堆与拷贝构造函数

主讲：陈笑沙

目录

6.1 关于堆

程序驻留内存得以运行，其内存布局有四个区域：

6.1 关于堆

程序驻留内存得以运行，其内存布局有四个区域：

- Code 区（代码区）

6.1 关于堆

程序驻留内存得以运行，其内存布局有四个区域：

- Code 区（代码区）
- Data 区（全局数据区）

6.1 关于堆

程序驻留内存得以运行，其内存布局有四个区域：

- Code 区（代码区）
- Data 区（全局数据区）
- Heap 区（堆区）

6.1 关于堆

程序驻留内存得以运行，其内存布局有四个区域：

- Code 区（代码区）
- Data 区（全局数据区）
- Heap 区（堆区）
- Stack 区（栈区）

6.1 关于堆

6.1 关于堆

- **代码区**存放可执行指令

6.1 关于堆

- **代码区**存放可执行指令
  - 只读、可能被多个进程共享（如动态库）、大小固定、程序运行时加载

6.1 关于堆

- **代码区**存放可执行指令
  - 只读、可能被多个进程共享（如动态库）、大小固定、程序运行时加载
- **全局数据区**保存全局变量和静态变量、常量字符串

6.1 关于堆

- **代码区**存放可执行指令
  - 只读、可能被多个进程共享（如动态库）、大小固定、程序运行时加载
- **全局数据区**保存全局变量和静态变量、常量字符串
  - 可读写、生命周期与程序一致

6.1 关于堆

- **代码区**存放可执行指令
  - 只读、可能被多个进程共享（如动态库）、大小固定、程序运行时加载
- **全局数据区**保存全局变量和静态变量、常量字符串
  - 可读写、生命周期与程序一致
- **堆区**动态内存分配和存储区

6.1 关于堆

- **代码区**存放可执行指令
  - 只读、可能被多个进程共享（如动态库）、大小固定、程序运行时加载
- **全局数据区**保存全局变量和静态变量、常量字符串
  - 可读写、生命周期与程序一致
- **堆区**动态内存分配和存储区
  - 手动管理、地址由低到高、大小不固定、运行时动态分配、碎片化问题可能比较严重

6.1 关于堆

- **代码区**存放可执行指令
  - 只读、可能被多个进程共享（如动态库）、大小固定、程序运行时加载
- **全局数据区**保存全局变量和静态变量、常量字符串
  - 可读写、生命周期与程序一致
- **堆区**动态内存分配和存储区
  - 手动管理、地址由低到高、大小不固定、运行时动态分配、碎片化问题可能比较严重
- **栈区**存放局部变量、函数参数、返回地址，保存函数调用的上下文（如寄存器状态）

6.1 关于堆

- **代码区**存放可执行指令
  - 只读、可能被多个进程共享（如动态库）、大小固定、程序运行时加载
- **全局数据区**保存全局变量和静态变量、常量字符串
  - 可读写、生命周期与程序一致
- **堆区**动态内存分配和存储区
  - 手动管理、地址由低到高、大小不固定、运行时动态分配、碎片化问题可能比较严重
- **栈区**存放局部变量、函数参数、返回地址，保存函数调用的上下文（如寄存器状态）
  - 自动管理、地址由高到低、大小有限、快速访问

6.1 关于堆

C 语言中的堆申请与释放

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

int num = 10; // 数据区

int main()
{
    int* arr = (int*)malloc(sizeof(int) * num); // 堆区申请
    for (int i = 0; i < num; i++) {
        arr[i] = i; // arr 和 i 为栈区变量
        printf("%d\n", arr[i]);
    }

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

可以在https://godbolt.org上查看汇编代码，直观感受底层原理（最好把 Compiler options 选项清空）

6.1 关于堆

C 语言中的堆申请与释放

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

int num = 10; // 数据区

int main()
{
    int* arr = (int*)malloc(sizeof(int) * num); // 堆区申请
    for (int i = 0; i < num; i++) {
        arr[i] = i; // arr 和 i 为栈区变量
        printf("%d\n", arr[i]);
    }

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

可以在https://godbolt.org上查看汇编代码，直观感受底层原理（最好把 Compiler options 选项清空）

6.1 关于堆

C 语言中的堆申请与释放

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

int num = 10; // 数据区

int main()
{
    int* arr = (int*)malloc(sizeof(int) * num); // 堆区申请
    for (int i = 0; i < num; i++) {
        arr[i] = i; // arr 和 i 为栈区变量
        printf("%d\n", arr[i]);
    }

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

可以在https://godbolt.org上查看汇编代码，直观感受底层原理（最好把 Compiler options 选项清空）

6.1 关于堆

C 语言中的堆申请与释放

```c
#include <stdio.h>

int num = 10; // 数据区

int main()
{
    int* arr = (int*)malloc(sizeof(int) * num); // 堆区申请
    for (int i = 0; i < num; i++) {
        arr[i] = i; // arr 和 i 为栈区变量
        printf("%d\n", arr[i]);
    }

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

可以在https://godbolt.org上查看汇编代码，直观感受底层原理（最好把 Compiler options 选项清空）

6.1 关于堆

C 语言中的堆申请与释放

```c
#include <stdio.h>

int num = 10; // 数据区

int main()
{
    int* arr = (int*)malloc(sizeof(int) * num); // 堆区申请
    for (int i = 0; i < num; i++) {
        arr[i] = i; // arr 和 i 为栈区变量
        printf("%d\n", arr[i]);
    }

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

可以在https://godbolt.org上查看汇编代码，直观感受底层原理（最好把 Compiler options 选项清空）

6.2 需要 new 和 delete 的原因

C++ 中有大量的对象数据，在申请堆的同时，还需要对内存进行初始化（构造函数）；释放内存时，需要调用析构函数。

```
 1 class Point {
 2 public:
 3     Point() { x = 1; y = 1; } // 无参构造函数
 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 需要 new 和 delete 的原因

C++ 中有大量的对象数据，在申请堆的同时，还需要对内存进行初始化（构造函数）；释放内存时，需要调用析构函数。

```
 1 class Point {
 2 public:
 3     Point() { x = 1; y = 1; } // 无参构造函数
 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 需要 new 和 delete 的原因

malloc 与 free 的不足。

```
1 int main() {
2     // 没有调用构造函数
3     Point* pd = (Point*)malloc(10 * sizeof(Point));
4     for (int i = 0; i < 10; i++)
5         pd[i].set(1, 1); // 需要这样初始化
6     free(pd); // 没有调用析构函数
7     return 0;
8 }
```

6.2 需要 new 和 delete 的原因

malloc 与 free 的不足。

```
1 int main() {
2     // 没有调用构造函数
3     Point* pd = (Point*)malloc(10 * sizeof(Point));
4     for (int i = 0; i < 10; i++)
5         pd[i].set(1, 1); // 需要这样初始化
6     free(pd); // 没有调用析构函数
7     return 0;
8 }
```

6.2 需要 new 和 delete 的原因

malloc 与 free 的不足。

```
1 int main() {
2     // 没有调用构造函数
3     Point* pd = (Point*)malloc(10 * sizeof(Point));
4     for (int i = 0; i < 10; i++)
5         pd[i].set(1, 1); // 需要这样初始化
6     free(pd); // 没有调用析构函数
7     return 0;
8 }
```

6.2 需要 new 和 delete 的原因

malloc 与 free 的不足。

```
1 int main() {
2     // 没有调用构造函数
3     Point* pd = (Point*)malloc(10 * sizeof(Point));
4     for (int i = 0; i < 10; i++)
5         pd[i].set(1, 1); // 需要这样初始化
6     free(pd); // 没有调用析构函数
7     return 0;
8 }
```

6.3 分配堆对象

```
1  int main() {
2      Point* pd = new Point; // 申请一个对象
3      Point* pds = new Point[10]; // 申请一个数组
4      pd→print(); // 已经初始化，不用再设置
5      pds[2].print();
6
7      delete pd; // 释放一个对象
8      delete[] pds; // 释放多个对象
9
10     return 0;
11 }
```

6.3 分配堆对象

```
int main() {
    Point* pd = new Point; // 申请一个对象
    Point* pds = new Point[10]; // 申请一个数组
    pd->print(); // 已经初始化，不用再设置
    pds[2].print();

    delete pd; // 释放一个对象
    delete[] pds; // 释放多个对象

    return 0;
}
```

6.3 分配堆对象

```
int main() {
    Point* pd = new Point; // 申请一个对象
    Point* pds = new Point[10]; // 申请一个数组
    pd->print(); // 已经初始化，不用再设置
    pds[2].print();

    delete pd; // 释放一个对象
    delete[] pds; // 释放多个对象

    return 0;
}
```

6.3 分配堆对象

```
1  int main() {
2      Point* pd = new Point; // 申请一个对象
3      Point* pds = new Point[10]; // 申请一个数组
4      pd→print(); // 已经初始化，不用再设置
5      pds[2].print();
6
7      delete pd; // 释放一个对象
8      delete[] pds; // 释放多个对象
9
10     return 0;
11 }
```

6.3 分配堆对象

```
1  int main() {
2      Point* pd = new Point; // 申请一个对象
3      Point* pds = new Point[10]; // 申请一个数组
4      pd->print(); // 已经初始化，不用再设置
5      pds[2].print();
6
7      delete pd; // 释放一个对象
8      delete[] pds; // 释放多个对象
9
10     return 0;
11 }
```

6.3 分配堆对象

```
int main() {
    Point* pd = new Point; // 申请一个对象
    Point* pds = new Point[10]; // 申请一个数组
    pd->print(); // 已经初始化，不用再设置
    pds[2].print();

    delete pd; // 释放一个对象
    delete[] pds; // 释放多个对象

    return 0;
}
```

6.3 分配堆对象

```
int main() {
    Point* pd = new Point; // 申请一个对象
    Point* pds = new Point[10]; // 申请一个数组
    pd->print(); // 已经初始化，不用再设置
    pds[2].print();

    delete pd; // 释放一个对象
    delete[] pds; // 释放多个对象

    return 0;
}
```

6.3 分配堆对象

```
 1 int main() {
 2     Point* pd = new Point; // 申请一个对象
 3     Point* pds = new Point[10]; // 申请一个数组
 4     pd→print(); // 已经初始化，不用再设置
 5     pds[2].print();
 6
 7     delete pd; // 释放一个对象
 8     delete[] pds; // 释放多个对象
 9
10     return 0;
11 }
```

6.3 分配堆对象

6.3 分配堆对象

- new 与 delete 为 C++ 操作符，不用引入新的头文件

6.3 分配堆对象

- new 与 delete 为 C++ 操作符，不用引入新的头文件
- 其功能覆盖 C 语言，可以申请基本数据类型

6.3 分配堆对象

- new 与 delete 为 C++ 操作符，不用引入新的头文件
- 其功能覆盖 C 语言，可以申请基本数据类型
- 性能与 C 语言相同，自动类型匹配，无需转换

6.3 分配堆对象

用 new 时，可以传入构造函数参数：

```
 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; // 此处不能有参数
15     return 0;
```

6.3 分配堆对象

用 new 时，可以传入构造函数参数：

```
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; // 此处不能有参数
15     return 0;
```

6.3 分配堆对象

用 new 时，可以传入构造函数参数：

```
 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; // 此处不能有参数
15     return 0;
16 }
```

6.3 分配堆对象

用 new 时，可以传入构造函数参数：

```
 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; // 此处不能有参数
15     return 0;
16 }
```

6.3 分配堆对象

用 new 时，可以传入构造函数参数：

```
public:
    Point(int x, int y): x(x), y(y){}
    void print() { cout << x << ", " << y; }
    void set(int x, int y) {
        this->x = x;
        this->y = y;
    }
};

int main() {
    Point* p = new Point(3, 4);
    p->print();
    delete p; // 此处不能有参数
    return 0;
}
```

6.3 分配堆对象

用 new 时，可以传入构造函数参数：

```
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; // 此处不能有参数
15     return 0;
```

6.4 拷贝构造函数

对象常见的操作：

6.4 拷贝构造函数

对象常见的操作：

- 创建（构造函数）

6.4 拷贝构造函数

对象常见的操作：

- 创建（构造函数）
- 比较

6.4 拷贝构造函数

对象常见的操作：

- 创建（构造函数）
- 比较
- 赋值

6.4 拷贝构造函数

对象常见的操作：

- 创建（构造函数）
- 比较
- 赋值
- 拷贝构造

6.4 拷贝构造函数

对象常见的操作：

- 创建（构造函数）
- 比较
- 赋值
- 拷贝构造
- 析构（析构函数）

6.4 拷贝构造函数

题外话

实质上，并不是所有对象都支持这些操作，C++ 中并没有对哪些对象需要支持哪些操作进行严格的限制，在 C++ 20 之后，引入了 concept 的概念，以解决这一问题。

6.4 拷贝构造函数

当一个类没有自定义这些操作时，编译器会自动添加一个默认的操作。

```
class Student {
public:
    Student(); // 默认构造函数
    Student(const Student& other); // 默认拷贝构造函数
    ~Student(); // 默认析构函数
    Student& operator=(const Student& other); // 默认赋值函数
    bool operator==(const Student& other) const; // 默认比较函数
};
```

6.4 拷贝构造函数

我们可以自定义拷贝构造函数：

```
#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;
```

6.4 拷贝构造函数

我们可以自定义拷贝构造函数：

```
#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;
  }
```

6.4 拷贝构造函数

我们可以自定义拷贝构造函数：

```
 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 拷贝构造函数

我们可以自定义拷贝构造函数：

```
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 拷贝构造函数

我们可以自定义拷贝构造函数：

```
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  }
```

6.4 拷贝构造函数

我们可以自定义拷贝构造函数：

```
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  }
```

6.4 拷贝构造函数

6.4 拷贝构造函数

- 如果将与自己同类的对象的引用作为参数时，该构造函数就称为拷贝构造函数。

6.4 拷贝构造函数

- 如果将与自己同类的对象的引用作为参数时，该构造函数就称为拷贝构造函数。
- 拷贝构造函数的特点

6.4 拷贝构造函数

- 如果将与自己同类的对象的引用作为参数时，该构造函数就称为拷贝构造函数。
- 拷贝构造函数的特点
  - 它是一个构造函数，当创建对象时系统会自动调用它。

6.4 拷贝构造函数

- 如果将与自己同类的对象的引用作为参数时，该构造函数就称为拷贝构造函数。
- 拷贝构造函数的特点
  - 它是一个构造函数，当创建对象时系统会自动调用它。
  - 它将一个已经创建好的对象作为参数，根据需要将该对象中的数据成员逐一对应地赋值给新对象。

6.4 拷贝构造函数

6.4 拷贝构造函数

- 如果没有定义拷贝构造函数，那么编译器会为该类产生一个缺省的拷贝构造函数。

6.4 拷贝构造函数

- 如果没有定义拷贝构造函数，那么编译器会为该类产生一个缺省的拷贝构造函数。
- 缺省的拷贝构造函数使用位拷贝的方法来完成对象到对象的复制。

6.5 浅拷贝与深拷贝

考虑以下情形，有什么问题?

```
#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; }
```

6.5 浅拷贝与深拷贝

考虑以下情形，有什么问题?

```
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 浅拷贝与深拷贝

考虑以下情形，有什么问题?

```
#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 浅拷贝与深拷贝

考虑以下情形，有什么问题?

```
 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 浅拷贝与深拷贝

考虑以下情形，有什么问题?

```
 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 浅拷贝与深拷贝

考虑以下情形，有什么问题?

```
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 浅拷贝与深拷贝

考虑以下情形，有什么问题?

```
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  }
```

6.5 浅拷贝与深拷贝

一个对象改变，另一个对象也发生改变，因为共用一片内存空间。

```
#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; }
```

6.5 浅拷贝与深拷贝

一个对象改变，另一个对象也发生改变，因为共用一片内存空间。

```
 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 浅拷贝与深拷贝

如何修改?

```
#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; }
```

6.5 浅拷贝与深拷贝

如何修改?

```
#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 = new int[this->size];
    for (int i = 0; i < this->size; i++)
        this->arr[i] = other.arr[i];
```

6.6 临时对象

考虑以下情况(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 临时对象

考虑以下情况(example/lec06/tempObject):

```
using namespace std;

class A {
public:
  A(string id) : id(id) {
    cout << "Constructing A with id " << id << endl;
  }
  A(const A &other) : id(other.id) {
    cout << "Copy " << other.id << endl;
  }

private:
  string id;
};
```

6.6 临时对象

考虑以下情况(example/lec06/tempObject):

```
13
14 private:
15   string id;
16 };
17
18 A returnValueFunc(A a) {
19   A b = a;
20   return b; // 如果没有返回值优化，此处会发生拷贝
21 }
22
23 int main() {
24   A a("a1");
25   A b = a;
26   A c = returnValueFunc(b);
```

6.6 临时对象

考虑以下情况(example/lec06/tempObject):

```
14 private:
15     string id;
16 };
17
18 A returnValueFunc(A a) {
19     A b = a;
20     return b; // 如果没有返回值优化，此处会发生拷贝
21 }
22
23 int main() {
24     A a("a1");
25     A b = a;
26     A c = returnValueFunc(b);
27     return 0;
28 }
```

6.6 临时对象

考虑以下情况(example/lec06/tempObject):

```
14 private:
15   string id;
16 };
17
18 A returnValueFunc(A a) {
19   A b = a;
20   return b; // 如果没有返回值优化，此处会发生拷贝
21 }
22
23 int main() {
24   A a("a1");
25   A b = a;
26   A c = returnValueFunc(b);
27   return 0;
28 }
```

6.6 临时对象

考虑以下情况(example/lec06/tempObject):

```
14 private:
15   string id;
16 };
17
18 A returnValueFunc(A a) {
19   A b = a;
20   return b; // 如果没有返回值优化，此处会发生拷贝
21 }
22
23 int main() {
24   A a("a1");
25   A b = a;
26   A c = returnValueFunc(b);
27   return 0;
28 }
```

6.6 临时对象

```
 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 临时对象

共发生了几次拷贝?

```cpp
#include <iostream>

using namespace std;

class A {
public:
  A(string id) : id(id) {
    cout << "Constructing A with id " << id << endl;
  }
  A(const A &other) : id(other.id) {
    cout << "Copy " << other.id << endl;
  }

private:
  string id;
```

6.6 临时对象

共发生了几次拷贝?

```
13
14 private:
15   string id;
16 };
17
18 A returnValueFunc(A a) {
19   A b = a;
20   return b; // 如果没有返回值优化，此处会发生拷贝
21 }
22
23 int main() {
24   A a("a1");
25   A b = a;
26   A c = returnValueFunc(b);
```

6.6 临时对象

6.6 临时对象

- C++ 中，函数参数如果按值传入，会发生拷贝

6.6 临时对象

- C++ 中，函数参数如果按值传入，会发生拷贝
- 函数参数按值返回，则会认为这是一个临时对象，进行优化

6.6 临时对象

- C++ 中，函数参数如果按值传入，会发生拷贝
- 函数参数按值返回，则会认为这是一个临时对象，进行优化
- 可以加上 `-fno-elide-constructors` 参数以关闭返回值优化

6.6 临时对象

再考虑以下情况：

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

6.6 临时对象

再考虑以下情况：

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

此时，表面上是用 A("a") 创建了一个 **无名对象**，然后调用拷贝构造函数初始化 a，但是实际上仍然会调用构造函数。

6.7 构造函数用于类型转换

只有一个参数的构造函数可以进行类型转换

```
1  class Student {
2      Student(const string& name) : name(name) {}
3  private:
4      string name;
5  };
6
7  void f(Student stu) { ... }
8
9  // 可以直接这样调用
10 f("Trump");
```

6.7 构造函数用于类型转换

只有一个参数的构造函数可以进行类型转换

```
1  class Student {
2      Student(const string& name) : name(name) {}
3  private:
4      string name;
5  };
6
7  void f(Student stu) { ... }
8
9  // 可以直接这样调用
10 f("Trump");
```

6.7 构造函数用于类型转换

只有一个参数的构造函数可以进行类型转换

```
1  class Student {
2      Student(const string& name) : name(name) {}
3  private:
4      string name;
5  };
6
7  void f(Student stu) { ... }
8
9  // 可以直接这样调用
10 f("Trump");
```

6.7 构造函数用于类型转换

只有一个参数的构造函数可以进行类型转换

```
1  class Student {
2      Student(const string& name) : name(name) {}
3  private:
4      string name;
5  };
6
7  void f(Student stu) { ... }
8
9  // 可以直接这样调用
10 f("Trump");
```

6.7 构造函数用于类型转换

只有一个参数的构造函数可以进行类型转换

```
1  class Student {
2      Student(const string& name) : name(name) {}
3  private:
4      string name;
5  };
6
7  void f(Student stu) { ... }
8
9  // 可以直接这样调用
10 f("Trump");
```

6.7 构造函数用于类型转换

只有一个参数的构造函数可以进行类型转换

```
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  // 可以直接这样调用
10 f("Trump");
```

6.7 构造函数用于类型转换

只有一个参数的构造函数可以进行类型转换

```
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  // 可以直接这样调用
10 f("Trump");
```

6.7 构造函数用于类型转换

只有一个参数的构造函数可以进行类型转换

```
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  // 可以直接这样调用
10 f("Trump");
```

6.7 构造函数用于类型转换

6.7 构造函数用于类型转换

再考虑如下情况：

6.7 构造函数用于类型转换

再考虑如下情况：

```
 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 构造函数用于类型转换

再考虑如下情况：

```
 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 构造函数用于类型转换

再考虑如下情况：

```
 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 构造函数用于类型转换

再考虑如下情况：

```
 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 构造函数用于类型转换

再考虑如下情况：

```
 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 构造函数用于类型转换

6.7 构造函数用于类型转换

- 这种情况一般是不需要的

6.7 构造函数用于类型转换

- 这种情况一般是不需要的
- 而且是有害的（隐形类型转换应当尽量避免，容易引起歧义）

6.7 构造函数用于类型转换

- 这种情况一般是不需要的
- 而且是有害的（隐形类型转换应当尽量避免，容易引起歧义）
- 好的编程风格：

6.7 构造函数用于类型转换

- 这种情况一般是不需要的
- 而且是有害的（隐形类型转换应当尽量避免，容易引起歧义）
- 好的编程风格：
  - 如果构造函数只有一个参数

6.7 构造函数用于类型转换

- 这种情况一般是不需要的
- 而且是有害的（隐形类型转换应当尽量避免，容易引起歧义）
- 好的编程风格：
  - 如果构造函数只有一个参数
  - 那么在构造函数前加 `explicit` 关键字

6.8 扩展阅读

6.8 扩展阅读

- 实际上，函数传递值发生拷贝是一个影响性能的问题，现代 C++ 引入了 `std::move` 来解决这一问题。

6.8 扩展阅读

- 实际上，函数传递值发生拷贝是一个影响性能的问题，现代 C++ 引入了 `std::move` 来解决这一问题。
- 同样是为了解决这一问题，Rust 语言引入了 `Onwership` 概念。

6.8 扩展阅读

- 实际上，函数传递值发生拷贝是一个影响性能的问题，现代 C++ 引入了 `std::move` 来解决这一问题。
- 同样是为了解决这一问题，Rust 语言引入了 `Onwership` 概念。
- 参考资料：

6.8 扩展阅读

- 实际上，函数传递值发生拷贝是一个影响性能的问题，现代 C++ 引入了 `std::move` 来解决这一问题。
- 同样是为了解决这一问题，Rust 语言引入了 Onwership 概念。
- 参考资料：
  - 一文读懂C++右值引用和std::move - 知乎

6.8 扩展阅读

- 实际上，函数传递值发生拷贝是一个影响性能的问题，现代 C++ 引入了 `std::move` 来解决这一问题。
- 同样是为了解决这一问题，Rust 语言引入了 `Onwership` 概念。
- 参考资料：
  - 一文读懂C++右值引用和std::move - 知乎
  - Rust：所有权（ownership） - 知乎

挑战

实现自己的 `vector`

- 不用使用泛型，只针对 `int` 类型
- 初始内存大小为 8
- 如果内存不够，自动扩展内存（当前内存数量乘以 2）
- `capacity` 返回内存大小
- `size` 返回元素数量
- `int get(int index) const` 与 `std::optional<int> safe_get(int index) const`
- `bool set(int index, int value)`
- `std::string to_string() const`

挑战

实现自己的 vector

进阶

挑战

实现自己的 `vector`

进阶

- 拷贝构造函数

挑战

实现自己的 `vector`

进阶

- 拷贝构造函数
- 针对右值引用的拷贝构造函数

挑战

实现自己的 `vector`

进阶

- 拷贝构造函数
- 针对右值引用的拷贝构造函数
- 使用 `std::move` 对这两个构造函数进行测试

挑战

实现自己的 `vector`

进阶

- 拷贝构造函数
- 针对右值引用的拷贝构造函数
- 使用 `std::move` 对这两个构造函数进行测试
- 实现 `push_back`：`capacity` 不足时自动扩充内存