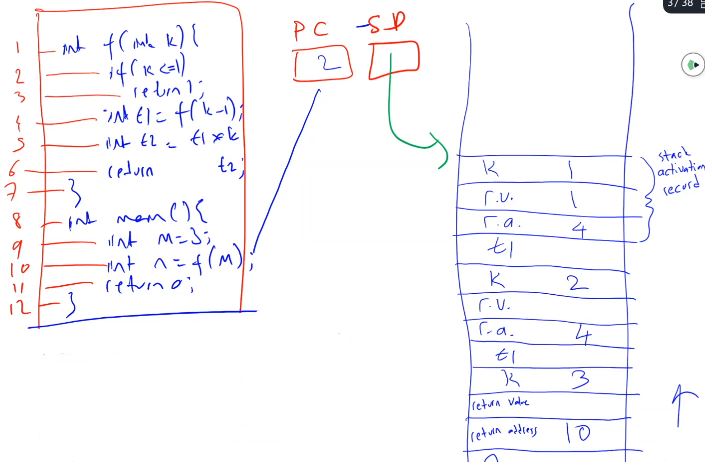
**RECURSION**

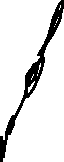
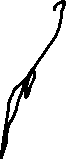
When we call a function, we store return address and return value in the stack.

When we call a function that takes an integer with a variable, copy constructor will be called on the function parameter.

When the function is done, stack activation record is popped up (empty).



Return address of main belongs to the OS. We don’t know that address but it is valid.



Stack activation record will be popped up from now on, we assign 1 to t1 and we create t2 which is t1\*k which is 2 and go on like this.

Then stack activation record will be t2, t1, k, rv, ra from top to bottom.

Function f never know who called it. It just do what it has to do.

There are 2 places for the copy constructor:

1. Once you copy t2 to return value position.
2. When you call function, you copy m.

**Merge Sort :** Divide your array into 2 and sort both parts, do it recursively. Fastest sort. Efficiency: nlogn

**One recursion example:**

writeVertical(1234)

1

2

3

4

void writeVertical(int i){

if (i < 10){

cout << i << endl;

return;

}

writeVertical(i/10);

cout << i % 10 << endl;

}

If you change last 2 lines’ position, then output would be reverse.

**Another recursion example:**

int strlen(const char \*s){

if (\*s == ‘\0’) return 0;

int t = strlen(s+1);

return 1 + t;

}

**Another recursion example:**

const char \* strfind(const char \* s, char t){

if (\*s == ‘\0’) return nullptr;

if (\*s == t) return s;

return strfind(s + 1, t);

}

This kind of functions are called “tail recursive” functions.

Tail recursion means that last statement in a function is a function call to itself. So I am returning whatever the function returns as seen in the strfind example. My return value is the same as return value of the strfind. That means I don’t have to keep a return value position for my function call and in fact before calling the function, I can get rid of all of my local variables because I don’t need them anymore. As soon as I call strfind, my function is done. I don’t have to keep any local variables.

We like tail recursive functions bc everytime you make a call to the function, you won’t use any stack space. This will not cause any out of stack or stack overflow problems. Function will reuse the stack space over and over again.

int f(int k){

if (k <= 1) return 1;

return k \* f(k-1);

}

This function is not tail recursive because last statement is multiplication, not function call.

You can do like this to make it tail recursive:

int f(int k){

return fh(k, 1);

}

int fh(int k, int fact){

if (k <= 1) return fact;

return fh(k-1, fact\*k);

}

You are saying that my return value and return address will be the same as return value and return address of the last function that is called with return.

There are some functions that cannot be tail recursive like merge sort bc you have to call same function twice (sort left, sort right). I have to sort them separately and combine them later.

**INFINITE RECURSION**

Base case must eventually be entered. If it doesn’t 🡪 “infinite” recursion

Actually there is no such thing as infinite recursion. If you keep calling a same function over and over again, you will end up using all of your memory and your program will be terminated.

There is infinite loop (i.e., for(;;); ). Your computer will run until it is dead.

For example: “void g() { g(); }” this wont run more than half a second bc you will run out of space. Every time you make a call, your memory is used for the return address. This will eat up your memory.

int main(){ return main();} 🡪 Tail recursion so not “infinite”

int main(){ return main() + 1;} 🡪 Compiler will handle this with optimization flags. So not infinite again.

For linked lists and trees, recursion is perfect fit bc each of your children is exactly same as you. They are only smaller.

If it is suitable write your program using recursion. If you can write it using the tail recursion, write the tail recursion. Don’t forget the optimize your code using the -O3 flags. So compiler will help you to produce efficient code.

**SECURITY PROBLEM**

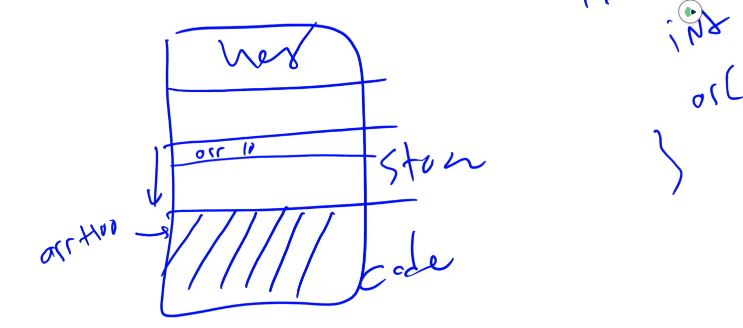
int f(){

int arr[10];

arr[100] = 7;

}

This is not a compiler error, this is logical error.



You are writing something to your data in your code segment. So you are messing up your code.

If I do this on purpose, if I write instructions to that position (arr[100] = 0xF27D;) that means you are writing new code in your code segment and since I am modifying my return addresses too, if I do this writing instruction in very very careful way, I can modify my code and I can jump to modified code that means I can install my malicious software inside somebody else’s code.

Recursive:

* Runs slower, uses more storage
* Elegant solution, less coding

Whatever you can do with recursion, you can do with iteration.

**Another recursion example:**

int power(int x, int n)

{

if(n < 0)

{

cout << “Illegal argument”;

exit(1);

}

if(n > 0) return (power(x, n-1) \* x);

else return(1);

}

**THINKING RECURSIVELY**

* Ignore details
  1. Forget how stack works
  2. Forget the suspended computations
  3. This is an abstraction and encapsulation principle
* Don’t trace entire recursive sequence
* Just check 3 properties
  1. No infinite recursion (have a valid base case)
  2. Stopping cases return correct values
  3. Recursive cases return correct values

**TAIL RECURSION**

A function that is tail recursive if it has the property that no further computation occurs after the recursive call; the function immediately returns.

Tail recursive functions can easily be converted to a more efficient iterative solution. May be done automatically by your compiler.

**Another recursion example:**

int strlen(const char \* s){

//return \*s == ‘\0’ ? 0 : 1 + strlen(s + 1);

return strlen2(s, 0); //Tail recursion

}

int strlen2(const char \* s, int i){

if (\*s == ‘\0’) return i;

return strlen2(s + 1, i + 1);

}

**MUTUAL RECURSION**

When 2 or more functions call each other it is called mutual recursion.

f g f g

f is calling g, g is calling f, f is calling g,…

Example:

* Determine if a string has an even or odd number of 1s by invoking a function that keeps track if the number of 1s seen so far is even or odd.
* Would result in stack overflow for long strings.

CHECK 13-05

**BINARY SEARCH**

* Extremely fast
  + Logarithmic efficiency (logn)
* Recursive function to search array
  + Determines if item is in list, and if so where in list it is
* Assumes array is sorted
* Breaks list in half
  + Determines if item in 1st or 2nd half
  + Then searches again just that half, of course recursively

CHECK 13-07

Functions that call 2 or more recursive calls at same function call. Those are very expensive. They cannot be implemented using the tail recursion.