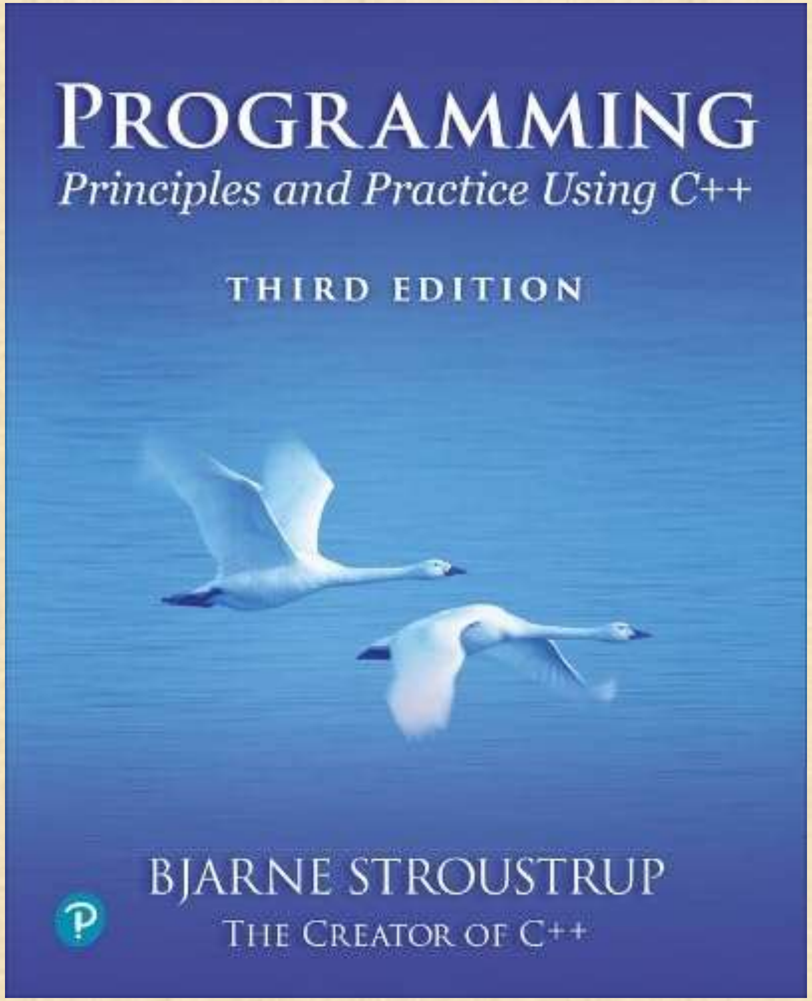
Chapter 7 – Technicalities, Functions, etc.



No amount of genius can overcome obsession with detail.
– Traditional

Abstract

- This lecture and the following present some technical details of the language to give a slightly broader view of C++'s basic facilities and to provide a more systematic view of those facilities. This also acts as a review of many of the notions presented so far, such as types, functions, and initialization, and provides an opportunity to explore our tool without adding new programming techniques or concepts.

Overview

- 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

- Are a necessary evil
  - A programming language is a foreign language
  - When learning a foreign language, you have to look at the grammar and vocabulary
  - We will do this in this chapter and the next
- 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)
  - So we must know the rules
    - Some of them (the C++23 standard is almost 1900 pages)
- However, never forget that
  - What we study is programming
  - Our output is programs/systems
  - A programming language is only a tool

Technicalities

- Don’t spend your time on minor syntax and semantic issues. There is more than one way to say everything
  - Just like in English
- 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.

Declarations

- A declaration introduces a name into a scope.
- A declaration also specifies a type for the named object.
- Sometimes a declaration includes an initializer.
- A name must be declared before it can be used in a C++ program.
- 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

- Declarations are frequently introduced into a program through "headers"
  - A header is a file containing declarations providing an interface to other parts of a program
- 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 "PPP.h"

  to your code, the PPP_support declarations (incl. the C++ standard library become available (including **cout**, etc.).

For example

- At least three errors:

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

- Add declarations:

```
#include "PPP.h"     // we find the declaration of cout in here

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

For example

- Define your own functions and variables:

```
#include "PPP.h"     // we find the declaration of cout in here

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

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

Definitions

A declaration that fully specifies the entity declared is called a definition

- 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
```

Declarations and definitions

- You can't *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

```
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
    - preferably written by someone else
- Declarations are used to specify interfaces
  - To your own code
  - 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 = isqrt(2);**
  - Functions (see §7.4)
    - **double sqrt(double d) { /* … */ }**
  - Namespaces (see §7.6)
  - Modules (see §7.7)
  - Types (classes and enumerations; see Chapter 8)
  - Templates (see Chapter 18)
  - Concepts (see §18.1.3)

Scope

- A program can have millions of lines of code
  - With 100,000s of declarations
  - Scopes are the construct we use to manage such masses of names
- The idea is simple
  - Keep a names local, except when we want it to be useful elsewhere
  - Locality is good!
  - Keep names as local as possible

```
double my_function(vector<int>& v)      // v is local to my_function (and refers to an argument)
{
        double sum = 0;                  // sum is local to my_function
        for (int x : v)                  // x is local to this for-statement
                sum+= x;
        return x;
}
```

Scope

- A scope is a region of program text
  - Global scope (outside any language construct)
  - Module scope
  - Namespace scope
  - Class scope
  - 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

```
int next(int x) {  return x+1; }
return next2(int x)  { next(next(x); }          // next() is visible from within next2()
```

- A name can be used only after the declaration of the name (“can’t look ahead” rule)
  - Except that class members can be used within the class before they are declared

Scope

- A module, namespace, or class scope has a name
- A namespace or class name can be used to refer to members from elsewhere

```
struct X {
        int set(int x)
        {
                int old = m;
                m = x;
                return old;
        }
        int m = 0;
};

X obj;
int old_x = X::set(7);                    // X's set (much more about classes in Chapter 8)
```

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!*

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

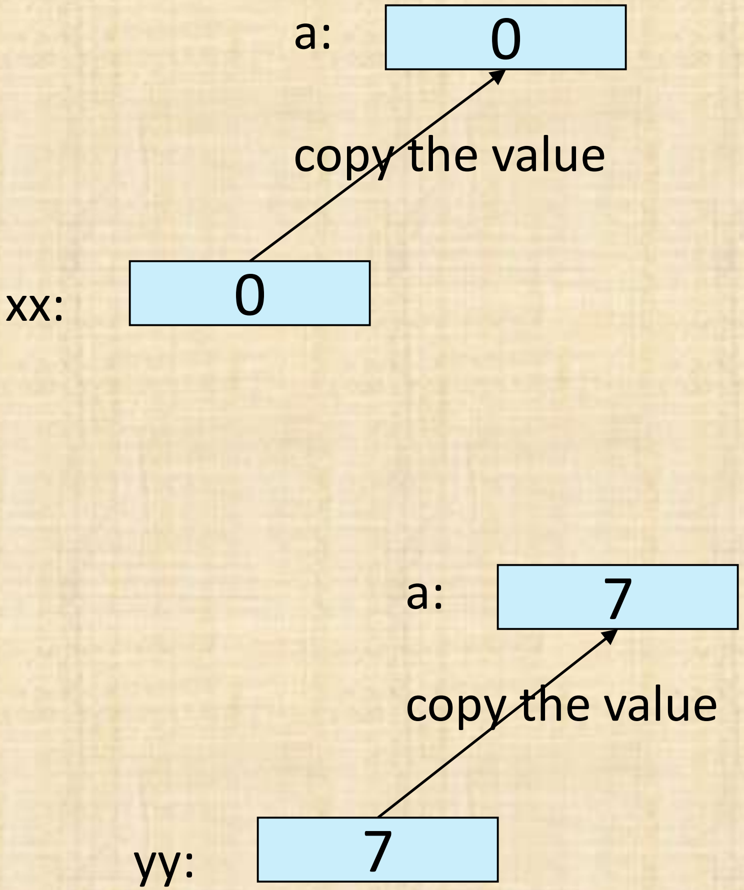
    ```
    double f(int a, double d) { return a*d; }
    ```
- Formal arguments are often called parameters
- If you don't want to return a value give **void** as the return type

  ```
  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

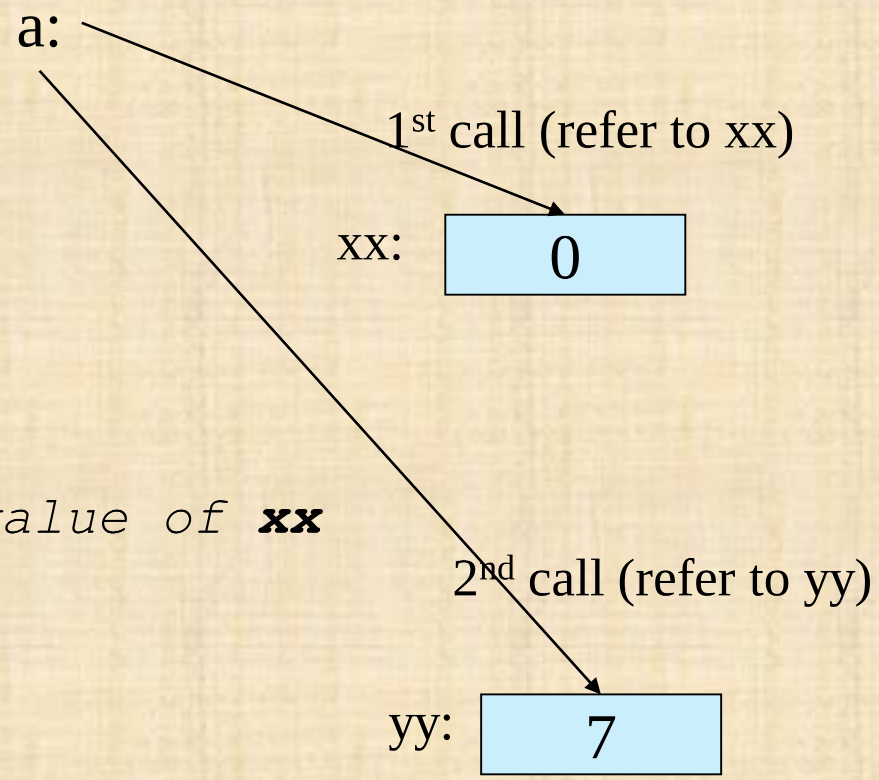
Functions: Call by Value

```
// call-by-value (send the function a copy of the argument's value)
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
}
```



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
}
```



Function call

- When you call a function, a “function activation record” hold its arguments, local variables, and what’s needed to return to the caller
  - For **expression()** from Chapter 7 (see 7.4.8), we need to store the argument **ts** and the local variables **left** and **t**:

Call of **expression()**:

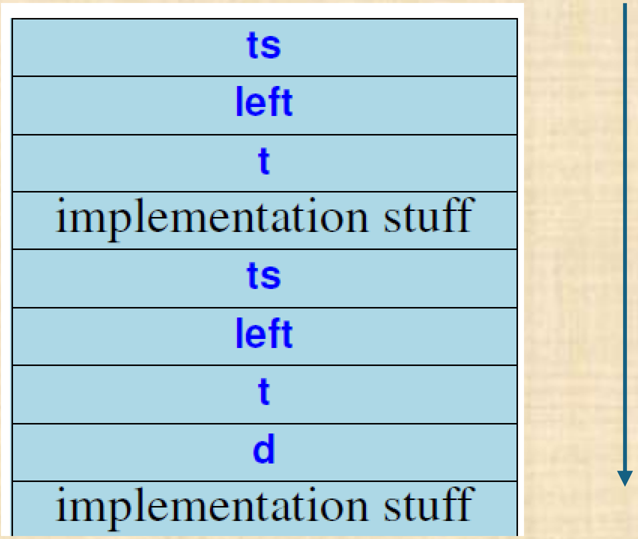
ts
left
t
implementation stuff

Function call

- Activation records are kept in a stack (the call stack).
- **expression()** calls **term()**:

Call of **expression():**

Call of **term():**



Function call

- **term()** calls **primary()**:

Call of **expression()**:	ts
	left
	t
	implementation stuff
Call of **term()**:	ts
	left
	t
	d
	implementation stuff
Call of **primary()**:	ts
	t
	d
	implementation stuff

Function call

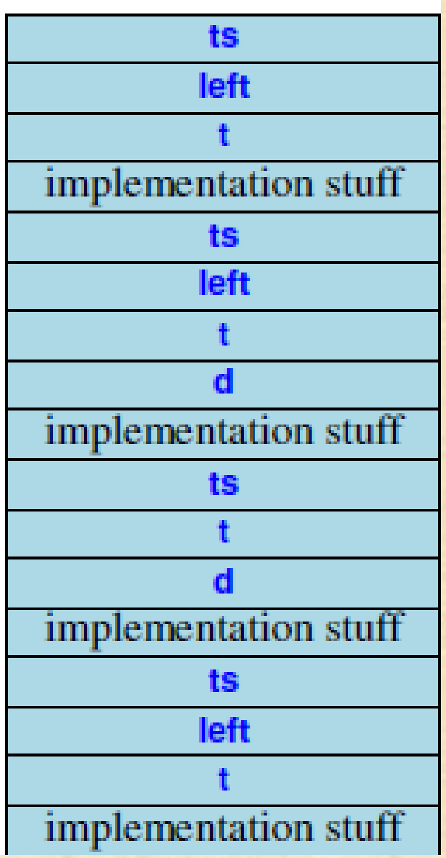
- **primary()** calls **expression()**:

Call of **expression()**:

Call of **term()**:

Call of **primary()**:

Call of **expression()**:



- And the on returns the stack frames are discarded
  - In the reverse order that they were created

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

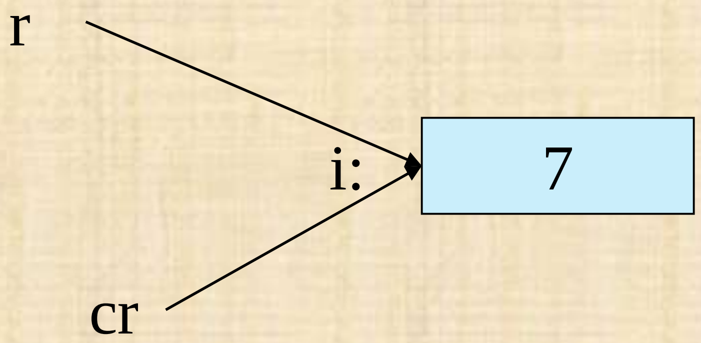
Call by value, reference, and const-reference

```
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"
}
```

References

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

```
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

For 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
```

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
```

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:

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

    #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 you are using and which (of many possible) objects of the same name you are referring to
- For example, **cout** is in namespace **std**, you can 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*

Next talk

- More technicalities, mostly related to classes