Seal

Grammar docmuent

在 Seal 中，一切 Code Block 操作由“{}”来表示，而一切语句由“;”结尾，例如想要调

用 print 输出 hello world

```
print(“Hello World”);
```

而在 Seal 采用强类型, Seal 有如下几个系统类型:

int - 声明整数　　float - 浮点数　　char - 字符

string - 字符串　　bool - 布尔值　　def - 无类型

double - 双浮点数

而在 Seal 中，用关键字 class 来声明类，且语法与 C++相似

```
class my_class {
    private:
        int obj_ct;

    public:
        my_class() { // Class Constructor
            obj_ct = 0;
        }

}
```

注意:Seal 中的实例化对象并不支持使用 const 修饰词进行修饰

值得注意的是, Seal 中的对象没有构析函数，因为 Seal 中对象的构析是由系统进行的操作,

如果想要在 Seal 中定义对象销毁的行为需要声明一个回调函数　　__ destroy __　，它

应该以 bool 为返回值, bool 的真假将直接决定 Seal 是否继续由 Seal 执行 gc

```
class temp_obj {
    private:
        int content = 0;

        bool __destroy__() {
            if (content != 0) {
                content = 0;

                return true; // Seal will not do system level destroy
            }
            else {
```

```
                return false; // Seal will do system level destroy
            }
        }
}
```

值得注意的是, Seal 中用户调用 delete 来删除对象的时候不会调用__destroy__函数,

__destroy__函数是在真正执行 Seal 系统级 GC 的时候才会调用的, 如果想要定义用户调

用 delete 对象的相应行为, 则该使用 operator 关键字, operator 关键字提供了重载类运

算符, 操作符的功能

```
class temp_obj {
    private:
        int content = 80;

    public:
        operator delete() {
            content = 0;
        }
}
```

除此之外, operator 关键字还支持重载-

> - 大于 < - 小于 <= - 小于或等于 >= - 大于或等于

++ - 自增 -- - 自减 + - 加 - - 减

[] - 数组取值操作符 * - 乘 / - 除

Seal 中对象也支持继承, 而如没有特备指定下, Seal 中对象的继承会一并继承父类中的所

有成员 (不论 private 或 public), 而 Seal 中的重载操作符为 [child class] <-

[Inheritance designation (Optional)] [father class], Seal 中的对象继承是可以访问基类

的所有成员无论 private 或 public

```
class father_class {
    public:
        int i = 0;

    private:
        int get() {
            return i;
```

```
        }
}

// Inherit all members from the parent class
class child_class <- father_class {
    child_class() {
        // Legal
        i = 0;

        i = get();
    }
}

class child_just_private_class <-[private] father_class {
    child_just_private_class() {
        // Legal
        get();

        // illegall, because this class just inherit private members
        // i = 8;
    }
}

class child_just_public_class <-[public] father_class {
    child_just_public_class() {
        // Legal
        i = 8;

        // illegall, because this class just inherit public members
        // get();
    }
}
```

如上代码展示了 Seal 中类的继承操作，值得注意的是 Seal 中类的继承是不会继承父类重载的操作符的。

来到变量的声明，从刚刚的势力代码已经看出来了 Seal 中类是如何声明的，那么 Seal 中如何声明数组呢，答案很简单-

```
int array = [2] { 114514, 1919 };
array[0] = 415411;
```

上面的代码演示了 Seal 中数组的声明以及读取

Seal 中的注释想必不用多说, Seal 中的注释只支持“//”的整行注释

Seal 中也有循环语句，分为 do, while, for

do:

```
int count = 0;
do {
  ++conut;

  printf(count, “cycle”);
} (count != 8)
```

while:

```
int count = 0;
while (!(count == 8)) {
    ++count;

    printf(count, “cycle”);
}
```

for (两种用法):

```
// The first usage
for (1 to 8)[index] {
    printf(index, “cycle”)
}
// The second usage
for (int index = 0; index < 8; ++index) {
    printf(index, “cycle”);
}
```

Seal 中的对象切片

在 Seal 中，对于 字符串、数组 对象支持对象切片行为

```
// Array slice example
int array = [6]{ 1, 1, 4, 5, 1, 4 };
int slice_obj = [] <- array[0 : 2]

// String slice example
string str = “#$Hello World!3$”
string slice_string = [] <- str[2 : 12]
```

Seal 中的枚举类

在 Seal 中，用 enum 来定义一个枚举类，通过篠. 篠来访问枚举成员

```
enum my_enum {
    A, B, C, D, E, F, G, H, I, J, K, L, M, N,
    O, P, Q, R, S, T, U, V, W, X, Y, Z
}

string A_String = my_enum.A.string();
int A_Enum_Int  = my_enum.A
my_enum Emun = my_enum.A
```

Seal 中的范式

在 Seal 中，对于不确定的类型可以使用范式的方法，类似 C++的模板，类，函数都支持范式，他们通过 parad 关键字来定义范式，而范式又可以有范式特化，且范式特化有一个类型选择器

```
parad<type>
class my_class {
    private:
        type content;
    public:
        type get_content() {
            return content
        }
}

parad<type>
type add(type left, type right) {
    return left + right;
}
// Paradigm specialization
parad<type = char>
type add(type left, type right) {
    return to_string(left) + to_string(right);
}

// Paradigm type selector
parad[selector=!{ char, int, double, float, string }]<type>
type add() {
```

```
    dynamic_error(true, “Unsupported type \”” + to_string(type) + “\””);
}
```

范式类型的判断

每一个类型都有一个属于自己的 type id，可通过 type id 判断范式类型

```
parad<type>
bool is_char_type() {
    if (type_id(type) == type_id(char)) {
        return true;
    }

    return false;
}
```

Seal 中如何调用范式对象/函数

```
MyVector<int> vector;
Add<int>(8, 7);
```

Seal 中的变量修饰符

Seal 中支持非常多的变量修饰符，此处全部列举出来

static　静态修饰符，意义为将原 heap 的数据强行压入 stack

new_static　新静态修饰符，意为每一次执行 new_static 修饰的命令时都会将变量压入 stack，而非仅压入一次

un_gc　-　指该变量不会被 gc 处理，注意如果使用此变量解释器将会在该变量成员栈压入一个名为　$no_gc　的成员，会增加内存负担，而且 un_gc 的内存需要程序员自己进行内存的管理，且原引用计数所用的 unsigned long long 空间不会被缓解

force_delete　-　在使用该修饰词修饰的变量的 delete 行为将会是　true delete　，注意，如果使用此修饰符会在变量的类成员栈压入一个名为　$force_delete　的成员，会增加内存负担

no_refer_count　-　在使用该修饰词修饰的变量的 refer_count 将会被设置为 longmax，从而达到禁用引用计数的方法，注意，此修饰符不会减少 refer_count 而带

来的内存占用，仅仅用于禁用引用计数

no_old - 将被修饰的变量设置为 常青变量 ，被修饰的变量将会一直占用分代回收 gc 的 young area，则不会被分代回收 gc 所回收

const - 将被修饰的变量设置为常量，该常量不可修改

Seal 中动态库操作

Seal Dynamic Library 是 Seal 提供的动态库操作 API，它可以让你操作动态库内函数，函数的返回值将会由解释器处理为 Seal 端类型并返回

```
dynamic_library new_library(“./test.dll”);
dynamic_library_function
function = new_library.load_function(“createWindow”);

function.type result = function.run(860, 708, “My Window”);

if (type_id(result) == type_id(bool)) {
   if (result == true) {
     printf(“Successfully created window”);
    }
    else {
     printf(“Created window error”);
    }
}
```

然而不是所有的返回值 Seal 都可以处理，如果 dll 返回值为结构体/类, Seal 则无法处理，如果想要 Seal 能够正常处理结构体/类返回值则需要使用专门针对 Seal 的 Seal Lib 文件 (.slib)，且需要使用 Seal C/C++ Level Operation Library

```
seal_library slib(“./test.slib”);
seal_library_function function = slib.load_function(“createWindow”);

function.type result = function.run(860, 708, “My Window”);
if (result.ok == false) {
    printf(“Created window failed with error : ”, result.error);
}
```

```
else {
    printf(“Successfully created window”);
}
```

Seal 中的类型别名, Seal 中使用 type_alias 关键字定义类型别名对类型定义别名，而定义的新别名的 type id 与原对象的 type id 是一致的 (支持范式类型)

```
type_alias plastic = int;
```

除了 type_alias 关键字外, Seal 中还有一个 type_define 关键字，功能与 type_alias 一致，不过 type_define 定义的新别名的 type_id 与原 type 的 type_id 是不一样的

```
type_alias  plastic = int;
type_define plastic_def = int;

if (type_id(plastic) == type_id(int)) { // True
   printf(“int type is equals plastic type”);
}

if (type_id(plastic_def) == type_id(int)) { // False
   printf(“plastic_def type is equals plastic type”);
}
else {
   printf(“plastic_def type is not equals plastic type”);
}
```

范式类型的别名定义与普通类型无差别

Seal 中的 function 类型

Seal 中有一个方便管理函数的 function 类型, function 类型可以使用范式定义也可以不使用, function 类型有几个函数成员, get_type()、execute ()、is_empty()，其中 is_empty 是指其函数是否定义行为，如果是个空函数则返回 true，否则则返回为 false，若要检查 function 是否为 NULL 则需要使用通过重载的 == 操作符来实现

```
bool test_function() {
    printf(“Hello World”);

    return true;
}
```

```
function no_paradigm_function = test_function();
function<def> with_paradigm_function = test_function();

if (no_paradigm_function == with_paradigm_function) { // True
  printf(“Equal”);
}

no_paradigm_function.get_type() result = test_function();

if (result == true) { // It will be true
  printf(“result is true”);
}
```

值得注意的是无论 function 对象是否使用范式声明，都无差别，之所以支持范式则是使代

码美观，其次，function 对象支持所有变量修饰词

Seal 中的轻函数

对于个 function 对象或者是 function 对象的传参，往往不需要特别定义一个函数，于是

Seal 有一个轻函数的功能，使用关键字 func

```
parad<type>
bool compare(type left, type right, function<bool(type, type)>compare) {
   return compare(left, right);
}

class my_class {
    public:
        int content = 0;
}

my_class objs = [2]{ 1, 2 };

compare<my_class>(objs[0]), objs[1], func(my_class left, my_class right){
  return left.content == right.content;
} <- bool)
```

Seal 中的 retr_for

retr_for 类似于 JavaScript 的 foreach，它支持对数组的遍历

```
int array = [8] = { 0, 1, 2, 3, 4, 5, 6, 7 };

retr_for (meber <- array) {
  if (member % 2 == 0) {
     printf(“Divisible by 2 : ”, member);
  }
}
```

Seal 中的不定参数

实际开发过程中会遇到很多函数需要不定量的参数, Seal 提供了一个不定参数，可使用适用于轻函数以及函数，其中不定参数作为一个独立的对象，可以被 retr_for 遍历

```
int add(int args...) {
   args.size(); // Get size
   args[0]; // Get 0th member

   int result = 0;
   retr_for (member <- args) {
    result += member;
    }

    return result;
}

add(1, 2, 3, 4, 5, 6, 7, 8);
```

Seal 中的 SAFENULL 关键字

Seal 中的 SAFENULL 关键字对于系统类型是完全兼容的，而对于非系统类型例如用户自定义类型则不完全兼容, SAFENULL 会尝试寻找自定义类型的默认构造函数，若未找到则会抛出错误，如果想要解释器不初始化设置为空内存则使用 NULL，使用 SAFENULL 可以更安全地设置一个对象为空

Seal 中的结构体

Seal 中使用 strcut 定义一个结构体，结构体可以被理解为缺省为 public 的 class，定义方法与 class 相同只不过不支持 public 与 private 操作符，故不演示代码

Seal 中的 from 关键字

Seal 中引用另一个 Seal 源文件需要使用 from 关键字, from 关键字会自动对互相引用进行优化, 无需担心互相引用从而引发的问题

```
<count.se>
int count(string count_string, char count_char) {
    int result = 0;

    retr_for (get_char : count_string) {
      if (get_char == count_char) {
          ++result;
      }
    }

    return result;
}

<main.se>
from “count.se”

count(“AAAAAAAA”, ‘A’);
```

Seal 中的循环操作符

Seal 中, 有 break 关键字与 continue 关键字他们的作用与大多数的语言作用一致这里不做赘述

Seal 中的 switch 选择关键字

Seal 中支持 switch 选择关键字, switch 中通过 case 来选择, 而 case 是对象可重载的操作符

```
class my_int {
  public:
     operator bool case(int be_case) {
          return be_case == content;
      }
  private:
     int content = 0;
}

my_int obj;
switch (obj) {
```

```
    case 0: {
      print(“It is zero”);
    }
}
```

Seal 中的 using 关键字

Seal 中通过 Seal 关键字来导出代码块内容

```
namespace a {
    int b = 0;
}

using a;
b = 1; // Legal
```