From C to C++
A Very Crash Course

Some New Features in C++

- Declare variables anywhere
- Input output with cin/cout
- Use new and delete
  - Forget about malloc and free
- Pass by reference
- Function overloading
- OOP!!!!

“using namespace std;”

```
#include <iostream>
using namespace std;

const double PI = 3.14159;

main () {
  int radius;

  cout << "Enter a radius " ;
  cin  >> radius;

  double area = PI * radius * radius;
  double circumference = 2 * PI * radius;

  cout << "Area is " << area << endl;
  cout << "Circumference is " << circumference << endl;
}
```

Declare variables anywhere

```
#include <iostream>
using namespace std;

const double PI = 3.14159;

main () {
  int radius;

  cout << "Enter a radius " ;
  cin  >> radius;

  double area = PI * radius * radius;
  double circumference = 2 * PI * radius;

  cout << "Area is " << area << endl;
  cout << "Circumference is " << circumference << endl;
}
```

C++: Input & Output

C

```c
#include <stdio.h>

int main() {
  printf("Hello World!\n");
  return 0;
}
```

C++

```cpp
#include <iostream>
using namespace std;

int main() {
  cout << "Hello World!" << endl;
  return 0;
}
```

C++: Input & Output

C++

```
#include <iostream>
using namespace std;

int main() {
  cout << “Is the answer of 1 + 1 is" << 1 + 1 << ”?” << endl;
  return 0;
}
```

Cin, cout

```
#include <iostream>
using namespace std;

const double PI = 3.14159;

main () {
  int radius;

  cout << "Enter a radius " ;
  cin  >> radius;

  double area = PI * radius * radius;
  double circumference = 2 * PI * radius;

  cout << "Area is " << area << endl;
  cout << "Circumference is " << circumference << endl;
}
```

Memory Allocation

```
#include <iostream>
using namespace std;

int main() {
  int num;
  double *student_mark;
  cout << “How many students do you have?" << endl;
  cin >> num;
  student_mark = new double [num]
  for (int i = 0; i< num; i++)
    cout << “Enter the mark of Student “ << i + 1 << “: “;
    cin >> student_mark[i];
  }
  // do something about the mark
  delete [] student_mark;
  return 0;
}
```

Pass by Values in C

- Without “Pass by Reference”

```
void swap(int* a, int* b)
{
   int temp = *a;
   *a = *b;
   *b = temp;
}
```

- And use the function by

```
int main() {
   int x,y;
   x = 1;
   y = 2;
   swap(&x, &y);
}
```

Pass by Reference in C++

```
void swap(int& a, int& b)
{
  int temp = a;
  a = b;
  b = temp;
}
```

- And use the function by

```
int main() {
  int x,y;
  x = 1;
  y = 2;
  swap(x, y);
}
```

Various Parameter Passing in C++

- Consider three functions

```
function1(int a)
{
    a = 10;
}

function2(int& a)
{
    a = 10;
}

function3(int* a)
{
    *a = 10;
}
```

- and the calling:

```
int main()
{
    int x1 = 1;
    int x2 = 1;
    int x3 = 1;
    function1(x1);
    function2(x2);
    function3(&x3);
    cout << x1 << endl;
    cout << x2 << endl;
    cout << x3 << endl;
}
```

- Output:

```
1
10
10
```

Parameter Passing

- function1 is called **pass by value**
  - `a` and `x1` are two *separated* variables with two different memory locations (*two individual copies of data*), changing `a` will not make `x1` changed
- function2 is called **pass by reference**
  - `a` and `x2` are the same entity (one memory location), changing `a` will change `x2`
- function3 is called **pass by pointer**
  - `a` is a pointer, and it is pointing at the memory of `x3`, so changing the memory pointed by `a` will change `x3`

Function Overloading

```
int max(int a, int b) {
  if (a > b) return a;
  else return b;
}

int max(int a, int b, int c) {
  return max(max(a, b), c);
}

int max(double a, double b) {
  if (a – b > 0.00001) return a;
  else return b;
}
```

Overloading a Function name

- If we have 2 or more function definitions for the *same function name*, that is called overloading.

- When you overloaded a function name, the function definitions must have *different numbers of formal parameters* or *the formal parameters must be of different types (i.e. different signatures).*

OOP!
Object-Oriented Programming

How Old People Store Their Money in the Past?

How do We Store Money Now?

Difference between

Difference between

Cookie Can

- Only Store Money
- Anyone can access the money

ATM Machine

- Provides a lot of functions
  - Query
  - Withdraw
  - Deposit
  - Transfer, etc..
- “Clearance/Access levels”
  - You can access your own money
  - Bank staffs/technicians can open up the machine

Bank Account: using C


```
typedef struct {
   int acc_num;
   double balance;
} BankAcct;

void initialize(BankAcct &acct, int bal) {
     ...
}

int withdraw(BankAcct *acct, double amt) {
     ...
}

void deposit(BankAcct *acct, double amt) {
     ...
}
```

Usage

- Correct usage

```
BankAcct ba;
initialize(&ba, 1000);
deposit(&a, 42.2);
withdraw(&a, 500);
```

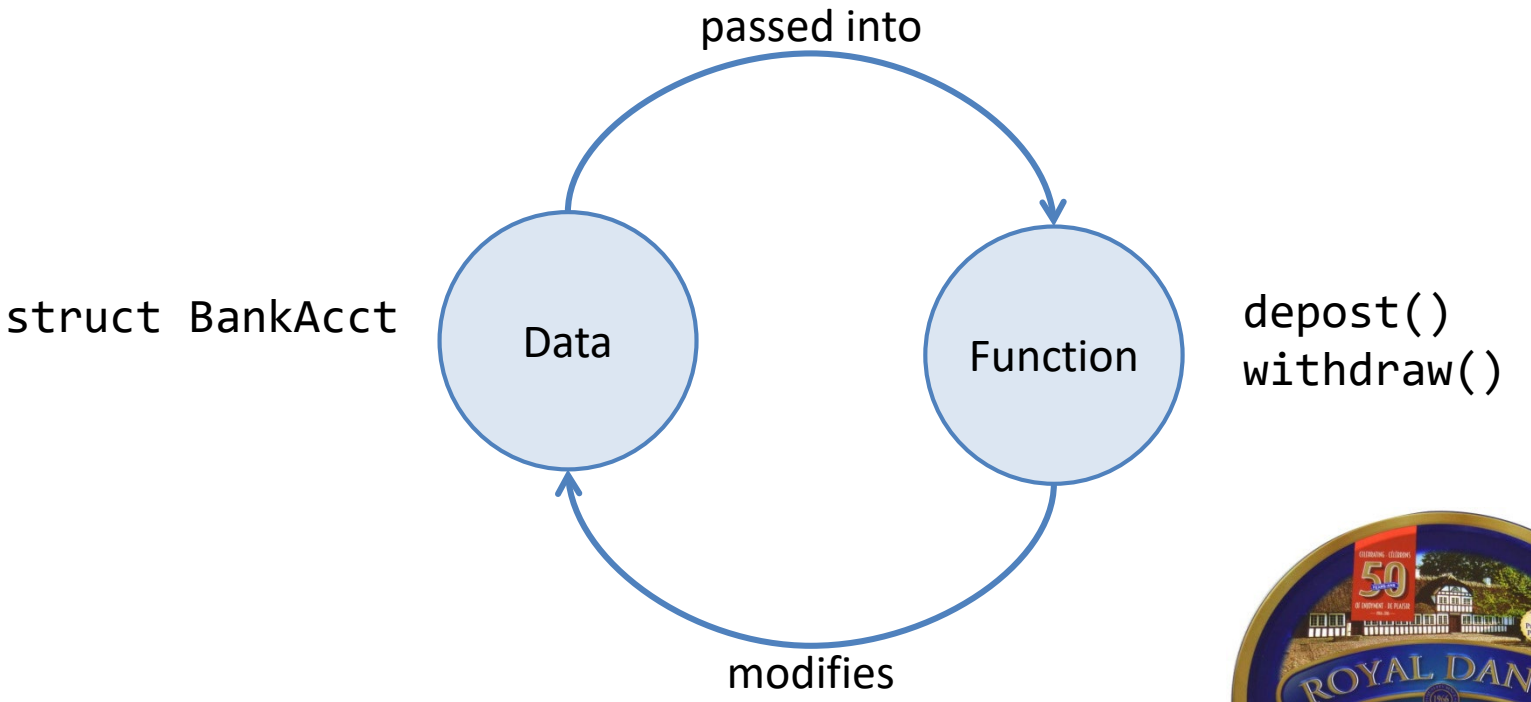
- Wrong and malicious exploits

```
BankAcct ba;
deposit(&ba, 42.2);
initialize(&ba, 1000);
ba.balance += 1000000000;
Withdraw(&ba,999999999999);
```

Procedural Languages

In our C implementation

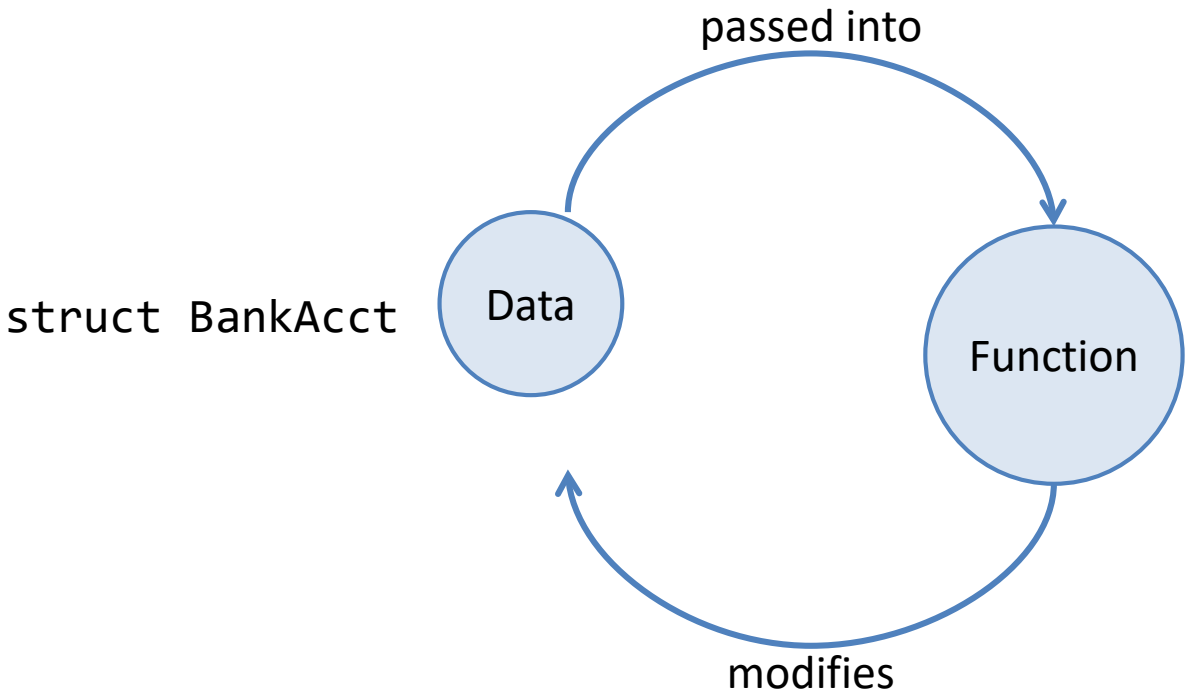
– Data (struct) and process (functions) are separate entities



Procedural Languages

In our C implementation

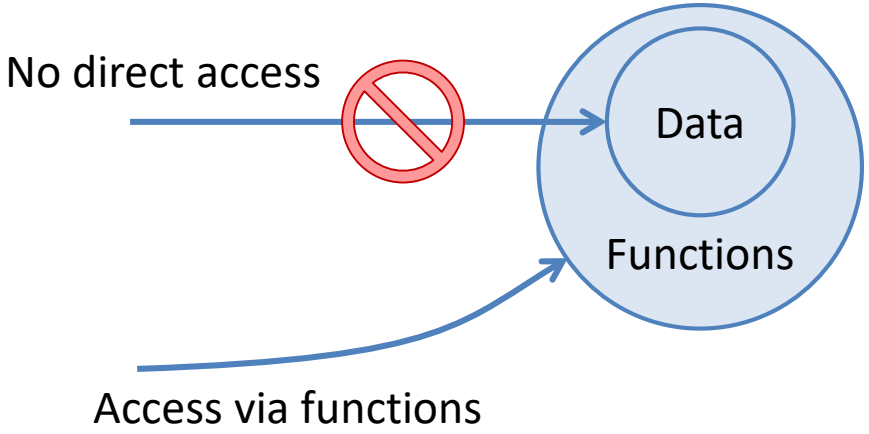
– Data (struct) and process (functions) are separate entities



Conceptual view of OOP

Encapsulation

- Representation of data is encapsulated in the object
- No direct access to data
- Only access using exposed functions
- Data + Function abstraction



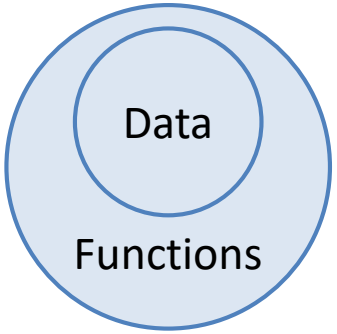
Class vs Instance

Class defines a new data type

– Like a blueprint to create objects

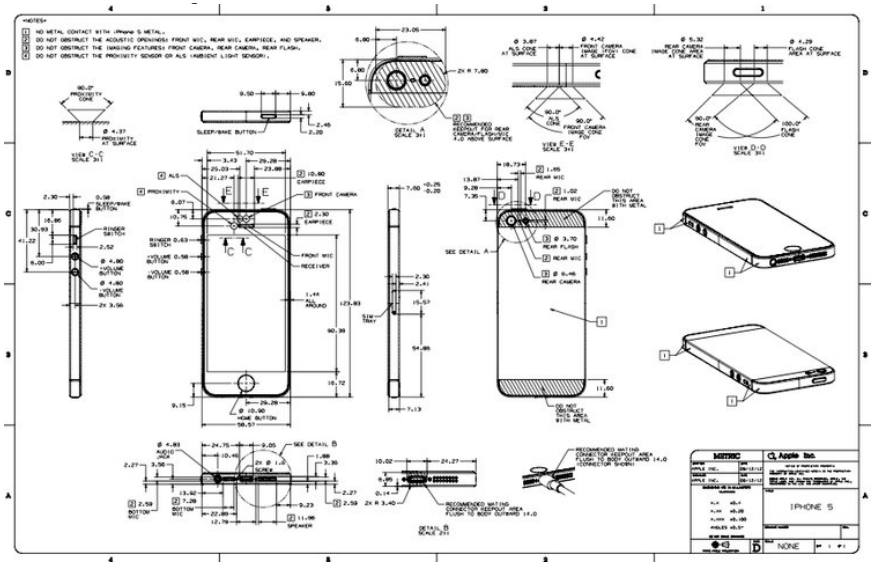
Instance is an object of the class

– Instantiation of a class



Classes vs Instances

- Class



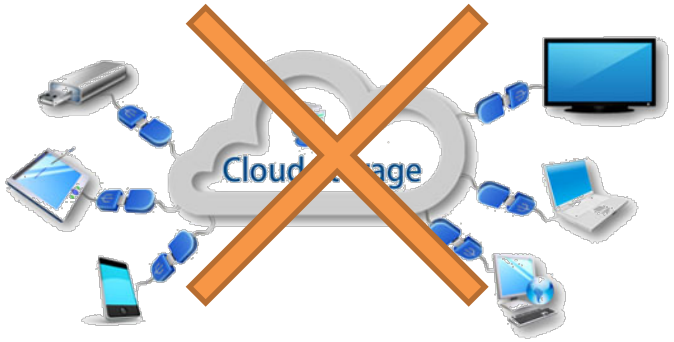
- Instance
- - Actual copies you use

One blueprint can produce a lot of copies of iPhone

One class can produce a lot of copies of instances

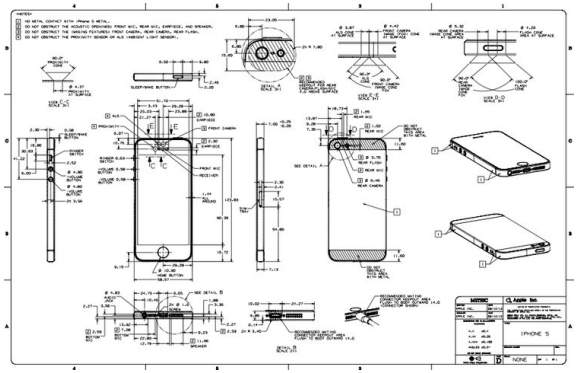
Terminology

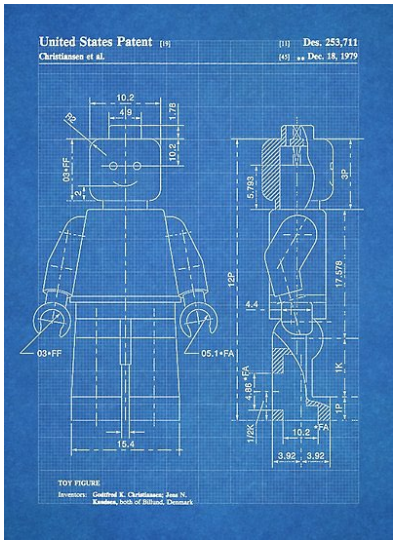
- Class:
  - specifies the common behavior of entities.
  - a blueprint that defines properties and behavior of an object.
  - Also specifies some *common* storages
    - Common as in, every instance has one own copy
    - Not every instance shares the same copy

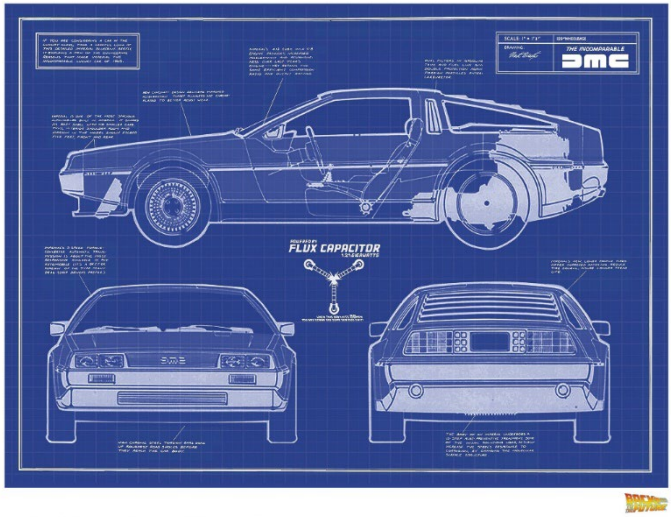


Designing our own class

- C++ OOP means we can design our own class and methods!







- Let's try to design a class called "BankAccount"

Bank Account: using C


```
typedef struct {
   int acc_num;
   double balance;
} BankAcct;

void initialize(BankAcct &acct, int bal) {
    ...
}

int withdraw(BankAcct *acct, double amt) {
    ...
}

void deposit(BankAcct *acct, double amt) {
    ...
}
```

Bank Account: using C++

```
class BankAcct {

private:
  int _acc_num;
  double _balance;

public:
  int withdraw(double amt) {
    if (_balance < amt) return 0;
    _balance -= amount;
    return 1;
  }

 void deposit(double amt) {
   ...
  }
};
```

`class` follows normal identifier rule

`private` indicates no visibility from outside

variables are called attributes or properties

`public`ly accessible definitions

functions are called methods

methods can access all attributes

Alternative Implementation: Bank Account

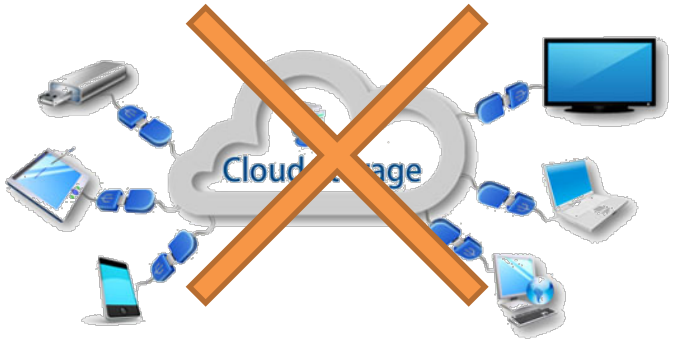
```
class BankAcct {

private:
  int _acc_num;
  double _balance;

public:
  int withdraw(double amt);
  void deposit(double amt) {
    ...
  }
};

int BankAcct::withdraw(double amt){
    if (_balance < amt) return 0;
    _balance -= amount;
    return 1;
}
```

ONE Class and THREE Instances

```
// Assume BankAcct class declared previously
int main() {
    BankAcct AlanAcc,BillyAcc,PeterAcc;

    AlanAcc.deposit(1000);
    AlanAcc.withdraw(500);

    BillyAcc.deposit(9000);
    BillyAcc.withdraw(1000);

    PeterAcc.deposit(100);
    PeterAcc.withdraw(10);

}
```

Terminology

- Class:
  - specifies the common behavior of entities.
  - a blueprint that defines properties and behavior of an object.
  - Also specifies some *common* storages
    - Common as in, every instance has one own copy
    - Not every instance shares the same copy



ONE Class and THREE Instances

```
// Assume BankAcct class declared previously
int main() {
    BankAcct AlanAcc,BillyAcc,PeterAcc;

    AlanAcc.deposit(1000);
    AlanAcc.withdraw(500);

    BillyAcc.deposit(9000);
    BillyAcc.withdraw(1000);

    PeterAcc.deposit(100);
    PeterAcc.withdraw(10);

}
```

Private vs Public

- Outside the class, you can only use/access the public attributes or functions
- Private attributes/functions are used internally
  - You cannot directly modify

~~`AlanAcc._acc_num = 1000`~~

Instances

Instance attributes/properties

- belong to the instance
- each instance has their own data

Instance methods

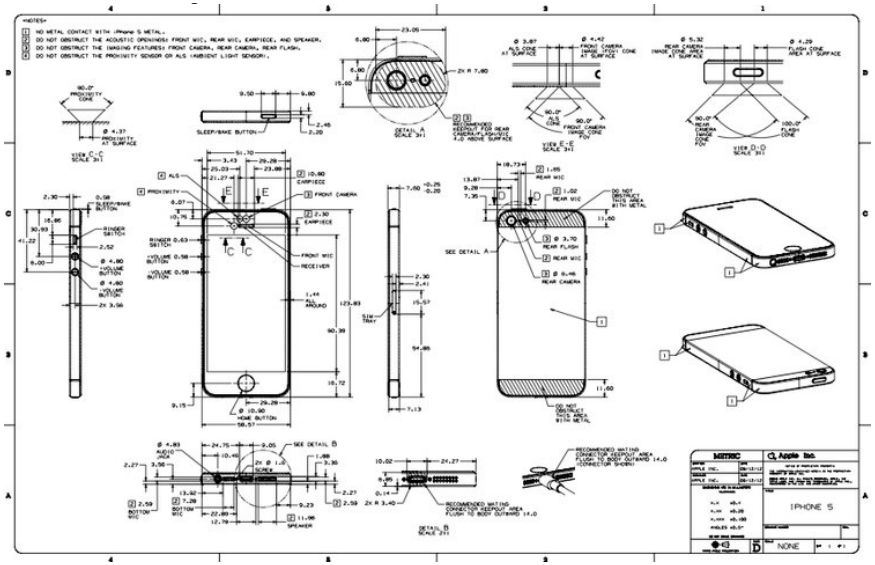
- belong to the instance
- and operate on its own data

ONE Class and THREE Instances

```
// Assume BankAcct class declared previously
int main() {
    BankAcct AlanAcc,BillyAcc,PeterAcc;

    AlanAcc.deposit(1000);
    AlanAcc.withdraw(500);

    BillyAcc.deposit(9000);
    BillyAcc.withdraw(1000);

    PeterAcc.deposit(100);
    PeterAcc.withdraw(10);

}
```

Classes vs Instances

- Class



- Instance
- - Actual copies you use

One blueprint can produce a lot of copies of iPhone

One class can produce a lot of copies of instances

Pointers in C/C++

Pointers

- A Pointer is like
  - A Pokeball!!!!

- The object POINTED by a pointer is like a
  - Pokemon!!!

Pokemon

- In order to carry your Pokemons easily, you carry Pokeballs instead

Pokemon

- In the story, you CANNOT fight with the Pokeball

- A Pokeball is only useful if it contains a Pokemon

Pointers

- You can think of a C/C++ pointer is a Pokeball
- And the object it's pointer to is the Pokemon

- So you can create a Pokeball by declareing a pointer

```
int *ptr;
```

Pointers

- You can think of a C/C++ pointer is a Pokeball
- And the object it's pointer to is the Pokemon

- You have to create a Pokemon to be put into the Pokeball by "new"

```
new int;
```

- At the same time, you "capture" (associate) your Pokemon(object) by the Pokeball (pointer)

```
int *ptr = new int;
```

Pointers

- You can think of a C/C++ pointer is a Pokeball
- And the object it's pointer to is the Pokemon

- Whenever you need the Pokemon, you need to "summon" it out of the Pokeball by "*"

```
*ptr = 10;
cout << *ptr << endl;
```

A Typical Usage of Pointers

- You can think of a C/C++ pointer is a Pokeball
- And the object it's pointer to is the Pokemon

```
int main() {
   int *ptr;
   ptr = new int;
   *ptr = 10
   cout << *ptr << endl;
   delete ptr;
}
```

Create a empty Pokeball

Create a Pokemon and put it into a Pokeball

When you need to work with the Pokemon, instead of the Pokeball, like feeding him, you have to "summon" him out of the ball by "*"

Sadly, the Pokemon has to ~~*die*~~ ~~*leave*~~ be freed at the end

One big difference of the “*”

- All the “*” have different meaning in different places

```
int main() {
   int *ptr;
   ptr = new int;
   *ptr = 10
   cout << *ptr << endl;
   delete ptr;
}
```

Indicate “ptr” is a pointer

When you need to work with the Pokemon, instead of the Pokeball, like feeding him, you have to “summon” him out of the ball by “*”

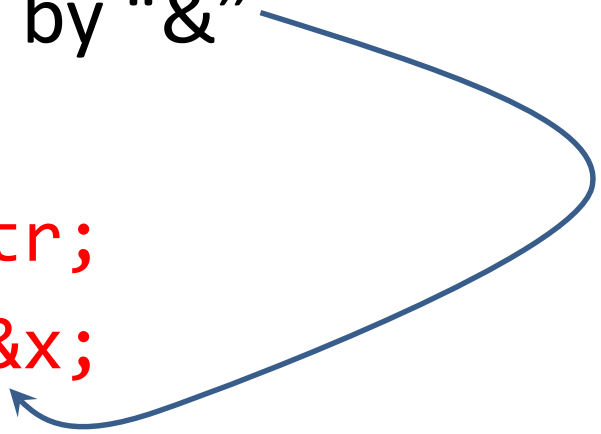
No need “*” even though you are destroying the “Pokemon”, not the “Pokeball”

Pointers

- You can think of a C/C++ pointer is a Pokeball
- And the object it's pointer to is the Pokemon

- Calling a Pokemon from a Pokeball by "*"
- If you have a Pokemon, you want to FIND it's Pokeball by "&"

```
int x;
int *ptr;
ptr = &x;
```



Swap Fucntion

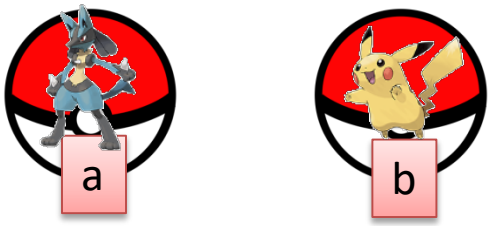
```
int main() {
   int x = 1, y = 2;
   swap(&x, &y);
   cout << x << “,” << y;
}

Output:
   2,1
```

Two Pokemons

Pass the two Pokeballs into the function

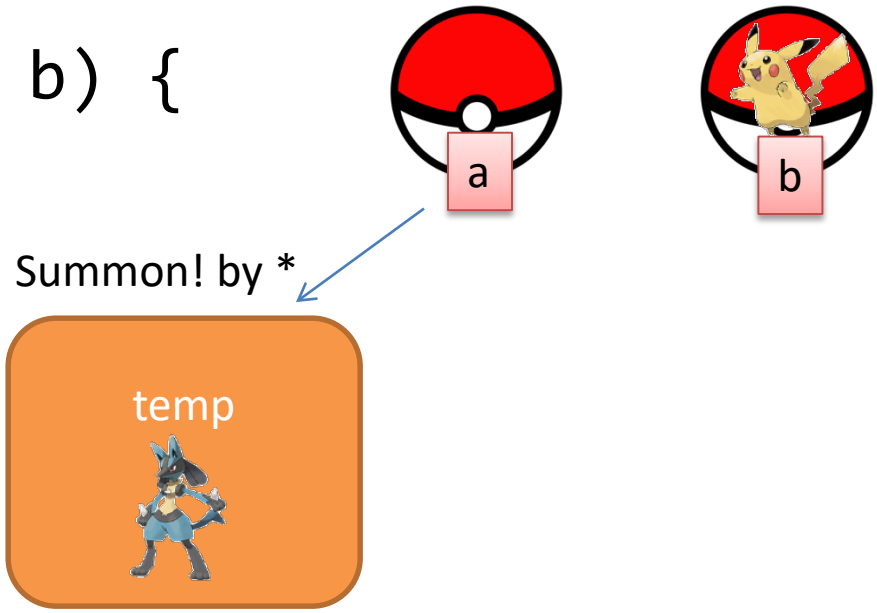
Swap Function

```
void swap(int* a, int* b) {
   int temp;
   temp = *a;
   *a = *b;
   *b = temp;
}
```



Swap Function

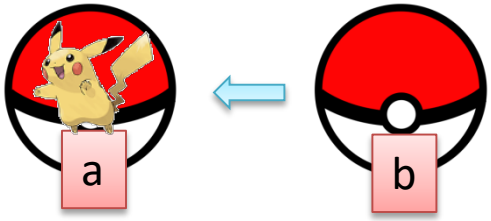
```
void swap(int* a, int* b) {
   int temp;
   temp = *a;
   *a = *b;
   *b = temp;
}
```



Swap Function

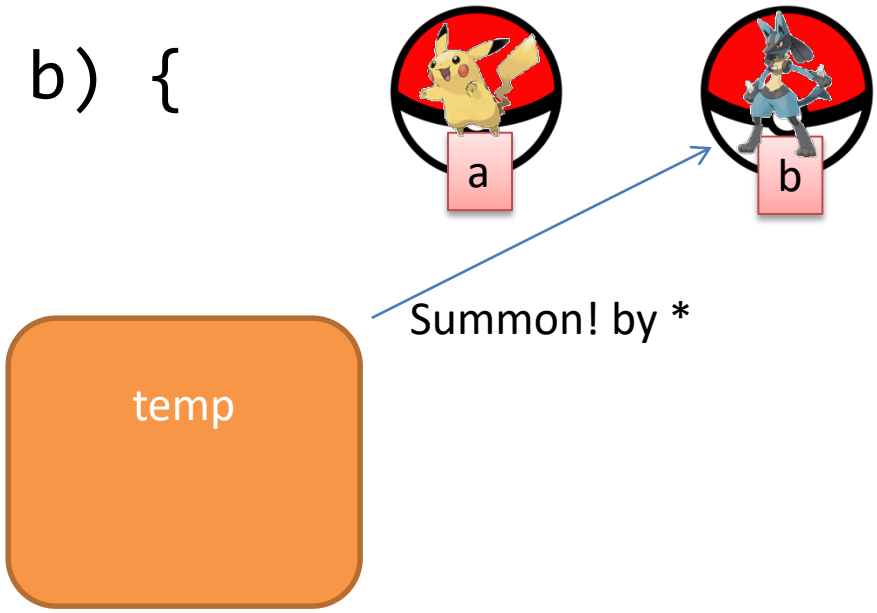
```
void swap(int* a, int* b) {
   int temp;
   temp = *a;
   *a = *b;
   *b = temp;
}
```

Summon! by *



Swap Function

```
void swap(int* a, int* b) {
   int temp;
   temp = *a;
   *a = *b;
   *b = temp;
}
```



However

- It is so inconvenient to have more than one Pokemon!

A Pokeball that has Many Pokemons

- = array!!!!

Array = Pointers

- `int *arr = new int [6];`

Six Pokemons!!!!

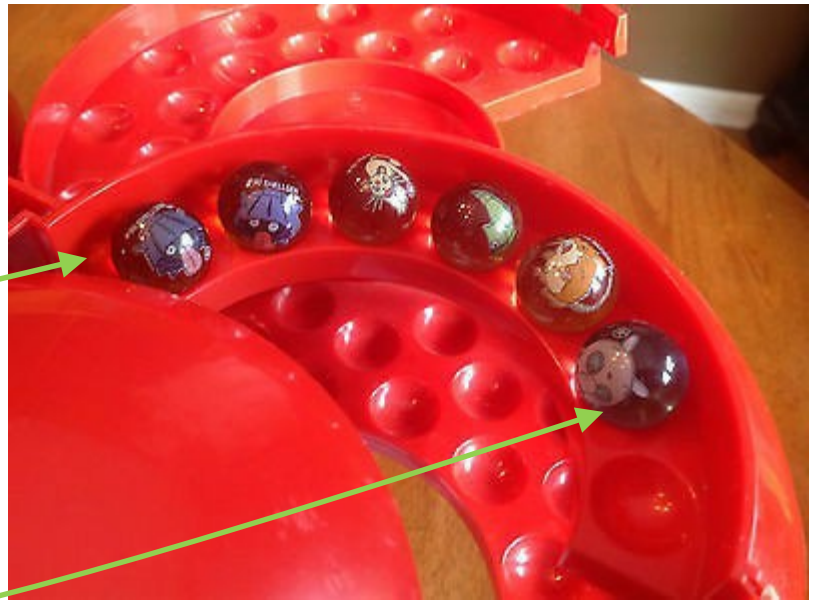
One Pokeball

Array = Pointers

- `int *arr = new int [6];`
- `arr[0];`
- `arr[5];`

First Pokemon

LastPokemon



Note that no need to use “*” to “summon”. The blanket [] does the trick

Free the memory

```
int *arr = new int [6];
```

- Use ”[]”

```
delete [] arr;
```

- Compare to freeing one only

```
int main() {
  int *ptr;
  ptr = new int;
  *ptr = 10
  cout << *ptr << endl;
  delete ptr;
}
```

More C++, OOP