G205
Fundamentals of Computer Engineering

CLASS 3, Monday Sept. 15 2003
Stefano Basagni
Fall 2003
M-W, 9:50am-11:30am, 410 Ell
Function Calls

- A function is called via its name
- *Actual parameters* are passed for the *formal parameters*
- *An environment* is created in memory that hosts parameters variables local to the function
- The environment is cancelled upon return
A function can call itself (except main) = RECURSION

Recursion allows natural implementation of recursive definitions. For each n >= 0:

\[ n! = n \times (n-1)! \quad 0! = 1! = 1 \]

```cpp
int recFact(int n) {
    if (n < 2) return 1;
    else
        return n * recFact(n - 1);
}
```
C++ Basics: Recursion, 2

Alternative for ITERATIVE definition:

\[ n! = n \times (n-1) \times \ldots \times 2, \quad n > 2, \quad 0! = 1! = 1 \]

```c
int factorial(int n) {
    int fact = 1;
    for (int c = n; c > 1; c --)
        fact *= c;
    return fact;
}
```
Rec vs. Iter: Running Times

- Recursive factorial:
  - $T_r(n) = c$ if $n < 2$
  - $T_r(n) = T_r(n-1) + c$ if $n > 1$  (*

- (*) is called a *recurrence relation*

- It is solved by repeated substitutions
Repeated Substitutions

Assume \( n > 2 \):

\[
T_r(n) = T_r(n-1) + c = T_r(n-2) + c + c = T_r(n-3) + 3c = \ldots = T_r(n-k) + kc
\]

Termination: \( n-k=1 \) \( \Rightarrow \) \( k=n-1 \)

\[
T_r(n) = T_r(n-k) + kc = T_r(n-n+1) + (n-1)c = T_r(1) + (n-1)c = c + (n-1)c = cn \text{ in } O(n)
\]
Recursion vs. Iteration

- The running time $T_i(n)$ of the iterative factorial is in $O(n)$ (induced by the for loop)
- factorial and recFact have the same running time, BUT
- factorial is based on a simple repetition structure
- recFact uses repeatedly the function call mechanism ("stack of environments")
- Time and space consuming
Other C++ Function Features

- Empty parameter list
- \textit{inline} functions
- Parameter passing
  - Pass-by-value
  - Pass-by-reference
- Default arguments
- Function overloading
- Function templates
Arrays, 1

- Data structures of related (same name) data items of the *same type*
- *Static* data structures (remain the same size)
- Elements of an array are referred to by their position (index): $A[0...n]$ has $n+1$ items: $A[0], A[1], ..., A[n]$
Arrays, 2

- **Declaration:**
  
  ```
  type arrayName[ arraySize ]
  ```

  - int c[12];

- **Passing arrays to functions**
  
  ```
  InsertionSort(c,12); // function call
  ```

- **Always passed by reference**

- **Multiple-Subscripted Arrays**
  
  ```
  int M[row][col]
  ```

  ```
  void testFun( int N[][13] );
  ```
Pointers, 1

- A pointer is a memory address ...
- A pointer variable contain a memory address as its value, a non-pointer variable contains a value
- Declaration: `double *ptr;`
  - `ptr` is the address of a location that contains a value of type double
Pointers, 2

- Init: 0 or NULL or an address
- `int *yPtr = NULL;` ➔ yPtr points to nothing
- `int y = 5;` 
  `yPtr = &y;`
- Dereferencing operator
  - `*yPtr` is “usable” as `y`
Structures, 1

Aggregate data types, from C

```c
struct Time {
    int hour;
    int minute;
    int second;
};
```

Declaration:
- `Time ts; // A variable of type Time`
- `Time tsa[15]; // An array of structures`
Structures, 2

- Accessing structure *members*:
  - Dot (.) operator: ts.hour;
  - Arrow (->) operator: access via a pointer
    - Time *timePtr = &ts;
    - timePtr -> hour; (= (*timePtr).hour;)

- Defines a data type, not the operations
- Not the best for *Abstract Data Types*
Abstract Data Types

A type (set of values) whose internal form is hidden behind a set of access functions. Objects of the type are created and inspected only by calls to the access functions. This allows the implementation of the type to be changed without requiring any changes outside the module in which it is defined.
Implementing ADTs with a CLASS

- Central concept to OOP: implements objects
- Data+Operations, implements ADTs
- Classes are user-defined types which define data AND the set of (all and only) functions that manipulates the data
Class Interface

class Time {
    public:
        Time();
        void setTime( int, int, int );
    private:
        int hour;
        int minute;
        int second;
};
Class Implementation

Time::Time() { // Constructor
    hour = minute = second = 0;
}

void Time::setTime(int h, int m, int s) {
    hour = (h>=0 && h < 24) ? h : 0;
    minute = (m>=0 && h < 60) ? m : 0;
    second = (s>=0 && h < 60) ? s : 0;
}
Class Use

Legal and illegal operations:

```cpp
int main() {
    Time t; // Instantiate and initialize an object of class time
    t.setTime(13, 27, 6); // Change time
    cout << t.hour; // Attempt to access private member
    ...
}
```
Other Class Features

- Access functions and utility functions
- User-defined constructors
  - Default arguments
- Destructors
- Friend functions
- Const objects and const functions
- Operator overloading
Assignments

- Deitel & Deitel book, chapters 5 and 6
- Updated information on the class web page:
  www.ece.neu.edu/courses/eceg205/2003fa