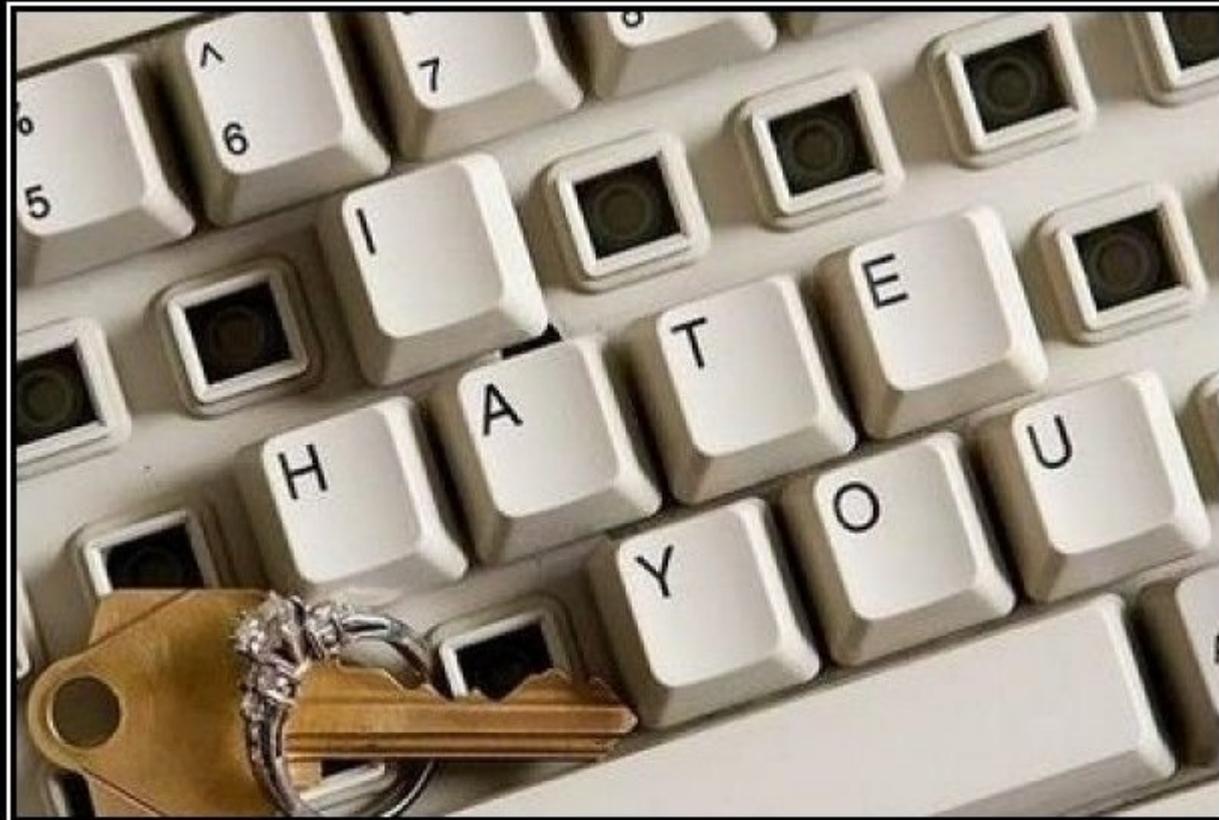


# Review



**DEAR JOHN**

Because sending a text message or email is so impersonal.

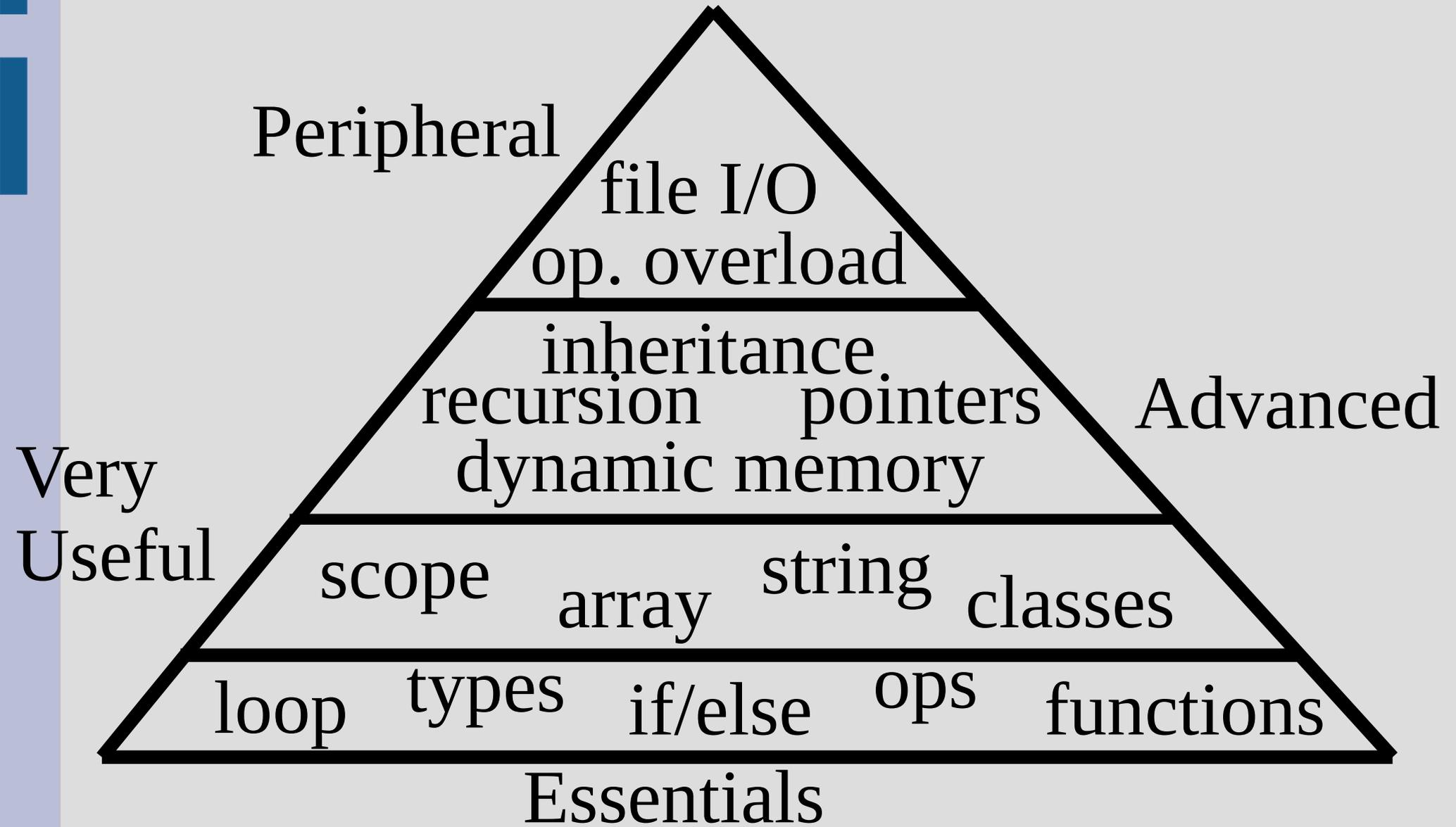
# Final exam

Final exam will be 11-12 problems, drop any 2

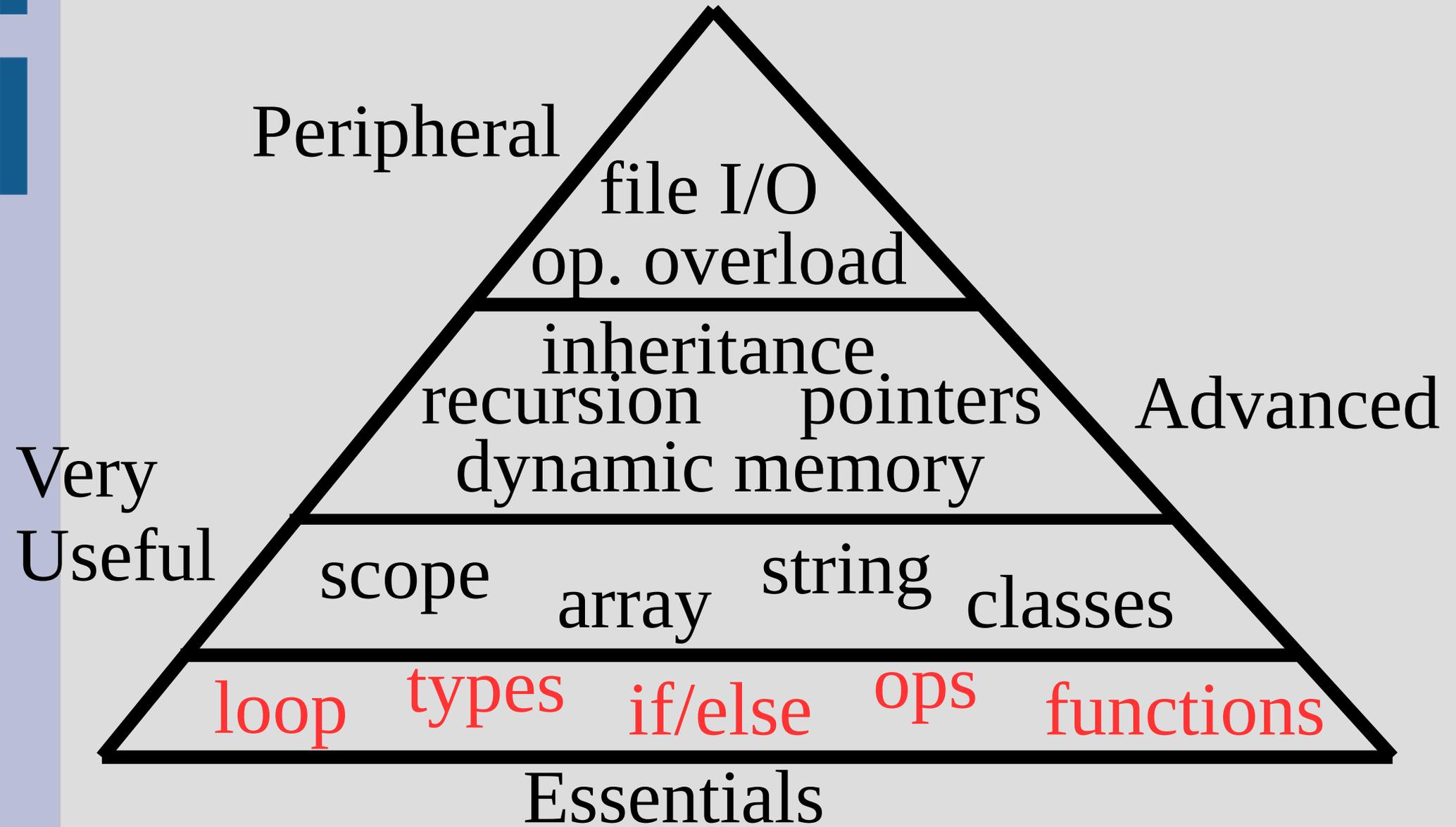
Cumulative up to and including week 13  
(emphasis on weeks 10-13: classes & pointers)

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

return type

function header

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

parameters (order matters!)

```
return x+y;
```

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)

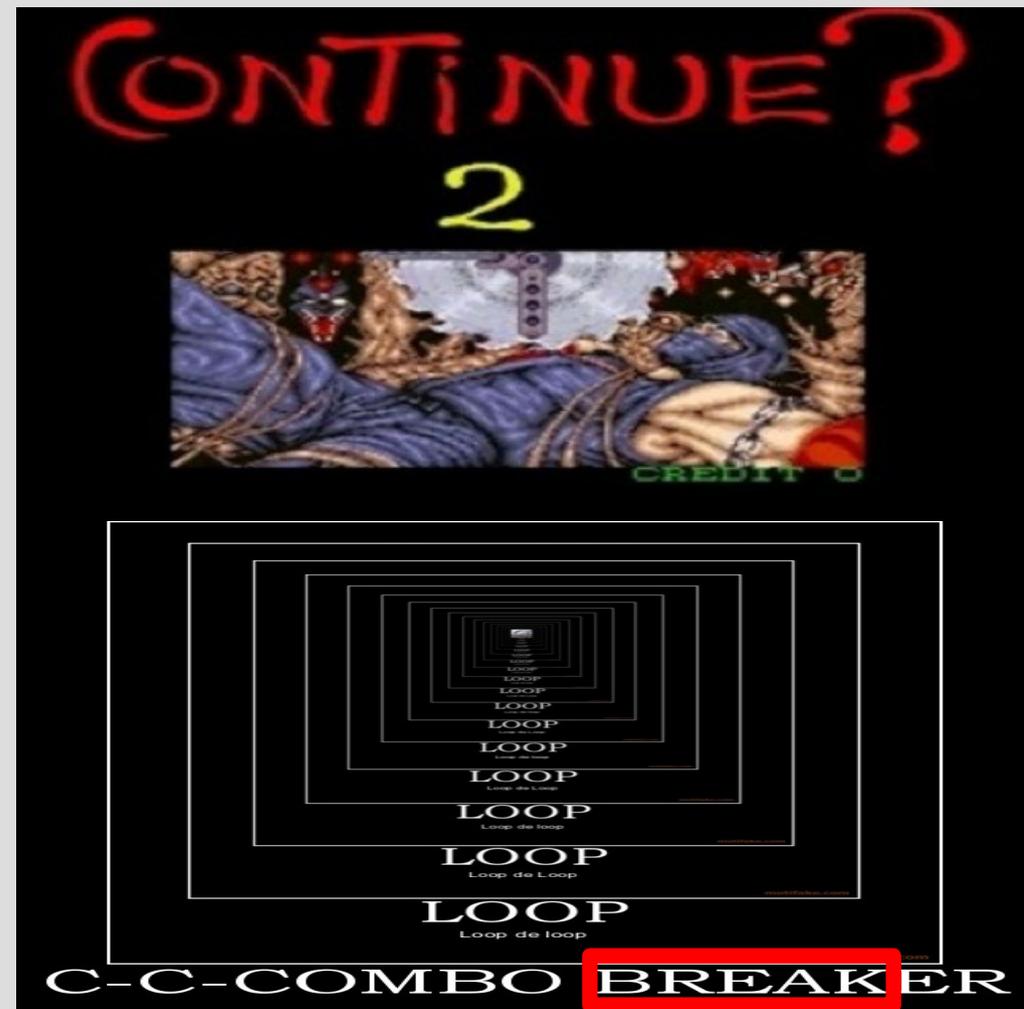
do-while - always run at least once (user input)

# continue/break

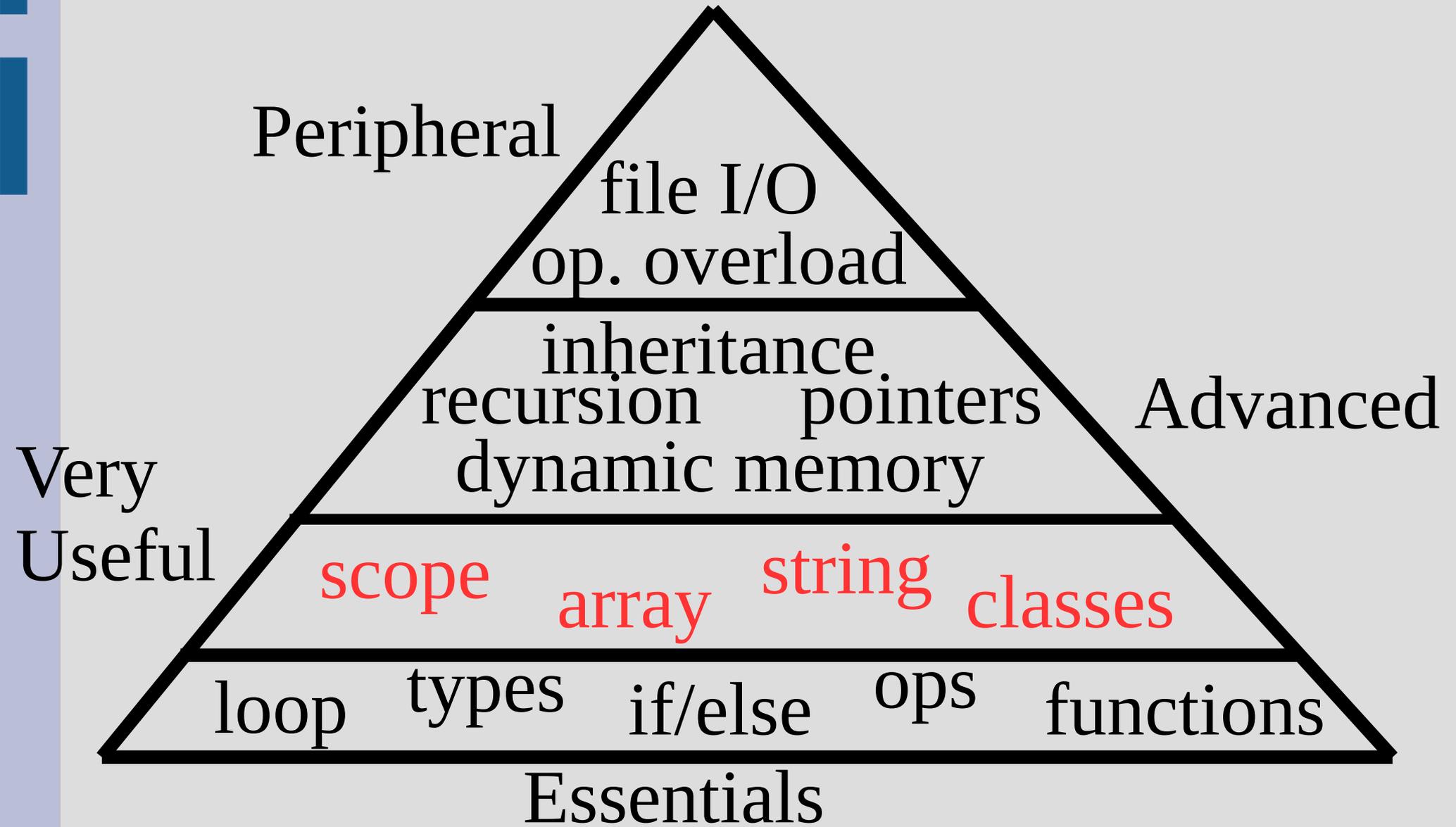
There are two commands that help control loops:

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

break stops the loop



# Review: Overview



# 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);  
}
```

main()'s x lives here

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

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

# Scope



# Arrays

Arrays store multiple things of the same type

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

**Type**, [] means array



length of array

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

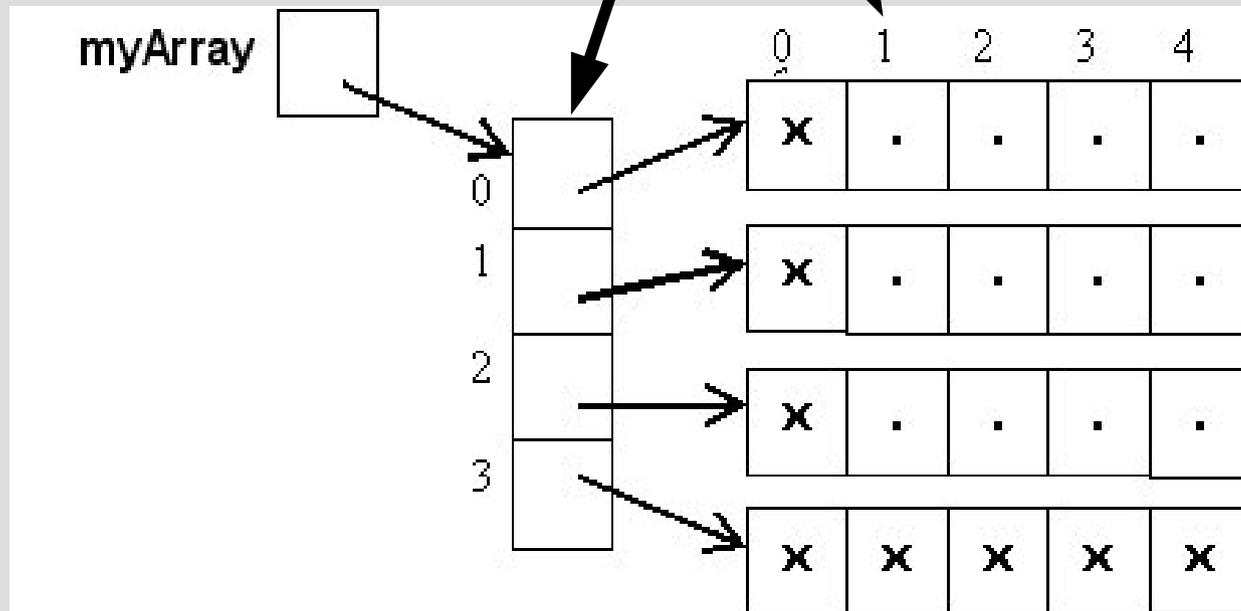
Arrays are memory addresses, shares with functions (cannot call-by-reference)

# 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

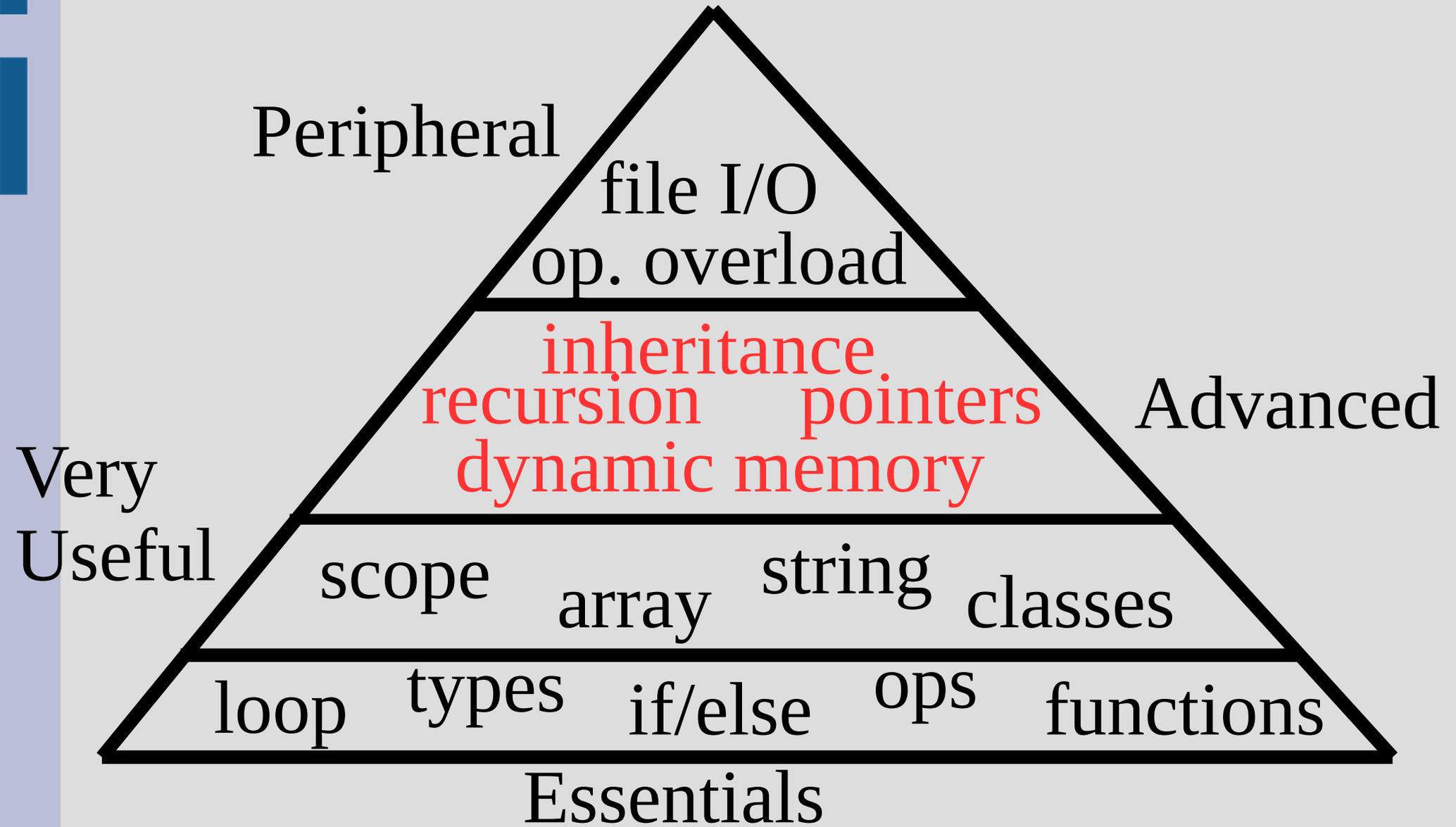
# Classes

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, ...

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

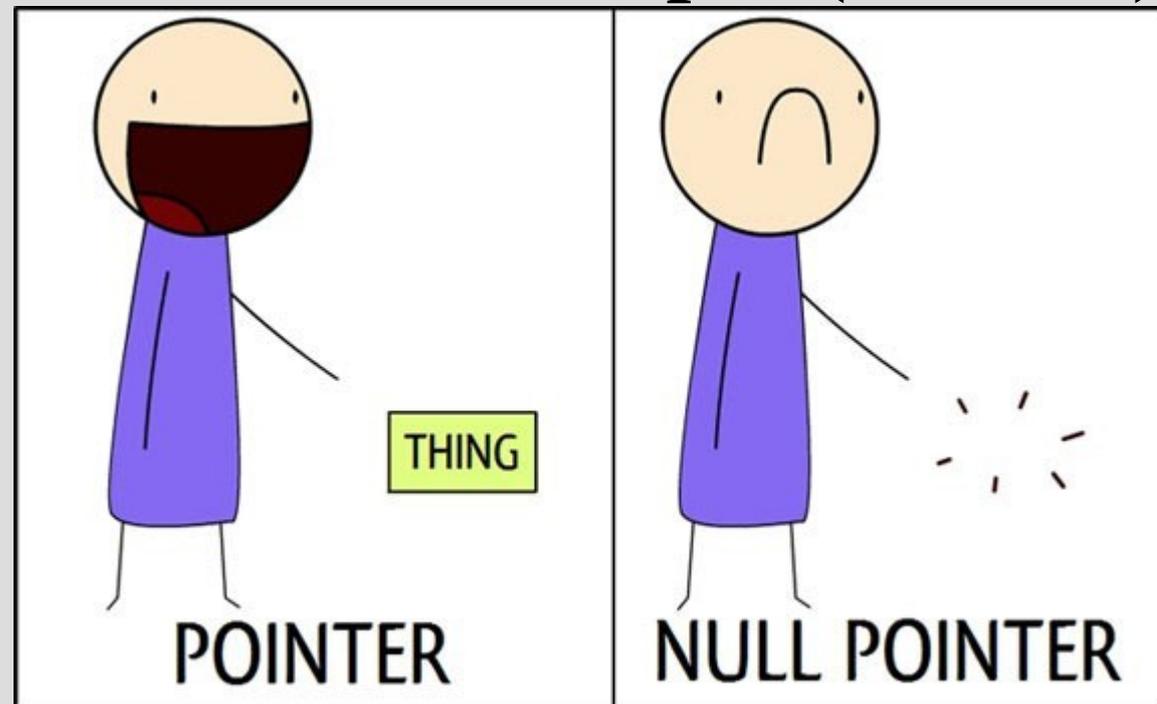
If you try to go to a place outside your memory, you will seg fault

```
Terminal  
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 destructor (which does the “delete”ing)

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

destructor  
copy constructor  
operator =

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

- destructor
- copy-constructor
- overload “=” operator

# Inheritance

To 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

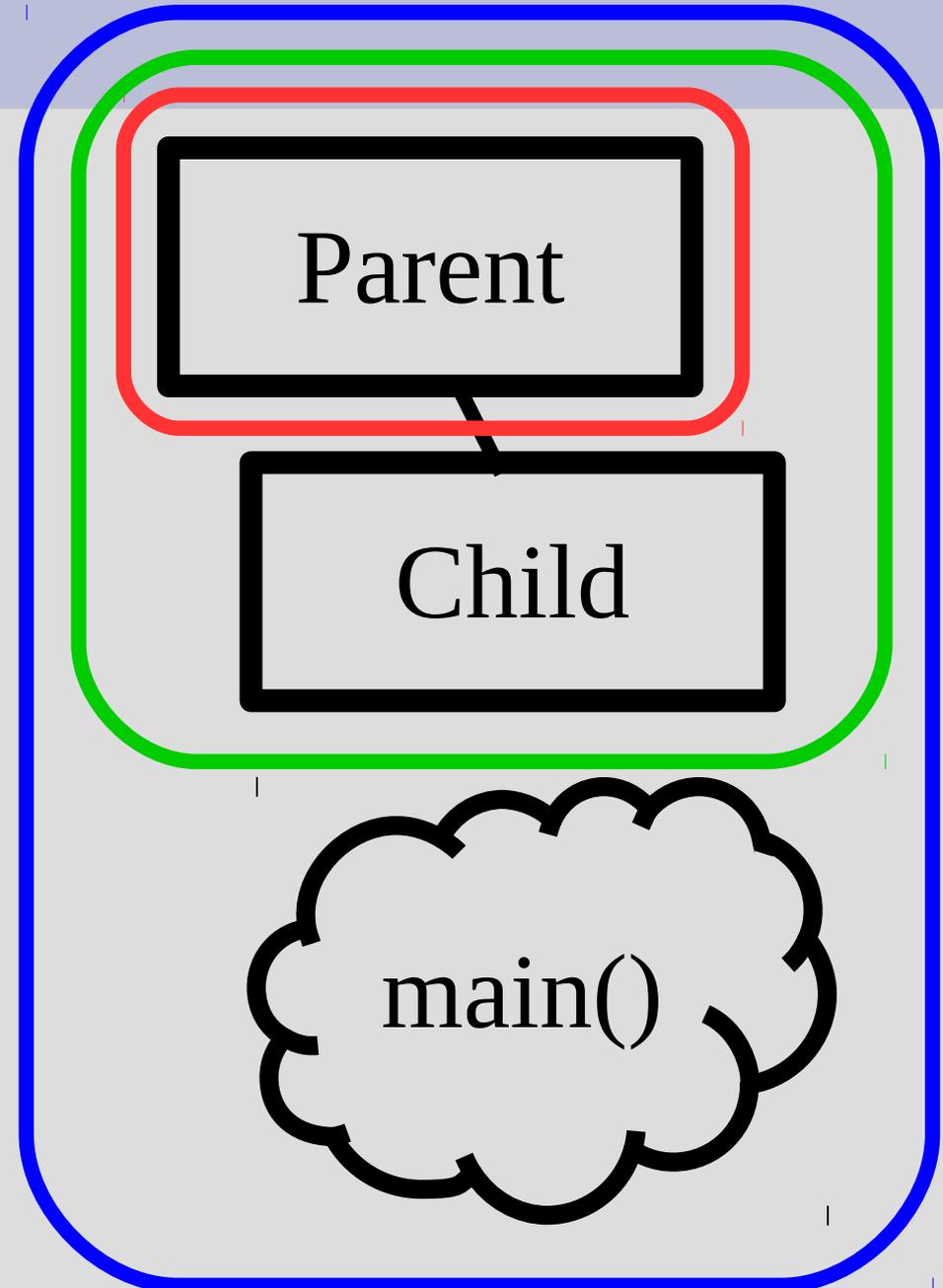
Picture:

Red = private

Green = protected

Blue = public

Variables should be either **private** or **protected**



# Dynamic binding

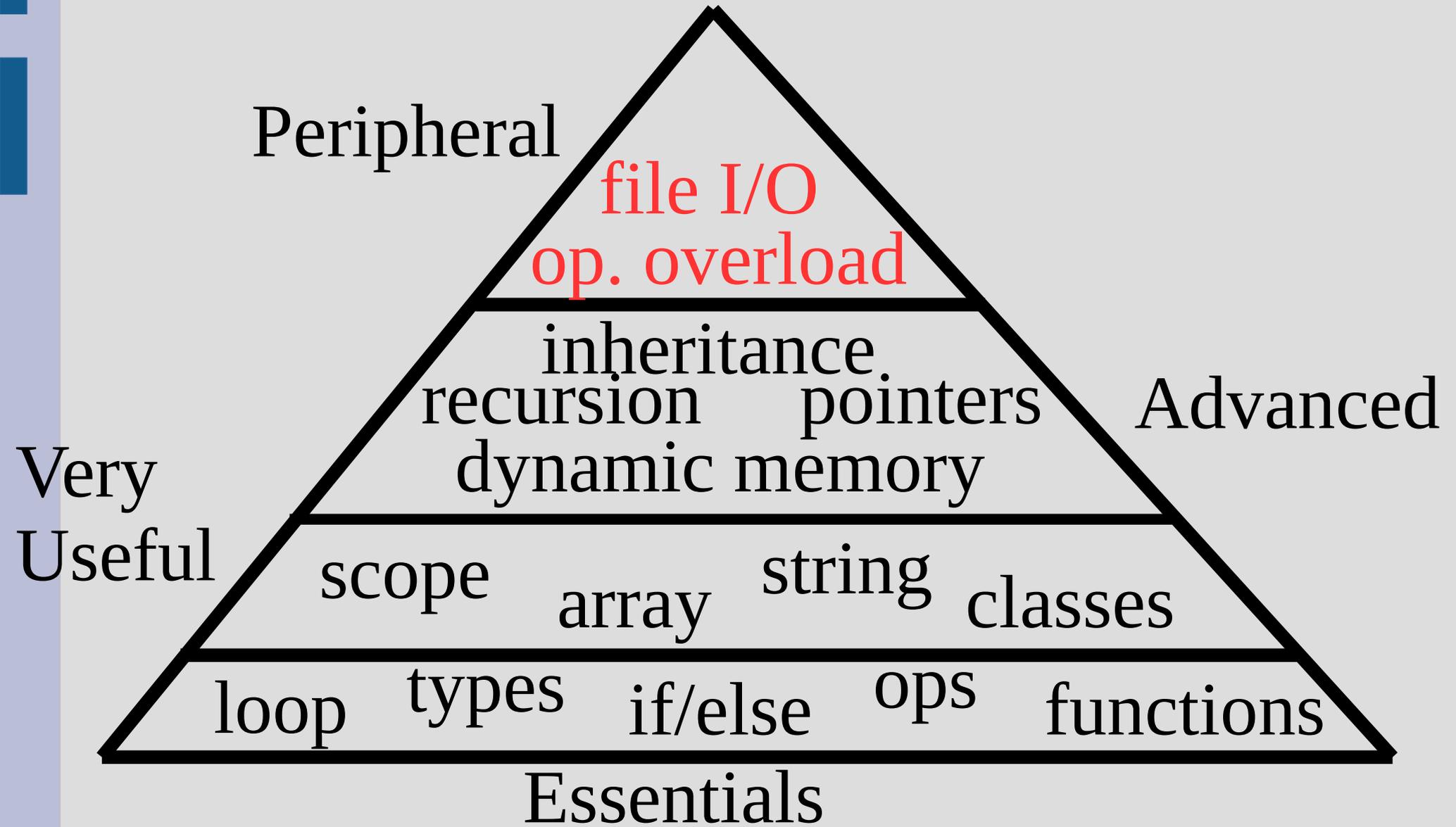
Store child as parent, can keep all of child if you use pointers

```
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



# 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]

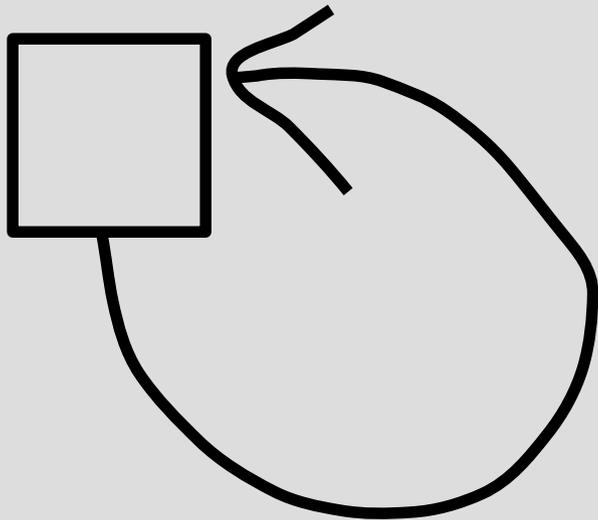
# Problems

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?

# The End

**TAKE BASIC CODING COURSE**

**EXPERT PROGRAMMER**