Creational Design Patterns

Creational Design Patterns

Two main goals:

1. Encapsulate knowledge about which concrete classes the system uses

2. Hide how instances of these classes are created and built

Creational Design Patterns

- System at large knows about objects through their interfaces defined by abstract classes

- Give us flexibility in:
  - *what* gets created
  - *who* creates it
  - *how* it gets created
  - *when* it gets created

Creational Design Patterns

- Singleton
- Factory Method
- Abstract Factory
- Builder
- Prototype

Creational Patterns: Singleton

- Consider a class called `Logger`
  - Logs information to a file
  - Needed by many different parts of an application

Creational Patterns: Singleton

Logger.h

```
class Logger
{
   public:
      Logger();
      virtual ~Logger();
      const Logger& log(const std::string& message) const;
      const Logger& operator<<(const  std::string&  message) const;

   private:
      mutable  std::ofstream  _output;
};
```

create a destructor virtual, enforcing cleaning after running.

return const

C++ Programming

Const Correctness, Part 1

Recall …

- `const` is a keyword declaring a type constraint that indicates that certain data cannot be modified
- Note that this does not imply that the data is read-only in memory
- The constraint is enforced by the compiler

```
const int answer = 42;

answer =  43;     // compilation error!
```

`const` and Pointers

- Both pointers and the data they point to can be `const`

```
char* p = "Hello";                          // non-const pointer
                                            // non-const data

const char* p = "Hello";                    // non-const pointer
                                            // const data

char* const p = "Hello";                    // const pointer
                                            // non-const data

const char* const p = "Hello";              // const pointer
                                            // const data
```

- Use the right-to-left rule to read

Right-to-Left Rule

```
const int** a

int* const * b

const int* const * c

const int* const * const d
```

Right-to-Left Rule

```
// pointer to a pointer to a const int
const int** a


int* const * b


const int* const * c


const int* const * const d
```

Right-to-Left Rule

```
// pointer to a pointer to a const int
const int** a

// pointer to a const pointer to an int
int* const * b


const int* const * c


const int* const * const d
```

Right-to-Left Rule

```
// pointer to a pointer to a const int
const int** a

// pointer to a const pointer to an int
int* const * b

// pointer to a const pointer to a const int
const int* const * c


const int* const * const d
```

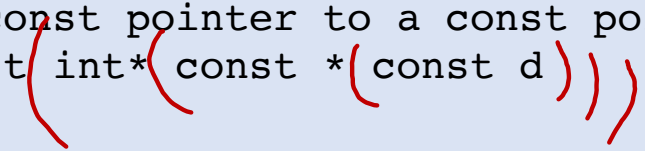
Right-to-Left Rule

```
// pointer to a pointer to a const int
const int** a

// pointer to a const pointer to an int
int* const * b

// pointer to a const pointer to a const int
const int* const * c

// const pointer to a const pointer to a const int
const int* const * const d
```



`const` Data vs `const` Pointers

```
char* h = "Hello";
char* w = "World";

h[1] = 'u';                          // will it compile?
                                        yes
h = w;                               // will it compile?
                                        yes
```

`const` Data vs `const` Pointers

```
char* h = "Hello";
char* w = "World";

h[1] = 'u';                          // yes … non-const data

h = w;                               // yes … non-const pointer
```

`const` Data vs `const` Pointers

is the data itself const?

is the pointer const?

```
const char* h = "Hello";
char* w = "World";

h[1] = 'u';                          // will it compile?
                                        No
h = w;                               // will it compile?
                                        yes
```

`const` Data vs `const` Pointers

```
const char* h = "Hello";
char* w = "World";

h[1] = 'u';                          // no … const data

h = w;                               // yes … non-const pointer
```

`const` Data vs `const` Pointers

```
char* const h = "Hello";
char* w = "World";

h[1] = 'u';          // will it compile?

h = w;               // will it compile?
```

h is a pointer to const h

data in h is now const.

Yes

pointer itself is const.

No.

this is trying to assign the pointer pointing h pointing to w now.

`const` Data vs `const` Pointers

```
char* const h = "Hello";
char* w = "World";

h[1] = 'u';                          // yes … non-const data

h = w;                               // no … const pointer
```

`const` Data vs `const` Pointers

```
const char* const h = "Hello";
char* w = "World";

h[1] = 'u';                          // will it compile?

h = w;                               // will it compile?
```

`const` Data vs `const` Pointers

```
const char* const h = "Hello";
char* w = "World";

h[1] = 'u';                          // no … const data

h = w;                               // no … const pointer
```

`const` and Pointers

- When the data pointed to is constant, some add `const` before the type name; others add it after
- You will see both in the real world

```
const Widget* const w
Widget const* const w
```

w is a const pointing at a const Widget.

const Widget in both cases.

Pointer Assignments Involving `const`

- Address of non-`const` object can be assigned to a `const` pointer

```
int i = 4;

const int* j = &i;
```

j is a pointer to const int data.

- In this case, we are promising not to change an object that was previously okay to change

Pointer Assignments Involving `const`

- You cannot assign the address of `const` object to a non-`const` pointer

```
const int i = 4;

int* j = &i;          // compilation error!
```

const int *j // this matches the type.
So it works here.
j pointing at const int.

- The second line causes a compilation error because we're saying that we might change `i` through the pointer

Pointer Assignments Involving `const`

- Exception: string literals

```
char* c = "Hello";
```

- `"Hello"` is a `const char *`, but we can assign it to a `char *`
- Explanation from the horse's mouth:

This is allowed because in previous definitions of C and C++, the type of a string literal was char. Allowing the assignment of a string literal to a char* ensures that millions of lines of C and C++ remain valid. It is, however, an error to try to modify a string literal through such a pointer.*

– Bjarne Stroustrup, The C++ Programming Language