Advanced C++

Daniel Rutschmann

Swiss Olympiad in Informatics

October 11, 2018

Modifying in functions

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

Modifying in functions

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

Modifying in functions

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

Output:

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

a is a **reference** pointing to *x*, 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_mp(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_mp(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 `std::swap(a, b)` 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<vector<int> > table;
2 void process(int const&x, int const&y){
3     int &val = table[x][y];
4     val = 3 * val + 1;
5     while(val % 2 == 0){
6         val/=2;
7     }
8 }
```

This avoids writing `table[x][y]` 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.

Default initialization

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

```
signed main(){
    int x; // x has indeterminate value
    cout << x << "\n" // undefined behaviour
    array<int, 3> v; // indeterminate values
    cout << v[1] << "\n" // 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
    cout << x << " " << y << "\n" // prints 0 0
    array<int, 3> v{}; // v is {0, 0, 0}
    cout << v[1] << "\n" // 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.

```
1 signed main(){
2     vector<vector<int> > v = vector<vector<int> >();
3     // just use vector<vector<int> > v{};
4 }
```

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

```
1 signed main(){
2     int a();
3     // a is a function that takes no arguments
4     // and returns an int
5 }
```

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  cout << a << b << " " << zero << nine;
```

Note that char promotes to int in operations.

```
1  char a = 'a';
2  cout << 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  cout << s << " " << t << " " << st << "\n";
6  // abc123 123 abc123
7  cin >> t; // 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     cin >> x[i] >> y[i] >> id[i];
4 }
5 // How do we sort them by x-coordinate?
6 // sort(x.begin(), x.end())
7 // -> y and id don't match anymore.
```

Struct

A struct can bundle values together.

```
1  struct Point{
2      int x, y, id;
3  };
4  // list initialization {x, y, id}
5  Point origin{0, 0, -1};
6  vector<Point> points;
7  for(Point &e:points){
8      cin >> e.x >> e.y >> e.id;
9  }
10 // Now we can sort them
11 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      // compare by x-coordinate
4      bool operator<(Point const&o)const{
5          return x < o.x;
6      }
7      bool operator==(Point const&o)const{
8          return x == o.x;
9      }
10     // add two points
11     Point operator+(Point const&o)const{
12         return Point{x+o.x, y+o.y, -1};
13     }
14 };
15 Point x{1, 0, 1}, y{0, 1, 2};
16 Point z = x + y;
17 if(y < z) x = y;
```

Pairs

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

```
1  pair<int, char> p(42, 'x');
2  cout << p.first << " " << p.second; // 42 x
3  pair<int, char> q = make_pair(42, 'x');
4  p.second = 'n';
5  if (p < q) cout << "Yes"; // lexicographic comparison
6  if(p == q)  cout << "Nope";
7  int a;
8  char c;
9  tie(a, c) = p; // unpack pair
10 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<3>(p) =  3;
cout << get<0>(p);
get<4>(p) = 'x';
```

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

```
array<int, 5> a;
a[2] = 1;
```

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