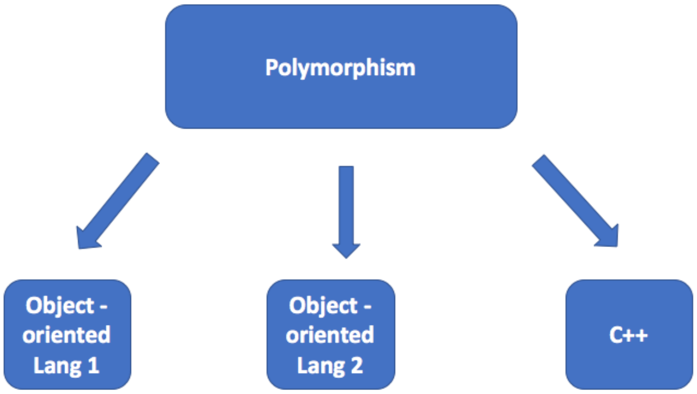
Polymorphism

- The third main principle of object-oriented programing
  - The first two were: encapsulation and inheritance
- Just like I said with the first two-this is a concept that is implemented/supported in a language (different languages can support polymorphism in different ways)

Polymorphism- The provision of a single interface to multiple derived classes, enabling the same method call to invoke different derived methods to generate different results **(also see Stroustrup's glossary for his definition).**

- The quality of being able to assume different forms (poly=many, morphe=form)
- Usually: different situations use different forms
  - Same outcome, different way to get there
  - Same object, different forms in different scenarios


Polymorphism

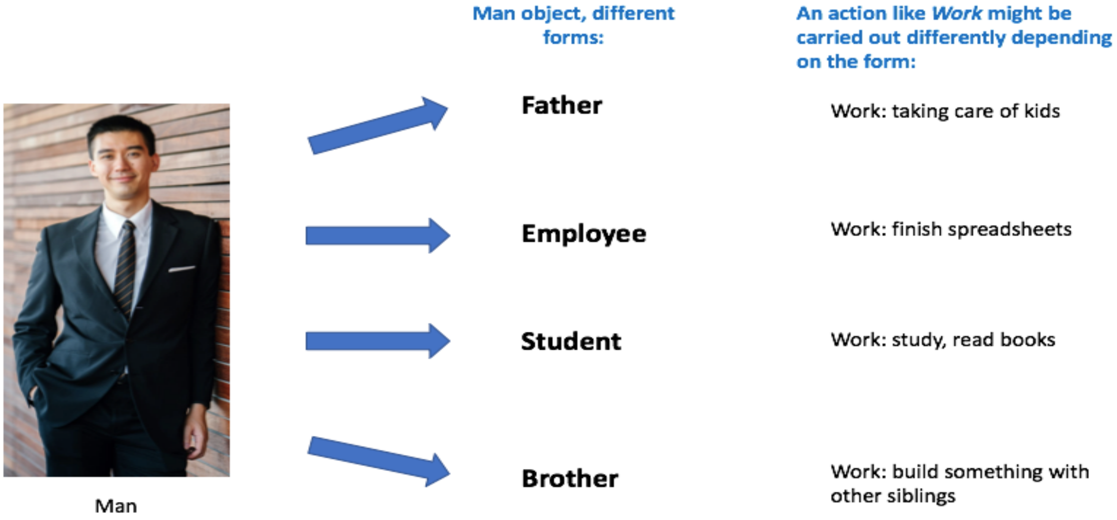
Object - oriented Lang 1

Object - oriented Lang 2

C++

This is a concept that occurs in real life-we need a way to represent it in code.

Real life examples:


Man object, different forms:

An action like *Work* might be carried out differently depending on the form:

Father

Work: taking care of kids

Employee

Work: finish spreadsheets

Student

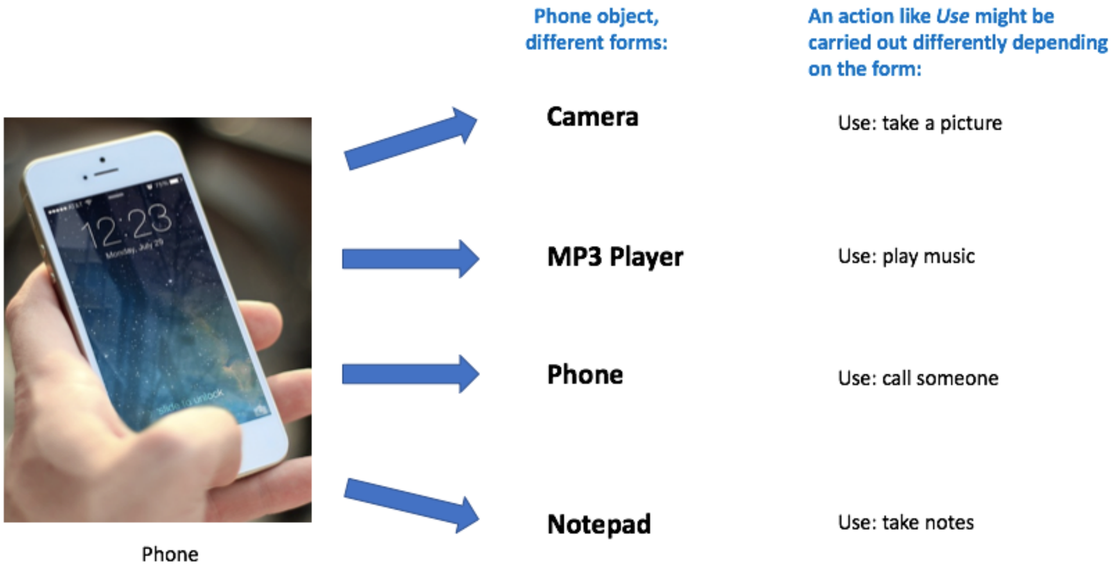
Work: study, read books

Brother

Work: build something with other siblings

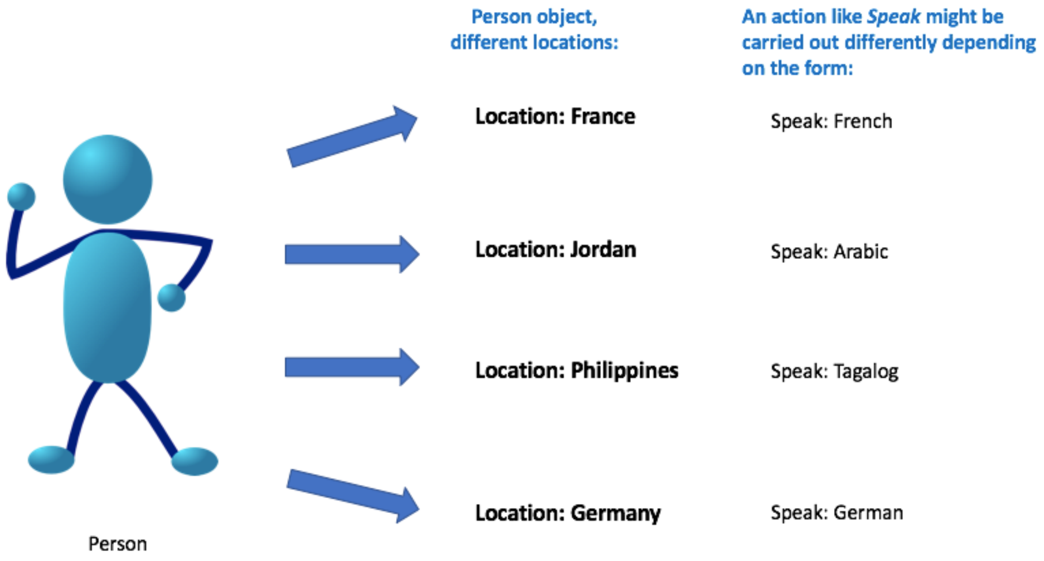
Man

:



Phone object, different forms:	An action like *Use* might be carried out differently depending on the form:
Camera	Use: take a picture
MP3 Player	Use: play music
Phone	Use: call someone
Notepad	Use: take notes

Phone



Person object, different locations:	An action like *Speak* might be carried out differently depending on the form:
Location: France	Speak: French
Location: Jordan	Speak: Arabic
Location: Philippines	Speak: Tagalog
Location: Germany	Speak: German

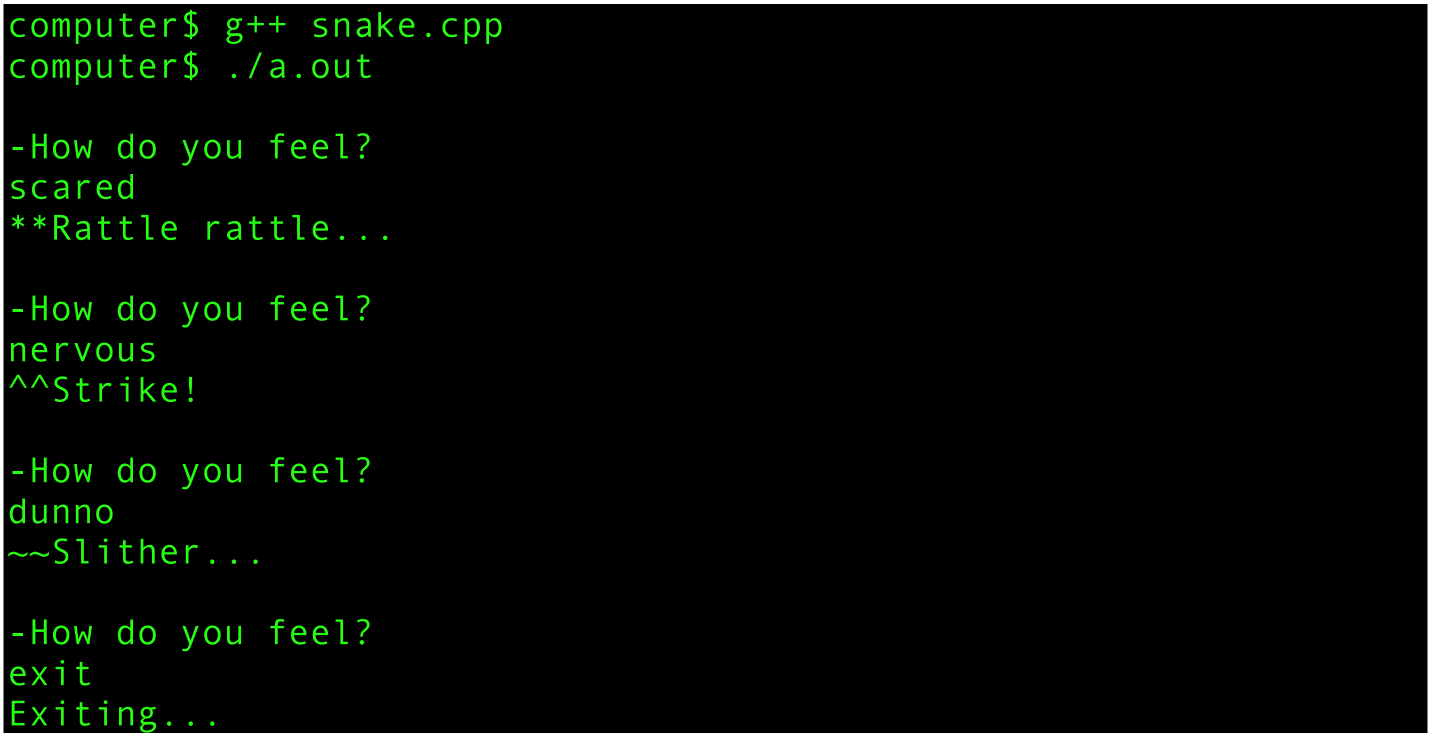
Person

<u>Three main ways to support polymorphism *(concept)* in C++ *(the language)*:</u>

- virtual functions (overriding functions)
  - *Function in derived class has the same name and arguments (but different execution inside)*
  - *Same name of the function indicates the action being carried out is the same, but the way to do it (execution inside) differs*
- overloading operators
  - *Same operator, different actions depending on the situation*
- overloading functions
  - *Same function names, different parameters*

Example 1 (virtual keyword and overriding functions):

If Juan feels scared, that means he saw a rattlesnake. If he gets nervous, that means he saw a cobra. Any other feeling indicates a snake that he does not know the type of.



```
computer$ g++ snake.cpp
computer$ ./a.out

-How do you feel?
scared
**Rattle rattle...

-How do you feel?
nervous
^^Strike!

-How do you feel?
dunno
~~Slither...

-How do you feel?
exit
Exiting...
```

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

class Snake {

public:
 virtual void action() //notice I am using virtual (next example shows what happens if you don't)
 {
  cout << "~~Slither..." <<endl;
 }

};

class Rattlesnake:public Snake{

public:
 void action() //overriding the action given in the Snake class
 {
  cout << "**Rattle rattle..." <<endl;
 }

};

class Cobra:public Snake{
```

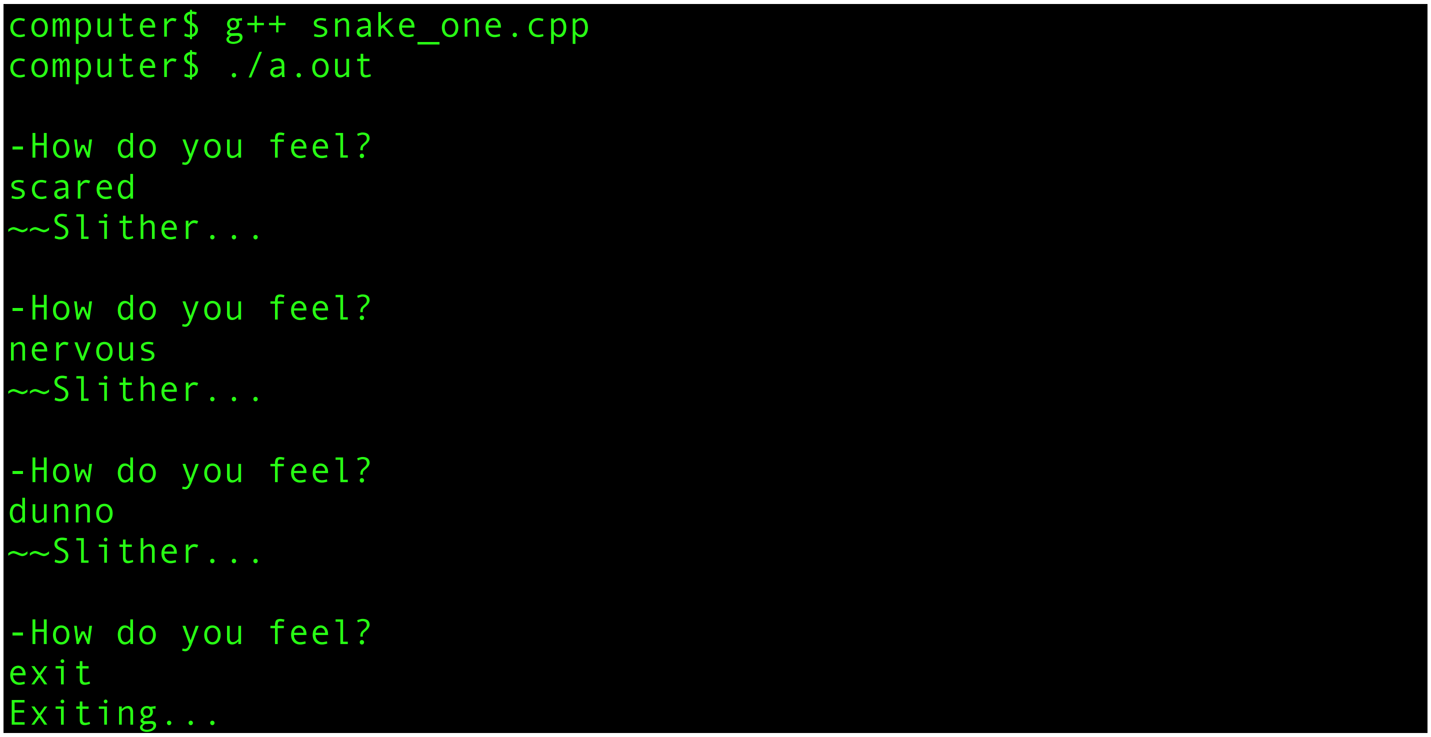
```
public:
  void action()
  {
    cout << "^^Strike!" <<endl;
  }

};


int main (int argc, char **argv) {

string answer;
Snake s1;
Rattlesnake r1;
Cobra c1;

Snake *ptr_s1; //we know it will be a snake, but not which one. Our base pointer can handle any

while(answer!="exit")
{
  cout<<"\n-How do you feel?"<<endl;
  cin >> answer;

  if(answer=="exit")
  {
    cout<<"Exiting..."<<endl;
    continue;
  }

  if(answer=="scared") //rattlesnake
  {
    ptr_s1=&r1;
  }

  else if(answer=="nervous") //cobra
  {
    ptr_s1=&c1;
  }

  else //unknown snake-just a snake
  {
    ptr_s1=&s1;
  }

  ptr_s1->action();
}
```

```
}
```

Not using virtual:

Notice that the following just keeps using ~~Slither (the action defined in the Snake base class). Since we are using a Snake pointer, we are accessing the function in the Snake class (and nothing different in any derived classes).

We use *virtual* to specify we want to use the overridden function in the derived class (previous example).



```
computer$ g++ snake_one.cpp
computer$ ./a.out

-How do you feel?
scared
~~Slither...

-How do you feel?
nervous
~~Slither...

-How do you feel?
dunno
~~Slither...

-How do you feel?
exit
Exiting...
```

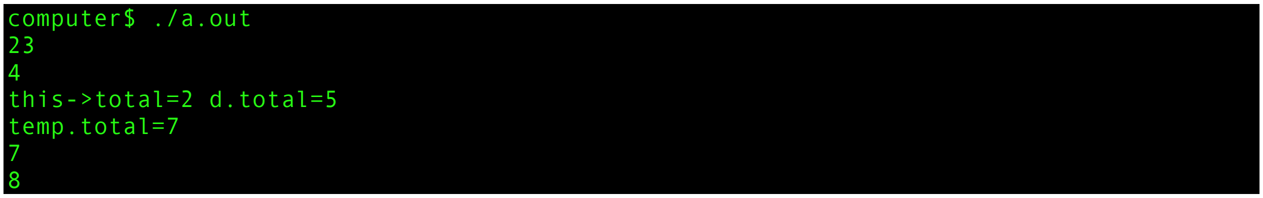
(I changed the following in the code using virtual)

```
class Snake {

public:
 void action() //no virtual here
 {
   cout << "~~Slither..." <<endl;
 }

};
```

Operator overloading:

Adding dog kennels

List of operators you can overload:
https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.3.0/com.ibm.zos.v2r3.cbclx01/cplr318.htm



```
computer$ ./a.out
23
4
this->total=2 d.total=5
temp.total=7
7
8
```

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

class Dog_kennel
{

  public:
     int total;

     Dog_kennel(int n)
     {
       total=n;
     }

     //define the behavior of the overloaded operator
     void operator ++ () //nothing is returned
     {
       total+=20;
     }



     //define the behavior of the overloaded operator
     Dog_kennel operator + (Dog_kennel &d)  //d is second operand, the operation returns a Dog_kennel
     {
       Dog_kennel temp(0);  //just making an empty dog kennel to hold added dog kennels together

       //notice this-> is 2 and d (what is passed in) is 5
       cout<<"this->total="<<this->total<<" d.total="<<d.total<<endl;

       temp.total=this->total + d.total; //add total together (each dog kennel has a member variable called total)-keep it in the dog kennel created above

       cout<<"temp.total="<<temp.total<<endl;
       return temp; //return the temp dog kennel that stores the added total
     }
};



int main(int argc, char **argv)
{
   /*FOR ++: SAME OPERATOR, DIFFERENT SCENARIOS (dog kennels or ints?) MEAN DIFFERENT ACTIONS*/
   Dog_kennel dg(3); //starting with 3 dogs
   ++dg; //operator adds 20 when we call it on a dog kennel (we defined that above)
   cout << dg.total<< endl; //new total (23)

   int n=3;
   ++n; //note we can still use it the way we are used to
   cout<<n<<endl; //n only increments by 1

   /*FOR +: SAME OPERATOR, DIFFERENT SCENARIOS (dog kennels or ints?) MEAN DIFFERENT ACTIONS*/

   Dog_kennel dg1(2); //this kennel has 2
   Dog_kennel dg2(5); //this kennel has 5

   dg=dg1+dg2; //we can now add our Dog_kennel objects (2+5) 5 is "passed in" as the second operand
   cout << dg.total<< endl; //new value is 7 (since we added dg1 and dg2)

   int y=4;
   cout<<(n+y)<<endl; //we can still use it the way we are used to

}
```

Function overloading:

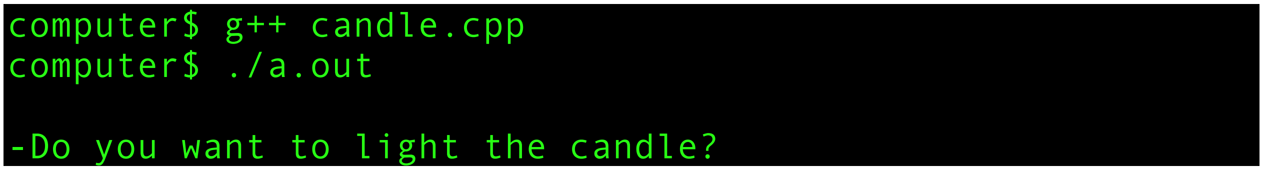
If a person with a match wants to light a candle, he or she can only do so if:

- they are indoors
- they are outdoors and the temperature is below 90 degrees (Fahrenheit)

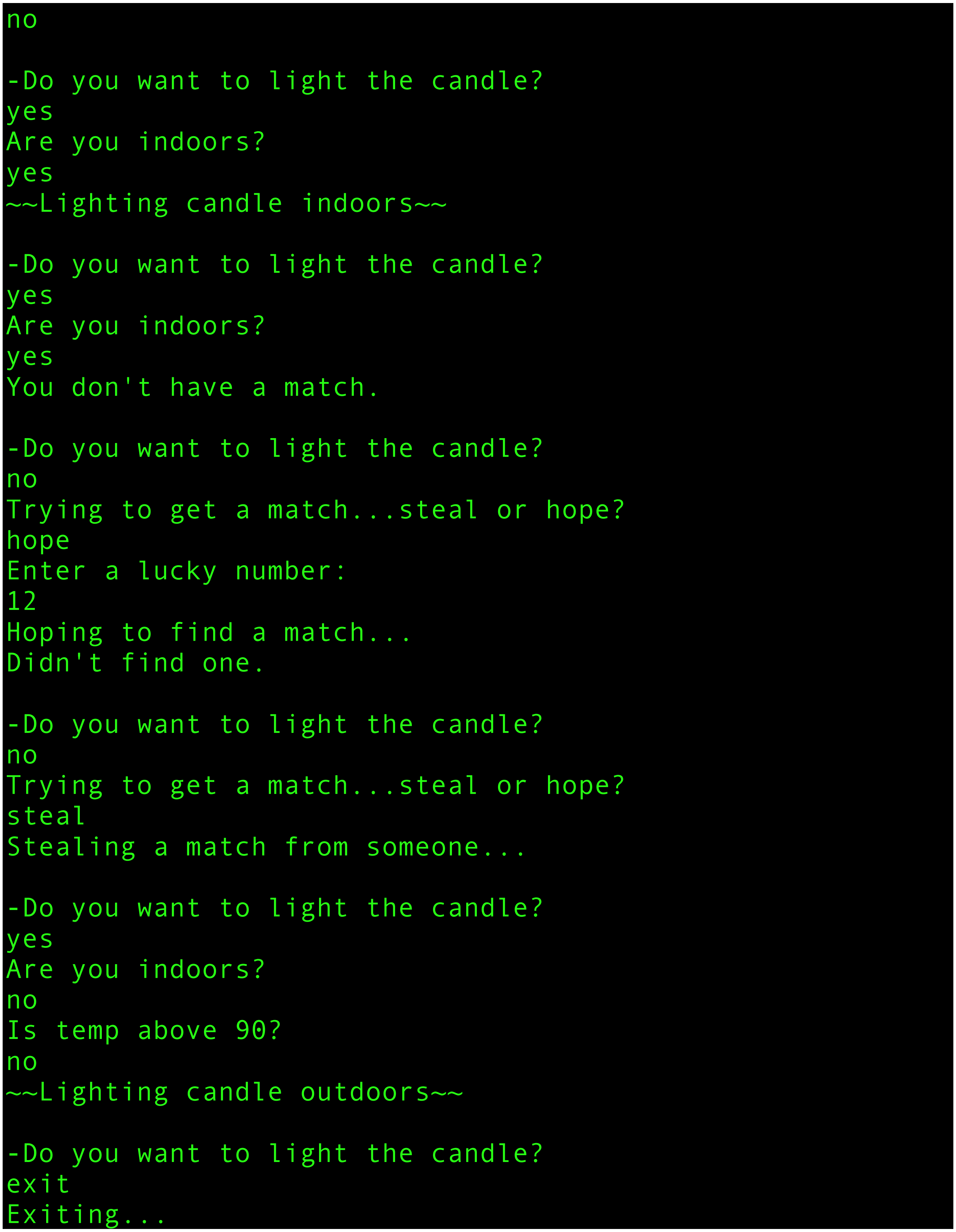
A person can only hold one match at a time. Without a match, a person cannot light a candle.

If a person does not want to light a candle and does not have a match, he or she can either steal one or hope to find one. (I represented hope here by guessing lucky number 13)

Function overloading example indicated in purple



```
computer$ g++ candle.cpp
computer$ ./a.out

-Do you want to light the candle?
```



```
no

-Do you want to light the candle?
yes
Are you indoors?
yes
~~Lighting candle indoors~~

-Do you want to light the candle?
yes
Are you indoors?
yes
You don't have a match.

-Do you want to light the candle?
no
Trying to get a match...steal or hope?
hope
Enter a lucky number:
12
Hoping to find a match...
Didn't find one.

-Do you want to light the candle?
no
Trying to get a match...steal or hope?
steal
Stealing a match from someone...

-Do you want to light the candle?
yes
Are you indoors?
no
Is temp above 90?
no
~~Lighting candle outdoors~~

-Do you want to light the candle?
exit
Exiting...
```

```
#include <iostream>
#include <string>

using namespace std;
```

```
class Candle
{
  public:
  bool on_off;

  Candle(bool initial_state)
  {
    on_off=initial_state;
  }

};

class Person{

  bool match;

  public:
  Person(bool m)
  {
    match=m;
  }

  bool get_match()
  {
    return match;
  }

//The function light_candle is overloaded below based on which situation you are in: indoors or outdoors
//indoors
  void light_candle(Candle &c) //note I am passing by reference so I can change the candle characteristic
inside the Person object
  {
    if(match)
    {
      cout << "~~Lighting candle indoors~~"<<endl;
      c.on_off=true;
      match=false; //match used
    }

    else{
      cout <<"You don't have a match."<<endl;
    }

  }

//outdoors
  void light_candle(bool weather, Candle &c)
  {
    if(weather&&match)
    {
      cout << "~~Lighting candle outdoors~~"<<endl;
      c.on_off=true;
      match=false; //match used
    }

    else
    {
      cout << "Too hot to light candle."<<endl;
    }

}

  //The function acquire_match is overloaded based on which situation you are in-stealing a match or hoping to find one
  void acquire_match()
  {
    cout <<"Stealing a match from someone..."<<endl;
    match=true;
  }

  void acquire_match(int n)
  {
    cout <<"Hoping to find a match..."<<endl;

    if(n==13)
    {
      cout <<"Found one..."<<endl;
      match=true;
    }

    else
    {
      cout <<"Didn't find one."<<endl;
    }

  }

};

int main(int argc, char **argv)
{
  Candle c1(false); //starting in off state
  Person p1(true); //starting with a match

  string answer;
  int n;

  while(answer!="exit")
```

```
{
 cout<<"\n-Do you want to light the candle?"<<endl;
 cin >> answer;

 if(answer=="exit")
 {
   cout<<"Exiting..."<<endl;
   continue;
 }

 if(answer=="yes")
 {
  cout<<"Are you indoors?"<<endl;
  cin >> answer;

  if(answer=="yes")
  {
   p1.light_candle(c1);
  }

  else //outdoors
  {
   cout<<"Is temp above 90?"<<endl;
   cin >> answer;

   if(answer=="yes")
   {
    p1.light_candle(false,c1);
   }

   else{
    p1.light_candle(true,c1);
   }
  }

 }

 else //assume no
 {

  if(!p1.get_match()) //don't currently have a match
  {
   cout <<"Trying to get a match...steal or hope?"<<endl;
   cin >>answer;

   if(answer=="steal")
   {
    p1.acquire_match();
   }
```

```
    else //assume hope
    {
     cout <<"Enter a lucky number: "<<endl;
     cin >> n;
     p1.acquire_match(n);

    }
   }
  }
 }

}
```