

CS3520 Programming in C++ Classes and Functions

**Ken Baclawski
Fall 2016**

Topics

- Assignment #3
- Object-Orientation Fundamentals
- First tree program
 - Header
 - Source
 - Main program
- Call Stack and Exceptions
- Parameter Passing
- Second tree program
 - Header
 - Source
 - Main program



Assignment #3

Assignment #3

- Read words from the standard input.
- For each set of 5 words make one row in an HTML table.
- If a word is repeated in one column, then it should appear only once in the HTML table using rowspan.
- Ignore any extra words (i.e., a partial row of the table) at the end of the input.

Assignment #3 Example Input

A B C D E
A P C Q E
A B C D E
W B C Z X

Assignment #3 Example Output

	B		D	
A	P		Q	E
		C	D	
W	B		Z	X

Assignment #3 Example Output

```
<table border='1'>
<tr><td rowspan='3'>A</td><td rowspan='1'>B</td><td rowspan='4'>C</td><td rowspan='1'>D</td><td rowspan='3'>E</td></tr>
<tr><td rowspan='1'>P</td><td rowspan='1'>Q</td></tr>
<tr><td rowspan='2'>B</td><td rowspan='1'>D</td></tr>
<tr><td rowspan='1'>W</td><td rowspan='1'>Z</td><td rowspan='1'>X</td></tr>
</table>
```

Assignment #3 Example Output

- The border attribute specifies the width of the border
 - Use 1 pixel
- `<tr>...</tr>` specifies one row of the table
- `<td>...</td>` specifies one entry of the table
- The `rowspan` attribute specifies the number of rows that should be used by the entry

A	B	C	D	E
	P		Q	
	B		D	
W			Z	X

```
<table border='1'>
<tr><td rowspan='3'>A</td><td rowspan='1'>B</td>...</tr>
<tr><td rowspan='1'>P</td><td rowspan='1'>Q</td></tr>
<tr><td rowspan='2'>B</td><td rowspan='1'>D</td></tr>
<tr><td rowspan='1'>W</td><td rowspan='1'>Z</td>...</tr>
</table>
```


Assignment #3 Submitting and Grading

- Submit `Main.cpp` as well as the header (`.h`) and source files (`.cpp`) of any classes (if you are using classes).
- Classes are not required
 - It is okay to write all code in the main function of `Main.cpp`
 - If you want to have functions other than the main function, then they must be in one or more classes
- No command-line arguments
- Grading:
 - Compile with no errors or warnings (20%)
 - Correct execution on test data (30%)
 - Documentation (20%)
 - Correct style (30%)



Object-Orientation

Object-Oriented Programming

- Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects",
 - Objects may contain data, called *attributes* or *fields*
 - Objects contain code, called *methods*
- Methods can access and modify data attributes of the object
 - Within a method, the object is called “this” or “self”.
- OOP languages are highly diverse
 - Most popular are class-based
- Class-based OOP
 - Objects are instances of classes
 - The class of an object determines its type

Source: Wikipedia article on OOP

OOP Advantages and Disadvantages

- Advantages
 - Closer match to problem domain
 - Much more readable programs
 - Separation of concerns
 - More easily modified
 - More easily tested
 - More reliable
- Disadvantages
 - Takes longer to write a program
 - Requires training and discipline

Object versus Value

Object

has object identity

remains the same even
when its attributes change

Examples: person,
company, tree

A person is the same
person even if the person
changes their address

Value

has no object identity

changing a value produces
a different value with no
connection to the previous
value

Examples: 5, 3:30pm,
“Hello, world!”

The number 5 cannot be
changed to mean 6

Object-Orientation

- Objects have:
 - A state that is determined by data:
 - attributes
 - relationships with other objects
 - Behavior that is determined by functions (methods)
- The fundamental features of object-oriented systems:
 - Support for objects
 - Encapsulation of state and behavior
 - Information hiding

Class-Based Programming

- A class is a collection of objects, called the *instances* of the class
- A class *encapsulates* data and functionality in a module
 - Functions are grouped with their data
 - Data is only accessible to the methods of the class
- Limiting access to data is called “data hiding” or “information hiding”
- Large impact on productivity, especially for large systems
- Allowing access to the internals of a class from outside the class is called “breaking encapsulation”
 - This has a significant detrimental effect on productivity

Class in C++

- Evolved from the C `struct`
- In C++ `struct` and `class` are nearly synonymous
 - Differ only in defaults
 - One should not rely on the defaults
 - *Never* use the `struct` keyword
- A class is specified in two files:
 - Header file declares the interface
 - Source file defines the methods
- Everything declared in the interface is called a *member*
 - Only C++ uses this terminology

Style Requirements for Classes

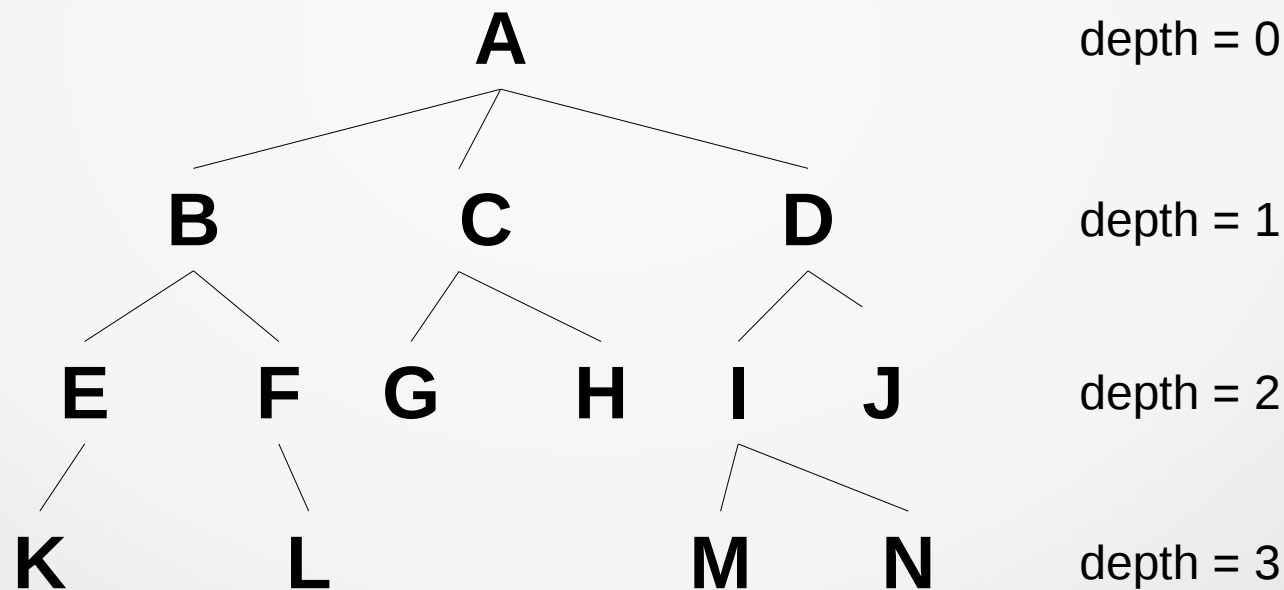
- Class names should be nouns or noun phrases
 - Write name in camelcase starting with uppercase
- Member function (method) names should be verbs or verb phrases
 - Write name in camelcase starting with lowercase
 - Can be public, private or protected.
- Data members are usually nouns or noun phrases
 - Write name in camelcase starting with lowercase and ending with an underscore
 - Always private
- All data and functions *must* be in classes
 - No global data or functions
 - The main program is the only exception



The tree1 program

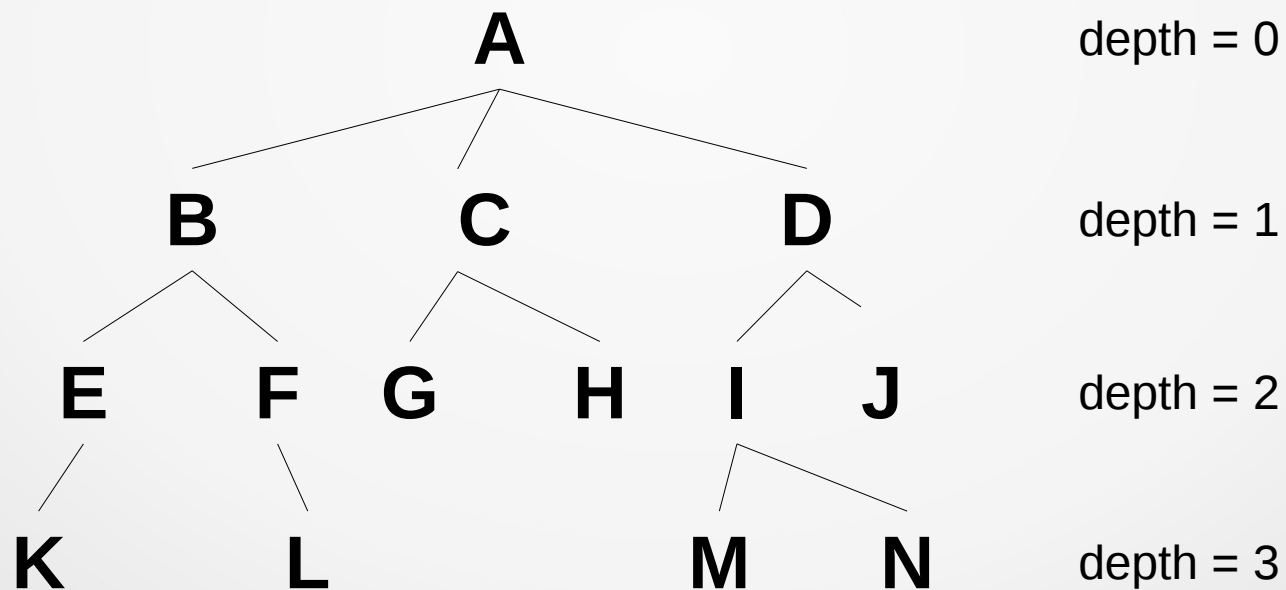
The tree1 package

- Each node of the tree has a label
- Read a tree from the standard input
- Print the tree to the standard output



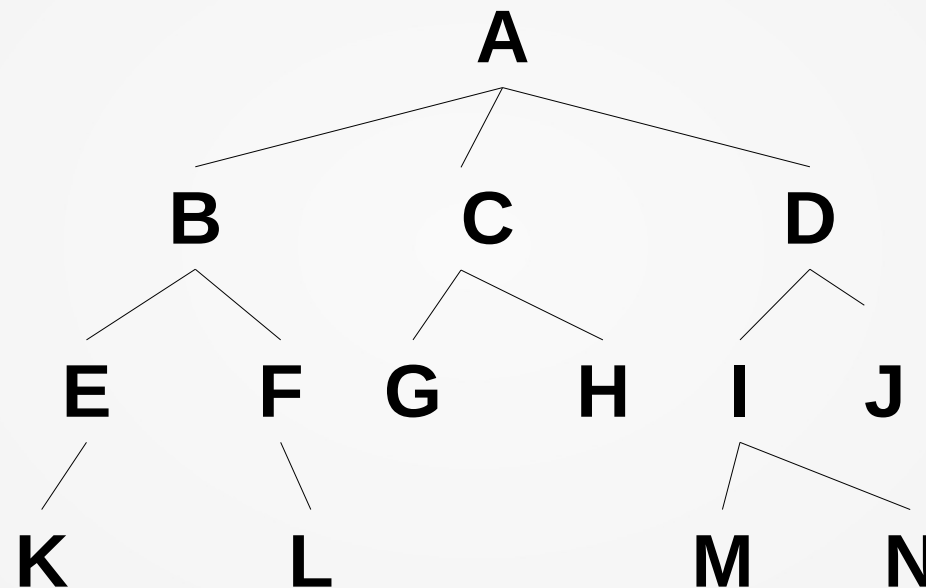
Sample Input

A[B[E[K[]]F[L[]]]C[G[]H[]]D[I[M[]N[]]J[]]



Sample Output

```
A [
  B [
    E [
      K [ ]
    ]
    F [
      L [ ]
    ]
  ]
  C [
    G [ ]
    H [ ]
  ]
  D [
    I [
      M [ ]
      N [ ]
    ]
    J [ ]
  ]
]
```



depth = 0

depth = 1

depth = 2

depth = 3

The tree data structure

- Very commonly used data structure
 - When a language is parsed the result is a tree: the *parse tree*
 - Important for compilers and search algorithms
- A tree consists of *nodes*
 - A node may have *child nodes*
 - The top node is the *root node*
 - Every node except for the root node has exactly one parent node
- Tree structures are *recursive*
 - Every node is the root node of a *subtree*

The tree1 package

```
#ifndef TREE1_TREE_H
#define TREE1_TREE_H

... includes ...

namespace ns {

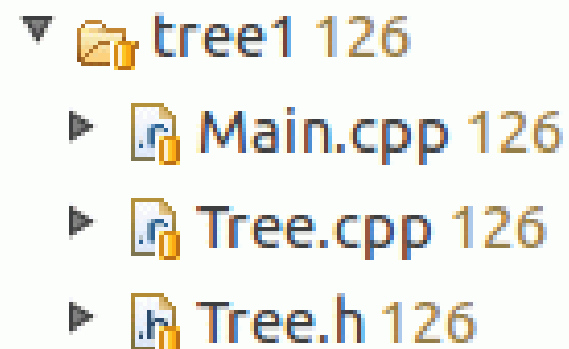
class Tree {
public:
... public methods ...

private:
... data ...
... private methods ...
};
}
#endif
```

Tree.h

```
... includes ...
using namespace "Tree.h"
... other using statements ...
...
void Tree::print(ostream& out) {
...
}
...
```

Tree.cpp



```
▼ tree1 126
  ► Main.cpp 126
  ► Tree.cpp 126
  ► Tree.h 126
```



Header File: Class Interface

Beginning of Tree.h

```
#ifndef TREE1_TREE_H
#define TREE1_TREE_H

#include <iostream>
#include <vector>

namespace tree1 {

/**
 * A simple tree class. A tree has a label and a list
 * of child trees. The tree can only be constructed by
 * reading it from an input stream. The purpose of the
 * class is to introduce object-oriented programming
 * techniques.
 *
 * @author Ken Baclawski
 */
class Tree {
public:
```

Namespaces

- Used to prevent name clashes
- Standard Template Library (STL) has namespace `std::`
- The `using` statement allows one to use names in a namespace without the namespace prefix
 - Can use a single name or a whole namespace
 - Using a whole namespace can add a very large number of names, so some style guidelines forbid it.
- Each example program and assignment has its own namespace
 - `asst01`, `asst02`, etc.
 - namespace name *must* be all lowercase

Beginning of Tree.h

- `#ifndef/#define` prevents double inclusion
 - Always on the first two lines of the file
- Put all includes next
- **No** using statements!
- Namespace declaration
- Class documentation block
 - Brief description ends with a period
- Class declaration
 - Always followed by `public:`

```
#ifndef TREE1_TREE_H
#define TREE1_TREE_H

#include <iostream>
#include <vector>

namespace tree1 {

/**
 * A simple tree class. A tree has a label and a list
 * of child trees. The tree can only be constructed by
 * reading it from an input stream. The purpose of the
 * class is to introduce object-oriented programming
 * techniques.
 *
 * @author Ken Baclawski
 */
class Tree {
public:
```

Includes

- `#include <...>` for system header files
- `#include "..."` for your own header files
 - There will be one header file for each class
 - A class header file can include header files of other classes
 - Cyclic includes are not possible
- These are literal (verbatim) includes not imports

Public Interface of the Tree class

```
public:
    /**
     * Construct a tree by parsing an input stream
     * An exception is thrown if the input is
     * not a valid tree.
     */
    Tree(** The input stream that specifies
           the tree. */ std::istream& in);

    /**
     * Print this tree on an output stream.
     * The tree is traversed and the labels
     * are printed in depth-first order.
     * The output of this method can be used
     * as input to the constructor.
     */
    void print(** The output stream the tree will
                 be printed on. */ std::ostream& out) const;
```

Public Interface of Tree class

- Each method has a documentation block
 - Brief description ends with period
- Each parameter has a documentation block
 - Best to document it immediately before the parameter
 - Can be documented within the method documentation block

```
public:
    /**
     * Construct a tree by parsing an input stream.
     * An exception is thrown if the input is
     * not a valid tree.
     */
    Tree(** The input stream that specifies
           the tree. */ std::istream& in);

    /**
     * Print this tree on an output stream.
     * The tree is traversed and the labels
     * are printed in depth-first order.
     * The output of this method can be used
     * as input to the constructor.
     */
    void print(** The output stream the tree will
                  be printed on. */ std::ostream& out) const;
```

Alternative Documentation Style

- Alternative style for documenting parameters
 - Either style is okay
- Disadvantages
 - Hard to be sure every parameter is documented if there are many parameters
 - Easy to misspell a parameter name
 - Requires more effort
- Advantages
 - Method interface is smaller

```
public:
    /**
     * Construct a tree from an input stream.
     * An exception is thrown if the input is
     * not a valid tree.
     * @param in The input stream that specifies
     * the tree.
     */
    Tree(std::istream& in);

    /**
     * Print this tree on an output stream.
     * The tree is traversed and the labels
     * are printed in depth-first order.
     * The output of this method can be used
     * as input to the constructor.
     * @param out The output stream that tree
     * will be printed on.
     */
    void print(std::ostream& out) const;
```

Public Interface of Tree class

- The first function is a constructor
 - Creates and initializes instances
 - Name is the same as the class
 - Can have several constructors with different parameters
- Exceptions are explained later
- Must show `std::` prefix because there are no using statements
- Parameter is passed by reference
 - The `&` indicates the reference type
 - Parameter passing is a big topic
- Other methods must have a *return type*
 - The `void` type means nothing is returned

```
public:
    /**
     * Construct a tree from an input stream.
     * An exception is thrown if the input is
     * not a valid tree.
     */
    Tree(** The input stream that specifies
           the tree. */ std::istream& in);

    /**
     * Print this tree on an output stream.
     * The tree is traversed and the labels
     * are printed in depth-first order.
     * The output of this method can be used
     * as input to the constructor.
     */
    void print(** The output stream the tree will
                  be printed on. */ std::ostream& out) const;
```


Method Annotations

- Written after the method interface
- The `const` annotation
 - Method does not change the object
 - *Observer* method
 - Other methods are *modifier* methods
- Other annotations will be introduced later
- One can specify what kinds of objects can be thrown
 - Similar to the Java syntax
 - Now deprecated in C++

Data Members of Tree class

- Data members
 - Always private
 - Also called attributes or fields
- Field names end with underscore
 - Helps distinguish fields from parameters
 - One of many naming conventions for fields
 - Style requirement for this course
- Always initialize the fields with built-in types

```
private:
    /** The label of this tree. */
    const std::string label_;
    /** The depth of this tree. */
    const int depth_ = 0;
    /** The subtrees of this tree. */
    std::vector<Tree> children_;
```

Private Methods of the Tree class

- A constructor can be private
 - Can only be used by other methods of the class
- Static methods do not have access to the data members of the class
 - Use public static methods rather than global functions
 - Private static methods are used when the same code occurs often in the implementation of other methods
- Class definitions always end with a semicolon in C++
- Final brace closes the namespace
- Final #endif closes the #ifndef

```
/**
 * Recursively parse a tree from
 * an input stream. This method throws
 * an exception if the end of data has
 * been reached without finding a word.
 */
Tree(** The label of the tree. */
    const std::string& label,
    /** The depth of the tree. */
    const int depth,
    /** The input stream for reading
    the subtrees. */ std::istream& in);

/**
 * Read one word from the input stream.
 * This method throws an exception if
 * the end of data has been reached
 * without finding a word.
 *
 * @return The word that was extracted
 * from the input stream.
 */
static std::string readWord(
    /** The input stream. */ std::istream& in);
};

}

#endif
```

Style Requirements for Header Files

- Parts *must* be public, protected and private in this order
- File extension *must* be `.h`
- Base name of the file *must* be the class name
- *Must* not have any implementations of methods
- *Must* start with `#ifndef` and `#define`
- *Must* have all `#include` statements immediately after `#define`
- *Must* enclose interface in a namespace block
- Should not have any `using` statements
- *Must* end with `#endif`



Source File: Class Implementation

Beginning of Tree.cpp

- Start with includes
- Always include the class header file
- Library includes are specified in angle brackets
- Program includes are specified in quotation marks
- It is okay to have `using` statements in the source file
- Should include the namespace of the package

```
#include <vector>
#include <iostream>
#include <stdexcept>
#include "Tree.h"
using namespace std;
using namespace tree1;
```

Rest of Tree.cpp

- The method implementations are in any order
- Each method must have the `Tree::` prefix
- Parameter types must match the interface
 - Parameter names do not have to be the same as in the interface
- No documentation blocks are required
 - Already documented in the header file

```
Tree::Tree(const string& label,
           const int depth, istream& in)
    : label_(label), depth_(depth) {
    ...
}

Tree::Tree(istream& in)
    : Tree(readWord(in), 0, in) {}

string Tree::readWord(istream& in) {
    ...
}

void Tree::print(ostream& out) const {
    ...
}
```

Method Implementation

- Reads a word from an input stream
- Throws an exception if the input stream has ended or is invalid
- Use `throw` to raise an exception
 - It immediately terminates the method invocation
 - Used for error conditions
 - Any object or value can be thrown
 - The `domain_error` exception is from `<stdexcept>`
- This code occurred three times in the implementations of other methods, so it was convenient to make it a method
- The method does not use any of the data members so it was declared to be `static`

```
string Tree::readWord(istream& in) {  
    // Check that there is something to read.  
  
    if (!in) {  
        throw domain_error("Unexpected "  
                           "end of input");  
    }  
  
    // Read the next word and return it.  
  
    string word;  
    in >> word;  
    return word;  
}
```


Initialization of data members

- Initialization of data members precedes the body of the function
 - Overrides the initializations in the interface
- Initializations occur in the order of data member declaration
- Never depend on the order of initialization
- If a data member is `const` then it can only be initialized before the body of the function

```
Tree::Tree(const string& label,  
           const int depth, istream& in)  
    : label_(label), depth_(depth) {  
    ...  
}
```

Constructor implementation

- Comments within the body of a constructor or method are required
 - There should never be a block of code longer than 6 lines without a comment
- The `for(;;)` statement is an infinite loop
 - Pronounced “forever”
 - Stroustrup preferred this to `while(true)`
- This constructor is recursive
 - Proof of termination:
Every call to the constructor reads at least one word, so it will eventually finish
- Can you state and prove the loop invariant?

```
Tree::Tree(const string& label, const int depth,
           istream& in)
: label_(label), depth_(depth) {

    // The list of children must begin with a left bracket.

    const string bracket = readWord(in);
    if (bracket != "[") {
        throw domain_error("Incorrect tree format: "
                           "list does not start with a left bracket");
    }

    // Read the subtrees until a right bracket.

    for(;;) {
        const string word = readWord(in);

        // If the word is a right bracket,
        // then the tree is complete.

        if (word == "]") {
            return;
        }

        // Otherwise, construct a tree
        // and add it to the list of children.

        children_.push_back(Tree(word, depth_ + 1, in));
    }
}
```

Constructor Implementation

- A constructor can invoke another constructor
- This is a new feature of C++11
- All this constructor does is invoke another one
- The constructor being invoked is private, so the only way that a Tree object can be constructed outside the class is with this constructor

```
Tree::Tree(istream& in)
    : Tree(readWord(in), 0, in) {}
```

Implementation of `print` method

- The `print` method does not modify the tree so it is annotated with `const`
 - It is an observer method
- The indentation makes the output more readable
 - It is whitespace so it does not affect whether the output can be parsed
- A subtree with no child trees is called a *leaf node*

```
void Tree::print(ostream& out) const {  
    // The indentation is twice  
    // the depth of the tree.  
  
    const string indentation(2 * depth_, ' ');  
  
    // Indent the label and open  
    // the child list with a left bracket.  
  
    out << indentation << label_ << " [";  
  
    // If there are no children,  
    // then end the list on the same line.  
  
    if (children_.empty()) {  
        out << " ]" << endl;  
    } else {
```

Implementation of `print` method

- The `print` method is recursive
 - Each child tree is printed using the same method
 - The child subtrees have greater depth so they will be indented more
- The `for` loop syntax is new
 - Iterates over the elements of a container
 - Much simpler than the syntax in the textbook

```
// If there are children, then
// list them on separate lines.

out << endl;

// Explore the tree recursively
// using depth-first search.

for (const Tree& child : children_) {
    child.print(out);
}

// Close the list with an indented
// right bracket.

out << indentation << "]" << endl;
}
```

Style Requirements for Source Files

- The class implementation is in the *source* file.
 - File extension *must* be `.cpp`
 - Base name of the file *must* be the class name.
 - *Must* not use the `inline` specifier
 - *Must* have all `#include` statements at beginning
 - May have `using` statements
- Some classes do not have a source file.



Main Program

Beginning of Main.cpp

```
#include <iostream>
#include <stdexcept>
#include "Tree.h"
using namespace std;
using namespace tree1;

/**
 * @namespace tree1 Simple Tree Structure. The tree structure is read from the
 * standard input and then printed to the standard output. This program is not
 * the most efficient way to construct a tree. The purpose of the program is
 * to introduce object-oriented programming techniques.
 *
 * @author Ken Baclawski
 */

/**
 * Main program for the simple tree structure. Read a tree and print it.
 * @return The status code. Status is 0 for normal termination
 * and 1 for incorrect input.
 */
int main() {
    ...
}
```


Rest of Main.cpp

```
// Catch any exceptions that might be thrown.

try {

    // Read the tree and print it.

    const Tree tree(cin);
    tree.print(cout);
} catch (const exception& e) {

    // If an exception was thrown, print what it says.

    cerr << e.what() << endl;

    // Return an error status code.

    return 1;
}

// Return normal status.

return 0;
}
```

Implementation of `main` function

- Begin with `includes` and `using` statements
- Should always have a `using` statement for the package
- In doxygen the `@` and `\` characters are interchangeable
- The `try` block is used for catching exceptions
- One should always `catch` using a constant reference
- The `what ()` method returns the explanation of the exception



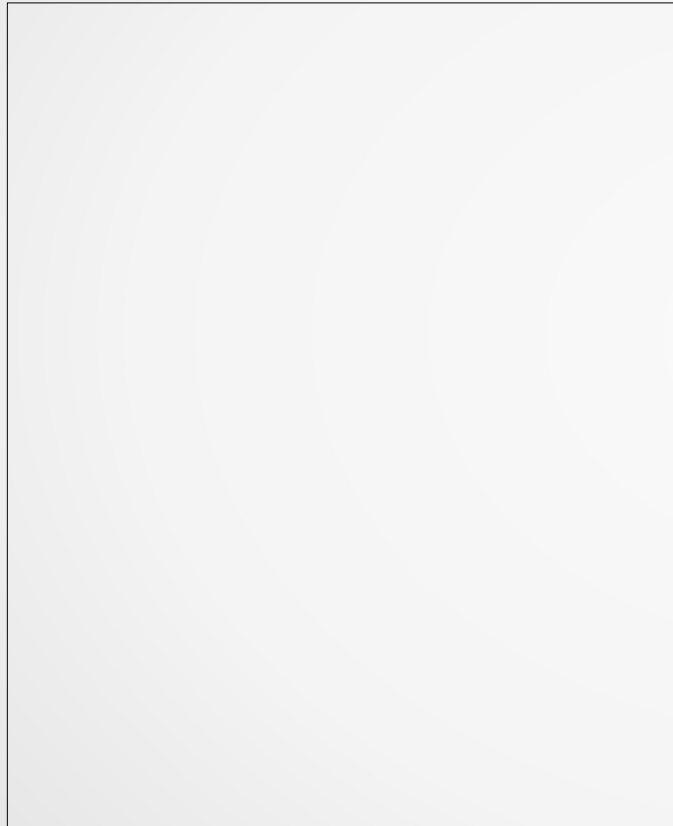
The Call Stack and Exceptions

Procedures

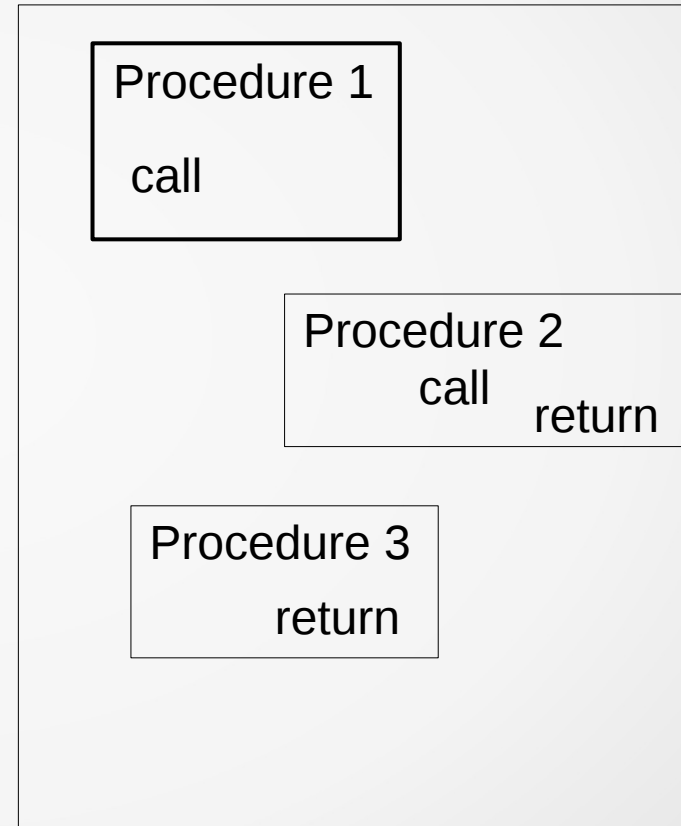
- Procedures are fundamental to software
- A procedure is a block of memory containing executable instructions
- The following are the steps for a procedure call
 - Evaluate the parameters
 - The results of the evaluations are the *arguments*
 - Allocate memory to store the arguments
 - Allocate memory to store the return address
 - Jump to the procedure address
 - Execute the procedure code using the arguments
 - Jump to the return address
 - Deallocate the argument and return address memory

Procedures

Memory area for data



Memory area for executable code



Procedures

Memory area for data

arguments
for first call

Memory area for executable code

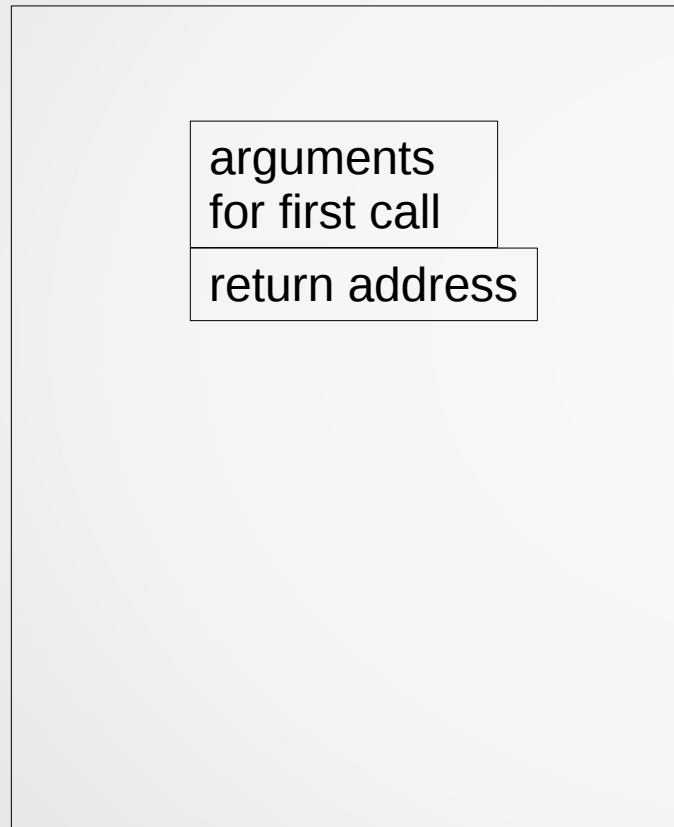
Procedure 1
call

Procedure 2
call return

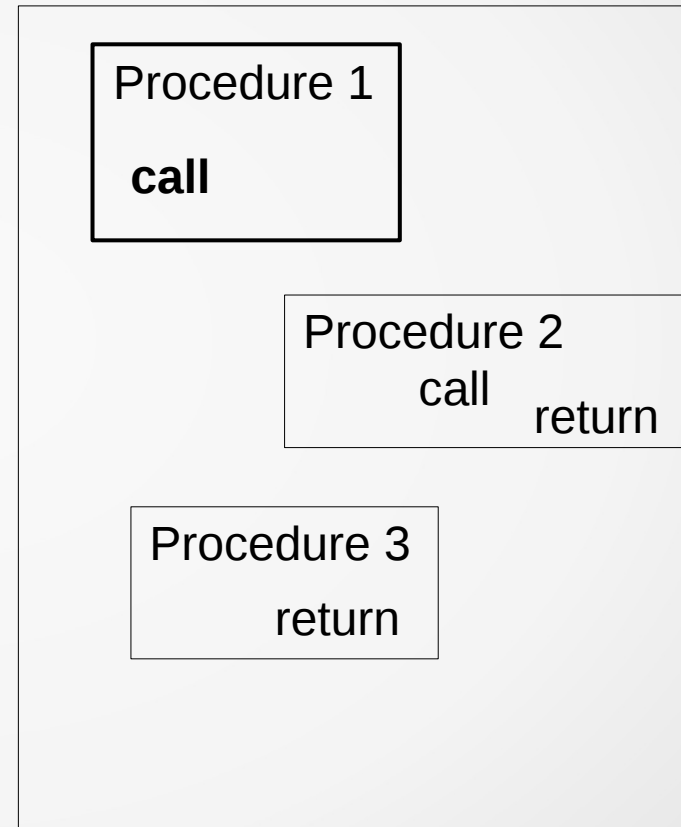
Procedure 3
return

Procedures

Memory area for data

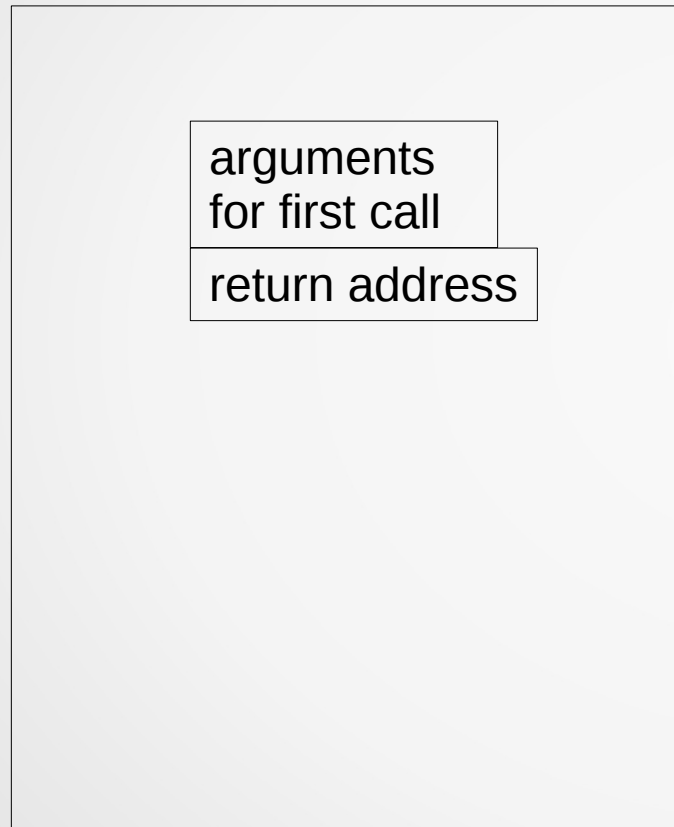


Memory area for executable code

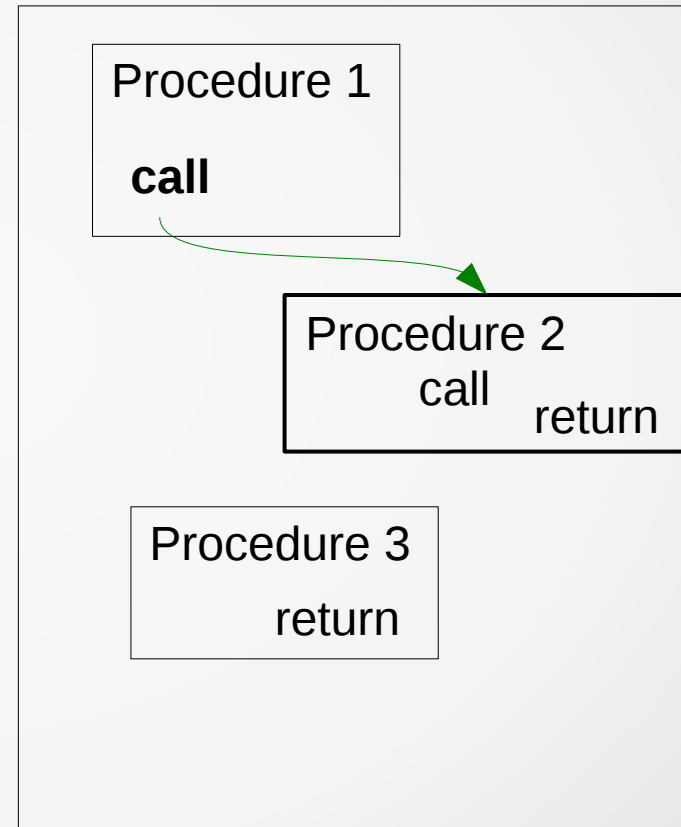


Procedures

Memory area for data

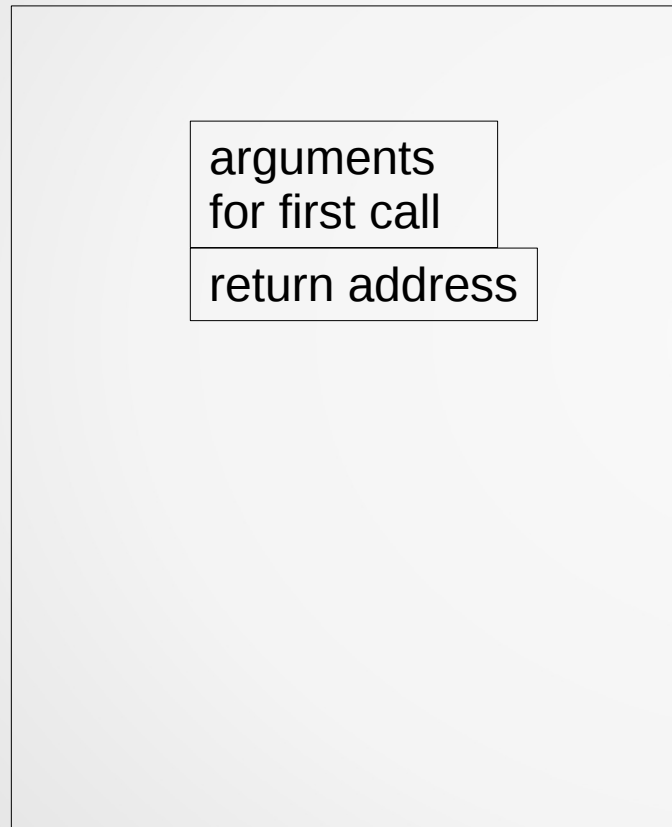


Memory area for executable code

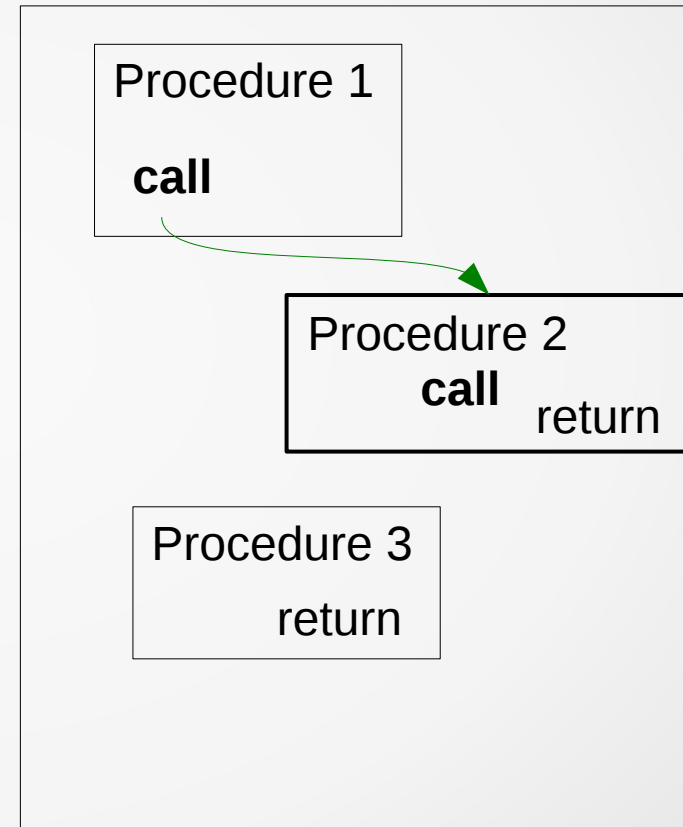


Procedures

Memory area for data

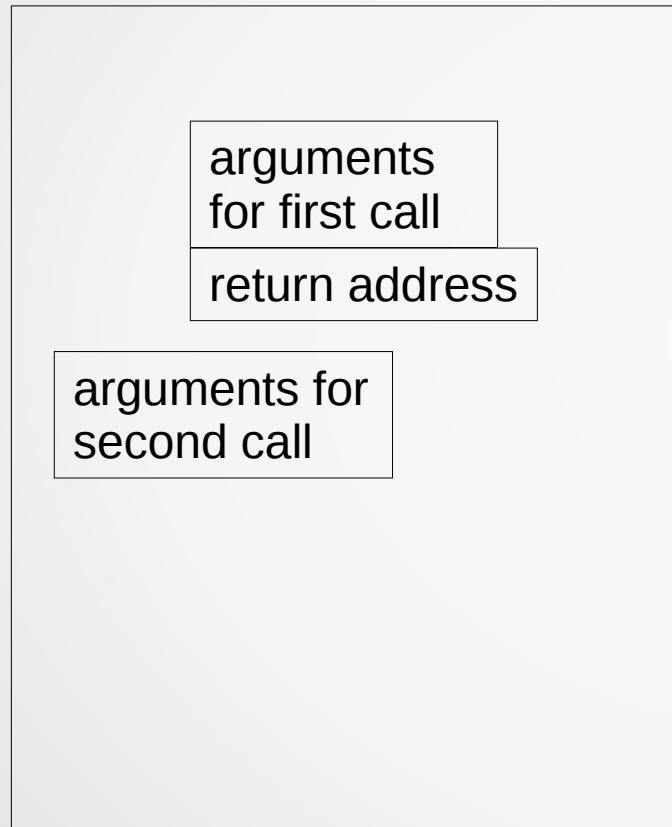


Memory area for executable code

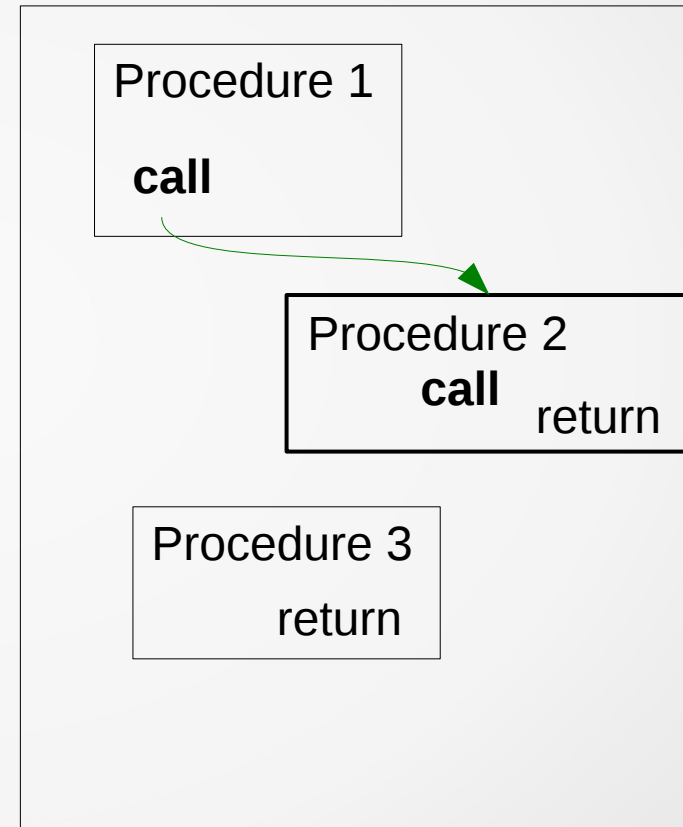


Procedures

Memory area for data

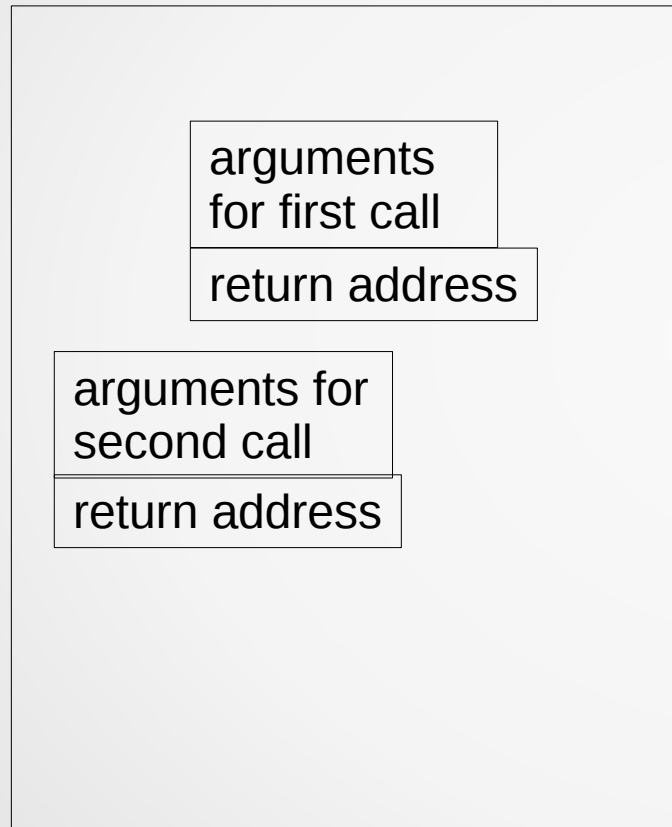


Memory area for executable code

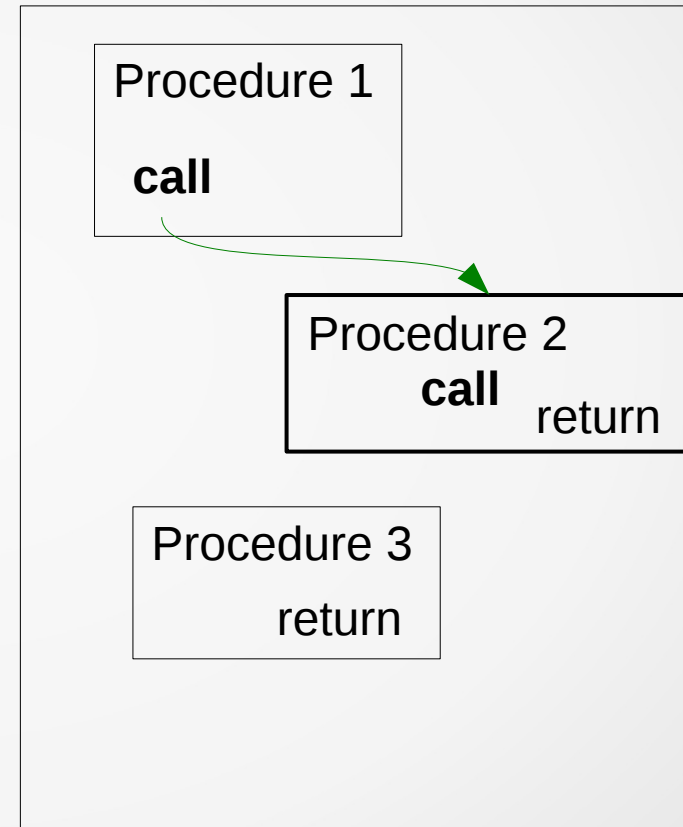


Procedures

Memory area for data

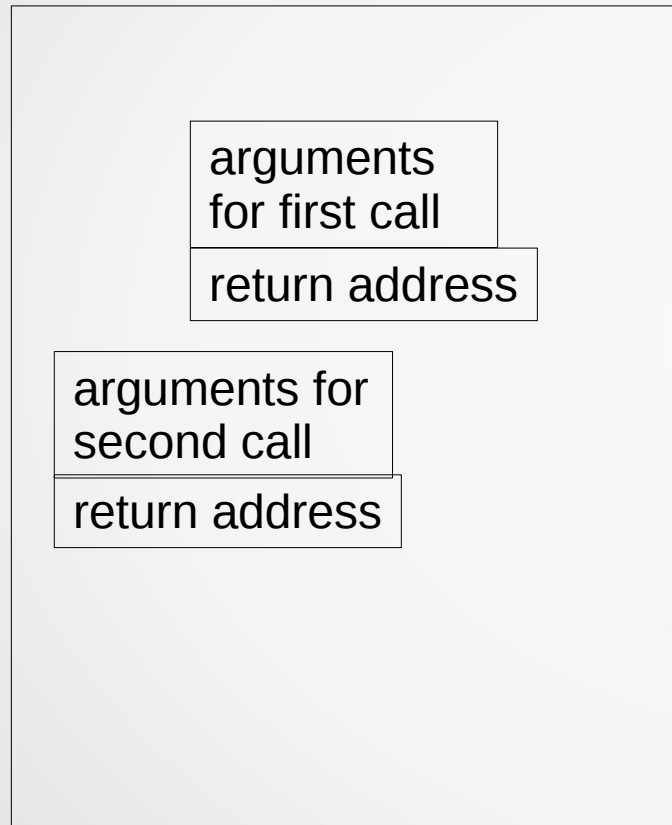


Memory area for executable code

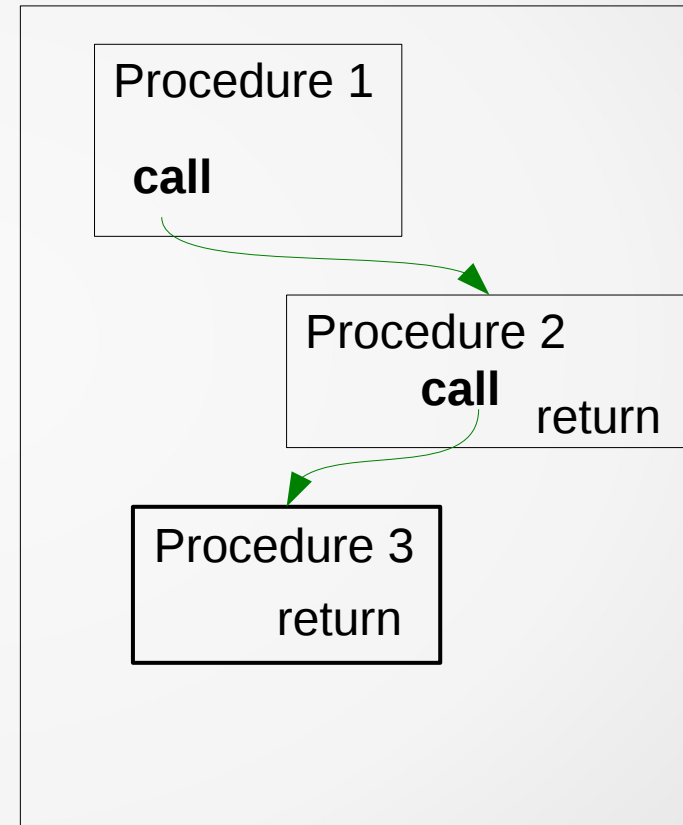


Procedures

Memory area for data

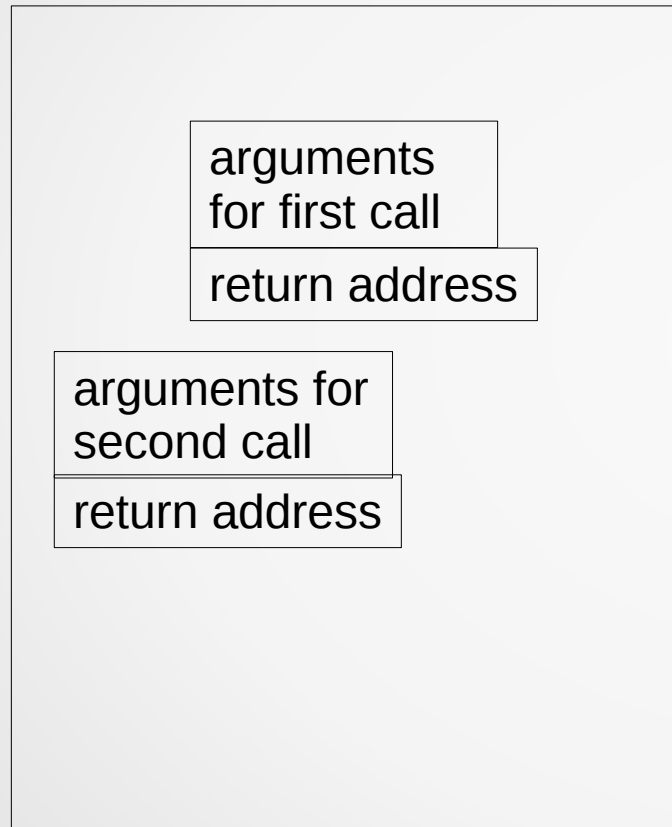


Memory area for executable code

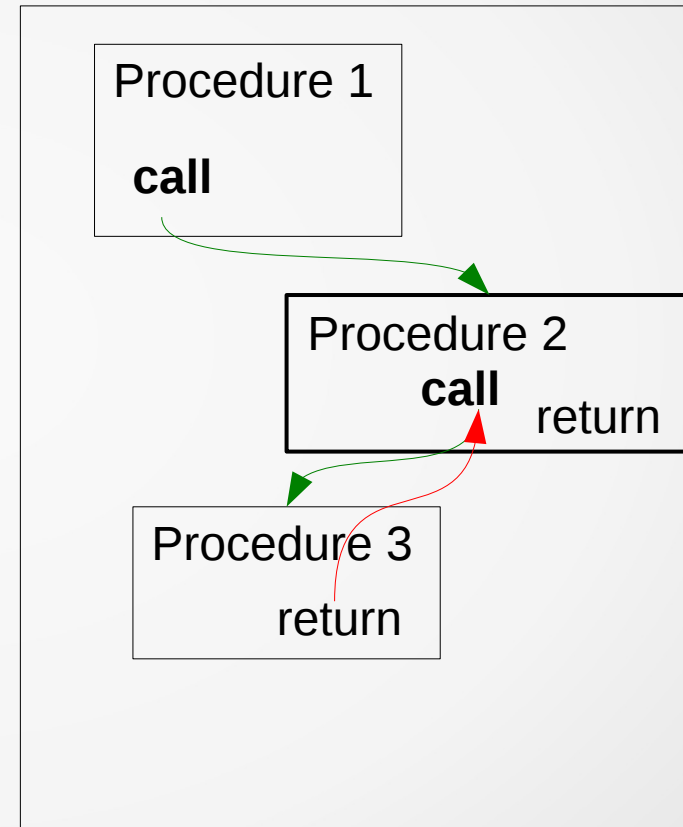


Procedures

Memory area for data

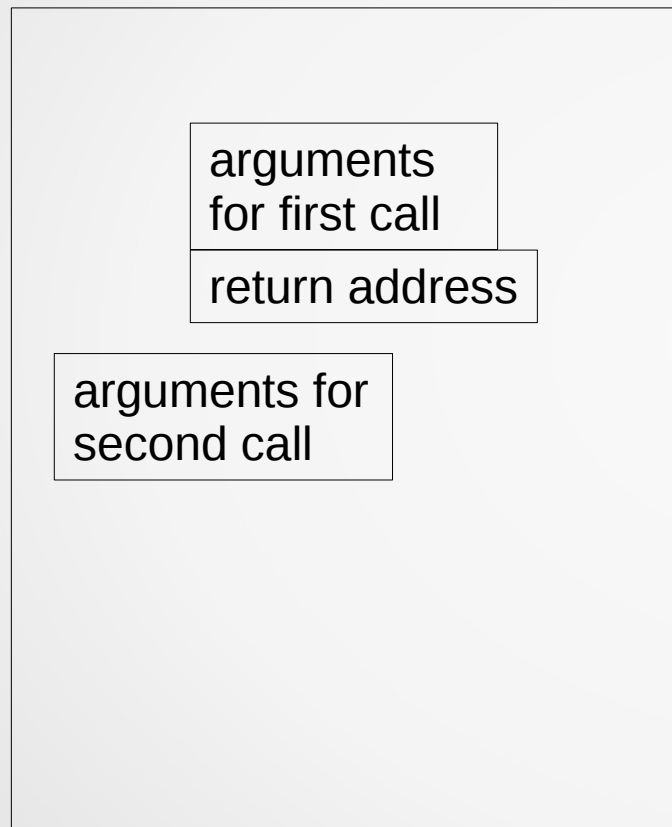


Memory area for executable code

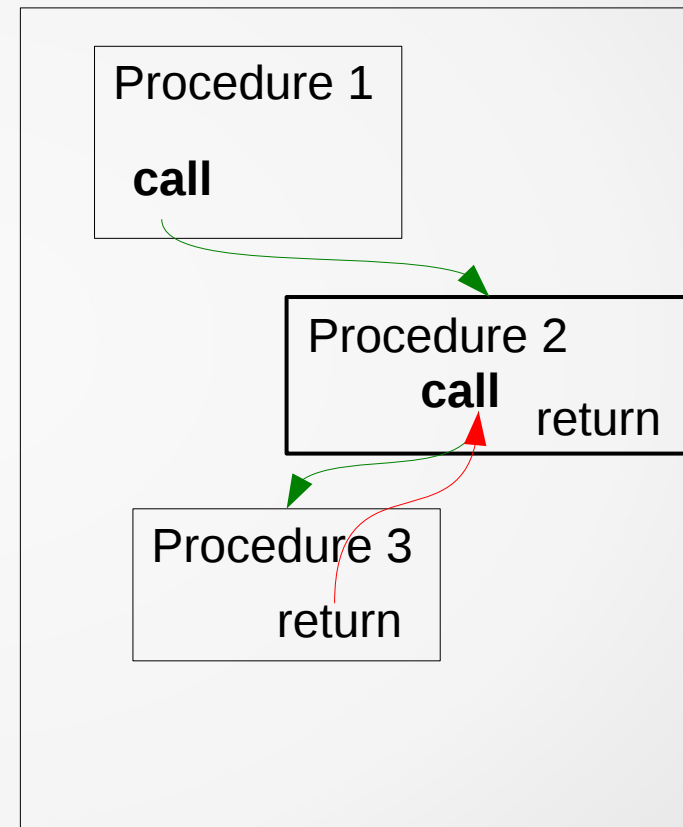


Procedures

Memory area for data

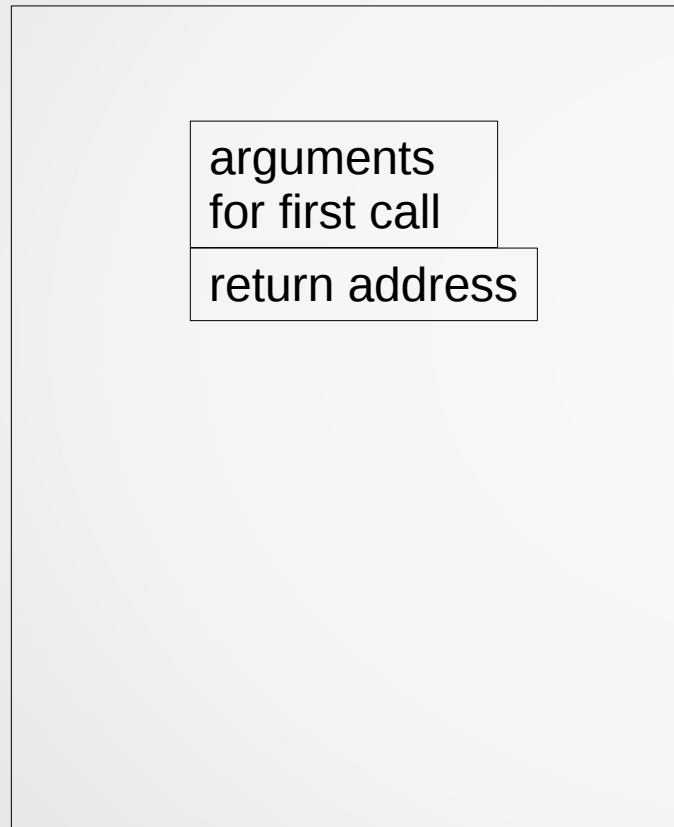


Memory area for executable code

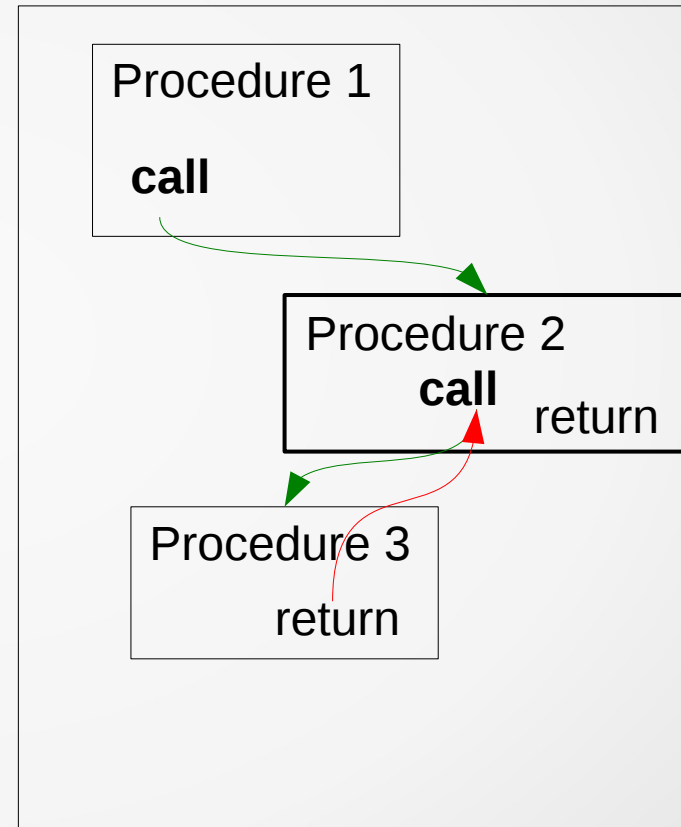


Procedures

Memory area for data

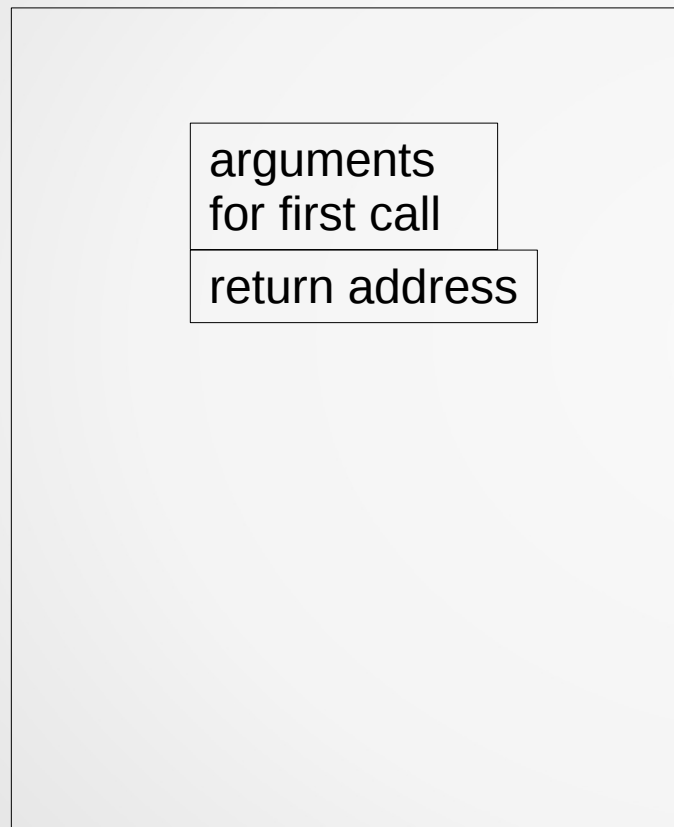


Memory area for executable code

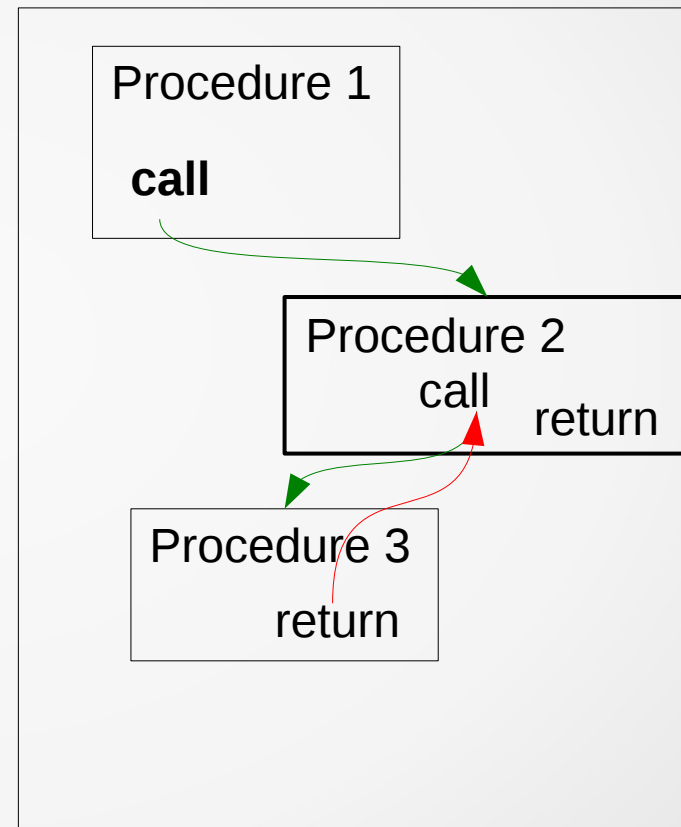


Procedures

Memory area for data

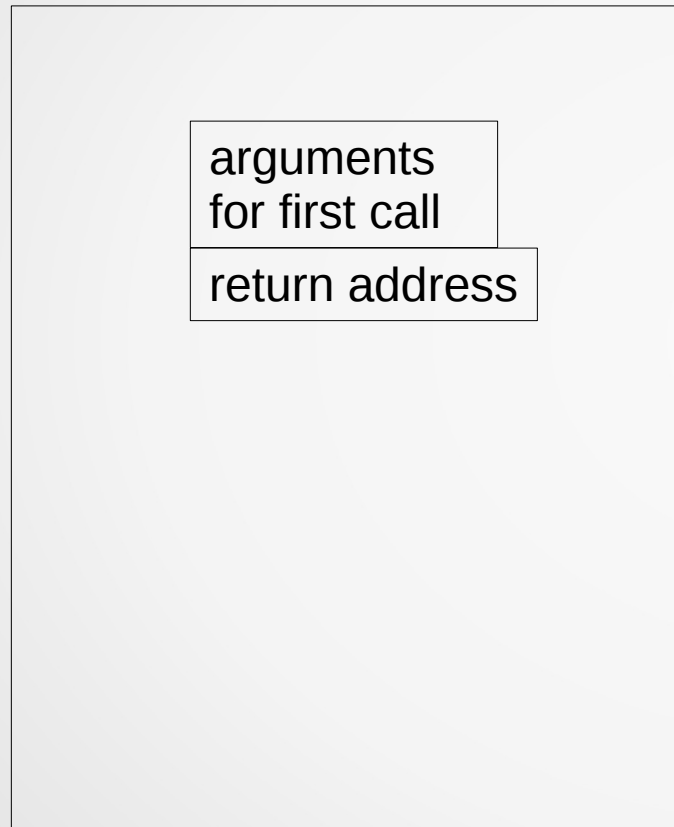


Memory area for executable code

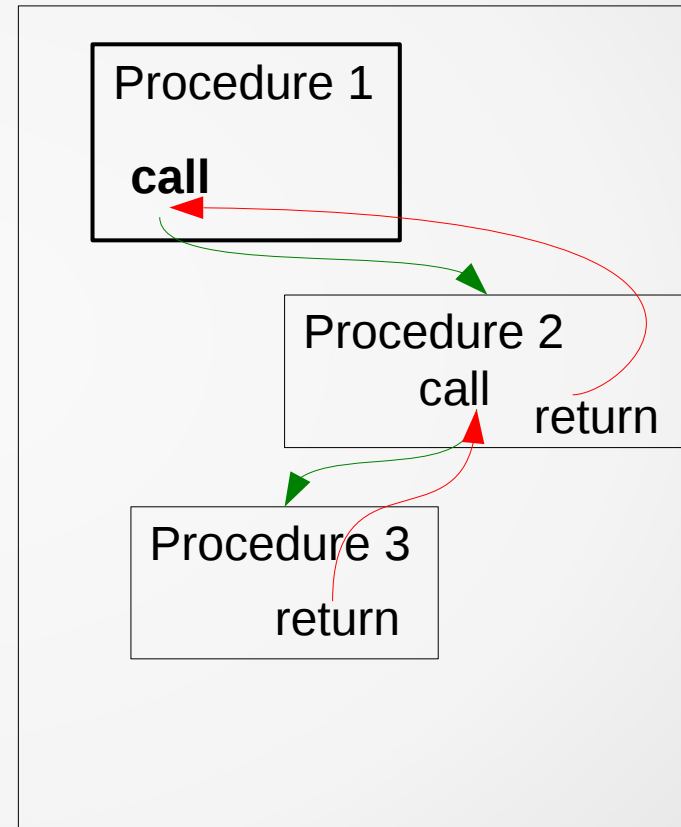


Procedures

Memory area for data

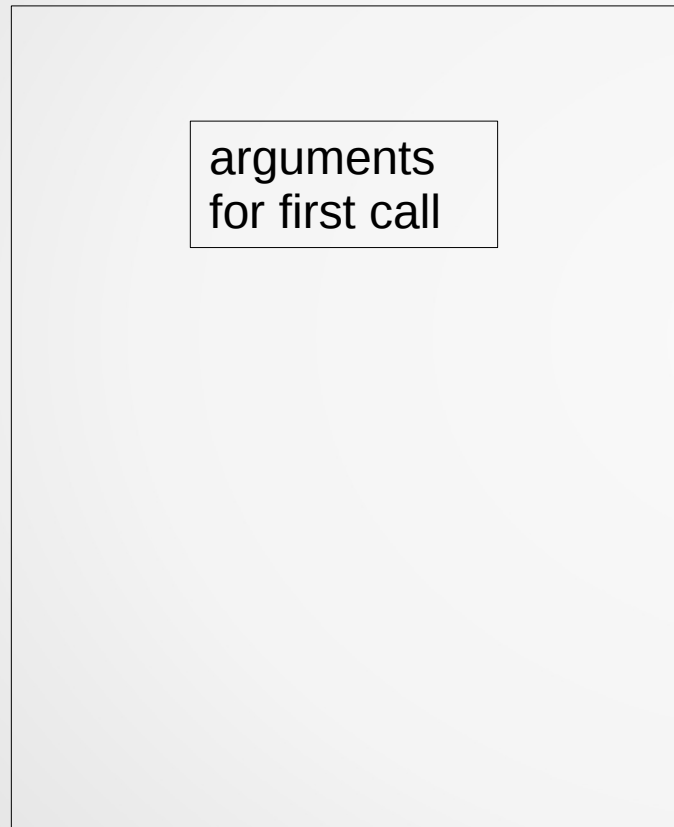


Memory area for executable code

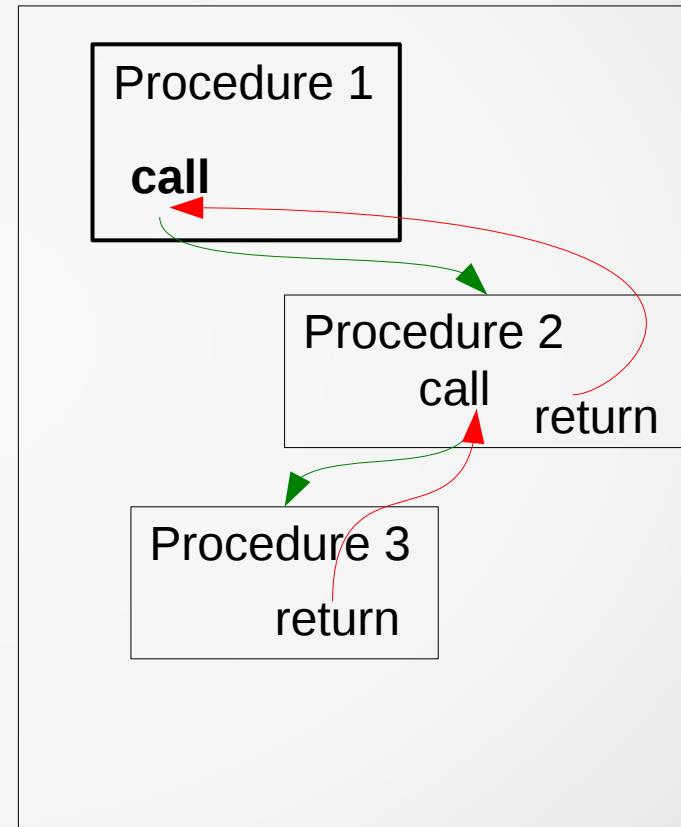


Procedures

Memory area for data

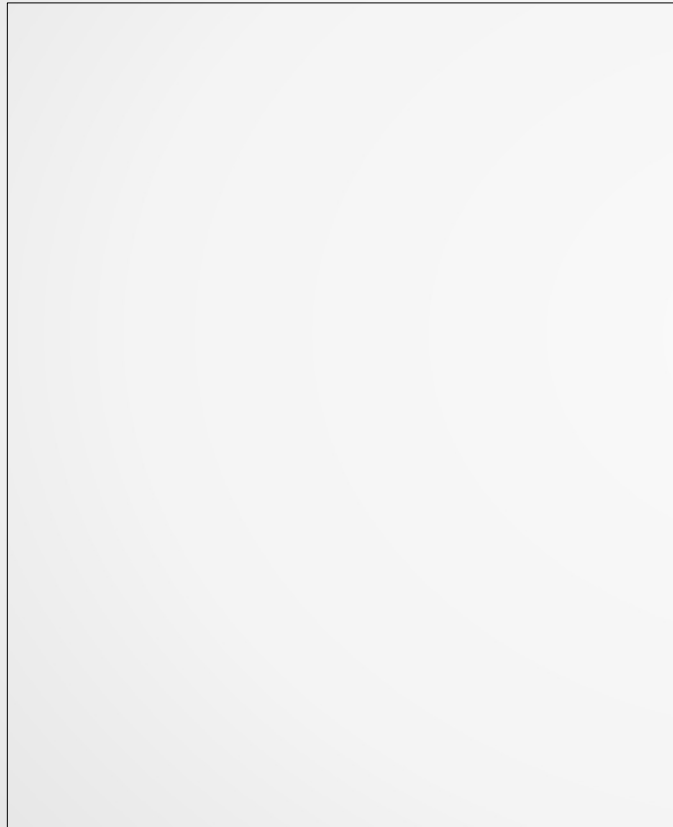


Memory area for executable code

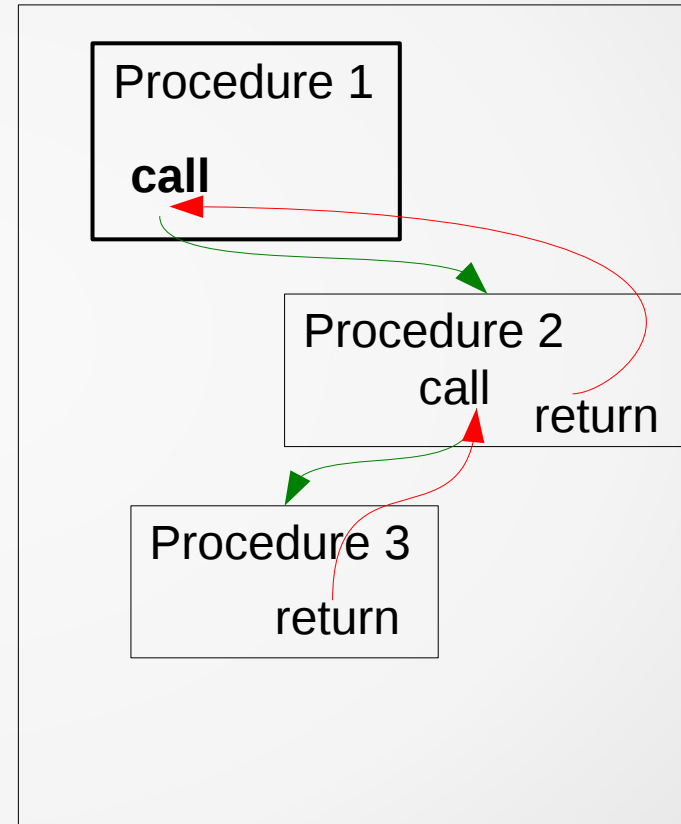


Procedures

Memory area for data

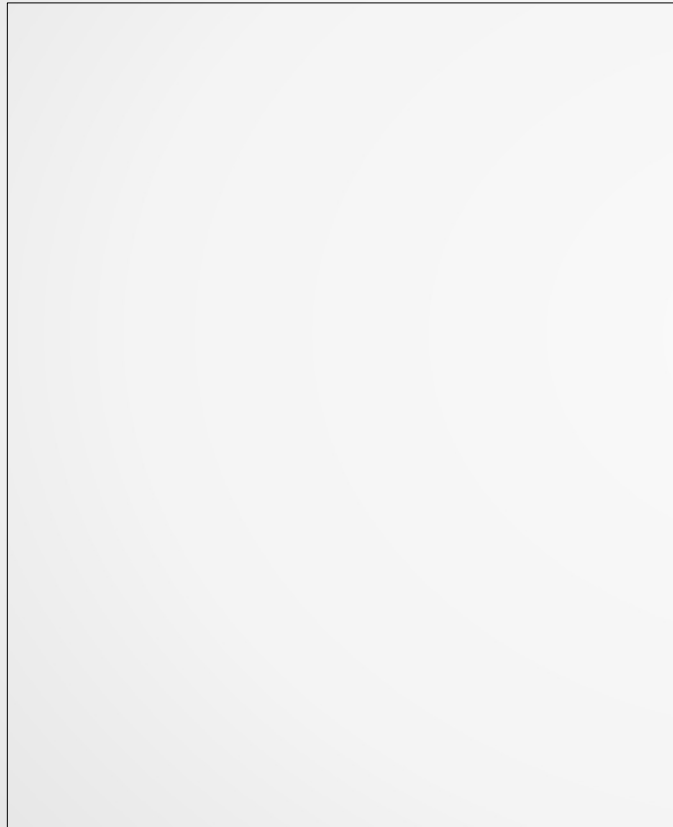


Memory area for executable code

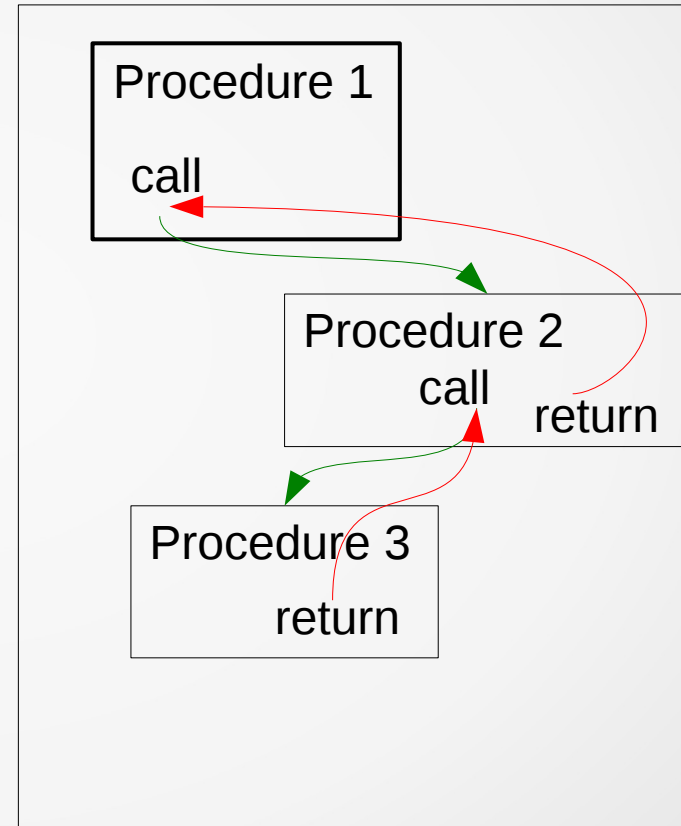


Procedures

Memory area for data



Memory area for executable code



Parameters

- What are the parameters for a method
- Consider the method of the University class
 Student getStudent(string name)
 that gets the student record for a student with the specified name
- What are the parameters?
 - The name is a parameter, of course
 - The University object is also a parameter
 - The return value is yet another parameter
- All of these parameters are passed to the getStudent method
 - The name is copied
 - A reference to the University object is passed
 - A location where the return value is to be placed must also be passed
- Parameter passing mechanisms apply to all of these parameters

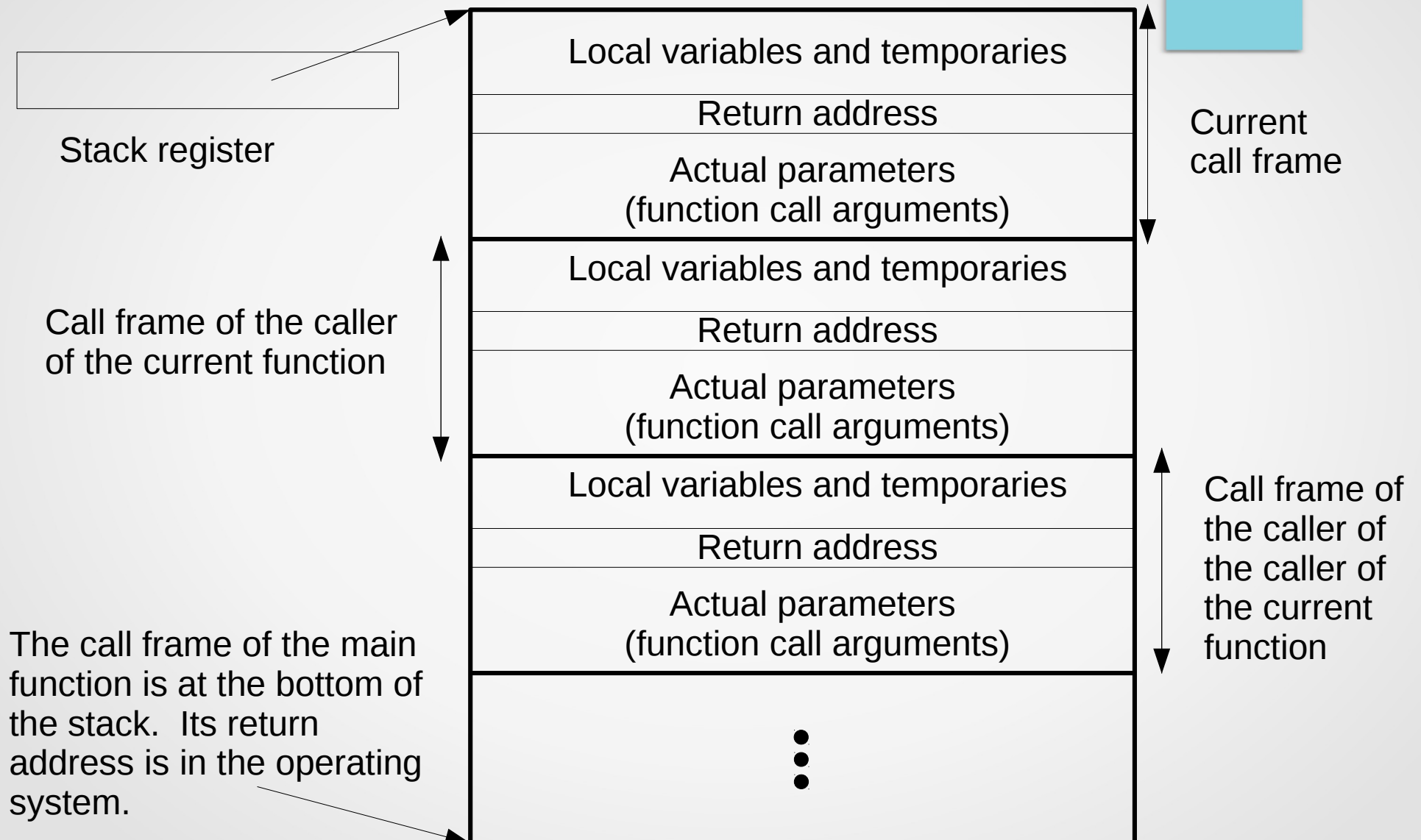
Invoking methods

- Non-static methods are invoked by specifying an object followed by the method name and arguments
 - In `Main.cpp`, the `print` method is invoked with `tree.print()`;
- Within the non-static methods of a class, method calls and uses of data members do not need to specify an object
 - There is always an instance of the class that is the context of non-static methods
 - The instance is called `this`.
 - In `Tree.cpp`,
for `(const Tree& tree : children_)`
could have been written
for `(const Tree& tree : this.children_)`
- Static methods do not specify an object
 - There is no instance of the class that is the context of a static method.

Stack Frames

- When a function is invoked, memory is allocated on the *execution stack* also called the *call stack* (and other names)
 - Allocation is performed by changing the value of the *stack pointer*
 - Most modern machine architectures have a dedicated hardware register for the stack pointer
- The allocated memory is the *stack frame* of the invocation
- All arguments, local variables, temporary variables, and the return address are allocated space on the stack

Execution Stack



Exceptions and Returns

- Syntax: `return ...;`
 - stops execution of your program
 - unwinds exactly one stack frame
 - resumes execution where the function was called
 - uses the returned object as the value of the function call
 - the returned object must have the declared type of the function
- Syntax: `throw ...;`
 - stops execution of your program
 - unwinds zero or more stack frames
 - resumes execution at the first matching `catch` statement
 - calls the `catch` statement with the object that was thrown as the argument
 - any object can be thrown but can only be caught by a `catch` with a compatible type



Parameter Passing

Parameters

- Parameters pass information to and from procedures
- C++ supports many more parameter passing mechanisms than most other languages
 - Gives programmer much more control
 - Adds complexity
- C++ distinguishes different kinds of value
 - Values can have names (variables) in a program
 - Values can be unnamed
 - Temporaries generated within an expression

C++ Value Categories

- Each expression in a program has a value category.
- An *lvalue* is a value with a name.
 - One can take the address of an *lvalue*
 - If an *lvalue* is not `const` then one can modify it
- An *rvalue* is a temporary value without a name.
 - One cannot take the address of an *rvalue*
 - Now subcategorized into *prvalue* and *xvalue*.
- The names come from assignment statements. An *lvalue* can be on the left, an *rvalue* can only be on the right.
- The actual definitions of these categories are complex because of the many contexts that are possible

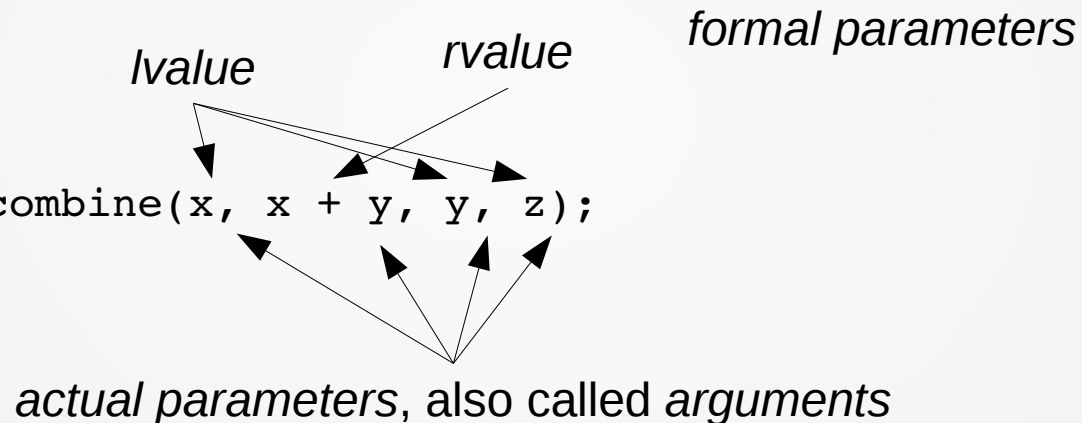
Declaring and calling a function

Function declaration:

```
double combine(double p1, const double p2, double& p3, const double& p4);
```

Function invocation:

```
double x = 5.5;  
double y = 6.6;  
double z = 7.7;  
double result = combine(x, x + y, y, z);
```



Parameter Passing Mechanisms

- Call by value
 - Copies the value
 - The only mechanism in Java and C
 - Some people refer to a pointer parameter as call by reference, but this is incorrect. Passing a pointer is a form of call by value.
- Call by reference
 - No copying is done
 - The function parameter is an alias
 - The argument must be an lvalue
- Call by value-result
 - Copies as in call by value, then copies the final value back when the function returns.
 - Not supported by C++

Parameter passing examples

- The depth parameter is *passed by value*
 - The value is copied
 - Built-in types are normally passed by value
 - Large objects should not be passed by value because of the copying overhead
- The in parameter is *passed by reference*
 - The in parameter is an alias for the argument
 - Any methods invoked on in will affect the argument exactly as if the argument was used
- The label parameter is *passed by constant reference*
 - Also an alias
 - Cannot modify it
 - Used for passing large objects without the overhead of copying

```
Tree::Tree(const string& label, const int depth,
           istream& in)
    : label_(label), depth_(depth) {

    // The list of children must begin with a left bracket.

    const string bracket = readWord(in);
    if (bracket != "[") {
        throw domain_error("Incorrect tree format: "
                           "list does not start with a left bracket");
    }

    // Read the subtrees until a right bracket.

    for(;;) {
        const string word = readWord(in);

        // If the word is a right bracket,
        // then the tree is complete.

        if (word == "]") {
            return;
        }

        // Otherwise, construct a tree
        // and add it to the list of children.

        children_.push_back(Tree(word, depth_ + 1, in));
    }
}
```

Parameter passing arguments

- The label parameter
 - The argument may be an lvalue or rvalue
- The depth parameter
 - The argument may be an lvalue or rvalue
- The in parameter
 - The argument must be an lvalue
- Is it possible to pass an rvalue by reference?
 - Yes! New feature of C++11

Declaration

```
Tree(const string& label,  
      const int depth,  
      istream& in);
```

Invocation

```
Tree(readWord(in), 0, in)
```


Pointers

- These will be covered later
- For the moment you just need to know that you *must* not declare any pointer variables.
- Pointers and pointer arithmetic are regarded as dangerous.
- Unfortunately, they are often used by libraries.
- The recommended way to deal with pointer variables is to wrap them in a safe construct
 - A user-defined wrapper class
 - A smart pointer

Coding Style Requirements

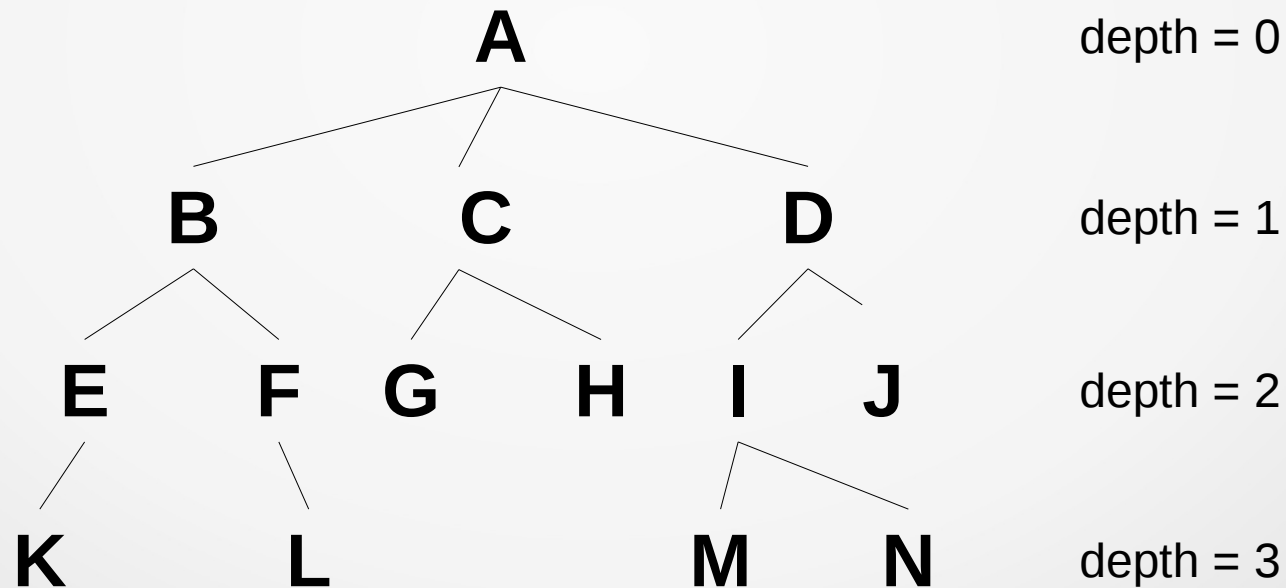
- Writing a reference (or later a pointer) type:
 - Write
`const double&`
NOT
`const double &`
 - No space between the type name and the ampersand
 - The ampersand logically belongs to the type, not what follows it.
- Never declare a variable to be a `register` variable.
 - Makes your code less portable
 - You are unlikely to be able to optimize better than the compiler already does.



The tree2 package

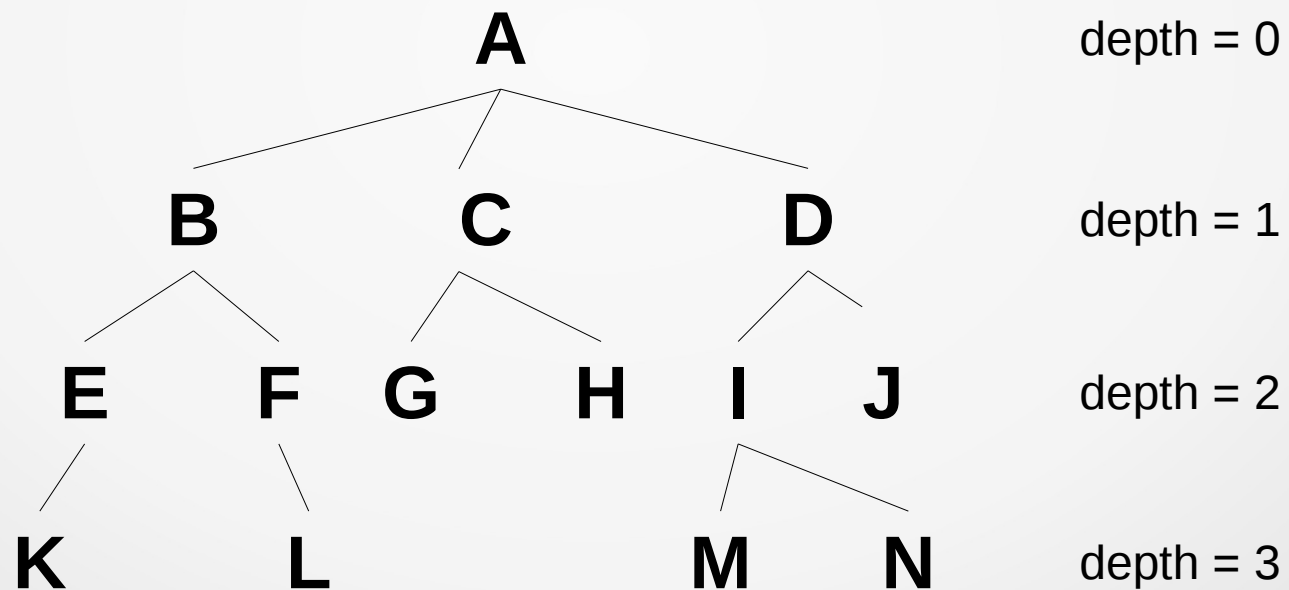
The tree2 package

- Each node of the tree has a label and identifier
- Read a tree from the standard input
- Sort the tree in reverse alphabetical order
- Print the tree to the standard output, showing identifiers



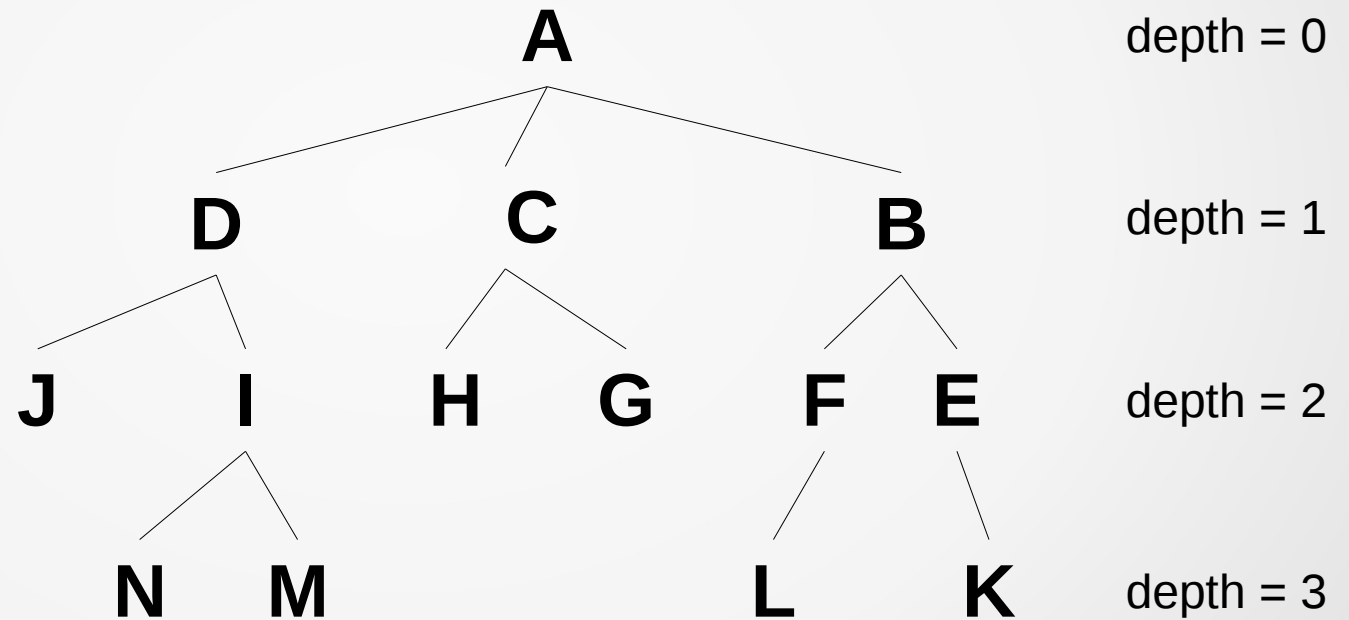
Sample Input

A[B[E[K[]]F[L[]]]C[G[]H[]]D[I[M[]N[]]J[]]



Sample Output

```
A 1 [  
  D 10 [  
    J 14 [ ]  
    I 11 [  
      N 13 [ ]  
      M 12 [ ]  
    ]  
  ]  
C 7 [  
  H 9 [ ]  
  G 8 [ ]  
]  
B 2 [  
  F 5 [  
    L 6 [ ]  
  ]  
  E 3 [  
    K 4 [ ]  
  ]  
]  
]
```





Header File: Class Interface

Beginning of Tree.h

```
#ifndef TREE2_TREE_H
#define TREE2_TREE_H
```

```
#include <iostream>
#include <vector>
```

```
namespace tree2 {
```

```
/**
```

```
 * A sortable tree class. A tree has a label and
 * identifier and may have a list of child trees. The
 * tree can only be constructed by reading it from an
 * input stream. The sort method sorts the children of
 * every subtree by their labels.
```

```
 *
```

```
 * @author Ken Baclawski
```

```
 */
```

```
class Tree {
public:
```

The namespace is now tree2.
The documentation block explains
what the class supports.

Public Interface of the Tree class

```
public:
    /**
     * Construct a tree from an input stream. An
     * exception is thrown if the input is not a valid
     * tree.
     */
    Tree(** The input stream that is being read. */
         std::istream& in);

    /**
     * Print this tree to an output stream. The tree is
     * traversed and the labels are printed in
     * depth-first order. The output of this method can
     * be used as input to the constructor.
     */
    void print(** The output stream the tree will
               be printed on. */ std::ostream& out) const;

    /**
     * Sort the tree. For every subtree the children are
     * sorted reverse alphabetically by their labels.
     */
    void sort();
```

There is one
new method in
the public
interface: sort

Data Members of the Tree class

- The `label_` and `depth_` fields are no longer `const`
 - The sort algorithm will modify them
 - It swaps pairs of objects in place by modifying their fields
- The `nextId_` field is static
 - There is only one instance of this field for the whole program
 - Ordinary (non-static) fields are allocated for every instance of the class
- Static fields cannot be initialized in the header file
 - Must be initialized in the source file

```
private:
    /** The label of this tree. */
    std::string label_;
    /** The depth of this tree. */
    int depth_ = 0;
    /** The subtrees of this tree. */
    std::vector<Tree> children_;
    /** The identifier of the next tree. */
    static int nextId_;
    /** The identifier of the tree. */
    int id_ = 0;
```

Private Methods of the Tree class

- These are the same as in tree1

```
/**
 * Recursively parse a tree from
 * an input stream. This method throws
 * an exception if the end of data has
 * been reached without finding a word.
 */
Tree(** The label of the tree. */
    const std::string& label,
    /** The depth of the tree. */ int depth,
    /** The input stream for reading
     * the subtrees. */ std::istream& in);

/**
 * Read one word from the input stream.
 * This method throws an exception if
 * the end of data has been reached
 * without finding a word.
 *
 * @return The word that was extracted
 * from the input stream.
 */
static std::string readWord(
    /** The input stream. */ std::istream& in);
};

}

#endif
```



Source File: Class Implementation

Beginning of Tree.cpp

Same as for tree1 except

- The algorithm library is included to get the sort function
- The namespace is tree2
- The static field `nextId_` is initialized

```
#include <vector>
#include <iostream>
#include <stdexcept>
#include <algorithm>
#include "Tree.h"
using namespace std;
using namespace tree2;

int Tree::nextId_ = 0;
```

Rest of Tree.cpp

- Only the differences with tree1 will be discussed
- The readWord and the public constructor are the same

```
Tree::Tree(const string& label,
           const int depth,
           istream& in)
    : label_(label), depth_(depth) {
    ...
}

Tree::Tree(istream& in)
    : Tree(readWord(in), 0, in) {}

string Tree::readWord(istream& in) {
    ...
}

void Tree::print(ostream& out) const {
    ...
}

void Tree::sort() {
    ...
}
```

Private constructor implementation

- The only difference is that the `id_` field must be initialized
- It is initialized to the value of the static variable `nextId_`
- Then `nextId_` is incremented
- The rest of the constructor is the same as in `tree1`

```
Tree::Tree(const string& label,
           const int depth,
           istream& in)
    : label_(label), depth_(depth),
      id_(nextId_++) {
    ...
}
```

Implementation of the `print` method

- The only difference is printing the identifier

```
void Tree::print(ostream& out) const {  
    ...  
    out << indentation << label_ << " " << id_ << " [";  
    ...  
}
```


The sort method implementation

```
void Tree::sort() {  
  
    // Sort the vector of children alphabetically by label.  
    // The namespace of sort must be explicitly specified  
    // because this method is also named sort.  
  
    std::sort(children_.begin(), children_.end(),  
               [](const Tree& a, const Tree& b)  
               { return a.label_ > b.label_; });  
  
    // Sort each of the subtrees.  
  
    for (Tree& child : children_) {  
        child.sort();  
    }  
}
```

The sort method implementation

- The `tree2::Tree::sort` method uses the `std::sort` method
 - The names must be disambiguated explicitly
- The third parameter of `std::sort` specifies how to compare two `Tree` objects
 - This is a *lambda expression*
- The for loop uses `Tree&` rather than `Tree` as the type of the child variable
 - The `Tree::sort` method must change the tree in the `children_` vector, not a copy of it

```
void Tree::sort() {  
  
    // Sort the vector of children alphabetically by label.  
    // The namespace of sort must be explicitly specified  
    // because this method is also named sort.  
  
    std::sort(children_.begin(), children_.end(),  
              [](const Tree& a, const Tree& b)  
                { return a.label_ > b.label_; });  
  
    // Sort each of the subtrees.  
  
    for (Tree& child : children_) {  
        child.sort();  
    }  
}
```

Lambda Expressions

- This is an anonymous function
- The parameters use constant references so that the trees will not be copied
- The type of the lambda expression is determined by type inference
 - It is the type of the return value
 - One can have only one return statement

```
[ ](const Tree& a, const Tree& b)  
    { return a.label_ > b.label_; }
```

- Lambda expressions will be covered in more detail later
- For now, we will only use the simple form shown in this example

Next Class

- Iterators
- UML Diagrams