Recursion with C++

Plan for today

- We will talk about:
  - Definition of recursive function
  - Call stack with function activation records
  - Examples

Recursive Functions

[Def] *Recursive function* is a function that calls itself, either directly or indirectly.

The C++ standard document indicates that `main` should not be called within a program or recursively.

Recursive Functions

[Def] *Recursive function* is a function that calls itself, either directly or indirectly.

The C++ standard document indicates that `main` should not be called within a program or recursively.

Recursion concepts:

- every recursive function should have base case(s)
- every recursive call/recursion step of a function should be to “solve a smaller problem”, which should eventually *converge* to a base case.

Recursive Functions

[Def] *Recursive function* is a function that calls itself, either directly or indirectly.

The C++ standard document indicates that `main` should not be called within a program or recursively.

Recursion concepts:

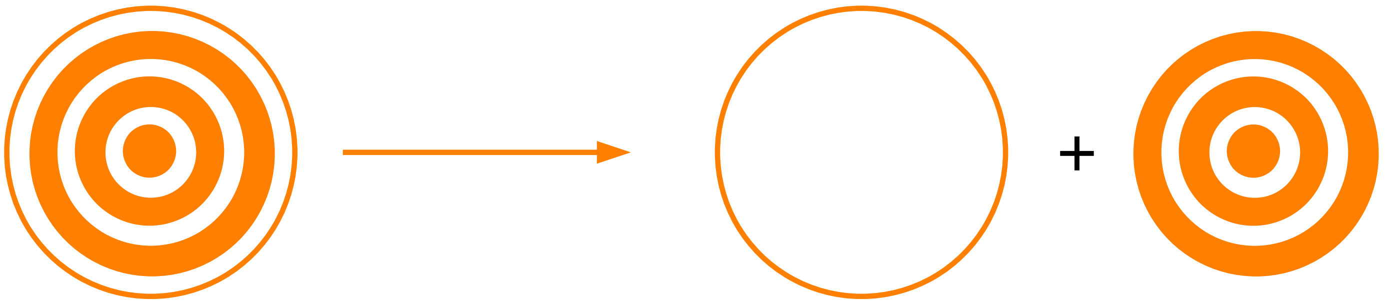
- every recursive function should have base case(s)
- every recursive call/recursion step of a function should be to “solve a smaller problem”, which should eventually *converge* to a base case.

Some notes:

- often the recursive step includes the keyword return
- the recursion step executes while the original call to the function is still “open”

Recursive Functions

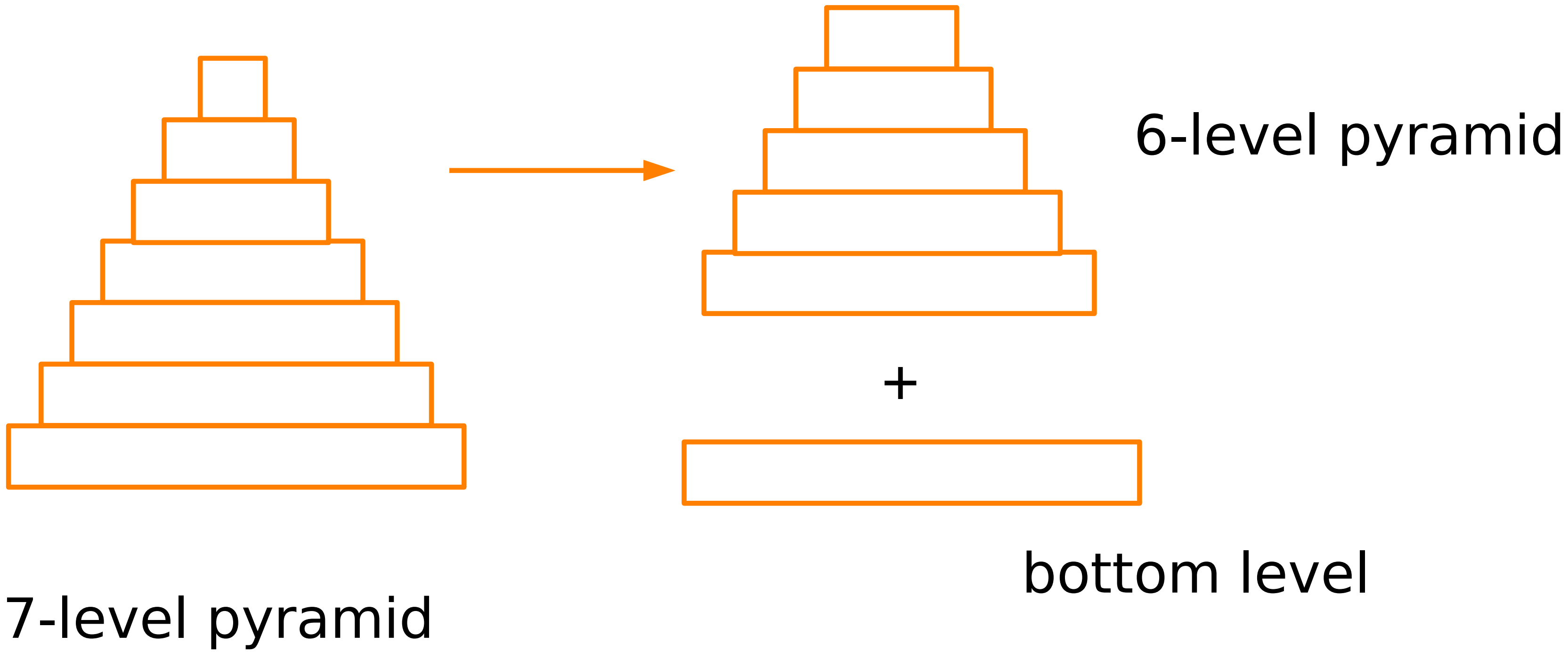
Examples of ***structural recursion***:



a bullseye

Recursive Functions

Examples of ***structural recursion***:



Recursive Functions

Let’s recall Fibonacci numbers: 0 1 1 2 3 5 8 13 …

Recursive Functions

Let’s recall Fibonacci numbers: 0 1 1 2 3 5 8 13 …

[Def, *recursive*]

$F(0)=0$ (*base case*)

$F(1)=1$ (*base case*)

$F(n)=F(n-1) + F(n-2)$ for all integers $n>1$

Recursive Functions

Let's recall Fibonacci numbers: 0 1 1 2 3 5 8 13 …

[Def, *recursive*]

$F(0)=0$ (*base case*)

$F(1)=1$ (*base case*)

$F(n)=F(n-1)+F(n-2)$ for all integers $n>1$

```
unsigned long fibonacci(int n){

    if (n == 0 or n == 1) { return n;}

  else {
   return fibonacci(n-1) + fibonacci(n-2);
    }
}
```

Recursive Functions

[Def, *recursive*]

$F(0)=0$ (*base case*)

$F(1)=1$ (*base case*)

$F(n)=F(n-1) + F(n-2)$ for all integers $n>1$

Let's come up with an iterative version!

Recursive Functions

[Def, *recursive*]

$F(0)=0$ (*base case*)

$F(1)=1$ (*base case*)

$F(n)=F(n-1) + F(n-2)$ for all integers $n>1$

Let's come up with an iterative version!

- Start with the first two Fibonacci numbers: 0 and 1,
- Grow them, one by one:
  - the next one should be 0 + 1 = 2
  - the next one should be 1+2 = 3
  - the next one should be 2 + 3 = 5, etc
- Stop when n-1 iterations are performed (to get the n^{th} Fibonacci number)

Recursive Functions

[Def, *recursive*]

$F(0)=0$ (*base case*)

$F(1)=1$ (*base case*)

$F(n)=F(n-1) + F(n-2)$ for all integers $n>1$

```
unsigned long fibonacci_it(int n) {
   unsigned long curr{ 1 }, prev{ 0 }, tmp;
   if (n == 0 or n == 1) { return n; }

   for (int i = 2; i <= n; i++) {
      tmp = curr;
      curr = curr + prev;
      prev = tmp;
   }
   return curr;
}
```

Recursive Functions

```
unsigned long fibonacci(int n){
   if (n == 0 or n == 1) { return n;}
   else {  return fibonacci(n-1) + fibonacci(n-2); }
}

unsigned long fibonacci_it(int n) {
   unsigned long curr{ 1 }, prev{ 0 }, tmp;
   if (n == 0 or n == 1) { return n; }

   for (int i = 2; i <= n; i++) {
      tmp = curr;
      curr = curr + prev;
      prev = tmp;
   }
   return curr;
}
```

Let's trace the call of `fibonacci(5)` and of `fibonacci_it(5)`.

Recursive Functions

let's convert iterative version to recursive version!

```
unsigned long fibonacci_it(unsigned long n) {

    unsigned long curr{ 1 }, prev{ 0 }, tmp;

    if (n == 0 or n == 1) { return n; }

    for (int i = 2; i <= n; i++) {
        tmp = curr;
        curr = curr + prev;
        prev = tmp;
    }

    return curr;
}
```

Recursive Functions

fib_rec

```
unsigned long fibonacci_it(unsigned long n) {
    unsigned long curr{ 1 }, prev{ 0 }, tmp;
    if (n == 0 or n == 1) { return n; }
    for (int i = 2; i <= n; i++) {
        tmp = curr;
        curr = curr + prev;
        prev = tmp;
    }
    return curr;
}
```

disassemble

fib_rec_helper(

Recursive Functions

```
fib_rec(unsigned long n)

{
 unsigned long curr{ 1 }, prev{ 0 }, tmp;

 if (n == 0 or n == 1) { return n; }
 else { return fib_rec_helper(...) }}
 for (int i = 2; i <= n; i++) {
     tmp = curr;
     curr = curr + prev;
     prev = tmp;
 }

 return curr;
}
```

disassemble

```
fib_rec_helper(
```

Recursive Functions

```
fib_rec(unsigned long n)

{
 unsigned long curr{ 1 }, prev{ 0 }, tmp;

 if (n == 0 or n == 1) { return n; }
 else { return fib_rec_helper(...) }}
 for (int i = 2; i <= n; i++) {
     tmp = curr;
     curr = curr + prev;
     prev = tmp;
 }

 return curr;
}
fib_rec_helper(
```

disassemble

base case of the recursion

all the changes - send through parameters of recursive function call

Recursive Functions

```
fib_rec(unsigned long n)

{
 unsigned long curr{ 1 }, prev{ 0 }, tmp;

 if (n == 0 or n == 1) { return n; }
 else { return fib_rec_helper(...) }}
 for (int i = 2; i <= n; i++) {
     tmp = curr;
     curr = curr + prev;
     prev = tmp;
 }

 return curr;
}
```

disassemble

base case of the recursion

all the changes send - through parameters of recursive function call

```
fib_rec_helper(prev, curr, i, n)
```

Recursive Functions

```
fib_rec(unsigned long n)

{
unsigned long curr{ 1 }, prev{ 0 }, tmp;

if (n == 0 or n == 1) { return n; }
else { return fib_rec_helper(prev, curr, 2, n) }}
for (int i = 2; i <= n; i++) {
        tmp = curr;
        curr = curr + prev;
        prev = tmp;
}

return curr;
}
fib_rec_helper(prev, curr, i, n)
```

disassemble

base case of the recursion

all the changes send - through parameters of recursive function call

Recursive Functions

```
fib_rec(unsigned long n)

{
unsigned long curr{ 1 }, prev{ 0 }, tmp;

if (n == 0 or n == 1) { return n; }
else { return fib_rec_helper(prev, curr, 2, n) }}
for (int i = 2; i <= n; i++) {
     tmp = curr;
     curr = curr + prev;
     prev = tmp;
}

return curr;
}
fib_rec_helper(prev, curr, i, n)
```

disassemble

base case of the recursion

all the changes send - through parameters of recursive function call

Recursive Functions

```
unsigned long fib_rec(unsigned long n) {

    unsigned long curr{ 1 }, prev{ 0 }, tmp;

    if (n == 0 or n == 1) { return n; }
    else { return fib_rec_helper(prev, curr, 2, n)  }
}

unsigned long fib_rec_helper(prev,curr,i,n)
    if (i == n) { return curr; }
    else { return fib_rec_helper(curr,prev+curr,i+1,n); }

        tmp = curr;
        curr = curr + prev;
        prev = tmp;
    }
```

all the changes - send through parameters of recursive function call

Recursive Functions

```
unsigned long fib_rec(unsigned long n) {

    unsigned long curr{ 1 }, prev{ 0 }, tmp;

    if (n == 0 or n == 1) { return n; }
    else { return fib_rec_helper(prev, curr, 2, n)  }
}

unsigned long fib_rec_helper(prev,curr,i,n)
    if (i == n) { return curr; }
    else { return fib_rec_helper(curr,prev+curr,i+1,n); }
    }
```

Recursive Functions

```
unsigned long fib_rec(unsigned long n) {

    unsigned long curr{ 1 }, prev{ 0 }, tmp;

    if (n == 0 or n == 1) { return n; }
    else { return fib_rec_helper(prev, curr, 2, n)  }
}

unsigned long fib_rec_helper(prev,curr,i,n)
    if (i == n) { return curr; }
    else { return fib_rec_helper(curr,prev+curr,i+1,n); }
    }
```

Let’s trace the call of `fib_rec(5)`

Recursive Functions

```
unsigned long fib_rec(unsigned long n) {

    unsigned long curr{ 1 }, prev{ 0 }, tmp;

    if (n == 0 or n == 1) { return n; }
    else { return fib_rec_helper(prev, curr, 2, n)  }
}

unsigned long fib_rec_helper(prev,curr,i,n)
    if (i == n) { return curr; }
    else { return fib_rec_helper(curr,prev+curr,i+1,n); }
    }
```

Let’s trace the call of `fib_rec(5)`

See the file `FibFunctions.cpp`

Recursive Functions: call stack

How are recursive called handled?

Recursive Functions: call stack

How are recursive calls handled?

- *call stack* with *function activation record*s

Recursive Functions: call stack

How are recursive calls handled?

- *call stack* with *function activation record*s
- when a function is called, the language implementation sets aside *function activation record* that contains a copy of all its parameters and local variables
- *activation record*s are stored in a *call stack*
  - *last record to be stored is the first one to be retrieved*

Recursive Functions: call stack

How are recursive calls handled?

- *call stack* with *function activation record*s
- when a function is called, the language implementation sets aside *function activation record* that contains a copy of all its parameters and local variables
- *activation record*s are stored in a *call stack*
  - *last* *record to be stored is the* *first* *one to be retrieved*

Can we run out of space is a class stack?

Recursive Functions: call stack

How are recursive calls handled?

- *call stack* with *function activation record*s
- when a function is called, the language implementation sets aside *function activation record* that contains a copy of all its parameters and local variables
- *activation record*s are stored in a *call stack*
  - *last record to be stored is the first one to be retrieved*

Can we run out of space is a class stack?

- Yes, it is often called *stack overflow*

Recursive Functions: call stack

How are recursive calls handled?

- *call stack* with *function activation record*s
- when a function is called, the language implementation sets aside *function activation record* that contains a copy of all its parameters and local variables
- *activation record*s are stored in a *call stack*
  - *last record to be stored is the first one to be retrieved*

Can we run out of space is a class stack?

- Yes, it is often called *stack overflow*
- If we forget a base case or do not make sure that each recursive call is "solving a smaller problem", we may end up with infinite sequence of function calls, which will cause the *stack overflow*.

In-class practice

Recall the factorial function: $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$, $n > 0$ and $0! = 1$

1. Come up with a recursive definition of the function

2. Implement the recursive definition of the factorial function

```
long int fact_rec(int n)
```

and test your function.

3. Is the implementation an efficient one? Trace the call of `fact_rec(5)`.

Palindromes

- [simple definition] A palindrome is a word that is spelled the same from both ends
  - Examples: anna, madam, racecar, etc.
- [definition] A palindrome is a word, number, phrase, or other sequence of symbols that reads the same backwards as forwards, ignoring punctuation symbols and lower/upper case
  - Examples: race car; Madam, I'm Adam!

Palindromes

- [simple definition] A palindrome is a word that is spelled the same from both ends
  - Examples: anna, madam, racecar, etc.
- [definition] A palindrome is a word, number, phrase, or other sequence of symbols that reads the same backwards as forwards, ignoring punctuation symbols and lower/upper case
  - Examples: race car; Madam, I'm Adam!
- Let's see how we can check whether a given word is a palindrome, following the simple definition and assuming that only lower case alphabetic letters are present.

Palindromes using string

Idea: start reading the string from the front and the back, compare the letters, move into the middle;

Palindromes using string

Idea: start reading the string from the front and the back, compare the letters, move into the middle;

```
bool is_palindrome(const string& s) {
   int first = 0;
   int last = s.length() - 1;
   while ( first < last) {
      if ( s[first] != s[last] ) return false;
      ++ first;
      --last;
   }
   return true;
}
```

Palindromes using array

Idea: start reading the string from the front and the back, compare the letters, move into the middle

```
bool is_palindrome(const char s[], int n) {
   int first = 0;
   int last = n - 1;
   while ( first < last) {
      if ( s[first] != s[last] ) return false;
      ++ first;
      --last;
   }
   return true;
}
```

Palindromes using pointers

Idea: start reading the string from the front and the back, compare the letters, move into the middle

```
bool is_palindrome(const char* first, const char*
last) {
    while ( first < last) {
        if ( *first != *last ) return false;
        ++ first;
        --last;
    }
    return true;
}
```

See the file **palindromes.cpp** for their use

Palindromes: recursive version

Let's come up with a recursive version of the palindromes check!

Idea of iterative version: start reading the string from the front and the back, compare the letters, move into the middle;

Idea of recursive version:

- Check the first and the last letters:
  - If they are the same, call the function on the string without the first and last letters (smaller string, i.e. smaller task)
  - If they are different, return false
- When to stop: if we got an empty string, or a string with one letter only
  - It means that the word is palindrome, return true

Resources used for these slides

- slides provided by B. Stroustrup at https://www.stroustrup.com/PPP2slides.html
- Class textbook
- C++ How to Program, 10th Edition, by Paul Deitel and Harvey Deitel, 2017, Pearson