

# Polymorphism ( +virtual keyword, header files)

---

Notice while overriding the function foo() in the base class that the following works fine (both the derived and base functions are called even though I'm not using virtual) :

```
computer$ g++ -std=c++11 practice.cpp
computer$ ./a.out
Base-A function.
Derived-B function.
```

```
#include <iostream>
```

```
using namespace std;
```

```
class A{
```

```
public:
```

```
    void foo()
```

```
    {
```

```
        cout<<"Base-A function."<<endl;
```

```
    }
```

```
};
```

```
class B:public A{
```

```
public:
```

```
    void foo()
```

```
    {
```

```
        cout<<"Derived-B function."<<endl;
```

```
    }
```

```
};
```

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

```
{
```

```
    A *a1=new A();
```

```
    B *b1=new B();
```

```
    a1->foo(); //calls the function in A
```

```
    b1->foo(); //calls the function in B
```

```
    //delete pointers
```

```
}
```

---

But what about here? Both are calling only the base function:

```
computer$ g++ -std=c++11 practice.cpp
Computers-MacBook-Air:C++ computer$ ./a.out
Base-A function.
Base-A function.
```

```
#include <iostream>
```

```
using namespace std;
```

```
class A{
```

```
public:
```

```
    void foo()
```

```
    {
```

```
        cout<<"Base-A function."<<endl;
```

```
    }
```

```
};
```

```
class B:public A{
```

```
public:
```

```
    void foo()
```

```
    {
```

```
        cout<<"Derived-B function."<<endl;
```

```
    }
```

```
};
```

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

```
{
```

```
    A *a1=new A();
```

```
    A *b1=new B(); //using a base pointer to create a derived object
```

```
    a1->foo();
```

```
    b1->foo(); //the base function for A will be called
```

```
    //delete pointers
```

```
}
```

**And here? Both are calling the base function:**

```
computer$ ./a.out
Base-A function.
Base-A function.
```

```
#include <iostream>
```

```
using namespace std;
```

```
class A{
```

```
public:
```

```
    void foo()
    {
        cout<<"Base-A function."<<endl;
    }
```

```
};
```

```
class B:public A{
```

```
public:
```

```
    void foo()
    {
        cout<<"Derived-B function."<<endl;
    }
```

```
};
```

```
class C{
```

```
public:
```

```
    void random_function(A* a1)
    {
        a1->foo();
    }
```

```
};
```

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

```
{
```

```
    A *a1=new A();
    B *b1=new B();
    C c1;
```

```
    c1.random_function(a1);
    c1.random_function(b1); //the base function foo() for A will be called here, not B
```

```
    //delete pointers
```

```
}
```

***NOTE you don't have this issue with calling the function if the parameter of the calling function is the derived class:***

```
computer$ g++ -std=c++11 practice.cpp
computer$ ./a.out
Derived-B function.
```

```
#include <iostream>
```

```
using namespace std;
```

```
class A{
```

```
public:
```

```
    void foo()
    {
        cout<<"Base-A function."<<endl;
    }
```

```
};
```

```
class B:public A{
```

```
public:
```

```
    void foo()
    {
        cout<<"Derived-B function."<<endl;
    }
```

```
};
```

```
class C{
```

```
public:
```

```
    void random_function(B* a1)
    {
        a1->foo();
    }
```

```
};
```

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

```
{
```

```
A *a1=new A();
```

```
B *b1=new B();
```

```
C c1;
```

```
c1.random_function(b1); //no issue since the parameter is the derived class
```

```
//delete pointers
```

```
}
```

### What is going on here?

- The issue is because when running the program and passing a base parameter, it is assumed that the base function should be called.
- By including virtual, we are saying run the function of the derived class even if the pointer is base one (and we want to use it).
  - Virtual makes sure the correct function is called (base or derived)

We can fix all of the above instances by including the keyword virtual on the function that is begin overridden in the derived class.

**Note that you can still access the base class function:**

```
computer$ ./a.out
Derived-B function.
Base-A function.
```

```
#include <iostream>
```

```
using namespace std;
```

```
class A{
```

```
public:
```

```
    virtual void foo()
```

```
    {
```

```
        cout<<"Base-A function."<<endl;
```

```
    }
```

```
};
```

```
class B:public A{
```

```
public:
```

```
    void foo()
```

```
    {
```

```
        cout<<"Derived-B function."<<endl;
```

```
    }
```

```
};
```

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

```
{
```

```
    A *a1=new A();
```

```
    B *b1=new B();
```

```
    b1->foo(); //this will call the overridden function
```

```
    b1->A::foo(); //I am able to access the base class function by using the scope resolution operator and calling the base function directly (remember, a base class function is behind the scenes when creating the derived class)
```

```
}
```

*Finally, note that the above concept (calling base class info) works in general (with base class variables/functions, with or without virtual):*

```
computer$ g++ -std=c++11 practice.cpp
computer$ ./a.out
7
Derived-B function.
3
Base-A function.
30
```

```
#include <iostream>
```

```
using namespace std;
```

```
class A{
```

```
public:
```

```
    int i=3;
```

```
    void foo()
```

```
    {
```

```
        cout<<"Base-A function."<<endl;
```

```
    }
```

```
};
```

```
class B:public A{
```

```
public:
```

```
    int i=7;
```

```
    void foo()
```

```
    {
```

```
        cout<<"Derived-B function."<<endl;
```

```

    }

};

int main(int argc, char **argv)
{
    B b1;

    cout<<b1.i<<endl; //calling the derived class value of i
    b1.foo(); //calling the derived class function

    cout<<b1.A::i<<endl; //calling the base class value of i
    b1.A::foo(); //calling the base class function

    b1.A::i=30; //you can access and change the base class variable
    cout<<b1.A::i<<endl;

}

```

Note that in these cases, since I'm not using virtual, it is just as if we have the same name for a function (and variable) by chance in the base and derived class (inheritance). It is not a direct implementation of polymorphism. This is because in polymorphism we are saying that we are purposely changing the implementation in the derived class by overriding the base class (using virtual), not accidentally having the same function name.

---

### Virtual destructors:

```

computer$ g++ -std=c++11 practice.cpp
computer$ ./a.out
A constructor
B constructor
A destructor

```

**Notice that we have created a B object using an A pointer, but when we run the program (above) only the A destructor is called:**

```

#include <iostream>

using namespace std;

class A{
public:
    A()
    {
        cout<<"A constructor"<<endl;
    }

    ~A()
    {

```

```

        cout<<"A destructor"<<endl;
    }

};

class B:public A{

public:
    B()
    {
        cout<<"B constructor"<<endl;
    }

    ~B()
    {
        cout<<"B destructor"<<endl;
    }
};

int main(int argc, char **argv)
{
    A* a1=new B(); //we are created a B object and it is pointed at by an A pointer
    delete(a1);
}

```

**We can fix this by making the destructor virtual (remember the virtual keyword allows the type of function to be called based on what is actually being pointed at NOT the pointer alone):**

```

computer$ g++ -std=c++11 practice.cpp
Computers-MacBook-Air:C++ computer$ ./a.out
A constructor
B constructor
B destructor
A destructor

```

```

#include <iostream>

using namespace std;

class A{

public:
    A()
    {
        cout<<"A constructor"<<endl;
    }

    virtual ~A()
    {

```



```

        cout<<"A destructor"<<endl;
    }
};

class B:public A{

public:
    B()
    {
        cout<<"B constructor"<<endl;
    }

    ~B()
    {
        cout<<"B destructor"<<endl;
    }
};

int main(int argc, char **argv)
{
    A* a1=new B(); //we are created a B object and it is pointed at by an A pointer
    delete(a1);
}

```

**Notice a smart pointer does it by itself (but we are not always guaranteed that someone will make a smart pointer to your class, so it's better to be safe):**

```

computer$ g++ -std=c++11 practice.cpp
Computers-MacBook-Air:C++ computer$ ./a.out
A constructor
B constructor
B destructor
A destructor

```

```

#include <iostream>

using namespace std;

class A{

public:
    A()
    {
        cout<<"A constructor"<<endl;
    }

    ~A()
    {
        cout<<"A destructor"<<endl;
    }
}

```

```
};

class B:public A{
public:
    B()
    {
        cout<<"B constructor"<<endl;
    }

    ~B()
    {
        cout<<"B destructor"<<endl;
    }
};

int main(int argc, char **argv)
{
    shared_ptr<A> p_ptr1=make_shared<B>();
}
```

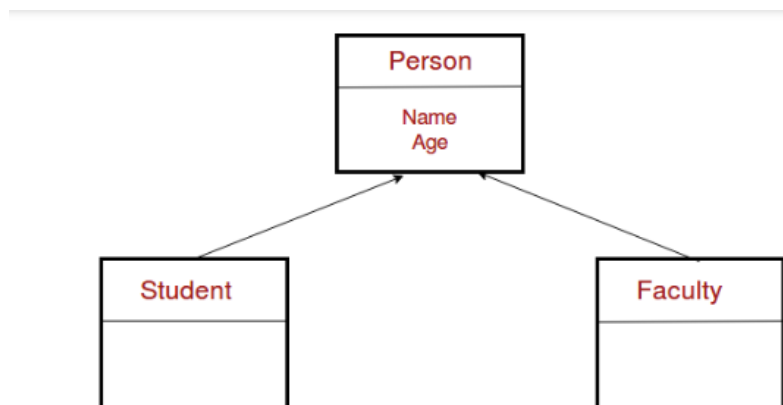
**Why we have no virtual constructors (this webpage has many good bits of info about C++ from the C++ creator):**

[http://www.stroustrup.com/bs\\_faq2.html#virtual-ctor](http://www.stroustrup.com/bs_faq2.html#virtual-ctor)

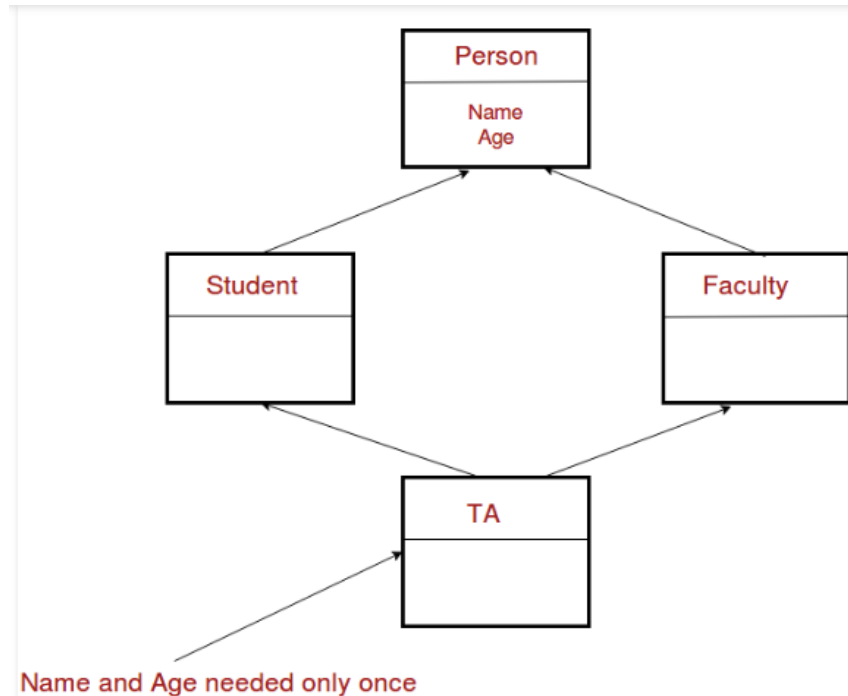
---

### Virtual inheritance:

Imagine that you have two classes inheriting from a single base class:



What happens if you want to now have a class inherit from both of these derived classes? You can end up with what is called the Diamond Problem (both Student and Faculty have *name* and *age*):



```
computer$ g++ practice.cpp
practice.cpp:48:13: error: non-static member 'name' found
in multiple base-class
    subobjects of type 'Person':
    class TA -> class Faculty -> class Person
    class TA -> class Student -> class Person
        cout<<ta1.name<<endl;
           ^
practice.cpp:5:11:      member found by ambiguous name
lookup
    string name;
           ^
1 error generated.
```

Which name we are talking about is not clear (we essentially have access to name twice-once from each class we are inheriting from ).

```
#include<iostream>
using namespace std;
class Person {
public:
```

```

        string name;
        int age;

    Person(int age, string name)
    {
        this->name=name;
        this->age=age;
    }
};

```

//we inherited name and age here

```

class Faculty : public Person {

public:
    Faculty(int age, string name):Person(age, name)
    {

    }

};

```

//we have also inherited name and age here

```

class Student : public Person {
    // data members of Student
public:
    Student(int age, string name):Person(age, name)
    {

    }

};

```

```

class TA : public Faculty, public Student
{
    public:
    TA(int age, string name):Student(age, name), Faculty(age, name)
    {

    }

};

```

```

int main(int argc, char **argv) {
    TA ta1(30,"bob");
    cout<<ta1.name<<endl;

}

```

**How can we fix this? We can use something called virtual inheritance:**

```

#include<iostream>
using namespace std;
class Person {
public:
    string name;
    int age;

    Person(int age, string name)
    {
        this->name=name;
        this->age=age;
    }
};

class Faculty : virtual public Person //use virtual here
{
public:
    Faculty(int age, string name):Person(age, name)
    {

    }
};

class Student : virtual public Person { //use virtual here

public:
    Student(int age, string name):Person(age, name)
    {

    }
};
//since we virtually inherited, we don't have the same issue from above
class TA : public Faculty, public Student
{
public:
    TA(int age, string name):Student(age, name), Faculty(age, name), Person(age,name) //include Person
    here
    {

    }
};

int main(int argc, char **argv)
{
    TA ta1(30,"bob");
    cout<<ta1.name<<endl;
}

```

---

We now no longer have the diamond problem.

---

## Header files

### Making header files:

- We have already used header files developed by other people
- We can make our own header files (and include them at the top of our program)
  - Reduce clutter
  - Make code re-usable
- For example, we can turn the following code (all kept in one file-as we have been doing so far)...

```
#include <iostream>

using namespace std;

class Animal{
    bool mammal; //true=yes, false=no

public:

    void set_mammal(bool b); //function prototype
    void print_out();

};

class Zebra: public Animal{

};

//function definitions-remember we can define the functions outside the class
void Animal::set_mammal(bool b)
{
    mammal=b;
}

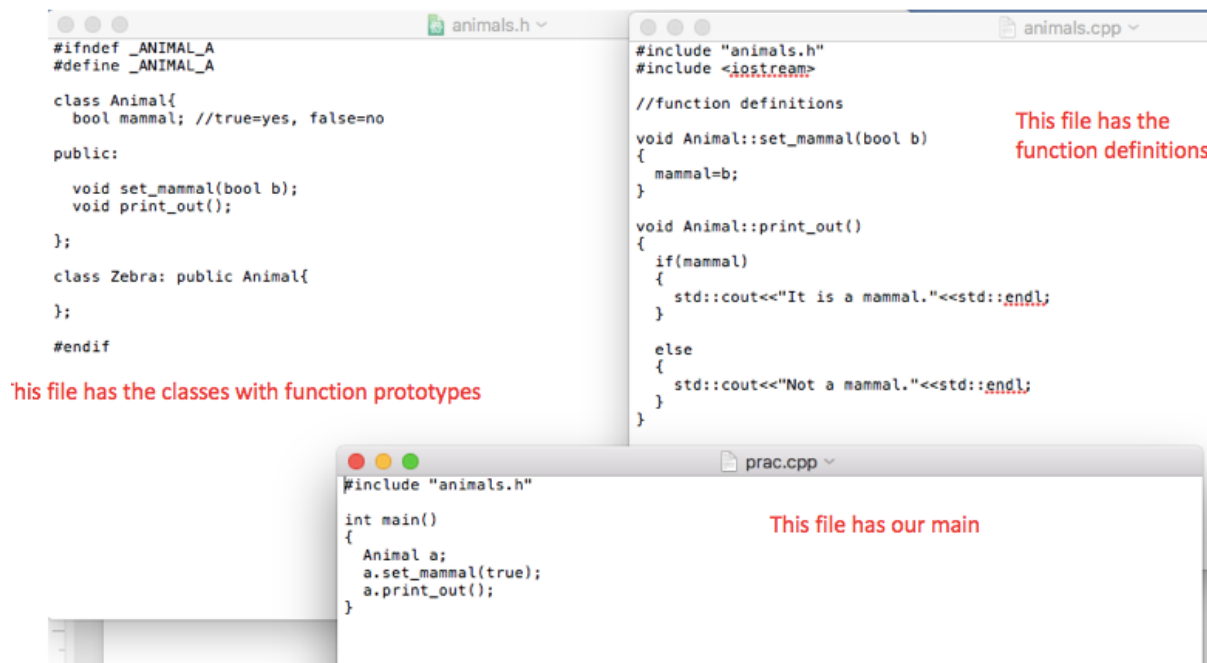
void Animal::print_out()
{
    if(mammal)
    {
        std::cout<<"It is a mammal."<<std::endl;
    }

    else
    {
        std::cout<<"Not a mammal."<<std::endl;
    }
}

int main()
{
    Animal a;
    a.set_mammal(true);
```

```
a.print_out();
}
```

...into this (notice how clean the file that holds the main is since our classes are kept in the other file):



When we compile, we include both .cpp files:

```
computer$ g++ prac.cpp animals.cpp
computer$ ./a.out
It is a mammal.
```

### #include "headerfile" vs #include <headerfile>:

Difference between something like **#include "headerfile"** vs **#include <headerfile>** (search path):

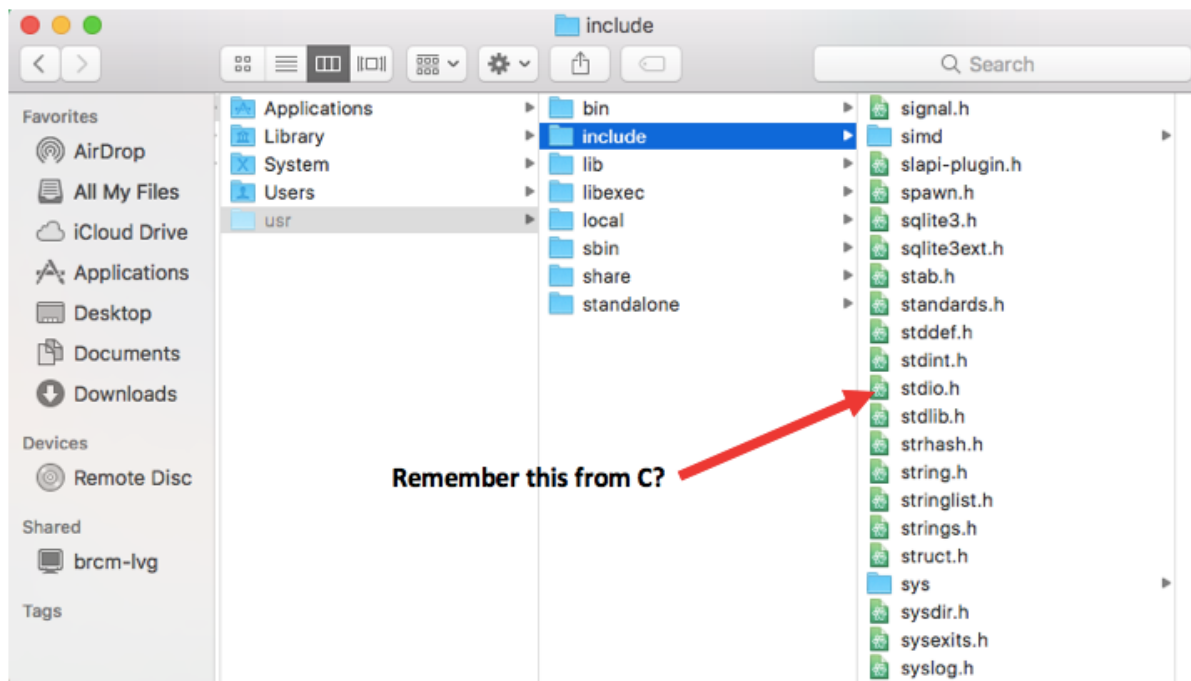
#### #include "headerfile"

1. First, search directory of current file
  - a. If my program is on my Desktop, we will look around my Desktop first for the headerfile
2. Second, look in preconfigured list of standard system directories (see picture below)

#### #include <headerfile>

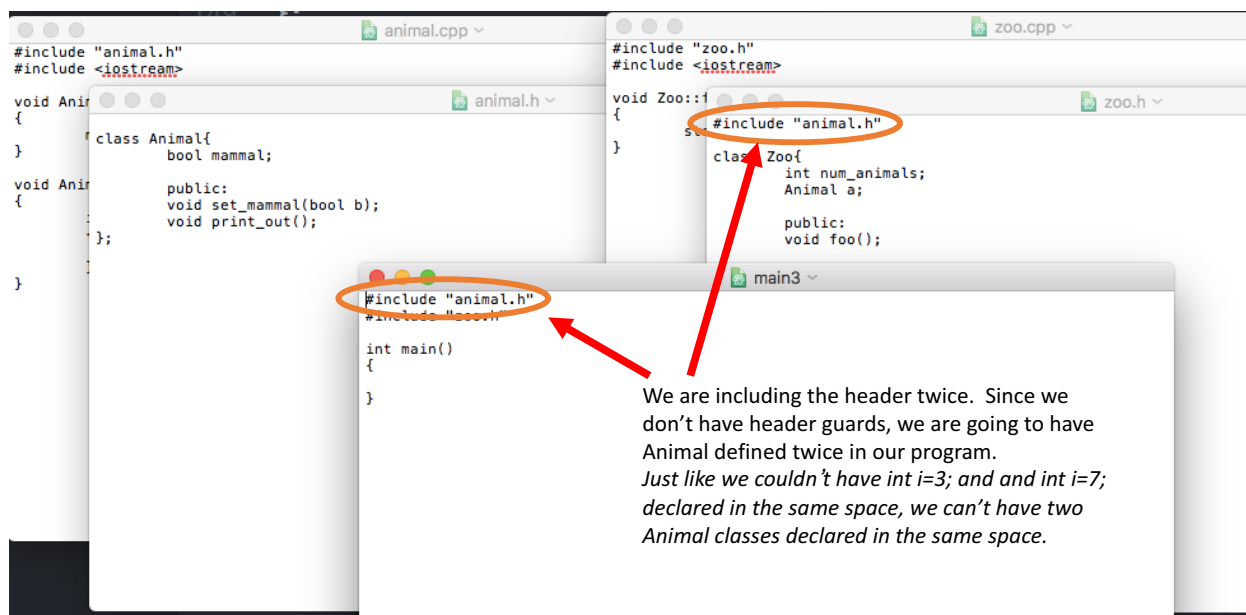
1. Look in preconfigured list of standard system directories only (see picture below)

**Note:** `#include` (called a *preprocessor directive*) tells the compiler to include a file (our header file) in our program. For example, we defined our functions in another file (like `header.h`) and we want to include these definitions so we can use them in our main.



## Header guards/inclusion guards:

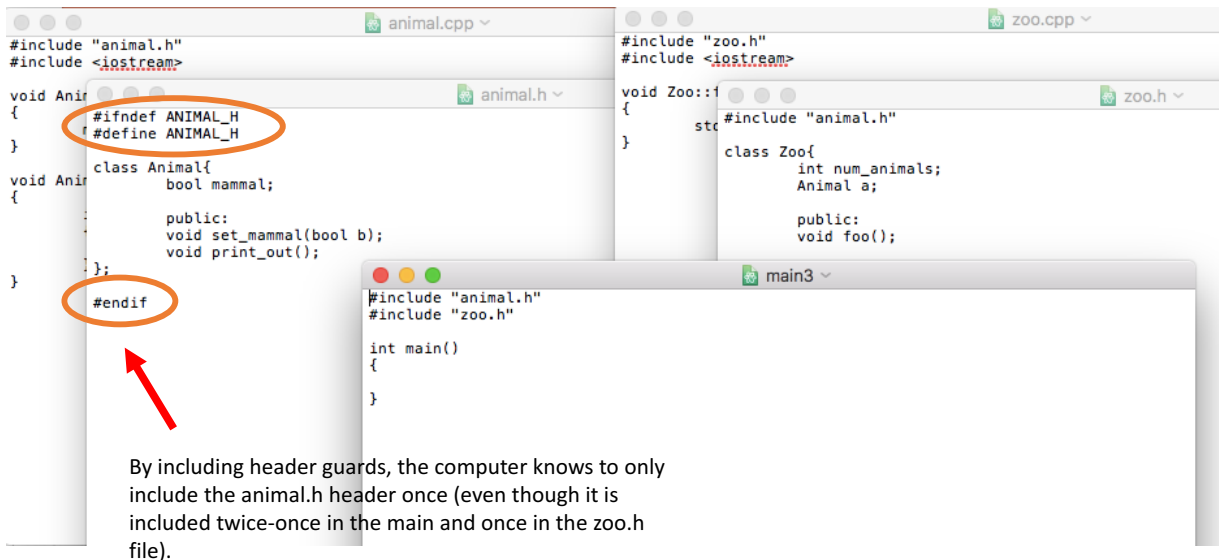
When making your header files (.h) you should always include header guards (see .h example below). By including them, we ensure we do not define something twice.





Errors when you don't use header guards:

```
computer$ g++ -std=c++14 main3.cpp animal.cpp zoo.cpp
In file included from main3.cpp:2:
In file included from ./zoo.h:1:
./animal.h:2:7: error: redefinition of 'Animal'
class Animal{
^
main3.cpp:1:10:      './animal.h' included multiple times, additional include site
here
#include "animal.h"
^
./zoo.h:1:10:      './animal.h' included multiple times, additional include site
here
#include "animal.h"
^
./animal.h:2:7:      unguarded header; consider using #ifdef guards or #pragma once
class Animal{
^
1 error generated.
```



How does this work? The header guards give a specific “name” to the header file (“name” is ANIMAL\_H). If we have already included it once and try to include it again, we check first to make sure the “name” ANIMAL\_H hasn’t already been included before.

For example, if the first time we try to include *animal.h* is in the *zoo.h* file, we go ahead and include the class *Animal* in our whole program. If we try to include *animal.h* again (in the *main3.cpp* file), we check first that our “name” ANIMAL\_H hasn’t already been used-since it has been used, we know not to include it a second time.

---

## Program 1:

Hermine is a famous Egyptologist. She is exploring a new pyramid that contains a mummy (it can be a king, queen or unknown mummy) and treasures. If the gender of the mummy is visible, the condition of the mummy is assumed to be good.

```
computer$ g++ main.cpp mummy.cpp
computer$ ./a.out
Looking around...

t
Found a treasure!
Total treasures so far: 1

Looking around...

t
Found a treasure!
Total treasures so far: 2

Looking around...
f
Gender?
female
Looking around...
exit

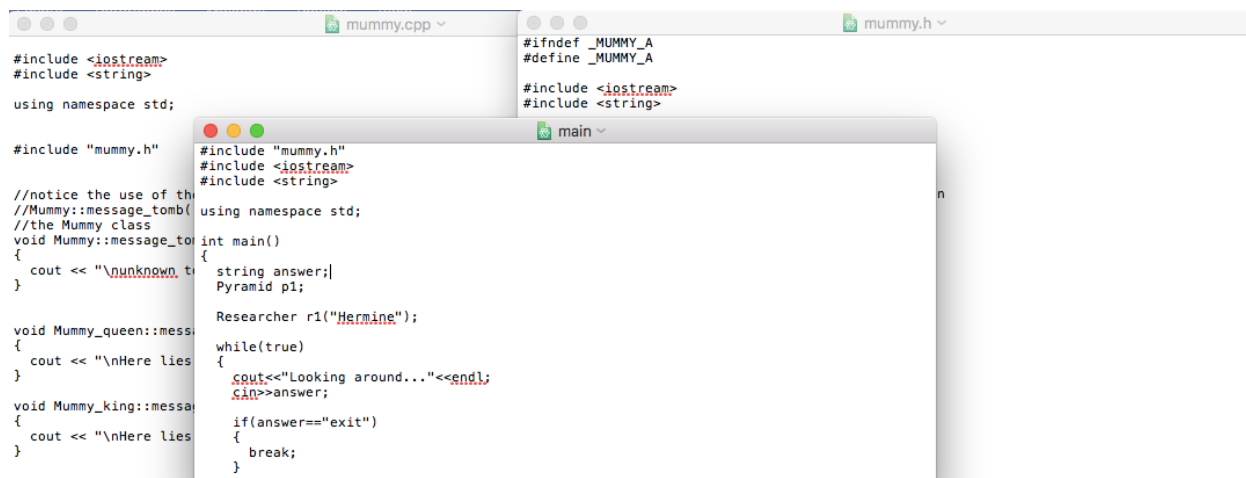
Here lies a queen.

computer$ ./a.out
Looking around...

f
Gender?
male
Looking around...
exit

Here lies a king.

computer$ ./a.out
Looking around...
exit
No mummy found in this pyramid yet...
```



```
mummy.cpp
#include <iostream>
#include <string>

using namespace std;

#include "mummy.h"

//notice the use of the Mummy class
//Mummy::message_tomb()
//the Mummy class
void Mummy::message_tomb()
{
    cout << "\nunknown tomb" << endl;
}

void Mummy_queen::message_tomb()
{
    cout << "\nHere lies a queen" << endl;
}

void Mummy_king::message_tomb()
{
    cout << "\nHere lies a king" << endl;
}

mummy.h
#ifndef _MUMMY_A
#define _MUMMY_A

#include <iostream>
#include <string>

enum Mummy_A {
    UNKNOWN,
    QUEEN,
    KING
};

class Mummy {
public:
    Mummy_A type;
    string message;
    void message_tomb();
};

class Mummy_queen : public Mummy {
public:
    Mummy_queen(Mummy_A type) : Mummy(type) {}
    void message_tomb();
};

class Mummy_king : public Mummy {
public:
    Mummy_king(Mummy_A type) : Mummy(type) {}
    void message_tomb();
};

main.cpp
#include "mummy.h"
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string answer;
    Pyramid p1;

    Researcher r1("Hermine");

    while(true)
    {
        cout << "Looking around..." << endl;
        cin >> answer;

        if(answer == "exit")
        {
            break;
        }
    }
}
```

## mummy.h

```
#ifndef _MUMMY_A
#define _MUMMY_A

#include <iostream>
#include <string>

using namespace std;

class Mummy{
public:
    bool condition; //true=good or false=bad condition

    virtual void message_tomb();

};

//inherit the features of mummies
class Mummy_queen: public Mummy{

//override the function
public:
    void message_tomb();

};

class Mummy_king: public Mummy{
public:
    void message_tomb();
};

class Pyramid{

public:
    int num_treasures;
    Mummy *mum;

    Pyramid();

    ~Pyramid();

    void check_mum();
};

class Researcher{

    string name;
```

public:

Researcher(string name);

//overloaded functions. polymorphism

void found\_mummy(string gender, bool c, Pyramid &p);

void found\_mummy(bool c, Pyramid &p);

void found\_treasure(Pyramid &p);

};

#endif

-----

mummy.cpp

#include <iostream>

#include <string>

#include "mummy.h"

using namespace std;

//notice the use of the scope resolution operator

//Mummy::message\_tomb() means the message tomb function is in the scope of

//the Mummy class

void Mummy::message\_tomb()

```
{
    cout << "\nunknown tomb"<<endl;
}
```

void Mummy\_queen::message\_tomb()

```
{
    cout << "\nHere lies a queen."<<endl;
}
```

void Mummy\_king::message\_tomb()

```
{
    cout << "\nHere lies a king."<<endl;
}
```

Pyramid::Pyramid()

```
{
```

```
mum=NULL;
num_treasures=0;
}
```

```
Pyramid::~~Pyramid()
{
    delete (mum);
}
```

```
void Pyramid::check_mum()
{
    if(mum==NULL)
    {
        cout<<"No mummy found in this pyramid yet...\n"<<endl;
    }

    else
    {
        mum->message_tomb();
    }
}
```

```
Researcher::Researcher(string name)
{
    this->name=name;
}
```

```
void Researcher::found_mummy(string gender, bool c, Pyramid &p)
{
    if(gender=="m" || gender=="male")
    {
        p.mum=new Mummy_king();
        (p.mum)->condition=c;
    }

    else //assume female
    {
        p.mum=new Mummy_queen();
        (p.mum)->condition=c;
    }
}
```

```
void Researcher::found_mummy(bool c, Pyramid &p)
{
    p.mum=new Mummy();
    (p.mum)->condition=c;
}
```

```
void Researcher::found_treasure(Pyramid &p)
{
    cout << "Found a treasure!"<<endl;
    p.num_treasures+=1;
}
```

-----  
**main.cpp**

```
#include "mummy.h"
#include <iostream>
#include <string>

using namespace std;

int main(int argc, char **argv)
{
    string answer;
    Pyramid p1;

    Researcher r1("Hermine");

    while(true)
    {
        cout<<"Looking around..."<<endl;
        cin>>answer;

        if(answer=="exit")
        {
            break;
        }

        if(answer=="f")
        {
            cout<<"Gender? "<<endl;
            cin>>answer;

            if(answer=="unknown") //second overloaded function
            {
                cout<<"Good or bad condition?"<<endl;
                cin>>answer;

                if(answer=="good")
                {
                    r1.found_mummy(true, p1);
                }

                else
                {
                    r1.found_mummy(false, p1);
                }
            }
        }
    }
}
```

```

    }
}

else //gender known, assumed always in good condition
{
    r1.found_mummy(answer,true, p1);
}

}

if(answer=="t") //found Treasure
{
    r1.found_treasure(p1);
    cout<<"Total treasures so far: "<<p1.num_treasures<<"\n"<<endl;
}
}

p1.check_mum();

}

```