Chapter-5 (Default Arguments and Ambiguity) Teach yourself C++, Herbert Schildt

PREPARED BY: LEC TASMIAH TAMZID ANANNYA, CS DEPT, AIUB

Default Argument

```
int totalMark (int phy, int chem){
        int total = phy+chem;
        return total;
}
int totalMark (int phy, int chem, int opt){
        int total = phy+chem+opt;
        return total;
}
void main(){
        cout << totalMark (80,90)<<endl;
        cout << totalMark (80,90,80);
}
```

Default Argument

```
int totalMark (int phy, int chem){
        int total = phy+chem;
        return total;
}
int totalMark (int phy, int chem, int opt){
        int total = phy+chem+opt;
        return total;
}
void main(){
        cout << totalMark (80,90)<<endl;
        cout << totalMark (80,90,80);
}
```

```
int totalMark (int phy, int chem,  int opt = 0)
{
    int total = phy+chem+opt;
    return total;
}


void main(){
    cout << totalMark (80,90)<<endl;
    cout << totalMark (80,90,80);
}
```

Default Argument

Default Argument

```
int totalMark (int phy, int chem){
        int total = phy+chem;
        return total;
}
int totalMark (int phy, int chem, int opt){
        int total = phy+chem+opt;
        return total;
}
void main(){
        cout << totalMark (80,90)<<endl;
        cout << totalMark (80,90,80);
}
```

```
int totalMark (int phy, int chem,  int opt = 0)
{
    int total = phy+chem+opt;
    return total;
}

void main(){
    cout << totalMark (80,90)<<endl;
    cout << totalMark (80,90,80);
}
```

Default Arguments - Restriction

Default Arguments - Restriction

```
int totalMark (int opt = 0, int phy, int chem){
 int total = phy+chem+opt;
 return total;
}
```

Default Arguments - Restriction

Error !
Must be to the right of any parameter that don't have defaults

```
int totalMark (int opt = 0, int phy, int chem){
  int total = phy+chem+opt;
  return total;
}
```

Default Arguments - Restriction

Error !
Must be to the right of any parameter that don't have defaults

```
int totalMark (int opt = 0, int phy, int chem){
 int total = phy+chem+opt;
 return total;
}
```

```
int totalMark (int phy, int chem, int opt=0);
int main(){…}
int totalMark (int phy, int chem, int opt = 0){
      return phy+chem+opt;
}
```

Default Arguments - Restriction

Error !
Must be to the right of any parameter that don't have defaults

```
int totalMark (int opt = 0, int phy, int chem){
 int total = phy+chem+opt;
 return total;
}
```

Error !
Default can't be specified both in prototype and definition

```
int totalMark (int phy, int chem, int opt=0);
int main(){…}
int totalMark (int phy, int chem, int opt = 0){
      return phy+chem+opt;
}
```

Default Arguments - Restriction

Error !
Must be to the right of any parameter that don't have defaults

```
int totalMark (int opt = 0, int phy, int chem){
 int total = phy+chem+opt;
 return total;
}
```

Error !
Default can't be specified both in prototype and definition

```
int totalMark (int phy, int chem, int opt=0);
int main(){…}
int totalMark (int phy, int chem, int opt = 0){
      return phy+chem+opt;
}
```

```
int totalMark (int phy, int chem, int opt = phy){
     return phy+chem+opt;
}
```

Default Arguments - Restriction

Error !
Must be to the right of any parameter that don't have defaults

```
int totalMark (int opt = 0, int phy, int chem){
 int total = phy+chem+opt;
 return total;
}
```

Error !
Default can't be specified both in prototype and definition

```
int totalMark (int phy, int chem, int opt=0);
int main(){…}
int totalMark (int phy, int chem, int opt = 0){
      return phy+chem+opt;
}
```

Error !
Default arguments must be constant or global variables

```
int totalMark (int phy, int chem, int opt = phy){
     return phy+chem+opt;
}
```

Example-Default Argument

```
int totalMark (int phy=0,  int opt = 0)
{
      int total = phy+opt;
      return total;
}
int main(){
cout<<totalMark();
cout << totalMark (80);
cout << totalMark (80,90);
}
```

Output:

Example-Default Argument

```
int totalMark (int phy=0,  int opt = 0)
{
      int total = phy+opt;
      return total;
}
int main(){
cout<<totalMark();
cout << totalMark (80);
cout << totalMark (80,90);
}
```

Output:

0

80

170

Example-Default Argument

```
int totalMark (int phy=0,  int opt = 0)
{
      int total = phy+opt;
      return total;
}
int main(){
cout<<totalMark();
cout << totalMark (80);
cout << totalMark (80,90);
}
```

Output:

0

80

170

```
int totalMark (int phy=0,  int opt) {
      int total = phy+opt;
      return total;
}
```

Example-Default Argument

```
int totalMark (int phy=0,  int opt = 0)
{
      int total = phy+opt;
      return total;
}
int main(){
cout<<totalMark();
cout << totalMark (80);
cout << totalMark (80,90);
}
```

Output:

0

80

170

```
int totalMark (int phy=0,  int opt) {
      int total = phy+opt;
      return total;
}
```

//ERROR!
Must be to the right of any parameter that don't have defaults

Example-Default Argument

```
double rect_area(double length, double width)
{
   return length*width;
}
double rect_area(double length)
{
   return length*length;
}
int main()
{
   cout<<rect_area(10.0, 5.8)<<endl;
   cout<<rect_area(10.0);
}
```

Example-Default Argument

```
double rect_area(double length, double width)
{
   return length*width;
}
double rect_area(double length)
{
   return length*length;
}
int main()
{
   cout<<rect_area(10.0, 5.8)<<endl;
   cout<<rect_area(10.0);
}
```

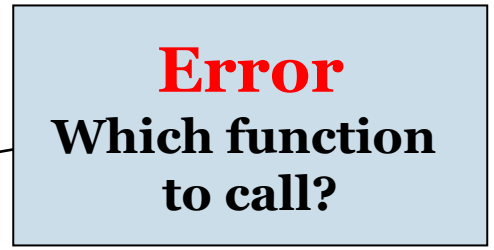
```
double rect_area(double length, double width=0)
{
   if(width!=0)
      return length*width;
   else
      return length*length;
}
int main()
{
   cout<<rect_area(10.0, 5.8)<<endl;
   cout<<rect_area(10.0);
}
```

Overloading and ambiguity- Default arguments

```
int ambigFun (int x) {cout<<x<<endl;}
int ambigFun (int x, int y=0) {cout<<x<<endl<<y<<endl;}
main(){
    int i,j;
    ambigFun (i,j); // O.K
    ambigFun (i);
}
```

Overloading and ambiguity- Default arguments

```
int ambigFun (int x) {cout<<x<<endl;}
int ambigFun (int x, int y=0) {cout<<x<<endl<<y<<endl;}
main(){
    int i,j;
    ambigFun (i,j); // O.K
    ambigFun (i);
}
```

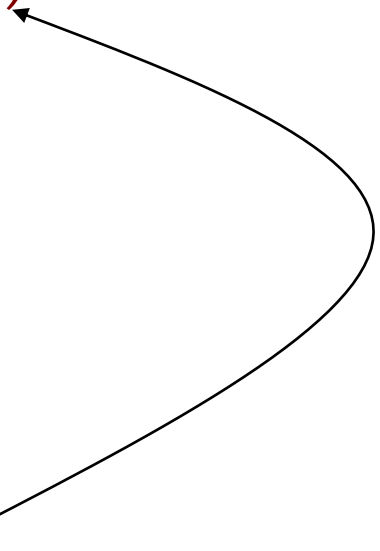


Overloading and ambiguity- automatic type conversion

```
float ambigFun(float f)
{  return f;}
int main()
{   int i=1;
    float f=2.3323;
    cout<<ambigFun(f)<<endl;
    cout<<ambigFun(i);   }
```

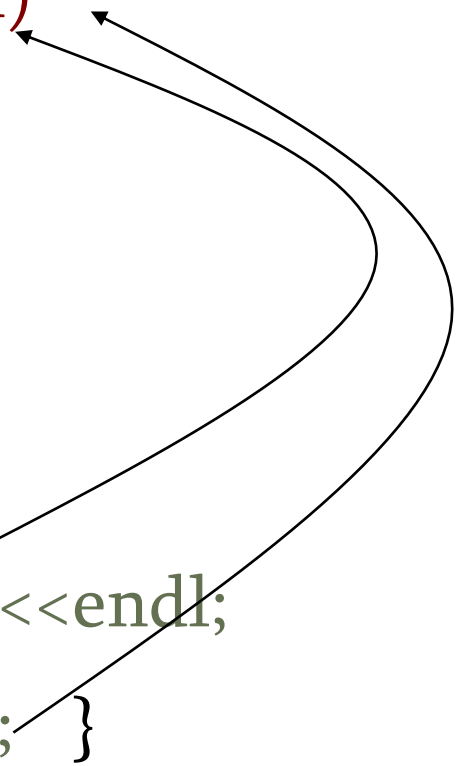
Overloading and ambiguity- automatic type conversion

```
float ambigFun(float f)
{  return f;}
int main()
{   int i=1;
    float f=2.3323;
    cout<<ambigFun(f)<<endl;
    cout<<ambigFun(i);   }
```



Overloading and ambiguity- automatic type conversion

```
float ambigFun(float f)
{  return f;}
int main()
{   int i=1;
    float f=2.3323;
    cout<<ambigFun(f)<<endl;
    cout<<ambigFun(i);   }
```

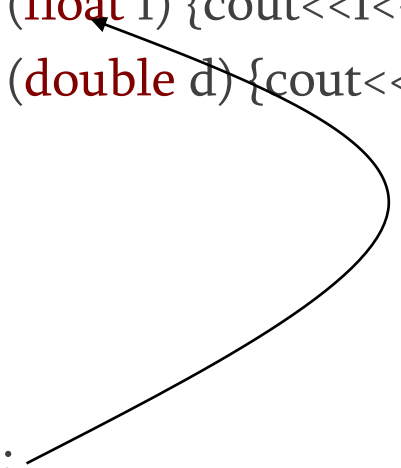


Overloading and ambiguity- automatic type conversion

```
int ambigFun (float f) {cout<<f<<endl;}
int ambigFun (double d) {cout<<d<<endl;}
main(){
 int i;
 float f ;
 double d;
 ambigFun (f);
 ambigFun (d);
 ambigFun (i);
 ambigFun (15);
}
```

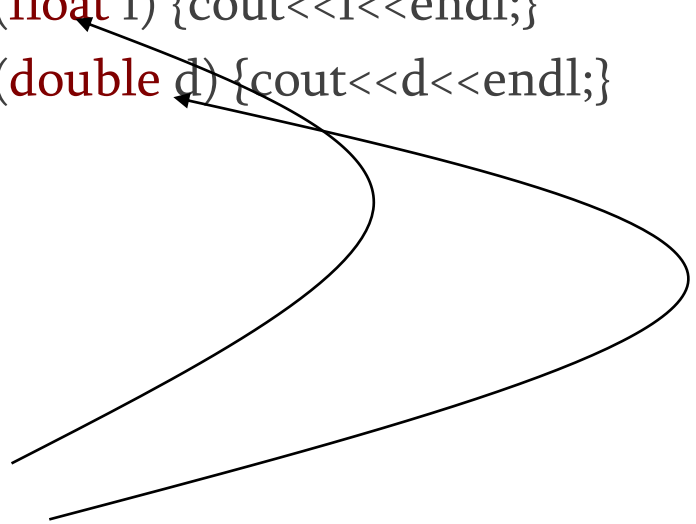
Overloading and ambiguity- automatic type conversion

```
int ambigFun (float f) {cout<<f<<endl;}
int ambigFun (double d) {cout<<d<<endl;}
main(){
 int i;
 float f ;
 double d;
 ambigFun (f);
 ambigFun (d);
 ambigFun (i);
 ambigFun (15);
}
```



Overloading and ambiguity- automatic type conversion

```
int ambigFun (float f) {cout<<f<<endl;}
int ambigFun (double d) {cout<<d<<endl;}
main(){
 int i;
 float f ;
 double d;
 ambigFun (f);
 ambigFun (d);
 ambigFun (i);
 ambigFun (15);
}
```



Overloading and ambiguity- automatic type conversion

```
int ambigFun (float f) {cout<<f<<endl;}
int ambigFun (double d) {cout<<d<<endl;}
main(){
 int i;
 float f ;
 double d;
 ambigFun (f);
 ambigFun (d);
 ambigFun (i);
 ambigFun (15);
}
```

Constructor and Destructor Functions-Chapter 2.1 and 2.2

Teach yourself C++, Herbert Schildt

PREPARED BY: LEC TASMIAH TAMZID ANANNYA, CS DEPT, AIUB

Constructor Functions

- A constructor is a member function of a class which initializes objects of a class.
- In C++, Constructor is automatically called when object(instance of class) is created.
- It is special member function of the class.

Constructor Functions

- A constructor is a member function of a class which initializes objects of a class.
- In C++, Constructor is automatically called when object(instance of class) is created.
- It is special member function of the class.

How constructors are different from a normal member function?

- A constructor has the same name as the class.
- It does not have a return type.
- A class's constructor is called each time an object of that class is created.

Default Constructors

- **Default constructor is the constructor which doesn't take any argument. It has no parameters.**

```
class construct
{
public:
    int a, b;

        // Default Constructor
    construct() ;
};

construct::construct()
{
        cout<<"Constructing…"<<endl;
        a = 10;
        b = 20;  }

int main()
{
     // Default constructor called automatically when the object is created
    construct c;
    cout << "a: "<< c.a << endl << "b: "<< c.b;
    return 0;
}
```

Default Constructors

- **Default constructor is the constructor which doesn't take any argument. It has no parameters.**

```
class construct
{
public:
    int a, b;

        // Default Constructor
    construct() ;
};

construct::construct()
{
        cout<<"Constructing…"<<endl;
        a = 10;
        b = 20;  }

int main()
{
     // Default constructor called automatically when the object is created
    construct c;
    cout << "a: "<< c.a << endl << "b: "<< c.b;
    return 0;
}
```

Constructing…
a:10
b:20

Default Constructors

Notice how contruct() is defined.

```
construct::construct()
{
      cout<<“Constructing…”<<endl;
      a = 10;
      b = 20;
   }
```

It has no return type.

It is illegal for a constructor to have a return type.

Parameterized Constructors

- It is possible to pass arguments to constructors.
- Typically, these arguments help initialize an object when it is created.
- To create a parameterized constructor, simply add parameters to it the way you would to any other function.
- When you define the constructor's body, use the parameters to initialize the object.

Parameterized Constructors

```
class Point
{
      int x, y;

   public:

      // Parameterized Constructor

      Point(int x1, int y1)
      {
         x = x1;
         y = y1;
      }

      int getX()   { return x; }

      int getY()   {return y;}

};
```

```
int main()

{

   // Constructor called

   Point p1(10, 15);



   // Access values assigned by constructor

cout << "p1.x = " << p1.getX() << ", p1.y = " <<
p1.getY();

   return 0;

}
```

Parameterized Constructors

- When an object is declared in a parameterized constructor, the initial values have to be passed as arguments to the constructor function.

In the previous example:

```
int main()
{
   // Constructor called
   Point p1;     //error because the object does not pass any parameter to the constructor
   // Access values assigned by constructor
cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
   return 0;
}
```

Parameterized Constructors

You can also send any variables in the constructor.

```
class Point
{
    private:
        int x, y;

public:

        // Parameterized Constructor

        Point(int x1, int y1)
        {
            x = x1;
            y = y1;
         }

        int getX()        {  return x;  }

        int getY()        {   return y; }
    };
```

```
int main()
{
     int a,b;
     cin>>a>>b;
    // Constructor called
    Point p1(a, b);


    // Access values assigned by constructor
cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
    return 0;
}
```

Note:

For global objects, an object’s constructor is called once, when the prgram first begins execution.

For local objects, the constructor is called each time the declaration statement is executed.

Destructors

- The complement of a constructor is called destructor.
- Destructor is a member function which destructs or deletes an object.

- **When is destructor called?**

  A destructor function is called automatically when the object goes out of scope:
  (1) the function ends
  (2) the program ends
  (3) a block containing local variables ends

- **How destructors are different from a normal member function?**
  - Destructors have same name as the class preceded by a tilde (~)
  - Destructors don’t take any argument and don’t return anything

Constructors and Destructors

```
class Line {
     double length;
   public:
     double getLength( );
     Line(double len);   // This is the constructor declaration
     ~Line();  // This is the destructor: declaration
};
Line::Line(double len) {
    length=len;
    cout << "Object is being created" << endl;}
Line::~Line(void) {
   cout << "Object is being deleted" << endl;}
double Line::getLength() {return length;}
```

```
int main() {
 Line line(6.0);
 cout << "Length of line : " << line.getLength();
 return 0;
}
```

Constructors and Destructors

```
class Line {
     double length;
   public:
     double getLength( );
     Line(double len);   // This is the constructor declaration
     ~Line();  // This is the destructor: declaration
};
Line::Line(double len) {
    length=len;
    cout << "Object is being created" << endl;}
Line::~Line(void) {
   cout << "Object is being deleted" << endl;}
double Line::getLength() {return length;}
```

```
int main() {
 Line line(6.0);
 cout << "Length of line : " << line.getLength();
 return 0;
}
```

Output:

Object is being created

Length of line : 6

Object is being deleted

Practice on Constructor and Destructor

- Create a class called **box** whose constructor function is passed three double values, length, width and height of the box. Have the **box** class compute the volume of the box and store it in another **double** variable. Include a member function of the class called **show_volume()** that will display the volume of each box.

- Practice all the previous examples using constructors instead of **setvalues** functions.