Inside Object II

- Reference
- Const
- Operator `new` and `delete`

Reference

Declaring references

- Reference is a new way to manipulate objects in C++

```
char c;         // a character
char *p = &c;   // a pointer to a character
char &r = c;    // a reference to a character
```

- `&` indicates the variable at its right is a reference
- Local or global variables
  - `type& refname = name;`
  - For ordinary variables, the initial value is required
- In parameter lists and member variables
  - `type& refname`
  - Binding defined by caller or constructor

References

- Declares a new name for an existing object

```
int X=47;
int &Y=X; // Y is a reference to X
// X and Y now refer to the same variable
cout<<"Y="<<y; //printsY=47
Y = 18;
cout<<"X="<<x; //printsX=18
```

Rules of references

- References must be initialized with a variable at definition
- Initialization establishes a binding
- In declaration

```
int x = 3;
int &y = x;
const int &z = x;
```

- As a function argument

```
void f ( int& x );
f(y); // initialized when function is called
```

Rules of references(cn'td)

- Bindings don’t change at run time, unlike pointers
- Assignment changes the object referred-to

```
int &y = x;
y = 12; // Changes value of x
```

- The target of a reference must have a location!

```
void func(int &);
func (i * 3);      // Warning or error!
```

Pointers vs. References

- References
  - can't be null
  - are dependent on an existing variable, they are an alias for an variable
  - can't change to a new "address" location
- Pointers
  - can be set to null
  - pointer is independent of existing objects
  - can change to point to a different address

Restrictions

- No references to references
- No pointers to references

```
int &*p;        // illegal
```

- Reference to pointer is ok

```
void f(int *&p);
```

- No arrays of references

Left Value vs Right Value

- Left-value can be simply regarded as value that can be used at the left of assignment:
  - Variable, reference
  - Result of operator `*`, `[]`, `.` and `->`
- Right-value are values can be used at the right hand of the assignment:
  - Literal
  - Expression
- A reference parameter can take left-value only —> reference is the alias of a left-value
- The passing of an argument is initializing of the parameter

左值、右值与右值引用

- 在赋值表达式中，出现在等号左边的就是“左值”，而在等号右边的，则称为“右值”
- 可以取地址的、有名字的就是左值，反之，不能取地址的、没有名字的就是右值
- 在C++11中，右值是由两个概念构成的，一个是将亡值（xvalue，eXpiring Value），另一个则是纯右值（prvalue，Pure Rvalue）

纯右值 vs 将亡值

- 纯右值就是C++98标准中右值的概念，讲的是用于辨识临时变量和一些不跟对象关联的值。比如非引用返回的函数返回的临时变量值就是一个纯右值。一些运算表达式，比如`1 + 3`产生的临时变量值，也是纯右值。而不跟对象关联的字面量值，比如：`2`、`'c'`、`true`，也是纯右值。此外，类型转换函数的返回值、lambda表达式等，也都是右值
- 将亡值则是C++11新增的跟右值引用相关的表达式，这样表达式通常是将要被移动的对象（移为他用），比如返回右值引用`T&&`的函数返回值、`std::move`的返回值，或者转换为`T&&`的类型转换函数的返回值

右值引用

- 右值引用就是对一个右值进行引用的类型。事实上，由于右值通常不具有名字，我们也只能通过引用的方式找到它的存在。通常情况下，我们只能是从右值表达式获得其引用。比如：

```
T && a = ReturnRvalue();
```

- 这个表达式中，假设`ReturnRvalue`返回一个右值，我们就声明了一个名为`a`的右值引用，其值等于`ReturnRvalue`函数返回的临时变量的值。
- 右值引用和左值引用都是属于引用类型。无论是声明一个左值引用还是右值引用，都必须立即进行初始化。而其原因可以理解为是引用类型本身自己并不拥有所绑定对象的内存，只是该对象的一个别名。左值引用是具名变量值的别名，而右值引用则是不具名（匿名）变量的别名

Right-value reference

```
int x=20; // left-value
int&& rx = x * 2;
// the result of x*2 is a right-value, rx extends its lifttime
int y = rx + 2; // In this way it can be reused:42
rx = 100;
// Once a right-value reference is initialized,
// this variable becomes a left-value that can be assigned
int&& rrx1 = x;
// Illegal: right-value reference can not be initialed by a left-value
const int&& rrx2 = x;
// Illegal: right-value reference can not be initialed by a left-value
```

```
T && a = ReturnRvalue();
```

- `ReturnRvalue` 函数返回的右值在表达式语句结束后，其生命也就终结了（通常我们也称其具有表达式生命期），而通过右值引用的声明，该右值又“重获新生”，其生命期将与右值引用类型变量 `a` 的生命期一样。只要 `a` 还“活着”，该右值临时量将会一直“存活”下去
- 所以相比于以下语句的声明方式：

```
T b = ReturnRvalue();
```

- 右值引用变量声明就会少一次对象的析构及一次对象的构造。因为 `a` 是右值引用，直接绑定了 `ReturnRvalue()` 返回的临时量，而 `b` 只是由临时值构造而成的，而临时量在表达式结束后会析构因应就会多一次析构和构造的开销

- 能够声明右值引用 `a` 的前提是 `ReturnRvalue` 返回的是一个右值。通常情况下，右值引用是不能够绑定到任何的左值的。比如下面的表达式就是无法通过编译的。

```
int c;
int && d = c;
```

- 相对地，在C++98标准中就已经出现的左值引用是否可以绑定到右值（由右值进行初始化）呢？比如：

```
T & e = ReturnRvalue();
const T & f = ReturnRvalue();
```

- `e` 的初始化会导致编译时错误，而 `f` 则不会

```
T & e = ReturnRvalue();
const T & f = ReturnRvalue();
```

- 在常量左值引用在C++98标准中开始就是个“万能”的引用类型
- 可以接受非常量左值、常量左值、右值对其进行初始化
- 而且在使用右值对其初始化的时候，常量左值引用还可以像右值引用一样将右值的生命期延长
- 相比于右值引用所引用的右值，常量左值所引用的右值在它的“余生”中只能是只读的
- 相对地，非常量左值只能接受非常量左值对其进行初始化

Parameter as right-value

```cpp
// take left-value
void fun(int& lref) {
    cout << "l-value" << endl;
}
// take right-value
void fun(int&& rref) {
    cout << "r-value" << endl;
}

int main() {
    int x = 10;
    fun(x); // output: l-value reference
    fun(10); // output: r-value reference
}
```

`std::move()`

- C++11中，<utility>中提供了函数`std::move`，功能是将一个左值强制转化为右值引用，继而我们可以通过右值引用使用该值，用于移动语义
- 被转化的左值，其生命期并没有随着左右值的转化而改变

3-21.cpp

- 在编写移动构造函数的时候，应该总是使用`std::move`转换拥有形如堆内存、文件句柄的等资源的成员为右值，这样一来，如果成员支持移动构造的话，就可以实现其移动语义，即使成员没有移动构造函数，也会调用拷贝构造，因为不会引起大的问题

- 移动语义一定是要改变临时变量的值
- `Moveable c(move(a));`这样的语句。这里的`a`本来是一个左值变量，通过`std::move`将其转换为右值。这样一来，`a.i`就被`c`的移动构造函数设置为指针空值。由于`a`的生命期实际要到所在的函数结束才结束，那么随后对表达式`*a.i`进行计算的时候，就会发生严重的运行时错误

Const reference as parameter

```
void fun(const int& clref) {
    cout << "l-value const reference\n";
}
```

- Such a function can accept right-value when there is no function takes right-vaue

Constant

`Const`

- declares a variable to have a constant value

```
const int x = 123;
x = 27; // illegal!
x++; // illegal!
int y = x; // Ok, copy const to non-const
y = x;     // Ok, same thing
const int z = y; // ok, const is safer
```

Constants

- Constants are variables
  - Observe scoping rules
  - Declared with `const` type modifier
- A const in C++ defaults to internal linkage
  - the compiler tries to avoid creating storage for a const -- holds the value in its symbol table.
  - extern forces storage to be allocated.

Compile time constants

```
const int bufsize = 1024;
```

- value must be initialized
- unless you make an explicit extern declaration:

```
extern const int bufsize;
```

- Compiler won't let you change it

Run-time constants

- const value can be exploited

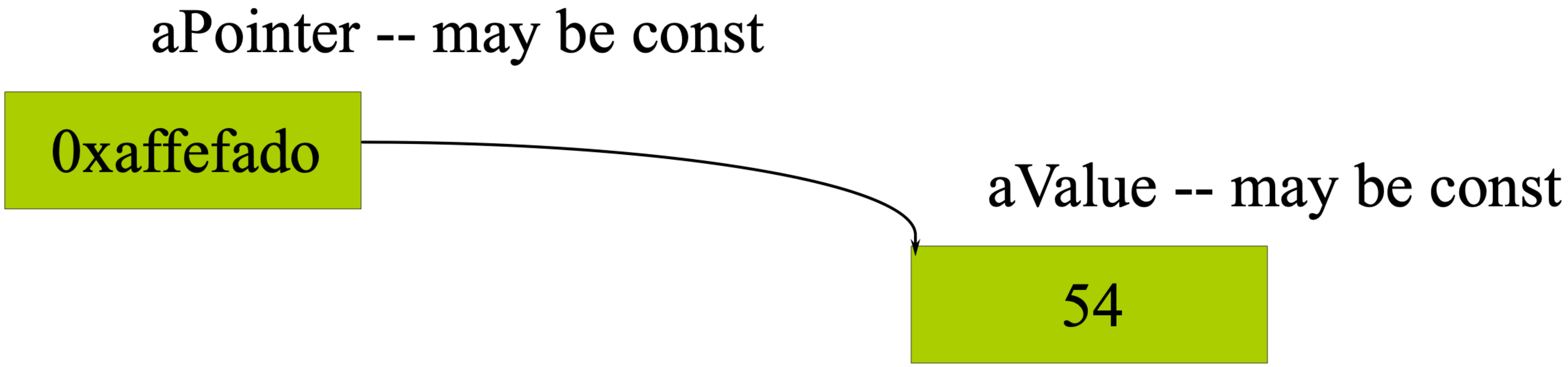
```
const int class_size = 12;
int finalGrade[class_size]; // ok
int x;
cin >> x;
const int size = x;
double classAverage[size]; // ok
```

Aggregates

- It’s possible to use const for aggregates, but storage will be allocated. In these situations, const means “a piece of storage that cannot be changed.”
- However, the value cannot be used at compile time because the compiler is not required to know the contents of the storage at compile time.

```
const int i[] = { 1, 2, 3, 4 };
float f[i[3]]; // Illegal
struct S { int i, j; };
const S s[] = { { 1, 2 }, { 3, 4 } };
double d[s[1].j]; // Illegal
```

Pointers and const



```
char * const q = "abc"; // q is const
*q = 'c'; // OK
q++; // ERROR
const char *p = "ABCD"; // (*p) is a const char
*p = 'b'; // ERROR! (*p) is the const
```

Quiz IV

What do these mean?

```
string p1("Fred");
const string* p = &p1;
string const* p = &p1;
string *const p = &p1;
```

Pointers and constants

	`int i;`	`const int ci = 3;`
`int *ip;`	`ip = &i;`	`ip = &ci; // ERROR`
`const int *cip`	`cip = &i;`	`cip = &ci;`

- Remember:

```
*ip  = 54;  // always legal since ip points to int
*cip = 54;  // never legal since cip points to const int
```

String Literals

```
char* s = "Hello, world!";
```

- `s` is a pointer initialized to point to a string constant
- This is actually a `const char *s` but compiler accepts it without the `const`
- Don't try and change the character values (it is undefined behavior)
- If you want to change the string, put it in an array:

```
char s[] = "Hello, world!";
```

Conversions

- Can always treat a non-const value as const

```
void f(const int* x);
int a = 15;
f(&a); // ok
const int b = a;

f(&b); // ok
b = a + 1; // Error!
```

You cannot treat a constant object as non-constant without an explicit cast (`const_cast`)

Passing by const value?

```cpp
void f1(const int i) {
    i++; // Illegal -- compile-time error
}
```

Returning by const value?

```
int f3() { return 1; }
const int f4() { return 1; }
int main() {
    const int j = f3(); // Works fine
    int k = f4(); // But this works fine too!
}
```

Passing and returning addresses

- Passing a whole object may cost you a lot. It is better to pass by a pointer. But it’s possible for the programmer to take it and modify the original value.
- In fact, whenever you’re passing an address into a function, you should make it a `const` if at all possible.

Example: ConstPointer.cpp, ConstReturning.cpp

const object

Constant objects

- What if an object is const?

```
const Currency the_raise(42, 38);
```

- What members can access the internals?
- How can the object be protected from change?
- Solution: declare member functions const
    - Programmer declares member functions to be safe

Const member functions

- Cannot modify their objects

```
int Date::set_day(int d){
    //...error check d here...
    day = d;    // ok, non-const so can modify
}
int Date::get_day() const {
    day++;       //ERROR modifies data member
    set_day(12); // ERROR calls non-const member
    return day;  // ok
}
```

Const member function definition

- Repeat the const keyword in the definition as well as the declaration

```
int get_day () const;
int get_day() const { return day };
```

- Function members that do not modify data should be const member functions are safe for const objects declared const

Const objects

- Const and non-const objects

```
// non-const object
Date  when(1,1,2001);     // not a const
int day = when.get_day(); // OK
when.set_day(13);         // OK
// const object
const Date birthday(12,25,1994);  // const
int day = birthday.get_day();     // OK
birthday.set_day(14);           // ERROR
```

Constant fields in class

```
class A {
    const int i;
};
```

- has to be initialized in initializer list of the constructor

Compile-time constants in classes

```
class HasArray {
   const int size;
   int array[size]; // ?
};
```

- use "anonymous enum" hack

```
class HasArray {
   enum { size = 100 };
   int array[size]; // OK!
};
```

- Or make the const value static:

```
class HasArray {
   static const int size = 100;
   int array[size];
}
```

Dynamically allocated memory

- `new`
  - `new int;`
  - `new Stash;`
  - `new int[10]`
- `delete`
  - `delete p;`
  - `delete[] p;`
- new is the way to allocate memory as a program runs. Pointers become the only access to that memory
- delete enables you to return memory to the memory pool when you are finished with it.
- `{}` can be used to pass init values to object(s) `new` generted.

Dynamic Arrays

```
int * psome = new int [10];
delete[] psome;
```

- The `new` operator returns the address of the first element of the block.
- The presence of the brackets tells the program that it should free the whole array, not just the element

The new-delete mech.

```
int *p=new int;
int *a=new int[10];
Student *q=new Student();
Student *r=new Student[10];
delete p;
a++;
delete[] a;
delete q;
delete r;
delete[] r;
```

Tips for new and delete

- Don't use `delete` to free memory that `new` didn't allocate.
- Don't use `delete` to free the same block of memory twice in succession.
- Use `delete []` if `new []` was used to allocate an array.
- Use `delete` (no brackets) if `new` was used to allocate a single entity.
- It's safe to apply `delete` to the null pointer (nothing happens).

What we've learned today?

- Reference
- Const
- Operator `new` and `delete`