Introduction to modern C++

Key points:

- **Syntax changes** nullptr, using, constexpr, lambda
- **Type inference** auto, decltype
- **Smart pointers** unique_ptr, shared_ptr
- **Rvalue reference** motivation, move, forward

We will dicuss not only about what they are, but also how are they implemented.

Syntax changes

- nullptr: **typed** null pointer constant

```
void f(int);
void f(void*);
f(0);        // calls f(int)
f(nullptr);  // calls f(void*)
```

- Alias declaration

```
using UPtrMapSS = std::unique_ptr
    <std::unordered_map<std::string, std::string>>;
using FP = void (*)(int, const std::string&);
template<typename T>
using MyAllocList = std::list<T, MyAlloc<T>>;
// more examples on dealing with templates
```

Syntax changes

- constexpr: compile time constants

```
constexpr int base = 3;
constexpr int exp = base + 2;
constexpr int pow(int base, int exp) noexcept
{
    auto result = 1;
    for (int i = 0; i < exp; ++i) result *= base;
    return result;
} // C++14
const std::array<int, pow(base, exp)> foo; // have 3^5 elements
                                           // can't be constexpr
```

What are the differences between `constexpr` and `const`?

Lambda expressions

```
// simple lambda expression
std::find_if(container.begin(), container.end(),
    [](int val) { return 0 < val && val < 10; });

// lambda expression that captures the environment
int x;
auto add_x = [x](int y) { return x + y; }
auto add_to_x = [&x](int y) { x += y; }
```

Note: lifetimes of the captured references are NOT extended!

Type inference

- Simple bottom-up type inference: similar to template type inference
- On expression, function return type, function parameter type, lambdas:

```
auto pow(int base, int exp)
{
    if (exp == 0) return 1;
    else return base * pow(base, exp - 1);
} // C++14.
auto lambda = [](auto x, auto y) {return x + y;};
```

Why so simple? What happens on ptrs/refs?

Smart pointers

- unique_ptr: based on life-cycle, “owns” a pointer
- shared_ptr: based on reference count, “shares” a pointer
- weak_ptr: to break shared_ptr cycle (and more)

How to transfer ownership of an unique_ptr?

How to implement a tree? What about a graph?

Rvalue references: motivation

- See the following example. Do you notice the efficiency problem?

```
class List { ... };
List a(10), b(20);
List c = a + b;  // two steps: call op+ of a on b,
                 // call constructor on (a+b)
```

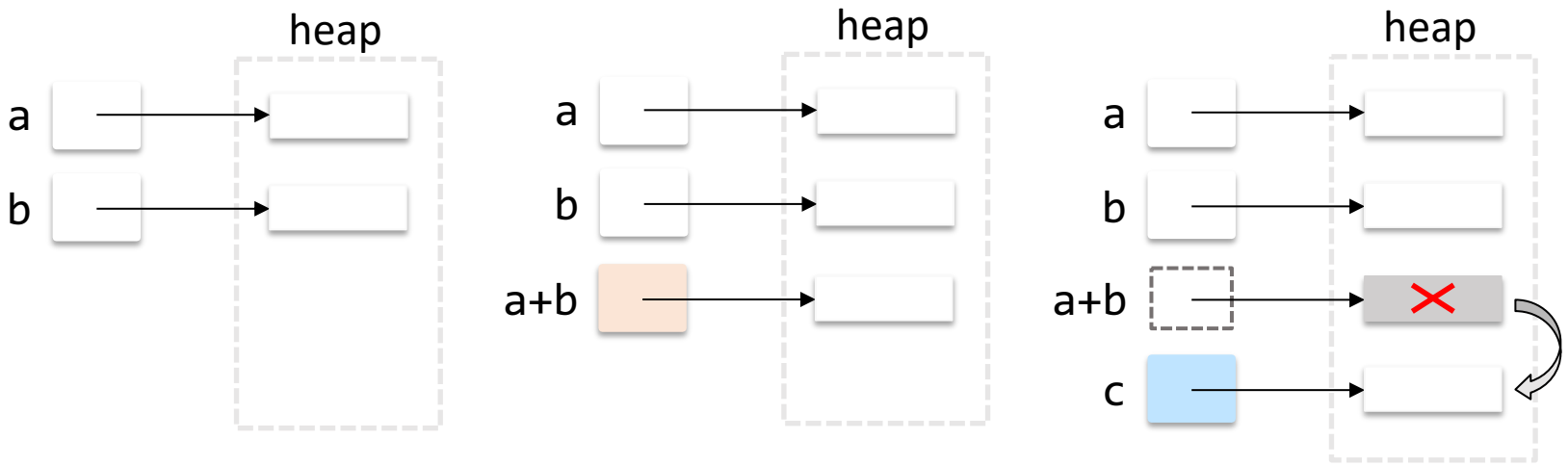
  How do you write constructor for List?

Using C++98 convention

```
class List {
public:
    int *data, len;
    ~List() { delete [] data; }
    List(int len) { data = new int [len]; this->len = len; }
    List(const List &l) {
        len = l.len;
        data = new int [len];
        for (int i = 0; i != len; ++i)
            data[i] = l.data[i];
    }
    List operator+(const List &b) {
        List rv(len + b.len);
        for (int i = 0; i != len; ++i) rv.data[i] = data[i];
        for (int i = 0; i != b.len; ++i) rv.data[i+len] = b.data[i];
        return rv;
    }
};
```

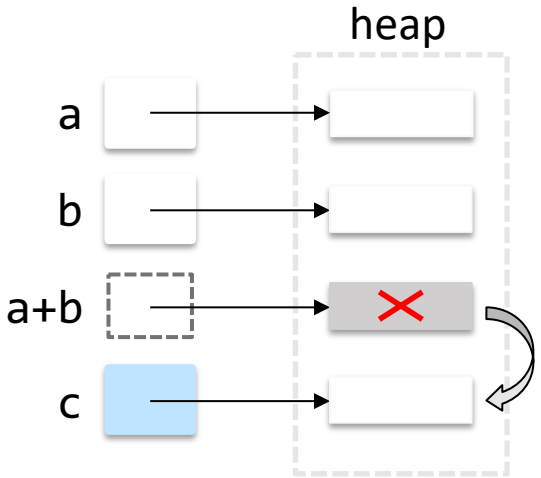
What happens?

```
List c = a + b;
```



Dilemma

- operator+ have to return a temp value, which is abandoned later.
- If we want to correctly perform copy between variables, we have to write a deep-copy constructor.
- If we don’t want to deep-copy temp variables, we’ll have to modify the temp variable so that it wouldn’t perform delete.



Approach

- We want access to a temporary object (+1s), in a way that is more flexible than const lvalue references.
- We want to distinguish temp values from regular values, so that we can treat them separately.
- Therefore, we need to add a new form of type into the type system, which is **rvalue reference**, denoted as `T&&`.

Rvalue reference

- Rvalue reference can only bind to rvalues, (non-const) lvalue reference can only bind to lvalues.
- After binding, rvalue reference can be used as a regular reference, which leads to a fact that **rvalue reference is an lvalue in expressions**:

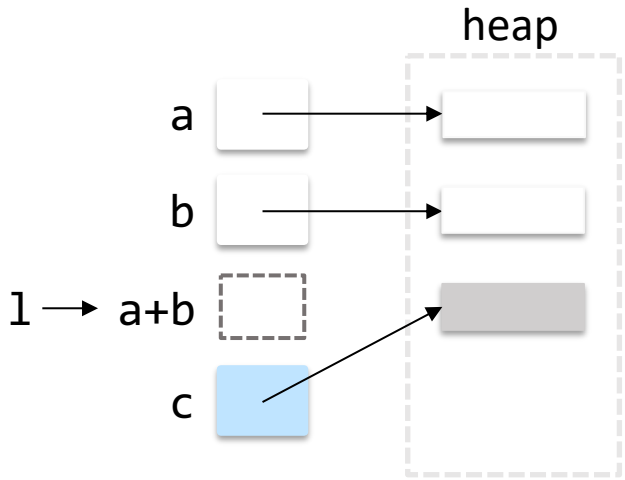
```
int &&r = 1+2;
r++;  // valid!
```

- Lifetime of that binded rvalue is extended to the lifetime of rvalue reference

New solution

- Treat lvalue reference normally: do deep-copy
- Treat rvalues separately (in what we call a move constructor):
  - directly ‘steal’ data
  - prevent temp object from destroying them
- The compiler will choose the correct overload for you

```
List(const List &l) {
    // ...
}
List(List &&l) {
    data = l.data;
    len = l.len;
    l.data = nullptr;
}
List operator+(const List &b) {
    // ...
}
```



std::move and std::forward

- std::move simply performs a type cast, which makes other funtions treat an lvalue as rvalue
- std::forward is a template function that pass arguments ‘as-is’ (in terms of type) to another function. Have a lot to do with templates.

Example: unique_ptr

- unique_ptr is, as you imagine, unique, so we shouldn't perform copy on it. Copy constructor is marked as deleted.
- When a function returns an unique_ptr, which is later assigned to a new one, we can let the new one 'steal' the underlying raw pointer, and reset the old one, in the move constructor.
- To explicitly do such 'stealing' on an lvalue (call the move constructor which accepts only rvalues), we do std::move.

Q&A