>

using namespace std;

class Array {
public:
  explicit Array(int size) {
    this->size = size;
    arr = new int[size];
  }
  Array(const Array &other) {
    this->size = other.size;
    this->arr = other.arr;
  }
```

```
15    void set(int i, int value) { arr[i] = value; }
```

6.5 SHALLOW COPY VS. DEEP COPY

A change in one object also affects another object because they share the same memory space.

```
 6  public:
 7   explicit Array(int size) {
 8     this->size = size;
 9     arr = new int[size];
10   }
11   Array(const Array &other) {
12     this->size = other.size;
13     this->arr = other.arr;
14   }
```

```
15    void set(int i, int value) { arr[i] = value; }
```

6.5 SHALLOW COPY VS. DEEP COPY

How to modify?

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Array {
6  public:
7    explicit Array(int size) {
8      this->size = size;
9      arr = new int[size];
10   }
11   Array(const Array &other) {
12     this->size = other.size;
13     this->arr = other.arr;
14   }
15   void set(int i, int value) { arr[i] = value; }
```

6.5 SHALLOW COPY VS. DEEP COPY

How to modify?

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Array {
6  public:
7   explicit Array(int size) {
8     this->size = size;
9     arr = new int[size];
10   }
11   Array(const Array &other) {
12     this->size = other.size;
13     this->arr = new int[this->size];
14     for (int i = 0; i < this->size; i++)
15        this->arr[i] = other.arr[i];
```

6.6 TEMPORARY OBJECTS

Consider following situation (example/lec06/tempObject):

```
1  #include <iostream>
2
3  using namespace std;
4
5  class A {
6  public:
7    A(string id) : id(id) {
8      cout << "Constructing A with id " << id << endl;
9    }
10   A(const A &other) : id(other.id) {
11     cout << "Copy " << other.id << endl;
12   }
13
14 private:
15   string id;
```

6.6 TEMPORARY OBJECTS

Consider following situation (example/lec06/tempObject):

```
4
5  class A {
6  public:
7    A(string id) : id(id) {
8      cout << "Constructing A with id " << id << endl;
9    }
10   A(const A &other) : id(other.id) {
11     cout << "Copy " << other.id << endl;
12   }
13
14 private:
15   string id;
16 };
17
```

6.6 TEMPORARY OBJECTS

Consider following situation (example/lec06/tempObject):

```
14  private:
15    string id;
16  };
17
18  A returnValueFunc(A a) {
19    A b = a;
20    // If there is no return value optimization,
21    // a copy will occur here
22    return b;
23  }
24
25  int main() {
26    A a("a1");
27    A b = a;
28    string id;
```

6.6 TEMPORARY OBJECTS

Consider following situation (example/lec06/tempObject):

```
16 };
17
18 A returnValueFunc(A a) {
19   A b = a;
20   // If there is no return value optimization,
21   // a copy will occur here
22   return b;
23 }
24
25 int main() {
26   A a("a1");
27   A b = a;
28   A c = returnValueFunc(b);
29   return 0;
30 }
   string id;
```

6.6 TEMPORARY OBJECTS

Consider following situation (example/lec06/tempObject):

```
16 };
17
18 A returnValueFunc(A a) {
19   A b = a;
20   // If there is no return value optimization,
21   // a copy will occur here
22   return b;
23 }
24
25 int main() {
26   A a("a1");
27   A b = a;
28   A c = returnValueFunc(b);
29   return 0;
30 }
   string id;
```

6.6 TEMPORARY OBJECTS

Consider following situation (example/lec06/tempObject):

```
16 };
17
18 A returnValueFunc(A a) {
19   A b = a;
20   // If there is no return value optimization,
21   // a copy will occur here
22   return b;
23 }
24
25 int main() {
26   A a("a1");
27   A b = a;
28   A c = returnValueFunc(b);
29   return 0;
30 }
   string id;
```

6.6 TEMPORARY OBJECTS

```
1  #include <iostream>
2
3  using namespace std;
4
5  class A {
6  public:
7    A(string id) : id(id) {
8      cout << "Constructing A with id " << id << endl;
9    }
10   A(const A &other) : id(other.id) {
11     cout << "Copy " << other.id << endl;
12   }
13
14 private:
15   string id;
```

6.6 TEMPORARY OBJECTS

How many copies occurred?

```
1  #include <iostream>
2
3  using namespace std;
4
5  class A {
6  public:
7    A(string id) : id(id) {
8      cout << "Constructing A with id " << id << endl;
9    }
10   A(const A &other) : id(other.id) {
11     cout << "Copy " << other.id << endl;
12   }
13
14 private:
15   string id;
```

6.6 TEMPORARY OBJECTS

How many copies occurred?

```
14  private:
15    string id;
16  };
17
18  A returnValueFunc(A a) {
19    A b = a;
20    // If there is no return value optimization,
21    // a copy will occur here
22    return b;
23  }
24
25  int main() {
26    A a("a1");
27    A b = a;
29    A ret=returnValueFunc(b);
```

6.6 TEMPORARY OBJECTS

6.6 TEMPORARY OBJECTS

- In C++, if function parameters are passed by value, a copy will occur

6.6 TEMPORARY OBJECTS

- In C++, if function parameters are passed by value, a copy will occur
- If function parameters are returned by value, it will be considered a temporary object and optimized

6.6 TEMPORARY OBJECTS

- In C++, if function parameters are passed by value, a copy will occur
- If function parameters are returned by value, it will be considered a temporary object and optimized
- You can add the -fno-elide-constructors parameter to disable return value optimization

6.6 TEMPORARY OBJECTS

Consider the following situation:

```
A a = A("a");
```

6.6 TEMPORARY OBJECTS

Consider the following situation:

```
A a = A("a");
```

At this point, it seems like an **unnamed object** is created using A("a"), and then the copy constructor is called to initialize a. However, in reality, the constructor will still be called.

6.7 CONSTRUCTORS FOR TYPE CONVERSION

A constructor with only one parameter can perform type conversion

```
1  class Student {
2      Student(const string& name) : name(name) {}
3  private:
4      string name;
5  };
6
7  void f(Student stu) {...}
8
9  // can be called like this
10 f("Trump");
```

6.7 CONSTRUCTORS FOR TYPE CONVERSION

A constructor with only one parameter can perform type conversion

```
1  class Student {
2      Student(const string& name) : name(name) {}
3  private:
4      string name;
5  };
6
7  void f(Student stu) {...}
8
9  // can be called like this
10 f("Trump");
```

6.7 CONSTRUCTORS FOR TYPE CONVERSION

A constructor with only one parameter can perform type conversion

```
1  class Student {
2      Student(const string& name) : name(name) {}
3  private:
4      string name;
5  };
6
7  void f(Student stu) {...}
8
9  // can be called like this
10 f("Trump");
```

6.7 CONSTRUCTORS FOR TYPE CONVERSION

A constructor with only one parameter can perform type conversion

```
1  class Student {
2      Student(const string& name) : name(name) {}
3  private:
4      string name;
5  };
6
7  void f(Student stu) {...}
8
9  // can be called like this
10 f("Trump");
```

6.7 CONSTRUCTORS FOR TYPE CONVERSION

A constructor with only one parameter can perform type conversion

```
1  class Student {
2      Student(const string& name) : name(name) {}
3  private:
4      string name;
5  };
6
7  void f(Student stu) {...}
8
9  // can be called like this
10 f("Trump");
```

6.7 CONSTRUCTORS FOR TYPE CONVERSION

A constructor with only one parameter can perform type conversion

```
1  class Student {
2      Student(const string& name) : name(name) {}
3  private:
4      string name;
5  };
6
7  void f(Student& stu) {...} // cannot do this
8
9  // can be called like this
10 f("Trump");
```

6.7 CONSTRUCTORS FOR TYPE CONVERSION

A constructor with only one parameter can perform type conversion

```
1  class Student {
2      Student(const string& name) : name(name) {}
3  private:
4      string name;
5  };
6
7  void f(const Student& stu) {...} // this is ok
8
9  // can be called like this
10 f("Trump");
```

6.7 CONSTRUCTORS FOR TYPE CONVERSION

A constructor with only one parameter can perform type conversion

```
1  class Student {
2      Student(const string& name) : name(name) {}
3  private:
4      string name;
5  };
6
7  void f(Student&& stu) {...} // this is also ok
8
9  // can be called like this
10 f("Trump");
```

6.7 CONSTRUCTORS FOR TYPE CONVERSION

6.7 CONSTRUCTORS FOR TYPE CONVERSION

Consider the following situation:

6.7 CONSTRUCTORS FOR TYPE CONVERSION

Consider the following situation:

```
1  class A {
2  public:
3    int i;
4    A(int n) : i(n) {}
5  };
6
7  class B {
8  public:
9    int i;
10   B(int n) : i(n) {}
11 };
12
13 void f(A a) {}
14 void f(B b) {} // overload
15
```

6.7 CONSTRUCTORS FOR TYPE CONVERSION

Consider the following situation:

```
1  class A {
2  public:
3    int i;
4    A(int n) : i(n) {}
5  };
6
7  class B {
8  public:
9    int i;
10   B(int n) : i(n) {}
11 };
12
13 void f(A a) {}
14 void f(B b) {} // overload
15
```

6.7 CONSTRUCTORS FOR TYPE CONVERSION

Consider the following situation:

```
 2  public:
 3    int i;
 4    A(int n) : i(n) {}
 5  };
 6
 7  class B {
 8  public:
 9    int i;
10    B(int n) : i(n) {}
11  };
12
13  void f(A a) {}
14  void f(B b) {} // overload
15
16  int main() {
```

6.7 CONSTRUCTORS FOR TYPE CONVERSION

Consider the following situation:

```
5  };
6
7  class B {
8  public:
9    int i;
10   B(int n) : i(n) {}
11 };
12
13 void f(A a) {}
14 void f(B b) {} // overload
15
16 int main() {
17   f(10); // compile error!
18   return 0;
19 }
```

6.7 CONSTRUCTORS FOR TYPE CONVERSION

Consider the following situation:

```
5  };
6
7  class B {
8  public:
9    int i;
10   B(int n) : i(n) {}
11 };
12
13 void f(A a) {}
14 void f(B b) {} // overload
15
16 int main() {
17   f(10); // compile error!
18   return 0;
19 }
```

6.7 CONSTRUCTORS FOR TYPE CONVERSION

6.7 CONSTRUCTORS FOR TYPE CONVERSION

- This situation is usually not needed

6.7 CONSTRUCTORS FOR TYPE CONVERSION

- This situation is usually not needed
- And it's harmful (implicit type conversion should be avoided as much as possible, as it can easily cause ambiguity)

6.7 CONSTRUCTORS FOR TYPE CONVERSION

- This situation is usually not needed
- And it's harmful (implicit type conversion should be avoided as much as possible, as it can easily cause ambiguity)
- Good programming style:

6.7 CONSTRUCTORS FOR TYPE CONVERSION

- This situation is usually not needed
- And it's harmful (implicit type conversion should be avoided as much as possible, as it can easily cause ambiguity)
- Good programming style:
  - If the constructor has only one parameter

6.7 CONSTRUCTORS FOR TYPE CONVERSION

- This situation is usually not needed
- And it's harmful (implicit type conversion should be avoided as much as possible, as it can easily cause ambiguity)
- Good programming style:
  - If the constructor has only one parameter
  - Then add the explicit keyword before the constructor

6.8 FURTHER READING

6.8 FURTHER READING

- In fact, function passing values by copy is a performance issue. Modern C++ introduced std::move to solve this problem.

6.8 FURTHER READING

- In fact, function passing values by copy is a performance issue. Modern C++ introduced std::move to solve this problem.
- Similarly, to solve this problem, the Rust language introduced the Onwership concept.

6.8 FURTHER READING

- In fact, function passing values by copy is a performance issue. Modern C++ introduced std::move to solve this problem.
- Similarly, to solve this problem, the Rust language introduced the Onwership concept.
- Reference materials:

6.8 FURTHER READING

- In fact, function passing values by copy is a performance issue. Modern C++ introduced std::move to solve this problem.
- Similarly, to solve this problem, the Rust language introduced the Onwership concept.
- Reference materials:
  - lvalues and rvalues in C++ - Youtube

6.8 FURTHER READING

- In fact, function passing values by copy is a performance issue. Modern C++ introduced std::move to solve this problem.
- Similarly, to solve this problem, the Rust language introduced the Onwership concept.
- Reference materials:
  - lvalues and rvalues in C++ - Youtube
  - Understanding Onwership - The Rust Programming Language

CHALLENGE

Implement your own vector

- Do not use generics, only for int type
- Initial memory size is 8
- If memory is insufficient, automatically expand memory (current memory quantity multiplied by 2)
- capacity returns memory size
- size returns the number of elements
- int get(int index) const and std::optional<int> safe_get(int index) const
- bool set(int index, int value)
- std::string to_string() const

CHALLENGE

Implement your own vector

ADVANCED

CHALLENGE

Implement your own `vector`

ADVANCED

- Copy constructor

CHALLENGE

Implement your own vector

ADVANCED

- Copy constructor
- Copy constructor for right value reference

CHALLENGE

Implement your own `vector`

ADVANCED

- Copy constructor
- Copy constructor for right value reference
- Test these two constructors using `std::move`

CHALLENGE

Implement your own vector

ADVANCED

- Copy constructor
- Copy constructor for right value reference
- Test these two constructors using std::move
- Implement push_back: automatically expand memory when capacity is insufficient