C++: Functions/Program Structure

Functions/Program Structure

- Program Structure
- Function Syntax
- A Special Parameter Type
- Recursion
- Header Files

- Additional slides (DIY, not covered in this lecture):
  - Using libraries and build utilities
  - Profiling

Program Structure

- Functions: used to break up complex problems into manageable elements
- Functions should be like math. theorems: they abbreviate knowledge, their proof is at another level.

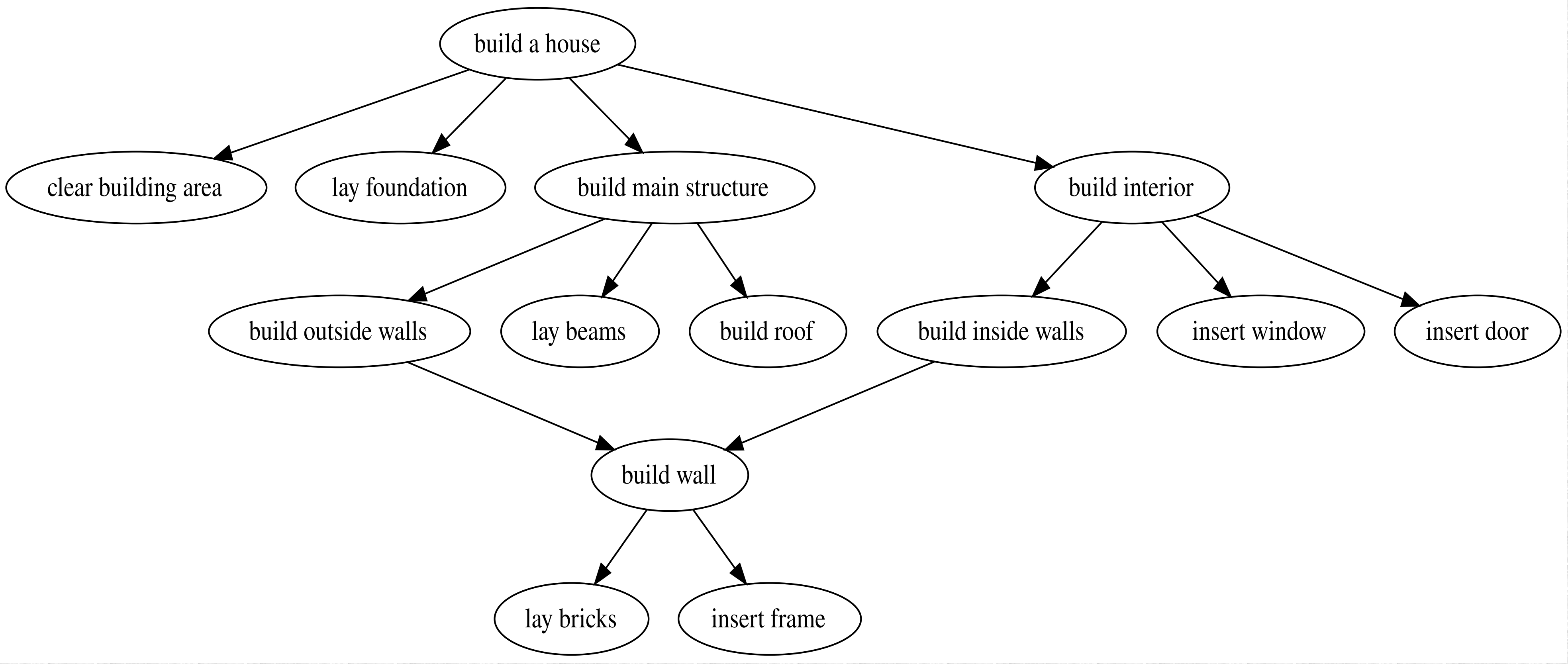
- *Test* your functions before using them!

Program Structure

- Levels.
  - e.g., from the *verbal* level to the *main* level
- Each new level is a 1:1 specification of the previous level, creating a *coordinate system*
- Each level should be intuitively clear given the previous level
- Some functions are merely *one* statement

Program Structure

- Levels: *verbal, main, decomposition, auxiliary*
- Decompose until ‘manageable’
  - Cf., task planning



Intermezzo: types & constness

- Built-in vs. structured types
  - `double gravity = 9.81;`
  - `std::string bird{ "penguin" };`
- Const vs. non-const
  - `size_t const wingspan_B737 = 34;`

  - *const* soon becomes important;

Intermezzo: terminology

prvalue

lvalue

```
string hi{“hi”}, yall;
yall = “y’all”;
hi + yall = string{“go”} + “away!”;
yall = hi;
```

glvalue

xvalue

- lvalue: available *L*ocation (*L*eft-hand assignment value)
- xvalue: e*X*piring value
- glvalue: *G*eneralized *L*value (lvalue or xvalue)
- rvalue: *R*ight-hand (non-address) value
- prvalue: *P*ure *R*value (literal or const value)

Terminology: pointers and references

- *Pointers* contain addresses, and can be 0:

```
size_t length = 13;
size_t *pLen = &length;
*pLen = 12;
```

- References bind to (g)lvalues, and **must** be initialized at definition time:

```
size_t &rLen = length;
```

- Rvalue references: bind to rvalues (temporary or no address at all):

```
string &&greet =
  string{"Hi "} + string{“all!\n”};
```

Function Example

```
struct Coords{
    int x, y;
};

int distance(Coords lhs, Coords rhs)
{
    lhs.x -= rhs.x; lhs.y -= rhs.y; // Obtain difference
    return sqrt(sqr(lhs.x) + sqr(lhs.y)); // Pythagorize
}

int main(int argc, char *argv[])
{
    x = stoi(argv[1]); y = stoi(argv[2]);
    Coords origin{0, 0};
    cout << distance(
                origin,
                Coords{x, y}
               );
}
```

function

parameter

argument

This Coords only exists briefly.

function call

Function Anatomy

- Generic Function Syntax:

```
returnType    name  (  parameterlist    )
{
    statement(s)
    and/or
    definition(s)
    and sometimes
    declaration(s)
}
```

Red: required

Example:

```
unsigned long long square(long int value)
{
    return value * value;
}
```

Function Anatomy

- Function elements:

```
returnType   name  (  parameterlist   )
{
    statement(s)
    and/or
    definition(s)
    and sometimes
    declaration(s)
}
```

Why does this function exist?

Function Anatomy

- Function elements:

What does the **function** tell its caller?

```
returnType   name  (  parameterlist  )
{
    statement(s)
    and/or
    definition(s)
    and sometimes
    declaration(s)
}
```

value:
builtin types, or objects created by the function

reference:
external *objects* used by the function.

pointer:
e.g., *char* *

Function Anatomy

What does the **caller** tell the function?

- Function elements:

```
returnType   name  (  parameterlist  )
{
    statement(s)
    and/or
    definition(s)
    and sometimes
    declaration(s)
}
```

Function Anatomy

- Function elements:

```
returnType    name   (  parameterlist   )
{
    statement(s)
    definition(s)
    declaration(s)
}
```

Define *actions* to be performed by the function,
Define *variables where first used*. (LOCALIZE!)

Functions should fit (approx.) on your screen

- *One* function, *one* responsibility
- Define *auxiliary* (support) functions
- Functions called by users should validate their input parameters
- Add *semantic* comment to statements, and *generic* comment below the function's definition

Function Anatomy

- Function parameters:
  - defined in a comma separated list.
  - parameters are *local variables, initialized by the caller*.
  - the initializing (outer) values are called *arguments*.
  - some examples of *parameters* and *arguments*:

```
string *out, string const &in, char const *prompt, int count
 &greeting ,      welcome     ,        "$>"       ,     3
```

Modifying arguments

- Streams change:

```
string read_line(istream input); // Can’t do this
```

- To modify arguments use parameters that reach the arguments:

```
istream &getline(istream &in, string &dest);
```

Reference Efficiency

- *Pass By Value*: often inefficient

```
size_t line_count(string book);          // copy the book ???
```

- Instead: pass by (const) reference:

```
size_t countLines(string const &book);
```

```
string &checkSpelling(string &book);
```

```
size_t nLines =
   countLines( checkSpelling(annotations)  );
```

Rvalue Efficiency

- This makes some sense*:

```
string sorted_merge(string sorted_left,
                    string const &sorted_right);
```

but if this happens:

```
cout <<
    sorted_merge(sorted_merge(dictNL, dictBE),
                 dict_SA);
```

Conflict

- then have this *too*:

```
string sorted_merge(string &&tmp_sorted_left,
                    string const &sorted_right);
```

- *More symmetrical and more often seen:

```
string sorted_merge(string const &sorted_left,
                    string const &sorted_right);
```

Array Parameters

- When defining *array* parameters:
  - ***always*** leave out their *first* dimensions.

```
External Variable:         Piece chessBoard[8]
[8]

Corresponding parameter:   fun(Piece board[][8])

Called as argument:        fun(chessBoard)
```

Functions

- Parameters (and return types):
  used for ***communication***
  - Who ***owns*** the information?
    - Use *const* for pointers and references if the function is not *conceptually* the owner
  - Use *pointer parameters* to assign values to variables *outside of* the function
  - Use (lvalue) *references* when passing *objects* instead of primitive data types to the function
  - Use *rvalue references* to refer to *temporary modifiable objects,* passed to the function

Functions

- More *Good Practices*:
  - *return by argument (RBA)*:
    - use *pointers* when modifying external variables
    - put these pointers at the *beginning* of the parameter list
  - Use *const* for *pointers* and *references* and maybe values) that refer to entities which are ***not*** modified by the function.
    - In *headers* it is pedantic to use const for *value parameters*
  - Consider using *struct*-types (POD) when multiple, related values are returned by functions.

Functions

- Structured binding declarations**:**
  - multiple values returned by functions.
  - Assume *fun()* returns a struct containing an *int*, a *std::string*, and a *double*:

```
    // inside a calling function:
auto [value, txt, amount] = fun();  // int value, string txt,
                                    // double amount

    // alternative:
auto &&[value, txt, amount] = fun();    // now the variables
                                    // refer to the (temporary)
                                    // struct fields
```

Functions

- Structured binding declarations
  - potentially useful in *init* sections of *for* stmnts: multiple types.

```
for (auto [count, collect, sum] = fun(); count--; )
{                          // int count, string collect, double sum
   // use collect and sum in this compound statement
}
```

  - *fun()* must return a *struct-type*[(1)]

[(1)]in part II we'll learn how to automate that.

Functions

- ***Value*** *parameters*:
  - Use value parameters for *builtin types* and class types initialized by the caller, but used as *local variables* by the function
  - Value parameters don't need ***const*** (but it doesn’t hurt if the parameter isn't altered)

```
void function(int value, string text)
{
    text += " whatever";
    value <<= 5;
    ...
}
```

Functions

- **Value** *return types*:
  - Use value return types when returning values of *builtin types* or class-type objects *created* by the function (*factory functions*)
  - Value return types don't need ***const***

```
string factory()
{
    cout << "Enter a character and a value: ";
    char ch;
    size_t value;
    cin >> ch >> value;
    return string(value, ch);     // why not {} ?
}
```

Functions

- ***Reference*** parameters/return values:
  - Are used to return *existing* objects, used by the function and returned to the caller
  - Use const with references and pointers:
    - for ***parameters***: if *not modified* by the *function*.
    - for ***return*** types: the *caller* may *not modify* the values they refer to.

```
void function(istream &cin, string const &callerText);

string const &function(string const &inText)
{
    // use (but not modify) inText, then return inText:
    return inText;
}
```

Functions

- ***Rvalue reference*** parameters:
  - Are used for *temporary* objects that cease to exists once the function returns
  - Within the function they're *not* anonymous. To *anonymize* them, use *std::move.*
  - Plain *return values* appear as anonymous values; so do *rvalue parameters* returned via *std::move*

```
string function(string &&anon)
{
    // use anon as an ordinary string object
    ...
    return std::move(anon); // return as rvalue ref.
}
```

Functions

- ***Pointer*** parameters/return values:
  - as parameters: are used to assign values to objects or variables *living outside of functions*
  - as return types: are used to *return* *newly allocated data*
  - use them when it's the *natural form* (e.g. NTBSs)
  - use const to prevent data modification

```
bool option(string *value, int opt);

char const *programName(string const &prog)
{
    return prog.c_str();
}
```

Function Overloading

- What is it?
  - Same function name, different parameters;
  - *Arguments* determine which function is used;
  - *Name mangling* is used to distinguish the functions in object modules
  - **C** *does not use name mangling*
    - **C** function declarations use their names as-is.

Function Overloading

- Why is it available?
  - Same function, different parameters. Examples:

```
string::find(char ch)                   // 1
string::find(string const &argument)   // 2
```

    - (1) can use efficient implementations given that its argument is a single char;
    - (2) uses code processing multiple characters.
- Note: *return types* are ***not*** considered when deciding which overloaded function to call.

Function Overloading

- How is it used?
  - *Arguments* determine which function is called.
  - Example:

```
string::find(char ch)                    // 1
string::find(string const &argument)    // 2
```

```
int main(int argc, char *argv[])
{
   string program{ argv[0] };

   std::cout << program.find('.');    // calls (1)
   std::cout << program.find(".out"); // calls (2)
}
```

Function Overloading

- How does it work?
  - *Name mangling* is used to distinguish overloaded versions
    - *Name mangling* allows the compiler to inform the *linker* about which function to call in the final program:

```
double sqr(int arg)    links to, e.g.: _Z4sqri
double sqr(double arg) links to, e.g.: _Z4sqrd
```

Function Overloading

- Calling **C** functions from **C++**
  - **C** does ***not*** use name mangling:

```
double sqr(int arg)        becomes   _Z4sqri (C++)
double sqr(double arg)     remains:  sqr     (C)
```

  - Consequently, the compiler must know the *language context*:
    - ***C** headers must be prepared for being used by **C++** programs*
      - using *#ifdef* preprocessor directives,or:
      - use/define special **C++** headers:

```
Prefer:   #include <cstdlib>
over:     #include <stdlib.h>
```

Function Arguments

- Default arguments:
  - The compiler may provide *default* arguments to functions
  - Defaults may be used from the *last* to the *first* parameter
  - Default arguments are provided by *declarations* (*headers*), *not* by *definitions* (*sources*).
  - Ambiguity must be prevented (by us!)

Function Arguments

- Default arguments example:

```
string::find(char ch, size_type pos = 0);

istream &getline(istream &in, string &str,
                 char delim = '\n');
```

A Special Parameter Type

- Consider multiple, but an unspecified number of arguments
  - *Assignment*: define a function computing the sum of double values
  - Examples of how it's used (sort of...):

```
cout << sum(1, 2.5, 3.14);
cout << sum(12.45);
cout << sum(1, 2.3, 4, 5.6, -7, -8.9, 10, 11.12);
```

  - Why can't we use function overloading?

A Special Parameter Type

- Solution to the variable number of arguments problem:
  - *Initializer lists* accept any number of values.
  - To use initializer lists, do:

    #include <initializer_list>
  - Here's a function declaration:

```
double sum(std::initializer_list<double> list);
```

  - Here's how the function can be called:

```
sum({ 1, -2.3, 4, 5.6 });
sum({ -7, 8.9 });
```

adopt the habit to add a blank after { and before } to make the list stand out

A Special Parameter Type

- The implementation of the function *sum*:
  - the initializer list's member *size* returns the number of its elements
  - Use a range-based for-loop to visit all elements.
- Implementation:

```
double sum(std::initializer_list<double> list)
{
    cout << "There are " << list.size() << " elements\n";

    double ret = 0;

    for (double value: list)
        ret += value;

    return ret;
}
```

Recursion

- Recursive function: a function calling itself
  - Either directly
  - Or indirectly
    - by calling another function that calls our function.
- Why use recursion?

Recursion

- Direct recursion:
  - a function calling itself.

- Example:

```
void showValues(size_t idx)
{
    cout << idx << '\n';
    if (idx == 0)
        return;
    showValues(idx - 1);
}
```

Recursion

- Indirect recursion:
  - a function calls another function which in turn calls the first function.

- Example:

```
void extra(size_t idx)
{
   showValues(idx - 1);
}
void showValues(size_t idx)
{
   cout << idx << '\n';
   if (idx == 0)
      return;
   extra(idx);
}
```

Recursion

- Why use recursion?
  - The underlying problem is recursively defined
  - A *recursive data structure* is processed
- But *avoid* recursion:
  - When using *tail recursion* (cf. *showValue*'s initial implementation)
  - When an *easy to understand* *iterative* algorithm is available (also: *showValue)*

Recursion

- A nice example using recursion
- We're going to play *compiler*...
  - Assignment:
    - print a *size_t* having an unknown value.
  - Problem:
    - We don't now what its *most significant digit* is.

Recursion

- Assignment:
  - print a *size_t* having an unknown value.
- Problem:
  - We don't now what its *most significant digit* is.
- Solution:
  - We know what its *least significant digit* is.

Recursion

- Assignment: print a *size_t* having an unknown value.
  - We don't now what its *most significant digit* is.
  - But its *least significant digit* is known.
  - Approach:
    - Determine the *least significant digit* (LSD);
    - Process a smaller value by removing the LSD;
      - *Recursion:* If the smaller value exceeds zero, apply the algorithm to that smaller value;
    - print the LSD.

Recursion: Examples

- Print a *size_t* having an unknown value.
  - Implementation:

```
void printDecimal(size_t nr)
{
    size_t lsd = nr % 10;      compute the LSD
    size_t smaller = nr / 10;  compute the smaller value

    if (smaller > 0)           if the smaller value exceeds
        printDecimal(smaller); zero, apply the algorithm to it

    cout << static_cast<char>   print the LSD
                ('0' + lsd);
}
```

Recursion

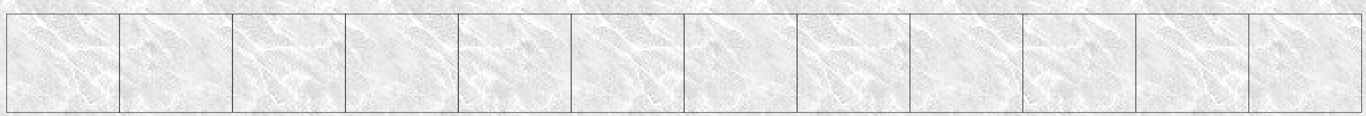
- Another historic example:
  sort an array (*quicksort, Hoare, 1962)*

Starting point: an array of *n* elements. Define *left* (its leftmost index) and *right* (the index just beyond the last element)

quicksort(array, left, right)

left

right

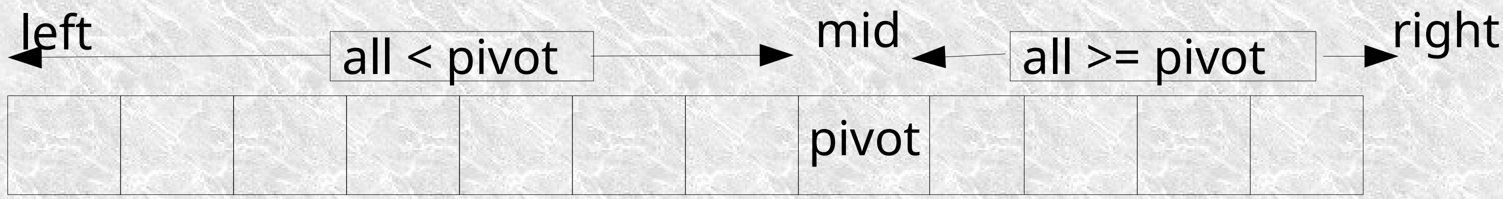


- We have not specified the array's actual size
- We're done once *left* >= *right*

Recursion: Examples

- Step one: *partitioning*.

Pick an element, e.g., the element *pivot* = *array[left]*, and put all elements smaller than *pivot* to the left of *pivot*, and all other elements to the right of *pivot*, placing *pivot* at *array[mid]*

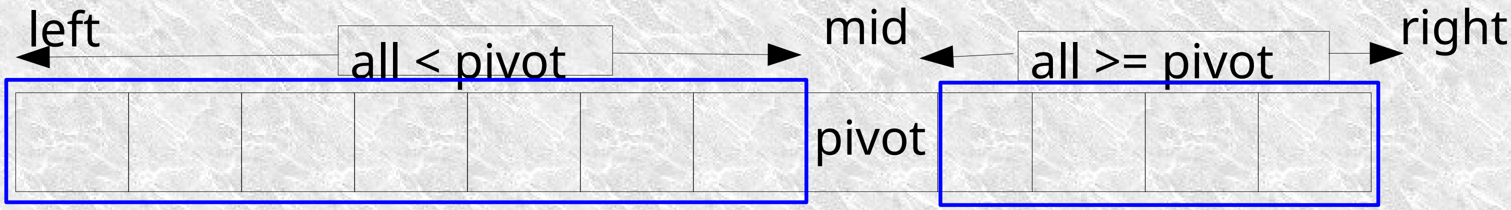


pivot = partition(array, left, right)

Recursion: Examples

- Step one: *partitioning*.

Pick an element, e.g., the element *pivot* = *array[left]*, and put all elements smaller than *pivot* to the left of *pivot*, and all other elements to the right of *pivot*, placing *pivot* at *array[mid]*

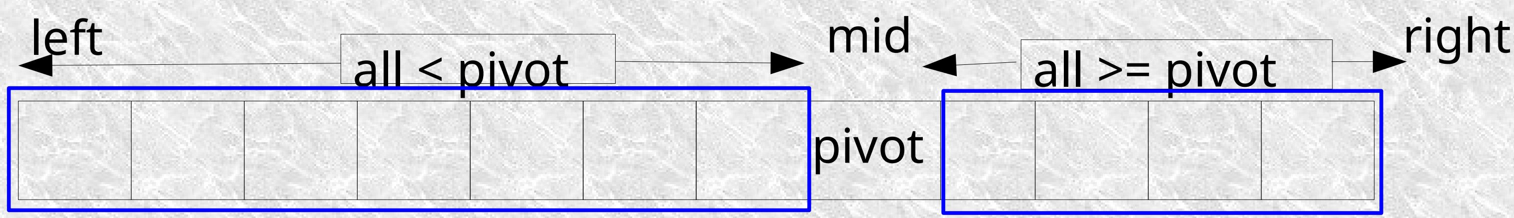


- Note: the array *left* to *mid* is an array like the original array
  the array *mid+1* to *right* is an array like the original array

Recursion: Examples

- Step two: *recursion*.

Pick an element, e.g., the element *pivot* = *array[left]*, and put all elements smaller than *pivot* to the left of *pivot*, and all other elements to the right of *pivot*, placing *pivot* at *array[mid]*



Use the same algorithm to sort, resp. the array *left* to *mid,*

quicksort(array, left, mid)

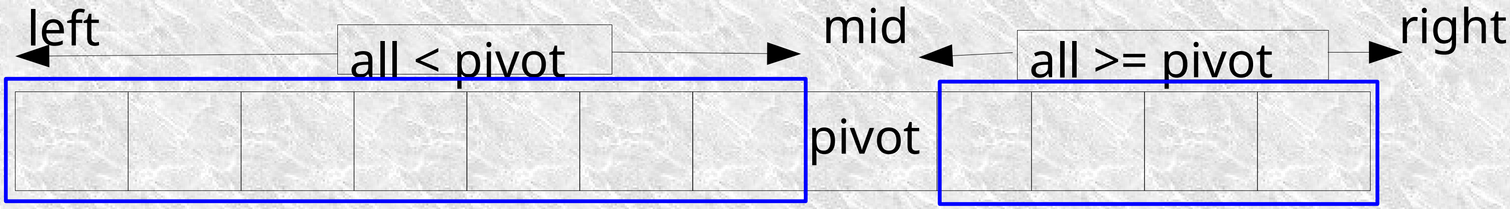
and to sort the array *mid+1* to *right,*

quicksort(array, mid+1, right)

Recursion: Examples

- Quicksort: *implementation*

Pick an element, e.g., the element *pivot* = *array[left]*, and put all elements smaller than *pivot* to the left of *pivot*, and all other elements to the right of *pivot*, placing *pivot* at *array[mid]*



The implementation:

```
void quicksort(Type &array, size_t left, size_t right)
{
    if (left >= right)
        return;
    size_t mid = partition(array, left, right);
    quicksort(array, left, mid);
    quicksort(array, mid + 1, right);
}
```

Header Files

- Source files: *function and variable definitions*
- Header files: *declarations* and *type definitions*
  - Used by the compiler
    - to learn which symbols (also: types) are available.
    - Header files contain declarations of *related* entities. E.g., all functions used in a program.

***Never* define header files for just one entity (e.g., for just a single function).**

Header Files

```
mathdecl.h defines constants
like M_PI, and declares
functions, like sin().
```

```
sphere.h
uses sin(), so:
#include <mathdecl.h>
```

```
navigate.h
uses sin(), so:
#include <mathdecl.h>
```

```
#include <sphere.h>
#include <navigate.h>

void fun()
...
```

- There's a ***slight*** problem:
  fun() won't compile...

Header Files

- To prevent double inclusions:
  - define *include guards*:

```
#ifndef INCLUDED_MYHEADER_H_
#define INCLUDED_MYHEADER_H_

// remaining contents of the header file

#endif
```

  - Include guards: are ***only required*** for header files which are *potentially included by other header files*

Header Files

- ***C*** *headers must be prepared for use by* ***C++***, using *#ifdef* preprocessor directives:

```
// C header useable in C++ sources

#ifdef __cplusplus
   extern "C" {
#endif

double sqrt(double param);   // A C-function

#ifdef __cplusplus
   }
#endif
```

Header Files

- Special **C++** headers can be constructed.

```
// A specialized C++ header (e.g., `cstring')

extern "C" {
#include "string.h"
}
```

Libraries

- Libraries are used to store object modules
- *Object modules* (not just functions!) are linked to programs
- Therefore, use *one function per file*:
  - improves source-contents relationship
  - simplifies maintenance
  - reduces (re)compilation times
  - reduces program sizes
  - improves possibilities for profiling
  - allows you to make better use of a profiler

Libraries

- Library construction:
  - 1. compile sources

Libraries

- Library construction:
  - 1. compile sources
  - 2. add objects to library

Libraries

- Library construction:
  - 1. compile sources
  - 2. add objects to library
  - 3. .o files may be removed (copies are in the libraries)

Libraries

- Library construction:
  - 1. compile sources
  - 2. add objects to library
  - 3. .o files may be removed (copies are in the libraries)
  - 4. store the library at a well-known or general location.

Libraries

- Library *use*:
  - g++ -o program program.o -Lpath/to/lib -lname
    - path/to/lib: path to library
    - name: name of the library, without *lib* prefix and *.a* extension

```
// assume library libmylib.a is in LIBRARY_PATH or, e.g., /lib
// assume program.cc uses objects from libmylib.a

g++ -o program program.cc -lmylib -s
```

Profiling

- What is efficient?
  - In general: hard to predict
  - A *profiler* should be used to find the *bottlenecks.*

- To be able to profile:
  - Profiling looks at the function-level
  - Use *-pg* when compiling and linking, no *-s*
  - Run the program
  - Profile: gprof -bp a.out gmon.out

Profiling

- Example:

```
size_t fun(std::string value);
size_t fun2(std::string const &value);
size_t callFun(string const &prog);    // calls fun  10,000,000 times
size_t callFun2(string const &prog)    // calls fun2 10,000,000 times
```

```
g++ -O2 -pg main.cc
a.out
```

```
gprof -bp a.out gmon.out
  %   cumulative   self     self
 time seconds   seconds   ms/call  name
100.00   0.03     0.03    30.00    callFun
  0.00   0.03     0.00     0.00    callFun2
```

Profiling

- Afterthoughts:
  - Do not trust your judgment: *profile!*
  - Don't overdo things. Profiling is seldom needed in this course
  - Profiling beats debugging. Don't fall for debuggers. Avoid them.
  - Basic rules of optimization:
    - First rule: *don't do it.*
    - Second rule: *don't do it yet.*

Functions/Program Structure

- Program Structure
- Function Syntax
- A Special Parameter Type
- Recursion
- Header Files

- Additional slides (DIY, not covered in this lecture):
  - Using libraries and build utilities
  - Profiling

Functions/Program Structure

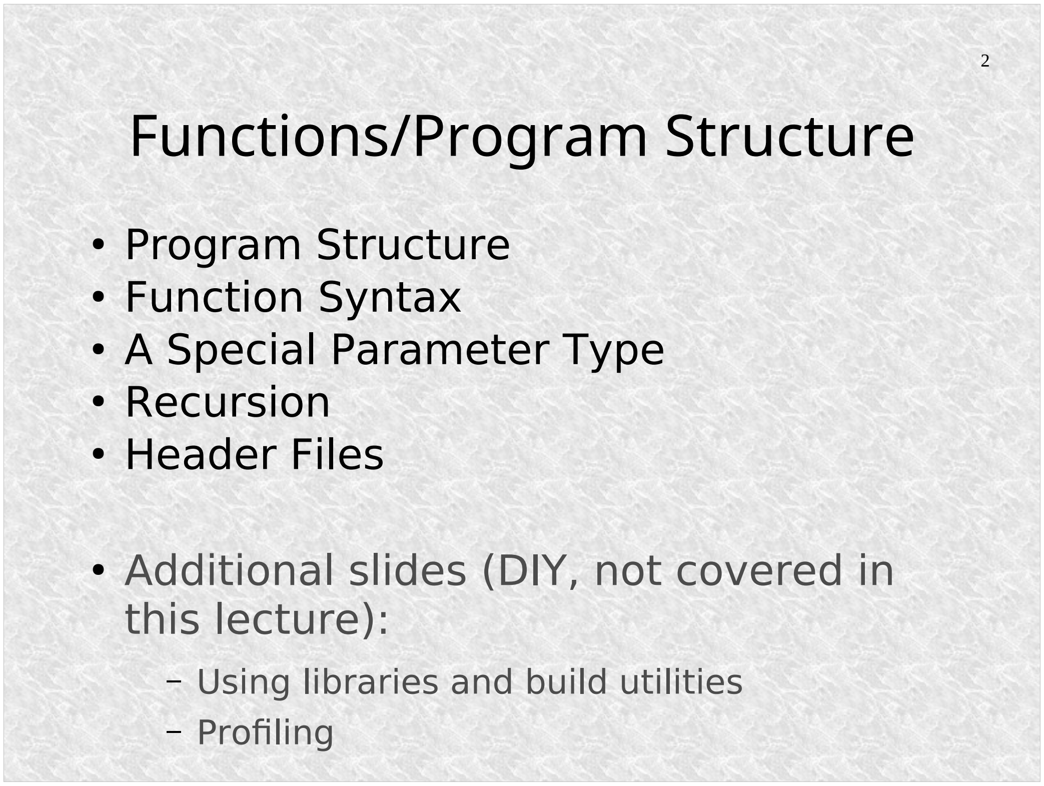
That's All, Folks,
about Functions.

C++: Functions/Program Structure

This picture was taken on one of the dirt roads in Death Valley. It's a road junction called Teakettle Junction. It shows the large variety of teakettles.

Functions are like this: they come in many, many variants, and each should have its own, well-defined purpose.

Make sure that you get their semantics right before implementing them, and make sure their implementation follows their semantic analysis: again specify their coordinate systems, like what we do with main.



Syntax: how to specify task.

Recursion: hole in the bucket

Profiling: where do I waste most time (and space)?

Program Structure

- Functions: used to break up complex problems into manageable elements
- Functions should be like math. theorems: they abbreviate knowledge, their proof is at another level.

- *Test* your functions before using them!

Unmanageable is a 400-line main() function.

Program ~= recipes
String of beads, (multithreading is for chefs).
Nested, beads inside beads

Proof that function works? No.
But per-function testing, and corner cases.
Your functions **must** do what they promise.
(If they don’t, they’ll make you miserable.)

Program Structure

- Levels.
  - e.g., from the *verbal* level to the *main* level
- Each new level is a 1:1 specification of the previous level, creating a *coordinate system*
- Each level should be intuitively clear given the previous level
- Some functions are merely *one* statement

< 15 lines

Breaking up ~ planning.

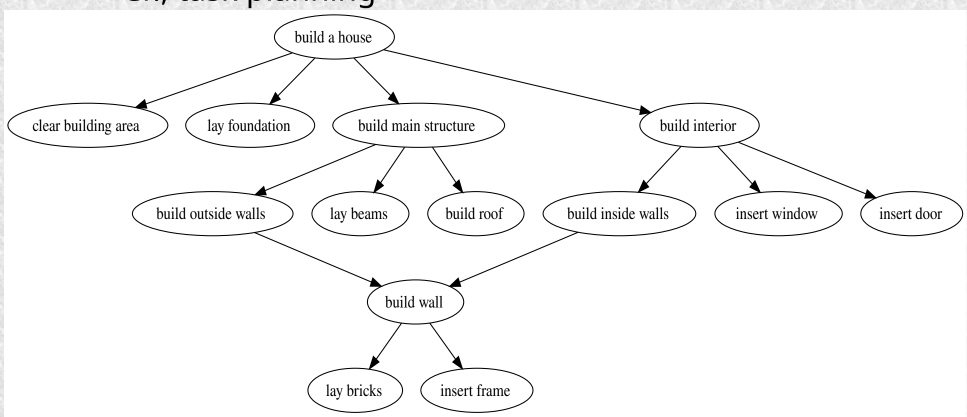
Single-statement functions are **good** if
- the call is simpler than the statement, or
- the call is clearer than the statement.

Compiler will optimize.

5

Program Structure

- Levels: *verbal, main, decomposition, auxiliary*
- Decompose until ‘manageable’
  - Cf., task planning



Verbal: “build a house”

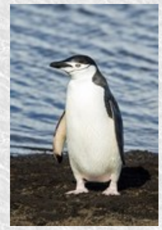
Decomposition: deeper & shallower sections

Good strategy: tell yourself what the program should do before you start an editor. Make a drawing.

The editor limits you to a linear format.
The compiler limits you to C++ code.

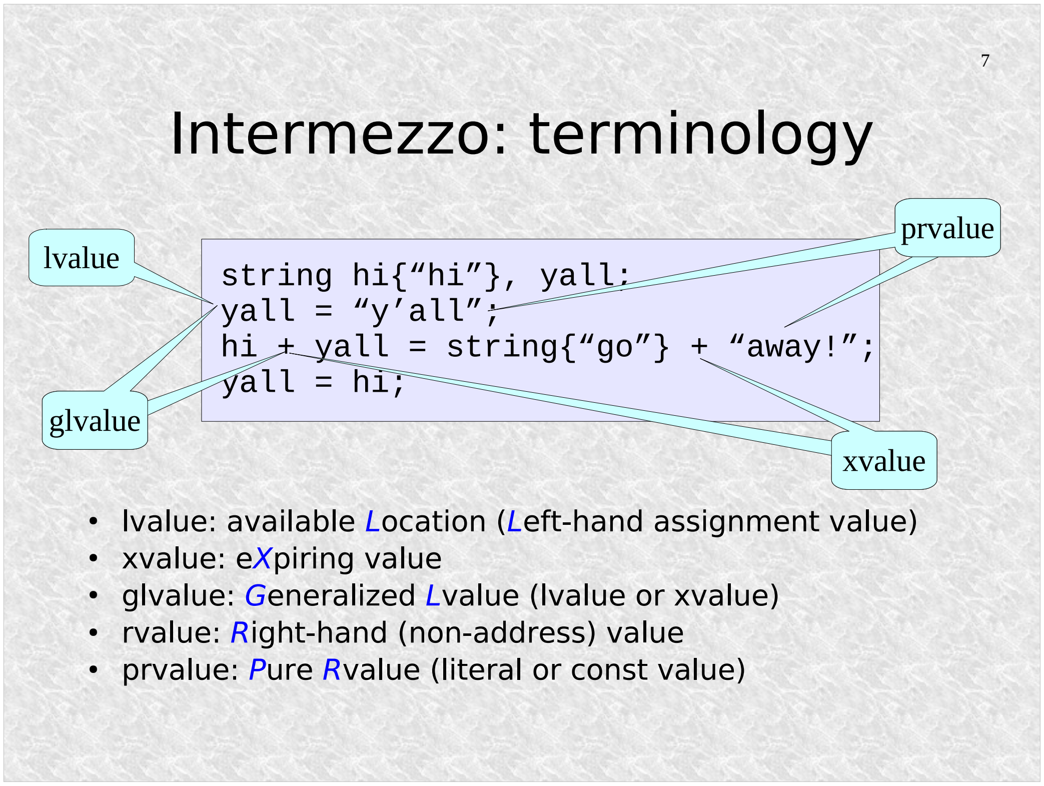
Then thinking, don’t be limited.

Intermezzo: types & constness



- Built-in vs. structured types
  - `double gravity = 9.81;`
  - `std::string bird{ "penguin" };`
- Const vs. non-const
  - `size_t const wingspan_B737 = 34;`

  - *const* soon becomes important;



In practice we use *lvalue* and *rvalue*

Third line is pointless, except for side effects!

`Hi + Yall` is a temporary object, as is `string{"go"} + "away"`. They go out of scope at the semicolon. At that moment, they are **destructed**.

8

Terminology:
pointers and references

- *Pointers* contain addresses, and can be 0:

```
size_t length = 13;
size_t *pLen = &length;
*pLen = 12;
```

- References bind to (g)lvalues, and **must** be initialized at definition time:

```
size_t &rLen = length;
```

- Rvalue references: bind to rvalues (temporary or no address at all):

```
string &&greet =
  string{"Hi "} + string{“all!\n”};
```

Rvalue references become important once we start returning objects from functions and once we pass temporaries to functions, which then must be modified by those functions.

A reference becomes an alias for what it binds to.

Lifetime extension for the value bound to
- rvalue references and
- const lvalue references

Function Example

```
struct Coords{
    int x, y;
};

int distance(Coords lhs, Coords rhs)
{
    lhs.x -= rhs.x; lhs.y -= rhs.y; // Obtain difference
    return sqrt(sqr(lhs.x) + sqr(lhs.y)); // Pythagorize
}

int main(int argc, char *argv[])
{
    x = stoi(argv[1]); y = stoi(argv[2]);
    Coords origin{0, 0};
    cout << distance(
                origin,
                Coords{x, y}
               );
}
```

function

parameter

argument

This Coords only exists briefly.

function call

A function is a piece of code with a name, and values that go in and out. It facilitates code reuse.

C++ (and C) default: call by value.
The *parameter* is a **copy** of the *argument*.
So lhs is a **copy** of origin.
And origin remains unaltered.

Coords{x, y} exists just until the ‘;’

Function Anatomy

- Generic Function Syntax:

```
returnType   name  (  parameterlist  )
{
    statement(s)
    and/or
    definition(s)
    and sometimes
    declaration(s)
}
```

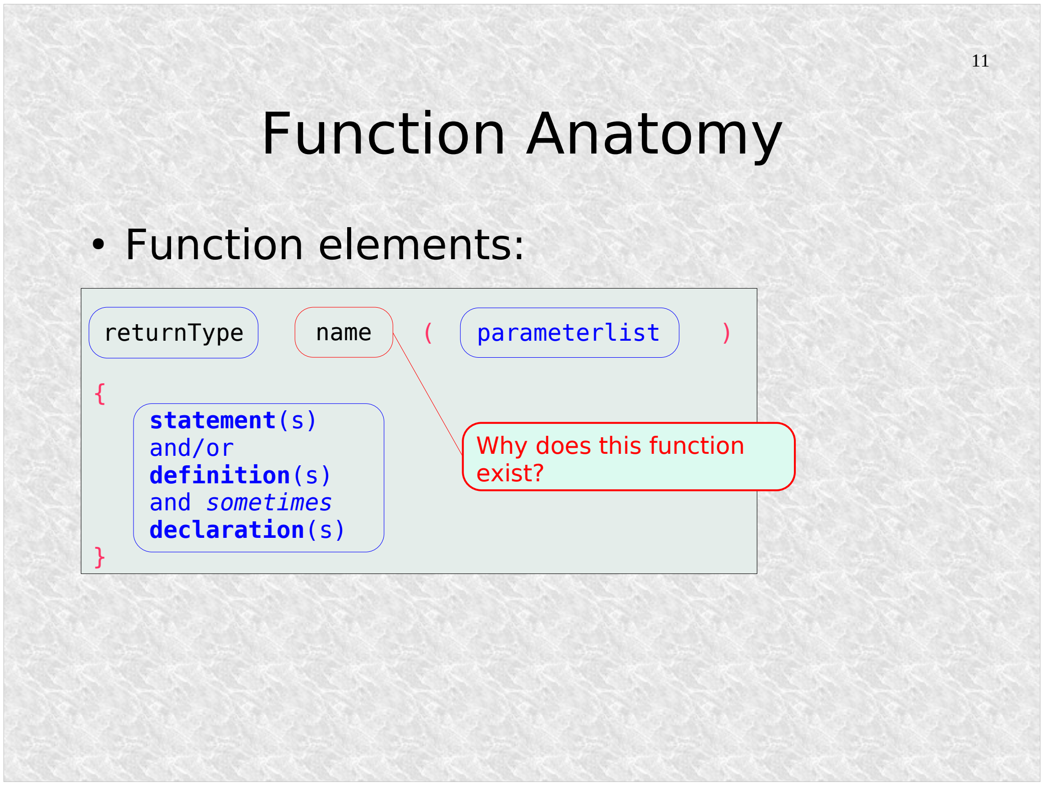
Red: required

Example:

```
unsigned long long square(long int value)
{
    return value * value;
}
```

Red: obligatory parts
Blue: optional parts

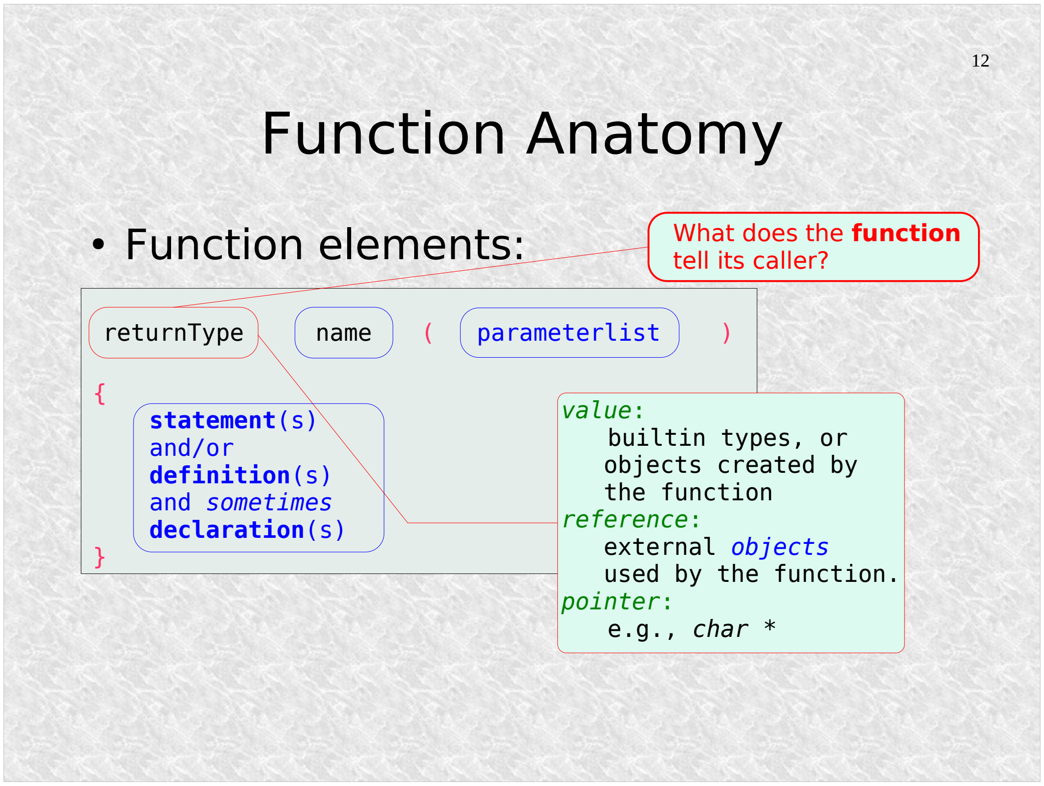
Since C++11, function doesn't need a name
if you use it only once, and define it then-and-there



Name the function for what it does!

A function performs a task in a grand design (See the C boek for examples: stepwise decomposition)

A function is a *capability* of an object of a *class* type: member functions like *string::length()* telling us something about the length of the information stored in a string object.

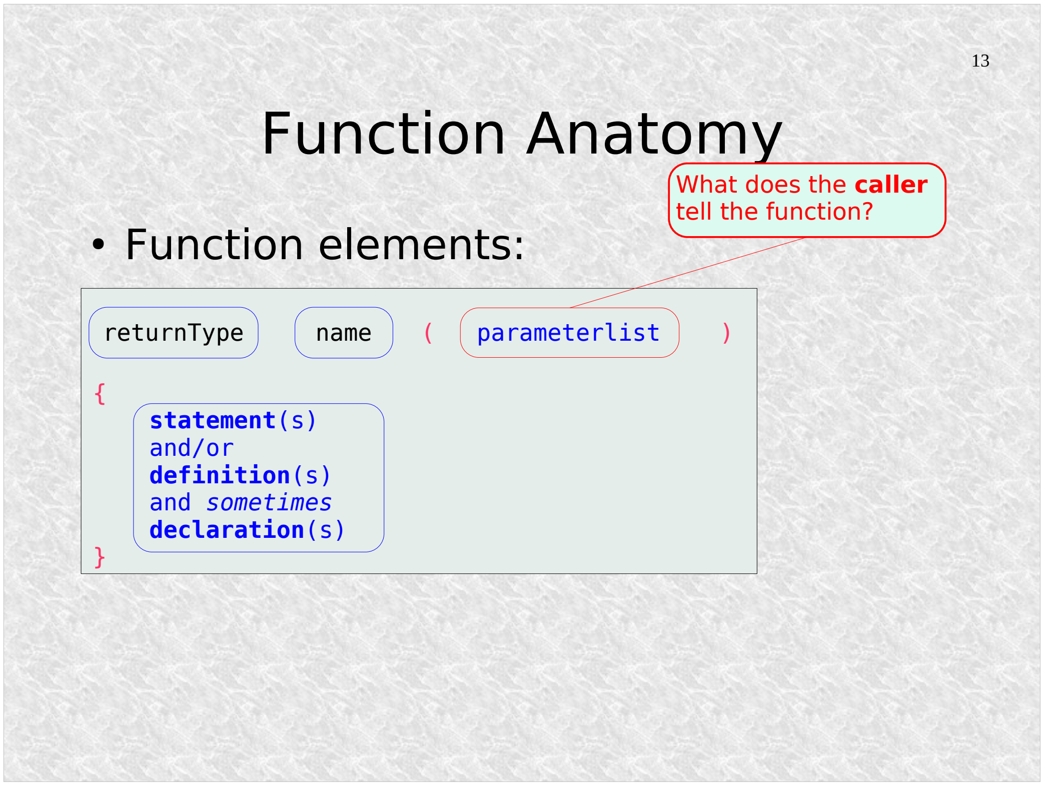


Void: nothing, the function is a Pascal-style procedure

Value: the return type is the type of the value and can be considered an *rvalue* (primitive types) or *anonymous lvalue* (class types); Distinguish *bool*, int, and *enum* return types.

Reference: the returned value refers to an existing value or object living ***outside*** of the function.

Pointer: the returned value *should point to* an existing value or object living outside of the function, or should be 0 (zero).



Value: function's parameter is initialized by a *copy* of the caller's *argument*
Reference: function's parameter is an *alias* for the caller's argument, which must be a variable or object. *A const* reference: function cannot modify the external variable/object
Pointer: function's parameter receives the *address* of a value or object living somewhere in memory. A *const* pointer: the pointer may not be modified by the function. A pointer to a *const:* the value or object may not be modified by the function.
Rvalue reference: the argument is an anonymous, temporary object, with which the function can do whatever it pleases.

Note: *signed* operands become *unsigned* in mixed expressions containing *unsigned* expressions.

Function Anatomy

- Function elements:

```
returnType  name  (  parameterlist  )
{
   statement(s)
   definition(s)
   declaration(s)
}
```

Define *actions* to be performed by the function,
Define *variables where first used*. (LOCALIZE!)

Functions should fit (approx.) on your screen

- *One* function, *one* responsibility
- Define *auxiliary* (support) functions
- Functions called by users should validate their input parameters
- Add *semantic* comment to statements, and *generic* comment below the function's definition

Define variables where first used:
- for documentation purposes
- to reduce complexity (possible interactions)
- to reduce namespace pollution
- when using objects, it reduces the amount of work, since one initialization may be prevented.

One exception: (explain in comment!)
define outside loop if initialization is expensive

Don't clutter the workbench,
grab a tool as you need it.

Generic comment below: normally you do maintenance on the code, and you don't need the generic explanation.

Function Anatomy

- Function parameters:
  - defined in a comma separated list.
  - parameters are *local variables, initialized by the caller.*
  - the initializing (outer) values are called *arguments.*
  - some examples of *parameters* and *arguments*:

`string *out`	,	`string const &in`	,	`char const *prompt`	,	`int count`
`&greeting`	,	`welcome`	,	`"$>"`	,	`3`

Examples of **arguments**:

pointer: put & before variable/object name:

&str

char const *: argument may be plain text or variable containing plain text:

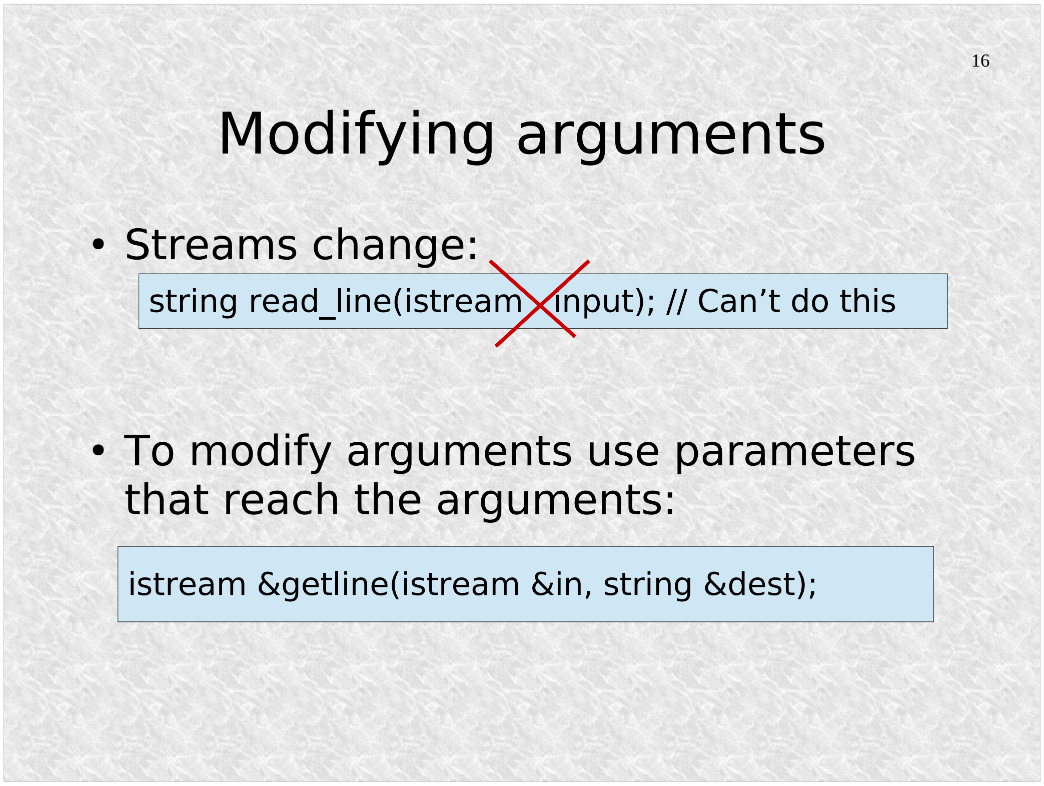
"hello", argv[1], str.c_str()

reference: use the variable/object name:

inString

value: use with simple values or (primitive) variables

5, intvar, doublevar



Input is a copy, but *input can’t reach the argument.

When manipulated inside the function, the stream changes its state. It makes no sense to do this to a copy of the original stream. Moreover, streams cannot be copied.

A pointer needs the ‘&’address-of in the calling function. That’s nicely clear.

A reference is less clear, but very widely used.

Giving back changed (or fully new) arguments to the calling function is called RBA – Return By Argument.

Reference Efficiency

- *Pass By Value*: often inefficient

```
size_t line_count(string book);        // copy the book ???
```

- Instead: pass by (const) reference:

```
size_t countLines(string const &book);
```

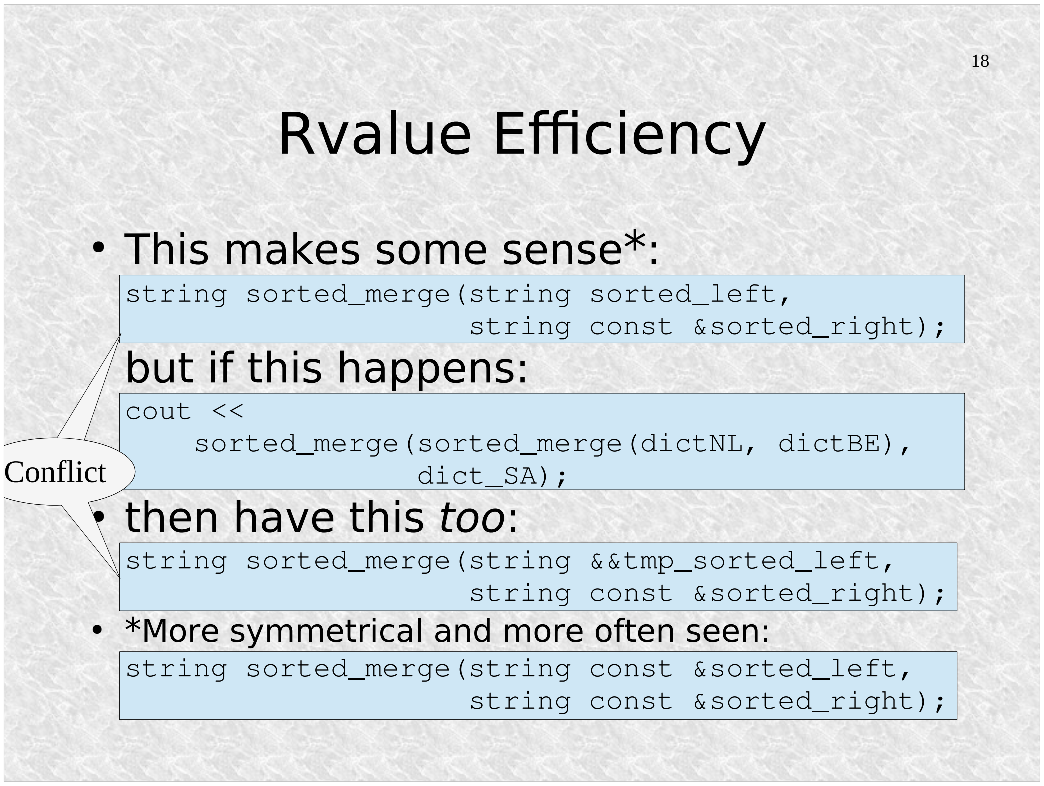
```
string &checkSpelling(string &book);
```

```
size_t nLines =
   countLines( checkSpelling(annotations)  );
```

Const reference: function can only do things that don’t change the object. Passing it on makes less sense (but is possible).

Reference: function can do anything to original object. Then it makes sense to pass on the reference.

Const references make no sense for built-in types. Handling the actual type is faster.

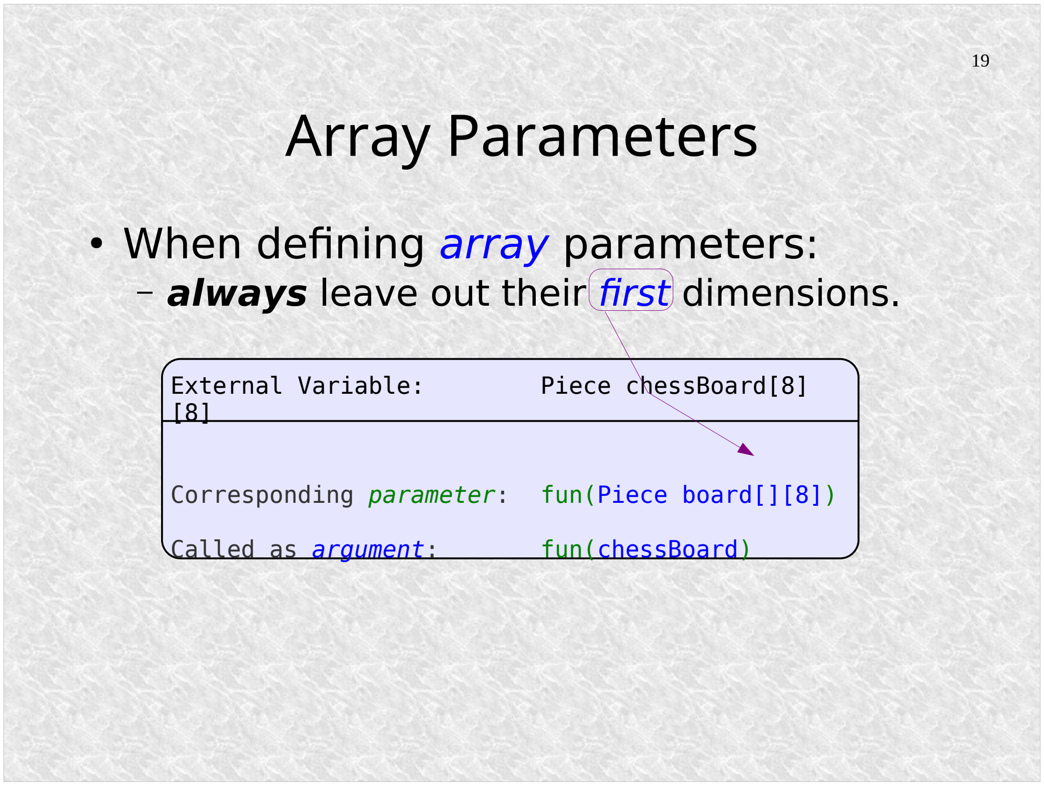


tmp_sorted_left Is a <u>nameless temporary object</u> in the calling function. sorted_merge Can do with it whatever it pleases, because it ceases to exist right after the call. Inside sorted_merge, move-semantics may be needed, involving std::move.

The problem of having sorted_merge called with an rvalue for sorted_right is tackled in part III of the course.

It is allowed to have two functions with the same name. The compiler guarantees that each is called in the right case. Cf. later slides

A non-rvalue version of sorted_merge is still needed.



Why this is true is covered in the upcoming lectures.

As an aside: defining an array parameter (like char *argv[]) is frowned upon. This too is covered in the upcoming lectures.

Functions

- Parameters (and return types):
  used for ***communication***
  - Who ***owns*** the information?
    - Use *const* for pointers and references if the function is not *conceptually* the owner
  - Use *pointer parameters* to assign values to variables *outside of* the function
  - Use (lvalue) *references* when passing *objects* instead of primitive data types to the function
  - Use *rvalue references* to refer to *temporary modifiable objects,* passed to the function

Char const *msg

Functions

- More *Good Practices*:
  - *return by argument (RBA)*:
    - use *pointers* when modifying external variables
    - put these pointers at the *beginning* of the parameter list
  - Use *const* for *pointers* and *references* and maybe values) that refer to entities which are ***not*** modified by the function.
    - In *headers* it is pedantic to use const for *value parameters*
  - Consider using *struct*-types (POD) when multiple, related values are returned by functions.

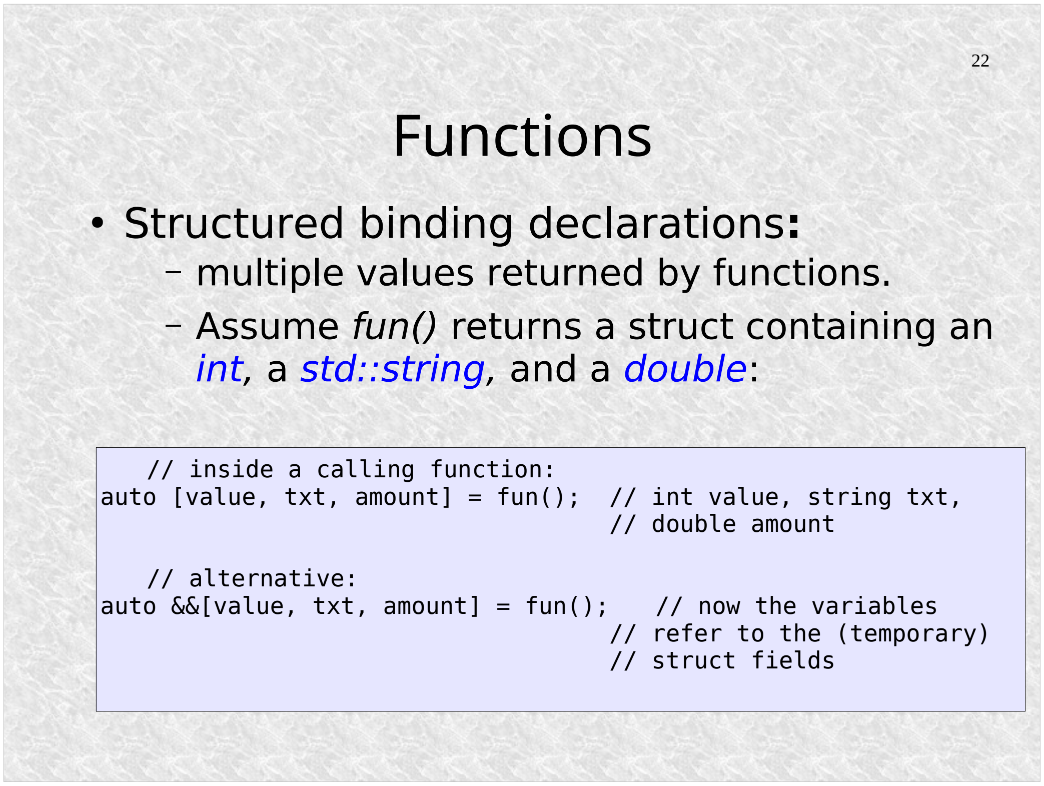
Return by argument:

This is not the function's returntype, but here parameters are used to return the function's products.

Use, e.g., a *bool* return type to tell the caller about the function's (un)successful completion.

const should be placed ***before*** what remains const, preferably *not* before the type:
Use: *string const &myString*
rather than: *const string &myString*
(see the Annotations and future lectures)

If you are going to be *pedantic*, do it only in the function definition, not in the declaration. (?! Yes, that works.)



Return by argument:

This is not the function's returntype, but here parameters are used to return the function's products.

Use, e.g., a *bool* return type to tell the caller about the function's (un)successful completion.

const should be placed ***before*** what remains const, preferably *not* before the type:
Use: *string const &myString*
rather than: *const string &myString*

About structured binding declarations:
It's also possible to do *auto* **&***[one, two three]*

(cf. the Annotations and also future lectures)

Functions

- Structured binding declarations
  - potentially useful in *init* sections of *for* stmnts: multiple types.

```
for (auto [count, collect, sum] = fun(); count--; )
{                       // int count, string collect, double sum
   // use collect and sum in this compound statement
}
```

  - *fun()* must return a *struct-type*[(1)]

[(1)]in part II we'll learn how to automate that.

Functions

- ***Value*** *parameters*:
  - Use value parameters for *builtin types* and class types initialized by the caller, but used as *local variables* by the function
  - Value parameters don't need ***const*** (but it doesn’t hurt if the parameter isn't altered)

```
void function(int value, string text)
{
    text += " whatever";
    value <<= 5;
    ...
}
```

e.g. int max(int lh, int rh)

25

Functions

- **Value** *return types*:
  - Use value return types when returning values of *builtin types* or class-type objects *created* by the function (*factory functions*)
  - Value return types don't need ***const***

```
string factory()
{
    cout << "Enter a character and a value: ";
    char ch;
    size_t value;
    cin >> ch >> value;
    return string(value, ch);     // why not {} ?
}
```

Demo stack frame :-)

Demo questions: why is it bad to return a reference to a local object?
Demo remark: upon returning an object, it becomes an anonymous temporary rvalue. No need to return it as an rvalue reference.

But it **is** ok to take a reference to such an rvalue:

```
string &&joined =
    join(Annotations, Cbook); //ok
```

Functions

- ***Reference*** parameters/return values:
  - Are used to return *existing* objects, used by the function and returned to the caller
  - Use const with references and pointers:
    - for ***parameters***: if *not modified* by the *function*.
    - for ***return*** types: the *caller* may *not modify* the values they refer to.

```
void function(istream &cin, string const &callerText);

string const &function(string const &inText)
{
    // use (but not modify) inText, then return inText:
    return inText;
}
```

doesn't modify: this is a *promise* the function makes to its caller.

may not modify: this is a *requirement*. If the caller disobeys the program may by invalidated.

Functions

- ***Rvalue reference*** parameters:
  - Are used for *temporary* objects that cease to exists once the function returns
  - Within the function they're *not* anonymous. To *anonymize* them, use *std::move.*
  - Plain *return values* appear as anonymous values; so do *rvalue parameters* returned via *std::move*

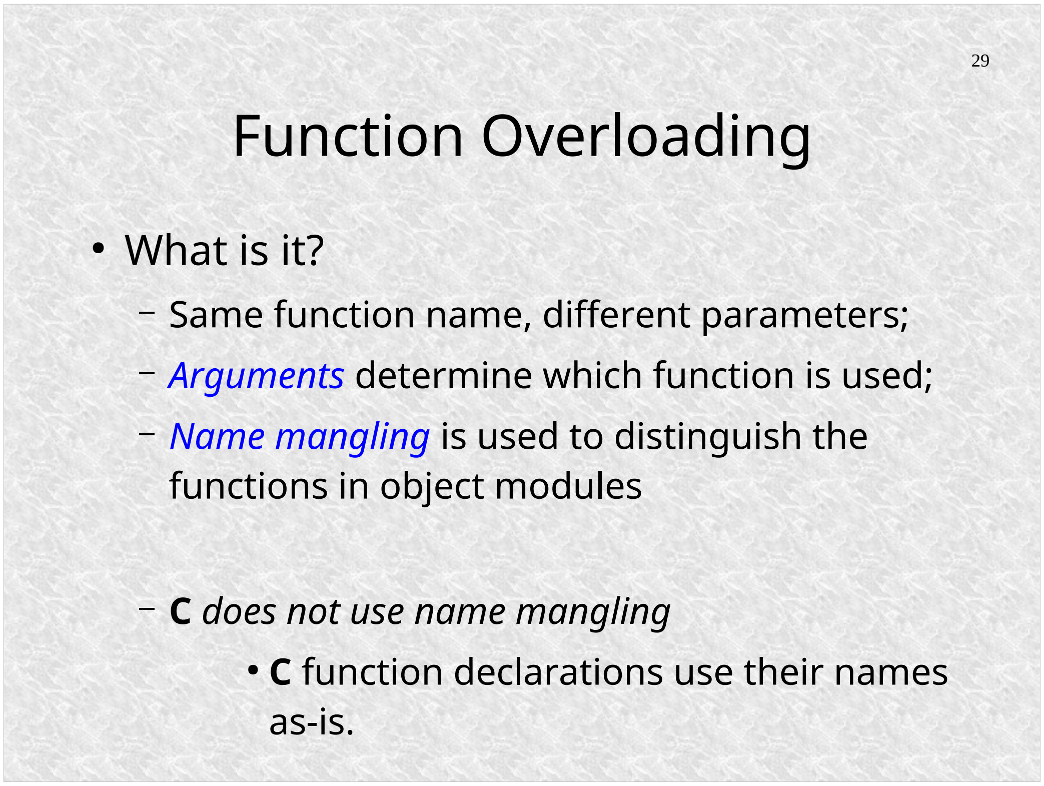
```
string function(string &&anon)
{
    // use anon as an ordinary string object
    ...
    return std::move(anon); // return as rvalue ref.
}
```

Functions

- ***Pointer*** parameters/return values:
  - as parameters: are used to assign values to objects or variables *living outside of functions*
  - as return types: are used to *return newly allocated data*
  - use them when it's the *natural form* (e.g. NTBSs)
  - use const to prevent data modification

```
bool option(string *value, int opt);

char const *programName(string const &prog)
{
    return prog.c_str();
}
```



Different *parameters*, not: different *return values*

Rvalue references

Functions can also be overloaded by defining const- and non-const variants of functions. This topic will be covered in Part II of this course.

Demo mangle

Function Overloading

- Why is it available?
  - Same function, different parameters. Examples:

```
string::find(char ch)                // 1
string::find(string const &argument)    // 2
```

    - (1) can use efficient implementations given that its argument is a single char;
    - (2) uses code processing multiple characters.
- Note: *return types* are ***not*** considered when deciding which overloaded function to call.

Return type may be ignored
Or converted

Function Overloading

- How is it used?
  - *Arguments* determine which function is called.
  - Example:

```
string::find(char ch)                  // 1
string::find(string const &argument)   // 2
```

```
int main(int argc, char *argv[])
{
    string program{ argv[0] };

    std::cout << program.find('.');    // calls (1)
    std::cout << program.find(".out"); // calls (2)
}
```

Do not get lazy
Different concepts => different names

The compiler may apply *argument conversion*:
char → int
object → object (const) reference
and quite a few more.

Function Overloading

- How does it work?
  - *Name mangling* is used to distinguish overloaded versions
    - *Name mangling* allows the compiler to inform the *linker* about which function to call in the final program:

```
double sqr(int arg)    links to, e.g.: _Z4sqri
double sqr(double arg) links to, e.g.: _Z4sqrd
```

What happens if we define a function called void _Z4sqri() ?

In that case name mangling occurs for *this* name, and the compiler may use:
_Z8_Z4sqriv

33

Function Overloading

- Calling **C** functions from **C++**
  - **C** does ***not*** use name mangling:

```
double sqr(int arg)        becomes   _Z4sqri (C++)
double sqr(double arg)     remains:  sqr     (C)
```

  - Consequently, the compiler must know the *language context*:
    - ***C** headers must be prepared for being used by **C++** programs*
      - using *#ifdef* preprocessor directives,or:
      - use/define special **C++** headers:

```
Prefer:   #include <cstdlib>
over:     #include <stdlib.h>
```

So, linking sqrt(16) from a library of **C** functions is referring to a diffent function than linking sqrt(16) from a library of **C++** functions.

The special header files (see later) are used to inform the compiler whether it's necessary to do name-mangling or not.

The Annotations give examples of headers used by both **C** and **C++**

Function Arguments

- Default arguments:
  - The compiler may provide *default* arguments to functions
  - Defaults may be used from the *last* to the *first* parameter
  - Default arguments are provided by *declarations* (*headers*), *not* by *definitions* (*sources*).
  - Ambiguity must be prevented (by us!)

Two functions only differ in defaulted parameter. When called without it: ambiguity.

Function Arguments

- Default arguments example:

```
string::find(char ch, size_type pos = 0);

istream &getline(istream &in, string &str,
                 char delim = '\n');
```

Realize: default *arguments*.

This also suggests that the *definition* should not use defaults, as they would be default *parameters*.

A Special Parameter Type

- Consider multiple, but an unspecified number of arguments
  - *Assignment*: define a function computing the sum of double values
  - Examples of how it's used (sort of...):

```
cout << sum(1, 2.5, 3.14);
cout << sum(12.45);
cout << sum(1, 2.3, 4, 5.6, -7, -8.9, 10, 11.12);
```

  - Why can't we use function overloading?

We actually can use overloading if there is a maximum to the number of arguments. But it’s a silly amount of repetitive work...

A Special Parameter Type

- Solution to the variable number of arguments problem:
  - *Initializer lists* accept any number of values.
  - To use initializer lists, do:

    #include <initializer_list>
  - Here's a function declaration:

```
double sum(std::initializer_list<double> list);
```

  - Here's how the function can be called:

```
sum({ 1, -2.3, 4, 5.6 });
sum({ -7, 8.9 });
```

adopt the habit to add a blank after { and before } to make the list stand out

A Special Parameter Type

- The implementation of the function *sum*:
  - the initializer list's member *size* returns the number of its elements
  - Use a range-based for-loop to visit all elements.
- Implementation:

```
double sum(std::initializer_list<double> list)
{
    cout << "There are " << list.size() << " elements\n";

    double ret = 0;

    for (double value: list)
        ret += value;

    return ret;
}
```

Note that we copy initializer lists, as they are about as small as two pointers.

Recursion

- Recursive function: a function calling itself
  - Either directly
  - Or indirectly
    - by calling another function that calls our function.
- Why use recursion?

Circles in call graph

Recursion has nothing to do with **C++**, but is a programming technique.

Refer to the **C**-book for more examples of recursion.

Factorial

Recursion

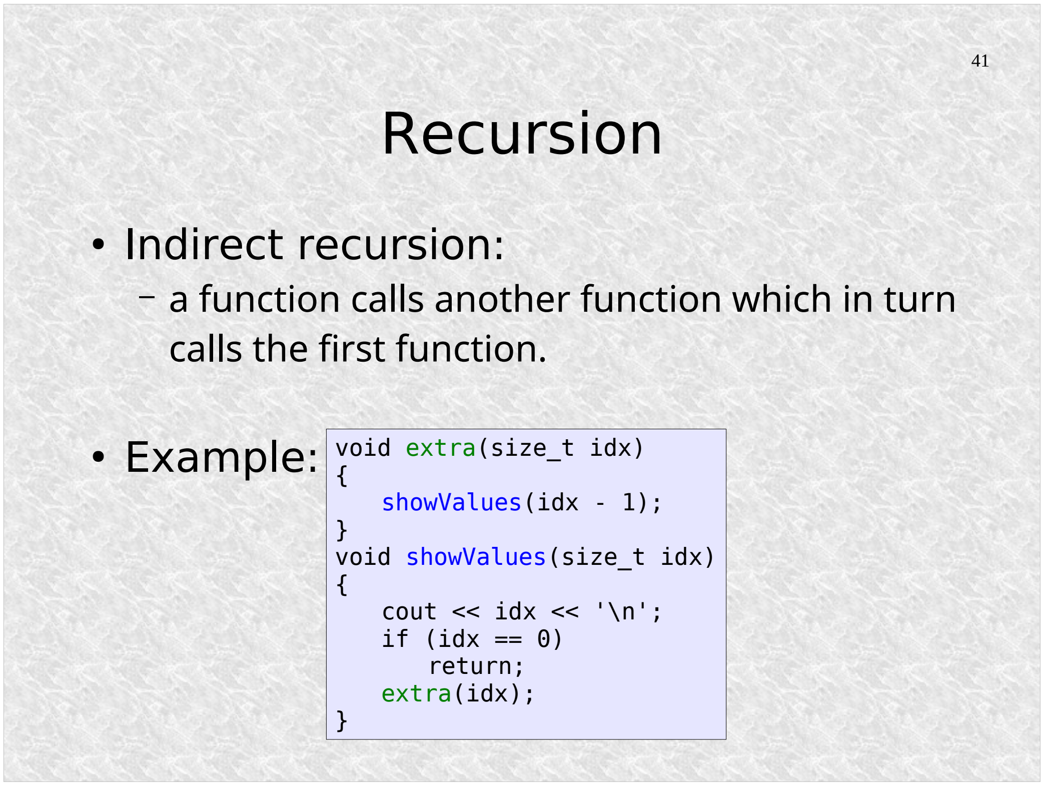
- Direct recursion:
  - a function calling itself.

- Example:

```
void showValues(size_t idx)
{
    cout << idx << '\n';
    if (idx == 0)
        return;
    showValues(idx - 1);
}
```

There is always a condition terminating the recursion. Here it is *if (idx == 0)*

There is always a statement in which the function calls itself: there problem is reduced to a *smaller* problem and that feature should be clearly visible in the problem statement.



What is the problem here?

The compiler doesn't know what *showValues()* is, when *extra()* is compiled:

Forward references are required when indirect recursion is used.

Usually that is not even noticed by the programmer, as either the class header file or a program header file already contains the function declarations.

Recursion

- Why use recursion?
  - The underlying problem is recursively defined
  - A *recursive data structure* is processed
- But *avoid* recursion:
  - When using *tail recursion* (cf. *showValue*'s initial implementation)
  - When an *easy to understand* *iterative* algorithm is available (also: *showValue)*

Factorial

Tree

Recursion

- A nice example using recursion
- We're going to play *compiler*...
  - Assignment:
    - print a *size_t* having an unknown value.
  - Problem:
    - We don't now what its *most significant digit* is.

Recursion

- Assignment:
  - print a *size_t* having an unknown value.
- Problem:
  - We don't now what its *most* *significant digit* is.
- Solution:
  - We know what its *least* *significant digit* is.

Recursion

- Assignment: print a *size_t* having an unknown value.
  - We don't now what its *most significant digit* is.
  - But its *least significant digit* is known.
  - Approach:
    - Determine the *least significant digit* (LSD);
    - Process a smaller value by removing the LSD;
      - *Recursion:* If the smaller value exceeds zero, apply the algorithm to that smaller value;
    - print the LSD.

Stack comes in handy.
A call to a function is a frame on the stack.

We store less-significant digits on the stack until we arrive at the MSD.
Then we take frames off the stack and print the digits stored there as we go.

Recursion: Examples

- Print a *size_t* having an unknown value.
  - Implementation:

```
void printDecimal(size_t nr)
{
    size_t lsd = nr % 10;          compute the LSD
    size_t smaller = nr / 10;      compute the smaller value

    if (smaller > 0)               if the smaller value exceeds
        printDecimal(smaller);     zero, apply the algorithm to it

    cout << static_cast<char>      print the LSD
                ('0' + lsd);
}
```

No tail recursion: here recursion is properly applied.

In order to print an unsigned we would have to know how many positions the value would require.

In that case we could set up an array to store the value.

This implementation, however, always works, irrespective of the value of MAX_UNSIGNED.

(Assuming that there's enough room in the stack, which will hold true in all practical situations)

Recursion

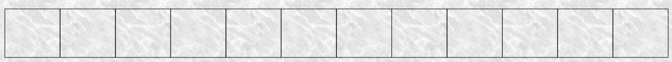
- Another historic example:
  sort an array (*quicksort, Hoare, 1962)*

Starting point: an array of *n* elements. Define *left* (its leftmost index) and *right* (the index just beyond the last element)

quicksort(array, left, right)

left

right

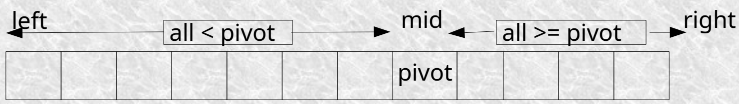


- We have not specified the array's actual size
- We're done once *left* >= *right*

Recursion: Examples

- Step one: *partitioning.*

Pick an element, e.g., the element *pivot* = *array[left]*, and put all elements smaller than *pivot* to the left of *pivot*, and all other elements to the right of *pivot*, placing *pivot* at *array[mid]*

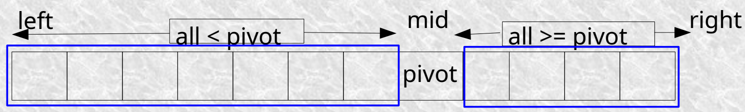


pivot = partition(array, left, right)

Recursion: Examples

- Step one: *partitioning*.

Pick an element, e.g., the element *pivot* = *array[left]*, and put all elements smaller than *pivot* to the left of *pivot*, and all other elements to the right of *pivot*, placing *pivot* at *array[mid]*

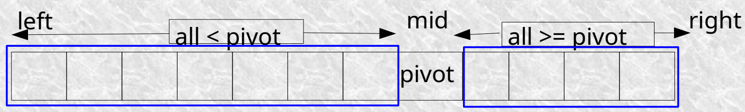


- Note: the array *left* to *mid* is an array like the original array
  the array *mid+1* to *right* is an array like the original array

Recursion: Examples

- Step two: *recursion.*

Pick an element, e.g., the element *pivot* = *array[left]*, and put all elements smaller than *pivot* to the left of *pivot*, and all other elements to the right of *pivot*, placing *pivot* at *array[mid]*



Use the same algorithm to sort, resp. the array *left* to *mid,*

quicksort(array, left, mid)

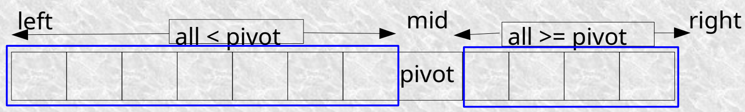
and to sort the array *mid+1* to *right,*

quicksort(array, mid+1, right)

Recursion: Examples

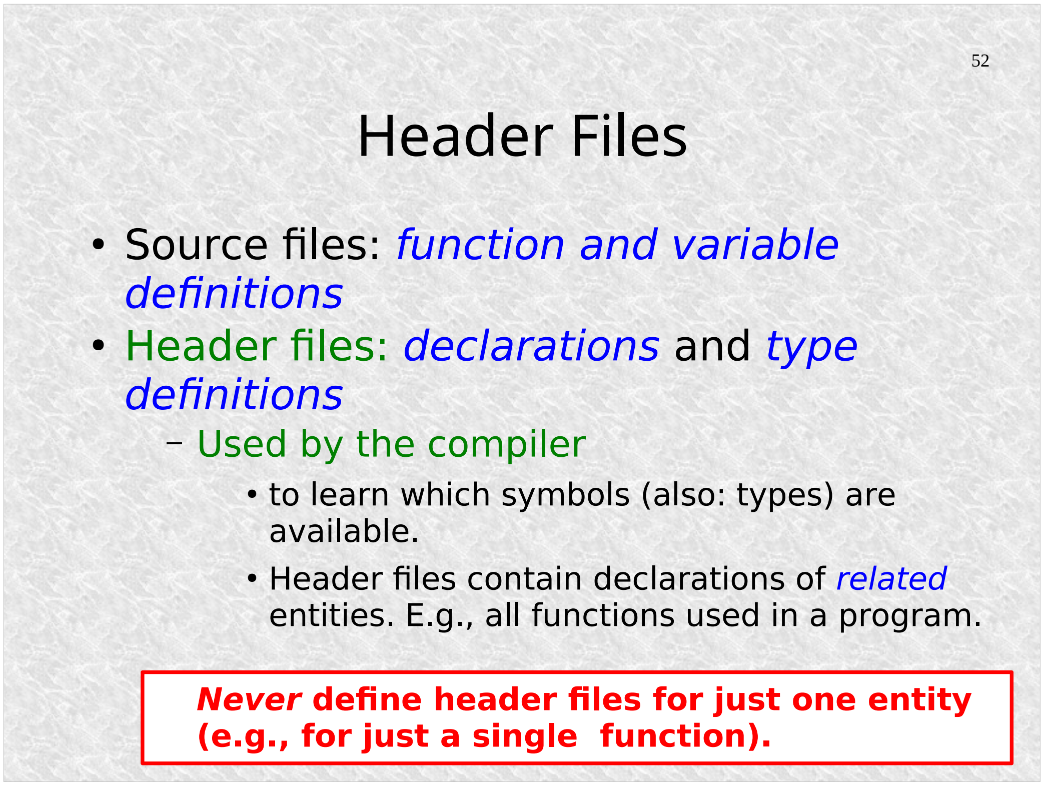
- Quicksort: *implementation*

Pick an element, e.g., the element *pivot* = *array[left]*, and put all elements smaller than *pivot* to the left of *pivot*, and all other elements to the right of *pivot*, placing *pivot* at *array[mid]*



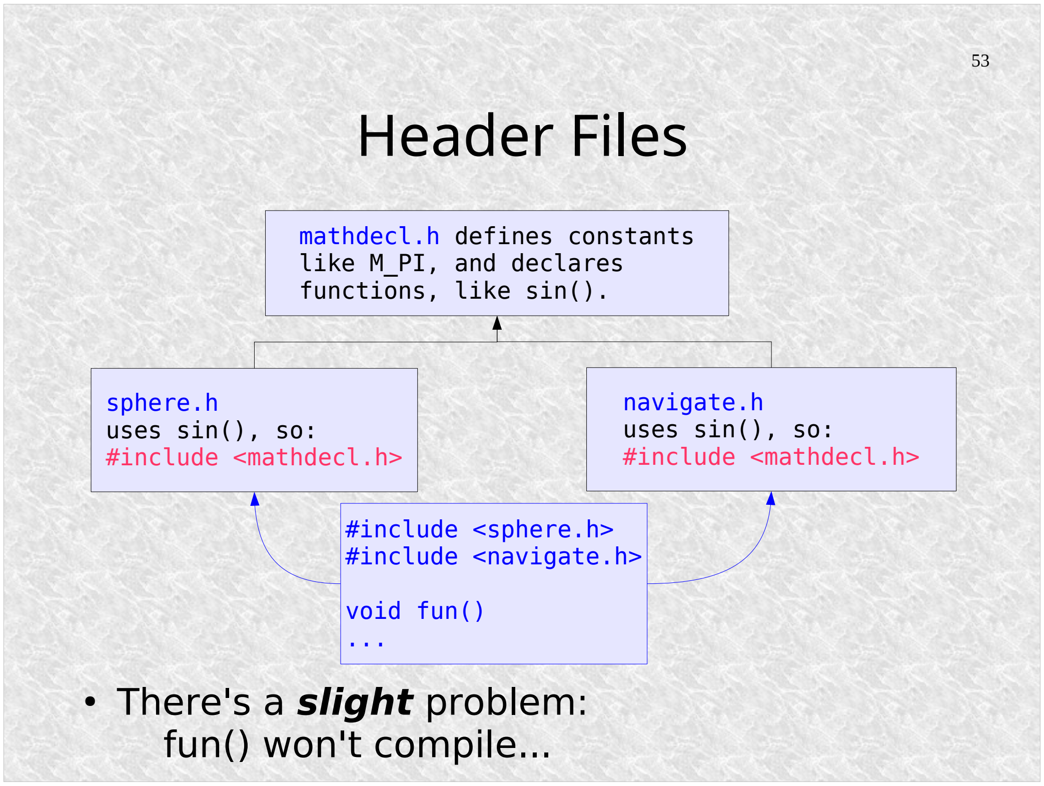
The implementation:

```
void quicksort(Type &array, size_t left, size_t right)
{
    if (left >= right)
        return;
    size_t mid = partition(array, left, right);
    quicksort(array, left, mid);
    quicksort(array, mid + 1, right);
}
```



This is the prelude to the way header files should be organized. Here we focus on *include guards*, but the concept will be expanded in a few lectures from now.

The compiler *assumes* that the declared symbols will eventually become available using whatever *object module(s)* or *libraries*.



Note the directions of the arrows:
they point to the files that are used by the file from where the arrows originate.

math.h is included twice.

Declarations aren’t even a problem. But e.g. M_PI is multiply defined in the bottom file.

54

Header Files

- To prevent double inclusions:
  - define *include guards:*

```
#ifndef INCLUDED_MYHEADER_H_
#define INCLUDED_MYHEADER_H_

// remaining contents of the header file

#endif
```

  - Include guards: are ***only required*** for header files which are *potentially included by other header files*

Double inclusions become a problem if header files define types. In **C++** this happens frequently, so include guards *must* be used.

Emacs users may profitably use my emacs macros (in dot.emacs in the cygwin files). They define:
Esc x i - put an includeguard around a file
Esc x is - insert system: #include <filename>
Esc x il - insert local: #include "filename"

Prefer include guards over #pragma once.
(See lecture on build utilities.)

Header Files

- ***C** headers must be prepared for use by **C++***, using *#ifdef* preprocessor directives:

```
// C header useable in C++ sources

#ifdef __cplusplus
   extern "C" {
#endif

double sqrt(double param);   // A C-function

#ifdef __cplusplus
   }
#endif
```

The header file is shown without the surrounding include guard. Note that the include guard should always be used.

Header Files

- Special **C++** headers can be constructed.

```
// A specialized C++ header (e.g., `cstring')

extern "C" {
#include "string.h"
}
```

Note: the include guard must still be added.

57

Libraries

- Libraries are used to store object modules
- *Object modules* (not just functions!) are linked to programs
- Therefore, use *one function per file*:
  - improves source-contents relationship
  - simplifies maintenance
  - reduces (re)compilation times
  - reduces program sizes
  - improves possibilities for profiling
  - allows you to make better use of a profiler

Exception to the rule:
Functions that are only called by one other function can be put in the same source file as their caller.

In that case, use an anonymous namespace, and define the calling function first. (But do declare the callee even *firster*, of course.)

Simply call the compiler to compile all sources:

```
g++ -c *.cc
```

This produces files having .o extensions.

59

Libraries

- Library construction:
  - 1. compile sources
  - 2. add objects to library

Use the program **ar**(1) to `archive' the object modules in a library:

```
ar rsv libfun.a *.o
```

This adds (replaces) all object files in the library, and creates an internal index that can be used later on by the linker.

Now the object files can be removed:

```
rm *.o
```

Libraries

- Library construction:
  - 1. compile sources
  - 2. add objects to library
  - 3. .o files may be removed (copies are in the libraries)
  - 4. store the library at a well-known or general location.

The `well-known' location may also be added or defined in the environment variable

```
    LIBRARY_PATH
```

E.g.,

```
    LIBRARY_PATH=path/to/my/libraries
```

Or:

```
    LIBRARY_PATH=$LIBRARY_PATH:\
               path/to/my/libraries
```

Libraries

- Library *use*:
  - g++ -o program program.o -Lpath/to/lib -lname
    - path/to/lib: path to library
    - name: name of the library, without *lib* prefix and *.a* extension

```
// assume library libmylib.a is in LIBRARY_PATH or, e.g., /lib
// assume program.cc uses objects from libmylib.a

g++ -o program program.cc -lmylib -s
```

Profiling

- What is efficient?
  - In general: hard to predict
  - A *profiler* should be used to find the *bottlenecks.*

- To be able to profile:
  - Profiling looks at the function-level
  - Use *-pg* when compiling and linking, no *-s*
  - Run the program
  - Profile: gprof -bp a.out gmon.out

Profiling

- Example:

```
size_t fun(std::string value);
size_t fun2(std::string const &value);
size_t callFun(string const &prog);    // calls fun  10,000,000 times
size_t callFun2(string const &prog)    // calls fun2 10,000,000 times
```

```
g++ -O2 -pg main.cc
a.out
```

```
gprof -bp a.out gmon.out
  %   cumulative   self    self
 time seconds    seconds   ms/call  name
100.00   0.03     0.03    30.00    callFun
  0.00   0.03     0.00     0.00    callFun2
```

Also -fprofile-generate
And -fprofile-use

Profiling

- Afterthoughts:
  - Do not trust your judgment: *profile!*
  - Don't overdo things. Profiling is seldom needed in this course
  - Profiling beats debugging. Don't fall for debuggers. Avoid them.
  - Basic rules of optimization:
    - First rule: *don't do it.*
    - Second rule: *don't do it yet.*

p.o. == sqrt(all evil)

Functions/Program Structure

- Program Structure
- Function Syntax
- A Special Parameter Type
- Recursion
- Header Files

- Additional slides (DIY, not covered in this lecture):
  - Using libraries and build utilities
  - Profiling

Syntax: how to specify task.

Recursion: hole in the bucket

Profiling: where do I waste most time (and space)?

Functions/Program Structure

That's All, Folks,
about Functions.