CSCI 135
Functions

Review

What do we know so far?

- Basic data types
  - Integral types: int/short/long, unsigned versions, char
  - Floating types: float, double
  - C++ Strings (a little)
- Assignment statements, including expressions
- Control Constructs
  - Selection: if, switch, ternary operator
  - Iteration: while, do while, for

Any language with iteration and assignment is expressive enough to represent all programs, BUT other constructs may be more convenient.

Modular Programming

Idea: separate the functionality of a program into independent modules, so that each module is concerned with only one aspect of the code.

- Breaks code down into manageable chunks to write (and divide across development team)
- Easier to debug – only one idea behind each module
- Easier to maintain – buggy modules may be replaced without worrying about ramifications on other modules
- Outsiders only need to know a module's interface, not the details of its implementation.
- Avoids duplication of code $\Rightarrow$ more compact, and also easier to modify if code has bug
- Allows for **information hiding** (Parnas, 1972) and **separation of concerns** (Dijkstra, 1974).

⚠C/C++ do not support modules, but functions can be used for some of the same benefits

⚠Objects are often confused for modules

Functions

General Functions:

- C: Every program is just a collection of functions – You've already used the `main` function.
- C++: Like C, with some complications due to objects (coming later)
- Other languages: also called procedures, subprograms

C Functions:

- Operators (special type of function): arithmetic (+, -, ...), logical (!, &&, ||, ...), etc.
- Predefined functions: *e.g.*, String.length(): String $\rightarrow$ Int
- Programmer-Defined functions: used similarly to predefined functions

⚠Unlike math functions, C functions may be executed for what they do (called a *side effect*) instead of what they return

Predefined Functions

Defined in libraries. C library names are preceded with "c" when included in a C++ program (*e.g.*, cmath vs. math). Some useful C++ ones:

- <cmath>, <cstdlib>: Original C libraries (called math and stdlib in C), for various mathematical functions (*e.g.*, root, trig, logs) and assorted utilities respectively
- <climits>: Sizes of integral types (for platform dependent code, or to make code portable)
- <ctime>: time related functions
- <cstdio>: C-style I/O (Not used in 135)
- <iostream>: `cin`, `cout`, `cerr` (C++ only)
- Lots of other libraries for various useful functions!
  ▸ See `http://www.cplusplus.com/reference/` (or in `/usr/include`)

Using a Library Function

```
#include <cmath>                    allows use of sqrt,pow
#include <iostream>
   using namespace std;
main() {
   double x,y;
   x = pow(3.,2.)                   stores 9.0 in x
   y = sqrt(x+7.)+2.5;              stores 6.5 in y
   cout << y << endl;               Using fct. cout for side effect
   return (0);
}
```

Terminology:

- The sqrt(...) above is called a function **call** (also **invocation**).
- The x+7. above is called the **argument** (also **actual argument**, **actual**)

A function call is just another expression; thus, used as before.

Ex: `y = sqrt(pow(3,2)+7)+2.5`

Functions: Math vs. C/C++

Math: A function takes 0 or more arguments, and returns a value of some type.
Important fact: the returned value is based purely on the arguments (NOT on any prior history/memory)

Functional Languages: Same as math, with a notable special case: the returned value can be another func. (called λ function).

C/C++ differs from math:

- Returned value may be based on prior history (*i.e.*, *impure* function)
- Sometimes uses non-standard notation
- Might not return a value (said to return a `void` type); executed purely for side effect.

Ex: `cout << x` is a [non-standard notation] function (<<) that takes two arguments (cout,x), executes a side effect, and returns something.

Functions With Memory

Ex: Pseudorandom numbers generator (PRNG) to simulate die roll

```
#include <cstdlib>
#include <ctime>
int main() {
   srand(time(0));      Seed PRNG with current time
   unsigned int r;      0 ≤ r ≤ maxint
   r = rand();          Number depends on history
   r = r % 6 + 1;       Transform to 1...6
   return (r);
}
```

`rand` is a predefined function that depends on history. But all C variables (so far) have lifetime local to enclosing unit (function in this case).

? How do we write our own history-dependent functions?

! Coming later (hint: storage classes)

Programmer Defined Functions

Issues to Consider:

- Function Declaration (called *prototype*):
  - Indicates function signature (number of arguments, types of arguments, return value type)
  - Needed for compiler to generate assembly code
- Function Definition: Code for what function does
- Function Call: 1) evaluate actuals, 2) transfer control to function, 3) 'replace' function call with its return value in calling expression.

Function Prototypes

<return_type> fctName (<formalParameterList>);

where formalParameterList is a comma-separated list of type–name pairs.
Ex: `double mypow(double base, double exp);`
Ex: `void foo();`

- Must be placed before any calls (so that compiler knows how to generate assembly for call
- Typically placed above `main()` (*i.e.*, in global scope) or at top of `main()` (*i.e.*, in main's scope)
- Several different acceptable syntaxes for formalParameterList (our text uses the above).
  Ex: `double mypow(double, double);`
- Terminology:"Formal" and "Logical" used interchangably (for parameters).

Function Definition

Written just like main! But a few differences:

- Function introduces its own scope (variables defined after function header are local to function; thus may have same name as variables in other functions)
- Can use logical parameters defined in header as variables. Do not redefine!
- Must return a value of the appropriate type (optional for void).

```
int intpow(int base, int exp) {
  int res=1;
  if (exp<0) {
    cout << "Error" << endl;
    return (0);
  }
  else
    for (int i=0; i<exp; i++) res = res * base;
  return (res);
}
```

Body is indented

Res is in intpow's scope

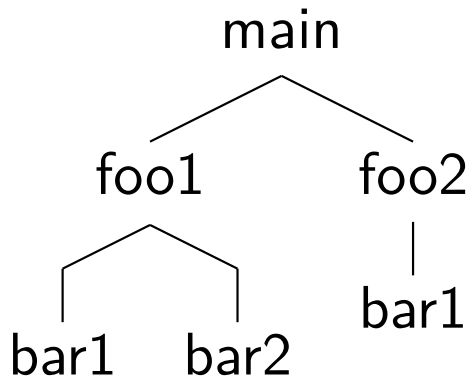
Side effect

Or just `res = 0`

Function Definition Placement

Need to place it at right place so that its visible when being called and compiler can generate correct code!

- Outside `main()`; not part of `main()` (or another function)
- All functions are independent and equal; no function is part of another.
- Use prototype of function in callers if they may be above it (recommended even if not).
- Most real programs will have a [deep] chain of function calls (**call tree**).



Example: Function Placement

Ex:

```
main() {
   double foo(int k);     Need prototype to make foo callable
   ...          // Calls foo
}
int bar(...) {                Definition of bar
   ...
}
double foo(int n) {           after bar; so no prototype needed
   int bar(...);              (but recommended anyway)
   ...          // Calls bar
}
```

Revisiting The Birthday Example

From before:

```
int age;
do {          // get positive age
   cout << ``Enter age (>0) '';
   cin >> age;
} while (age<=0);
// Print happy birthday message
if (age < 16)     // No frame printed if age < 16
   cout << ``Happy Birthday'' << endl;
else {                // Print framed message
   // print line 0 of frame: age asterisks
   // print line 1 of frame: * <age-2 spaces> *
   // print centered message
   // print line 3 of frame: * <age-2 spaces> *
   // print line 4 of frame: age asterisks
}
```

Revisiting The Birthday Example

```
int age;
void printAsterisks(int);      prototype
...                            other prototypes
do {        // get positive age
   cout << ``Enter age (>0) '';
   cin >> age;
} while (age<=0);
// Print happy birthday message
if (age < 16)     // No frame printed if age < 16
   cout << ``Happy Birthday'' << endl;
else {                // Print framed message
   printAsterisks(age);          replacing pseudocode
   printFrameLine(age-2);        actual is an expression
   printMsgLine(age);
   printFrameLine(age-2);        reusing function!
   printAsterisks(age);
}
```

⚠When decomposing program, think about function interface (inputs and outputs), NOT implementation/definition!

Birthday Example - 2

```
// Precondition: size>0
// Postcondition: size contiguous asterisks appear
//   on output (followed by newline)
void printAsterisks(int size)
{
  int i;
  for (i=0; i<size; i++)
    cout << "*";
  cout << "endl";
  return;
}
// other functions ...
```

might not work if size ≤ 0

size is a formal

local to printAsterisks scope

Don't declare size formal!

optional for void function

⚠When writing function body, think **only** about this function, not about callers (or callees). Write pre/post conditions for the function before starting.

Lessons for Decomposing into Functions

- A function should cover only one concern – *separation of concerns*. If your function does two indepent things, break it into two functions.
- If some body of code seems reusable, make it a function. Don't repeat that code in multiple places.
- Common guideline: Functions should fit into a standard display height.
- Think about function interface, not implementation – *i.e.*, **what** it does **not how** it does it.
- Be explicit about what the function's inputs/outputs (including types)
- State function pre-/post- conditions in comments. Writer of function should be able to code it with **only** this information. *i.e.*, pre-/post-conditions are a function **contract**.
- Don't forget concepts/terminology: actual vs. formal, prototype vs. definition

Parameter Types

Several types of formal parameters!

- Call by value: *i.e.*, input-only (from callee's perspective), read-only (can only assign to callee-local copy of parameter) $\Rightarrow$ communication of information from caller to callee, not the other direction

- Call by reference: *i.e.*, input/output, read/write (callee can write to parameter), 2-way communication

Often have some parameters passed by value and others passed by reference.

Parameter Types

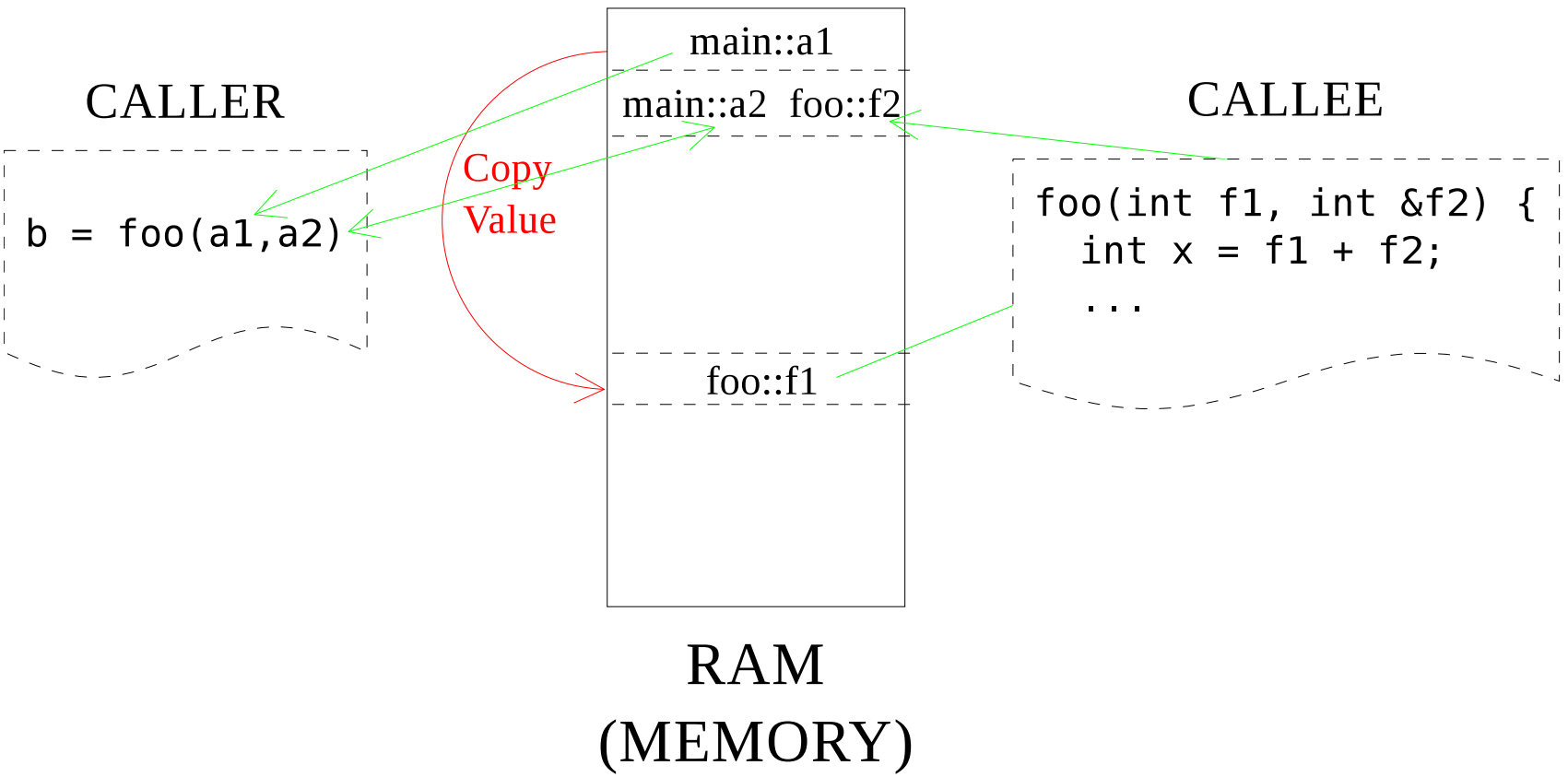
Several types of formal parameters!

- Call by value: *i.e.*, input-only (from callee's perspective), read-only (can only assign to callee-local copy of parameter)
  $\Rightarrow$ communication of information from caller to callee, not the other direction
  Implementation: evaluate actual (in caller), *copy* resulting value to somewhere callee accesses as a formal
- Call by reference: *i.e.*, input/output, read/write (callee can write to parameter), 2-way communication
  Implementation: send location (address) of variable in memory to callee

Often have some parameters passed by value and others passed by reference.

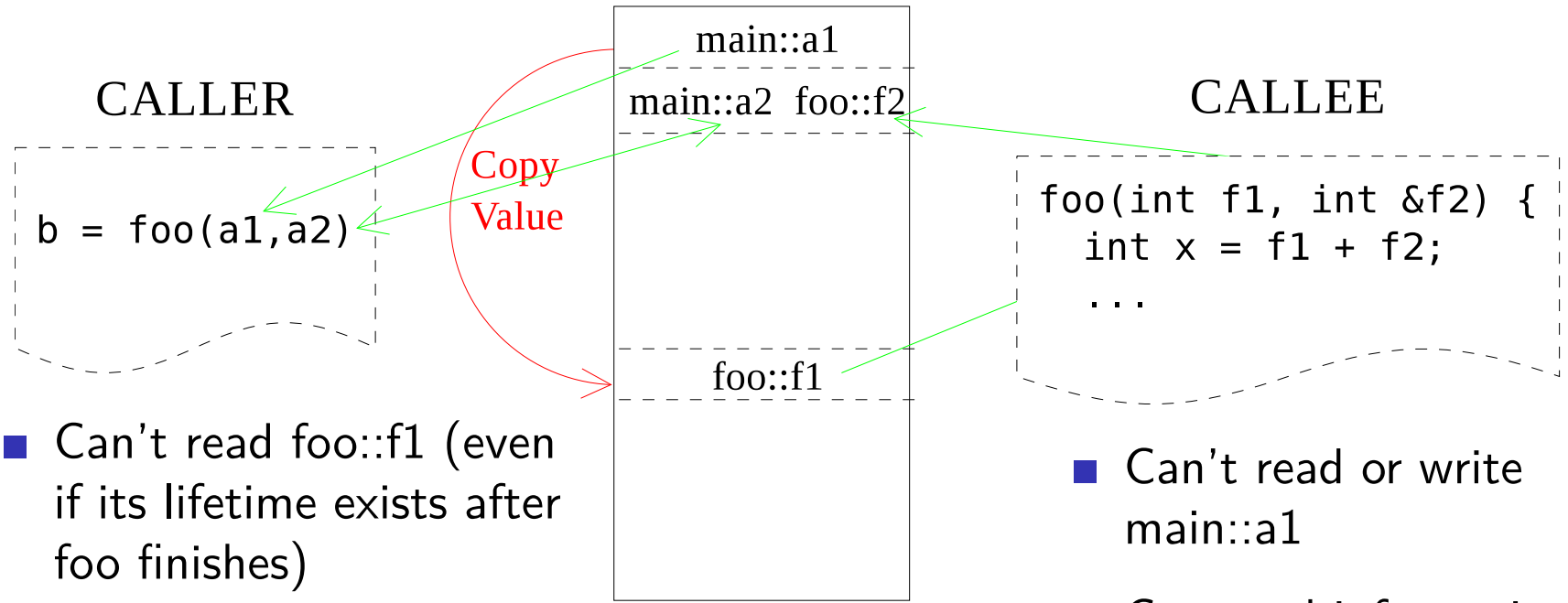
Implementing Parameter Passing

Ex: Let a1 and a2 be [caller's] actuals corresponding to [callee's] formals f1 and f2, passed by value and reference, respectively.
Note: actuals are expressions, not necessarily identifiers.



Implementing Parameter Passing

Ex: Let a1 and a2 be [caller's] actuals corresponding to [callee's] formals f1 and f2, passed by value and reference, respectively.
Note: actuals are expressions, not necessarily identifiers.



- Can't read foo::f1 (even if its lifetime exists after foo finishes)
- Can access foo::f2 indirectly through main::a2 alias

- Can't read or write main::a1
- Can send information back to main::a2 by writing to foo::f2

Example: Pass By Value

```
int main() {
  double celsToFaren(double);
  for (int tempc = 0; tempc<100; tempc+=10) {
    int tempf = (int) celsToFaren((double) tempc);
    cout << tempc << " celsius is "
         << tempf " fahrenheit" << endl;
  };
}
double celsToFaren(double cels) {
  return (cels * 1.8 + 32.);
}
```

- Formal and actual can (and typically do) have different names (and actual might be an expression anyway).
- Possible (but bad practice) for callee (celsToFaren) to assign to its formal parameter if passed by value like here (Ex: `cels = cels+20.0`).
- Even if celsToFaren assigns to cels, main::tempc does not change value.

Example: Pass By Reference

```
void swapValues(int& var1, int& var2); prototype
int main()
{
   int num1 = 5; int num2 = 8;
   swapValues(num1, num2);
   cout << num1 << num2 << endl;          prints 8 5
};
void swapValues(int& num1, int& num2)  & indicates
{                                          reference
   int temp = num1;
   num1 = num2;                          assigns main::var1
   num2 = temp;                          (and main::var2)
   return;
};
```

Parameter Guidelines

- As always, use meaningful identifier names (for both function and parameter names). Don't forget logical parameter names should be meaningful inside function (not to caller).
- Define function interface first, which includes parameter passing types (value, reference) and parameter data types (int, float, ...).
- Parameter order is important! And parameters can't be omitted.
- Document pre-/post- conditions of function, using names of logical parameters (not actuals, which are meaningless to function). Don't forget, this will be given to another programmer in team who knows nothing about caller.
- Most parameters will typically be passed by value. Pass by reference is most often used for:
  1. Parameter needs to be modified (*e.g.*, input routines)
  2. Parameter is big (many bytes) and we wish to avoid copying it

Making Pass By Reference Safer

⚠Reference arguments are dangerous since they allow the callee to change the caller (also a potential security risk). But they are still useful for big parameters.
Solution: `const` keyword to indicate that the parameter can not be assigned in callee – *i.e.*, parameter is read-only and caller data is protected.

```
int main() {
  void foo(const int &, int &);
  int m,n;
  foo(m,n);
};
void foo(const int & parm1, int & parm2) {
  parm2 = 20;          assigns 20 to main::n
  parm1 = 10;          Compile time error!
};
```

Overloading

Recall: our swap function only worked for int variables. What if we wanted to write another swap function that worked for other types (*e.g.*, double)?

⇒ Overload `swap` to refer to multiple functions, one for swapping ints, one for swapping doubles, etc.

You've already seen this, since you used + for all numeric types as well as string concatentation.

```
void swap(int& num1,
          int& num2)
{
  int temp = num1;
  num1 = num2;
  num2 = temp;
};
```

```
void swap(double& num1,
          double& num2)
{
  double temp = num1;
  num1 = num2;
  num2 = temp;
};
```

Overloading Issues

The two swaps are different functions, even though they share the same name.

❓ Which one gets called?
❗ Depends on context (in caller); *i.e.*, Compiler resolves invocation depending on whether caller calls swap with two doubles or two ints as arguments.

❓ What if first argument in caller is a double, and second argument is an int?
❗ Compiler looks for a compatible signature after implicit type conversion. In this case, its not possible since a pass-by-reference int can't be promoted to a double (but would have called the double version if it had been pass by value).
⚠ Lots of rules for prioritizing type conversion; Don't write code that may lead to confusing choices.

Example: Overloading Type Resolution

Suppose the following functions exist (with pass-by-value):

1. void foo(int n, double x);
2. void foo(double n, int x);
3. void foo(int n, int x);

? Which function is called by following calls?

- f(5,6):
- f(5.,6):
- f(5.1,6.):

Example: Overloading Type Resolution

Suppose the following functions exist (with pass-by-value):

1. void foo(int n, double x);
2. void foo(double n, int x);
3. void foo(int n, int x);

❓ Which function is called by following calls?

- f(5,6): # 3
- f(5.,6): # 2
- f(5.1,6.): # ???; depends on C/C++ rules

⚠ Avoid such programs!

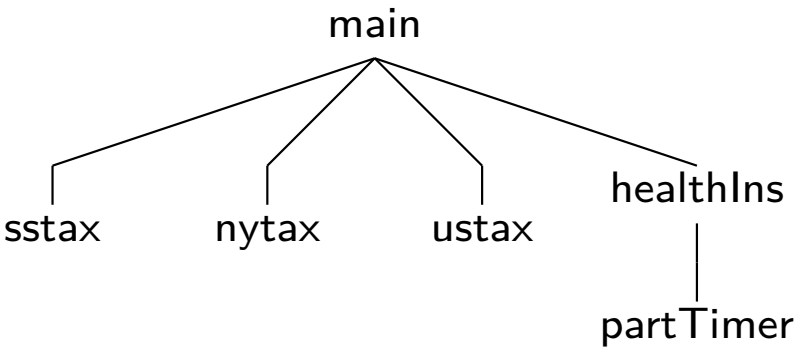
Default Arguments

- Allows some arguments to be omitted in call, with default values substituted (if omitted)
- Example:
  - `double volume (double length, double width=1, double height=2);`
  - width and height are defaulted if not substituted
  - A volume(5) call would actually call volume(5,1,2)
- Careful: Don't include default values in prototype. Prototype for above example would be:
  `double volume(double length, double width, double height);`

Example: Functional Decomposition

Problem: Given a gross salary, along with deductions (for social security, New York state tax, federal tax, and health insurance), print a net salary. Also, part-timers don't get health care, and are defined by a complicated formula based on their gross salary and hours worked.

Decomposition

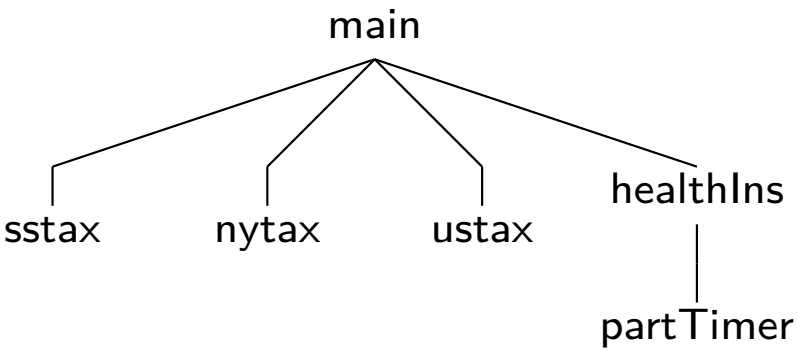


Note: main doesn't need to know about tax brackets!

Example: Functional Decomposition

Problem: Given a gross salary, along with deductions (for social security, New York state tax, federal tax, and health insurance), print a net salary. Also, part-timers don't get health care, and are defined by a complicated formula based on their gross salary and hours worked.

Decomposition



Note: main doesn't need to know about tax brackets!

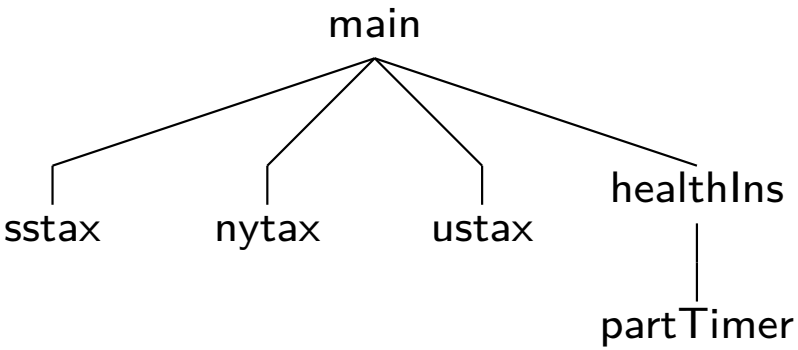
Function Prototypes

```
double sstax(double grossSalary);
double nytax(double grossSalary);
double ustax(double grossSalary);
double healthIns(double grossSalary,
                 int hours);
bool partTimer(double grossSalary,
               int hours);
```

Example: Functional Decomposition

Problem: Given a gross salary, along with deductions (for social security, New York state tax, federal tax, and health insurance), print a net salary. Also, part-timers don't get health care, and are defined by a complicated formula based on their gross salary and hours worked.

Decomposition



Note: main doesn't need to know about tax brackets!

Function Prototypes

```
double sstax(double grossSalary);
double nytax(double grossSalary);
double ustax(double grossSalary);
double healthIns(double grossSalary,
                 int hours);
bool partTimer(double grossSalary,
               int hours);
```

Next Step: Write pre-/post- conditions for each function. *Then*, start coding!

Testing a Multi-Function Program

A simplified approach:

- Unit Testing: testing of a single function *assuming* all other functions work.
- Integration Testing: testing of entire program

Unit Testing (of `foo`):

1. Write **stubs** for any functions not yet written. These stubs don't need to be completely correct, but need to be well-formed (*i.e.*, compilable, without syntax errors).
2. Compile whole program
3. Identify meaningful test cases for `foo`. At a minimum, border cases and a sampling of main cases
4. Write a `main` **driver** program that calls foo.
5. Test that `foo` works for these cases by running the driver program.

Exercises

1. Rewrite any of your earlier labs using functions.
2. Consider a pass-by-reference formal whose corresponding actual is an expression that is not a simple variable. Does your compiler allow it? Explain why the resulting behavior makes sense. Repeat if the parameter is a const.