Chapter 8: Technicalities: Functions, etc.

Krzysztof Dymkowski

October 27, 2015

Declaration and definition: variable

Declaration:

```
int i;
```

Initialization:

```
i=1 ;
```

Definition (Declaration + Initialization):

```
int i=1 ;
```

- The type is defined (can be done only once)
- Name must be declared before it can be used
- Declared variables should be initialized as soon as possible (allocation of memory)
- The type and the value are defined in one statement

Declaration and definition function

Declaration:

```
double add(double n, double m);
```

Definition:

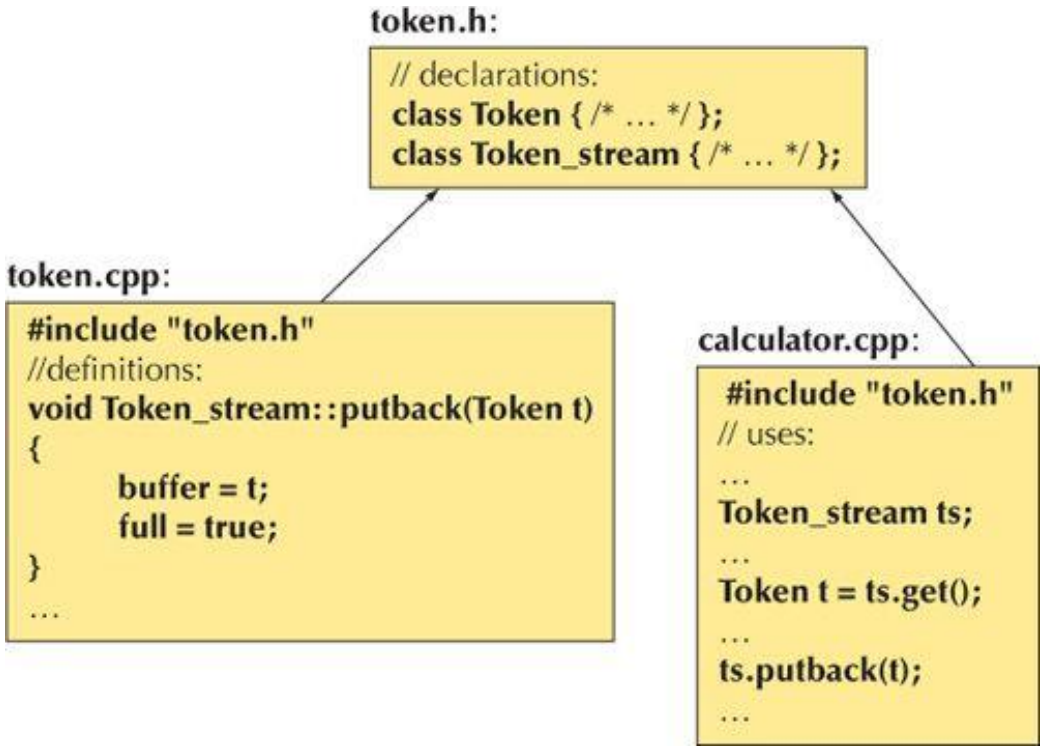
```
double add(double n, double m) {
return n+m;
}
```

- The type of input parameters and return type is given
- Definition includes body of the function (all statements which are executed when function is called)

- Names of parameters help to write description of function but can be left out if necessary
- return-statement returns value from a function (it must be present if function is not void)

headers

- When project is big it is usually divided into smaller pieces, each in a different directory
- Usually one puts declarations of functions in separate files: header files and include them at the beginning of other file if needed



- # include <....> : header file from system library
- # include ¨ ....¨ : user-defined header file

scope

- Scope: a region of program text where variable is "visible"
- Scope is defined by $\{\ldots\}$. It can be put anywhere in the code.
- Body of function enclosed within $\{\ldots\}$ also defines a scope (similar for classes, namespaces, statements like if etc.)
- Special type of scope: *global scope*. The area of text in the whole file

```
// no r, i, or v here
class My_vector {
        vector<int> v;      // v is in class scope
public:
        int largest()
        {
                int r = 0;                      // r is local (smallest nonnegative int)
                for (int i = 0; i<v.size(); ++i)
                        r = max(r,abs(v[i]));   // i is in the for's statement scope
                // no I here
                return r;
        }
        // no r here
};
// no v here

int x;                  // global variable — avoid those where you can
int y;
```

- Variable defined in a scope cannot be seen outside.
- Names in a scope can be seen from within scopes nested within it

Hiding (shadowing) variables

```
// no r, i, or v here
class My_vector {
        vector<int> v;      // v is in class scope
public:
        int largest()
        {
                int r = 0;                      // r is local (smallest nonnegative int)
                for (int i = 0; i<v.size(); ++i)
                        r = max(r,abs(v[i]));   // i is in the for's statement scope
                // no I here
                return r;
        }
        // no r here
};
// no v here

int x;                  // global variable — avoid those where you can
int y;

int f()
{
        int x;          // local variable, hides the global x
        x = 7;          // the local x
        {
                int x = y;  // local x initialized by global y, hides the previous local x
                ++x;        // the x from the previous line
        }
        ++x;            // the x from the first line of f()
        return x;
}
```

- Names live within their scope, each time local scope is entered local variables are created and they are destroyed when program lefts local scope
- Function f returns 2 instead of 9
- Try to avoid global variables!
- The larger the scope of a name is, the longer and more descriptive its name should be
- Use indentation to mark a scope

Function parameters: pass by value

```
// pass-by-value (give the function a copy of the value passed)
int f(int x)
{
        x = x+1;                    // give the local x a new value
        return x;
}

int main()
{
        int xx = 0;
        cout << f(xx) << '\n';      // write: 1
        cout << xx << '\n';         // write: 0; f() doesn't change xx

        int yy = 7;
        cout << f(yy) << '\n';      // write: 8
        cout << yy << '\n';         // write: 7; f() doesn't change yy
}
```

- when pass by value an argument is copied at the beginning and function works on its copy without modifying an original value.
- optimal if parameters don't accupy to much memory. Not so good for large arrays (costly copying)

Function parameters: pass-by-const-reference

```
void print(const vector<double>& v)    // pass-by-const-reference
{
        cout << "{ ";
        for (int i = 0; i<v.size(); ++i) {
                cout << v[i];
                if (i!=v.size()–1) cout << ", ";
        }
        cout << " }\n";
}
```

- The & means "reference"
- Pass by reference means send an address of memory where object is located
- const means that a function cannot modify input parameter

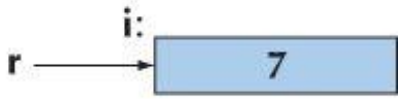
Function parameters: pass-by-reference

```
void init(vector<double>& v)          // pass-by-reference
{
        for (int i = 0; i<v.size(); ++i) v[i] = i;
}

void g(int x)
{
        vector<double> vd1(10);       // small vector
        vector<double> vd2(1000000);  // large vector
        vector<double> vd3(x);        // vector of some unknown size

        init(vd1);
        init(vd2);
        init(vd3);
}
```

- Function init works on the original object and can modify it
- In general references are useful when we have to manipulate few objects at the same time or the object is really large

References

```
int i = 7;

int& r = i;       // r is a reference to i
r = 9;            // i becomes 9
i = 10;
cout << r << ' ' << i << '\n';      // write: 10 10
```



- Whenever we use r we actually use i

References are great but ...:

- Use return statement to return a result of a function instead of modifying objects sent by reference
- You should use pass-by-reference only when you have to

More about functions

- Stack of activation records : all input, local parameter + "implementation stuff"
- constexpr functions:
  - to evaluate expression at compile time
  - arguments are constant expressions (known at the compile time)

```
constexpr double xscale = 10;      // scaling factors
constexpr double yscale = 0.8;

constexpr Point scale(Point p) { return {xscale*p.x,yscale*p.y}; };
```

Expression evaluation

How you shouldn't do...

```
v[i] = ++i;                    // don't: undefined order of evaluation
v[++i] = i;                    // don't: undefined order of evaluation
int x = ++i + ++i;             // don't: undefined order of evaluation
cout << ++i << ' ' << i << '\n';   // don't: undefined order of evaluation
f(++i,++i);                    // don't: undefined order of evaluation
```

- Undefined behaviour so...
- Don't read or write variable twice in that same expression

More about scope

- The static local variable:
  - is initialized (constructed) only the first time its function is called.
  - its value is remembered after every execution of function (it behaves like global variable but it is accessible only but its function)
- Namespaces: we can group functions, constants and even classes in the convenient way:

```
namespace TextLib {
    class Text { /* ... */ };
    class Glyph { /* ... */ };
    class Line { /* ... */ };
    // ...
}
```

- One can later refer to those classes for example like:
  - TextLib::Text (this inside of functions) or
  - using TextLib::Text (this should be put at the beginning)
- statement *using namespace TextLib;*

  Makes classes Text, Glyph and Line directly accessible
- However one should use directive "using" with caution (one may lose track of which names come from where...)