DM560

Introduction to Programming in C++

Technicalities: Declarations, Scopes, Functions, Namespaces

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

[*Based on slides by Bjarne Stroustrup*]

Outline

C++ Users: CERN's Higgs Boson Hunt

(2012) Higgs boson found! In email from CERN:

"all related computing done in C++."

and

a follow-up from Fermilab:

"The reports are correct, almost all significant computing done for high energy physics (and not only the LHC) is done in C++. And many (though not all) experiments are now in the process of migrating to the use of C++11."

Sources:

- The Big Data Software Problem Behind CERN's Higgs Boson Hunt
- Root a data analysis framework based on C++ https://root.cern.ch//

C++ Users: Mars Rover

https://www.youtube.com/watch?v=3SdSKZFoUa8
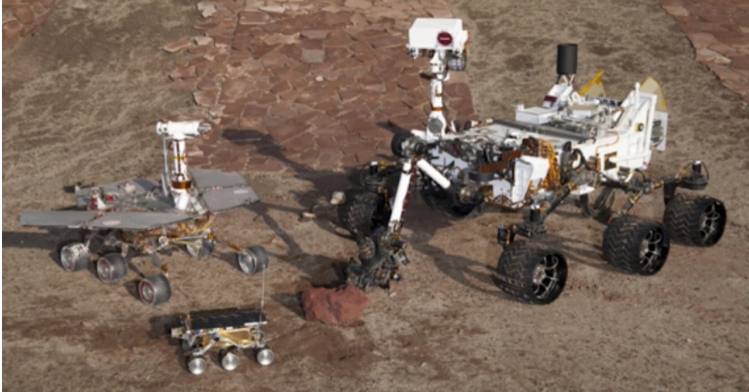
Overview

Goals:

- broaden the view of C++'s basic facilities (types, functions, and initialization)
- provide a more systematic view of those facilities.

Contents:
- Language Technicalities
- Declarations
    - Definitions
    - Headers and the preprocessor
    - Scope
- Functions
    - Declarations and definitions
    - Arguments
    - Call by value, reference, and const reference
- Namespaces
  using declarations

Language Technicalities

- They are like the vocabulary and the grammar in a foreign language

- In a formal language the rules governing the composition of well-formed expressions are referred to as syntax

- We need to learn them because programs must be precisely and completely specified
  - A computer is a very stupid (though very fast) machine
  - A computer can't guess what you "really meant to say" (and shouldn't try to)

- However, here, we will look at only some of the C++ rules:
  (the C++14 standard is 1,358 pages)

Language Technicalities

But! Don't spend your time on minor syntax and semantic issues

⤳ Remember: we study programming concepts, the programming language is only a tool.

- Most design and programming concepts are universal, or at least very widely supported by popular programming languages
  So what you learn using C++ you can use with many other languages

- Language technicalities are specific to a given language
  But many of the technicalities from C++ presented here have obvious counterparts in C, Java, C#, etc.

- Moreover, there is more than one way to say everything

Outline

Declarations

A name must be declared before it can be used in a C++ program.

- A declaration introduces a name into a scope.

- A declaration also specifies a type for the named object.

- Sometimes a declaration includes an initializer.

Examples:

```
int a = 7;              // an int variable named 'a' is declared
const double cd = 8.7;  // a double-precision floating-point constant
double sqrt(double);    // a function taking a double argument and
                        //  returning a double result
vector<Token> v;        // a vector variable of Tokens (variable)
```

Declarations

A header is a file containing declarations providing an interface to other parts of a program

- Declarations are frequently introduced into a program through headers

- This allows for abstraction—you don't have to know the details of a function like cout in order to use it.
  When you add:

  ```
  #include "std_lib_facilities.h"
  ```

  to your code, the declarations in the file std_lib_facilities.h become available (including cout, etc.).

Examples

At least three errors:

```
int main()
{
    cout << f(i) << '\n';
}
```

Add declarations:

```
#include <iostream>   // we find the declaration of cout in here

using namespace std;

int main()
{
    cout << f(i) << '\n';
}
```

Examples

Define your own functions and variables:

```cpp
#include <iostream>    // we find the declaration of cout in here

using namespace std;

int f(int x ) { /* ... */ }     // declaration of f

int main()
{
    int i = 7;      // declaration of i
    cout << f(i) << '\n';
}
```

Definitions

A definition is a declaration that (also) fully specifies the entity declared

Examples:

```
int a = 7;
int b;                    // an (uninitialized) int
vector<double> v;         // an empty vector of doubles
double sqrt(double) { ... };  // a function with a body
struct Point { int x; int y; };
```

Examples of declarations that are not definitions

```
double sqrt(double);      // function body missing
struct Point;             // class members specified elsewhere
extern int a;             // extern means "not definition"
                          // "extern" is archaic; we will hardly use it
```

Declarations and Definitions

- You *cannot* define something twice.
  A definition says what something is.

  Examples

```
int a;          // definition
int a;          // error: double definition
double sqrt(double d) { ... }   // definition
double sqrt(double d) { ... }   // error: double definition
```

- You can declare something twice
  A declaration says how something can be used

  Example:

```
int a = 7;                      // definition (also a declaration)
extern int a;                   // declaration
double sqrt(double);            // declaration
double sqrt(double d) { ... }   // definition (also a declaration)
```

Why both Declarations and Definitions?

- To refer to something, we need (only) its declaration

- Often we want the definition "elsewhere"
  Later in a file, in another file possibly written by someone else

- Declarations are used to specify interfaces:
  to your own code, and to libraries
  ⇝ Libraries are key: we can't write all ourselves, and wouldn't want to

- In larger programs, place all declarations in header files to ease sharing

Kinds of Declarations

The most interesting are

- Variables
  ```
  int x;
  vector<int> vi2 {1,2,3,4};
  ```

- Constants
  ```
  void f(const X&);
  constexpr int i = sqrt(2);
  ```

- Functions
  ```
  double sqrt(double d) { /*... */}
  ```

- Namespaces

- Types (classes and enumerations)

- Templates (see Chapter 19)
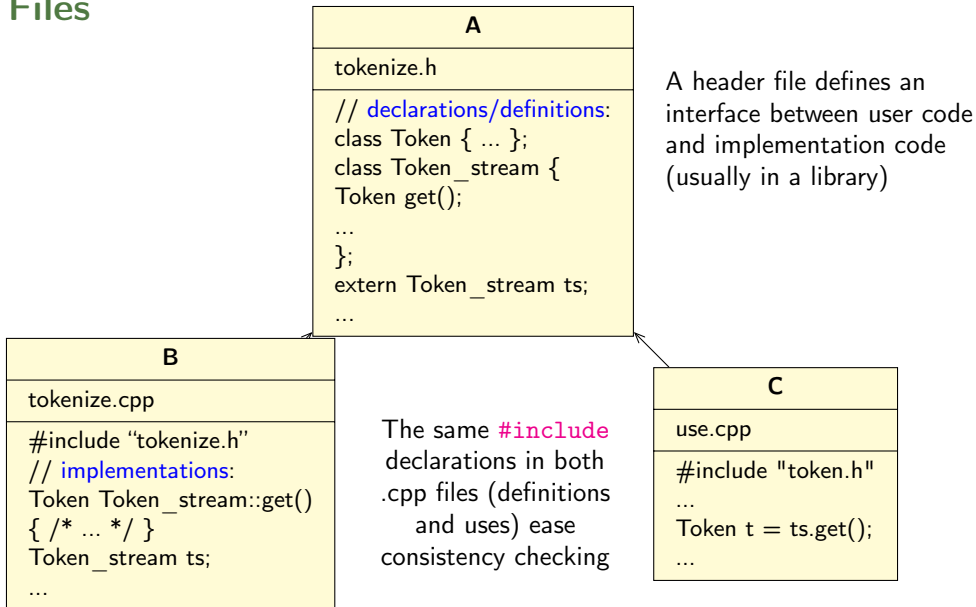
Header files and Preprocessor

- A header is a file that holds declarations of functions, types, constants, and other program components.

- The construct

```
#include "std_lib_facilities.h"
```

  is a preprocessor directive that adds declarations to your program
  Typically, the header file is simply a text (source code) file

- A header gives you access to functions, types, etc. that you want to use in your programs.
  - Usually, you don't really care about how they are written.
  - The actual functions, types, etc. are defined in other source code files
  - Often as part of libraries

Source Files

A

tokenize.h

// declarations/definitions:
class Token { ... };
class Token_stream {
Token get();
...
};
extern Token_stream ts;
...

A header file defines an interface between user code and implementation code (usually in a library)

B

tokenize.cpp

#include "tokenize.h"
// implementations:
Token Token_stream::get()
{ /* ... */ }
Token_stream ts;
...

The same #include declarations in both .cpp files (definitions and uses) ease consistency checking

C

use.cpp

#include "token.h"
...
Token t = ts.get();
...

Conditional Compilation

A common use of preprocessor is to avoid reading files more than once:

```
#ifndef MY_TOKENIZE_H
#define MY_TOKENIZE_H
  /* here the header information */
#endif
```

These directives are called macros

```
#ifdef WINDOWS
  #include ''my_windows_header.h''
#else
  #include ''my_linux_header.h''
#endif
```

Outline

Scope

- A scope is a region of program text
  - Global scope (outside any language construct)
  - Class scope (within a class)
  - Local scope (between { ... } braces)
  - Statement scope (e.g. in a for-statement)

- A name in a scope can be seen from within its scope and within scopes nested within that scope
  - Only after the declaration of the name ("can't look ahead" rule)
  - Class members can be used within the class before they are declared

- A scope keeps "things" local
  - Prevents one's variables, functions, etc., from interfering with others'
    Remember: real programs have many thousands of entities
  - Locality is good! Keep names as local as possible

Scope

```cpp
#include "std_lib_facilities.h"        // get max and abs from here
// no r, i, or v here
class My_vector {
  vector<int> v;                       // v is in class scope
public:
  int largest()                        // largest is in class scope
  {
    int r = 0;                         // r is local
    for (int i = 0; i<v.size(); ++i)   // i is in statement scope
      r = max(r,abs(v[i]));
    // no i here
    return r;
  }
  // no r here
};
// no v here
```

Scopes Nest

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

int f()
{
  int x;                  // local variable (Note - now there are two x's)
  x = 7;                  // local x, not the global x
  {
    int x = y;  // another local x, initialized by the global y
                // (Now there are three x's)
    ++x;                  // increment the local x in this scope
  }
}
```

Avoid such complicated nesting and hiding: keep it simple!

Outline

Recap: Why Functions?

- Chop a program into manageable pieces
  "divide and conquer"

- Match our understanding of the problem domain
  - Name logical operations
  - A function should do one thing well

- Functions make the program easier to read

- A function can be useful in many places in a program

- Ease testing, distribution of labor, and maintenance

- Keep functions small
  Easier to understand, specify, and debug

Functions

- General form:

```
return_type name (formal arguments);            // a declaration
return_type name (formal arguments) body        // a definition
```

  For example:

```
double f(int a, double d) { return a*d; }
```

- Formal arguments are often called parameters
- If you don't want to return a value give void as the return type

```
void increase_power_to(int level);
```

  Here, void means "doesn't return a value"

- A body is a block or a try block
  For example

```
{ /* code */ }  // a block
try { /* code */ } catch(exception& e) { /* code */ }   // a try block
```
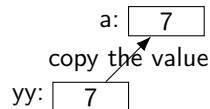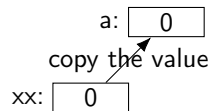
- Functions represent/implement computations/calculations

Functions: Call by Value

call-by-value ≡ send the function a copy of the argument's value

```cpp
int f(int a) { a = a+1; return a; }

int main()
{
  int xx = 0;
  cout << f(xx) << '\n';         // writes 1
  cout << xx << '\n';    // writes 0; f() doesn't change xx
  int yy = 7;
  cout << f(yy) << '\n'; // writes 8;  f() doesn't change yy
  cout << yy << '\n';    // writes 7
}
```
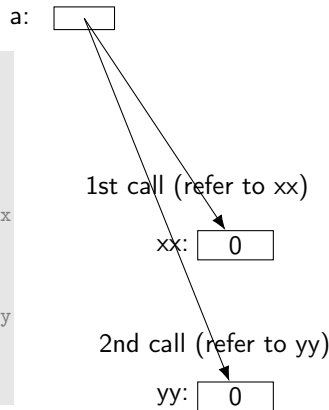
a: ☐ 0
copy the value
xx: ☐ 0

a: ☐ 7
copy the value
yy: ☐ 7

Functions: Call by Reference

call-by-reference ≡ pass a reference to the argument

```
int f(int& a) { a = a+1; return a; }

int main()
{
        int xx = 0;
        cout << f(xx) << '\n';   // writes 1
                                 // f() changed the value of xx
        cout << xx << '\n';      // writes 1
        int yy = 7;
        cout << f(yy) << '\n';   // writes 8
                                 // f() changes the value of yy
        cout << yy << '\n';      // writes 8

}
```

a: ☐

1st call (refer to xx)

xx: ☐ 0

2nd call (refer to yy)

yy: ☐ 0

Functions

- Avoid (non-const) reference arguments when you can
  They can lead to obscure bugs when you forget which arguments can be changed

```
int incr1(int a) { return a+1; }
void incr2(int& a) { ++a; }
int x = 7;
x = incr1(x);   // pretty obvious
incr2(x);       // pretty obscure
```

- So why have reference arguments?
    - Occasionally, they are essential
      E.g., for changing several values
      For manipulating containers (e.g., vector)
    - const reference arguments are very often useful

Call by Value / by Reference / by const-Reference

```
void f(int a, int& r, const int& cr) { ++a; ++r; ++cr; } // error: cr is const
void g(int a, int& r, const int& cr) { ++a; ++r; int x = cr; ++x; } // ok

int main()
{
  int x = 0;
  int y = 0;
  int z = 0;
  g(x,y,z);      // x==0; y==1; z==0
  g(1,2,3);      // error: reference argument r needs a variable to refer to
  g(1,y,3);      // ok: since cr is const we can pass ''a temporary''
}
```
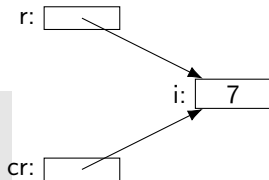
const references are very useful for passing large objects

References

- reference is a general concept
  Not just for call-by-reference

```cpp
int i = 7;
int& r = i;
r = 9;  // i becomes 9
const int& cr = i;
// cr = 7;       // error: cr refers to const
i = 8;
cout << cr << endl;  // write out the value of i (that's 8)
```

- You can
  think of a reference as an alternative name for an object

- You can't
  modify an object through a const reference
  make a reference refer to another object after initialization

Example

A range-for loop:

```
for (string s : v) cout << s << '\n';              // s is a copy of some v[i]
for (string& s : v) cout << s << '\n';             // no copy
for (const string& s : v) cout << s << '\n';       // and we don't modify v
```

Compile-time Functions

You can define functions that can be evaluated at compile time: constexpr functions

```
constexpr double xscale = 10;    // scaling factors
constexpr double yscale = .8;

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

constexpr Point x = scale({123,456});    // evaluated at compile time

void use(Point p)
{
        constexpr Point x1 = scale(p);    // error: compile-time evaluation
                                          // requested for variable argument
        Point x2 = scale(p);              // OK: run-time evaluation
}
```

Note: these functions must be very simple, just a return statement.

Guidance for Passing Variables

- Use call-by-value for very small objects

- Use call-by-const-reference for large objects

- Use call-by-reference only when you have to

- Return a result rather than modify an object through a reference argument

For example:

```
class Image { /* objects are potentially huge */ };
void f(Image i);  ... f(my_image);   // oops: this could be s-l-o-o-o-w
void f(Image& i); ... f(my_image); // no copy, but f() can modify my_image
void f(const Image&); ... f(my_image);  // f() won't mess with my_image
Image make_image();    // most likely fast! (''move semantics'' - later)
```

Outline

Namespaces

Consider this code from two programmers Jack and Jill

```
class Glob { /*...*/ };         // in Jack's header file jack.h
class Widget { /*...*/ };       // also in jack.h
```

```
class Blob { /*...*/ };         // in Jill's header file  jill.h
class Widget { /*...*/ };       // also in jill.h
```

```
#include "jack.h";     // this is in your code
#include "jill.h";              // so is this

void my_func(Widget p)  // oops! - error: multiple definitions of Widget
{
  // ...
}
```

Namespaces

- The compiler will not compile multiple definitions; such clashes can occur from multiple headers.
- One way to prevent this problem is with namespaces:

```cpp
namespace Jack {                    //  in Jack's header file
  class Glob{ /*...*/ };
  class Widget{ /*...*/ };
}
```

```cpp
#include "jack.h";        // this is in your code
#include "jill.h";                  // so is this

void my_func(Jack::Widget p)    // OK, Jack's Widget class will not
{                                       // clash with a different Widget
  // ...
}
```

Namespaces

- A namespace is a named scope

- The `::` syntax is used to specify which namespace we are using and which (of many possible) objects of the same name we are referring to

- For example, `cout` is in namespace `std`, we could write:

```
std::cout << "Please enter stuff... \n";
```

using Declarations and Directives

- To avoid the tedium of

```
std::cout << "Please enter stuff... \n";
```

- you could write a using declaration

```
using std::cout;        // when I say cout, I mean std::cout
cout << "Please enter stuff... \n";     // ok: std::cout
cin >> x;               // error: cin not in scope
```

- or you could write a using directive

```
using namespace std; // ``make all names from namespace std available''
cout << "Please enter stuff... \n";     // ok: std::cout
cin >> x;                               // ok: std::cin
```

- More about header files in chapter 12

Summary