# CS100 Recitation 9

GKxx

# class 的基础：以 `Dynarray` 为例

# 设计

实现一个“动态数组”：

```
int n; std::cin >> n;
Dynarray arr(n); // arr has n value-initialized ints.
for (int i = 0; i != n; ++i)
  std::cin >> arr.at(i); // arr.at(i) is just like arr[i]
                         // 如何做到 arr[i]？以后再说
for (std::size_t i = 0; i != arr.size(); ++i)
  arr.at(i) += arr.at(i - 1) * 2;
// ...
```

最后 `arr` 应当自己释放自己所占用的内存。

# Dynarray：成员

```
class Dynarray {
  std::size_t m_size;
  int *m_storage;
public:
  bool empty() const {
    return m_size == 0u;
  }
  std::size_t size() const {
    return m_size;
  }
};
```

# Dynarray：构造函数

构造函数定义的是**初始化的方式**。

```
Dynarray a;        // 对应默认构造函数 Dynarray()
Dynarray b(n);     // 对应构造函数 Dynarray(std::size_t)
Dynarray c(n, x); // 对应构造函数 Dynarray(std::size_t, int)
```

```
class Dynarray {
public:
  Dynarray();
  Dynarray(std::size_t);
  Dynarray(std::size_t, int);
};
```

## Dynarray：构造函数

假如我们需要一个默认构造函数，它应当具有什么行为？

- 初始化为 `0` 个元素的数组？
  - 这样的数组能使用吗？
- 初始化为 `DYNARRAY_DEFAULT_SIZE` 个元素的数组？

# Dynarray：构造函数

假如我们需要一个默认构造函数，它应当具有什么行为？

- 初始化为 `0` 个元素的数组？
  - 在不知道如何拷贝的情况下，我们似乎没有什么能改变数组大小的操作。
  - 如果不能改变大小，那“`0` 个元素的数组”有什么用？
- 初始化为 `DYNARRAY_DEFAULT_SIZE` 个元素的数组？
  - 也许在某些应用场景下，这是合理的。

# 默认构造函数 (default constructor)：复习

- 默认构造函数的“默认”体现在哪？
- 哪些使用方式依赖于默认构造函数？
- 如果没有自定义默认构造函数，这个类会有默认构造函数吗？
  - 如果有，其行为是什么？
- 默认构造函数 `= default` 意味着什么？

# 默认构造函数 (default constructor)：复习

- 默认构造函数的"默认"体现在哪？——无参数
- 哪些使用方式依赖于默认构造函数？——默认初始化和值初始化
  - ```
    Dynarray a, b{};
    Dynarray *c = new Dynarray, *d = new Dynarray(), *e = new Dynarray{};
    Dynarray f[1000]; std::array<Dynarray, 1000> g; // 这是啥？
    ```
- 如果没有自定义默认构造函数，这个类会有默认构造函数吗？
  ——如果有其它自定义的构造函数，编译器就不会合成默认构造函数
- 默认构造函数 `= default` 意味着什么？
    ```

## 默认构造函数 (default constructor)：复习

如果没有其它自定义的构造函数，或者我们显式地以 `= default` 要求编译器合成一个默认构造函数：

- 编译器会合成一个具有默认行为的默认构造函数。
- “默认行为”：按照成员声明顺序逐个初始化它们
  - 对于有类内初始值的成员，使用类内初始值
  - 对于其它成员，进行**默认初始化**

# 析构函数 (destructor, dtor)

`Dynarray` 必须在自己不再被使用的时候释放所拥有的内存：

```
void fun() {
  Dynarray a(n);
  if (condition) {
    Dynarray b(m);
    // ...
  } // Now b should release the memory it allocated.
  // ...
} // Now a should release the memory it allocated.
```

## 析构函数 (destructor, dtor)

**析构函数**是在这个对象被销毁的时候自动调用的函数。

- 它通常用来完成一些最后的清理
- 特别地，那些**拥有一定资源**的类通常在析构函数里释放它们所拥有的资源

# 定义一个析构函数

```
class Student {
  std::string name;
  std::string id;
  int entranceYear;
public:
  ~Student() {
    std::cout << "dtor of "
              << name << "called.\n";
  }
};
```

```
{
  Student alice("Alice", "2020123123");
  if (condition) {
    Student bob("Bob", "2020123124");
    // ...
  } // Output "dtor of Bob called."
} // Output "dtor of Alice called."
```

- 析构函数的名字是 `~ClassName`
- 析构函数不接受参数，不声明返回值类型
- 一个类只能有一个析构函数

# Dynarray：析构函数

非常直接的实现

```
class Dynarray {
  std::size_t m_size;
  int *m_storage;
public:
  Dynarray(std::size_t n)
    : m_size(n), m_storage(new int[n]{}) {}
  ~Dynarray() {
    delete[] m_storage;
  }
};
```

# 析构函数的具体行为

考虑一个问题：`Student` 是否拥有什么资源？

```
class Student {
  std::string name;
  std::string id;
  int entranceYear;
  // ...
};
```

# 析构函数的具体行为

`Student` 本身不拥有什么资源，但是 `std::string` 有。

- `std::string` 当然会在它自己的析构函数里释放内存。

```
class Student {
  std::string name;
  std::string id;
  int entranceYear;
  // ...
};
```

在一个 `Student` 被销毁的时候：

- 它显然应该销毁它的所有成员
- 销毁一个成员，显然应该调用那个成员的析构函数。

# 析构函数的具体行为

**析构函数在执行完函数体之后，会自动销毁它的所有数据成员**

- 对于类类型成员，会调用它的析构函数来销毁它。
- 销毁成员的顺序？

# 析构函数的具体行为

**析构函数在执行完函数体之后，会自动销毁它的所有数据成员**

- 对于类类型成员，会调用它的析构函数来销毁它。
- 按照成员的声明顺序**倒序**销毁它们。

对比一下构造函数：

- 在执行函数体**之前**初始化所有成员。
- 对于类类型成员，调用它的构造函数进行初始化。
- 初始化的顺序是成员的声明顺序。

# 析构函数的具体行为

对于成员的销毁是不需要我们写的，会自动完成。

```
class Student {
  std::string name;
  std::string id;
  int entranceYear;
public:
  ~Student() {} // 编译器会在最后插入代码来调用 id 和 name 的析构函数。
};
```

`Student` 类的析构函数只需一个空函数体即可。

等价的写法：`~Student() = default;`

**在任何一个成员函数的函数体中，这个对象的所有数据成员都处在已经初始化、未被析构的状态下。**

# 析构函数的具体行为

初学者典型的错误：

```
class Student {
  std::string name;
  std::string id;
  int entranceYear;
public:
  ~Student() {
    // 完全误解了析构函数的函数体的用途。
    delete this;
  }
};
```

可以，但没必要：

```
class Student {
  std::string name;
  std::string id;
  int entranceYear;
public:
  ~Student() {
    // 毫无意义的行为。
    name.clear(); id.clear();
    entranceYear = 0;
  }
};
```

# 析构函数的具体行为

初学者典型的错误：

```
class Dynarray {
  std::size_t m_size;
  int *m_storage;
public:
  ~Student() {
    delete[] m_storage;
    delete this; // 错误！
  }
};
```

可以，但没必要：

```
class Dynarray {
  std::size_t m_size;
  int *m_storage;
public:
  ~Student() {
    delete[] m_storage;
    // 下面的赋值毫无意义。
    m_size = 0;
    m_storage = nullptr;
  }
};
```

## 一个类不能没有析构函数，就像...

如果一个类的析构函数是不可调用的，就意味着它无法被销毁

- 就如同不可降解的塑料

C++ 不允许定义这样的类类型的对象！

什么情况下析构函数不可调用？

# 一个类不能没有析构函数，就像...

如果一个类的析构函数是不可调用的，就意味着它无法被销毁

C++ 不允许定义这样的类类型的对象！

什么情况下析构函数不可调用？

- 你可以显式地 `~ClassName() = delete;` 将析构函数定义为“删除的”。
- 如果析构函数是 `private` 的，它就不能在类外、`friend` 外被调用。

如果一个类的析构函数是不可用的，创建这个类型的对象的唯一方式是 `new` ，但你还没法 `delete` 它。

# 析构函数还能干什么？

```
void drawUI(/* ... */) {
  hide_cursor();
  disable_input_echo();

  // 画一些东西 ...

  // 要记得恢复光标和输入显示
  show_cursor();
  enable_input_echo();
}
```

```
void loadMap(const char *path) {
  FILE *file = fopen(path, "r");
  // 从文件中读取一些内容 ...
  fclose(file);
}
```

```
void criticalSection(std::mutex &m) {
  m.lock();
  // ...
  m.unlock();
}
```

# 析构函数还能干什么？

```
struct PrinterGuard {
  PrinterGuard() {
    hide_cursor();
    disable_input_echo();
  }
  ~PrinterGuard() {
    show_cursor();
    enable_input_echo();
  }
};
void drawUI(/* ... */) {
  PrinterGuard guard;
  // 画一些东西 ...
  // guard 被销毁，光标和输入显示被恢复
}
```

```
void loadMap(const std::string &path) {
  std::ifstream file(path);

  // ...

  // std::ifstream 的析构函数会关闭文件
}
```

```
void criticalSection(std::mutex &m) {
  std::lock_guard guard(m);
  // ...
  // std::lock_guard 的析构函数会解锁 m
}
```

## `new` 和 `delete`

```
std::string *p = new std::string("Hello");
std::cout << *p << std::endl;
delete p;
p = new std::string; // 调用默认构造函数，*p 为空串 ""
std::cout << p->size() << std::endl;
delete p;
```

`new` 表达式会**先分配内存**，然后**构造对象**。

- 如果这是一个类类型，它必然会调用一个构造函数来构造对象。在没有指定如何构造的情况下，它会调用**默认构造函数**。
- 相比之下，来自 C 的 `malloc` 只会分配内存，不构造任何对象（不做任何初始化）。`calloc` 会将这个内存**清零**，而非调用默认构造函数。

## `new` 和 `delete`

```
std::string *p = new std::string("Hello");
std::cout << *p << std::endl;
delete p;
p = new std::string; // 调用了默认构造函数，*p 为空串 ""
std::cout << p->size() << std::endl;
delete p;
```

`delete` 表达式会**先销毁对象**，然后释放这片内存。

- 如果这是一个类类型，它必然会调用析构函数来销毁这个对象。
- 相比之下，`free` 只释放内存，不调用析构函数。

## `Dynarray`：元素访问

`.at(i)` 应该返回什么？

```
class Dynarray {
  std::size_t m_size;
  int *m_storage;
public:
  ??? at(std::size_t n) {
    return m_storage[n];
  }
};
```

# `Dynarray`：元素访问

```
arr.at(0) = 42;
std::cout << arr.at(1);
```

如果我们希望通过 `arr.at(i)` 来修改这个元素，那必然要返回引用。

```
class Dynarray {
  std::size_t m_size;
  int *m_storage;
public:
  int &at(std::size_t n) {
    return m_storage[n];
  }
};
```

这样可以吗？

## `Dynarray`：元素访问

试一试：

```
void print(const Dynarray &arr) {
  for (std::size_t i = 0; i != arr.size(); ++i)
    std::cout << arr.at(i) << ' ';
}
```

无法编译：`at` 不是 `const` 成员函数，无法在 `const Dynarray &` 上调用！

## `const` 成员函数：复习

- `const` 写在哪？
- `const` 成员函数的 `const` 是作用于谁的？
- 在什么对象上能调用 `const` 成员函数？
- `const` 成员函数能做哪些事？不能做哪些事？

# `const` 成员函数：复习

- `const` 写在哪？——参数列表后
- `const` 成员函数的 `const` 是作用于谁的？
  - 加在隐式的 `this` 指针上的**底层** **`const`**
  - 表示当前对象是 `const`，其所有数据成员也都是 `const`
- 在什么对象上能调用 `const` 成员函数？
  - 什么对象上都可以，因为添加底层 `const` 永远没问题。
- `const` 成员函数能做哪些事？不能做哪些事？
  - 不能修改数据成员，不能调用数据成员的 non-`const` 成员函数
  - 不能调用自身的 non-`const` 成员函数

## `Dynarray`：元素访问

加个 `const` ？

```
class Dynarray {
  std::size_t m_size;
  int *m_storage;
public:
  int &at(std::size_t n) const {
    return m_storage[n];
  }
};
```

结果是在 `const Dynarray` 上，你可以得到其中元素的 non-`const` 引用，进而修改它！

# `Dynarray`：元素访问

正确的解决方案：`const` 和 non-`const` 的重载

```
class Dynarray {
  std::size_t m_size;
  int *m_storage;
public:
  const int &at(std::size_t n) const {
    return m_storage[n];
  }
  int &at(std::size_t n) {
    return m_storage[n];
  }
};
```

- 在 `const` 对象上，它只能调用 `const` 版本，得到 reference-to-`const`，无法修改
- 在非 `const` 对象上会调用哪个？

# `Dynarray`：元素访问

```
class Dynarray {
public:
  const int &at(std::size_t n) const {
    return m_storage[n];
  }
  int &at(std::size_t n) {
    return m_storage[n];
  }
};
arr.at(i) = 42;
```

```
// 左边的代码就如同：
const int &at(const Dynarray *this,
              std::size_t n) {
  return this->m_storage[n];
}
int &at(Dynarray *this, std::size_t n){
  return this->m_storage[n];
}
at(&arr, i) = 42;
```

- 在 `const` 对象上，它只能调用 `const` 版本，得到 reference-to-`const`，无法修改
- 在非 `const` 对象上：两个版本都可以调用，但是
  - 调用 non-`const` 版本是完美匹配，调用 `const` 版本是**添加底层 `const`** 得到的匹配，因此前者是更好的匹配。

# 在 `const` vs non-`const` 重载中避免重复

假如我们要模仿标准库的行为的话，`at` 函数应该提供边界检查...

```
class Dynarray {
public:
  const int &at(std::size_t n) const {
    if (n >= m_length) // 为什么不需要判断 n < 0？
      throw std::out_of_range{"Dynarray subscript out of range."};
    return m_storage[n];
  }
  int &at(std::size_t n) {
    if (n >= m_length)
      throw std::out_of_range{"Dynarray subscript out of range."};
    return m_storage[n];
  }
};
```

# 在 `const` vs non-`const` 重载中避免重复

假如我们还想在 `at` 访问中做一些其它的记录和检查...

```
class Dynarray {
public:
  const int &at(std::size_t n) const {
    if (n >= m_length)
      throw std::out_of_range{"Dynarray subscript out of range."};
    log_access();
    verify_integrity();
    return m_storage[n];
  }
  int &at(std::size_t n) {
    if (n >= m_length)
      throw std::out_of_range{"Dynarray subscript out of range."};
    log_access();
    verify_integrity();
    return m_storage[n];
  }
};
```

## 在 `const` vs non-`const` 重载中避免重复

将一模一样的代码编写两遍实在是太麻烦了... 有没有办法避免重复？

- C++23 deducing-this 能帮得上忙

假如没有额外的语法特性... 能不能让一个函数调用另一个？

- 让谁调用谁？

## 在 `const` vs non-`const` 重载中避免重复

将一模一样的代码编写两遍实在是太麻烦了... 有没有办法避免重复？

- C++23 deducing-this 能帮得上忙

假如没有额外的语法特性... 能不能让一个函数调用另一个？

假如我们让 non-`const` 版本的函数调用 `const` 版本的函数：

- 首先，我们需要显式地为 `this` 添加底层 `const`。
- `const` 版本的函数返回的是 `const int &`，我们得把它的底层 `const` 去除。

# 在 `const` vs non-`const` 重载中避免重复

- 先用 `static_cast<const Dynarray *>(this)` 为 `this` 添加底层 `const`
- 这时调用 `->at(n)`，就会匹配 `const` 版本的 `at`
- 将返回的 `const int &` 的底层 `const` 用 `const_cast` 去除

```
class Dynarray {
public:
  const int &at(std::size_t n) const {
    if (n >= m_length)
      throw std::out_of_range{"Dynarray subscript out of range."};
    log_access();
    verify_integrity();
    return m_storage[n];
  }
  int &at(std::size_t n) {
    return const_cast<int &>(static_cast<const Dynarray *>(this)->at(n));
  }
};
```

# 在 `const` vs non-`const` 重载中避免重复

能不能反过来，让 `const` 版本调用 non-`const` 版本？

```
class Dynarray {
public:
  int &at(std::size_t n) {
    if (n >= m_length)
      throw std::out_of_range{"Dynarray subscript out of range."};
    log_access();
    verify_integrity();
    return m_storage[n];
  }
  const int &at(std::size_t n) const {
    return const_cast<Dynarray *>(this)->at(n);
  }
};
```

## 在 `const` vs non-`const` 重载中避免重复

能不能反过来，让 `const` 版本调用 non-`const` 版本？

- **不能**！`const` 成员函数里一定不会修改对象的状态，但是 non-`const` 成员函数并没有这般承诺！
- 如果在 non-`const` 版本的实现里一不小心修改了对象的状态，让 `const` 版本调用它将导致灾难。

比较一下两种方法中对于“危险的” `const_cast` 的使用？

- “先添加，再去除”：OK
- “先去除，再添加”：危险