Computer Architecture

Discussion 10

CPP Qualifier:const

Yanpeng Zhao
zhaoyp1@shanghaitech.edu.cn

Qualifier : const

- Why do we need *const*?
  - Define a variable whose value we know cannot be changed.
  - Used as macro definition:

    ```
    const int MAX = 100;
    ```
  - Facilitate type-checking: e.g., *const int x = 10;* so that the compiler knows that x cannot be modified.
  - Prevent programming mistakes
  - ```
    const int x = 10; x = 12;  // error
    ```
- How to use it?
  - Constant variables, declared and must be initialized

```
const int x = 10;  // ok
const int x;  // error
```
```

Scope of const

- By default, const Objects are local to a file
  - Which implies the same const variable cannot be shared by different files and we have to define it in each file.
- Use in multiple files
  - Take the advantage of the qualifier *extern* on both its definition and declarations.

```
// heada.cpp defines and initializes a const that is accessible to other files
extern const int MULTIPLE_FILE = 12;
// headb.h same MULTIPLE_FILE as defined in heada.cpp
extern const int MULTIPLE_FILE;
```

const constant variables

- Common variable

```
TYPE const var = XXX;  ⇔  const TYPE var = XXX;
```

- Const array

```
int const arr[2] = {1, 2} ⇔ const int arr[2] = {1, 2};
```

- Const object

```
Class A;  const A b = a;  ⇔  A const b = a;
```

- Other objects
  - Bind a reference to an object of a const type.

```
Type a; TYPE const &var = a;
```

const and references

- Reference to *const*
  - Which is a reference that refers to a const type.
  - Cannot be used to change the object to which the reference is bound.
- *const* Reference is a Reference to *const*
  - A reference is not an object, so we cannot make a reference itself const.

```
const int ci = 1024;
const int &r1 = ci;  // ok: both reference and underlying object are const
r1 = 42;  // error: r1 is a reference to const
int &r2 = ci;  // error: non const reference to a const object
```

const and references

- Bind a reference to const to a nonconst object
  - We can initialize a reference to const from any expression that can be converted to the type of the reference.

```
int i = 42;
const int &r1 = i;  // we can bind a const int& to a plain int object
const int &r3 = r1 * 2; // ok: r3 is a reference to const
```

  - How is that implemented (the compiler makes it)

```
double dval = 3.14;
const int &ri = dval;
```

```
const int temp = dval;  // create a temporary const int from the double
const int &ri = temp;  // bind ri to that temporary
```

const and references

- One more word
  - A reference to const restricts only what we can do through that reference

```
int i = 42;
int &r1 = i;  // r1 bound to i
const int &r2 = i;  // r2 also bound to i; but cannot be used to change i
r1 = 0;  // r1 is not const; i is now 0
r2 = 0;  // error: r2 is a reference to cons
```

const and pointers

- As with referenes
  - Define pointers that point to either const or nonconst types
  - A pointer to const may not be used to change the object to which the pointer points

```
const double pi = 3.14;  // pi is const; its value may not be changed
double *ptr = &pi;  // error: ptr is a plain pointer
const double *cptr = &pi; // ok: cptr may point to a double that is const
*cptr = 42;  // error: cannot assign to *cptr
```

  - A pointer to const says nothing about whether the object to which the pointer points is const

```
doubled dval = 3.14;   cptr = &dval;
```

const and pointers

- Differs from references
  - Pointers are objects
  - Indicate that the pointer is const by putting the const after the *.

```
int errNumb = 0;
int *const curErr = &errNumb;  // curErr will always point to errNumb
const double pi = 3.14159;
const double *const pip = &pi; // pip is a const pointer to a const object
```

  - A pointer is itself const says nothing about whether we can use the pointer to change the underlying object.

```
*curErr = 0; // ok: reset the value of the object to which curErr is bound
```

exercises

- so what are the differences between these codes

```
const TYPE* p = XXX;
TYPE* const p = XXX;
TYPE const *p = XXX;
Const TYPE* const p = XXX;
```

- one more look at const array

```
// global variables, not in the function
const int SIZES[3] = {1, 11, 111};
int arr[SIZES[2]]; // right? Why?
```

const and functions

- Common function
  - Return value, which cannot be changed

```
const int f1();  // trivial, why?
const int* f1(); // const pointer
int* const f1(); // pointer to a const
const int& f1(); // trivial, why?
```

  - Parameter, which cannot be changed in the function

```
void f1(const int p);  // trivial, formal parameters are the copies of arguments
void f1(int* const p); // trivial, formal parameters are the copies of arguments
void f1(const int* p); // what the pointer points to is const
void f1(const int& p); // what the reference refers to is const
```

const and class

- const Member function
  - Which cannot change the state of the object

```
<return-value> <class>::<member-function>(<args>) const {}
```

  - A function declared const that doesn't prohibit non-const functions from using it; the rule is this:
    - Const functions can always be called
    - Non-const functions can only be called by non-const objects

```
class A { void f1(); void f2() const; protected: int common_var; }
```

```
const A a;
a.f1(); // error
a.f2(); // ok
```

```
const A *a = new A();
A->f1(); // error
A->f2(); // ok
```

```
A a;
a.f1(); // ok
a.f2(); // ok
```

```
void A::f1() {}
void A::f2() const
{ f1(); // error }
```

const and class

- const Member function
  - Overloading: when you want to have both const and nonconst version of the function that returns a nonconst reference:

```
const <return-value> <class>::<member-function>(<args>) const {}
<return-value> <class>::<member-function>(<args>) {}
```

```
class A {
    int & get_common_var();
    const int & get_common_var() const;
    protected: int common_var;
};
int & A::get_common_var() { return common_var; }
const int & A::get_common_var() const { return common_var; }
```

const and class

- const Member variable
  - How to define the constant variable for the class

```
class A {
    const int SIZE = 100; // error
    int arr[SIZE];
};
```

Value of SIZE can be obtained only after the object of A is created.

  - via qualifier *enum*

```
class A {
    enum {SIZE0 = 10, SIZE1 = 20};
    int arr[SIZE0];
};
```

enum belongs to the class. The value is resolved at compile time, integer by default.

```
std::cout << A::SIZE0 << std::endl;
```

const and class

- const Member variable
  - Cannot be changed
  - Which must be initialized in the initialization list

```
class A {
    A(); // error:uninitialized const member in 'const int'
    const int var;
}; // dynamically initialized
```

```
class A {
    A(); // error
    A(int v, int u):var(v){}
    const int var;
}; // dynamically initialized
```

  - Or via qualifier *static*

```
class A {
    static const int var;
};
const int A::var = XXX;
```

Same as the way static variables in the class are initialized. Statically initialized.

const and class

- Static variables of the class
  - Which are shared by all the objects of the class

```
// the way static
variables are initialized
class A {
public:
    static int static_var;
};
int A::static_var = 99;
```

```
// guess what
std::cout << A::static_var++ << std::endl;

A a1, a2;
std::cout << a1.static_var << std::endl;
std::cout << a2.static_var << std::endl;
```

Using the *this* pointer

- Allows objects to access their own address
- Implicit first argument on non-static member function call to the object
- The type of the *this* pointer depends upon the type of the object and whether the member function using this is const
  - In a non-const member function of A, this has type

```
A * const // constant pointer to an A object
```

  - In a const member function of A, this has type

```
const A * const // constant pointer to a constant A object
```

- Uhmm…

```
static void f1() const {} // is this right? Why?
```

const and class

- If we have to change const variables
  - Use qualifier *mutable*

```
mutable class A {
    int get_mutable_var();
    void set_mutable_var(int v) const;
    private: mutable int mutable_var;
};
void A::set_mutable_var(int v) const { mutable_var = v; }
```

const and class

- If we have to change const variables
  - Use qualifier *mutable*

```
mutable class A {
    int get_mutable_var();
    void set_mutable_var(int v) const;
    private: mutable int mutable_var;
};
void A::set_mutable_var(int v) const { mutable_var = v; }
```

const cast

- Use a *const_cast* in order to temporarily strip away the const-ness of the object

```
// a bad version of strlen that doesn't declare its argument const
int bad_strlen (char *x)
{
    strlen( x );
}

// note that the extra const is actually implicit in this declaration since
const char *x = "abc"; // string literals are constant

// cast away const-ness for our strlen function
bad_strlen( const_cast<char *>(x) );
```

const iterators

- Like normal iterators, except that they cannot be used to modify the underlying data

```
std::vector<int> vec;
vec.push_back( 3 );
vec.push_back( 4 );
vec.push_back( 8 );

for ( std::vector<int>::const_iterator itr = vec.begin(), end = vec.end();
      itr != end;
      ++itr ) {
        // just print out the values...
        std::cout<< *itr <<std::endl;
}
```

But…

- Just because you can return a const reference doesn‘t mean that you should return a const reference!
  - For instance, return the reference to the local data in a function, which (unless it is static) will be no longer valid.
- Efficiency Gains?
  - One common justification for const correctness is based on the misconception that constness can be used as the basis for optimizations.
  - A variable declared const will not necessarily remain unchanged. E.g., using *const_cast, mutable*

const vs #define

- The way stored
  - const: only one copy
  - define: memory allocated whenever it is used
- Type-checking
  - const: yes, has a specific data type
  - define: no, no data type, only does macro expansion
- Behaviors of the compiler
  - const: resolve the value at the compiling time or the running time
  - define: macro expansion at the preprocessing phase

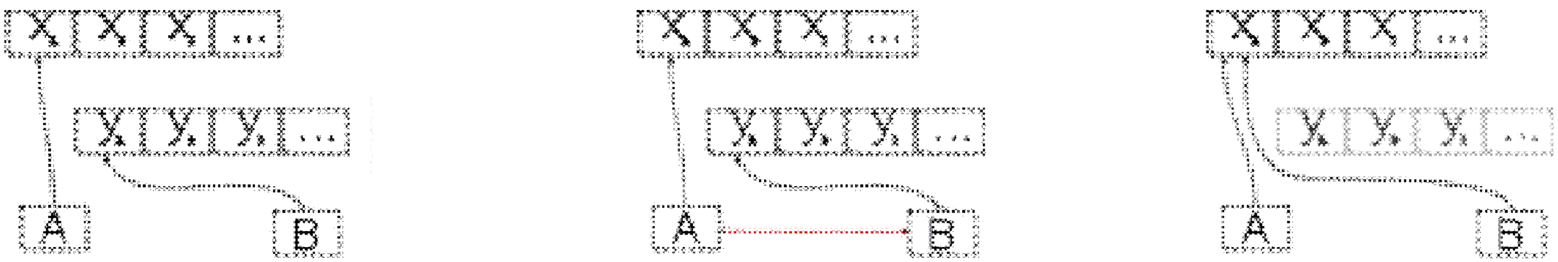
assignment operator overload

- '=' cannot deal with the objects containing the pointer variables, which may cause the shallow copy

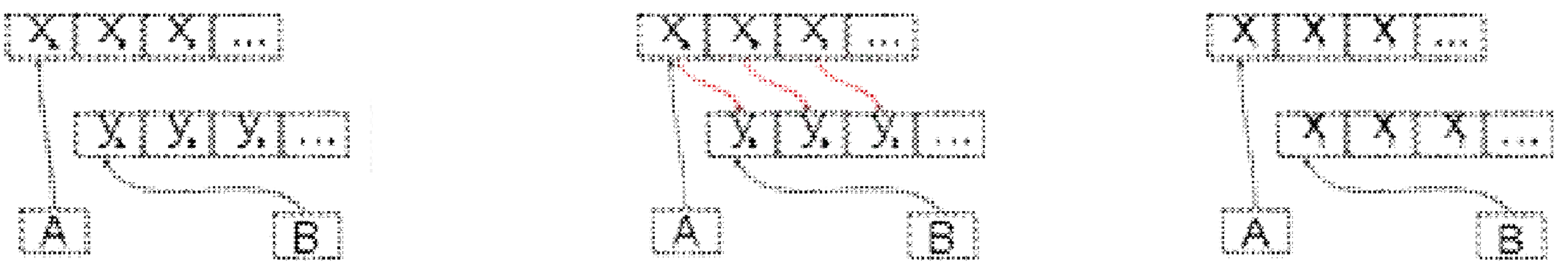
```
class C {
private: int idx; int *val;
public: C() : val(new int) {}
    C & operator=(const C & c) {
        if ( this != &c ) {
            this->idx = c.idx;
            this->val = c.val;
        }
        return *this;
    }
};
```

```
class C {
private: int idx; int *val;
public:
    C() : val(new int) {}
    // shallow copy
    C(const C & c) : idx(c.idx),
val(c.val) {}
    // deep copy
    C(const C & c) : idx(c.idx),
val(new int(*c.val)) {}
```

shallow copy vs deep copy



Shallow copy



Deep copy

From: https://en.wikipedia.org/wiki/Object_copying

Thanks

```
Q ? std::cout << “Oh no!\n” : std::cout << “Bye!\n”;
```

Borůvka's algorithm for MST

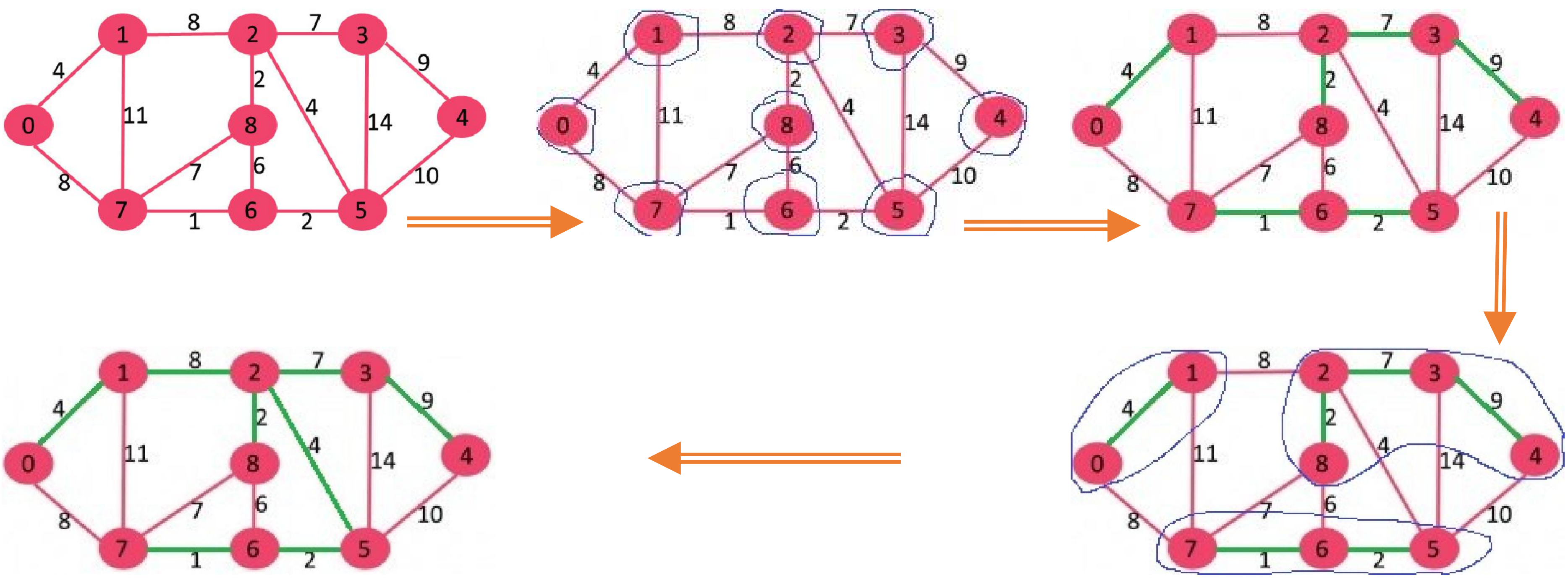
- To parallel the MST algorithm

```
Input: A connected graph G whose edges have distinct weights
Initialize a forest T to be a set of one-vertex trees, one for each vertex of the graph.
While T has more than one component:
    For each component C of T:
        Begin with an empty set of edges S
        For each vertex v in C:
            Find the cheapest edge from v to a vertex outside of C, and add it to S
        Add the cheapest edge in S to T
    Combine trees connected by edges to form bigger components
Output: T is the minimum spanning tree of G.
```

From: https://en.wikipedia.org/wiki/Bor%C5%AFvka%27s_algorithm

Borůvka's algorithm for MST

- An example



From: http://www.geeksforgeeks.org/greedy-algorithms-set-9-boruvkas-algorithm/