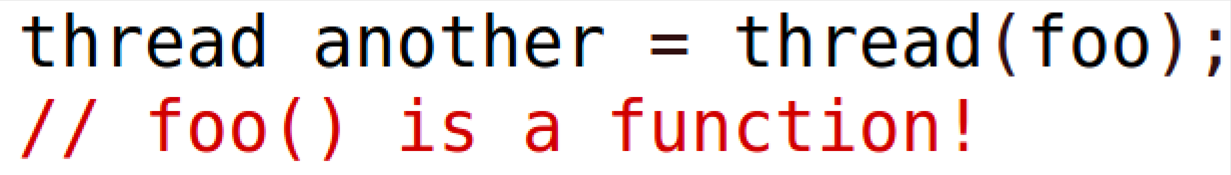
Parallel processing

YO DAWG, I HEARD YOU LIKED PROCESSORS...
...SO WE PUT PROCESSORS IN YOUR PROCESSOR SO YOU CAN PROCESS WHILE YOU PROCESS!

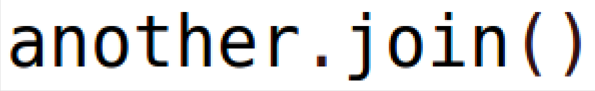
Highlights

- Making threads



```
thread another = thread(foo);
// foo() is a function!
```

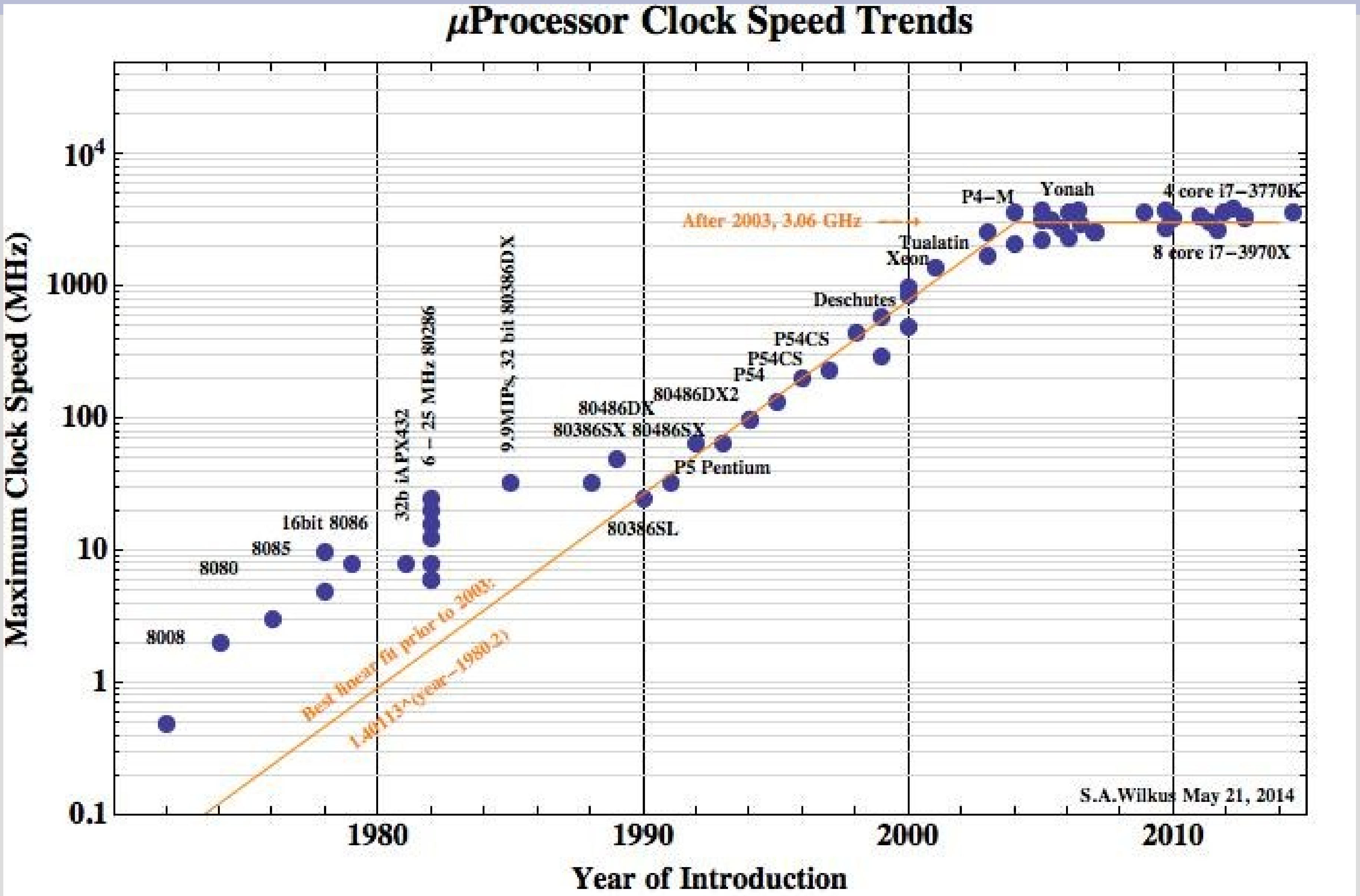
- Waiting for threads



```
another.join()
```

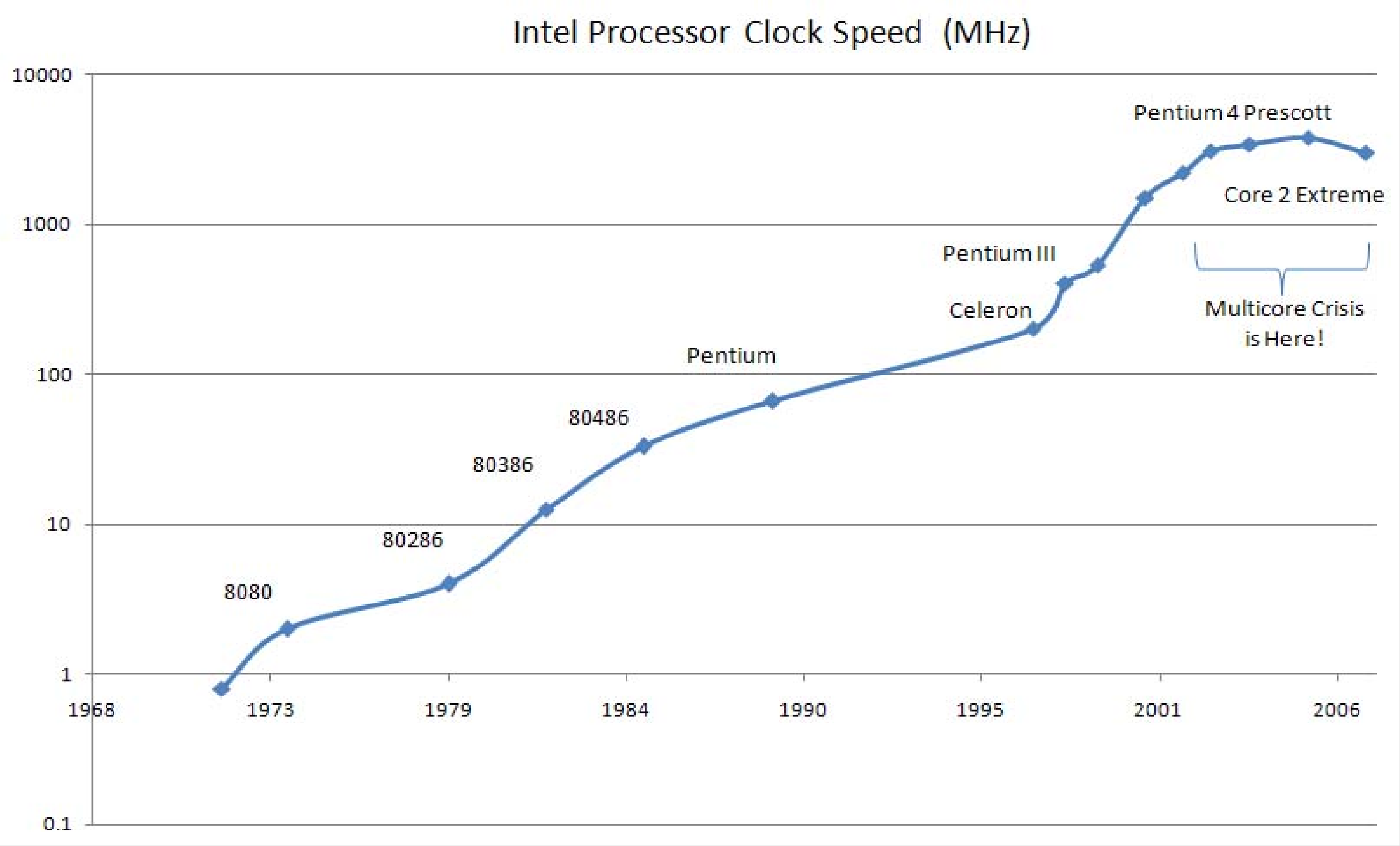
- Review (classes, pointers, inheritance)

Review: CPUs

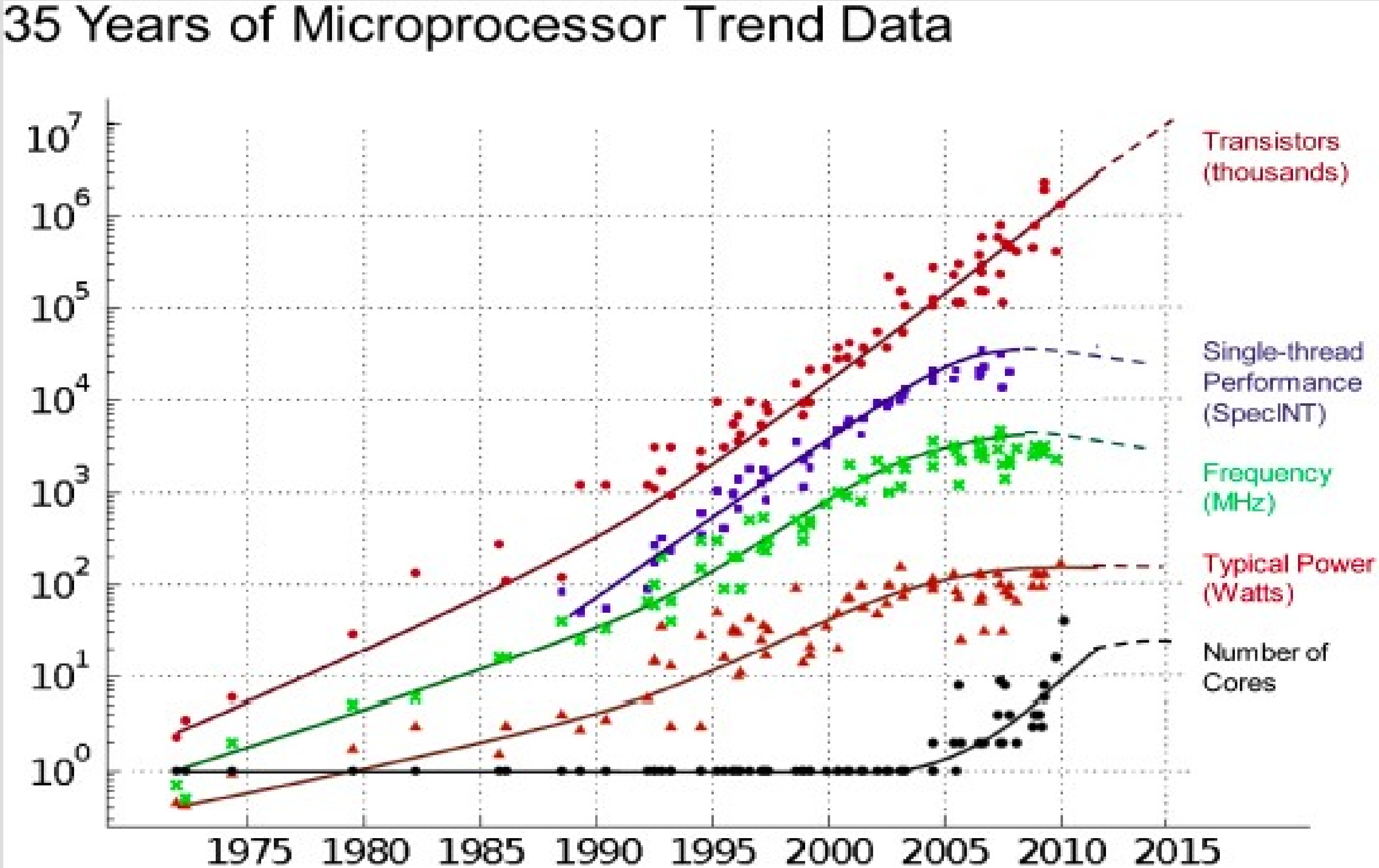

μProcessor Clock Speed Trends
Maximum Clock Speed (MHz)
10^4
1000
100
10
1
0.1
1980
1990
2000
2010
Year of Introduction
8008
8080
8085
16bit 8086
32b iAPX432
6 – 25 MHz 80286
9.9MIPs 32 bit 80386DX
80486DX
80486DX2
80386SX 80486SX
P5 Pentium
80386SL
P54
P54CS
P54CS
Deschutes
Xeon
Tualatin
P4–M
Yonah
4 core i7–3770K
8 core i7–3970X
After 2003, 3.06 GHz
Best linear fit prior to 2003:
1.40113^(year–1980.2)
S.A.Wilkus May 21, 2014

Review: CPUs

In the 2000s, computing too a major turn: multi-core processors (CPUs)

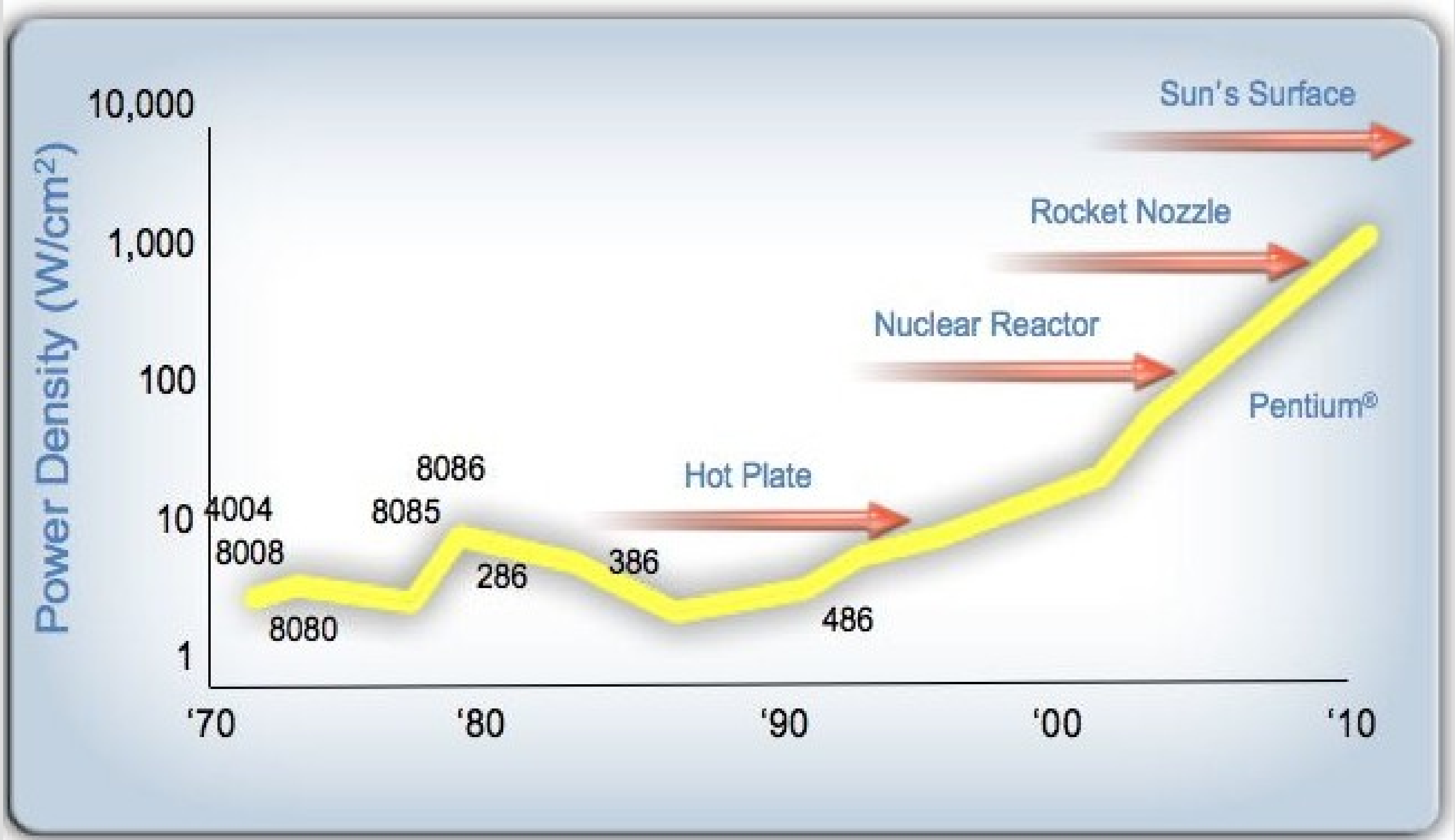


Review: CPUs

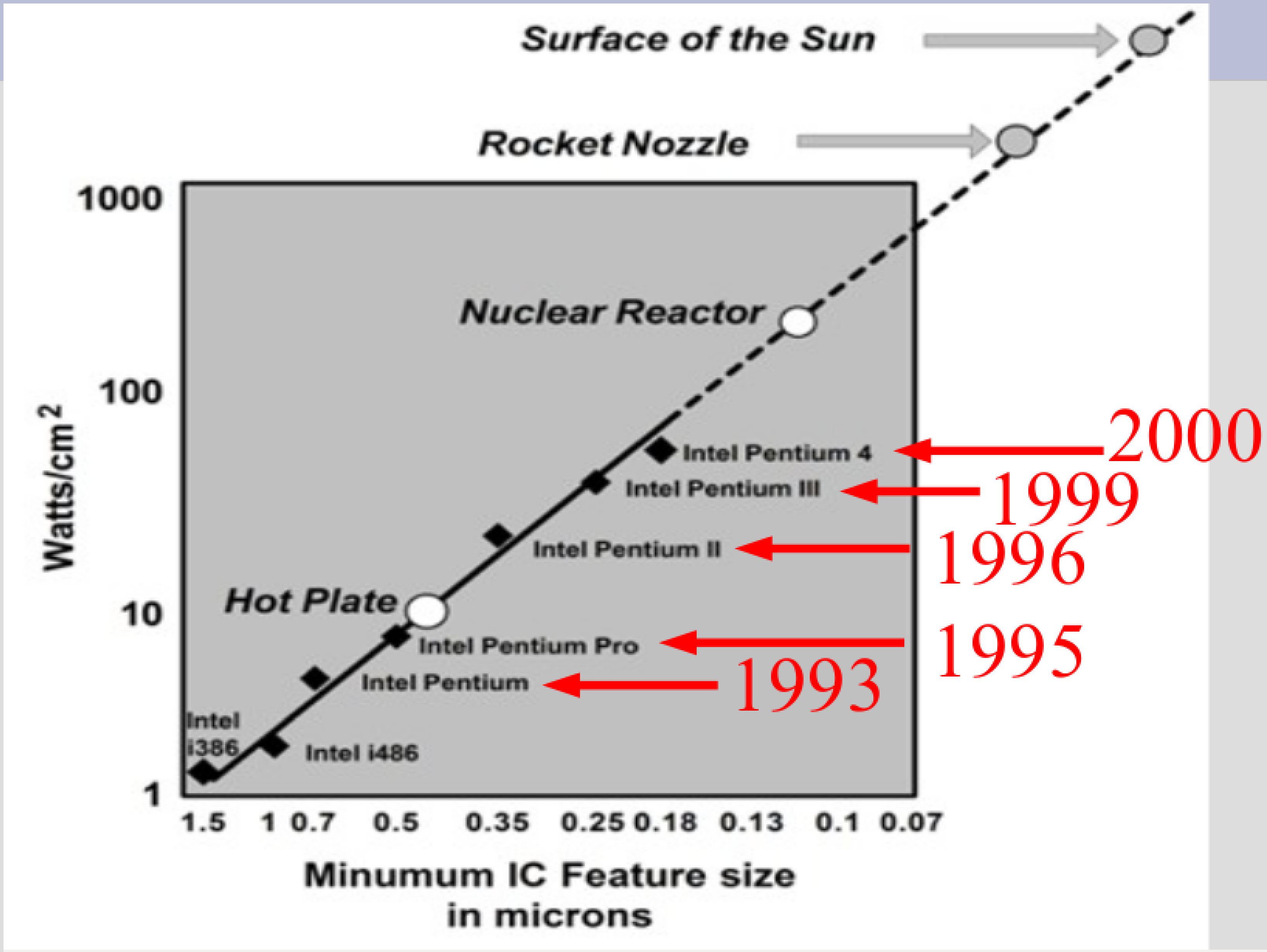

35 Years of Microprocessor Trend Data
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
1975
1980
1985
1990
1995
2000
2005
2010
2015
Transistors (thousands)
Single-thread Performance (SpecINT)
Frequency (MHz)
Typical Power (Watts)
Number of Cores

Review: CPUs

The major reason is due to heat/energy density



Review: CPUs



2000

1999

1996

1995

1993

Review: CPUs

This trend will almost surely not reverse

There will be new major advancements in computing eventually (quantum computing?)

But “cloud computing”, which has programs that “run” across multiple computers are going nowhere anytime soon

Terminology

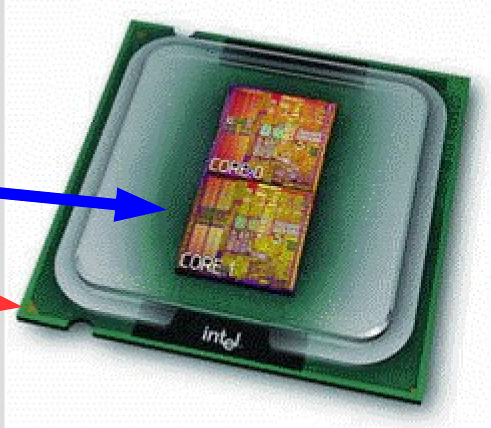
CPU = area of computer that does thinking

Core = processor = a thinking unit

Cores

CPU

front/back



Program = code = instructions on what to do

Thread = parallel process = an independent part of the program/code

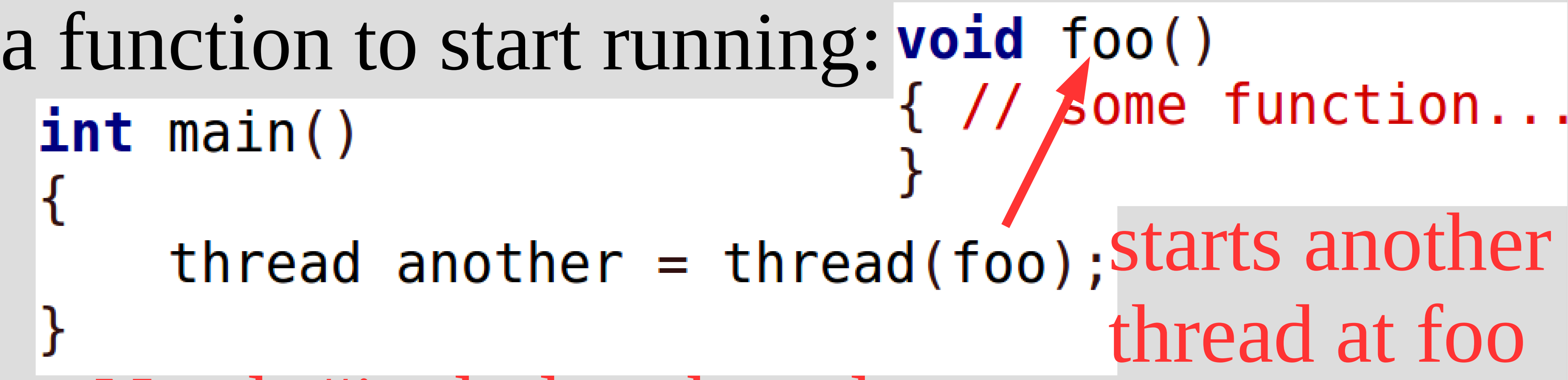
Program = string,

thread = 1 part of that

Parallel: how

So far our computer programs have run through code one line at a time

To get multiple parts running at the same time, you must create a new thread and give it a function to start running:



```
void foo()
{ // some function...
}
```

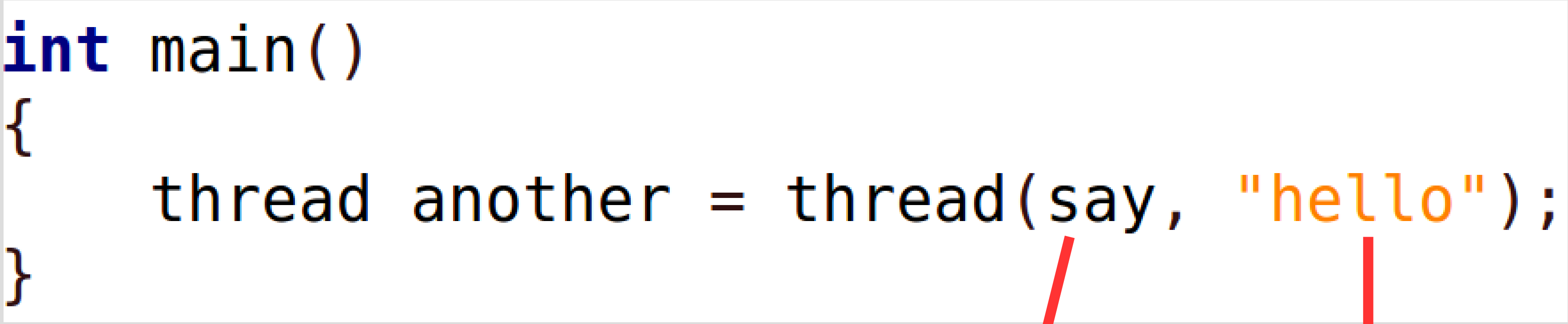
```
int main()
{
    thread another = thread(foo);
}
```

starts another thread at foo

Need: #include <thread>

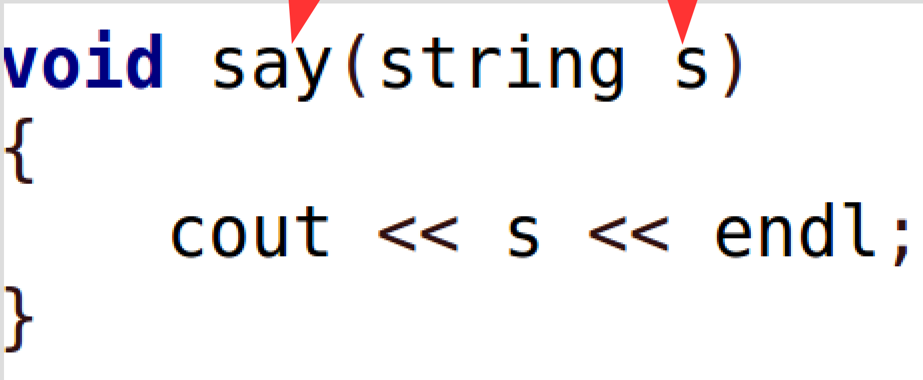
Parallel: how

If the function wants arguments, just add them after the function in the thread constructor:



```
int main()
{
    thread another = thread(say, "hello");
}
```

This will start function “say” with first input as “hello”



```
void say(string s)
{
    cout << s << endl;
}
```

(see: createThreads.cpp)

Parallel: basics

The major drawback of distributed computing (within a single computer or between) is **resource synchronization** (i.e. sharing info)

This causes two types of large problems:

1. Conflicts when multiple threads want to use the same resource

2. Logic errors due to parts of the program having different information

1. Resource conflict

Siblings anyone?

1. Resource conflict

Public bathroom?

All your programs so far have had 1 restroom, but some parts of your program could be sped up by making 2 lines(as long as no issues)

1. Resource conflict

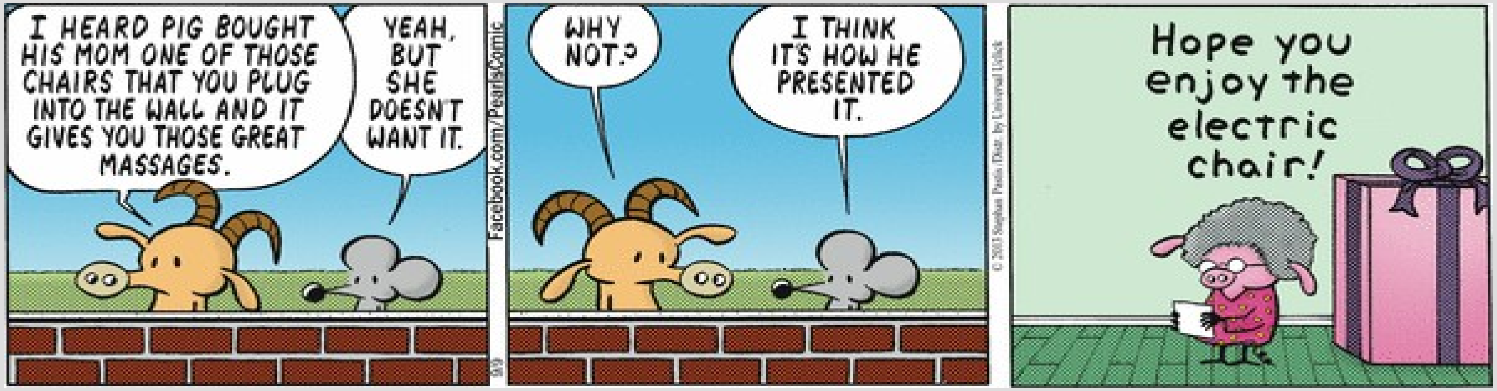
We will actually learn how to cause minor resource conflicts to ensure no logic errors

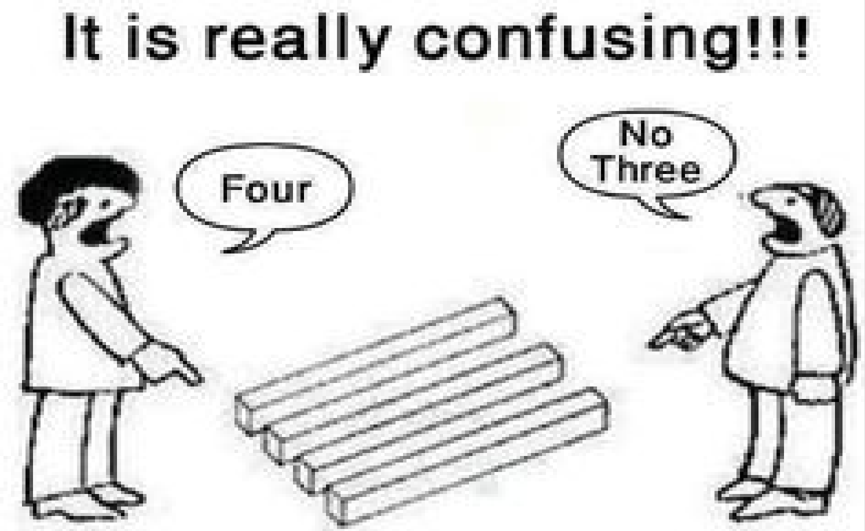
This is similar to a cost of calling your forgetful relative to remind them of something

This only needs to be done for the important matters that involve both of you (e.g. when the family get-together is happening)

2. Different information

If you and another person try to do something together, but not coordinated... disaster


I HEARD PIG BOUGHT HIS MOM ONE OF THOSE CHAIRS THAT YOU PLUG INTO THE WALL AND IT GIVES YOU THOSE GREAT MASSAGES.
YEAH, BUT SHE DOESN'T WANT IT.
Facebook.com/PearlsComic
WHY NOT?
I THINK IT'S HOW HE PRESENTED IT.
© 2013 Stephan Pastis/Dist. by Universal Uclick
Hope you enjoy the electric chair!



2. Different information

Each part of the computer has its own local set of information, much like separate people

Suppose we handed out tally counters and told two people to count the amount of people

2. Different information

However, two people could easily tally the number entering this room...

Simply stand one by each door and add them

Our goal is to design programs that have these two separate parts that can be done simultaneously (which tries to avoid sharing parts)

Parallel: how

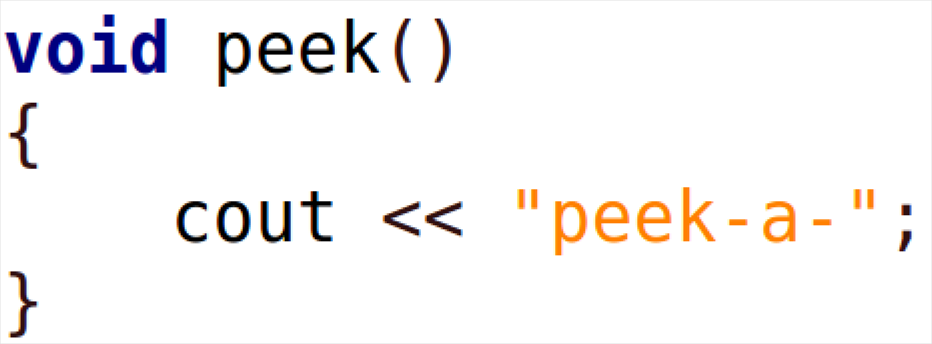
However, main() will keep moving on without any regard to what these threads are doing

If you want to synchronize them at some later point, you can run the join() function

This tells the code to wait here until the thread is done (i.e. returns from the function)

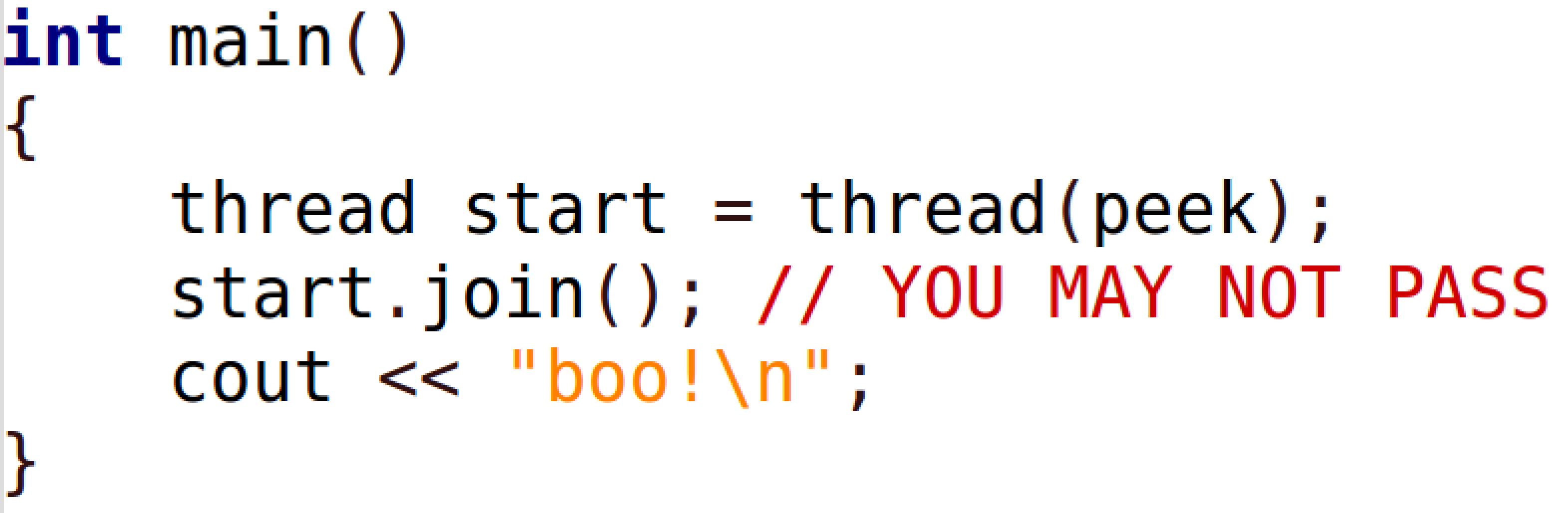
Parallel: how

Consider this:



```
void peek()
{
    cout << "peek-a-";
}
```

The start.join() stops main until the peek() function returns



```
int main()
{
    thread start = thread(peek);
    start.join(); // YOU MAY NOT PASS
    cout << "boo!\n";
}
```

(see: waitForThreads.cpp)

Parallel: advanced

None of these fix our counting issue (this is, in fact, not something we want to parallelize)

I only have 4 cores in my computer, so if I have more than 3 extra threads (my normal program is one) they fight over thinking time

Each thread speeds along, and my operating system decides which thread is going to get a turn and when (semi-random)

Parallel: advanced

We can force threads to not fall all over themselves by using a mutex (stands for “mutual exclusion”)

Mutexes have two functions:

1. lock
2. unlock

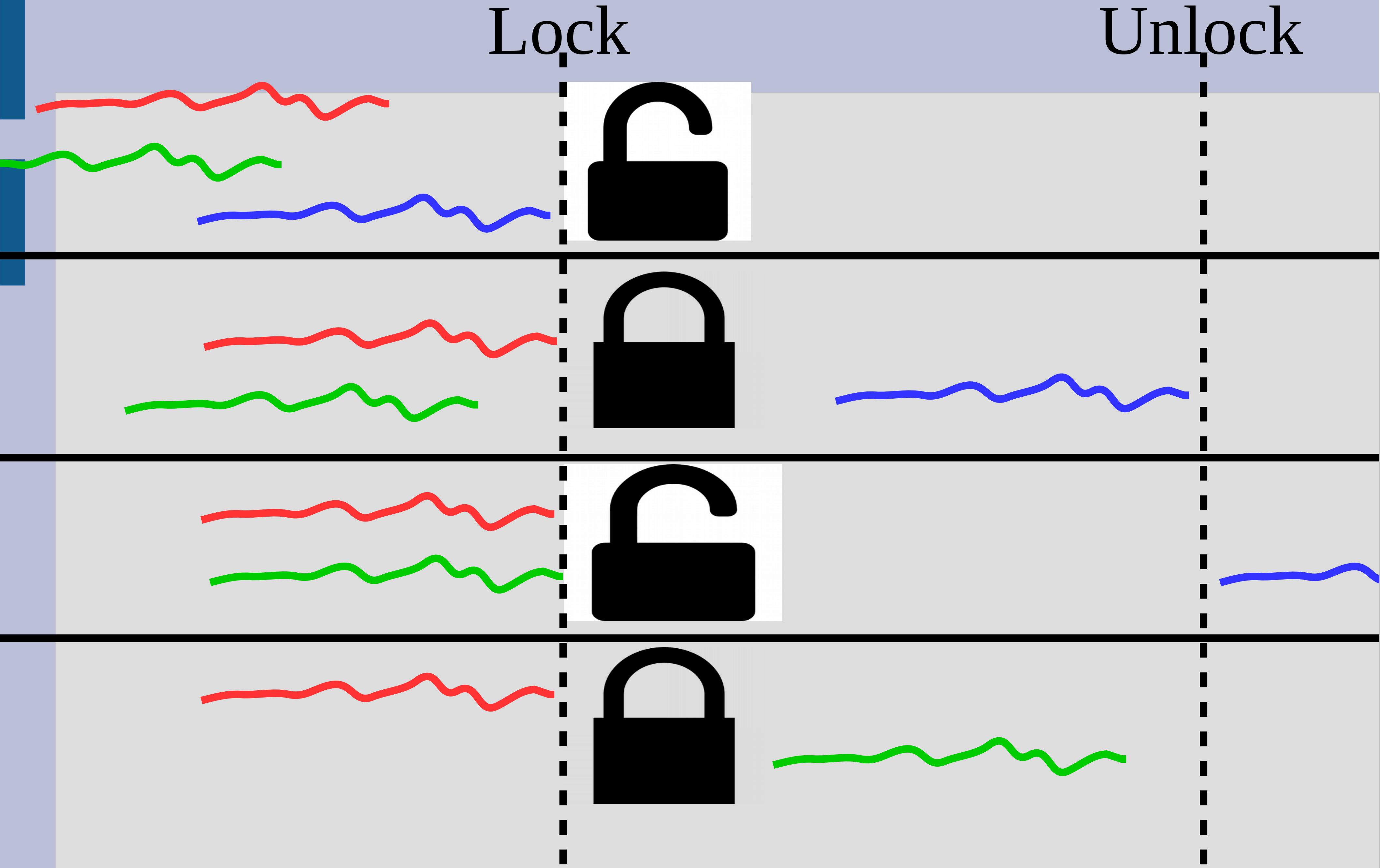
After one thread “locks” this mutex, no others can pass their “locks” until it is “unlocked”

Parallel: advanced

You can think about a “muxtex” like a porta-potty or airplane lavatory indicator:

It is a variable (information) that lets you know if you can proceed or have to wait (when it is your turn, you indicate that this mutex is “occupied” by you now via “lock()”)

Parallel: advanced



Parallel: advanced

These mutex locks are needed if we are trying to share memory between threads

Without this, there can be miscommunications about the values of the data if one thread is trying to change while another is reading

A very simple example of this is having multiple threads go: x++
(see: sharingBetweenThreads.cpp)

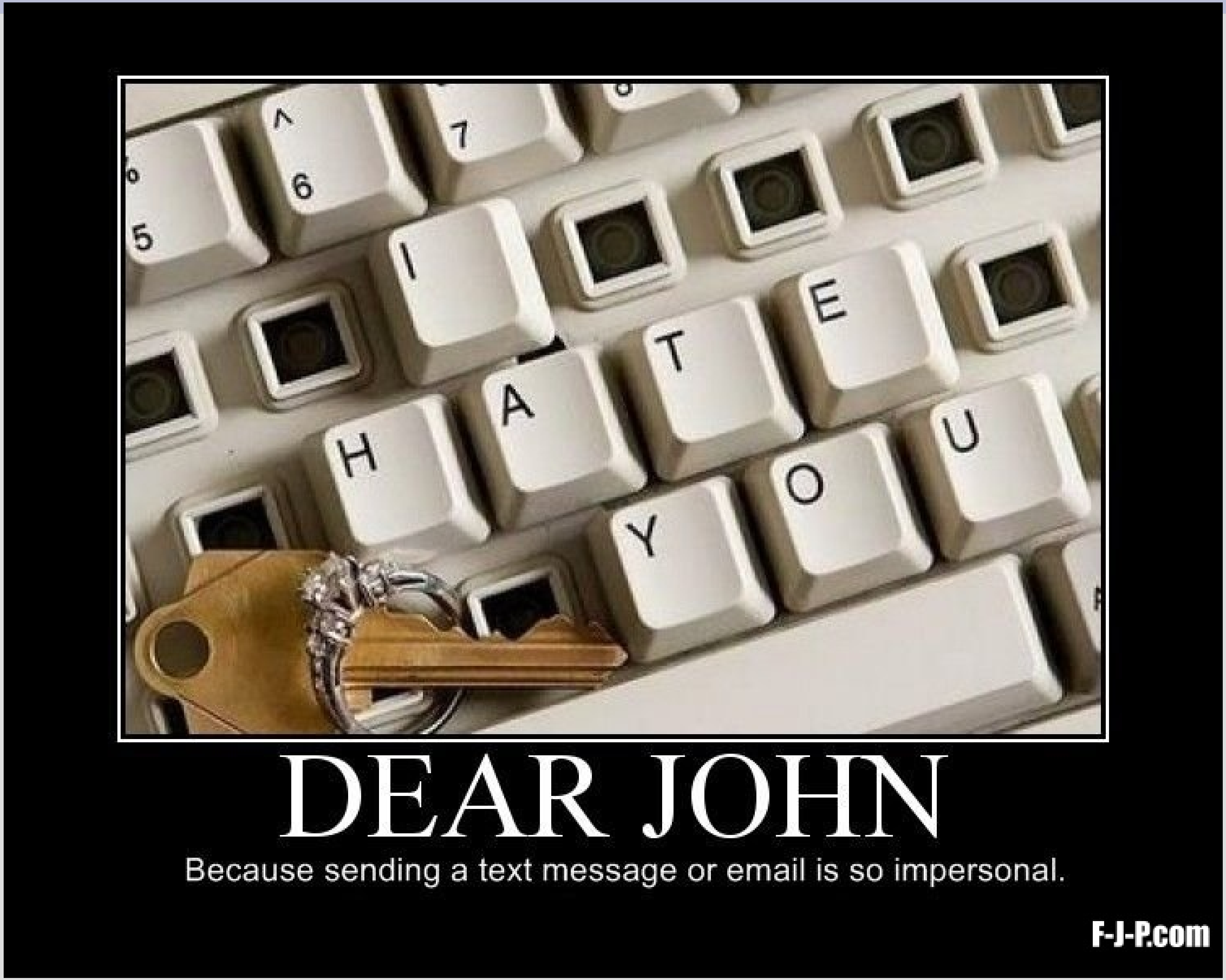
Parallel: advanced

You have to be careful when locking a mutex, as if that thread crashes or you forget to unlock ... then your program is in an infinite loop

There are way around this:
- Timed locks
- atomic operations instead of mutex

The important part is deciding what parts can be parallelized and writing code to achieve this

Review


I
HATE
YOU
DEAR JOHN
Because sending a text message or email is so impersonal.
F-J-P.com

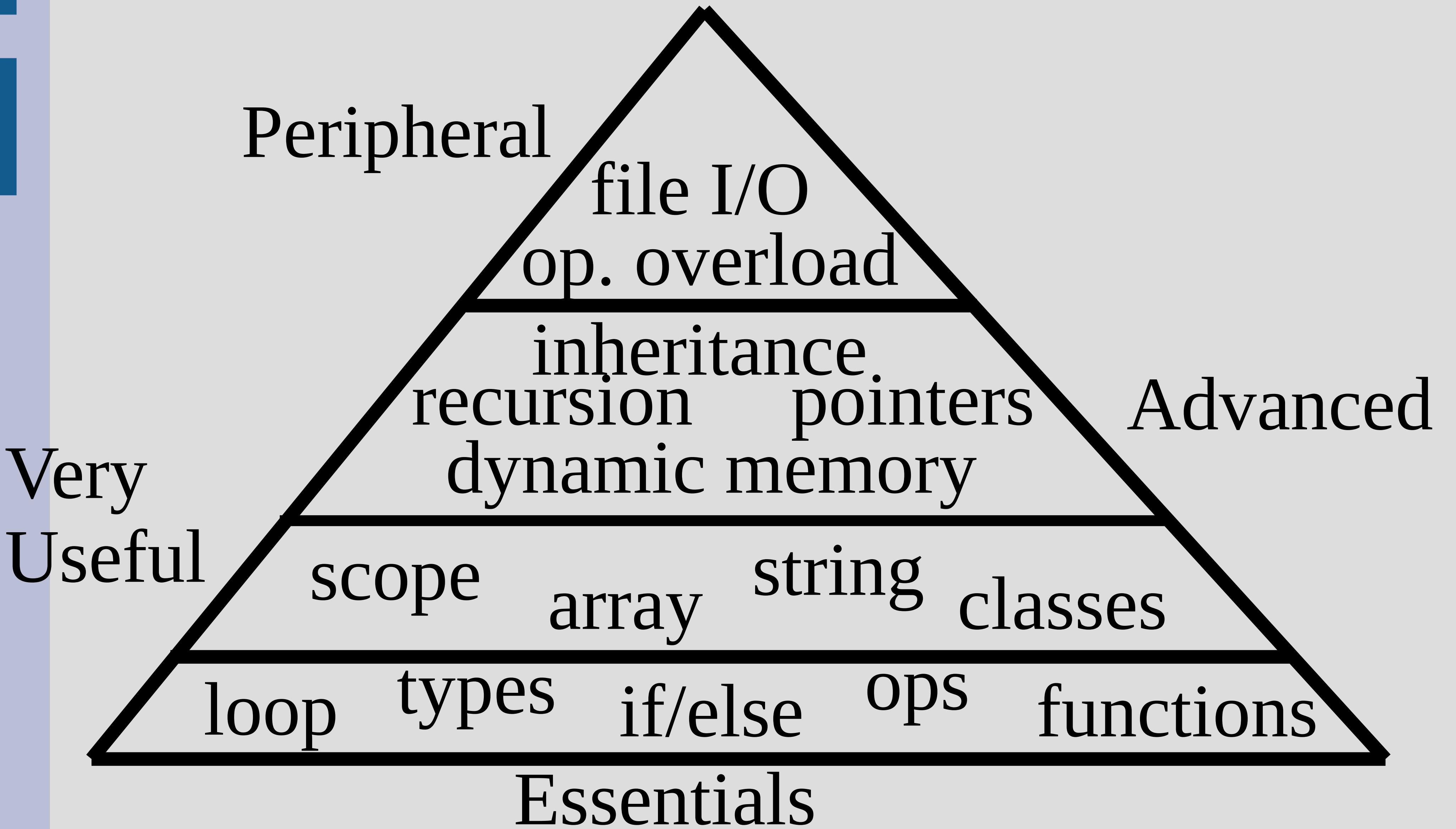
Final exam

Final exam will be 12 problems, drop any 2

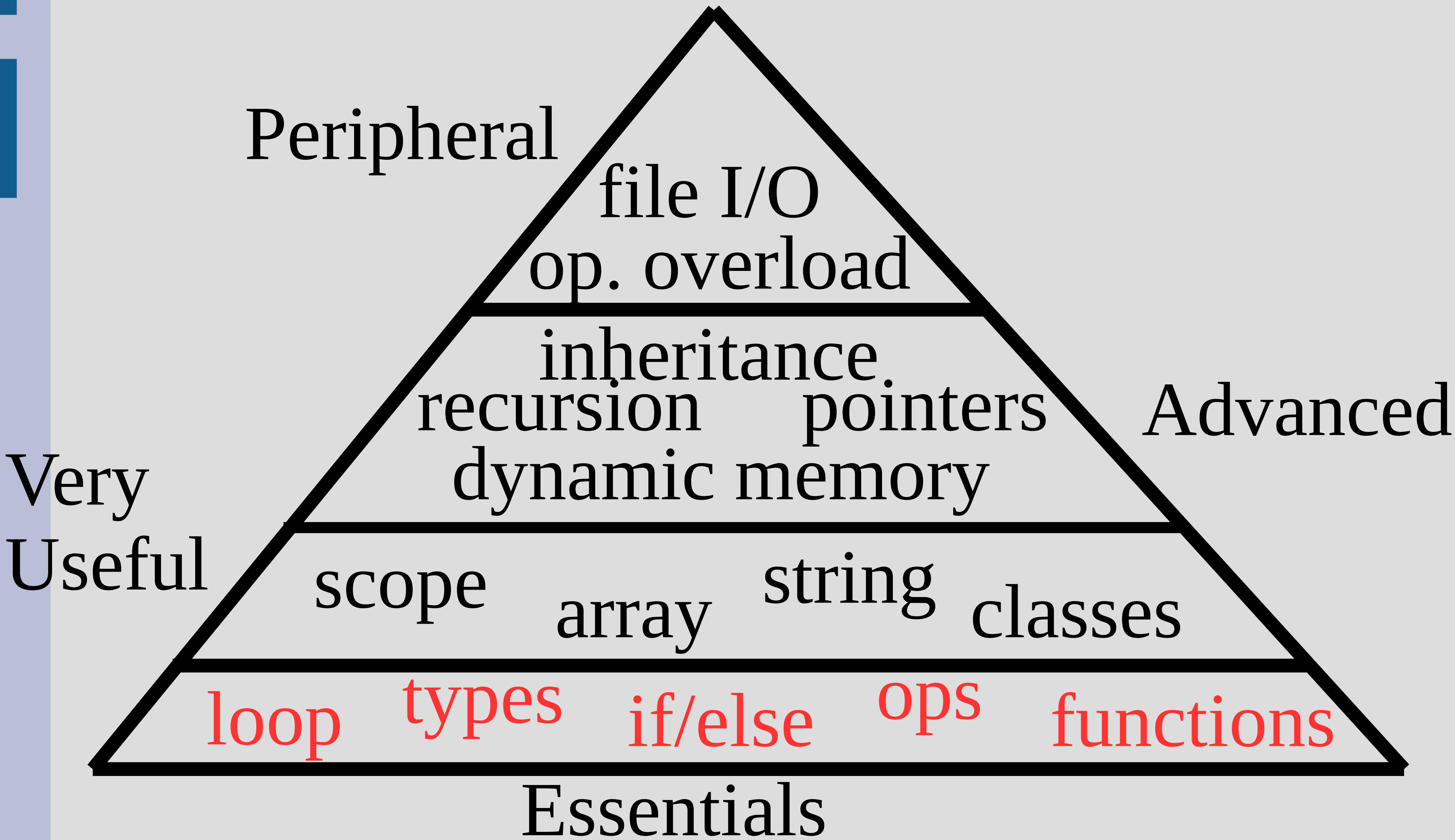
Cumulative up to and including week 14
(emphasis on weeks 9-14: classes & pointers)

2 hours exam time, so 12 min per problem
(midterm 2 had 8-ish)

Review: Overview



Review: Overview



Fundamental Types

bool - true or false
char - (character) A letter or number
int - (integer) Whole numbers
double - Larger decimal numbers

long - (long integers) Larger whole numbers
float - Decimal numbers

Functions

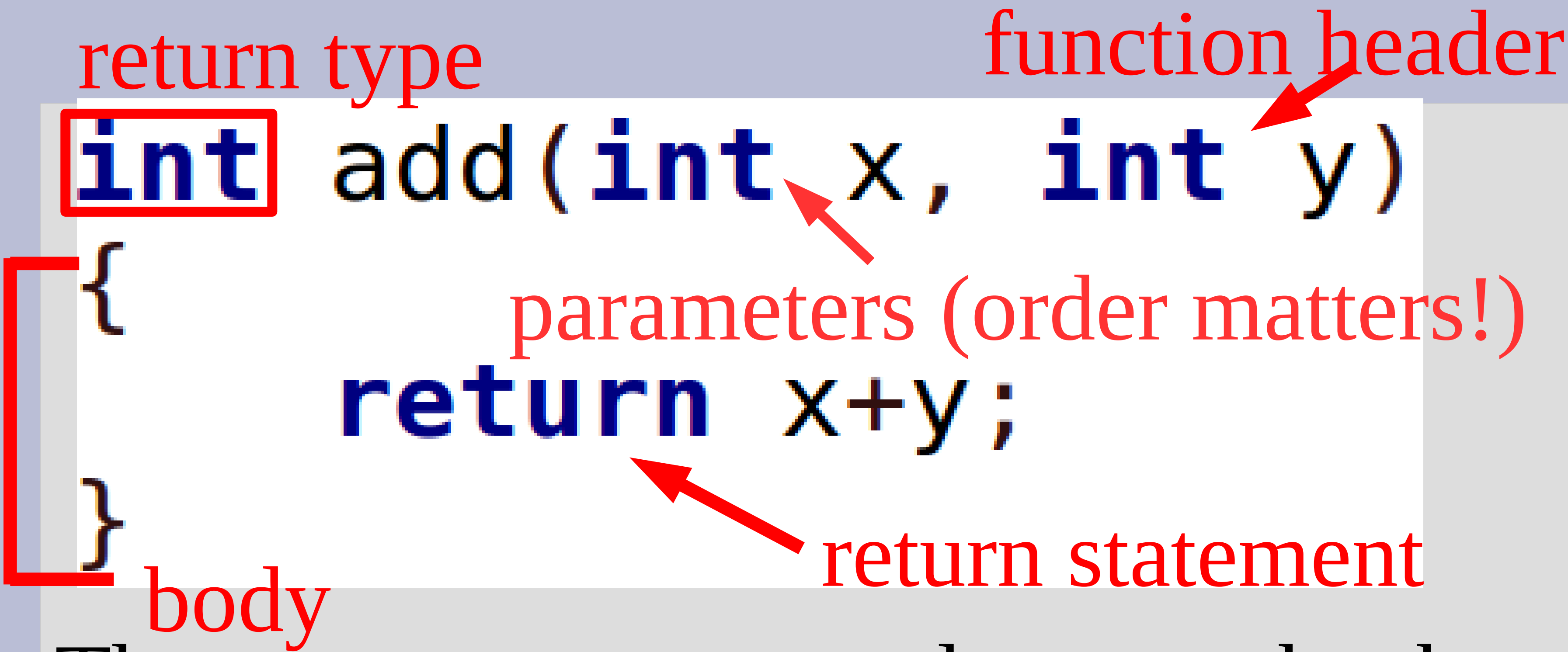
Functions allow you to reuse pieces of code (either your own or someone else's)

Every function has a return type, specifically the type of object returned

sqrt(2) returns a double, as the number will probably have a fractional part

The “2” is an argument to the sqrt function

Functions



```
int add(int x, int y)
{
        return x+y;
}
```

return type

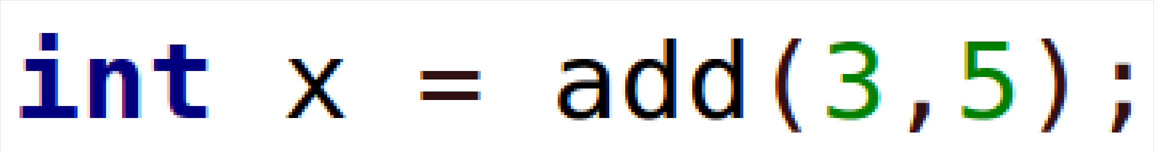
function header

parameters (order matters!)

return statement

body

The return statement value must be the same as the return type (or convertible)



```
int x = add(3,5);
```

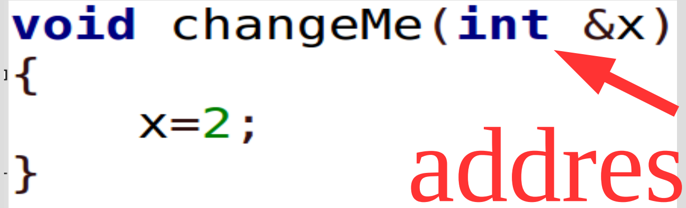
3 to x, 5 to y... value 8 returned and stored in x

Functions

Function call stack (after returning, start from where the previous function called it)

Overloading - same function name, different arguments (typically similar)

Call-by-reference (not copy)



```
void changeMe(int &x)
{
    x=2;
}
```

addresses share

Functions should be minimal

Order of operations

Order of precedence (higher operations first):

:: (scope resolution)

functions, . (dot), -> (sorta binary operators)

&, *, -, +, ++, -- and ! (unary operators)

*, / and % (binary operators)

+ and - (binary operators)

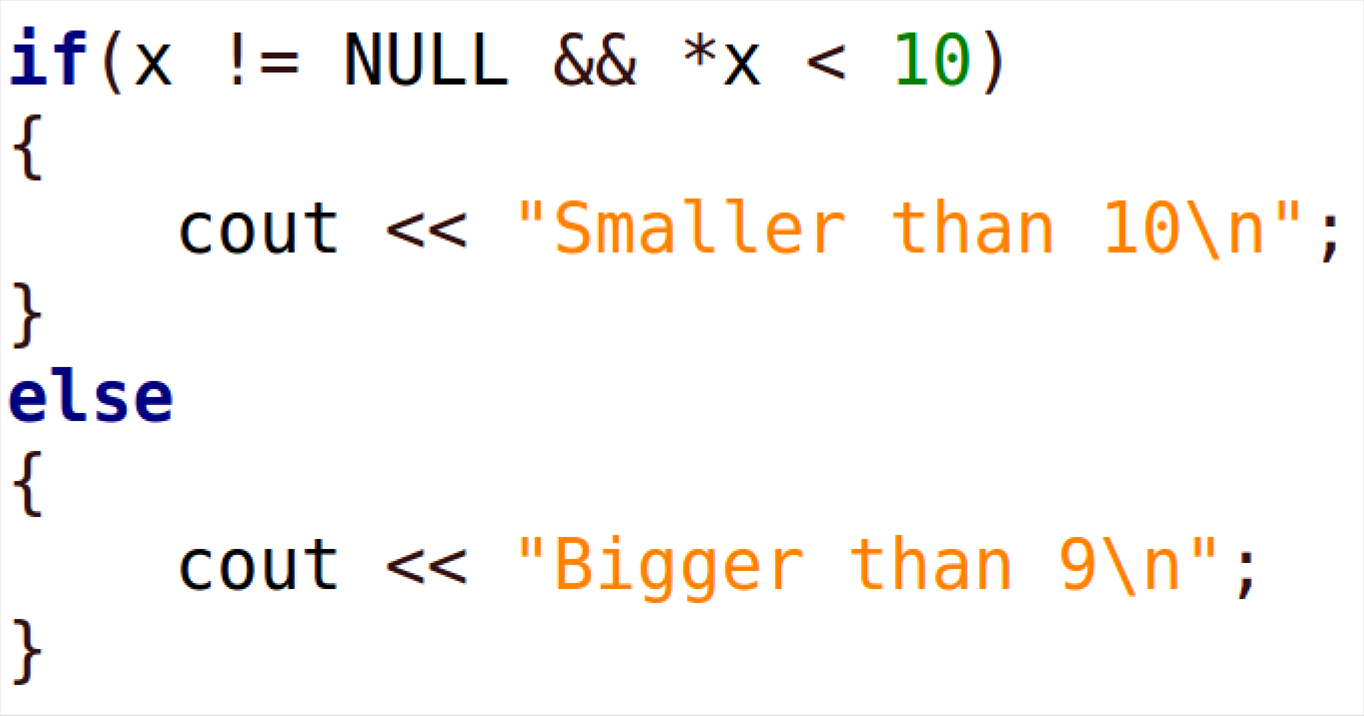
==, >=, <= and != (binary operators)

&& and || (binary operators)

=, +=, -=, *=, /=, %= (binary operators)

if/else

-an else statement needs an associated if
-else/if construct ensures only one block is run
-short circuit evaluation

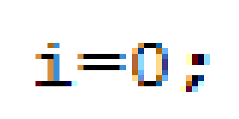


```
if(x != NULL && *x < 10)
{
    cout << "Smaller than 10\n";
}
else
{
    cout << "Bigger than 9\n";
}
```

Loops

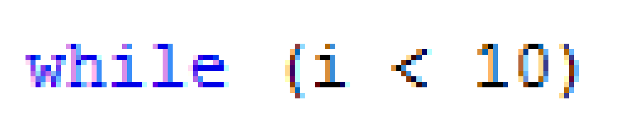
3 parts to any (good) loop:

-Test variable initialized



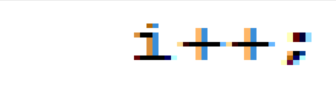
```
i=0;
```

-bool expression



```
while (i < 10)
```

-Test variable updated inside loop



```
    i++;
```

3 types of loops:

while - general purpose

for - known number of iterations (arrays)

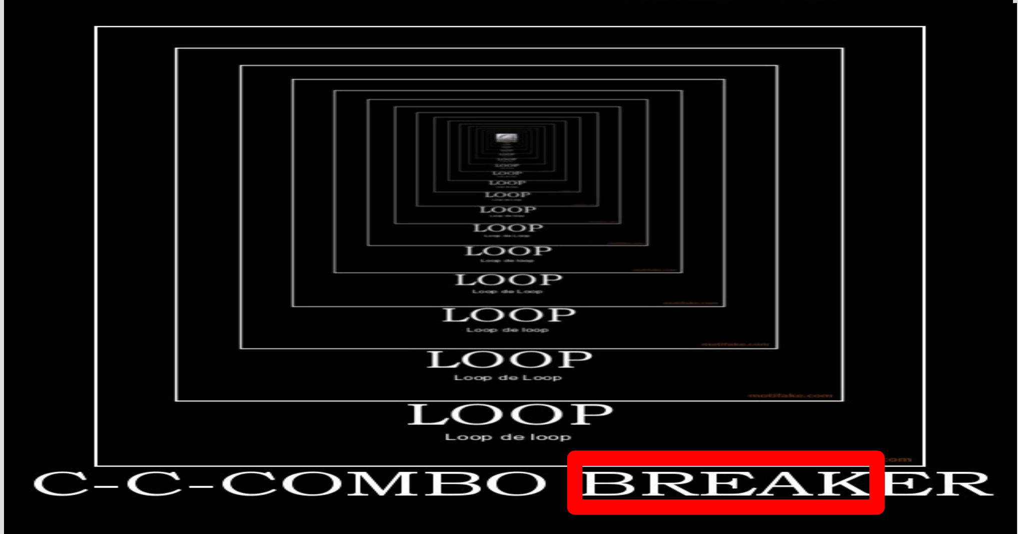
do-while - always run at least once (user input)

continue/break

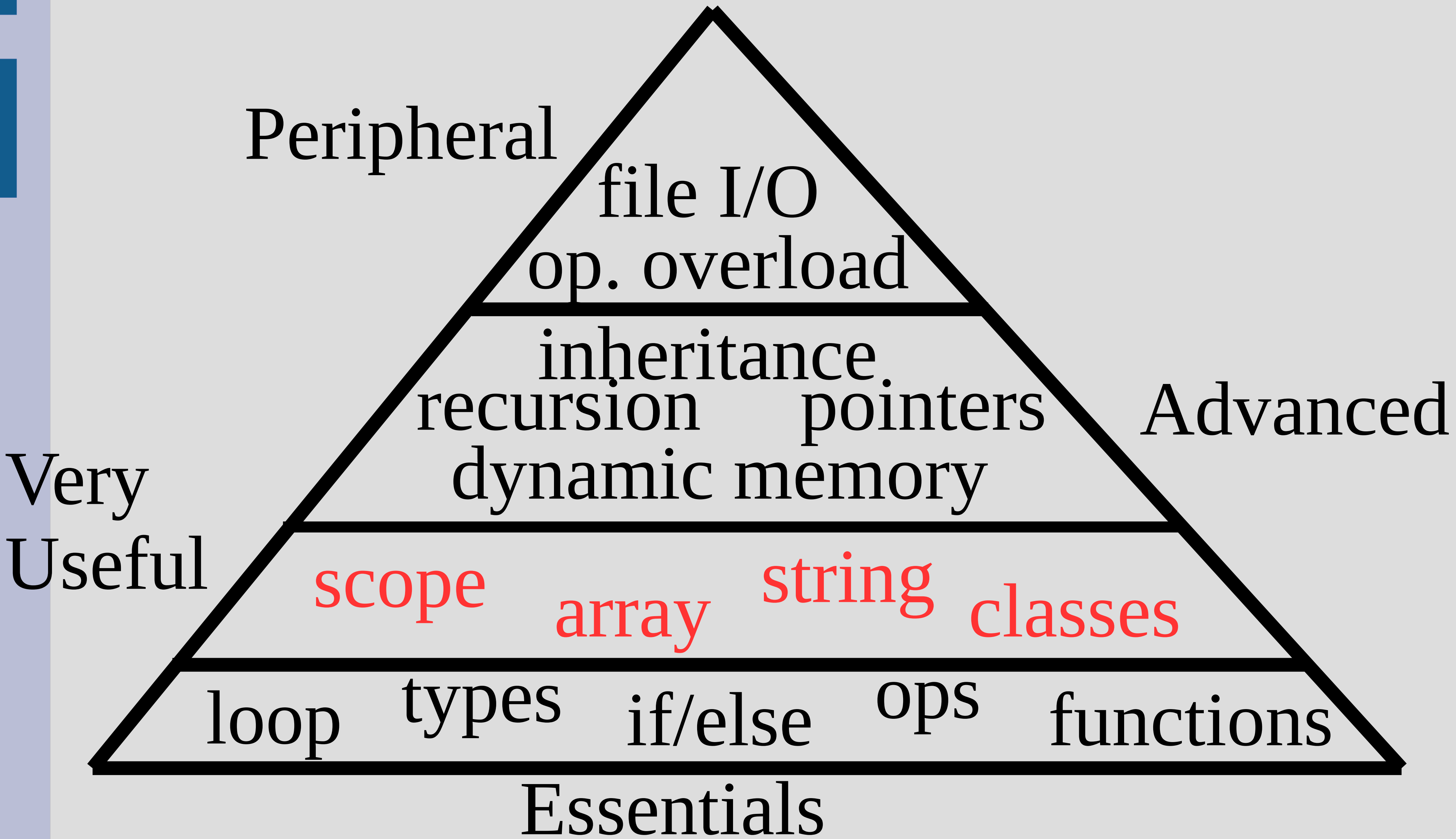
There are two commands that help control loops:

continue tells the loop to start over again (next iteration)

break stops the loop



Review: Overview



Peripheral

file I/O
op. overload

inheritance
recursion pointers
dynamic memory

Advanced

Very
Useful

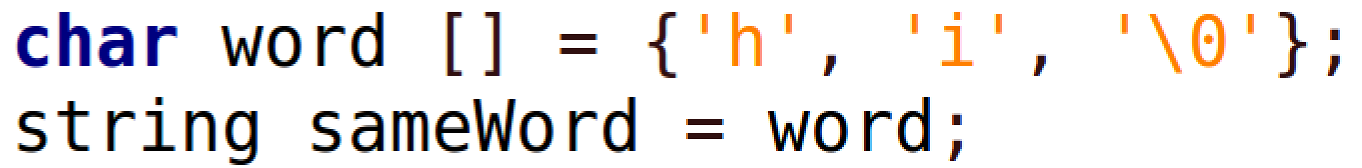
scope array string classes

loop types if/else ops functions

Essentials

C-Strings and strings

c-string uses null character to tell when to end



```
char word [] = {'h', 'i', '\0'};
string sameWord = word;
```

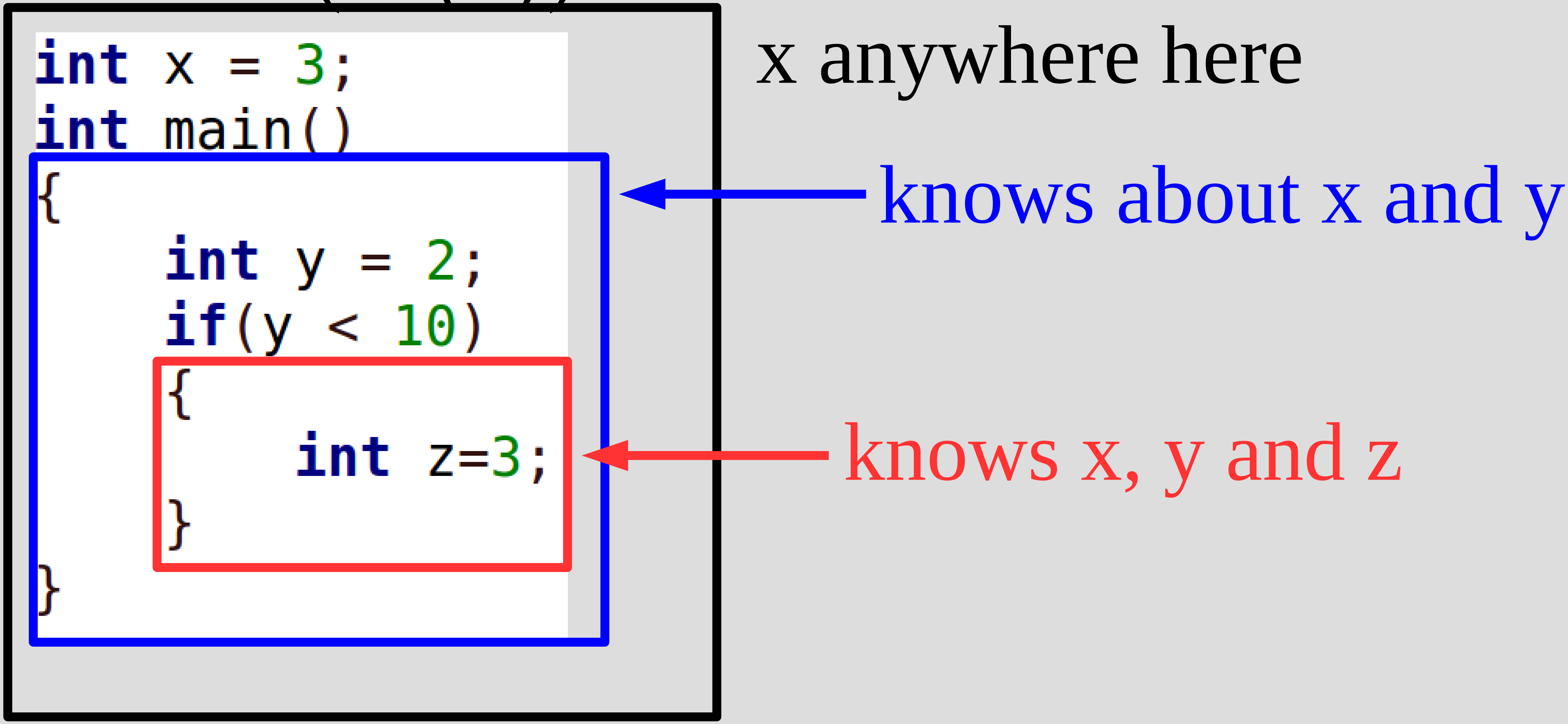
(c++) string is a class (which is a type) and is newer and has many functions:

- find(), substr(), at() or [ ], etc.

Essential for dealing with more than one char at a time

Scope

Variables exist in the braces where it is declared (in { })



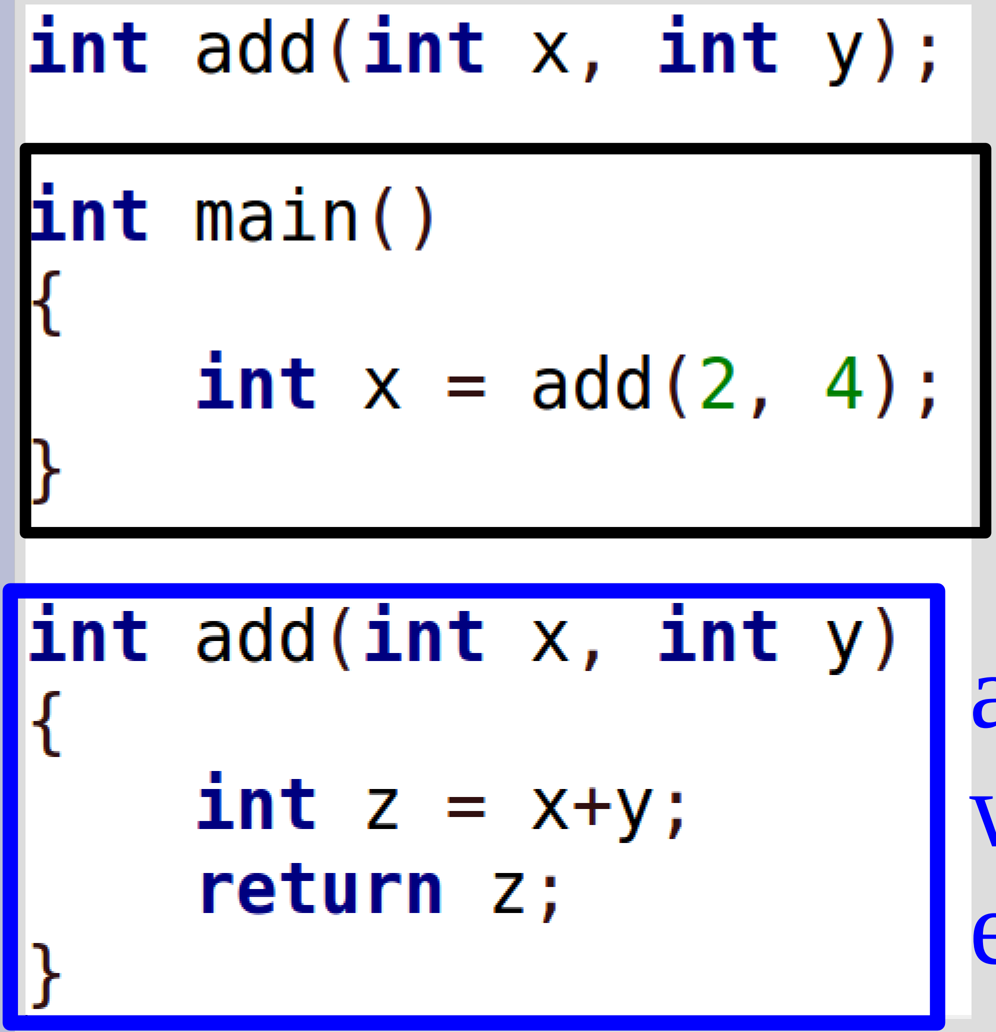
```
int x = 3;
int main()
{
    int y = 2;
    if(y < 10)
    {
        int z=3;
    }
}
```

x anywhere here

knows about x and y

knows x, y and z

Scope

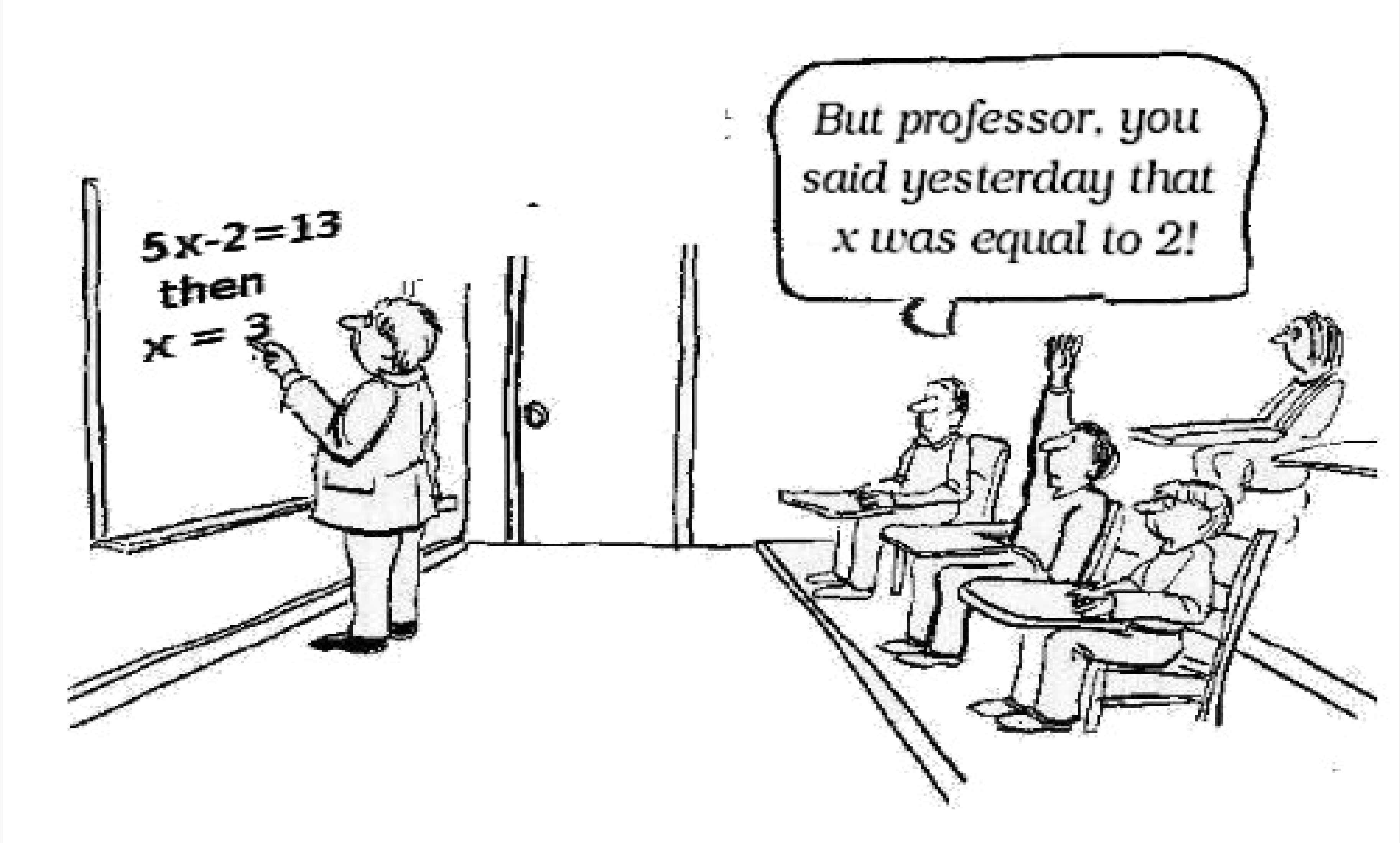


```
int add(int x, int y);

int main()
{
    int x = add(2, 4);
}

int add(int x, int y)
{
    int z = x+y;
    return z;
}
```

main()'s x lives here

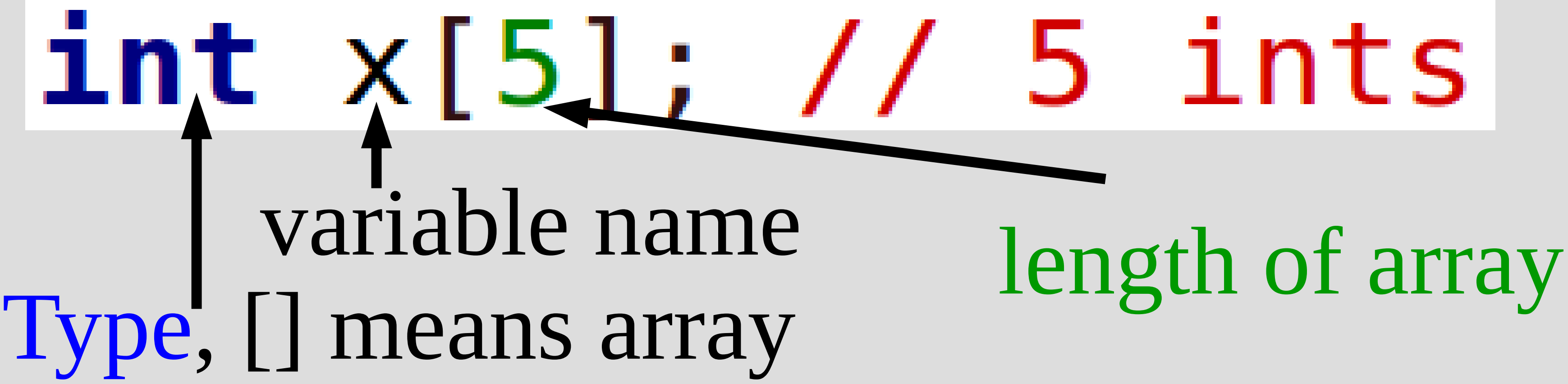
add() has a different x, which along with y and z exist in here

Scope


5x-2=13
then
x = 3
But professor, you said yesterday that x was equal to 2!

Arrays

Arrays store multiple things of the same type



```
int x[5]; // 5 ints
```

variable name

length of array

Type, [] means array

After declaration **any use of [ ]** is interpreted as element indexing

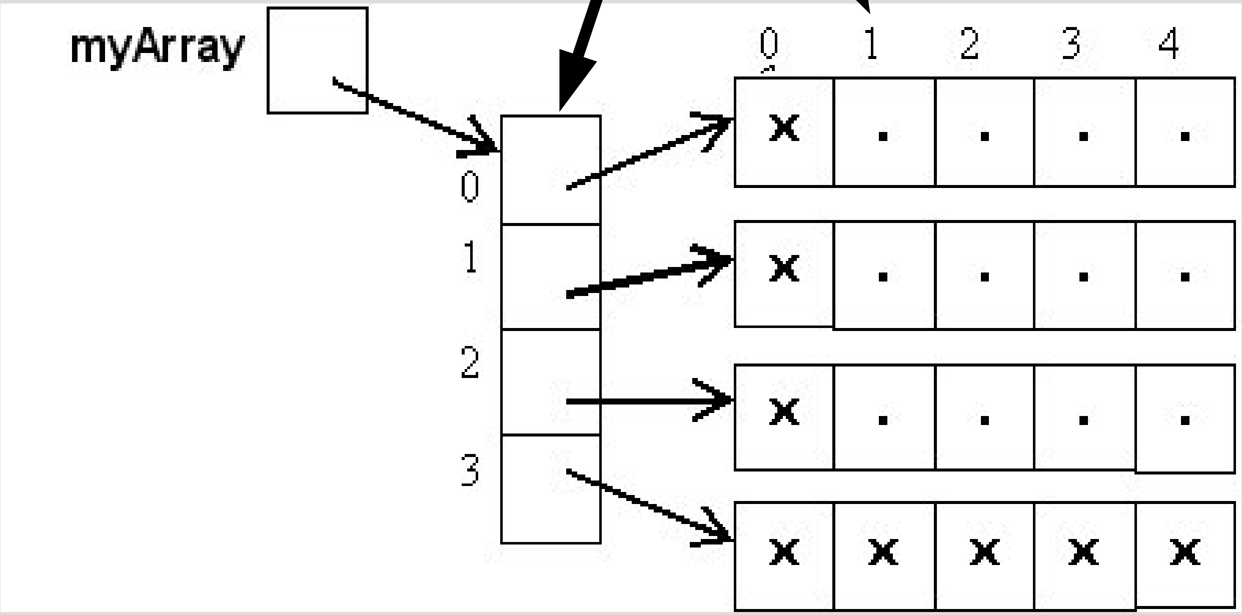
Arrays are memory addresses, shares with functions (cannot call-by-reference)

Multidimensional Arrays

```
string myArray[4][5];
```

four rows

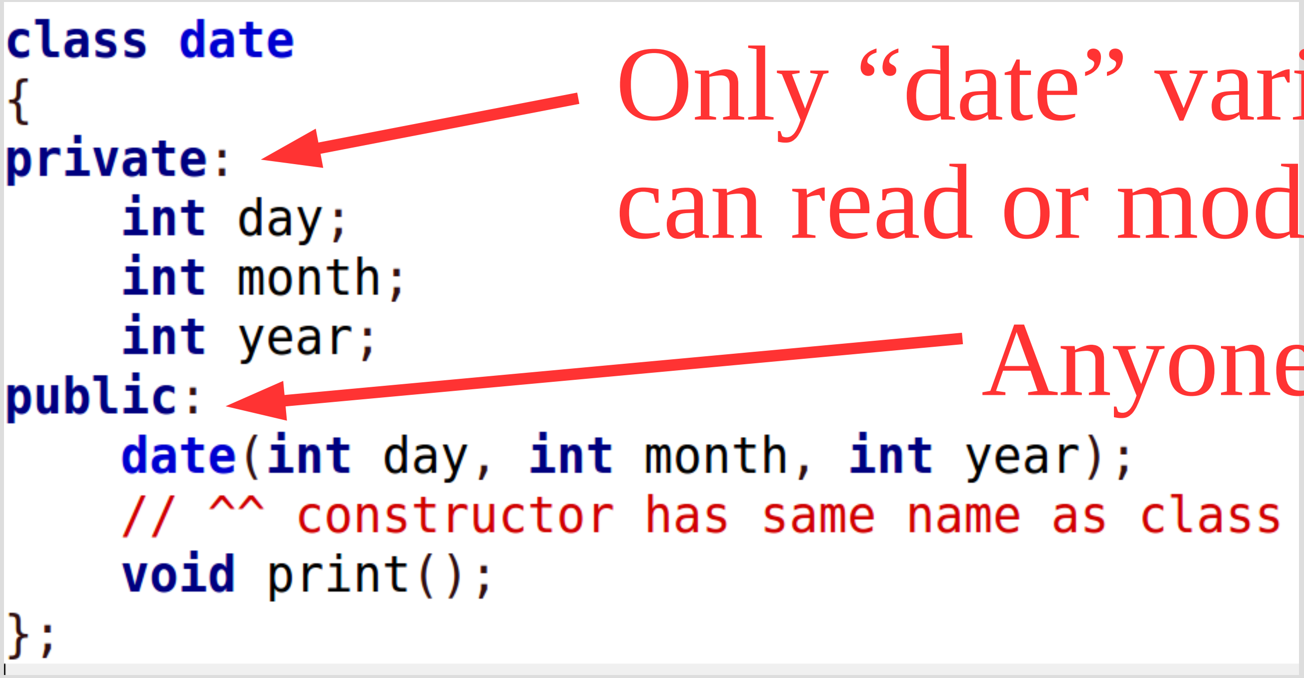
five columns



Must specify (some parts of) size when using as argument in function

Classes

A class is a way to bundle functions and variables (different types) into one logical unit



```
class date
{
private:
    int day;
    int month;
    int year;
public:
    date(int day, int month, int year);
    // ^^ constructor has same name as class
    void print();
};
```

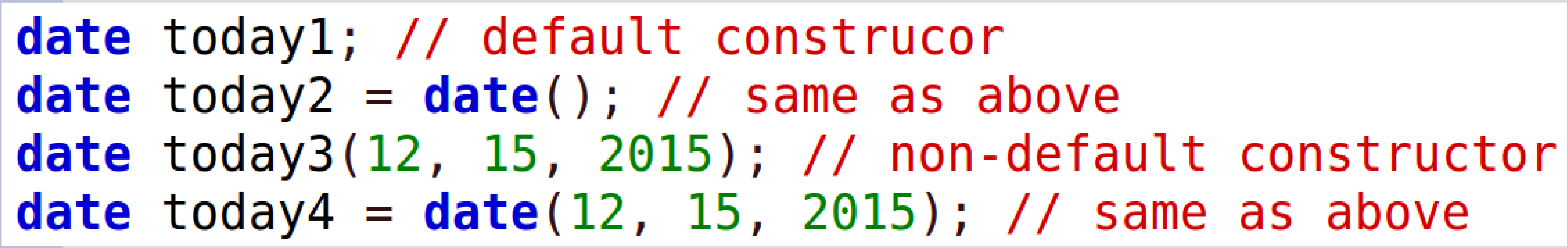
Only “date” variables can read or modify

Anyone can edit/use

Classes are custom made types (like int), that you make and define

Classes

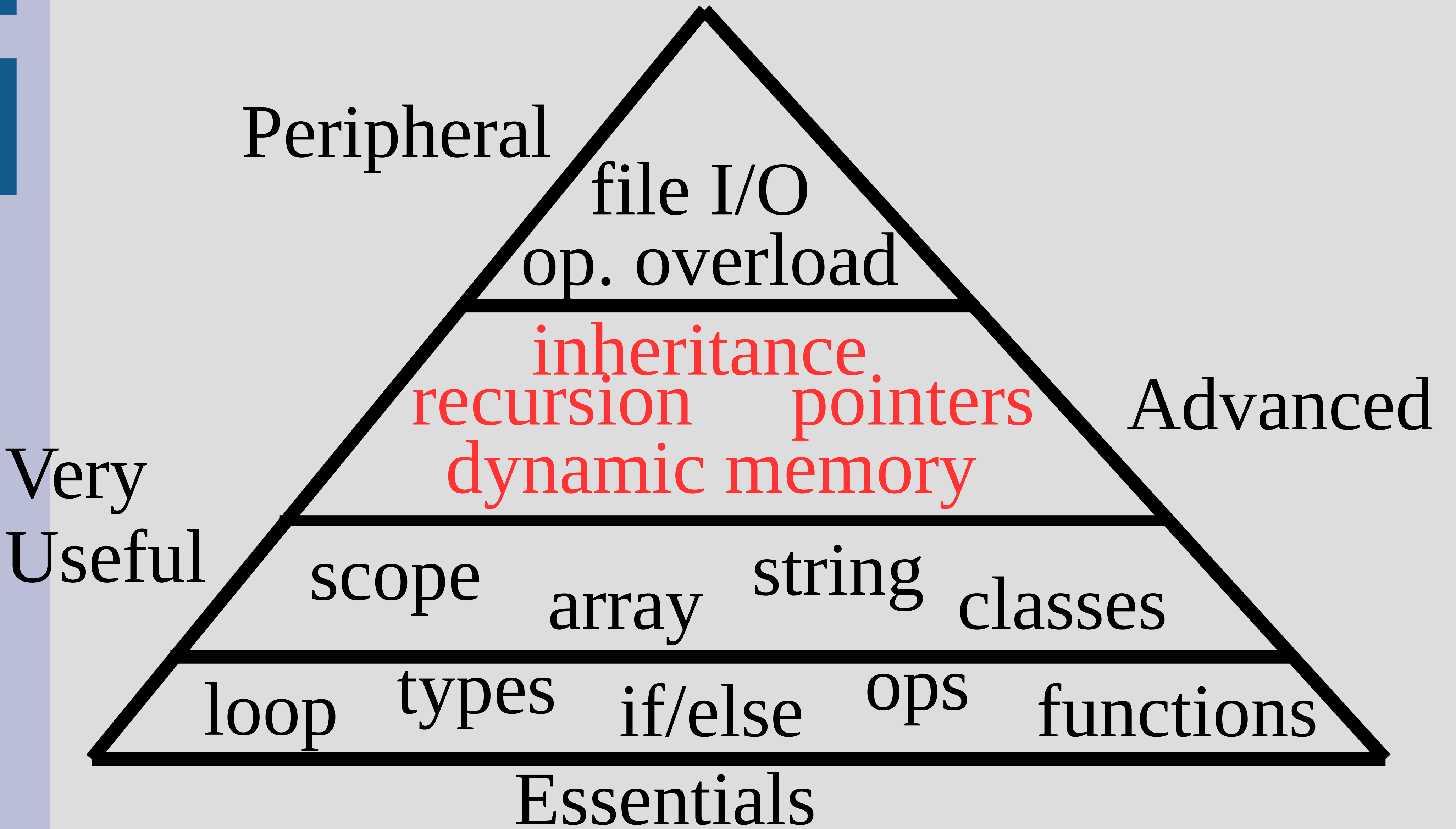
Every time you actually create an object of the class type, you must run a constructor



```
date today1; // default construcor
date today2 = date(); // same as above
date today3(12, 15, 2015); // non-default constructor
date today4 = date(12, 15, 2015); // same as above
```

Constructors should initialize (probably) all variables inside the class

Review: Overview



Recursion

There are two important parts of recursion:

-A stopping case that ends the recursion

-A reduction case that reduces the problem

Identify the problem sub-structure, then move inputs towards the base case

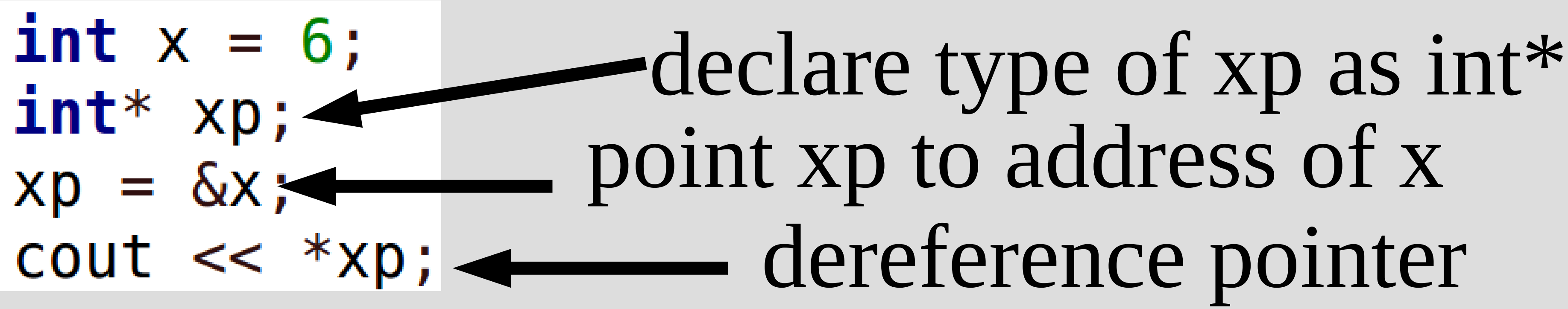
$$F_n = F_{n-1} + F_{n-2},$$

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

You can assume your function works as you want it to (and it will if you do it properly!)

Pointers

A pointer is used to store a memory address and denoted by a * (star!)



```
int x = 6;
int* xp;
xp = &x;
cout << *xp;
```

declare type of xp as int*

point xp to address of x

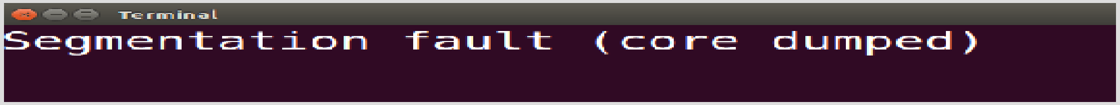
dereference pointer

As arrays, the * on the declaration is special (declares a type only)

Every other use of * will try to go where the variables is pointing to

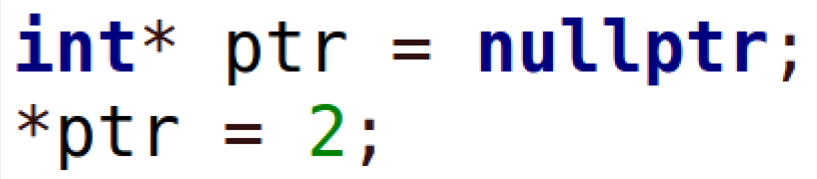
Pointers - nullptr

If you try to go to a place outside your memory, you will seg fault

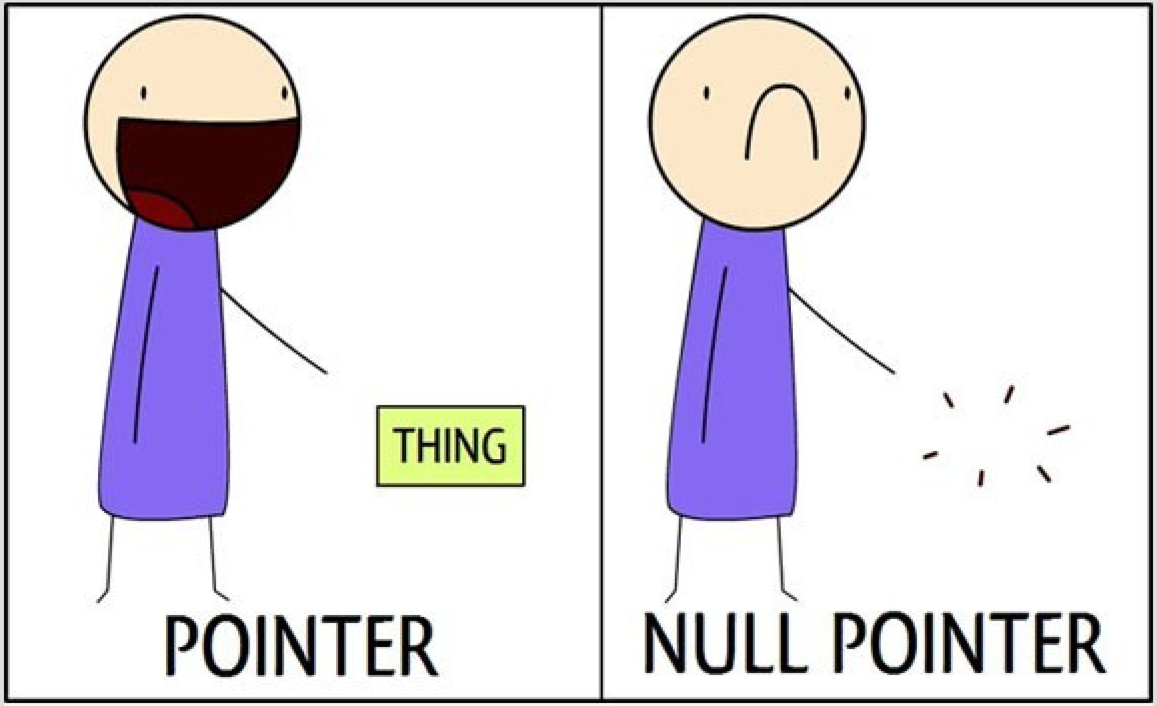


```
Segmentation fault (core dumped)
```

This is especially true with the nullptr (NULL)



```
int* ptr = nullptr;
*ptr = 2;
```

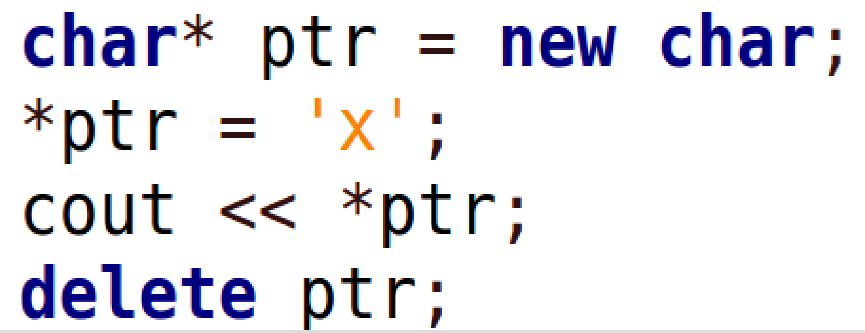


(Typically the values when uninitialized)

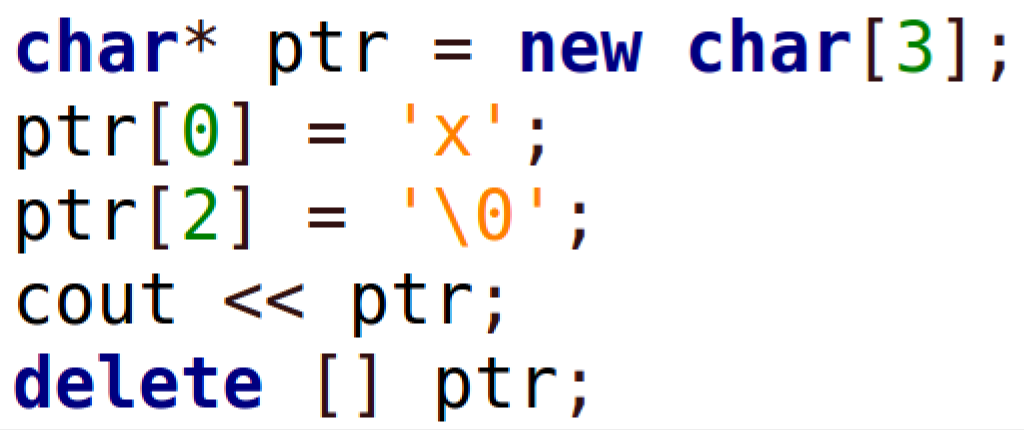
Dynamic memory

Dynamic memory makes variables without names (much as array elements do not have individual names)

Pointers can hold both a single variable or an array of variables:



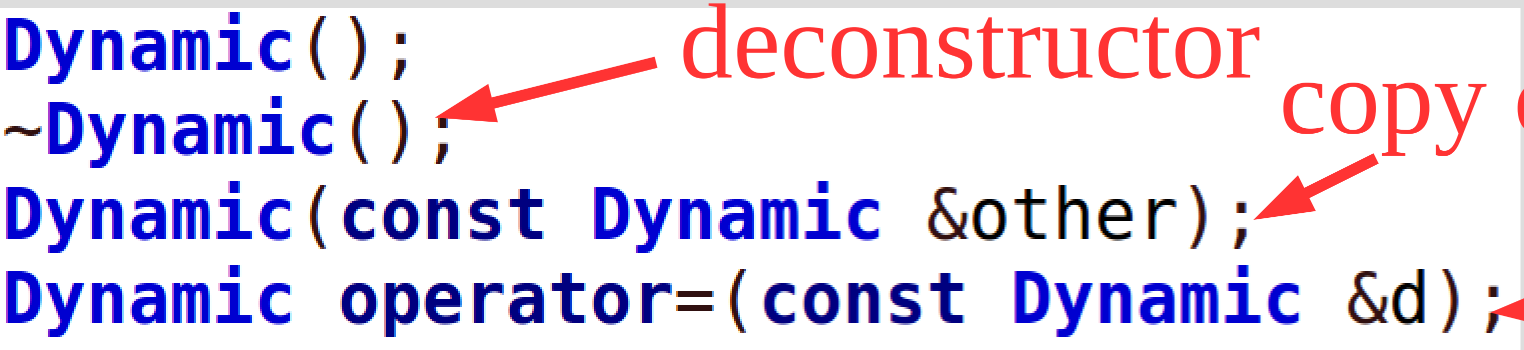
```
char* ptr = new char;
*ptr = 'x';
cout << *ptr;
delete ptr;
```



```
char* ptr = new char[3];
ptr[0] = 'x';
ptr[2] = '\0';
cout << ptr;
delete [] ptr;
```

Dynamic memory in classes

If a variable inside a class uses dynamic memory, we should build a deconstructor (which does the "delete"ing)



```
Dynamic();
~Dynamic();
Dynamic(const Dynamic &other);
Dynamic operator=(const Dynamic &d);
```

deconstructor

copy constructor

operator =

If we need one of these, then we need them all:

-deconstructor

-copy-constructor

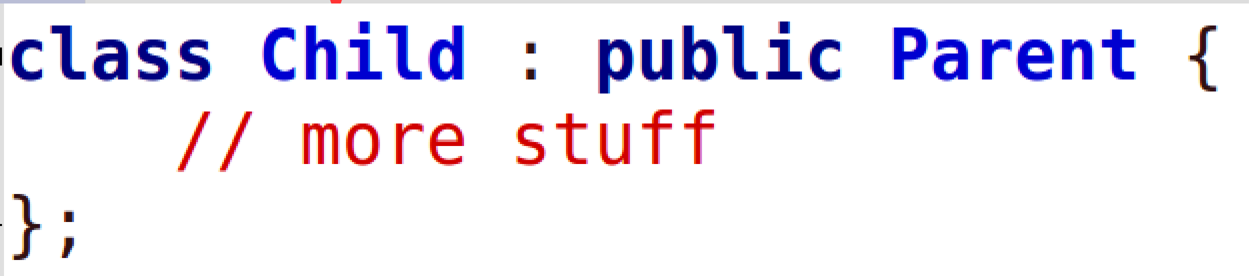
-overload "=" operator

Inheritance

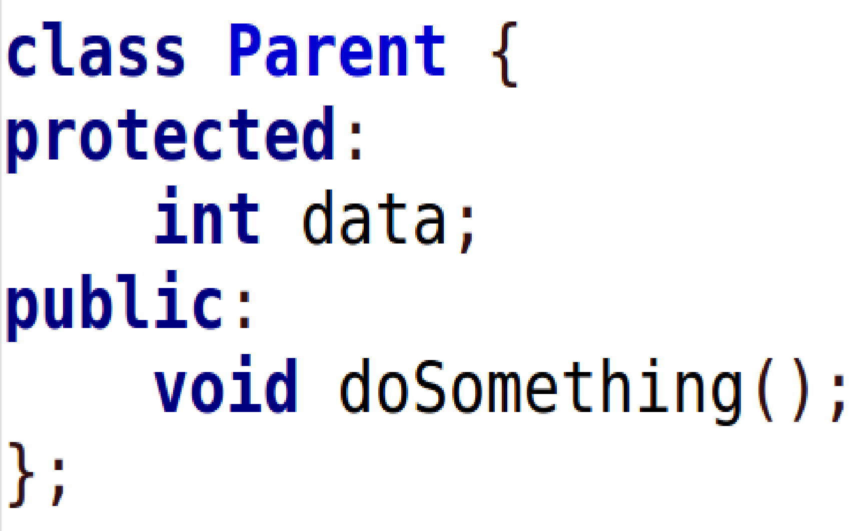
To create create a child class from a parent class, use a : in the (child) class declaration

This shares functions and variables from the parent class to the child

child class parent class



```
class Child : public Parent {
    // more stuff
};
```



```
class Parent {
protected:
    int data;
public:
    void doSomething();
};
```

protected

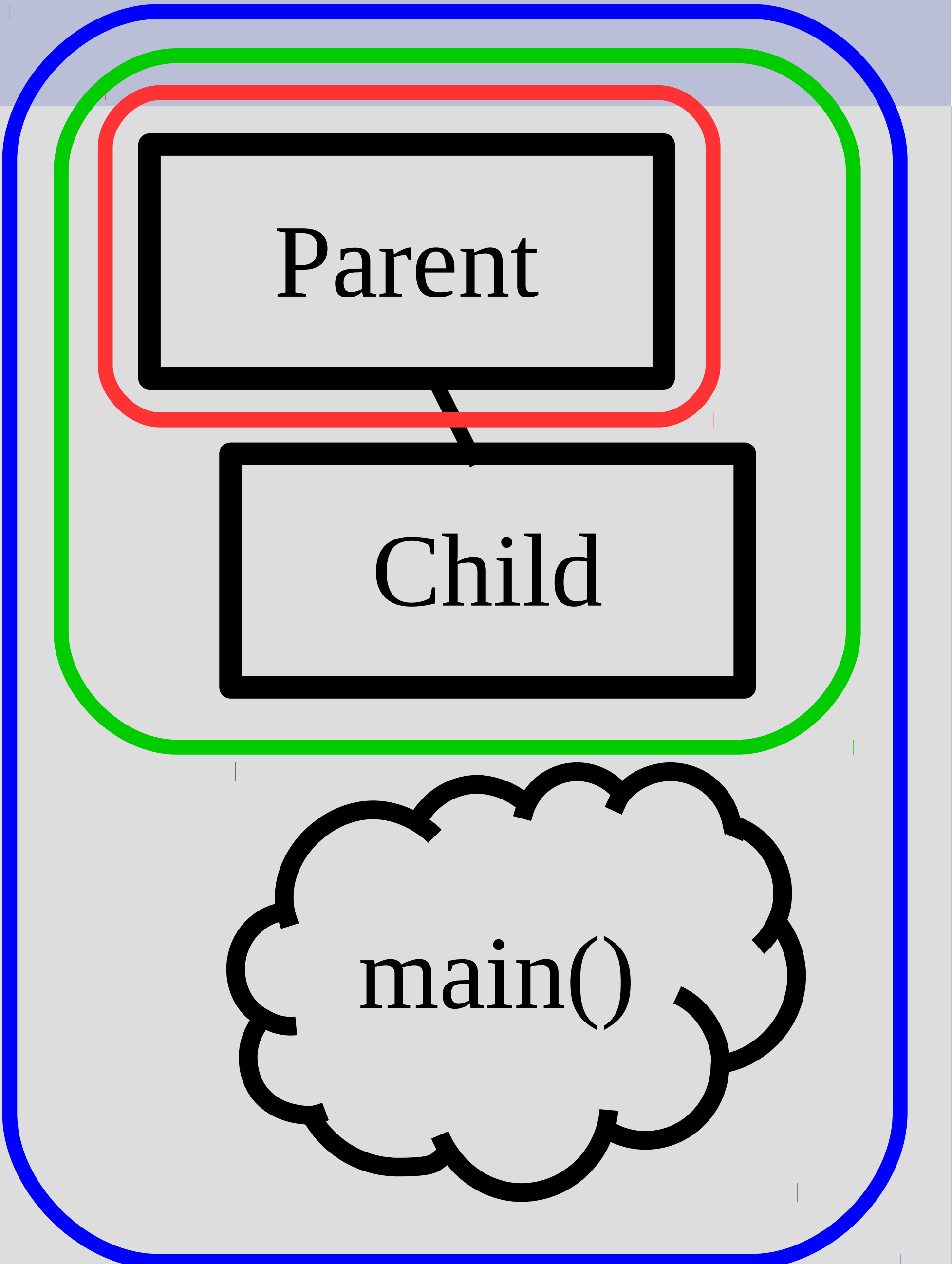
Picture:

Red = private

Green = protected

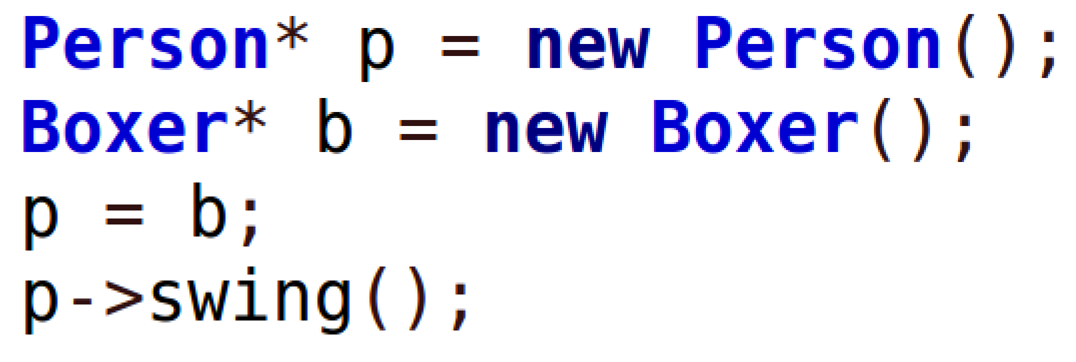
Blue = public

Variables should be either private or protected



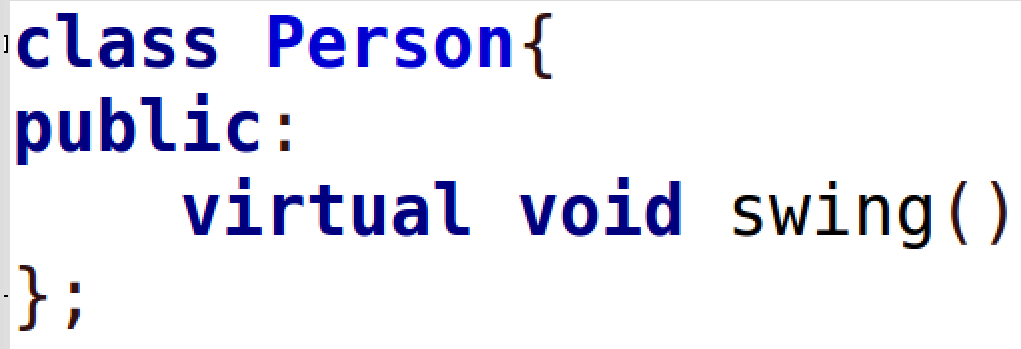
Dynamic binding

Store child as parent, can keep all of child if you use pointers



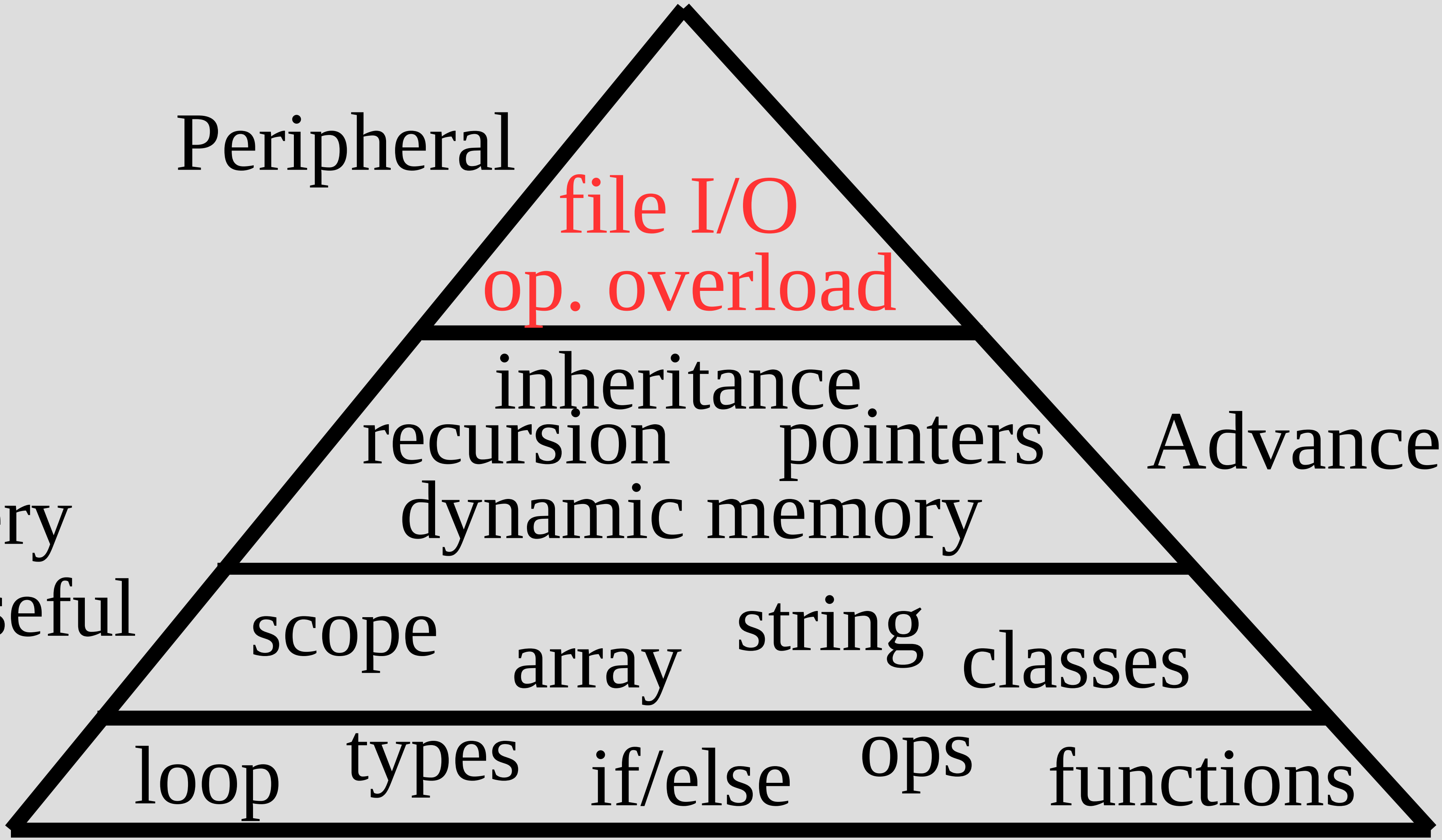
```
Person* p = new Person();
Boxer* b = new Boxer();
p = b;
p->swing();
```

Add virtual to use more appropriate function in pointed object:



```
class Person{
public:
    virtual void swing()
};
```

Review: Overview



Peripheral

file I/O
op. overload

inheritance
recursion pointers
dynamic memory

Advanced

Very
Useful

scope array string classes

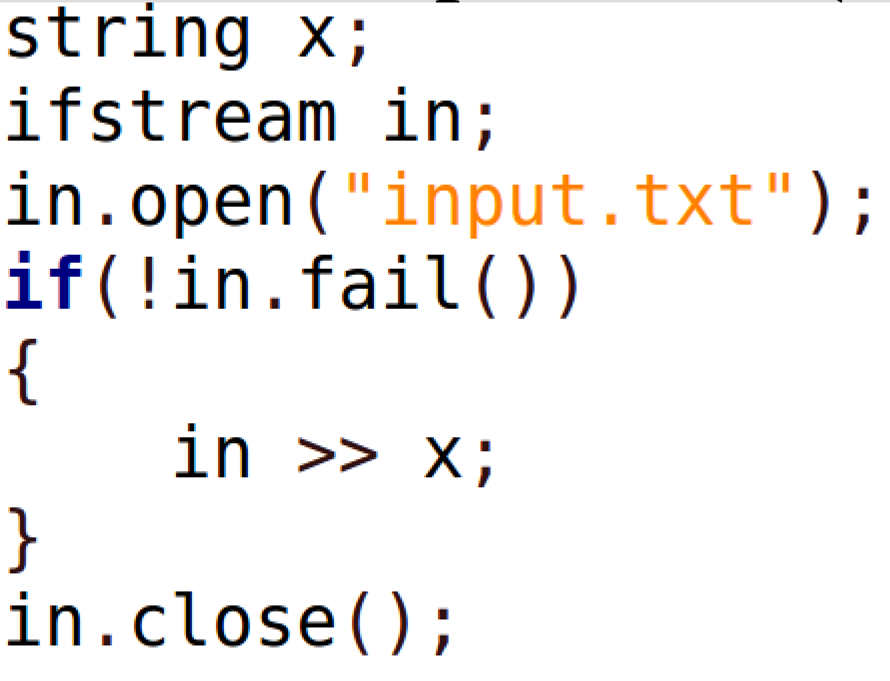
loop types if/else ops functions

Essentials

File I/O

4 steps to file I/O:

Declare, open, use (loop), close



```
string x;
ifstream in;
in.open("input.txt");
if(!in.fail())
{
    in >> x;
}
in.close();
```

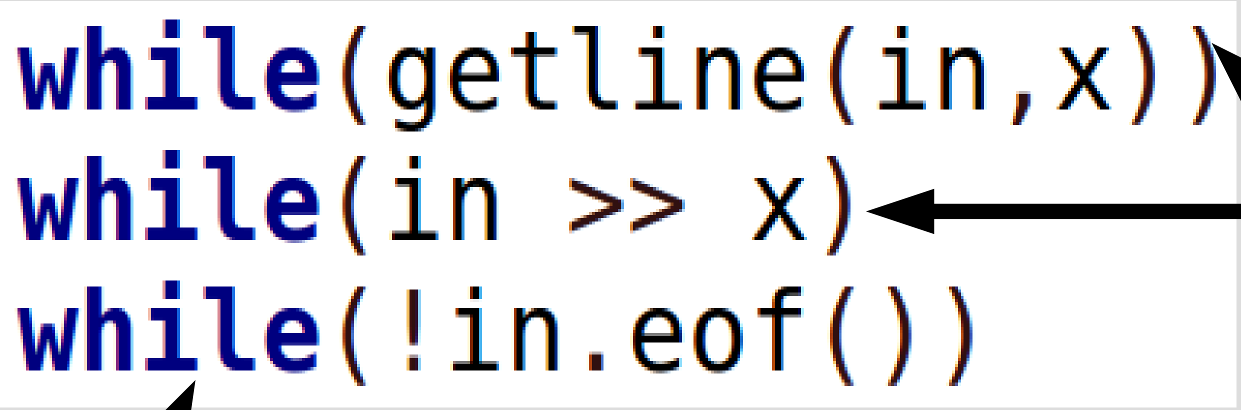
input should check to see if file opened

output overrides file by default

After this point use the variable (“in” above) in place of cin/cout for read/write (respective)

End of file (EOF)

3 ways of looping over whole file (reading)



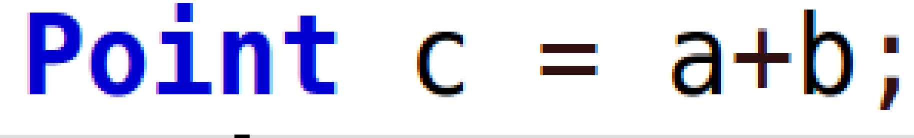
```
while(getline(in,x))
while(in >> x)
while(!in.eof())
```

reads from file

does not read from file (just tells if at end)

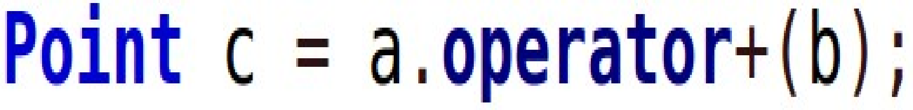
eof() will not be true **until** a read fails, so must check for eof() immediately after reading

Operator overloading

Will convert: 

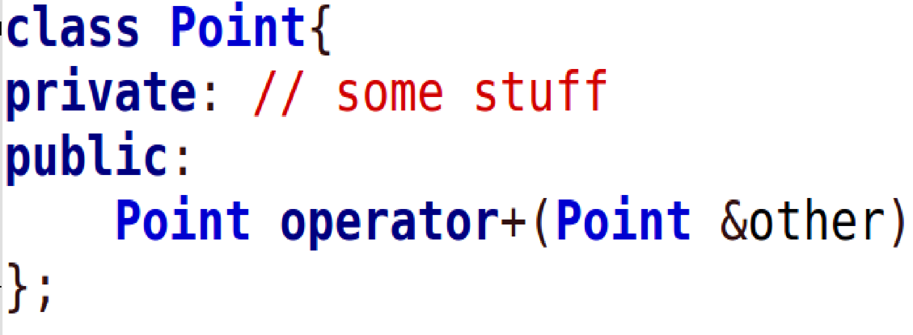
```
Point c = a+b;
```

function in class:



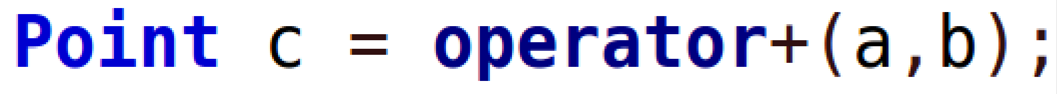
```
Point c = a.operator+(b);
```

... defined as...



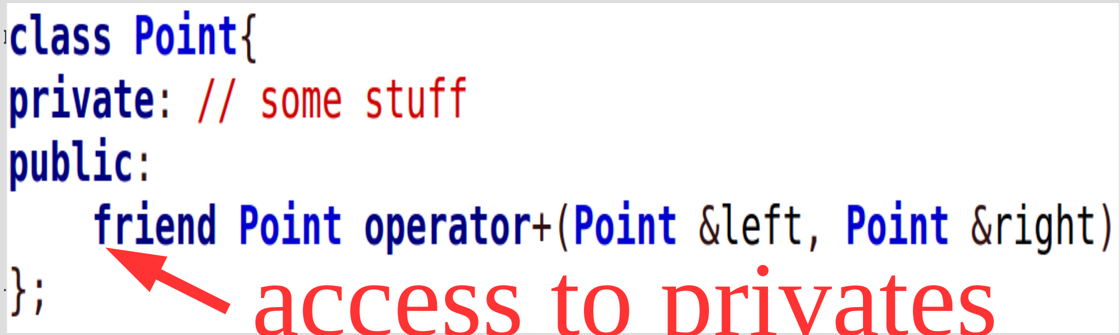
```
class Point{
private: // some stuff
public:
    Point operator+(Point &other)
};
```

friend function:



```
Point c = operator+(a,b);
```

... defined as...



```
class Point{
private: // some stuff
public:
    friend Point operator+(Point &left, Point &right)
};
```

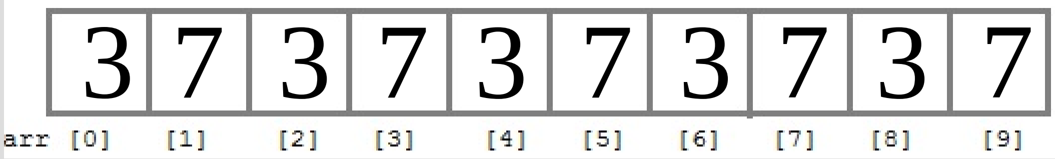
access to privates

Use friend over in-class version if order matters (i.e. “cout << c” not “c << cout”)

Problems

Suppose you want a length 10 array, but all the odd indexes are represented by the same number

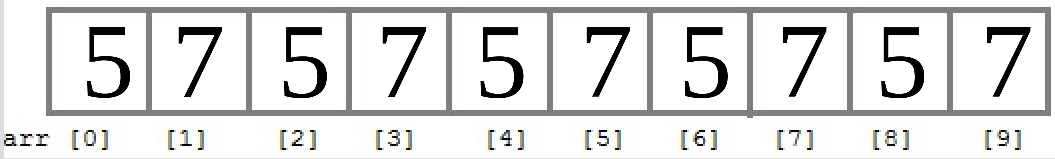
This is also true for the even numbers:



	3	7	3	7	3	7	3	7	3	7
arr	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

(picture not quite accurate)

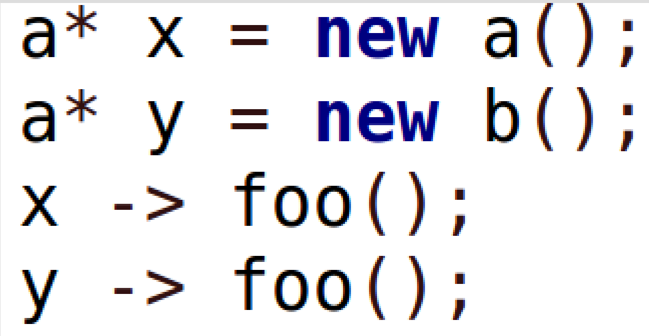
change x[0] to 5:



	5	7	5	7	5	7	5	7	5	7
arr	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Problems

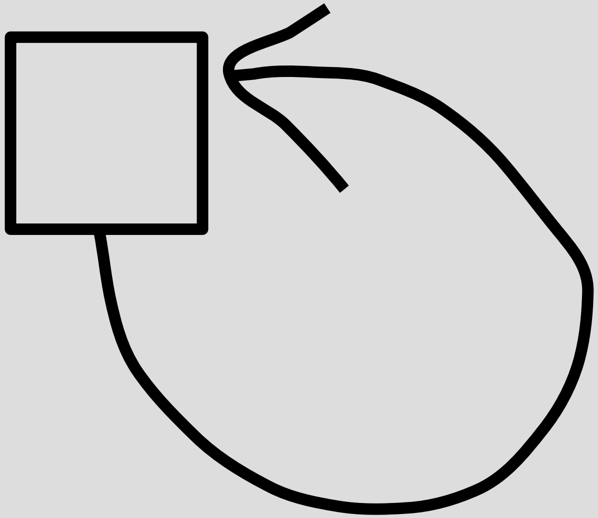
Write some code to make the lines below syntactically correct and cout different things:



```
a* x = new a();
a* y = new b();
x -> foo();
y -> foo();
```

Problems

Can you make a pointer point to itself? Why or why not?



Problems

Suppose there exists a “seat” class

Write the “classroom” class with a constructor that takes in an integer and makes a dynamic array of that many seats

What else does the classroom class need to have?