导读

预览

- 基本介绍
- 章节结构
    - 数据结构接口使用 | 代码演示
        - dslings - 测试代码
        - dslings - 检测结果
        - 代码接口介绍
    - 数据结构接口实现 | 类型定义
    - 数据结构接口实现 | 接口实现
    - 数据结构接口实现 | 完整代码
- 代码练习dslings
    - 代码下载
    - 安装xmake
    - dslings自动检测
        - 错误提示
        - 代码通过提示
        - XLINGS_WAIT - dslings等待标志
- 总结

动手写数据结构(d2ds)是一本强调动手实践的开源电子书, 每一章节都会介绍一个数据结构的基本用法和对应的具体实现。本书使用C++作为数据结构的开发语言, 并使用"编译器驱动开发模式"面向接口编程的形式, 来介绍常见数据结构的主体功能和实现。同时, 在d2ds仓库中也为每章节配有对应的练习代码和dslings检测程序, 真正让读者感受到"动手写"的感觉。下面我们就来详细的介绍一下 章节结构 和 dslings的使用。

> 注: 练习代码采用了类似rustlings的代码检测风格

章节结构

核心分两大部分, **数据结构接口使用 + 数据结构接口实现**。如下:

数据结构接口使用

dslings - MaxValue代码示例

```
// dslings.2.cpp - readonly
//
// 描述:
//  通过实现一个MaxVal类型(保存最大值), 来介绍dslings的"编译器驱动开发"
//  即根据编译器的错误提示来完成这个训练流程的演示Demo, 并且通常为了降低难度会把一个'数据结构'的实现分成多个检测模块.
//  如: dslings.0.cpp dslings.1.cpp dslings.2.cpp
//
// 目标/要求:
//  - 不修改该代码检测文件
//  - 在exercises/dslings.hpp中完成你的代码设计
//  - 通过所有编译器检测 和 断言
//

#include <dstruct.hpp>

#include "common/common.hpp"
#include "exercises/dslings.hpp"

int main() {

    d2ds::MaxValue mVal(2);

    d2ds_assert_eq(mVal.get(), 2);

    mVal.set(-1);
    d2ds_assert_eq(mVal.get(), 2);

    mVal.set(100);
    d2ds_assert_eq(mVal.get(), 100);

// random test
    dstruct::Array<int, 10> data;
    d2ds::random_data_generator(data, 0, 200);
    d2ds::ds_print(data);

    int maxVal = 0;
    for (int i = 0; i < data.size(); i++) {
        mVal.set(data[i]);
        if (data[i] > maxVal) {
            maxVal = data[i];
        }
    }
```

```
    d2ds_assert_eq(mVal.get(), maxVal);

    XLINGS_WAIT

    return 0;
}
```

dslings - 检测结果

```
🌍Progress: [==>----------] 2/12

[Target: 0.dslings-2]

✅ Successfully ran tests/dslings.2.cpp!

🎉   The code is compiling!   🎉

Output:
====================
[D2DS LOGI]: - ✅ | mVal.get() == 2 (2 == 2)
[D2DS LOGI]: - ✅ | mVal.get() == 2 (2 == 2)
[D2DS LOGI]: - ✅ | mVal.get() == 100 (100 == 100)
[D2DS LOGI]: - ✅ | mVal.get() == maxVal (191 == 191)
[D2DS LOGW]:    main: tests/dslings.2.cpp:46 - Delete the XLINGS_WAIT to
continue...

====================

Book: https://sunrisepeak.github.io/d2ds
```

代码接口介绍

MaxValue一个数据最大值检查器

MaxValue构造函数设置默认值

```
    d2ds::MaxValue mVal(2);
```

get函数获取当前最大值

```
    d2ds_assert_eq(mVal.get(), 2);
```

set函数设置一个值

如果当前最大值小于这个值则需要进行替换

```
    mVal.set(-1);
    d2ds_assert_eq(mVal.get(), 2);

    mVal.set(100);
    d2ds_assert_eq(mVal.get(), 100);
```

MaxVal的应用测试 - 获取最大数组中最大值

```
// random test
    dstruct::Array<int, 10> data;
    d2ds::random_data_generator(data, 0, 200);
    d2ds::ds_print(data);

    int maxVal = 0;
    for (int i = 0; i < data.size(); i++) {
        mVal.set(data[i]);
        if (data[i] > maxVal) {
            maxVal = data[i];
        }
    }

    d2ds_assert_eq(mVal.get(), maxVal);
```

数据结构接口实现

根据代码示例和接口描述来实现这个数据结构

类型定义

```
class MaxValue {
public:
    MaxValue(int val) : mMaxVal_e { val } { }

private:
    int mMaxVal_e;
};
```

get接口实现

```
class MaxValue {
public:
    //...
    int get() {
        return mMaxVal_e;
    }

private:
    int mMaxVal_e;
};
```

set接口实现

```
class MaxValue {
public:
    //...
    void set(int val) {
        if (val > mMaxVal_e) {
            mMaxVal_e = val;
        }
    }

private:
    int mMaxVal_e;
};
```

代码练习dslings

用dslings的**编译器驱动开发模式**来进行代码练习

代码下载

```
git clone --recursive git@github.com:Sunrisepeak/d2ds.git
```

安装xmake

```
sudo add-apt-repository ppa:xmake-io/xmake
sudo apt-get update
sudo apt-get install g++ gdb xmake make -y
```

dslings自动检测

在本地d2ds仓库的根目录执行如下命令

```
xmake dslings
```

程序就开始自动的测试/校验, 直到一个没有完成(或错误的)练习代码, 并给出对应的练习位置以及相关的错误信息提示

注

- 执行命令前, 请确保电脑已经配置了C++环境, 并安装了xmake构建工具
- 强烈建议使用vscode作为代码练习的编辑器, 这样dslings在控制台给出的练习代码路径, 只需要用**ctrl+鼠标左键**点击就可以自动转跳到目标位置

错误提示

```
🌍Progress: [>-----] 0/5

[Target: 0.dslings-0]

❌ Error: Compilation/Running failed for tests/dslings.0.cpp:

 The code exist some error!

Output:
====================
[ 25%]: cache compiling.release tests/dslings.0.cpp
error: tests/dslings.0.cpp:20:11: error: ‘MaxValue’ is not a member of ‘d2ds’
   20 |     d2ds::MaxValue mVal(2);
      |           ^~~~~~~~
In file included from /usr/include/c++/11/cassert:44,
                 from ./tests/common.hpp:6,
                 from tests/dslings.0.cpp:14:
tests/dslings.0.cpp:22:12: error: ‘mVal’ was not declared in this scope
   22 |     d2ds_assert_eq(mVal.get(), 2);
      |            ^~~~
  > in tests/dslings.0.cpp


====================

Book: https://sunrisepeak.github.io/d2ds
```

执行命令后dslings程序会停在最近的未完成的练习, 并会"实时"检测和这个练习相关的数据结构代码的实现。 我们可以根据dslings在控制台的输出找到对应的练习代码:

```
// dslings.0.cpp - readonly
//
// 描述:
//  通过实现一个MaxVal类型(保存最大值), 来介绍dslings的"编译器驱动开发"
//  即根据编译器的错误提示来完成这个训练流程的演示Demo, 并且通常为了降低难度会把一个'数据结构'的实现分成多个检测模块.
//  如: dslings.0.cpp dslings.1.cpp dslings.2.cpp
//
// 目标/要求:
//  - 不修改该代码检测文件
//  - 在exercises/dslings.hpp中完成你的代码设计
//  - 通过所有编译器检测 和 断言
//

#include "common/common.hpp"

#include "exercises/dslings.hpp"

int main() {

    d2ds::MaxValue mVal(2);

    d2ds_assert_eq(mVal.get(), 2);

    return 0;
}
```

根据对应的练习代码中给的描述和要求完成该练习, 过程中可以结合dslings在控制台的提示来进行相关数据结构练习的代码设计。当正确完成代码后, dslings就会更新控制的输出给出对应的提示。

代码通过提示

```
🌏Progress: [=>----] 1/5

[Target: 0.dslings-0]

✅ Successfully ran tests/dslings.0.cpp!

🎉   The code is compiling!   🎉

Output:
====================

====================

Book: https://sunrisepeak.github.io/d2ds
```

XLINGS_WAIT - dslings等待标志

当完整完成一个小节的练习的时候, dslings检测程序会进入等待状态, 并且打印出类似如下的提示消息

```
[D2DS LOGW]:    main: tests/dslings.2.cpp:46 - Delete the XLINGS_WAIT to
continue...
```

按照消息中给出的文件地址, 选择注释掉(或删除)程序中的 `XLINGS_WAIT` 标志, dslings就会进入下一个练习并开启自动检测

总结

好的, 到这里你应该已经了解了本书的叙述逻辑和结构 - **[数据结构使用 + 数据结构实现 + 对应代码练习]**。但该项目现任处于持续构建中(WIP), 依然存在相当多的问题。如果你在这个过程中你发现了一些项目的问题或自己遇到了一些问题, 欢迎到d2ds讨论区反馈和交流。那么下面可以开始你的**动手写数据结构**之旅了...

资源

开源电子书 + 代码练习 + 公开课 + 论坛讨论

开源电子书

在线阅读

书籍原始文件

代码练习

---

d2ds数据结构代码练习目录

d2ds数据结构代码检测项目录

公开课

课程主页

B站 - 视频列表

YouTube - Playlist

论坛

---

d2ds书籍 | dslings

d2ds课程内容讨论

参与贡献

---

d2ds - issues | task

d2ds-courses - issues | task

其他

---

show-your-code-2024

DStruct 数据结构模板库

d2ds-dslings | 代码练习

通过使用dslings自动化检测的**编译器驱动开发模式**来进行代码练习

代码下载

```
git clone --recursive git@github.com:Sunrisepeak/d2ds.git
```

安装xmake

linux/macos

使用bash执行tools目录下的安装脚本

```
bash tools/install.unix.sh
```

windows

执行tools目录下的安装脚本 或 直接双击运行

```
tools\install.win.bat
```

dslings使用流程

第一步: 开启代码检测

在本地d2ds仓库的根目录执行如下命令

```
xmake dslings
```

程序开始自动的测试/校验, 直到一个没有完成(或检测不通过)的练习代码。dslings会在控制台输出提示信息。如:

- 练习进度
- 练习的代码路径信息
- 编译期错误信息提示
- 运行时错误提示

注

- 执行命令前, 请确保电脑已经配置了C++环境, 并安装了xmake构建工具
- 建议使用vscode作为代码练习的编辑器, 用**ctrl+鼠标左键**点击路径就可以自动转跳到目标位置
- 由于vscode的C/C++插件会检测文件变化, 可以参考issue-5来避免卡顿

第二步: 根据dslings提示, 找到对应的练习代码

```
🌍Progress: [>-----------------------------] 0/29

[Target: 0.dslings-0]

❌ Error: Compilation/Running failed for tests/dslings.0.cpp:

 The code exist some error!

Output:
====================
[ 50%]: cache compiling.release tests/dslings.0.cpp
error: tests/dslings.0.cpp:20:11: error: ‘MaxValue’ is not a member of ‘d2ds’
   20 |     d2ds::MaxValue mVal(2);
      |           ^~~~~~~~
In file included from tests/dslings.0.cpp:14:
tests/dslings.0.cpp:22:20: error: ‘mVal’ was not declared in this scope
   22 |     d2ds_assert_eq(mVal.get(), 2);
      |                    ^~~~
./common/common.hpp:28:9: note: in definition of macro ‘d2ds_assert_eq’
   28 |     if (a != b) {\
      |         ^
  > in tests/dslings.0.cpp


====================

Homepage: https://github.com/Sunrisepeak/d2ds-courses
```

执行命令后dslings程序会停在最近的未完成的练习, 并会"实时"检测和这个练习相关的数据结构代码的实现。我们可以根据dslings在控制台的输出找到对应的练习代码:

```cpp
// dslings.0.cpp - readonly
//
// 描述:
//  通过实现一个MaxVal类型(保存最大值), 来介绍dslings的"编译器驱动开发"
//  即根据编译器的错误提示来完成这个训练流程的演示Demo, 并且通常为了降低难度会把一个'数据结构'的实现分成多个检测模块.
//  如: dslings.0.cpp dslings.1.cpp dslings.2.cpp
//
// 目标/要求:
//  - 不修改该代码检测文件
//  - 在exercises/dslings.hpp中完成你的代码设计
//  - 通过所有编译器检测 和 断言
//

#include "common/common.hpp"

#include "exercises/dslings.hpp"

int main() {

    d2ds::MaxValue mVal(2);

    d2ds_assert_eq(mVal.get(), 2);

    HONLY_LOGI_P("Hello D2DS!");

    XLINGS_WAIT

    return 0;
}
```

第三步: 阅读练习描述和要求并完成练习

根据对应的练习代码中给的描述和要求完成该练习, 过程中可以结合dslings在控制台的提示来进行相关数据结构练习的代码设计。当正确完成代码后, dslings就会更新控制的输出给出对应的提示

```
🌏Progress: [>----------------------------] 0/29

[Target: 0.dslings-0]

✅ Successfully ran tests/dslings.0.cpp!

🎉   The code is compiling!   🎉

Output:
====================
[D2DS LOGI]: - ✅ | mVal.get() == 2 (2 == 2)
[D2DS LOGI]: - Hello D2DS!
[D2DS LOGW]: main: tests/dslings.0.cpp:26 - 🥳 Delete the XLINGS_WAIT to
continue...

====================

Homepage: https://github.com/Sunrisepeak/d2ds-courses
```

第四步: 注释XLINGS_WAIT, 进入下一个练习

根据dslings在控制台的提示信息, 找到 `tests/dslings.0.cpp:26` , 并进行注释或者删除。dslings就会进入下一个练习并进行检测

```
int main() {

    d2ds::MaxValue mVal(2);

    d2ds_assert_eq(mVal.get(), 2);

    HONLY_LOGI_P("Hello D2DS!");

    XLINGS_WAIT

    return 0;
}
```

工具 | 快捷命令

xmake dslings

从指定练习开始检测, 支持模糊匹配

```
# xmake dslings 默认从第一开始检测
xmake dslings -s [target]
#xmake dslings -s vector
```

xmake d2ds

查看版本信息

```
xmake d2ds info
```

查看工具使用

```
xmake d2ds help
```

同步(主仓库)最新代码

```
xmake d2ds update
```

数组

数组在编程当中是最常用到的数据结构, 本章将会探讨关于定长数组Array和动态数组Vector核心部分的实现细节

定长数组Array

预览

Array是一个对原生数组轻量级封装的容器, 在性能于原生数组几乎相等的情况下, 携带了大小信息提供了类型安全可以避免许多由于数组边界问题导致的错误。同时Array也相对提供了更丰富的接口, 并且提供了接口标准化可能性, 方便使用数据结构的通用算法。

原生数组

```
void arr_init(int *arr, int size) {
    for (int i = 0; i < size; i++) {
        arr[i] = -1;
    }
}

int main() {
    int arr[10];
    arr_init(arr, 10);
    return 0;
}
```

Array数组

```
void arr_init(const Array<int, 10> &arr) {
    for (int i = 0; i < arr.size(); i++) {
        arr[i] = -1;
    }
}

template <typename IntArrayType>
void process(const IntArrayType &arr) {
    for (int &val : arr) {
        val *= 2;
    }
}

int main() {
    Array<int, 10> arr;
    arr_init(arr, 10);
    return 0;
}
```

从上面两个简单的例子, 在多数情况下**原生数组**的数据长度信息是需要开发人员额外记忆的, 这可能会引发一些潜在风险, 而对于**Array容器**的对象本身就会携带长度信息, 并且这个携带信息的代价不需要而外的存储空间。此外, 也有利用像 `process` 这种"通用型算法"的设计。换句话说, Array也能使用更多的通用数据结构算法。

> 注: Array使用栈区内存, Vector使用的是"动态分配"内存, 对于固定大小的数组使用Array要比Vector在性能上更有优势

Array核心实现

类型定义 - 固定类型模板参数

在Array的模板参数中的第二个参数上使用N来标识数组长度信息, 需要注意的是这里的N和类型参数T时有差异的, 它是一个 `unsigned int` 类型的**非类型模板参数**, 可以简单视为固定类型(指定类型)的信息标识, 在模板参数中就指定类型。

```
template <typename T, unsigned int N>
class Array {

};
```

数据初始化 - 列表初始化器的使用

数组的列表初始化, 在编程中非常常用和方便对数据做初始化的方式, 如下:

```
d2ds::Array<int, 5> intArr { 5, 4, 3, 2 /*, 1*/ };
```

对于自定义类型的Array模板, 要想支持这个特性可以使用 `initializer_list` 来获取列表中的数据, 并且使用迭代器进行数据的访问, 所以在Array中实现一个支持 `initializer_list` 的构造器即可

```
template <typename T, unsigned int N>
class Array {
public:
    Array(std::initializer_list<T> list) {
        int i = 0;
        for (auto it = list.begin(); it != list.end() && i < N; it++) {
            mData_e[i] = *it;
            i++;
        }
    }

private:
    T mData_e[N == 0 ? 1 : N];
};
```

注: 这里N存在为0的情况, 所以使用三目运算 `N == 0 ? 1 : N` 来保证数组长度至少为1

BigFive - 行为控制

BigFive核心指的是一个类型对象的**拷贝语义**和**移动语义**, 也常被称为**三五原则**(见 cppreference)。 多数情况下编译器是能自动生成这些代码的, 但作为一个库的话(特别是个数据结构容器库), 往往需要明确每个数据结构的行为, 这对数据结构中的数据复制和移动中的性能优化也是非常有帮助的。我们可以通过下面的类成员来实现其行为控制:

类成员	简述
`~ClassName()`	析构
`ClassName(const ClassName &)`	拷贝构造
`ClassName & operator=(const ClassName &)`	拷贝赋值
`ClassName(ClassName &&)`	移动构造

`ClassName & operator=(ClassName &&)`	移动赋值

析构行为

由于Array中也是使用原生数组来进行存储数据的, 这里使用使用默认的构造函数和析构和原生数组行为保持一致

```
template <typename T, unsigned int N>
class Array {
public: // bigFive
    Array() = default;
    ~Array() = default;
//...
};
```

拷贝语义

主要是方便数据结构中数据复制的方便

拷贝构造

```
d2ds::Array<BigFiveTest::Obj, 5> intArr1;
d2ds::Array<BigFiveTest::Obj, 5> intArr2(intArr1);
```

拷贝构造函数常用于"**一个同类型已存在的对象来初始化一个新对象**"的场景, 对于Array来说主要实现把已存在对象中的数据进行复制到新对象中即可

```
template <typename T, unsigned int N>
class Array {
public: // bigFive
//...
    Array(const Array &dsObj) {
        for (int i = 0; i < N; i++) {
            mData_e[i] = dsObj.mData_e[i];
        }
    }
//...
};
```

Note: 这里也可使用 `placement new` 来构造数据结构中的对象, 他的主要功能是把内存分配和对象构造进行分离--即在已有的内存上进行构造对象。更多关于**new/delete运算符的分析**将放到C++基础章节

拷贝赋值

```
d2ds::Array<BigFiveTest::Obj, 5> intArr1, intArr2;
// ...
intArr1 = intArr2;
```

通过赋值 = 运算符, 把一个Array中的数据复制到另一个数组中

```
template <typename T, unsigned int N>
class Array {
public: // bigFive
//...
    Array & operator=(const Array &dsObj) {
        D2DS_SELF_ASSIGNMENT_CHECKER
        for (int i = 0; i < N; i++) {
            mData_e[i] = dsObj.mData_e[i];
        }
        return *this;
    }
//...
};
```

移动语义

有些场景为了性能会使用**移动语义**, 只去改变数据的**所有权**来避免数据资源的重复制、频繁分配/释放带来的开销

移动构造

```
d2ds::Array<BigFiveTest::Obj, 5> intArr1;
//...
d2ds::Array<BigFiveTest::Obj, 5> intArr2 { std::move(intArr1) };
```

这里假设intArr1后续不在使用, 如果使用**拷贝构造**, 可能就会造成 `BigFiveTest::Obj` 中动态分配的资源没有必要的分配和复制。例如, 对象中有一个指针并指向一块资源, 使用移动构造去触发对象只复制资源的地址, 而不需要重新分配并做数据复制。**移动构造**中不仅要把数据结构的资源进行移动, 一些情况也要把**移动语义**传给直接管理的数据

```
template <typename T, unsigned int N>
class Array {
public: // bigFive
//...
    Array(Array &&dsObj) {
        for (int i = 0; i < N; i++) {
            mData_e[i] = std::move(dsObj.mData_e[i]);
        }
    }
//...
};
```

移动赋值

```
d2ds::Array<BigFiveTest::Obj, 5> intArr1, intArr2;
// ...
intArr1 = std::move(intArr2);
```

移动赋值和移动构造一样

```
template <typename T, unsigned int N>
class Array {
public: // bigFive
//...
    Array & operator=(Array &&dsObj) {
        D2DS_SELF_ASSIGNMENT_CHECKER
        for (int i = 0; i < N; i++) {
            mData_e[i] = std::move(dsObj.mData_e[i]);
        }
        return *this;
    }
//...
};
```

Note1: D2DS_SELF_ASSIGNMENT_CHECKER 宏是防止对象自我赋值情况的一个检测。例如:对象myObj赋值给自己(`myObj = myObj;`)。 该宏的实现原理(`if (this == &dsObj) return *this;`)是通过地址检查来规避这种情况

Note2: 对于Array的移动语义是不够直观的, 在Vector实现中有更直观的使用, 且更多关于BigFive-**拷贝语义**和**移动语义**的介绍将放到**C++基础章节**

数据访问 - 下标运算符重载

```
intArr[1] = 6;
intArr[4] = intArr[0];
```

实现类数组的下标访问的核心就是, 在对应类型中实现对下标运算符 `[]` 的重载

```
template <typename T, unsigned int N>
class Array {
public:
//...
    T & operator[](int index) {
        return mData_e[index];
    }
//...
};
```

常用函数实现

```
d2ds::Array<int, 5> intArr { 0, 1, 2, 3, 4 };
for (int i = 0; i < intArr.size(); i++) {
    d2ds_assert_eq(i, intArr[i]);
}
d2ds_assert_eq(4, intArr.back());
```

获取数据结构中的元素数量, 和获取最后一个元素都是常用的功能

size

```
template <typename T, unsigned int N>
class Array {
public:
    unsigned int size() const {
        return N;
    }
//...
};
```

back

```
template <typename T, unsigned int N>
class Array {
public:
    T back() const {
        return mData_e[N != 0 ? N - 1 : 0];
    }
//...
};
```

迭代器支持

对于Array的迭代器, 由于内部是使用数组来存储数据, 数据是在连续内存空间上的, 可以直接使用类型的指针作为迭代器类型来做迭代器支持和范围for的使用

```
template <typename T, unsigned int N>
class Array {
public:
    T * begin() {
        return mData_e;
    }

    T * end() {
        return mData_e + N;
    }
//...
};
```

Note: 关于范围for和迭代器相关的内容见**相关主体**部分

扩展功能 - 负下标访问

在里在给Array添加一个**扩展功能**, 像其他一些语言来支持负号下标访问

```
template <typename T, unsigned int N>
class Array {
public:
//...
    T & operator[](int index) {
        // add start
        if (index < 0)
            index = N + index;
        // add end
        d2ds_assert(index >= 0 && index < N);
        return mData_e[index];
    }
//...
};
```

总结

本章节介绍了固定长度的数据结构Array实现, 它既具备了原生数组的性能又具备了现代数据结构容器的安全性和扩展性。同时也介绍了很多来支持Array模板实现的技术和编程技巧: 列表初始化器、类行为控制、自定义下标访问等, 下一章我们将开始介绍不定长数组(动态数组)Vector的核心功能实现

动态数组Vector

预览

- 基本介绍
  - 定长数据
  - 变长数组
- Vector核心实现
  - 类型定义和数据初始化 - 自定义分配器支持
  - BigFive - 行为控制
  - 常用函数和数据访问
  - 数据增删和扩容机制 - resize
  - 迭代器支持 - 范围for
  - 功能扩展 - 向量加减法
- 总结

Vector是一个动态大小的数组, 元素存储在一个动态分配的连续空间。在使用中, 可以向Vector添加元素或删除数据结构中已有的元素, 内部会自动的根据数据量的大小进行扩大或缩小容量

手动管理

```
int main() {
    int *intArr = (int *)malloc(sizeof(int) * 2);
    intArr[0] = 1; intArr[1] = 2; // init

    // do something

    int *oldIntArr = intArr;
    intArr = (int *)malloc(sizeof(int) * 4);

    intArr[0] = oldIntArr[0]; intArr[1] = oldIntArr[1]; // copy
    free(oldIntArr);

    intArr[2] = 3;
    intArr[3] = 4;

    for (int i = 0; i < 4; i++) {
        std::cout << intArr[i] << " ";
    }
    std::cout << std::endl;
    for (int i = 0; i < 2; i++) {
        std::cout << intArr[i] << " ";
    }

    free(intArr);

    return 0;
};
```

自动管理

```
int main() {
    d2ds::Vector<int> intArr = { 1, 2 };

    intArr.push_back(3);
    intArr.push_back(4);

    for (int i = 0; i < 4; i++) {
        std::cout << intArr[i] << " ";
    }
    std::cout << std::endl;

    intArr.pop_back();
    intArr.pop_back();

    for (int i = 0; i < intArr.size() /* 2 */; i++) {
        std::cout << intArr[i] << " ";
    }

    return 0;
};
```

输出结果

```
1 2 3 4
1 2
```

上面使用 `Vector` 创建了一个 `intArr` 数组, 并在使用中通过 `push_back` 和 `pop_back` 改变了数组的长度, 而关于存储数据的内存的扩大和缩小全由 `Vector` 内部完成, 对使用者是"透明"的, 从而降低了开发者手动去管理内存分配的负担

Vector核心实现

类型定义和数据初始化

统一分配器接口

使用一个分配器类型作为作用域标识, 类型中包含两个静态成员函数用于内存的分配和释放

```
struct Allocator {
    static void * allocate(int bytes);
    static void deallocate(void *addr, int bytes);
};
```

其中 `allocate` 用于分配内存, 它的参数为请求的内存字节数; `deallocate` 用于内存的释放, addr为内存块地址, bytes为内存块的大小

类型定义

第一个模板参数用于接收数据类型, 第二个参数用于接收一个满足上面标准的分配器类型。为了方便使用, 使用 `DefaultAllocator` 作为分配器模板参数的默认类型, 这样开发者在不明确指定分配器的时候就会使用默认的分配器进行内存分配

```
template <typename T, typename Alloc = DefaultAllocator>
class Vector {

};
```

> 注: `DefaultAllocator` 是一个已经定义在 `d2ds` 命名空间的分配器。文件: common/common.hpp

数据初始化

```
d2ds::Vector<int> vec1;
d2ds::Vector<int> vec2(10);
d2ds::Vector<int> vec3 = { 1, 2, 3 };
```

定义数据成员, 并实现常见的默认初始化、指定长度的初始化、列表初始化器初始化

```
template <typename T, typename Alloc = DefaultAllocator>
class Vector {
public:

    Vector() : mSize_e { 0 }, mDataPtr_e { nullptr } { }

    Vector(int size) : mSize_e { size } {
        mDataPtr_e = static_cast<T *>(Alloc::allocate(sizeof(T) * mSize_e));
        for (int i = 0; i < mSize_e; i++) {
            new (mDataPtr_e + i) T();
        }
    }

    Vector(std::initializer_list<T> list) {
        mSize_e = list.end() - list.begin();
        mDataPtr_e = static_cast<T *>(Alloc::allocate(sizeof(T) * mSize_e));
        auto it = list.begin();
        T *dataPtr = mDataPtr_e;
        while (it != list.end()) {
            new (dataPtr) T(*it);
            it++; dataPtr++;
        }
    }

private:
    int mSize_e;
    T * mDataPtr_e;
};
```

定义一个 `mSize_e` 来标识元素数量, 使用 `Alloc` 进行内存分配来存储数据, 并由 `mDataPtr_e` 来管理。 同时配合使用**定位new**(placenment new)来完成数据的构造。这里把元素对象的创建划分成了两步: 第一步, 分配对应的内存; 第二步, 基于获得的内存进行构造对象

> 注: C++中使用 `new Obj()` 创建对象, 可以看作是 `ptr = malloc(sizeof(Obj)); new (ptr) Obj();` 这两步的组合。详情见深入理解new/delete章节

BigFive - 行为控制

析构行为

由于使用了内存分配和对象构造分离的模式, 所以在析构函数中需要对数据结构中的元素要先析构, 最后再释放内存。即需要满足如下构造/析构链, 让对象的创建和释放步骤对称:

- 分配对象内存A
- 基于内存A构造对象B
- 析构对象B
- 释放B对应的内存A

```
template <typename T, typename Alloc = DefaultAllocator>
class Vector {
public:
    ~Vector() {
        if (mSize_e) {
            for (int i = 0; i < mSize_e; i++) {
                (mDataPtr_e + i)->~T();
            }
            Alloc::deallocate(mDataPtr_e, mSize_e * sizeof(T));
        }
    }
}
```

拷贝语义

在拷贝构造函数中, 使用 `new (addr) T(const T &)` 把**拷贝构造语义**传递给数据结构中存储的元素

```
template <typename T, typename Alloc = DefaultAllocator>
class Vector {
public:
    Vector(const Vector &dsObj) : mSize_e { dsObj.mSize_e } {
        mDataPtr_e = (T *) Alloc::allocate(sizeof(T) * mSize_e);
        for (int i = 0; i < mSize_e; i++) {
            new (mDataPtr_e + i) T(dsObj.mDataPtr_e[i]);
        }
    }
}
```

在拷贝赋值函数中, 先调用析构函数进行数据清理, 同时也使用 `operator=` 进行语义传递

```
template <typename T, typename Alloc = DefaultAllocator>
class Vector {
public:
    Vector & operator=(const Vector &dsObj) {
        D2DS_SELF_ASSIGNMENT_CHECKER
        this->~Vector();
        mSize_e = dsObj.mSize_e;
        mDataPtr_e = static_cast<T *>(Alloc::allocate(sizeof(T) * mSize_e));
        for (int i = 0; i < mSize_e; i++) {
            mDataPtr_e[i] = dsObj.mDataPtr_e[i];
        }
        return *this;
    }
}
```

移动语义

在移动构造函数中, 只需要把要目标对象的资源移动到该对象, 然后对被移动的对象做重置操作即可。对于Vector来说, 只需进行**浅拷贝**数据成员, 并对被移动的对象置空

```
template <typename T, typename Alloc = DefaultAllocator>
class Vector {
public:
    Vector(Vector &&dsObj) : mSize_e { dsObj.mSize_e } {
        mDataPtr_e = dsObj.mDataPtr_e;
        // reset
        dsObj.mSize_e = 0;
        dsObj.mDataPtr_e = nullptr;
    }
}
```

在移动赋值函数中, 比移动构造多了对对象本身资源的释放操作

```
template <typename T, typename Alloc = DefaultAllocator>
class Vector {
public:
    Vector & operator=(Vector &&dsObj) {
        D2DS_SELF_ASSIGNMENT_CHECKER
        this->~Vector();
        mSize_e = dsObj.mSize_e;
        mDataPtr_e = dsObj.mDataPtr_e;
        // reset
        dsObj.mSize_e = 0;
        dsObj.mDataPtr_e = nullptr;
        return *this;
    }
}
```

常用函数和数据访问

常用函数 - size / empty

```
template <typename T, typename Alloc = DefaultAllocator>
class Vector {
public:
    int size() const {
        return mSize_e;
    }

    bool empty() const {
        return mSize_e == 0;
    }
}
```

数据访问

```
d2ds::Vector<int> intArr3 = { -1, -2, -3 };
const d2ds::Vector<int> constIntArr3 = { 1, 2, 3 };
```

Vector存在被 const 修饰的情况, 所以 operator= 也要对应实现一个 const 版本, 返回值为 const T &

```
template <typename T, typename Alloc = DefaultAllocator>
class Vector {
public:
    T & operator[](int index) {
        return mDataPtr_e[index];
    }

    const T & operator[](int index) const {
        return mDataPtr_e[index];
    }
}
```

数据增删 - 扩容和缓存机制

当动态数组Vector执行push操作进行添加元素时, 如果每次都需要重新分配内存这会极大的影响效率

```
void push(const int &obj) {
    newDataPtr = malloc(sizeof(int) * (size + 1)); // 分配内存
    copy(newDataPtr, oldDataPtr); // 复制数据
    free(oldDataPtr); // 释放内存
    newDataPtr[size + 1] = obj; // 添加新元素
    size++; // 数量加1
}
```

通过引入内存容量的缓存或者说预分配机制, 来避免过多的内存分配释放, 可以有效的降低它的影响。所以就需要引入另外一个标识 `mCapacity_e` 来标识当前内存最大容量, 而 `mSize_e` 用来标识当前数据结构中的实际元素数量, 所以 `mCapacity_e` 是大于等于 `mSize_e` 的

```
template <typename T, typename Alloc = DefaultAllocator>
class Vector {
private:
    int mSize_e, mCapacity_e;
    T * mDataPtr_e;
}
```

这里需要先说明一下, 扩容(缩容)机制通常是包含两个概念或步骤:

- 第一个是, 扩容(缩容)的条件, 也是执行实际操作的时机。通常扩容发生再数据增加操作, 缩容发生数据删除操作中
- 第二个是, 具体的扩容(缩容)规则。最简单的就是二倍扩容(缩容)

注: 成员变量的变动, 意味着对应的**BigFive**也需要修改

---

push_back 和 扩容

在每次扩容的时候, 可以选择基于当前容量的二倍进行扩容。例如: 当 `mCapacity_e` 等于4时, 做扩容时应该分配可以容纳8个元素的内存

```
d2ds::Vector<int> intArr = {0, 1, 2, 3};
intArr.push_back(4);
/*
old: mCapacity_e == 4, mSize_e == 4
               +---------------+
mDataPtr_e -> | 0 | 1 | 2 | 3 |
               +---------------+
new: mCapacity_e == 8, mSize_e == 5
               +-------------------------------+
mDataPtr_e -> | 0 | 1 | 2 | 3 | 4 |   |   |   |
               +-------------------------------+
*/
```

什么时候扩容? 最直观的是增加元素, 但容量又不够的时候。执行push_back时, 当 `mSize_e + 1 > mCapacity_e` 时就需要扩容来获取更大的空间用于新数据/元素的存放, 既是否扩容需要在存储新元素操作之前

```
template <typename T, typename Alloc = DefaultAllocator>
class Vector {
public:
    void push_back(const T &element) {
        if (mSize_e + 1 > mCapacity_e) {
            resize(mCapacity_e == 0 ? 2 : 2 * mCapacity_e);
        }
        new (mDataPtr_e + mSize_e) T(element);
        mSize_e++;
    }
}
```

pop_back 和 缩容

当数据量减少时, 同样需要释放过多的内存容量来避免内存浪费。这时就引入一个问题, 如果使用二倍原则, 是当数据结构中的真实数据量等于最大容量的1/2时进行重新分配吗? 考虑一下这样的场景:

```
d2ds::Vector<int> intArr = { 1, 2, 3, 4 };
for (int i = 0; i < 10; i++) {
    intArr.push_back(i); // 触发扩容
    // ...
    intArr.pop_back(); // 触发缩容
}
```

当频繁小数据量的增加和减少, 就会造成Vector内部不停的扩容和缩容操作, 这种现象也称为——**抖动**。

为了近可能的避免这种情况, 在执行缩容之后也应该保留/缓存一部分未使用的内存空间, 用于后续可能的数据增加操作。即扩容或者缩容都要保证一定的空闲内存, 用于后续可能的操作。如: 下面就是1/3触发条件, 2倍(1/2)扩容机制的内存变化情况

```
mCapacity_e == 8, mSize_e == 5
                +-------------------------------+
mDataPtr_e -> | 0 | 1 | 2 | 3 | 4 |   |   |   |
                +-------------------------------+

intArr.pop_back();

mCapacity_e == 8, mSize_e == 4
                +-------------------------------+
mDataPtr_e -> | 0 | 1 | 2 | 3 |   |   |   |   |
                +-------------------------------+

intArr.pop_back();

mCapacity_e == 8, mSize_e == 3
                +-------------------------------+
mDataPtr_e -> | 0 | 1 | 2 |   |   |   |   |   |
                +-------------------------------+

intArr.pop_back();

mCapacity_e == 4, mSize_e == 2
                +---------------+
mDataPtr_e -> | 0 | 1 |   |   |
                +---------------+
```

当 `mSize_e <= mCapacity_e / 3` 时就触发一次二倍扩容机制的执行, 把容量从8缩小一半到4, 此时实际存储的数据量 `mSize_e == 2` 。这里需要注意的是, 虽然 `pop_back` 不一定会释放Vector管理的内存, 但依然需要去调用被删除元素的析构函数去释放它额外管理的资源(如果存在)

```
template <typename T, typename Alloc = DefaultAllocator>
class Vector {
public:
    void pop_back() {
        mSize_e--;
        (mDataPtr_e + mSize_e)->~T();
        if (mSize_e <= mCapacity_e / 3) {
            resize(mCapacity_e / 2);
        }
    }
}
```

resize实现

对于resize的实现, 需要关注的核心点:

- 新老内存的分配和释放
- 老数据的迁移

首先进行分配一块能存n个元素的内存块, 然后在对数据进行迁移, 最后释放老的内存块。在进行数据迁移的过程中, 如果使用拷贝语义则需要通过**显式调用**析构进行释放老的内存, 如果使用移动语语义则可以避免**在所管理元素对象内部的资源的频繁分配释放**。为了能呈现主要骨架但有不过于复杂, 下面只实现了 `mSize_e <= n` 的情况的简化版本

```
template <typename T, typename Alloc = DefaultAllocator>
class Vector {
    void resize(int n) { // only mSize_e <= n
        auto newDataPtr = n == 0 ? nullptr : static_cast<T *>
(Alloc::allocate(n * sizeof(T)));

        for (int i = 0; i < mSize_e; i++) {
            new (newDataPtr + i) T(mDataPtr_e[i]);
            (mDataPtr_e + i)->~T();
        }

        if (mDataPtr_e) {
            // Note:
            //  memory-size is mCapacity_e * sizeof(T) rather than mSize_e *
sizeof(T)
            Alloc::deallocate(mDataPtr_e, mCapacity_e * sizeof(T));
        }

        mCapacity_e = n;
        mDataPtr_e = newDataPtr;
    }
}
```

迭代器支持

由于Vector用于存储数据元素的内存是连续的, 所以可以使用原生指针作为数据访问的迭代器

```
const d2ds::Vector<int> constIntArr = intArr;
int sum = 0;
for (auto &val : constIntArr) {
    sum += val;
}
```

为了让被 const 修饰的Vector, 可以正常使用迭代器访问数据, 所以可以再实现一套const版本的begin和end

```
template <typename T, typename Alloc = DefaultAllocator>
class Vector {
public:
    T * begin() {
        return mDataPtr_e;
    }

    T * end() {
        return mDataPtr_e + mSize_e;
    }

    const T * begin() const {
        return mDataPtr_e;
    }

    const T * end() const {
        return mDataPtr_e + mSize_e;
    }
};
```

功能扩展 - 向量加减法

假设有如下**OQ**、**OP**、**QP**三个向量

```
^
|   * P(2, 4)
|
|       *Q(4, 1)
*-------------->
O(0, 0)
```

```
d2ds::Vector<int> OQ = { 4, 1 };
d2ds::Vector<int> OP = { 2, 4 };
d2ds::Vector<int> QP = { -2, 3 };
d2ds_assert(OQ + QP == OP);
d2ds_assert(OP - OQ == QP);
```

下面通过重载 operator+ 和 operator- 来扩展下Vector再向量中的应用。这里为了直观我们直接假设向量是2维的, 在运算符重载函数中分别再实现向量的加减算法即可。怎么支持N维向量? 想必你心中已有答案

```cpp
namespace d2ds {

template <typename T>
bool operator==(const Vector<T> &v1, const Vector<T> &v2) {
    bool equal = v1.size() == v2.size();
    if (equal) {
        for (int i = 0; i < v1.size(); i++) {
            if (v1[i] != v2[i]) {
                equal = false;
                break;
            }
        }
    }
    return equal;
}

template <typename T>
Vector<T> operator+(const Vector<T> &v1, const Vector<T> &v2) {
    Vector<T> v(2);
    v[0] = v1[0] + v2[0];
    v[1] = v1[1] + v2[1];
    return std::move(v);
}

template <typename T>
Vector<T> operator-(const Vector<T> &v1, const Vector<T> &v2) {
    Vector<T> v(2);
    v[0] = v1[0] - v2[0];
    v[1] = v1[1] - v2[1];
    return std::move(v);
}

}
```

总结

本章节先是对比了一下, 对变长数组有需求的场景下。使用Vector自动管理内存和手动管理内存的差异和优势。然后，介绍了需要动态分配内存的数据结构如何去支持用户自定义分配的方法; 以及在内部自动管理内存的扩容机制的核心原理和对应**二倍扩容机制**的简单实现; 最后, 介绍了一个对Vector进行在向量领域的扩展应用。当然, 为了能够在呈现出动态数组Vector的核心原理下, 但又不过于复杂和拘迂细节, 本章中并没有去实现同样很常用的一些功能如: erase、back、data等。**但我相信在你学习完本章内容后的此时此刻, 你已基本具备自己去实现他们的能力**

链表

嵌入式单链表 - SinglyLink

预览

嵌入式链表是一种高性能且范型支持的链表。同时也是一种"底层"的数据结构。它在内存效率和性能上的优秀的表现, 使得在Linux内核、浏览器的v8引擎、Redis数据库等许多大型的开源项目中都有使用。对于大多数的数据结构实现, 关注的核心点可以归为**内存管理**、**类型控制**、**操作**这三个方面。通常这是库作者的工作, 而使用者只需要关心数据。而在嵌入式链表的使用中**内存管理**、**类型控制**是常需要使用者来显示控制的, 这使得它的使用难度远大于普通数据结构。这也是为什么它常应用到一些追求性能的系统模块, 而应用软件中却很少见到它的身影, 下面我们将从它的最小代码实现开始一步一步介绍其设计理念和使用技巧

结构 - 链域和数据域

```
struct ListNode {
    struct ListNode *next; // link区域
    int data; // data区域
};
```

一个链表节点可以分成**link区域**和**数据区域**。其中，数据区用于储存数据，link区域用于存储指向下一个节点的指针，把分散的节点串连成一个链表。

操作 - 链表的逻辑抽象

如果我们像上面一样, 把数据类型和链表进行耦合。就会发现, 每定义一个链表就**只能给特定的数据类型使用**, 很难实现通用数据结构。当然, 在C++中有很多方法来实现这种通用性。例如: 编

译器代码生成技术 - 模板

```
template <typename T>
struct ListNode {
    struct ListNode *next; // link区域
    T data; // data区域
};
```

但模板的本质就是需要手写两遍的代码量, 转为编译器来帮你手写了。从底层角度看**数据类型**和**链表**依然是耦合的, 并且C语言中是不支持模板的。对于链表的很多操作一定要和存储的数据类型进行绑定吗? 显然, 链表的操作从逻辑上是和数据类型无关的。例如下面把**数据区域**丢弃的链表代码:

```
struct SinglyLink {
    struct SinglyLink *next;
};

static void insert(SinglyLink *prev, SinglyLink *target) {
    target->next = prev->next;
    prev->next = target;
}

static void remove(SinglyLink *prev, SinglyLink *target) {
    prev->next = target->next;
    target->next = target;
}
```

这是不是链表? 是。但不存数据的链表有什么意义呢? 如果这时我说——这就是**嵌入式链表**的最小原型。你会不会产生如下疑问:

- 它的使用方法?
- 它存储和管理数据的原理?

下面我们将逐一回答

使用 - 数据存储和访问

开头的简介里也说了, 嵌入式链表只管理数据, 内存的分配和释放是由使用者完成的。这样只要节点中包含一个统一的**SinglyLink链接区域**, 所有节点就可以被组织起来

```
struct ListNodeInt {
    SinglyLink link; // link区域
    int data; // data区域
};

struct ListNodeDouble {
    SinglyLink link; // link区域
    double data; // data区域
};

int main() {
    ListNodeInt node1;
    auto node2Ptr = new ListNodeInt();

    insert(&(node1.link), (SinglyLink *)node2Ptr);

    auto linkPtr = (SinglyLink *)(&node1);
    while (linkPtr != nullptr) {
        auto nodePtr = (ListNodeInt *)linkPtr;
        std::cout << nodePtr->data << std::endl;
        linkPtr = linkPtr->next;
    }
    delete node2Ptr;
}
```

嵌入式链表, 可以**忽视**一个节点中除去link以外的数据。通过操作每个节点中link对链表做增加、删除和遍历的操作。在循环遍历链表时, link下面的数据类型的处理是交给使用这显示控制的, 即通过类型转换把link类型转为它本身的节点或数据类型, 进而区访问这个节点真实携带的数据信息。而这些数据的结构、大小等细节链表操作是不关心的, 它只关注对link区域的处理

优点

C语法实现通用链表

不需要使用复杂的代码生成技术和范型编程支持, 就可以实现高效的通用数据结构

性能更好

对于可变数据, 不使用二次分配内存的方式。link区域和data区域是位于同一块连续内存上, Cache更友好(相对两次分配)。同时相对于std::list在链表数据迁移的时候不需要额外释放和分配内存。

节点可位于多个链表

一个节点可以同时位于多个链表中。如: 一个Task节点可以同时位于eventList和runList中

```
struct Task {
    SinglyLink eventList;
    // ...
    SinglyLink runList;
};
```

设计/使用技巧

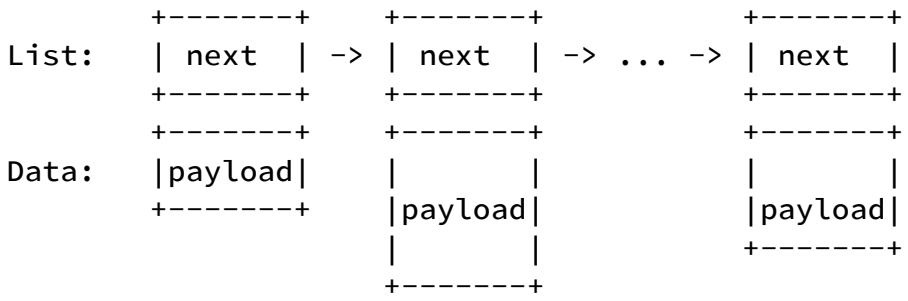
虽然上面介绍了嵌入式链表的总体设计思想——**只关心统一的链表操作**。而**数据类型**和**内存分配**的处理上会有些许不同, 下面就介绍三种经典的处理方法:

通信库 - 组合式

在很多消息通信的场景, 每个消息携带的数据量可能是不一样的。管理消息的链表, 可以通过组合的形式把Link域和变长数据域的内存做**物理拼接**

```
template <unsigned int N>
struct Msg {
    int size;
    char data[N];

    static void init(void *msg) {
        reinterpret_cast<Msg *>(msg)->size = N;
        // fill data
    }
};

int main() {
    auto node1 = (SinglyLink *) malloc(sizeof(SinglyLink) +
sizeof(Msg<1024>));
    auto node2 = (SinglyLink *) malloc(sizeof(SinglyLink) + sizeof(Msg<1024 *
3>));
    auto node3 = (SinglyLink *) malloc(sizeof(SinglyLink) + sizeof(Msg<1024 *
2>));

    Msg<1024>::init(node1 + 1);
    Msg<1024 * 3>::init(node2 + 1);
    Msg<1024 * 2>::init(node3 + 1);

    SinglyLink msg_list;

    insert(&msg_list, node1);
    insert(&msg_list, node2);
    insert(&msg_list, node3);

    //...

    free(node1);
    free(node2);
    free(node3);
    return 0;
}
```

在使用malloc分配内存时, 申请的大小是 `sizeof(link) + sizeof(data)` , 这样在物理上数据和只有link的嵌入式链表的节点在一块连续的内存上。在链表视角相当于 在每个节点下面挂载了一个**隐藏**的消息数据



这样可以使得每个链表节点的消息负载(长度)是可变的。数据的使用者解析时只需要对节点地址做1单位的偏移, 就能去解析数据的结构/格式。并由于link和data是通过一次申请来分配的, 所以在解析失败或节点释放的时候可以直接通过 `free(nodePtr)` 去释放内存块, 而不需要分两次释放

Linux内核 - 嵌入式

在最开始介绍设计思想的时候使用的就是者中把 `SinglyLink` "嵌入"到一个数据结构中的形式

```
struct MemBlock {
    SinglyLink link;
    int size;
};
```

由于link是被嵌入数据结构的第一个数据成员(无继承), 所以从link到MemBlock和MemBlock到link的转换都比较方便, 可以直接通过强制类型转换来实现

```
auto mbPtr = new MemBlock;
auto linkPtr = static_cast<SinglyLink *>(mbPtr);
//...
auto mbPtr_tmp = static_cast<SinglyLink *>(linkPtr);
```

但是在内核源码中常可以见到如下的**嵌入方式**

```
struct Demo {
    char member1;
    int member2
    SinglyLink link;
    double member3;
};
```

link成员并不是结构的第一个成员, 这个时候怎么做地址和类型的相互转换呢?

成员偏移量计算

```
#define offset_of(Type, member) ((size_t) & ((Type *)0)->member)

int main() {
    auto offset = offset_of(Demo, link);
    /*
    demoPtr = (Demo *)0;
    linkPtr = &(demoPtr->link);
    offset = (size_t) linkPtr - 0;
    */
    return 0;
}
```

这里通过把0转为一个Demo指针类型, 在取这个对象中link的地址。由于对象的基地址是0, 那么Demo中link成员的地址就是该成员相对Demo对象首地址的偏移量。由于中间设计到对空指针的操作, 虽然没有访问数据。但有些编译器上可能不生效或产生未知行为, 我们可以按照这个思路把0换成一个有效地址即可

```
#define offset_of(Type, member) \
[]() { Type t; return ((size_t) & (&t)->member - (size_t)&t); } ()

int main() {
    auto offset = offset_of(Demo, link);
    /*
    auto offset_of_func = []() {
        Demo t;
        Demo ptr = &t;
        baseAddr = (size_t)ptr;
        memberAddr = (size_t) & (ptr->link);
        return memberAddr - baseAddr;
    };
    offset = offset_of_func();
    */
    return 0;
}
```

以上只是为了解释**成员偏移量计算**的原理和可行性, 实际上编译器内部已经帮我们实现了这样的"函数", 我们直接使用就可以了

```
#include <cstddef>
size_t offset = offsetof(Demo, link);
```

通用类型转换操作

```
#define to_link(nodePtr) (&((nodePtr)->link))
#define to_node(linkPtr, NodeType, member) \
    (NodeType *)( (size_t)linkPtr - offset_of(NodeType, member) )
```

通过node指针转向link指针可以直接通过取成员地址实现, 而通过link指针**还原为原来的node指针**则需要上面介绍的**成员偏移量计算**工具的辅助。即: link指针 - link成员偏移量 -> 原数据结构地址

```
int main() {
    Demo nodeList;
    Demo node1, node2, node3

    insert(to_link(&nodeList), to_link(&node1));
    insert(to_link(&nodeList), to_link(&node2));
    insert(to_link(&nodeList), to_link(&node3));

    // ...

    SinglyLink *p = nodeList.link.next;
    while (p != to_link(&nodeList)) {
        Demo *demo = to_node(p, Demo, link);
        std::cout << demo->member1 << std::endl;
        p = p->next;
    }
    return 0;
}
```

V8引擎 - 继承式(C++)

继承的好处是, 子类指针可以向上转型, 自动完成从实际数据结构到link的转换。同时在link中通过模板来携带类型信息(并未有额外定义)。通过直接继承SinglyLink(编译期多态)来使用链表功能

```
template <typename T>
struct SinglyLink {
    struct T *next;
};

struct Demo : public SinglyLink<Demo> {
    int a;
    double b;
};
```

这里的SinglyLink里只使用了Demo类型的指针, 并不会引起循环定义问题。

```
int main() {

    SinglyLink<Demo> demoList;
    Demo d1, d2, d3

    insert(&demoList, &d1);
    insert(&demoList, &d2);
    insert(&demoList, &d3);

    for (auto it = demoList.next; it != nullptr; it = it->next) {
        std::cout << it->a << std::endl;
    }

    return 0;
}
```

这样实现的链表, 在节点遍历和数据访问上实现了一定的统一, 避免了使用to_link和to_node等类型转换操作

总结

本章节介绍了嵌入式(侵入式)链表的核心原理, 及常见的几种实现/使用方式。虽然嵌入式链表在性能和控制力上有一些优势, 但其使用上的复杂度对开发人员确是一种"负担"。所以, 它常出现在追求性能的系统编程场景, 而对于应用软件的开发, 往往封装度完整的链表(如:std::list)会更加适合。总的来说, 数据结构的选择是一个 `trade-off` 权衡的结果

单链表 - SLinkedList

预览

单链表是一个内存离散型的数据结构。由于存储的每一个元素都是一个独立的内存块, 所以在插入和删除元素时的复杂度是O(1)。因此, 链表常用于需要频繁进行插入/删除的场景, 来避免其他元素移动/拷贝的开销。

数组 VS 链表

\	数组	链表
内存	连续	离散
插入	O(n)	O(1)
删除	O(n)	O(1)

链表相对于数组的优势不仅仅**插入/删除**的操作非常快, 而且由于链表的内存是离散的, 所以在增加元素扩容的时候不需要像数组一样需要一个连续的存下所有的元素的大内存块, 只要能满足每个元素的内存大小即可。这样就一定程度的避免了无法使用小内存块的问题

SLinkedList核心实现

类型及数据成员定义

使用两个模板参数一个作为存储的数据类型, 一个作为分配节点的内存分配器。并且定义一个SLinkedListNode模板作为存储数据的链表节点: 它的第一个成员next表示指向下一个节点, 第二个成员data为模板的第一个参数类型, 用来存储实际的数据。同时在数据结构中使用using给节点起一个Node别名方便后续使用

```
template <typename T>
struct SLinkedListNode {
    SLinkedListNode *next;
    T data;
};

template <typename T, typename Alloc = DefaultAllocator>
class SLinkedList {
    using Node = SLinkedListNode<T>;
private:
    int mSize_e;
    Node mHead_e;
    Node *mTailPtr_e;
};
```

链表的数据成员中, 用Node类型定义一个链表的头节点, 方便对链表的管理。同时使用一个int类型的mSize_e来记录链表的长度, 避免每次都遍历链表来求解长度。最后, 为了单链表的尾插法的实现再加一个指向最后一个元素的节点指针mTailPtr_e。

数据成员的初始化

默认初始化

```
d2ds::SLinkedList<int> intList;
```

默认初始化对应的构造函数, 需要完成数据成员的初始化。这里构造函数后, 初始化列表的形式对成员进行初始化。其中头节点mHead_e中的next指针和mTailPtr_e初始化为自己/头节点 -- 循环单链表

```
template <typename T, typename Alloc = DefaultAllocator>
class SLinkedList {
public:
    SLinkedList() : mSize_e { 0 }, mHead_e { &mHead_e, T()}, mTailPtr_e {
&mHead_e } {

    }
};
```

列表初始化

```
d2ds::SLinkedList<int> intList = { 1, 2, 3 };
```

实现链表的列表初始化形式, 需要实现对应的插入操作 - push_back

```
template <typename T, typename Alloc = DefaultAllocator>
class SLinkedList {
public:
    void push_back(const T &data) {

        auto nodePtr = static_cast<Node *>(Alloc::allocate(sizeof(Node)));

        new (&(nodePtr->data)) T(data);
        nodePtr->next = &mHead_e;

        mTailPtr_e->next = nodePtr;
        mTailPtr_e = nodePtr;

        mSize_e++;
    }
};
```

push_back的实现主要分4个步骤

- 1.使用Alloc, 分配一块节点大小的内存
- 2.使用传进来的data对节点进行初始化, 并把节点的next指针指向头节点
- 3.修改尾节点的next指向新节点, 并更新mTailPtr_e指向新的尾部节点
- 4.size增加1

```
template <typename T, typename Alloc = DefaultAllocator>
class SLinkedList {
public:
    SLinkedList(std::initializer_list<T> list) : SLinkedList() {
        for (auto it = list.begin(); it != list.end(); it++) {
            push_back(*it);
        }
    }
};
```

有了push_back实现后, 只需要遍历list中的元素并使用push_back到最后即可

BigFive - 行为控制

析构实现

析构函数主要是释放所有数据里的资源, 以及每个节点对应的内存块。每次都删除第一个节点, 直到链表为空 `mHead_e.next == &mHead_e`

```
template <typename T, typename Alloc = DefaultAllocator>
class SLinkedList {
public:
    ~SLinkedList() {
        while (mHead_e.next != &mHead_e) {
            Node *nodePtr = mHead_e.next;
            mHead_e.next = nodePtr->next;
            nodePtr->data.~T();
            Alloc::deallocate(nodePtr, sizeof(Node));
        }
        mSize_e = 0;
        mTailPtr_e = &mHead_e;
    }
};
```

拷贝语义

由于拷贝语义主要是资源的复制。拷贝构造的实现主要就是使用for遍历目标对象, 并使用push_back一个一个添加数据

```
template <typename T, typename Alloc = DefaultAllocator>
class SLinkedList {
public:
    SLinkedList(const SLinkedList &dsObj) : SLinkedList() {
        for (Node *nodePtr = dsObj.mHead_e.next; nodePtr != &
(dsObj.mHead_e);) {
            push_back(nodePtr->data);
            nodePtr = nodePtr->next;
        }
    }

    SLinkedList & operator=(const SLinkedList &dsObj) {
        if (this != &dsObj) {
            this->~SLinkedList();
            for (Node *nodePtr = dsObj.mHead_e.next; nodePtr != &
(dsObj.mHead_e);) {
                push_back(nodePtr->data);
                nodePtr = nodePtr->next;
            }
        }
        return *this;
    }
};
```

拷贝赋值的主要实现思路和拷贝构造是一样的, 只需要额外做自赋值检查以及资源的提前释放即可

移动语义

移动语义的实现的核心是转移链表的管理权, 而链表的管理的核心是:

- mHead_e.next : 第一个节点的地址
- mTailPtr_e : 最后一个节点的地址
- mSize_e : 节点数量

```
template <typename T, typename Alloc = DefaultAllocator>
class SLinkedList {
public:
    SLinkedList(SLinkedList &&dsObj) : SLinkedList() {
        mHead_e.next = dsObj.mHead_e.next;
        mTailPtr_e = dsObj.mTailPtr_e;
        mSize_e = dsObj.mSize_e;
        mTailPtr_e->next = &mHead_e; // Note: update tail node

        // reset
        dsObj.mHead_e.next = &(dsObj.mHead_e);
        dsObj.mTailPtr_e = &(dsObj.mHead_e);
        dsObj.mSize_e = 0;
    }

    SLinkedList & operator=(SLinkedList &&dsObj) {
        if (this != &dsObj) {
            this->~SLinkedList();
            mHead_e.next = dsObj.mHead_e.next;
            mTailPtr_e = dsObj.mTailPtr_e;
            mSize_e = dsObj.mSize_e;
            mTailPtr_e->next = &mHead_e;

            // reset
            dsObj.mHead_e.next = &(dsObj.mHead_e);
            dsObj.mTailPtr_e = &(dsObj.mHead_e);
            dsObj.mSize_e = 0;
        }
        return *this;
    }
};
```

不需要额外的内存分配和释放, 直接转移链表节点的地址。然后在把目标dsObj中的数据重置。这里有一个要注意的点是: **由于是循环单链表, 所以需要更新最后一个节点/尾节点的next指向新的头节点**

常用函数

size/empty

size的实现可以直接返回mSize_e的记录, 避免重新遍历链表进行计算。而empty直接判断头节点的next是不是指向自己即可

```
template <typename T, typename Alloc = DefaultAllocator>
class SLinkedList {
public:
    int size() const {
        return mSize_e;
    }

    bool empty() const {
        return mHead_e.next == &mHead_e;
    }
};
```

front/back

访问第一个元素和最后一个元素可以直接通过头节点的next指针和尾指针来直接访问, 时间复杂度为O(1)

```
template <typename T, typename Alloc = DefaultAllocator>
class SLinkedList {
public:
    T & front() {
        return mHead_e.next->data;
    }

    T & back() {
        return mTailPtr_e->data;
    }
};
```

数据访问

```
d2ds::SLinkedList<int> intList = { 0, 1, 2, 3, 4, 5 };
for (int i = 0; i < intList.size(); i++) {
    d2ds_assert_eq(intList[i], i);
}
```

由于链表内存是离散的, 所以不支持随机访问。所以通过索引的方式访问元素的算法复杂度为O(n)。即每次都是通过遍历链表来找到对应位置的数据再返回(**链表数据结构一般不提供直接的索引访问方式**)。这里我们通过**模拟随机访问的方式**, 使用下标运算符来进行数据访问。

```
template <typename T, typename Alloc = DefaultAllocator>
class SLinkedList {
public:
    T & operator[](int index) {
        // d2ds_assert(index < mSize_e && mSize != 0);
        Node *nodePtr = mHead_e.next;
        for (int i = 0; i < index; i++) {
            nodePtr = nodePtr->next;
        }
        return nodePtr->data;
    }
};
```

数据增删

push_back/_pop_back

通过这两个操作, 可以实现再数据结构尾部进行增删数据。其中push_back在最开始已经实现了。而对于单链表来说, 可以使用尾部指针来优化push_back的复杂度到O(1)。 而_pop_back的实现要麻烦很多。由于删除一个节点需要找到它的前一个节点, 所以需要先遍历链表找到最后一个节点的前一个节点。然后, 再删除尾节点并把内存给释放了。步骤可以总结如下:

- 1.找到尾节点的前一个节点
- 2.从链表中删除节点
- 3.释放节点的数据和对应的内存
- 4.size减1并更新尾部节点

```
template <typename T, typename Alloc = DefaultAllocator>
class SLinkedList {
public:
    void _pop_back() {
        // assert(size() > 0);
        Node *nodePtr = &mHead_e;
        while (nodePtr->next != mTailPtr_e) {
            nodePtr = nodePtr->next;
        }
        // delete mTailPtr_e from list
        nodePtr->next = &mHead_e;
        // release
        mTailPtr_e->data.~T();
        Alloc::deallocate(mTailPtr_e, sizeof(Node));
        mSize_e--;

        mTailPtr_e = nodePtr; // update
    }
};
```

注: 可以重新实现mTailPtr_e, 让其指向倒数第二个节点来实现

---

push_front/pop_front

push_front的实现流程和push_back的实现类似, 只是插入节点的前置节点换成了头节点, 并且在push数据的时候要更新一下尾指针指向该节点

```
template <typename T, typename Alloc = DefaultAllocator>
class SLinkedList {
public:
    void push_front(const T &data) {

        auto nodePtr = static_cast<Node *>(Alloc::allocate(sizeof(Node)));
        new (&(nodePtr->data)) T(data);

        nodePtr->next = mHead_e.next;
        mHead_e.next = nodePtr;
        mSize_e++;

        if (mSize_e == 1) {
            mTailPtr_e = nodePtr;
        }
    }

    void pop_front() {
        Node *nodePtr = mHead_e.next;
        // delete from list
        mHead_e.next = nodePtr->next;
        // release
        nodePtr->data.~T();
        Alloc::deallocate(nodePtr, sizeof(Node));
        mSize_e--;

        if (mSize_e == 0) {
            mTailPtr_e = &mHead_e; // update
        }
    }
};
```

pop_front的实现相对于_pop_back, 减少了前置节点的查找过程。所以它实现的算法复杂度是O(1).通过头节点的next指针找到要删除的节点, 并把next更新到删除节点的下一个节点, 然后释放目标节点。同时, 这里也需要在删除到链表为空时, 重置mTailPtr_e尾部节点指针

SLinkedList迭代器实现

迭代器是一种访问数据的设计模式 - 它把数据的访问抽象成统一的操作/行为。如:

- `*` : 取数据
- `->` : 访问成员
- `++` : 移动到下一个数据
- `--` : 返回上一个数据

迭代器类型定义

在迭代器中定义一个指向目标节点的指针, 用来访问数据以及迭代到下一个数据。所以它的构造函数的输入只需要目标节点的指针即可

```
template <typename T>
struct SLinkedListIterator {
    using Node = SLinkedListNode<T>;
    SLinkedListIterator() : mNodePtr { nullptr } { }
    SLinkedListIterator(Node *nodePtr) : mNodePtr { nodePtr } { }
    Node *mNodePtr;
};
```

同时在SLinkedList定义一个数据结构对应的具体类型(T)的迭代器别名, 方便后面的使用

```
template <typename T, typename Alloc = DefaultAllocator>
class SLinkedList {
public:
    using Iterator = SLinkedListIterator<T>;
}
```

数据访问与类指针行为

```
struct MyObj {
    char a;
    int b;
    float c;
};
MyObj obj {'a', 1, 1.1};

d2ds::SLinkedListNode<MyObj> node;
node.data = obj;

d2ds::SLinkedList<MyObj>::Iterator iterator(&node);
d2ds_assert_eq(iterator->a, obj.a);
d2ds_assert_eq(iterator->b, obj.b);
d2ds_assert_eq(iterator->c, obj.c);

d2ds_assert_eq(*(iterator).c, 1.1f);
```

迭代器的本质是一个类, 但它使用起来就像是所管理数据类型的指针。可以通过 `->` 运算符访问对应的成员数据。同时也可以通过 `*` 元算符号访问到对象

```
template <typename T>
struct SLinkedListIterator {
    T * operator->() {
        return &(mNodePtr->data);
    }

    T & operator*() {
        return mNodePtr->data;
    }
};
```

实现这种类指针的行为, 只需要重载并实现 `operator*` 和 `operator->`

- `operator*` : 返回数据的引用
- `operator->` : 返回数据的指针

迭代器的判等

```
d2ds::SLinkedListNode<int> node;
d2ds::SLinkedList<int>::Iterator iterator1(&node);
d2ds::SLinkedList<int>::Iterator iterator2(&node);
d2ds::SLinkedList<int>::Iterator iterator3(nullptr);

d2ds_assert(iterator1 == iterator2);
d2ds_assert(iterator2 != iterator3);
```

对于单链表的迭代器, 不用通过判断节点中的数据是否相等, 而是通过直接判断迭代器中管理的节点地址是否相等, 这样即可以判断数据有可以判断是否属于同一个链表

```
template <typename T>
struct SLinkedListIterator {
    bool operator==(const SLinkedListIterator &it) const {
        return mNodePtr == it.mNodePtr;
    }

    bool operator!=(const SLinkedListIterator &it) const {
        return mNodePtr != it.mNodePtr;
    }
};
```

向前迭代/++操作

```
d2ds_assert(++iterator1 == iterator3);
d2ds_assert(iterator2++ != iterator3);
d2ds_assert(iterator2 == iterator3);
```

单链表只有一个next指向下一个节点, 所以对应的迭代器也只能向前迭代。所以这里只需要实现迭代器的++操作

- `Self & operator++()` : 对应的是前置++
- `Self operator++(int)` : 对应的是后置++

```
template <typename T>
struct SLinkedListIterator {
    SLinkedListIterator & operator++() {
        mNodePtr = mNodePtr->next;
        return *this;
    }

    SLinkedListIterator operator++(int) {
        auto old = *this;
        mNodePtr = mNodePtr->next;
        return old;
    }
};
```

其中前置++操作的实现, 只需要修改节点指针让其指向下一个节点, 然后返回自己。而由于后置++的特性(在下一条语句中才生效, 当前语句不变), 所以需要先保留一份旧数据, 然会更新mNodePtr指向下一个节点。最后返回旧数据, 这样就能实现 - **在下一条语句生效的性质**

支持范围for

```
d2ds::SLinkedList<int> intList = { 5, 4, 3, 2, 1 };

auto it = intList.begin();

d2ds_assert_eq(*it, 5);
it++; d2ds_assert_eq(*it, 4);
it++; d2ds_assert_eq(*it, 3);
it++; d2ds_assert_eq(*it, 2);
it++; d2ds_assert_eq(*it, 1);

d2ds_assert(++it == intList.end());

int tmp = 5;
for (auto val : intList) {
    d2ds_assert_eq(val, tmp);
    tmp--;
}
```

在有了迭代器后, 就可以通过实现数据结构的begin/end来支持范围for循环

```
template <typename T, typename Alloc = DefaultAllocator>
class SLinkedList {
public:
    Iterator begin() {
        return mHead_e.next;
    }

    Iterator end() {
        return &mHead_e;
    }
};
```

begin返回第一个节点对应的迭代器, 由于 `Iterator` 的构造函数接受Node类型的指针, 所以直接返回节点的指针就可以自动匹配返回值的迭代器类型。而end迭代器是指向最后节点的下一个节点 -- 头节点。用含头节点的地址的迭代器标识结束

迭代器版本 - 插入删除操作

```
d2ds::SLinkedList<int> intList = { 5, 4, 3, 2, 1 };
auto it = intList.begin();
intList.erase_after(it);
++it; d2ds_assert_eq(*it, 3);
for (int val : intList) {
    std::cout << " " << val;
}
/*
[D2DS LOGI]: - ✅ | *it == 3 (3 == 3)
 5 3 2 1
*/
```

由于单链表的节点只有next指针的限制, 只能删除当前迭代器的下一个节点。所以只能实现erase_after/insert_after。并且他们的实现步骤和push/pop操作的流程是一样的, 不一样的是前驱节点由传入的迭代器中获取的

```
template <typename T, typename Alloc = DefaultAllocator>
class SLinkedList {
public:
    void erase_after(Iterator pos) {
        // assert(pos.mNodePtr->next != &mHead_e);
        Node *nodePtr = pos.mNodePtr->next;
        pos.mNodePtr->next = nodePtr->next;

        nodePtr->data.~T();
        Alloc::deallocate(nodePtr, sizeof(Node));
        mSize_e--;

        if (pos.mNodePtr->next == &mHead_e) {
            mTailPtr_e = pos.mNodePtr->next;
        }
    }

    void insert_after(Iterator pos, const T &data) {
        auto nodePtr = static_cast<Node *>(Alloc::allocate(sizeof(Node)));
        new (&(nodePtr->data)) T(data);

        nodePtr->next = pos.mNodePtr->next;
        pos.mNodePtr->next = nodePtr;
        mSize_e++;

        if (nodePtr->next == &mHead_e) {
            mTailPtr_e = nodePtr;
        }
    }
};
```

总结

本章先是介绍了链表和数组的区别, 然后又从单链表的节点定义定义开始开始一步一步实现数据初始化、拷贝/移动语义、常用的函数、以及单链表一般不直接提供的功能(pop_back/operator[]), 从实现的角度来解释为什么不提的原因。最后, 又介绍了链表数据结构对应的迭代器的实现。其中关键的是对各种运算符(`*` / `->` /...)进行重载来实现类指针的行为。从单链表的实现上看, 它自身具有一定的限制在给定一个节点只能向下寻找, 这限制了迭代器的--操作的实现。并且不能直接删除当前节点(需要前驱节点)。下一章我们将开始介绍链表中最常使用的双链表的实现, 它能很好解决单链表的一些问题。

嵌入式双链表 - DoublyLink

预览

前面介绍过了**嵌入式单链表**, 但其并不支持向前查找。本章节就将介绍链表中最常用的双链表对应的"无数据域"的 -- **嵌入式双链表**

内存使用对比

在双链表中, 每一个节点都比单链表多了一个指向前一个节点的指针 `prev` , 所以对应的在存储相同数据量的元素N时, 内存的使用要比单链表多N x 8字节(设:指针占8字节)。但多数时候大家都原因花费这个内存的代价来换取使用的便利

```
struct SinglyLink {
    struct SinglyLink *next;
};

struct DoublyLink {
    struct DoublyLink *prev, *next;
};
```

单链表的局限性

双链表引入的这个 `prev` 指针就是为了解决单链表无法向前查找的问题, 这里我们可以用删除当前节点的操作来演示使用场景

- target: 我们需要删除的节点
- del: 链表删除操作

```
SinglyLink::del(prevNodePtr ?, &target);
DoublyLink::del(target.prev, &target);
```

通过代码可以直观的看出, 由于**从链表中删除节点需要知道该节点的前一个节点**, 所以从单链中删除target, 而找到target的前一个节点并不能很高效的完成。对于双链表只需要O(1)的复杂度。

DoublyLink核心实现

采用节点 + 静态成员函数的方式进行定义DoublyLink

定义和初始化

只包含两个指针, 一个指向前驱节点, 一个指向后继节点。节点的初始化就是把这两个指针指向自己。可以把这样的节点视为一个没有数据的链表头结点, 或一个不在链表中的节点

```
struct DoublyLink {
    struct DoublyLink *prev, *next;

    static void init(struct DoublyLink *target) {
        target->prev = target->next = target;
    }
};
```

插入和删除操作

双链表的插入操作, 需要处理三个节点中的4个指针。它们分别是:

- target的next指针 - 指向prev节点的next
- target的prev指针 - 指向prev节点
- prev的下一个节点的prev指针 - 指向target
- prev的next指针 - 指向target

```
struct DoublyLink {
    static void insert(DoublyLink *prev, DoublyLink *target) {
        target->next = prev->next;
        target->prev = prev;
        prev->next->prev = target;
        prev->next = target;
    }

    static void del(DoublyLink *prev, DoublyLink *target) {
        prev->next = target->next;
        target->next->prev = prev;
    }
};
```

双链表的删除操作, 只需要更新删除节点的

- 前一个节点的next指针 - 指向target的下一个节点
- 后一个节点的prev指针 - 指向target的前一个节点

> 注: 对于双链表的del接口, 其实可以不用显示的传前一个节点的指针prev。但这里为了和单链表的节点格式进行统一保留了prev。等价形式:
>
> - del(target)
> - del(target->prev, target)

逆序遍历

通过prev指针, 从后向前遍历将变的异常容易。如下的逆序遍历for循环的代码框架几乎和向后遍历一致的

```
// Node : linkPtr = malloc(sizeof(Link) + sizeof(Data))
for (auto linkPtr = head.prev; linkPtr != &head; linkPtr = linkPtr->prev) {
    MyData *dataPtr = reinterpret_cast<MyData *>(linkPtr + 1);
    // do something
}
```

使用示例

在**嵌入式单链表**章节从设计者视角详细的介绍了常用的使用示例。本章将中从实用角度出发来介绍。通过使用**组合式、嵌入式、继承式**的方式来实现同样一个功能来进行对比

组合式

```
// embedded-dlist.2.cpp - write
//
// 描述:
//   嵌入式双链表 - 组合式
//
// 目标/要求:
//  - 在对应的SHOW_YOUR_CODE代码块实现 逆序遍历 和 链表的释放
//  - 通过所有编译器检测 和 断言
//

#include "common/common.hpp"

#include "exercises/linked-list/EmbeddedList.hpp"

using d2ds::DefaultAllocator;

struct MyData {
    int id;
    char data;
};

int main() {

    d2ds::DefaultAllocator::debug() = true;

    d2ds::DoublyLink head;

    d2ds::DoublyLink::init(&head);

    for (int i = 0; i < 10; i++) {
        auto linkPtr = ( d2ds::DoublyLink* )
d2ds::DefaultAllocator::malloc(sizeof(d2ds::DoublyLink) + sizeof(MyData));
        d2ds::DoublyLink::init(linkPtr);
        auto dataPtr = (MyData *)(linkPtr + 1);
        dataPtr->id = i;
        dataPtr->data = 'a' + i;
        d2ds::DoublyLink::insert(&head, linkPtr);
    }

    dstruct::Stack<MyData> dataStack;
    for (auto linkPtr = head.next; linkPtr != &head; linkPtr = linkPtr->next)
{
```

```
        auto dataPtr = reinterpret_cast<MyData *>(linkPtr + 1);
        dataStack.push(*dataPtr);
    }

    SHOW_YOUR_CODE({ // reverse traverse
        for (auto linkPtr = head.prev; linkPtr != &head; linkPtr = linkPtr->prev) {
            MyData *dataPtr = reinterpret_cast<MyData *>(linkPtr + 1);

            DONT_CHANGE(
                auto myData = dataStack.top();
                d2ds_assert_eq(dataPtr->id, myData.id);
                d2ds_assert_eq(dataPtr->data, myData.data);
                dataStack.pop();
            )
        }
    })

    d2ds_assert(dataStack.empty());

    SHOW_YOUR_CODE({ // use DefaultAllocator::free(addr) to release
        d2ds::DoublyLink *target = head.next;
        while (target != &head) {
            d2ds::DoublyLink::del(&head, target);
            DefaultAllocator::free(target);
            target = head.next;
        }
    })

    d2ds_assert(head.next == &head);

    XLINGS_WAIT

    return 0;
}
```

嵌入式

```
// embedded-dlist.3.cpp - write
//
// 描述:
//   嵌入式双链表 - 嵌入式
//
// 目标/要求:
//  - 在对应的SHOW_YOUR_CODE代码块实现 逆序遍历 和 链表的释放
//  - 通过所有编译器检测 和 断言
//
```

```
#include "common/common.hpp"

#include "exercises/linked-list/EmbeddedList.hpp"

using d2ds::DefaultAllocator;

struct MyData {
    d2ds::DoublyLink link;
    int id;
    char data;
};

int main() {

    d2ds::DefaultAllocator::debug() = true;

    d2ds::DoublyLink head;

    d2ds::DoublyLink::init(&head);

    for (int i = 0; i < 10; i++) {
        auto dataPtr = (MyData *)
d2ds::DefaultAllocator::malloc(sizeof(MyData));
        d2ds::DoublyLink::init(&(dataPtr->link));
        dataPtr->id = i;
        dataPtr->data = 'a' + i;
        d2ds::DoublyLink::insert(&head, &(dataPtr->link));
    }

    dstruct::Stack<MyData> dataStack;
    for (auto linkPtr = head.next; linkPtr != &head; linkPtr = linkPtr->next)
{
        auto dataPtr = reinterpret_cast<MyData *>(linkPtr);
        dataStack.push(*dataPtr);
    }

    SHOW_YOUR_CODE({ // reverse traverse
        for (auto linkPtr = head.prev; linkPtr != &head; linkPtr = linkPtr-
>prev) {
            MyData *dataPtr = reinterpret_cast<MyData *>(linkPtr);

            DONT_CHANGE(
                auto myData = dataStack.top();
                d2ds_assert_eq(dataPtr->id, myData.id);
                d2ds_assert_eq(dataPtr->data, myData.data);
                dataStack.pop();
            )
        }
    })
```

```
    d2ds_assert(dataStack.empty());

    SHOW_YOUR_CODE({ // use DefaultAllocator::free(addr) to release
        d2ds::DoublyLink *target = head.next;
        while (target != &head) {
            d2ds::DoublyLink::del(&head, target);
            DefaultAllocator::free(target);
            target = head.next;
        }
    })

    d2ds_assert(head.next == &head);

    XLINGS_WAIT

    return 0;
}
```

继承式

```
// embedded-dlist.4.cpp - write
//
// 描述:
//   嵌入式双链表 - 继承式
//
// 目标/要求:
//  - 在对应的SHOW_YOUR_CODE代码块实现 逆序遍历 和 链表的释放
//  - 通过所有编译器检测 和 断言
//

#include "common/common.hpp"

#include "exercises/linked-list/EmbeddedList.hpp"

using d2ds::DefaultAllocator;

template <typename T>
struct ENode : d2ds::DoublyLink {
    ENode * prev() {
        return static_cast<ENode *>(d2ds::DoublyLink::prev);
    }

    ENode * next() {
        return static_cast<ENode *>(d2ds::DoublyLink::next);
    }
```

```
    T * data() {
        return static_cast<T *>(this);
    }
};

struct MyData : ENode<MyData> {
    int id;
    char data;
};

int main() {

    d2ds::DefaultAllocator::debug() = true;

    ENode<MyData> head;

    d2ds::DoublyLink::init(&head);

    for (int i = 0; i < 10; i++) {
        auto dataPtr = (MyData *)
d2ds::DefaultAllocator::malloc(sizeo(MyData));
        d2ds::DoublyLink::init(dataPtr);
        dataPtr->id = i;
        dataPtr->data = 'a' + i;
        d2ds::DoublyLink::insert(&head, dataPtr);
    }

    dstruct::Stack<MyData> dataStack;
    for (auto nodePtr = head.next(); nodePtr != &head; nodePtr = nodePtr-
>nex()) {
        auto dataPtr = nodePtr->data();
        dataStack.push(*dataPtr);
    }

    SHOW_YOUR_CODE({ // reverse traverse
        for (auto nodePtr = head.prev(); nodePtr != &head; nodePtr = nodePtr-
>prev()) {
            MyData *dataPtr = nodePtr->data();

            DONT_CHANGE(
                auto myData = dataStack.top();
                d2ds_assert_eq(dataPtr->id, myData.id);
                d2ds_assert_eq(dataPtr->data, myData.data);
                dataStack.pop();
            )
        }
    })

    d2ds_assert(dataStack.empty());
```

```
    SHOW_YOUR_CODE({ // use DefaultAllocator::free(addr) to release
        ENode *target = head.next();
        while (target != &head) {
            d2ds::DoublyLink::del(&head, target);
            DefaultAllocator::free(target);
            target = head.next();
        }
    })

    d2ds_assert(head.next() == &head);

    XLINGS_WAIT

    return 0;
}
```

总结

本章节是基于**嵌入式单链表**章节继续来介绍链表中最常用的双链表对应的嵌入式双链表的实现和使用示例。显示简单介绍了双链表的引入原因(相对与单链表)。然后, 又以同一个示例展示了嵌入式双链表的不同的使用方式

数据结构的基本概念

数据结构是计算机科学中存储、组织数据的方式，以便可以有效地访问和修改。它们是编程中的核心概念，决定了数据的逻辑结构以及数据之间的关系。数据结构的选择对程序的性能有很大影响，包括内存使用效率和执行速度。

线性数据结构

线性数据结构中的元素以线性顺序排列，这意味着它们只有一个前驱和一个后继。常见的线性数据结构包括：

- **数组：** 一组固定大小的元素序列，通常是相同类型的数据。
- **链表：** 由节点组成，每个节点包含数据和指向下一个节点的引用。
- **栈：** 遵循后进先出（LIFO）原则的集合，只能在一端（顶部）进行数据的添加和删除。
- **队列：** 遵循先进先出（FIFO）原则的集合，数据从后端添加，在前端移除。
- **双端队列（Deque）：** 一种允许从两端添加和移除元素的队列。

非线性数据结构

在非线性数据结构中，数据元素不是以线性方式排列，每个元素可能有多个前驱和后继。常见的非线性数据结构包括：

树

包含节点和边的集合，其中每个节点最多只有一个父节点和零个或多个子节点。

- **二叉树：** 每个节点最多有两个子节点的树结构。
- **堆：** 一种特殊的完全二叉树，满足某种特定顺序（最大堆或最小堆）。
- **二叉搜索树（BST）：** 一种二叉树，其中每个节点都满足左子树中所有元素的值小于节点的值，右子树中所有元素的值大于节点的值。

图

由节点（或顶点）和连接这些节点的边组成的集合。

- **有向图：** 边有方向的图。
- **无向图：** 边没有方向的图。
- **加权图：** 边被赋予了权重的图。

特殊类型或抽象数据结构（ADT）

除了上述基本分类，还存在特殊类型的数据结构，通常被视为抽象数据类型（ADT），它们更多地关注操作和行为，而不是实现的具体细节。主要包括:

- **散列表（哈希表）：** 通过哈希函数将键映射到数组中的位置来存储键值对。
- **集合：** 一组无序的不重复元素。
- **字典（映射）：** 一组存储键值对的数据结构，键是唯一的。
- **优先队列：** 元素出队顺序是根据它们的优先级来确定的。

C++ 基础

本章节将会介绍在实现d2ds中练习对应的数据结构时会遇到的一些C++相关的基础知识

- 范型编程
- 语法糖 | 范围for循环

template | 模板：范型编程初识

预览

范型编程是一种**代码生成技术**, 它能帮助我们节省写大量重复代码的时间。例如, 在我们实现数据结构的时候, 使用**范型编程**技术可以让我们写一套代码就能应用到多种类型的效果。当然, 要想深度掌握**范型编程**技术不是一个简单的事情, 它的难度不亚于学习一门新的语言。 但是幸运的是在**d2ds**中我们只涉及其最基础的部分, 下面我们将来简单的介绍它们。

函数模板 - max

实现 `d2ds::max` 函数, 实现获取两个a和b变量中的最大值

dslings - 测试代码

```cpp
// template.0.cpp - readonly
//
// 描述:
//  实现max函数模板
//
// 目标/要求:
//  - 不修改该代码检测文件
//  - 在exercises/other/cpp-base/Template.hpp中完成你的代码设计
//  - 通过所有断言检测
//

#include <dstruct.hpp>

#include "common/common.hpp"

#include "exercises/other/cpp-base/Template.hpp"

int main() {
    { // int
        int a = -1, b = 1;
        d2ds_assert_eq(d2ds::max(a, b), dstruct::max(a, b));
    }

    { // unsigned int
        unsigned int a = 4294967295, b = 1;
        d2ds_assert_eq(d2ds::max(a, b), dstruct::max(a, b));
    }

    { // double
        double a = 1.3, b = 3.1;
        d2ds_assert_eq(d2ds::max(a, b), dstruct::max(a, b));
    }

    return 0;
}
```

dslings - 检测结果

```
🌏Progress: [===>----------] 3/12

[Target: 1.template-0]

✅ Successfully ran tests/other/cpp-base/template.0.cpp!

🎉   The code is compiling!   🎉

Output:
====================
[D2DS LOGI]: - ✅ | d2ds::max(a, b) == dstruct::max(a, b) (1 == 1)
[D2DS LOGI]: - ✅ | d2ds::max(a, b) == dstruct::max(a, b) (4294967295 ==
4294967295)
[D2DS LOGI]: - ✅ | d2ds::max(a, b) == dstruct::max(a, b) (3.100000 ==
3.100000)
[D2DS LOGW]:    main: tests/other/cpp-base/template.0.cpp:35 - Delete the
XLINGS_WAIT to continue...

====================

Book: https://sunrisepeak.github.io/d2ds
```

max函数 - 重载版本实现

通过C++的函数**重载技术(overload)**, 我们分别对 `int` | `unsigned int` | `double` 类型版本的max进行实现

```
int max(int a, int b) {
    return a > b ? a : b;
}

unsigned int max(unsigned int a, unsigned int b) {
    return a > b ? a : b;
}

double max(double a, double b) {
    return a > b ? a : b;
}
```

这里通过观察**max函数**这三个类型的实现, 可以轻易感觉到它们只有参数和返回值类型不一样, 而函数体的代码都是一样的。此时的需求开发越多, 就会让开发者产生在做**重复工作**的感觉。部分想"偷懒"的程序员, 可能会借助IDE来设置代码模板来减轻工作量。幸运由于大多数程序员对这种"偷懒"的必要性达成了共识, 这促使了多数的编程语言对**范型编程|模板**做了支持, 在C++中

对应的就是 `template` 。

max函数 - 函数模板版本实现

通过函数模板, 可以写一套代码实现上面(重载实现)三套代码的效果

```
template <typename T>
T max(T a, T b) {
    return a > b ? a : b;
}
```

标识	解释
`template`	模板标识
`<>`	范形参数类型列表(可以有多个参数)
`typename T`	`typename` 为类型名修饰符, 后面跟着类型名标识 `T`

在数据结构实现中使用模板技术

- **第一个好处:** 只需要实现一套代码逻辑实现就可以支撑多种类型
- **第二个好处:** 编译器在编译期只会对使用到的类型做模板实例化, 对于没有用到的类型不会进行代码生成

如当只使用 `int` 和 `double` 类型时:

```
d2ds::max(1, 2);
d2ds::max(1.1, 0.8);
```

编译器通过按需进行代码生成来减少代码量, 只会实例化出如下两个版本:

```
int max(int a, int b) {
    return a > b ? a : b;
}

double max(double a, double b) {
    return a > b ? a : b;
}
```

注: 模板的代码生成技术在特定情况下, 也可能造成代码膨胀(code bloat)的问题

类模板 - Box

实现 d2ds::Box 用于存储**指定类型**(原生类型和自定义类型)的值

dslings - 测试代码

```
// template.2.cpp - readonly
//
// 描述:
//  实现Box类模板，来存储指定类型的值
//
// 目标/要求:
//  - 不修改该代码检测文件
//  - 在exercises/other/cpp-base/Template.hpp中完成你的代码设计
//  - 通过所有断言检测
//

#include <dstruct.hpp>

#include "common/common.hpp"

#include "exercises/other/cpp-base/Template.hpp"


int main() {
    {
        d2ds::Box<int> box;
        box.set_value(2);
        d2ds_assert_eq(box.get_value(), 2);
    }

    {
        d2ds::Box<dstruct::String> box;
        box.set_value("Hello, d2ds!");
        d2ds_assert(box.get_value() == dstruct::String("Hello, d2ds!"));
    }

    XLINGS_WAIT

    return 0;
}
```

dslings - 检测结果

```
🌏Progress: [=====>-------] 5/12

[Target: 1.template-2]

✅ Successfully ran tests/other/cpp-base/template.2.cpp!

🎉   The code is compiling!   🎉

Output:
====================
[D2DS LOGI]: - ✅ | box.get_value() == 2 (2 == 2)
[D2DS LOGI]: - ✅ | box.get_value() == dstruct::String("Hello, d2ds!")
[D2DS LOGW]:    main: tests/other/cpp-base/template.2.cpp:33 - Delete the
XLINGS_WAIT to continue...

====================

Book: https://sunrisepeak.github.io/d2ds
```

Box类模板 - 类型定义

```
d2ds::Box<int> intBox;
d2ds::Box<dstruct::String> stringBox;
```

类模板的定义和函数模板的定义是类似的, 都是在类名(函数签名)前使用 `template` 进行标识

```
template <typename T>
class Box {

};
```

Box类模板 - 具体实现

在类模板的作用域中, 可以直接把类型 `T` 当成一个正常的类型符号使用以及用它来完成对应的代码实现

```
template <typename T>
class Box {
public:
    Box() : mVal_e{} { }

    T get_value() const {
        return mVal_e;
    }

    void set_value(const T &val) {
        mVal_e = val;
    }

private:
    T mVal_e;
};
```

在Box的实现中, 使用 `T mVal_e;` 定义了一个存储用户指定类型值的成员变量。并且在 `get_value` 和 `set_value` 成员函数中也像使用正常的类型一样使用**类型名 `T`** 。它是一个未确定的类型的标识符, 在编译期编译器将会根据使用者指定的类型来去实例化出对应的版本, 就像上面函数模板一样。总之, 在编写模板代码的时候我们可以把 `T` 当成一个**未知类型名**, 像正常大多数类型名的用法一样来使用它。

> 注:
>
> 关于**模板**, 不同的群体有不同的称呼偏好。如: 模板XX - (模板函数 模板类) 也有 XX模板 - (函数模板 类模板), 。总体来说, 习惯性的称谓没有固定的统一形式(或者不必定要强制统一)。
>
> 但是为了避免歧义和方便交流讨论, 在同一本书中保持一致性是有必要的, 本书基本遵从后者(代码特征后置)的习惯, 即 XX类 | XX模板 | XX函数 分别对应的是 类 | 模板 | 函数

总结

本小节介绍了C++中**范型编程**中最基础和常用的两个部分 -- **函数模板**和**类模板**的定义方法。通过使用模板的编译器**代码生成**可以让我们设计的数据结构支持多种类型, 而又不用编写多份代码。如果你已经掌握了本节内容, 那就快去到数据结构实现的部分去实际使用C++的范型编程技术吧。

语法糖 | 范围for循环

预览

C++从C++11开始也像很多语言一样提供了范围for循环这个"语法糖"。它是用作对范围中的各个值（如容器中的所有元素）进行操作的较传统for循环更加可读的等价版本。下面以 `std::vector` 为例对比并演示一下它的使用:

使用普通for循环

通过 `std::vector` 的 `begin` 和 `end` 迭代器来获取数据结构(容器)中存储的数据。 其中**迭代器 `it` **的行为很像指针, 可以通过*号去"解引用"获取数据, 通过 `++` 让迭代器指向存储的下一个数据。

```cpp
#include <vector>

int main() {
    std::vector<int> vec { 1, 2, 3, 4 };

    int val;
    for (auto it = vec.begin(); it != vec.end(); it++) {
        val = *it;
        //...
    }

    return 0;
}
```

使用范围for循环

通过使用范围for循环简化了对数据结构中数据的访问, 不需要开发者直接去控制和判断迭代器就可以轻松访问到所有数据。这里需要注意的是 —— 其实这个简化了的for循环的本质也是像上面一样使用了迭代器的设计模式, 只是编译器帮我们省去了关于迭代器的相关操作, 原理上可视为等价的。

```cpp
#include <vector>

int main() {
    std::vector<int> vec { 1, 2, 3, 4 };

    for (int val : vec) {
        // ...
    }

    return 0;
}
```

自定义类型如何支持这个语法糖?

对于库开发者, 比起使用这个范围for循环, 更让其好奇和兴奋的是 —— **如何让自己写的数据结构也能支持这么好的性质, 这样大家用起来就会更爽了(这可能就是传说中的大家好才是真的好哈哈)**。下面将讨论自定义类型如何支持这个范围for语法糖。

这里引用一下cppreference上对它的解释

```
// https://en.cppreference.com/w/cpp/language/range-for
{ // until C++17
    auto && __range = range-expression ;
    for (auto __begin = begin-expr, __end = end-expr; __begin != __end;
++__begin)
    {
        range-declaration = *__begin;
        loop-statement
    }
}
```

为了更好的观察, 我们还是以上面 `std::vector` 的范围for作为例子, 给出编译对这个语法套进行代码展开的**可能实现**

```
{ // 没有展开的形式
    for (int val : vec) {
        // ...
    }
}
{ // 编译器代码展开的可能实现
    auto && __range = vec;
    for (auto __begin = __range.begin(), __end = __range.end(); __begin !=
__end; ++__begin) {
        auto && val = *__begin;
        // ...
    }
}
{ // 编译器代码展开的可能实现 -- 易读版
    auto __end = vec.end();
    for (auto it = vec.begin(); it != __end; ++it) {
        int val = *it;
        // ...
    }
}
```

从简化的 易读版 上可以看出, 和前面最开始的普通版本的for循环实现是差不多的, 并且我们可以总结出如下要素:

- 1.需要实现 `begin()`
- 2.需要实现 `end()`
- 3. `begin()` / `end()` 返回的类型需要具备指针的行为操作(或者至少要满足 `*` 和 `++` 操作)

下面我就以一个例子的实现来具体阐述和感受**自定类型**支持范围for的完整过程

模拟Python中range - 实现PyRange

range 类型表示不可变的数字序列，通常用于在 for 循环中循环指定的次数。下面我们将简单介绍一下Python中range对象在for循环中的应用, 然后在使用C++实现一个**支持范围for循环的PyRange**来近似模拟它的行为。

Python - range

- `range(start, stop)`
- `range(start, stop, step)`

```
speak@speak-pc:~/workspace/github/d2ds$ python3
Python 3.10.12 (main, Nov 20 2023, 15:14:05) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> for i in range(0, 10):
...     print(i)
...
0
1
2
3
4
5
6
7
8
9
>>> for i in range(0, 50,  5):
...     print(i)
...
0
5
10
15
20
25
30
35
40
45
>>>
```

dslings - 测试代码

为了简单PyRange只模拟**Python - range**中在for循环中应用的有限部分。下面是PyRange在范围for循环中生成索引数据(int)的代码示例:

```cpp
// range_for.3.cpp - readonly
//
// 描述:
//  实现PyRange在范围for循环的支持, 并保证数据生成的正确性
//
// 目标/要求:
//  - 不修改该代码检测文件
//  - 在exercises/other/cpp-base/RangeFor.hpp中完成你的代码设计
//  - 通过所有断言检测
//

#include <dstruct.hpp>

#include "common/common.hpp"

#include "exercises/other/cpp-base/RangeFor.hpp"

int main() {
    {
        int index = 0;
        for (int val : d2ds::PyRange(0, 10)) {
            d2ds_assert_eq(val, index);
            index++;
        }
    }

    {
        int index = 0, step = 5;
        for (auto val : d2ds::PyRange(0, 50, step)) {
            d2ds_assert_eq(val, index);
            index += step;
        }
    }

    XLINGS_WAIT

    return 0;
}
```

上述代码中, 在接口的使用上为了更像Python中的range, PyRange也遵从了如下设计

接口	简介
`PyRange(start, stop, step = 1)`	step为值变化步长默认为1.数据生成遵从左闭右开原则

dslings - 检测结果

```
🌍Progress: [=========>---] 9/12

[Target: 2.range_for-3]

✅ Successfully ran tests/other/cpp-base/range_for.3.cpp!

🎉   The code is compiling!   🎉

Output:
====================
[D2DS LOGI]: - ✅ | start < stop
[D2DS LOGI]: - ✅ | step > 0
[D2DS LOGI]: - ✅ | mLen_e <= 100
[D2DS LOGI]: - ✅ | val == index (0 == 0)
[D2DS LOGI]: - ✅ | val == index (1 == 1)
[D2DS LOGI]: - ✅ | val == index (2 == 2)
[D2DS LOGI]: - ✅ | val == index (3 == 3)
[D2DS LOGI]: - ✅ | val == index (4 == 4)
[D2DS LOGI]: - ✅ | val == index (5 == 5)
[D2DS LOGI]: - ✅ | val == index (6 == 6)
[D2DS LOGI]: - ✅ | val == index (7 == 7)
[D2DS LOGI]: - ✅ | val == index (8 == 8)
[D2DS LOGI]: - ✅ | val == index (9 == 9)
[D2DS LOGI]: - ✅ | start < stop
[D2DS LOGI]: - ✅ | step > 0
[D2DS LOGI]: - ✅ | mLen_e <= 100
[D2DS LOGI]: - ✅ | val == index (0 == 0)
[D2DS LOGI]: - ✅ | val == index (5 == 5)
[D2DS LOGI]: - ✅ | val == index (10 == 10)
[D2DS LOGI]: - ✅ | val == index (15 == 15)
[D2DS LOGI]: - ✅ | val == index (20 == 20)
[D2DS LOGI]: - ✅ | val == index (25 == 25)
[D2DS LOGI]: - ✅ | val == index (30 == 30)
[D2DS LOGI]: - ✅ | val == index (35 == 35)
[D2DS LOGI]: - ✅ | val == index (40 == 40)
[D2DS LOGI]: - ✅ | val == index (45 == 45)
[D2DS LOGW]:    main: tests/other/cpp-base/range_for.3.cpp:35 - Delete the
XLINGS_WAIT to continue...

====================

Book: https://sunrisepeak.github.io/d2ds
```

PyRange - 类型定义

```
d2ds::PyRange(0, 10);
d2ds::PyRange(0, 5, 200);
```

PyRange的构造函数为了简单, 使用了三个int作为输入参数, 并且为了支持上面两种使用模式最后一个参数step使用了默认参数为1的设置

```
class PyRange {
public:
    PyRange(int start, int stop, int step = 1) {

    }
};
```

PyRange - begin 和 end

```
d2ds::PyRange range(2, 10);
auto begin = range.begin();
auto end = range.end();
```

给PyRange实现两个无参数的成员函数begin和end, 搭出基本结构

```
class PyRange {
public:
    PyRange(int start, int stop, int step = 1) {

    }

public:
    void * begin() const {
        return nullptr;
    }

    void * end() const {
        return nullptr;
    }
};
```

PyRange - 迭代器的 * 和 ++ 操作

```
d2ds::PyRange range(0, 10);

auto begin = range.begin();
auto end = range.end();

d2ds_assert_eq(*begin, 0);
++begin;
d2ds_assert_eq(*begin, 1);
```

C++的范围for循环使用的迭代器, 是一种类指针行为的类型。幸运的是原生指针就符合这种迭代器的性质, 所以这里让 `begin/end` 返回 `const int *` 类型, 这就自动实现了*操作符解引用获取int类型数据和通过++自增运算符移动到下一个数据。

通过在PyRange内部设置一个数组mArr_e用来存储数据值和一个mLen_e来标识结束位置来简化实现, 虽然它看起来很不优雅。同时在构造函数中暂时也只实现step等于1的情况

```
class PyRange {
public:
    PyRange(int start, int stop, int step = 1) {
        mLen_e = stop - start;
        for (int i = 0; i < mLen_e; i++) {
            mArr_e[i] = i + start;
        }
    }

public:
    const int * begin() const {
        return mArr_e;
    }

    const int * end() const {
        return mArr_e + mLen_e;
    }

private:
    int mLen_e;
    int mArr_e[100];
};
```

注: 本文为了简单实现PyRange的方式是不够优雅的, 相对优雅一些的实现见设计模式 - 迭代器设计模式章节中的实现

PyRange - 完整实现

这里完善了PyRange构造函数中对step的支持和增加了一些参数限制的检测

```
class PyRange {
public:
    PyRange(int start, int stop, int step = 1) {

        mLen_e =  (stop - start) / step;

        d2ds_assert(start < stop);
        d2ds_assert(step > 0);
        d2ds_assert(mLen_e <= 100);

        for (int i = 0; i < mLen_e; i++) {
            mArr_e[i] = start;
            start = start + step;
        }
    }

public:
    const int * begin() const {
        return mArr_e;
    }

    const int * end() const {
        return mArr_e + mLen_e;
    }

private:
    int mLen_e;
    int mArr_e[100];
};
```

总结

本小节先是对比了普通for循环和范围for循环的使用, 然后通过分析编译器对**范围for循环**的代码展开结构, 来探究在自定义类型中如何实现**范围for循环**的支持, 最后通过实现一个模拟Python中常用的range对象 —— PyRange, 来进一步通过写代码的方式体验实现**范围for循环**支持的完整过程。那么, 现在快去给自己实现的数据结构添加**范围for循环**的语法糖支持吧(如果你的数据结构内存布局不是连续存储, 你可能还需要阅读设计模式 - 迭代器设计模式章节中的内容)...

行为控制

拷贝语义

移动语义

移动语义允许资源（如动态内存）从一个对象转移到另一个对象，这样可以避免不必要的资源复制，从而提高应用程序的性能和效率。它主要是通过**移动构造函数**和**移动赋值运算符**实现