Contents

Polymorphism

- The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.
- Real life example of polymorphism, a person at a same time can have different characteristic. Like a man at a same time is a father, a husband, a employee.
- So a same person posses have different behavior in different situations. This is called polymorphism.

Types of Polymorphism

- Compile Time Polymorphism
- Run Time Polymorphism

Compile Time Polymorphism

- Early binding refers to events that occur at compile time. In essence, early binding occurs when all information needed to call a function is known at compile time. (Put differently, early binding means that an object and a function call are bound during compilation).
- Examples of early binding include normal function calls (including standard library functions), overloaded function calls, and overloaded operators.
- The main advantage to early binding is efficiency. Because all information necessary to call a function is determined at compile time, these types of function calls are very fast.

Run Time Polymorphism

- The opposite of early binding is late binding.
- As it relates to C++, late binding refers to function calls that are not resolved until run time.
- As function calls are not determined at compile time, the object and the function are not linked until run time.

Run Time Polymorphism

- The main advantage to late binding is flexibility. Unlike early binding, late binding allows you to create programs that can respond to events occurring while the program executes without having to create a large amount of "contingency code."
- Keep in mind that because a function call is not resolved until run time, late binding can make for somewhat slower execution times.
- Virtual functions are used to achieve late binding.

Function Overloading

- Function overloading is the process of using the same name for two or more functions.
- The secret to overloading is that each redefinition of the function must use either different types of parameters or a different number of parameters.
- It is only through these differences that the compiler knows which function to call in any given situation.

Example

```
#include <iostream>
using namespace std;
int myfunc(int i); // these differ in types of parameters
double myfunc(double i);
int main()
{
cout << myfunc(10) << " "; // calls myfunc(int i)
cout << myfunc(5.4); // calls myfunc(double i)
return 0;
}
```

```
 double myfunc(double i)
 {
 return i;
}
int myfunc(int i)
{
 return i;
 }
```

Output:
10   5.4

Example 2

```
#include <iostream>
using namespace std;
int myfunc(int i); // these differ in number of parameters
int myfunc(int i, int j);
int main()
{
cout << myfunc(10) << " "; // calls myfunc(int i)
cout << myfunc(4, 5); // calls myfunc(int i, int j)
return 0;
}
```

```
int myfunc(int i)
{
return i;
}
int myfunc(int i, int j)
{
return i*j;
}
```

```
Output:
10   20
```

Key Points

- The key point about function overloading is that the functions must differ in regard to the types and/or number of parameters. Two functions differing only in their return types cannot be overloaded.
- For example, this is an invalid attempt to overload **myfunc( ):**

```
int myfunc(int i);
float myfunc(int i);
```

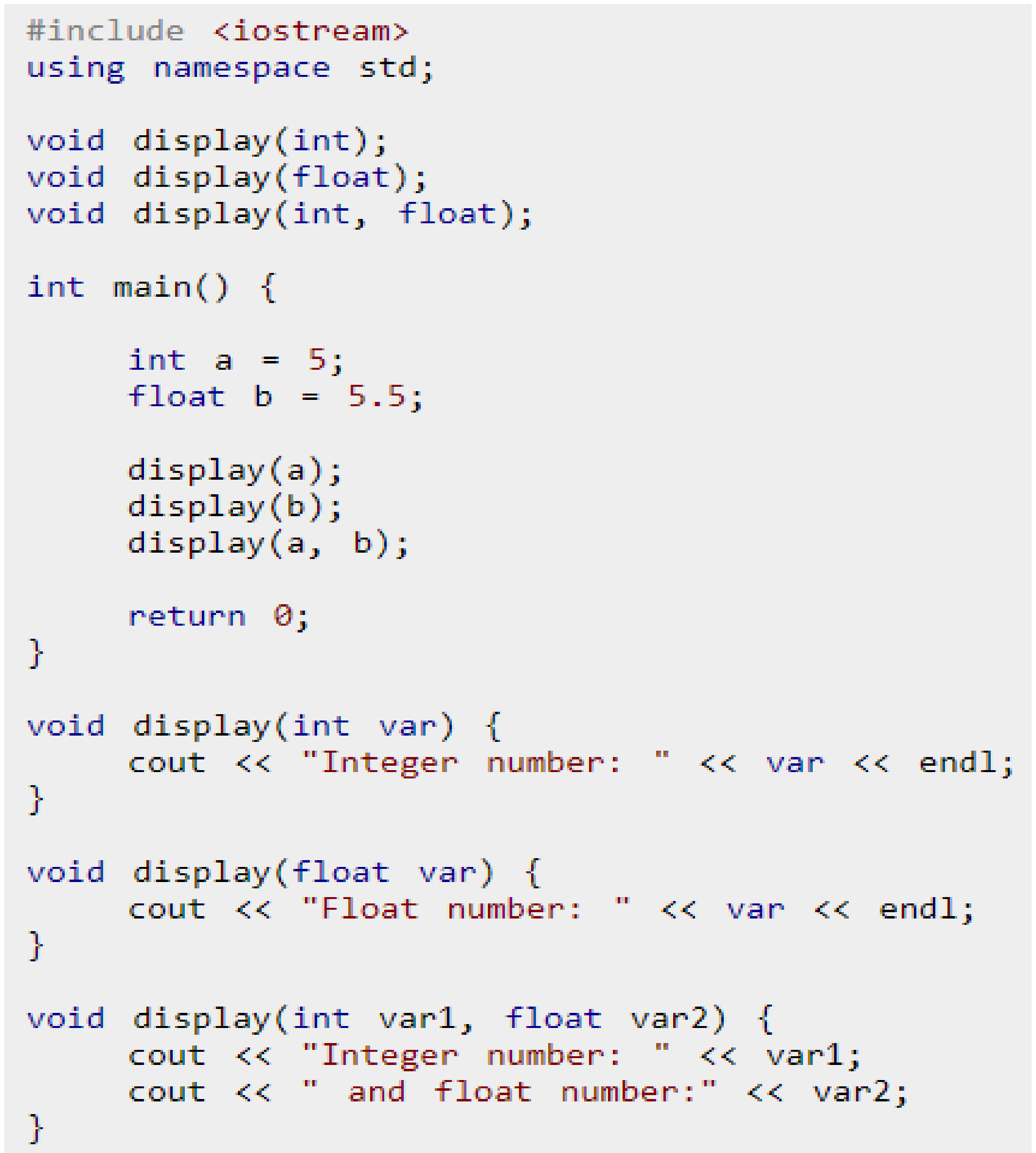
// Error: differing return types are insufficient when overloading.

- Sometimes, two function declarations will appear to differ, when in fact they do not. For example, consider the following declarations.

```
void f(int *p);
void f(int p[]); // error, *p is same as p[]
```

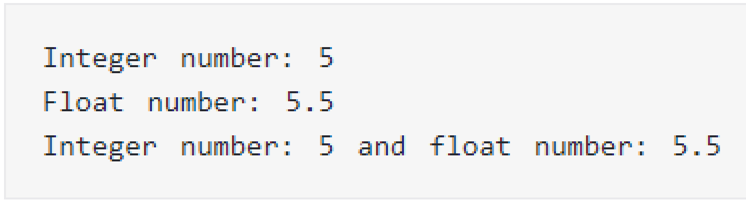
- Remember, to the compiler ***p is the same as p[ ]. Therefore, although the two** prototypes appear to differ in the types of their parameter, in actuality they do not.

Example 3



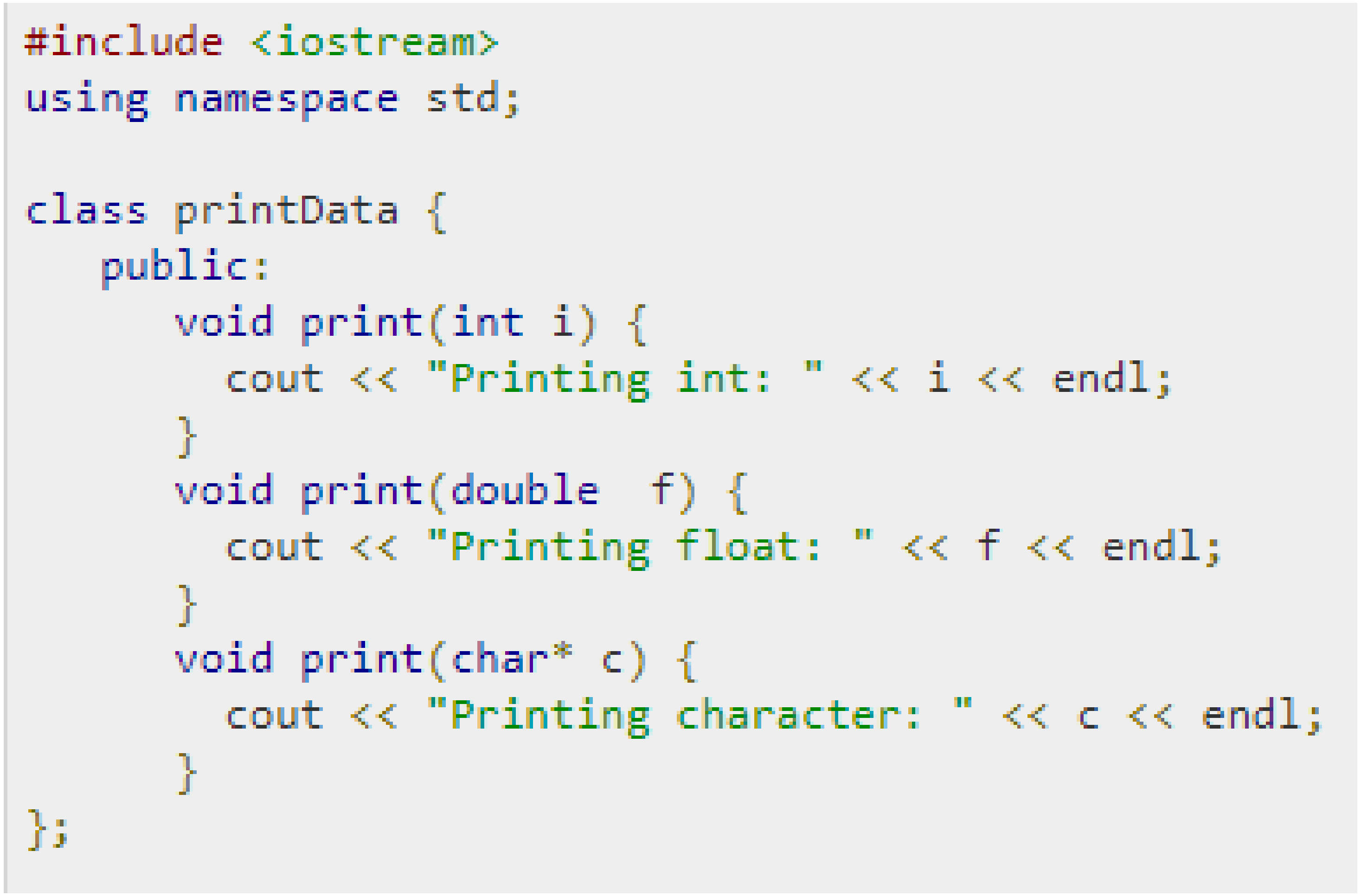
```cpp
#include <iostream>
using namespace std;

void display(int);
void display(float);
void display(int, float);

int main() {

     int a = 5;
     float b = 5.5;

     display(a);
     display(b);
     display(a, b);

     return 0;
}

void display(int var) {
     cout << "Integer number: " << var << endl;
}

void display(float var) {
     cout << "Float number: " << var << endl;
}

void display(int var1, float var2) {
     cout << "Integer number: " << var1;
     cout << " and float number:" << var2;
}
```

Output



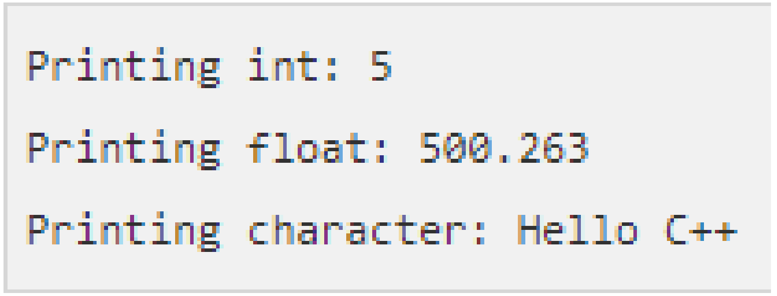
```
Integer number: 5
Float number: 5.5
Integer number: 5 and float number: 5.5
```

Example 4
(Function overloading in classes)



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

class printData {
   public:
      void print(int i) {
        cout << "Printing int: " << i << endl;
      }
      void print(double  f) {
        cout << "Printing float: " << f << endl;
      }
      void print(char* c) {
        cout << "Printing character: " << c << endl;
      }
};
```

```cpp
int main(void) {
   printData pd;

   // Call print to print integer
   pd.print(5);

   // Call print to print float
   pd.print(500.263);

   // Call print to print character
   pd.print("Hello C++");

   return 0;
}
```



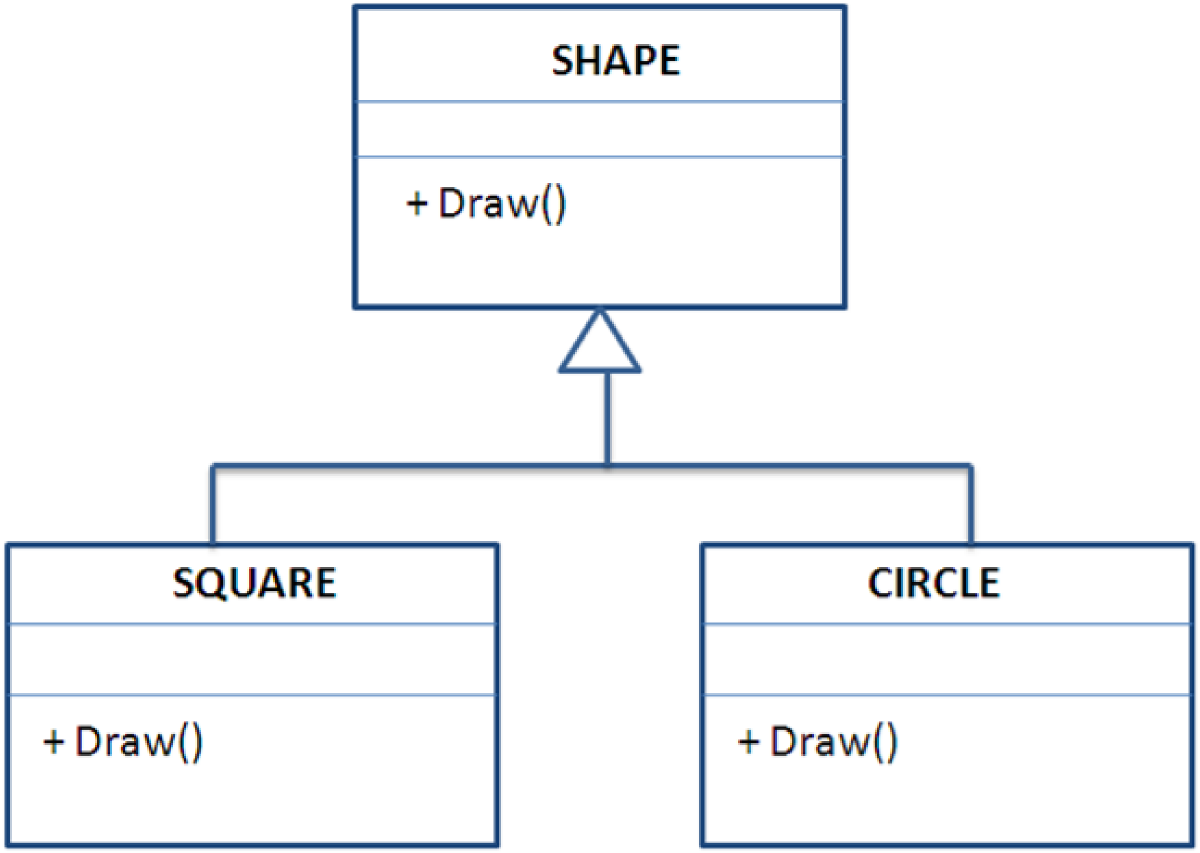
```
Printing int: 5
Printing float: 500.263
Printing character: Hello C++
```

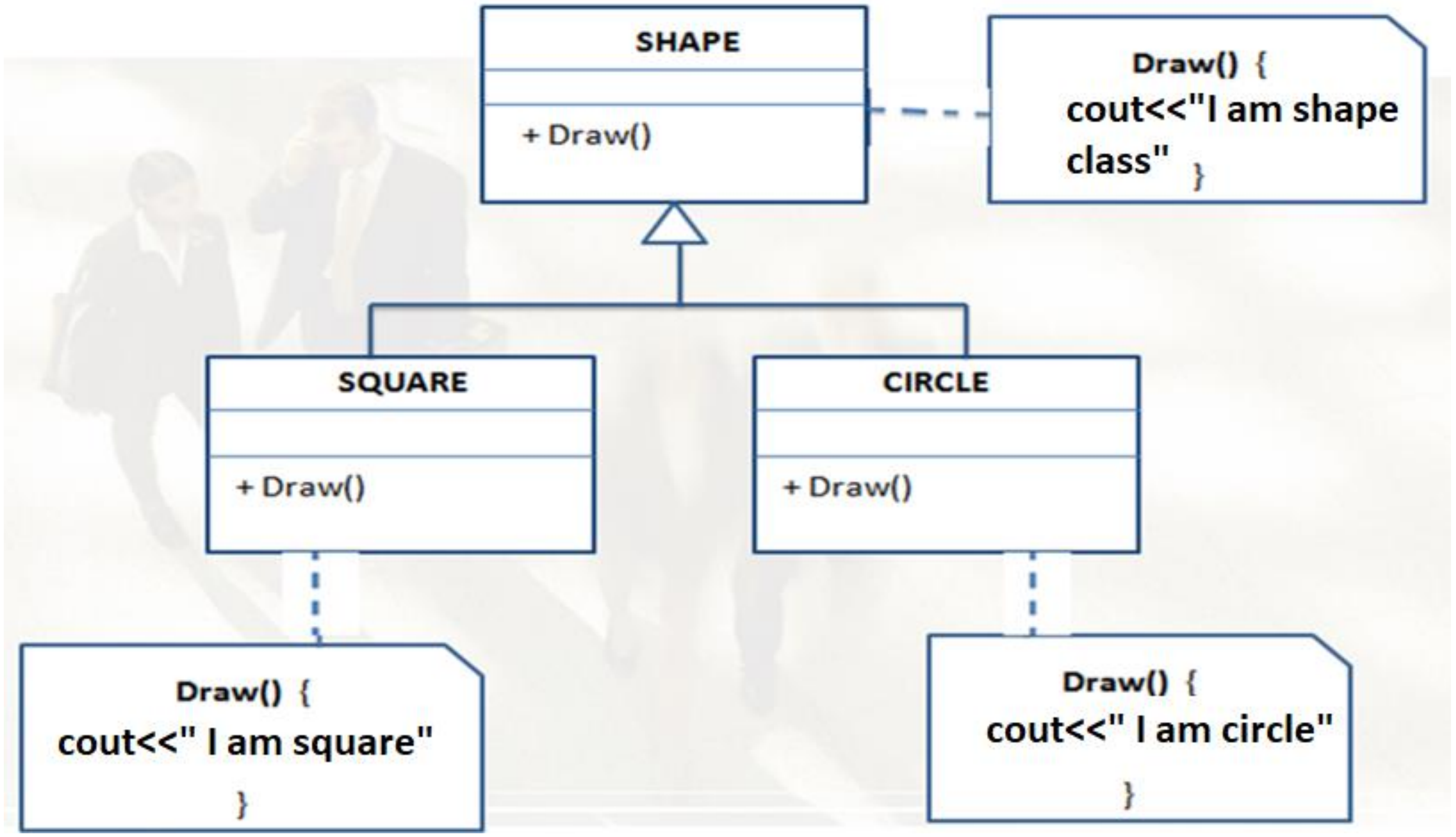
Inheritance based Polymorphism

Method overriding through class inheritance :

is a language feature that allows a subclass overrides (replaces) the implementation in the super class by providing a method that has same name, same parameters, and same return type as the method in the parent class.

Example

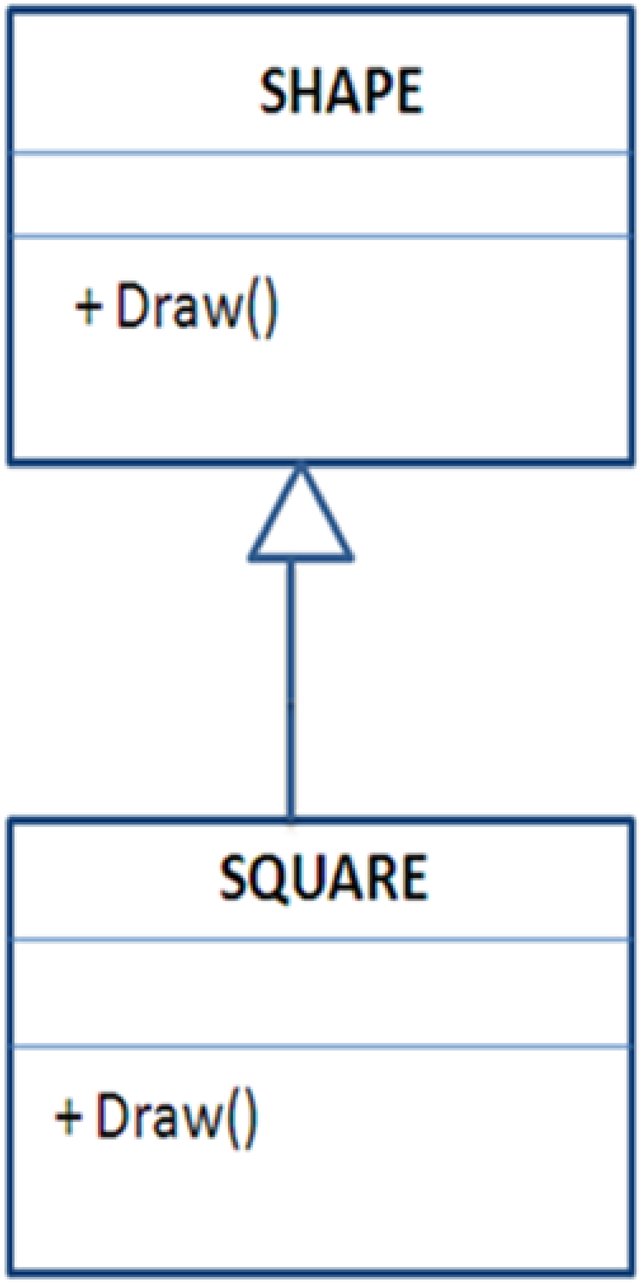




```
Draw() {
cout<<"I am shape
class" }
```

```
Draw() {
cout<<" I am square"
}
```

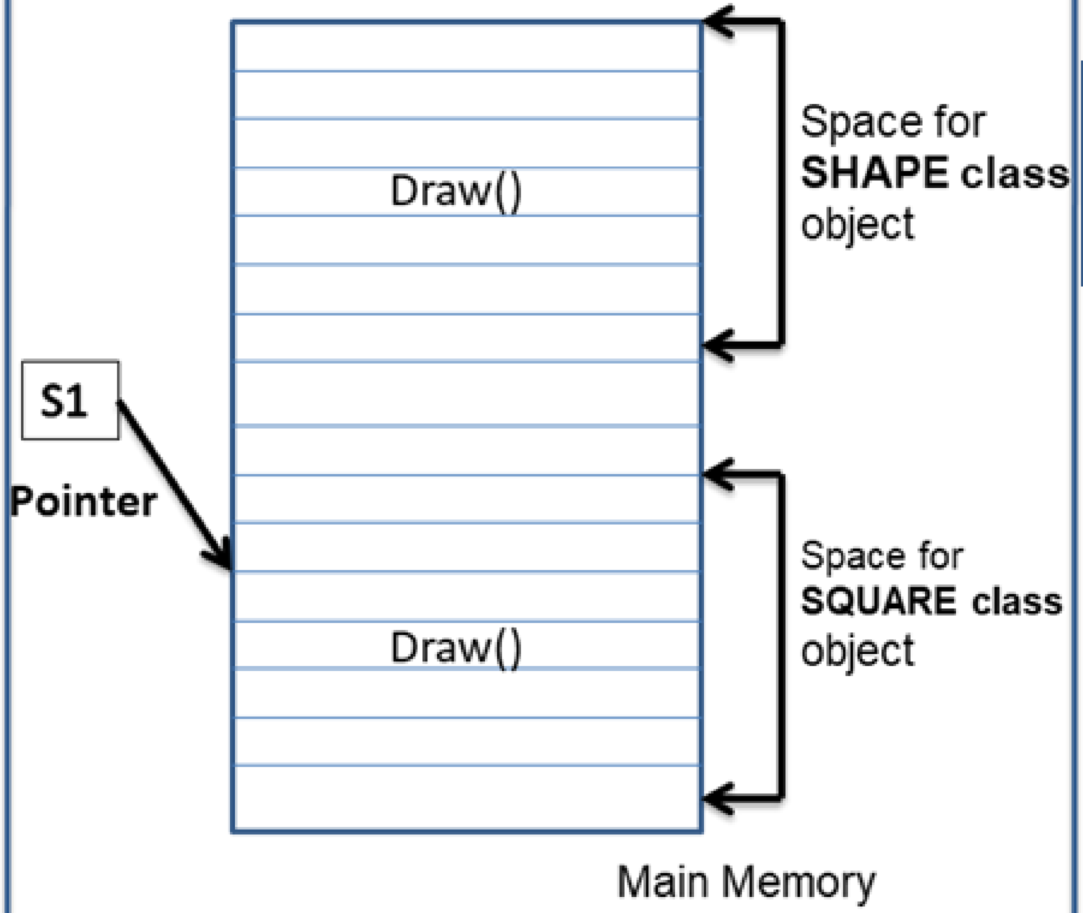
```
Draw() {
cout<<" I am circle"
}
```



```
Class SHAPE
{    public:
virtual void show()
{
  cout<<" I am shape
  class";
}
};

Class SQUARE : public SHAPE
{ public:
          void Draw()
{
cout<<"I am square";
}
};
```

Create Objects

```
SHAPE  *S1 = new SHAPE();

S1→Draw();


S1 = new SQUARE();

S1→Draw();
```

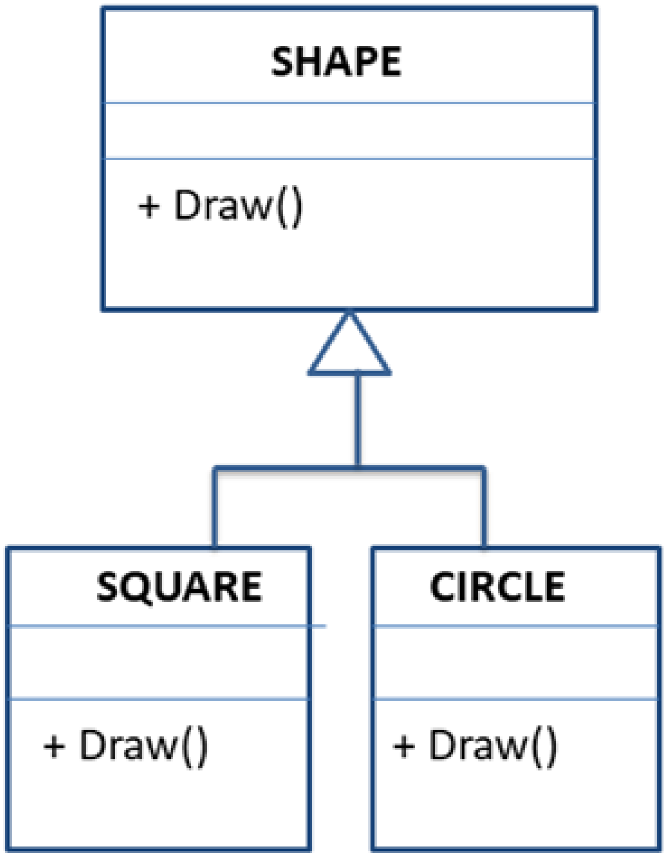


Output

I am shape class

I am Square

Extend the Logic



Create Objects

```
SHAPE *S1 = new SHAPE();
S1→Draw();

S1 = new SQUARE();
S1→Draw();

S1 = new CIRCLE();
S1→Draw();
```

Output

```
I am in
Shape class

I am
Square

I am
Circle
```

Class Activity (15 minutes)

- Can you showcase polymorphism (Drive function in various types of automobiles) in context of VEHICLES ?

Overloading Constructors

- Many times you will create a class for which there are two or more possible ways to construct an object.
- In these cases, you will want to provide an overloaded constructor for each way.
- This is a self-enforcing rule because if you attempt to create an object for which there is no matching constructor, a compile-time error results.

- Constructor can be overloaded in similar way as function overloading.

.

- Overloaded constructors have the same name (name of the class) but different number (or types or both) of arguments.

- Depending upon the number and type of arguments passed, specific constructor is called.

Example of Constructor Overloading

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

class example
{
   private:
      int a,b;
   public:
      example();     //Default Constructor
      example(int, int);    //Parameterized Constructor
      example(example &); //Copy Constructor
      void display();
};

example::example()
 {
    cout<<"Constructor is called";
  }

example::example(int x, int y)
 {
     a=x;
     b=y;
   }

example::example(example &p)
  {
      a=p.a;
      b=p.b;
    }
  void example::display()
    {
     cout<<a<<endl<<b;
     }
int main()
{
    example e1;
    example e2(4, 5);
    example e3(e2);
    e3.display();
    return 0;
}
```

Allowing Both Initialized and Uninitialized Objects

- Another common reason constructors are overloaded is to allow both initialized and uninitialized objects (or, more precisely, default initialized objects) to be created.
- This is especially important if you want to be able to create dynamic arrays of objects of some class, since it is not possible to initialize a dynamically allocated array.
- To allow uninitialized arrays of objects along with initialized objects, you must include a constructor that supports initialization and one that does not.

Example

```
#include <iostream>
#include <new>
using namespace std;
class powers {
int x;
public:
// overload constructor two ways
powers() { x = 0; } // no initializer
powers(int n) { x = n; } // initializer
int getx() { return x; }
void setx(int i) { x = i; }
};
int main()
{
powers ofTwo[] = {1, 2, 4, 8, 16}; //
    initialized
powers ofThree[5]; // uninitialized
powers *p;
int i;
// show powers of two
cout << "Powers of two: ";
for(i=0; i<5; i++) {
cout << ofTwo[i].getx() << " ";
}
cout << "\n\n";
// set powers of three
ofThree[0].setx(1);
ofThree[1].setx(3);
ofThree[2].setx(9);
ofThree[3].setx(27);
ofThree[4].setx(81);
// show powers of three
cout << "Powers of three: ";
for(i=0; i<5; i++) {
cout << ofThree[i].getx() << " ";
}
cout << "\n\n";
// dynamically allocate an array

p = new powers[5]; // no initialization
}
```

```
// initialize dynamic array with powers of
      two
for(i=0; i<5; i++) {
p[i].setx(ofTwo[i].getx());
}
// show powers of two
cout << "Powers of two: ";
for(i=0; i<5; i++) {
cout << p[i].getx() << " ";
}
cout << "\n\n";
delete [] p;
return 0;
}
```

Output:

```
Powers of two: 1 2 4 8 16
Powers of three: 1 3 9 27 81
Powers of two: 1 2 4 8 16
```

Copy Constructor

- A copy constructor is used to declare and initialize an object from another object.

- For example, the statement:

  example e2(e1);

  will define the object **e2** and at the same time initialize it to the value of **e1**.

- The process of initializing through a copy constructor is known as ***copy initialization***.

Example of Copy Constructor

```
#include<iostream>
using namespace std;
class example
{
  private:
    int a;
  public:
    example(int);    //Parameterized Constructor
    example(example &); //Copy Constructor
    void display();
};
example::example(int x)
    {
      a=x;
     }
example::example(example &p)
    {
      a=p.a;
     }
```

```
void example::display()
{
cout<<a;
}

int main()
{
   example e1(5);
   example e2(e1);   //or,  example e2=e1;
   e2.display();
   return 0;
}
```

OUTPUT

```
5
```

A reference variable has been used as an argument to the copy constructor.

We cannot pass the argument by value to a copy constructor.

Default Function Arguments

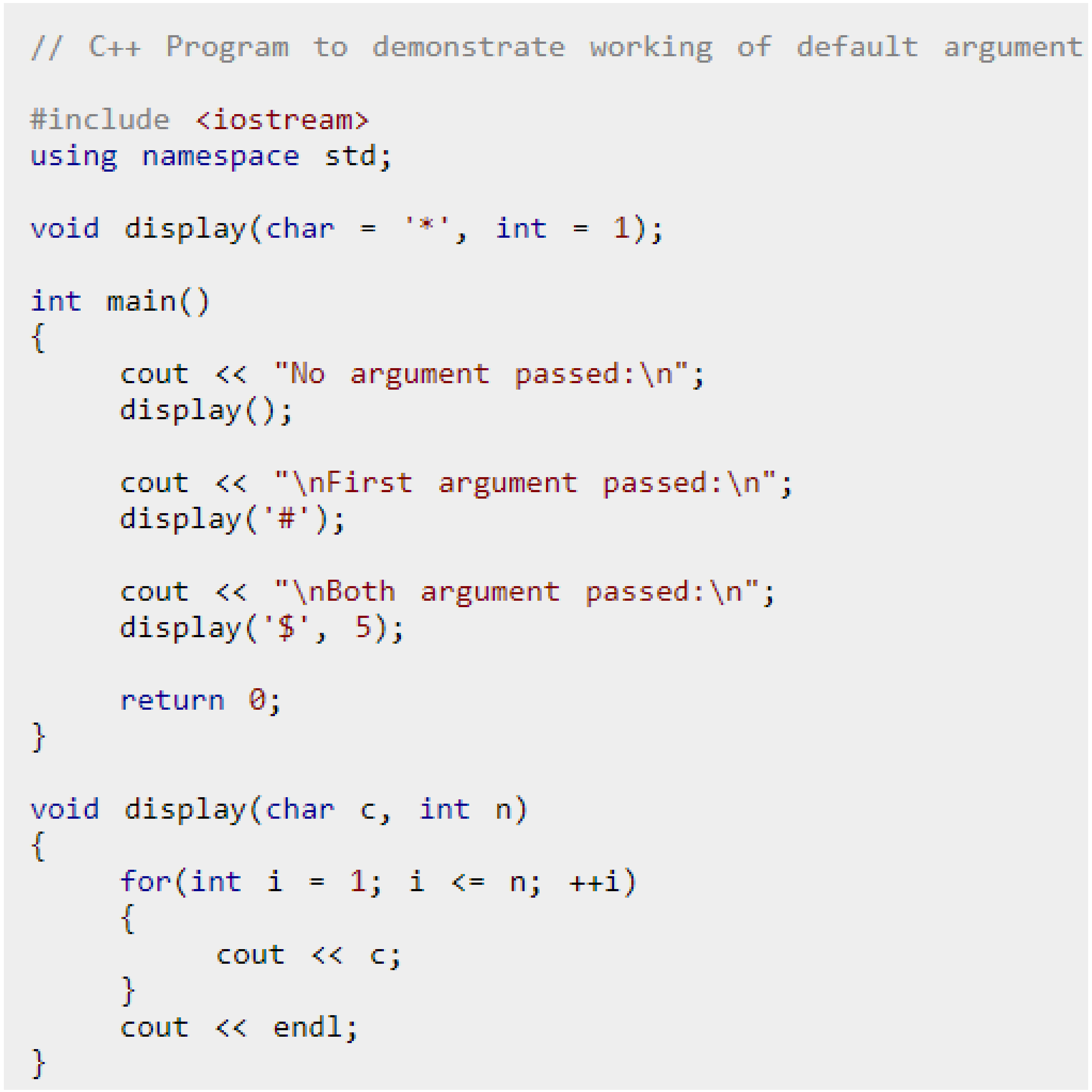
- C++ allows a function to assign a parameter a default value when no argument corresponding to that parameter is specified in a call to that function.
- For example, this declares myfunc( ) as taking one double argument with a default value of 0.0:

```
void myfunc(double d = 0.0)
{
// ...}

myfunc(198.234); // pass an explicit value
myfunc(); // let function use default
```

The first call passes the value 198.234 to d. The second call automatically gives d the default value zero.

- The idea behind default argument is simple. If a function is called by passing argument/s, those arguments are used by the function.
- But if the argument/s are not passed while invoking a function then, the default values are used.
- Default value/s are passed to argument/s in the function prototype.

Example



```cpp
// C++ Program to demonstrate working of default argument

#include <iostream>
using namespace std;

void display(char = '*', int = 1);

int main()
{
	cout << "No argument passed:\n";
	display();

	cout << "\nFirst argument passed:\n";
	display('#');

	cout << "\nBoth argument passed:\n";
	display('$', 5);

	return 0;
}

void display(char c, int n)
{
	for(int i = 1; i <= n; ++i)
	{
		cout << c;
	}
	cout << endl;
}
```

Output

```
No argument passed:
*

First argument passed:
#

Both argument passed:
$$$$$
```

Rules for using Default Arguments

1. Only the last argument must be given default value. You cannot have a default argument followed by non-default argument.

```
sum (int x,int y);
sum (int x,int y=0);
sum (int x=0,int y);  // This is Incorrect
```

2. If you default an argument, then you will have to default all the subsequent arguments after that.

```
sum (int x,int y=0);
sum (int x,int y=0,int z);  // This is incorrect
sum (int x,int y=10,int z=10);  // Correct
```

3. You can give any value a default value to argument, compatible with its datatype.

Common Mistakes when using Default Arguments

1. `void add(int a, int b = 3, int c, int d = 4);`

   The above function will not compile. You cannot miss a default argument in between two arguments.
   In this case, `c` should also be assigned a default value.

2. `void add(int a, int b = 3, int c, int d);`

   The above function will not compile as well. You must provide default values for each argument after `b`.
   In this case, `c` and `d` should also be assigned default values.
   If you want a single default argument, make sure the argument is the last one. `void add(int a, int b, int c, int d = 4);`

3. No matter how you use default arguments, a function should always be written so that it serves only one purpose.
   If your function does more than one thing or the logic seems too complicated, you can use Function overloading to separate the logic better.

Ambiguity in Function Overloading

- You can create a situation in which the compiler is unable to choose between two (or more) overloaded functions.
- When this happens, the situation is said to be *ambiguous.*
- Ambiguous statements are errors, and programs containing ambiguity will not compile.

Ambiguity in Function Overloading

```
int myfunc(double d);
// ...
cout << myfunc('c');
// not an error, conversion applied
```

- Although automatic type conversions are convenient, they are also a prime cause of ambiguity.

Example

```
#include <iostream>
using namespace std;
float myfunc(float i);
double myfunc(double i);

int main()
{
cout << myfunc(10.1) << " "; // unambiguous, calls
    myfunc(double)
cout << myfunc(10); // ambiguous
return 0;
}
```

Thanks