ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ

Εισαγωγή στον Προγραμματισμό
Introduction to Programming

Διάλεξη 7: Συναρτήσεις

Γ. Παπαγιαννάκης

Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο
ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ & ΘΡΗΣΚΕΥΜΑΤΩΝ, ΠΟΛΙΤΙΣΜΟΥ & ΑΘΛΗΤΙΣΜΟΥ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ
ΕΣΠΑ 2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ
Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

Άδειες Χρήσης

- Το παρόν εκπαιδευτικό υλικό υπόκειται στην άδεια χρήσης **Creative Commons** και ειδικότερα

Αναφορά Δημιουργού 3.0 - Μη εισαγόμενο Ελλάδα (Attribution 3.0– Unported GR)

- Για εκπαιδευτικό υλικό, όπως εικόνες, που υπόκειται σε άλλου τύπου άδειας χρήσης, η άδεια χρήσης αναφέρεται ρητώς.

Χρηματοδότηση

- Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στα πλαίσια του εκπαιδευτικού έργου του διδάσκοντα.

- Το έργο «**Ανοικτά Ακαδημαϊκά Μαθήματα στο Πανεπιστήμιο Κρήτης**» έχει χρηματοδοτήσει μόνο τη αναδιαμόρφωση του εκπαιδευτικού υλικού.

- Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.

Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο

ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ & ΘΡΗΣΚΕΥΜΑΤΩΝ, ΠΟΛΙΤΙΣΜΟΥ & ΑΘΛΗΤΙΣΜΟΥ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ

ΕΣΠΑ 2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ

Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

HY-150 Προγραμματισμός
CS-150 Programming

Lecture 7:
Technicalities: Functions etc.

G. Papagiannakis

Abstract

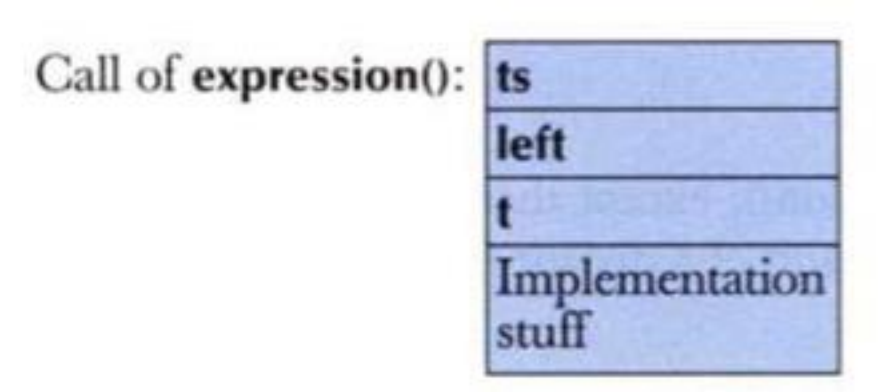
- This lecture and the following present some **technical details** of the language to give a slightly broader view of C++'s basic facilities and to provide a more systematic view of those facilities. This also acts as a **review** of many of the notions presented so far, such as **types**, **functions**,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

Latest standard version: C++11

- *...I like the way move semantics will simplify the way we return large data structures from functions and improve the performance of standard-library types, such as* *string* *and* *vector*...
- *For example, you wouldn't write a JavaScript engine in JavaScript, and you probably wouldn't write a "first to market" simple Web app in C++. You would write the foundations of a Google, an Amazon, a Facebook, or an Amadeus (airline ticketing) in C++, but maybe not the rapidly changing top layers of such systems. C++ comes in strong where power consumption is an issue -- for example, server farms and handheld devices....*

[http://www.infoworld.com/d/application-development/stroustrup-reveals-whats-new-in-c-11-187051?page=0,0]

Overview



- Language Technicalities
- Declarations
  - Definitions
  - Headers and the preprocessor
  - Scope
- Functions
  - Declarations and definitions
  - Arguments
- Call by value, reference, and **const** reference
- Namespaces
  - "Using" statements
- Recursive functions

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++ standard is 782 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.
  - Too many get the mistaken belief that the way things are done in their first programming language is "the one true way."

Declarations

- A declaration introduces a name into a scope.
  - A *scope is* a region of program text.
- 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 "../../std_lib_facilities.h"

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

Definitions

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

- Examples

```
int a = 7;
int b;                              // an int with the default value (0)
vector<double> v;                   // an empty vector of doubles
double sqrt(double) { … };          // i.e. 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 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

Header Files and the 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

token.h:

```
// declarations:
class Token { … };
class Token_stream {
    Token get();
    …
};
...
```

token.cpp:

```
#include "token.h"
//definitions:
Token Token_stream::get()
{ /* … */ }
…
```

use.cpp:

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

- A header file (here, **token.h**) defines an interface between user code and implementation code (usually in a library)
- The same **#include** declarations in both **.cpp** files (definitions and uses) ease consistency checking

Scope

- A scope is a region of program text
  - Examples
    - **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
    - After the declaration of the name (“can't look ahead” rule)
- A scope keeps “things” local
    - Prevents my variables, functions, etc., from interfering with yours
    - Remember: real programs have **many** thousands of entities
    - Locality is good!
      - Keep names as local as possible

Scope

```
#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
};//end of class
// 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!
```

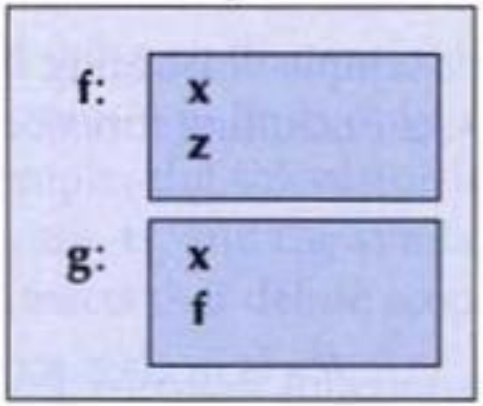
Global/local scope

```
void f(int x)          // f is global; x is local to f
{
        int z = x+7;   // z is local
}

int g(int x)           // g is global; x is local to g
{
        int f = x+2;   // f is local
        return 2*f;
}
```

Or graphically:

Global scope:



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(int level);
```

  - Here, **void** means "don't return a value"
- A body is a block or a try block
  - For example

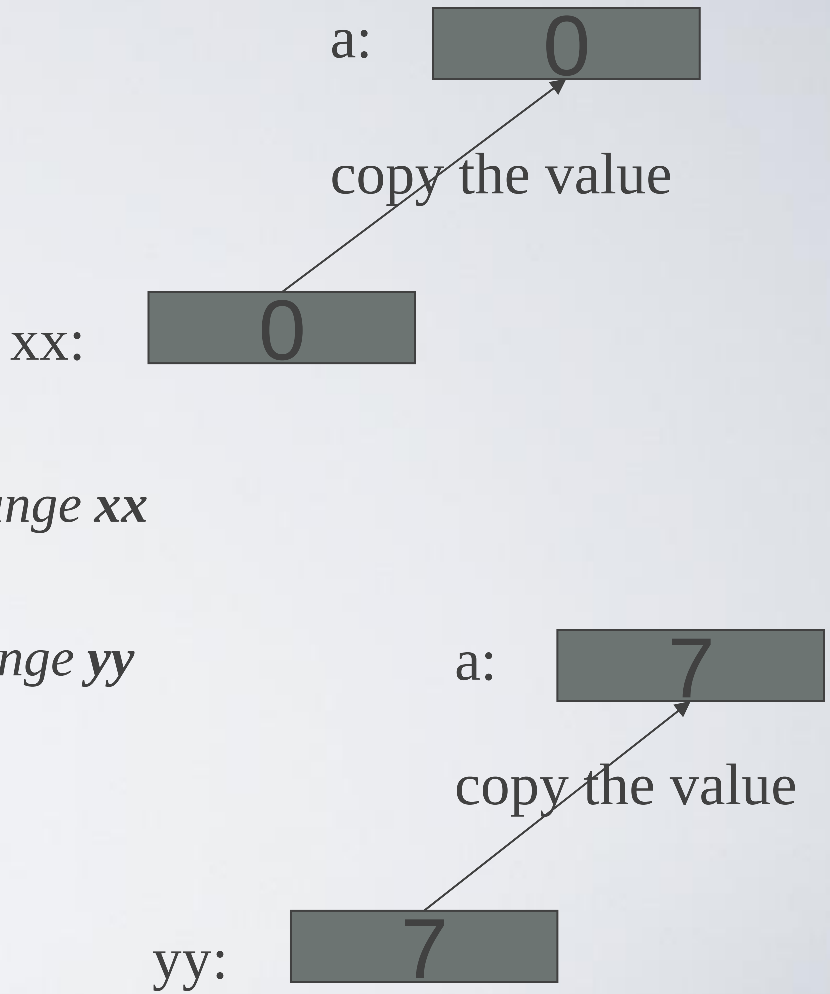
```
{ /* code */ }          // a block
try { /* code */ } catch(exception& e) { /* code */ }     // a try block
```

- 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) << endl;   // writes 1
   cout << xx << endl;      // writes 0; f() doesn't change xx
   int yy = 7;
   cout << f(yy) << endl; // writes 8;  f() doesn't change yy
   cout << yy << endl;      // writes 7
}
```

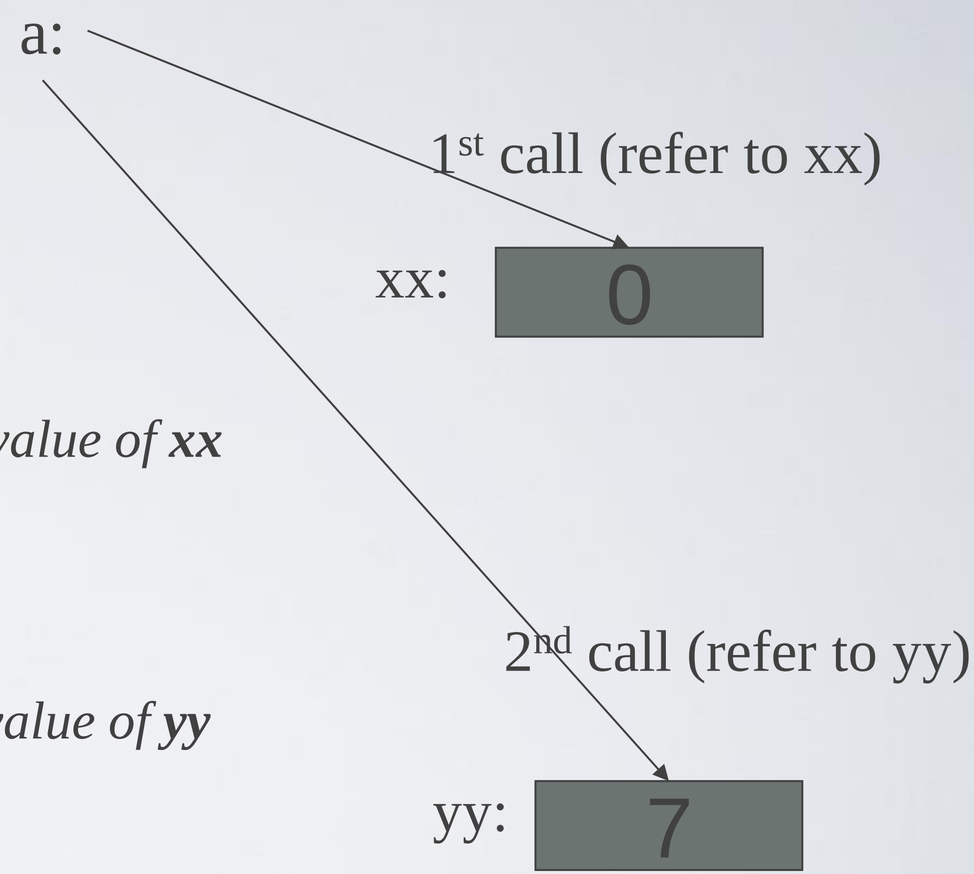


Functions: Call by Reference

```
// 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) << endl;   // writes 1
                             // f() changed the value of xx
    cout << xx << endl;      // writes 1
    int yy = 7;
    cout << f(yy) << endl; // writes 8
                             // f() changes the value of yy
    cout << yy << endl;      // writes 8

}
```



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

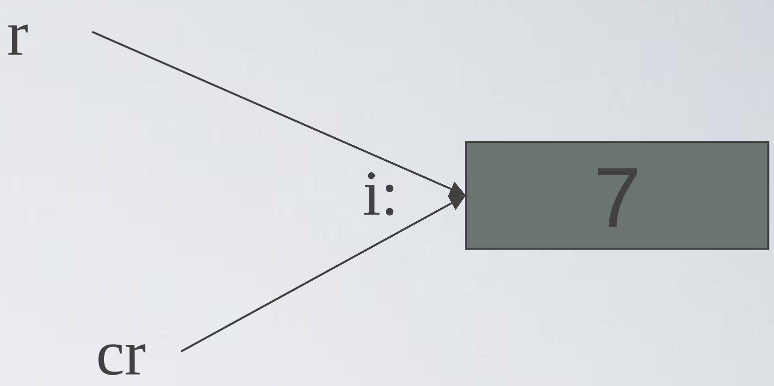
```
void f(int a, int& r, const int& cr) { ++a; ++r; ++cr; } // error: cr is const
void g(int a, int& r, const int& cr) { ++a; ++r; int x = cr; ++x; } // ok

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

```
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 <u>reference as an alternative name for an object</u>
- You can't
  - modify an object through a **const** reference
  - make a reference refer to another object after initialization

Guidance for Passing Variables

- Use call-by-value for very small objects
- Use call-by-const-reference for large objects
- Return a result rather than modify an object through a reference argument
- Use call-by-reference only when you have to

- 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***

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 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 Lecture 11

Function call implementation I

- Remember functions from Lectures 5, 6:

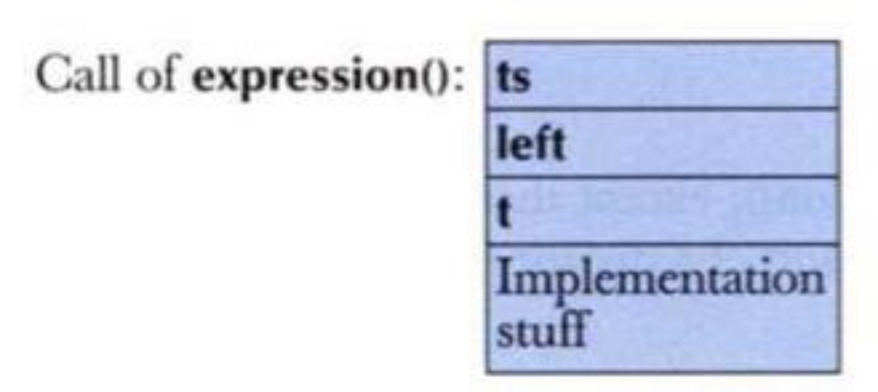
```
double term(Token_stream& ts)
{
        double left = primary(ts);
        Token t = ts.get();
        // . . .
                case '/':
                {
                        double d = primary(ts);
                        // . . .
                }
        // . . .
}
```

```
double primary(Token_stream& ts)
{
        Token t = ts.get ();
        switch (t.kind) {
        case '(':
                {       double d = expression(ts);
                        // . . .
                }
                // . . .
        }
}
```

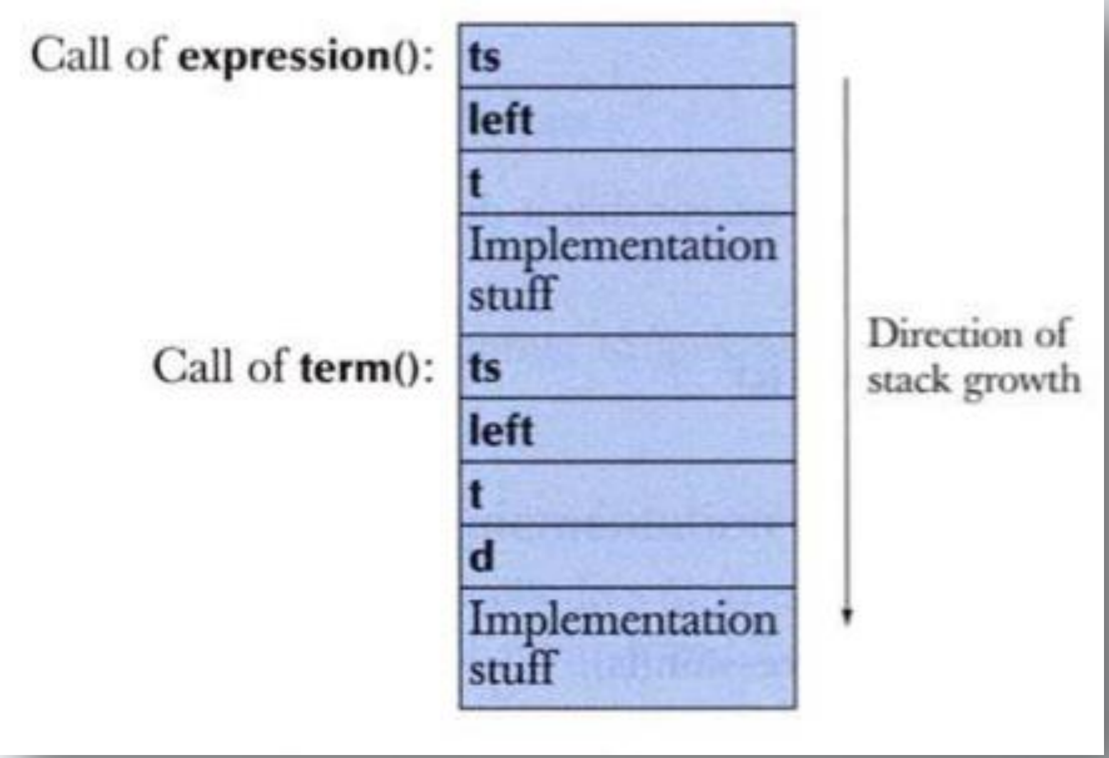
```
double expression(Token_stream& ts)
{
        double left = term(ts);
        Token t = ts.get();
        // . . .
}
```

Function call implementation II

- When a function is called, the language implementation sets aside a data structure containing a copy of all its parameters and local variables.
- For example, when expression() is first called, the compiler ensures that a structure like this is created: *a function activation record*

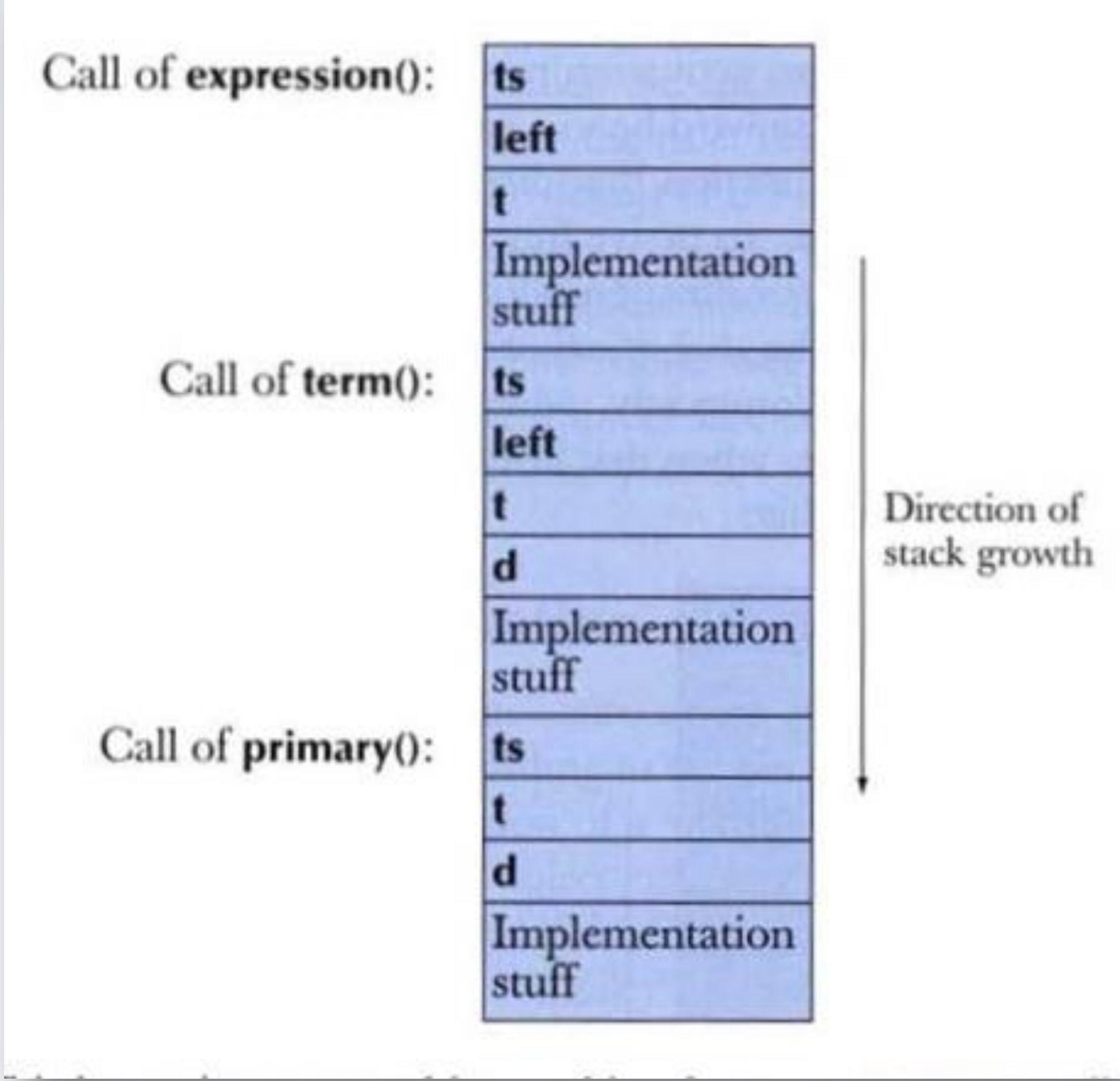


- So far, so good, and now expression() calls term(), so the compiler ensures that an activation record for this call of term() is generated :



Function call implementation III

- Now term() calls primary() and we get:



Order of evaluation

- The evaluation of a program - also called the *execution of a program* - proceeds through the statements according to the language rules.
- When this "thread of execution" reaches the definition of a variable,
- the variable is constructed;
- that is, memory is set aside for the object and the object is initialized.
- when the variable goes out of scope,
  - the variable is destroyed;
  - that is, the object it refers to is in principle removed and the compiler can use its memory for something else

Order of evaluation II

```
string program_name = "silly";
vector<string> v;                          // v is global

void f()
{
        string s;                          // s is local to f
        while (cin>>s && s!="quit") {
                string stripped;           // stripped is local to the loop
                string not_letters;
                for (int i=0; i<s.size(); ++i)    // i has statement scope
                        if (isalpha(s[i]))
                                stripped += s[i];
                        else
                                not_letters += s[i];
                v.push_back(stripped);
                // . . .
        }
        // . . .
}
```

Expression evaluation

- The order of evaluation of sub-expressions is governed by rules designed to please an optimizer rather than to make life simple for the programmer.
- That's unfortunate, but you should avoid complicated expressions anyway, and there is a simple rule that can keep you out of trouble:
    - if you change the value of a variable in an expression, don't read or write it twice in that same expression. For example:

```
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
```

Global initialization

- Using a global variable in anything but the most limited circumstances is usually not a good idea
- Such code is to be avoided for several reasons:
  - it uses global variables,
  - it gives the global variables short names,
  - it uses complicated initialization of the global variables.

```
// file f2.cpp
extern int y1;
int y2 = y1+2;        // y2 becomes 2 or 5
```

Recursively Defined functions

- For some problems, it's useful to have functions *call themselves*
- As often it is difficult to express the members of an object or numerical sequence explicitly.

e.g.: The **Fibonacci** sequence:

$\{f_n\} = 0,1,1,2,3,5,8,13,21,34,55,\ldots$

- There may, however, be some "local" connections that can give rise to a ***recursive definition*** –a formula that expresses higher terms in the sequence, in terms of lower terms.

e.g.: Recursive definition for $\{f_n\}$:

INITIALIZATION: $f_0 = 0, f_1 = 1$

RECURSION: $f_n = f_{n-1} + f_{n-2}$ for $n > 1$.

Recursive Definitions and Induction

- Recursive definition and inductive proofs are complement each other: a recursive definition usually gives rise to natural proofs involving the recursively defined sequence.
- This is follows from the format of a recursive definition as consisting of two parts:
- **Initialization** –analogous to induction **base cases**
- **Recursion** –analogous to **induction step**
- In both induction and recursion, the domino analogy is useful.

Recursion

- We must always make sure that the recursion *bottoms out*:
  - A recursive function must contain at least one non-recursive branch.
  - The recursive calls must eventually lead to a non-recursive branch.
- Recursion is one way to decompose a task into smaller subtasks. At least one of the subtasks is a smaller example of the same task.
- The smallest example of the same task has a non-recursive solution.
- Fibonacci numbers:

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

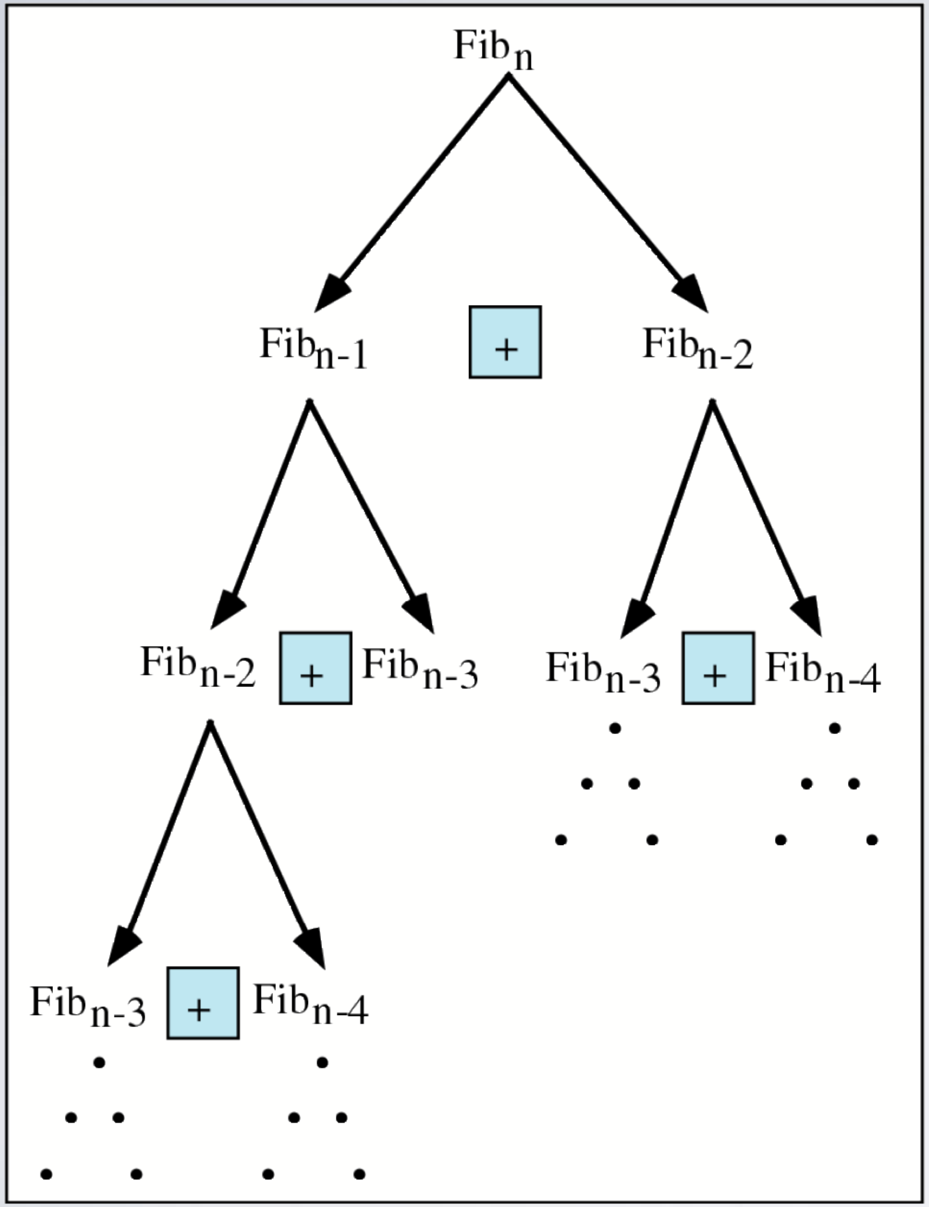
  where each number is the sum of the preceding two.
- Recursive definition:
  - **F(0) = 0;**
  - **F(1) = 1;**
  - **F(number) = F(number-1)+ F(number-2);**

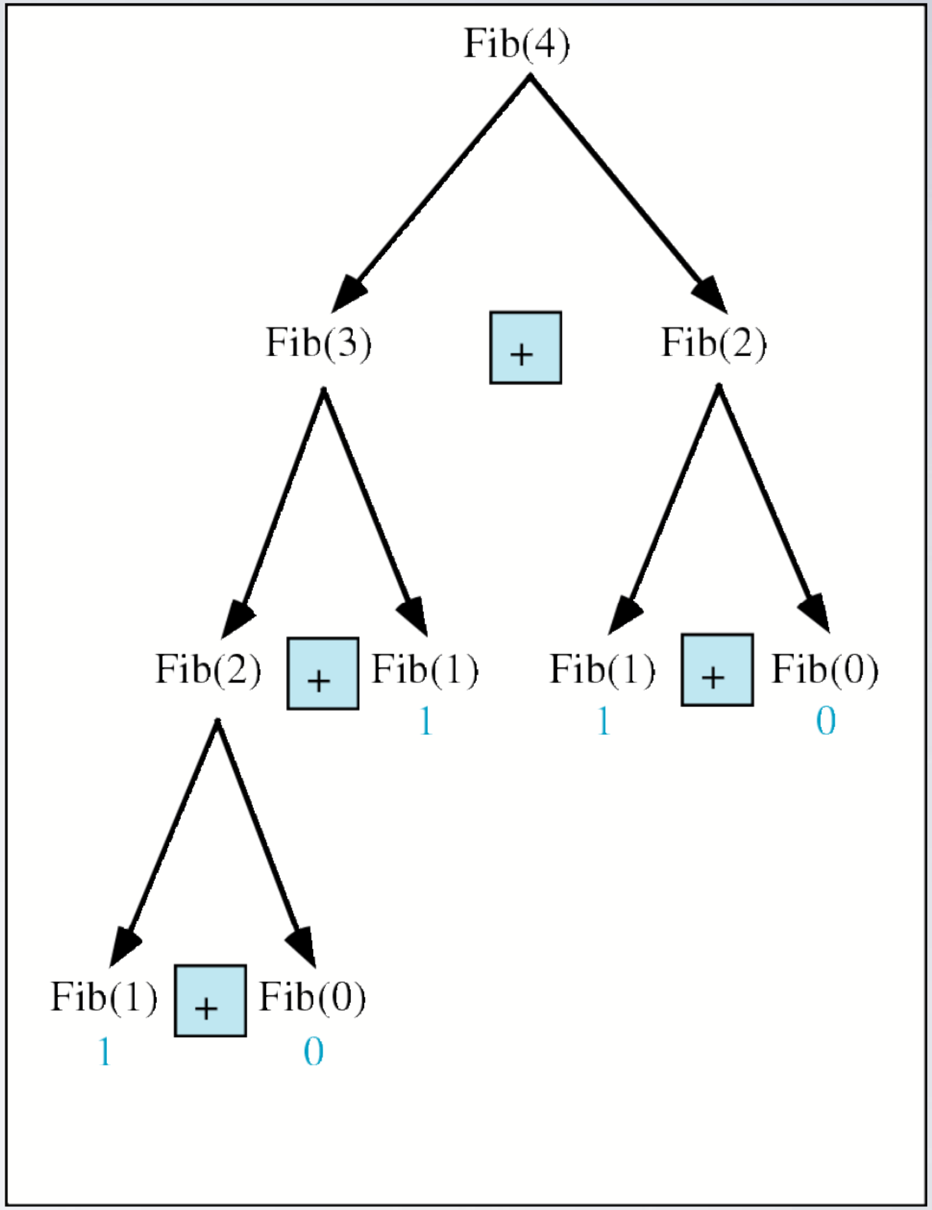
Recursive Example: Fibonacci numbers

```
//Calculate Fibonacci numbers using recursive function.
//A very inefficient way, but illustrates recursion well
int fib(int number)
{
  if (number == 0) return 0;
  if (number == 1) return 1;
  return (fib(number-1) + fib(number-2));
}

int main(){     // driver function
  int inp_number=0;
  cout << "Please enter an integer: ";
  cin >> inp_number;
  cout << "The Fibonacci number for "<< inp_number
         << " is "<< fib(inp_number)<<endl;
  return 0;
}
```

```
f(0) is 0
f(1) is 1
f(2) is 1
f(3) is 2
f(4) is 3
f(5) is 5
f(6) is 8
```


Fibn
Fibn-1
+
Fibn-2
Fibn-2
+
Fibn-3
Fibn-3
+
Fibn-4
Fibn-3
+
Fibn-4
(a) Fib(n)

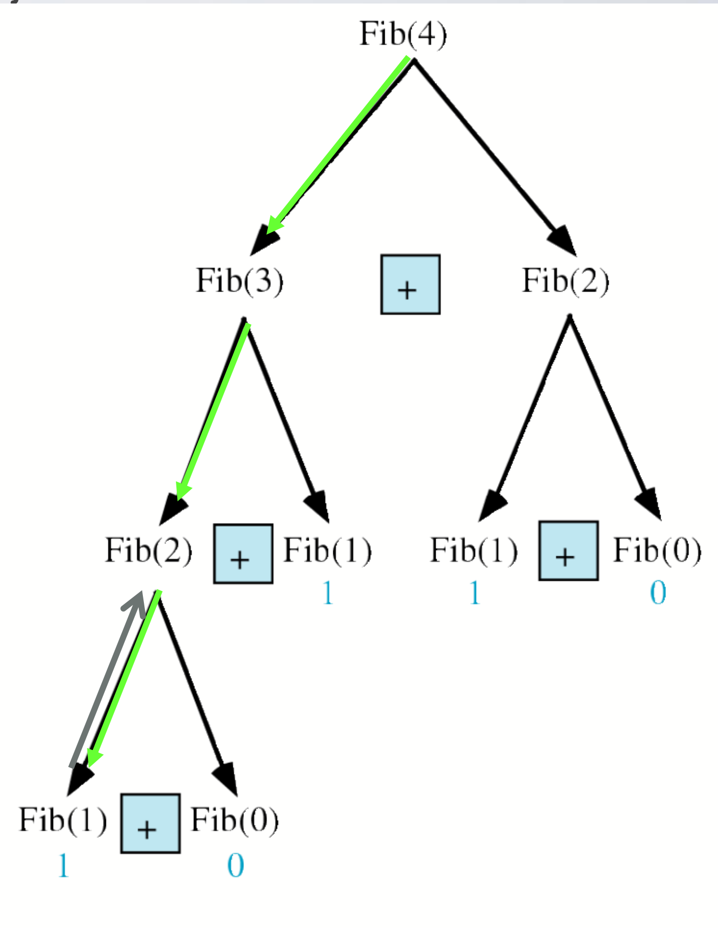

Fib(4)
Fib(3)
+
Fib(2)
Fib(2)
+
Fib(1)
1
Fib(1)
1
+
Fib(0)
0
Fib(1)
1
+
Fib(0)
0
(b) Fib(4)

Trace a Fibonacci Number

- Assume the input number is 4, that is, num=4:

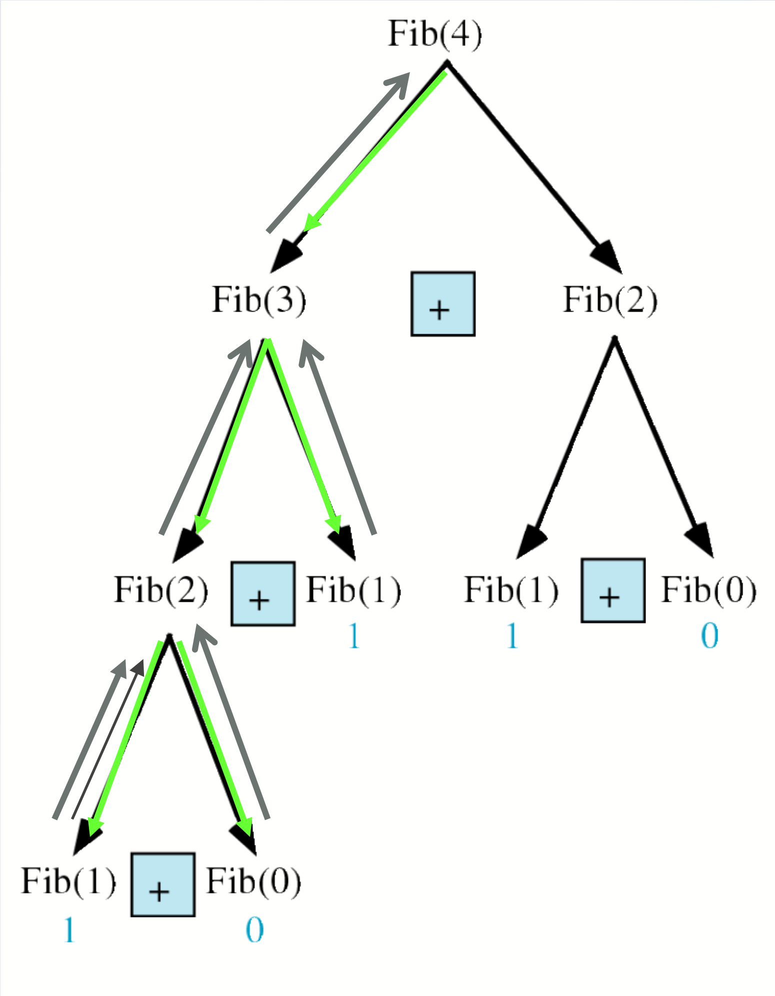
```
fib(4):
  4 == 0 ? No;    4 == 1?    No.
  fib(4) = fib(3) + fib(2)
  fib(3):
    3 == 0 ? No; 3 == 1? No.
    fib(3) = fib(2) + fib(1)
    fib(2):
      2 == 0? No; 2==1? No.
      fib(2) = fib(1)+fib(0)
       fib(1):
         1== 0 ? No; 1 == 1? Yes.
             fib(1) = 1;
      return fib(1);
```

```
int fib(int num)
{
   if (num == 0) return 0;
   if (num == 1) return 1;
   return
      (fib(num-1)+fib(num-
   2));
}
```



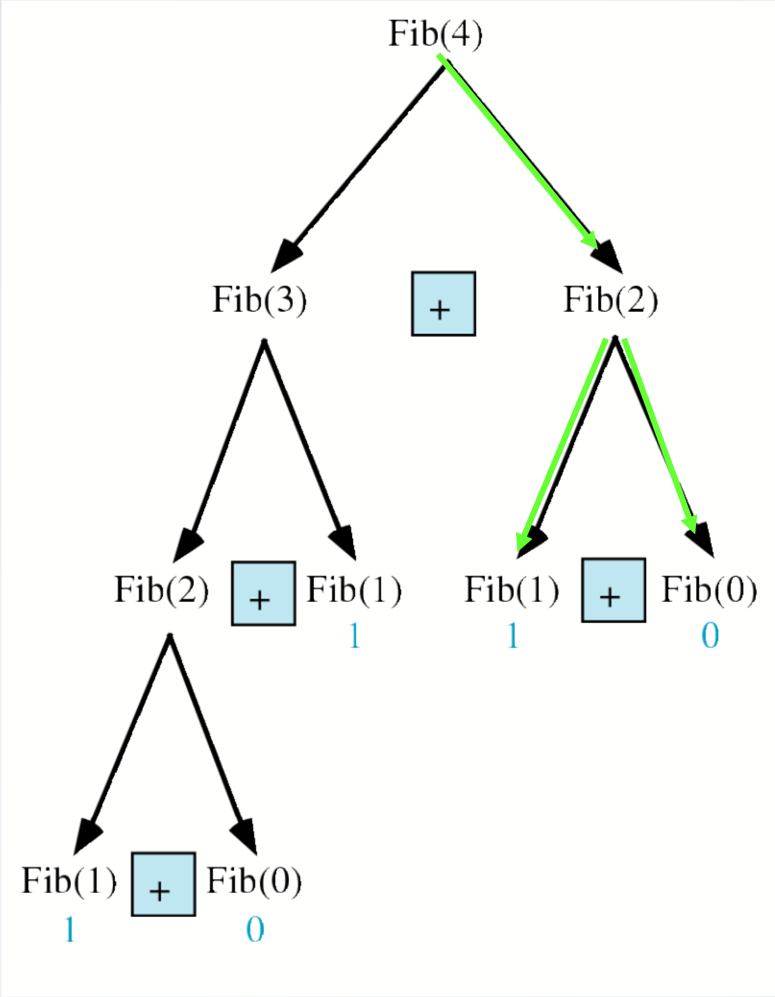
Trace a Fibonacci Number

```
fib(0):
        0 == 0 ?  Yes.
        fib(0) = 0;
        return fib(0);
   fib(2) = 1 + 0 = 1;
   return fib(2);
fib(3) = 1 + fib(1)
   fib(1):
     1 == 0 ? No; 1 == 1? Yes
     fib(1) = 1;
     return fib(1);
fib(3) = 1 + 1 = 2;
return fib(3)
```



Trace a Fibonacci Number

```
fib(2):
2 == 0 ? No; 2 == 1?  No.
fib(2) = fib(1) + fib(0)
fib(1):
   1 == 0 ? No; 1 == 1?  Yes.
       fib(1) = 1;
return fib(1);
 fib(0):
     0 == 0 ?   Yes.
       fib(0) = 0;
       return fib(0);
 fib(2) = 1 + 0 = 1;
 return fib(2);
fib(4) = fib(3) + fib(2)
       = 2 + 1 = 3;
return fib(4);
```



Fibonacci number w/o recursion

```
//Calculate Fibonacci numbers iteratively
//much more efficient than recursive solution

int fib(int n)
{
 int f[n+1];
 f[0] = 0; f[1] = 1;
  for (int i=2; i<= n; i++)
        f[i] = f[i-1] + f[i-2];
 return f[n];
}
```

Next talk

- More technicalities, mostly related to classes

Acknowledgements

Bjarne Stroustrup

Programming -- Principles and Practice Using C++

http://www.stroustrup.com/Programming/

Thank you!

Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο
ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ & ΘΡΗΣΚΕΥΜΑΤΩΝ, ΠΟΛΙΤΙΣΜΟΥ & ΑΘΛΗΤΙΣΜΟΥ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ
ΕΣΠΑ 2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ
Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης