Advanced C++

Daniel Rutschmann

Swiss Olympiad in Informatics

October 13, 2019

Modifying in functions

```
1  void tripple(int x){
2      x *= 3;
3      print("trippled to", x);
4  }
5  signed main(){
6      int a = 5;
7      tripple(a);
8      print("a:", a);
9  }
```

Modifying in functions

```
1  void tripple(int x){
2      x *= 3;
3      print("trippled to", x);
4  }
5  signed main(){
6      int a = 5;
7      tripple(a);
8      print("a:", a);
9  }
```

Output:

```
trippled to 15
a: 5
```

a is **copied** into *x*, so modifying *x* does not change *a*.

Modifying in functions

```
1  void tripple(int &x){
2      x *= 3;
3      print("trippled to", x);
4  }
5  signed main(){
6      int a = 5;
7      tripple(a);
8      print("a:", a);
9  }
```

Modifying in functions

```
1  void tripple(int &x){
2      x *= 3;
3      print("trippled to", x);
4  }
5  signed main(){
6      int a = 5;
7      tripple(a);
8      print("a:", a);
9  }
```

Output:

```
trippled to 15
a: 15
```

x is a **reference** pointing to a, so modifying x does change a.

Performance

```
1  vector<int> append_pm(vector<int> v, int val){
2      v.push_back(val);
3      v.push_back(-val);
4      return v;
5  }
6  signed main(){
7      vector<int> nums;
8      for(int i = 1; i < 100000; ++i){
9          nums = append_pm(nums, i);
10     }
11 }
```

nums is **copied** to *v* every time. This is slow ($\approx$ 4 seconds).

Performance

```
1  void append_pm(vector<int> &v, int val){
2      v.push_back(val);
3      v.push_back(-val);
4  }
5  signed main(){
6      vector<int> nums;
7      for(int i = 1; i < 100000; ++i){
8          append_pm(nums, i);
9      }
10 }
```

v points to *nums*, **no copy** created. This is fast (< 0.01 seconds).

Call by value

Use call *by value* if you want a copy that can be changed independently.

```
1  int next_odd_square(int x){
2      if(x % 2 == 0) ++x;
3      return x * x;
4  }
```

Call by reference

Use call *by reference* if you want to modify the original inside the function.

```
1  void swap_ints(int &a, int &b){
2      int tmp = a;
3      a = b;
4      b = tmp;
5  }
```

(Of course, you could just use `swap(a, b)` from the STL in this example.)

Call by const reference

Use call *by const reference* if you don't modify the variable inside the function.

```
1 int square(int const&x){
2     return x * x;
3 }
```

This is the most common case and *const* helps you catch bugs.

```
1 void add_x_to_y(int const&x, int &y){
2     // error: assignment of read-only reference 'x'
3     x += y;
4     // correct would be y+=x;
5 }
```

Note that *y* is passed by reference.

Local reference variables

You can also declare local variables as references.

```
1 vector<int> table;
2 void process(int const&x){
3     int &val = table[x];
4     val = 3 * val + 1;
5     while(val % 2 == 0){
6         val /= 2;
7     }
8 }
```

This avoids writing `table[x]` every time.

Dangling references

References should not outlife the variable they point to!

```
1  int& sum(int const x, int const&y){
2      int ret = x + y;
3      // A reference to ret is returned,
4      return ret;
5      // but ret leaves the scope here.
6  }
7  signed main(){
8      int &val = sum(2, 3);
9      // val points to ret, but ret no longer exists!
10 }
```

This is undefined behaviour.

Auto

Spelling out the type can be annoying.

```
1 vector<vector<int> > v;
2 vector<vector<int> >::iterator it = v.begin();
```

With auto you can avoid it.

```
1 vector<vector<int> > v;
2 auto it = v.begin();
```

Auto and references

You can combine auto with references.

```
1  int x = 3;
2  auto &b = x;
3  b = 2; // now x is 2
4  vector<int> v {1, 2, 3};
5  auto &c = v[1];
6  c = 6; // v is now [1, 6, 3]
7  v.resize(10); // c is now dangling -> DANGER
8  c = 12; // undefined behaviour
```

Ranged base for loops

Instead of

```
1  vector<vector<int> > v(n, vector<int>(m));
2  for(int i = 0; i < n; ++i){
3      for(int j = 0; j < m; ++j){
4          v[i][j] = read_int();
5      }
6  }
```

you can do

```
1  vector<vector<int> > v(n, vector<int>(m));
2  for(auto &row : v){
3      for(auto &element : row){
4          element = read_int();
5      }
6  }
```

Default initialization

Non-class types and arrays get initialized to indeterminate values.

```
signed main(){
    int x; // x has indeterminate value
    print(x); // undefined behaviour
    array<int, 3> v; // indeterminate values
    print(v[1]); // undefined behaviour
}
```

Class types get initialized by calling the default constructor.

```
signed main(){
    vector<int> v; // well defined, v is an empty vector
}
```

Zero initialization

You can intialize to 0 with brace initialzation.

```
signed main(){
    int x{}; // x is zero
    int y = 0; // y is zero too
    print(x, y); // prints 0 0
    array<int, 3> v{}; // v is {0, 0, 0}
    print(v[1]); // prints 0
}
```

This also works with class types.

```
signed main(){
    vector<int> v{}; // v is an empty vector
}
```

Initialization: Things to avoid

There's no need to call the constructor explicitly.

```
signed main(){
    vector<vector<int> > v = vector<vector<int> >();
    // just use vector<vector<int> > v{};
}
```

You can't use () for zero initialization, as that declares a function.

```
signed main(){
    int a();
    // a is a function that takes no arguments
    // and returns an int
}
```

Characters

Use `char` to store single characters.

```
1 char a = 'a';
2 char zero = '0';
3 // characters convert to integers
4 // see man ascii
5 char b = a + 1;
6 char nine = zero + 9;
7 print(a, b, zero, nine);
```

Note that char promotes to int in operations.

```
1 char a = 'a';
2 print(a+1); // prints 98 (=ascii value of 'b')
```

Strings

Use `string` to store single characters. This is more convenient than using `vector<char>`.

```
1  string s = "abc";
2  string t = "123";
3  string st = s + t; // concatenate
4  s += t;
5  print(s, t, st);
6  // abc123 123 abc123
7  t = read_string(); // read from stdin
```

String operations

```
1  string s = "abcdef"
2  string cd = s.substr(2, 2); // (pos, length)
3  int pos = s.find("de") // 3
4  string aaaa(4, 'a'); // (length, character)
```

Struct

Suppose we want to store 2-dimensional points with an id. Using multiple vectors is quite cumbersome.

```
1  vector<int> x(n), y(n), id(n);
2  for(int i=0;i<n;++i){
3      x[i] = read_int();
4      y[i] = read_int();
5      id[i] = read_int();
6  }
7  // How do we sort them by x-coordinate?
8  // sort(x.begin(), x.end())
9  // -> y and id don't match anymore.
```

Struct

A struct can bundle values together.

```
1  struct Point{
2      int x, y, id;
3  };
4  Point origin{0, 0, -1}; // {x, y, id}
5  vector<Point> points(n);
6  for(Point &e:points){
7      e.x = read_int();
8      e.y = read_int();
9      e.id = read_int();
10 }
11 // Now we can sort them
12 sort(points.begin(), points.end(), [](Point const&a,
   ↪  Point const&b){return a.x < b.x;});
```

Operator overloading

```
1  struct Point{
2      int x, y, id;
3  };
4  // compare by x-coordinate
5  bool operator<(Point const&a, Point const&b){
6      return a.x < b.x;
7  }
8  bool operator==(Point const&a, Point const&b){
9      return a.x == b.x;
10 }
11 Point operator+(Point const&a, Point const&b){
12     return Point{a.x + b.x, a.y + b.y, -1};
13 }
14 Point x{1, 0, 1}, y{0, 1, 2};
15 Point z = x + y;
16 if(y < z) x = y;
```

Pairs

A pair has two values: “first” and “second”.

```
1  pair<int, char> p{42, 'x'};
2  print(p.first, p.second); // 42 x
3  pair<int, char> q = make_pair(42, 'x');
4  p.second = 'n';
5  if (p < q) print("Yes"); // lexicographic comparison
6  if(p == q)  print("Nope");
7  int a;
8  char c;
9  tie(a, c) = p; // unpack pair
10 auto [u, v] = p; // unpack with c++17, works in for loops
11 vector<pair<int, int> > v; // container of pairs
```

Tuple

Nested pairs can get messy.

```
pair<pair<int, int>, pair<pair<bool, int>, char> > p;
p.second.first.second = 3;
```

A tuple can store any fixed number of variables.

```
tuple<int, int, bool, int, char> p;
get<4>(p) = 'x';
print(get<0>(p));
```

You can also use an array if all types are equal.

```
array<int, 5> a; // size know at compile time
a[2] = 1;
vector<array<int, 3> >; // works with containers
```

This can quickly get *messy*. (What was get<3>(p) again?). Use a struct instead!