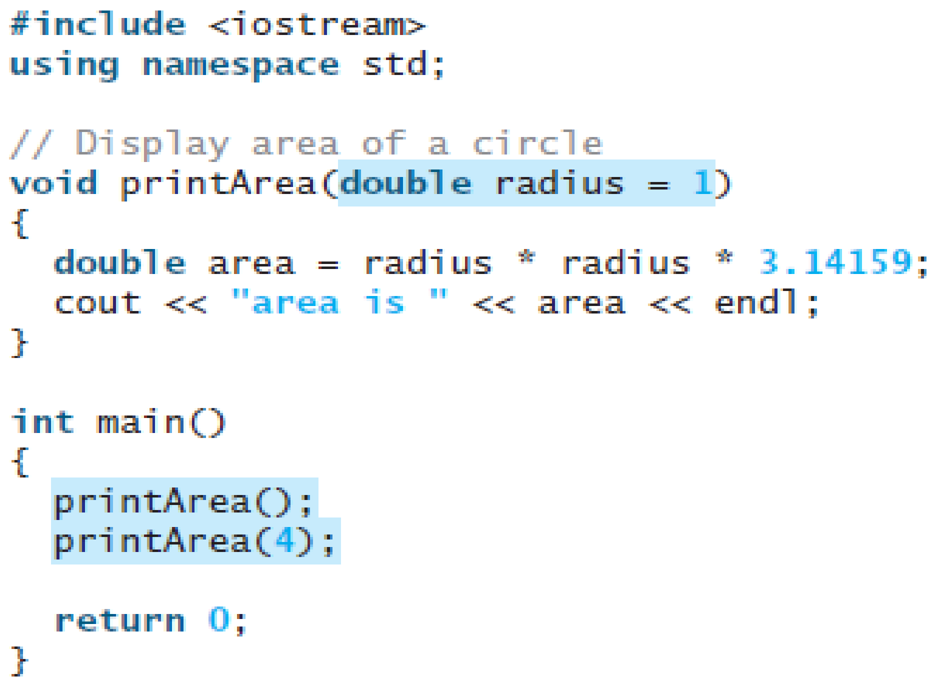
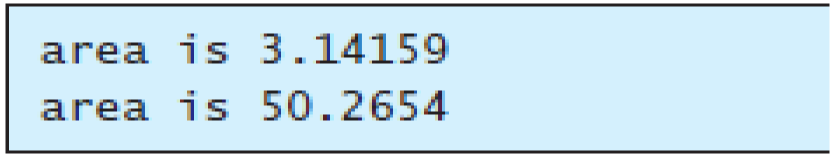
Function Overloading, Inline Functions & Default Arguments

Default Arguments

- *You can define default values for parameters in a function.*
- C++ allows you to declare functions with default argument values. The default values are passed to the parameters when a function is invoked without the arguments.



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

// Display area of a circle
void printArea(double radius = 1)
{
  double area = radius * radius * 3.14159;
  cout << "area is " << area << endl;
}

int main()
{
  printArea();
  printArea(4);

  return 0;
}
```



```
area is 3.14159
area is 50.2654
```

Default Arguments

- When a function contains a mixture of parameters with and without default values, those with default values must be declared last. For example, the following declarations are illegal:
  - ❑void t1(int x, int y = 0, int z); // Illegal
  - ❑void t2(int x = 0, int y = 0, int z); // Illegal
  - ❑However, the following declarations are fine:
  - ❑void t3(int x, int y = 0, int z = 0); // Legal
  - ❑void t4(int x = 0, int y = 0, int z = 0); // Legal
- When an argument is left out of a function, all arguments that come after it must be left out as well. For example, the following calls are illegal:

  t3(1, , 20);

  t4(, , 20);

  but the following calls are fine:

  t3(1); // Parameters y and z are assigned a default value

  t4(1, 2); // Parameter z is assigned a default value

Activity

- Which of the following function declarations are illegal?

```
void t1(int x, int y = 0, int z);
void t2(int x = 0, int y = 0, int z);
void t3(int x, int y = 0, int z = 0);
void t4(int x = 0, int y = 0, int z = 0);
```

Function Overloading

- *Overloading functions enables you to define the functions with the same name as long as their signatures are different.*
- Overloading functions can make programs clearer and more readable. Functions that perform the same task with different types of parameters should be given the same name.
- Overloaded functions must have different parameter lists. You cannot overload functions based on different return types.

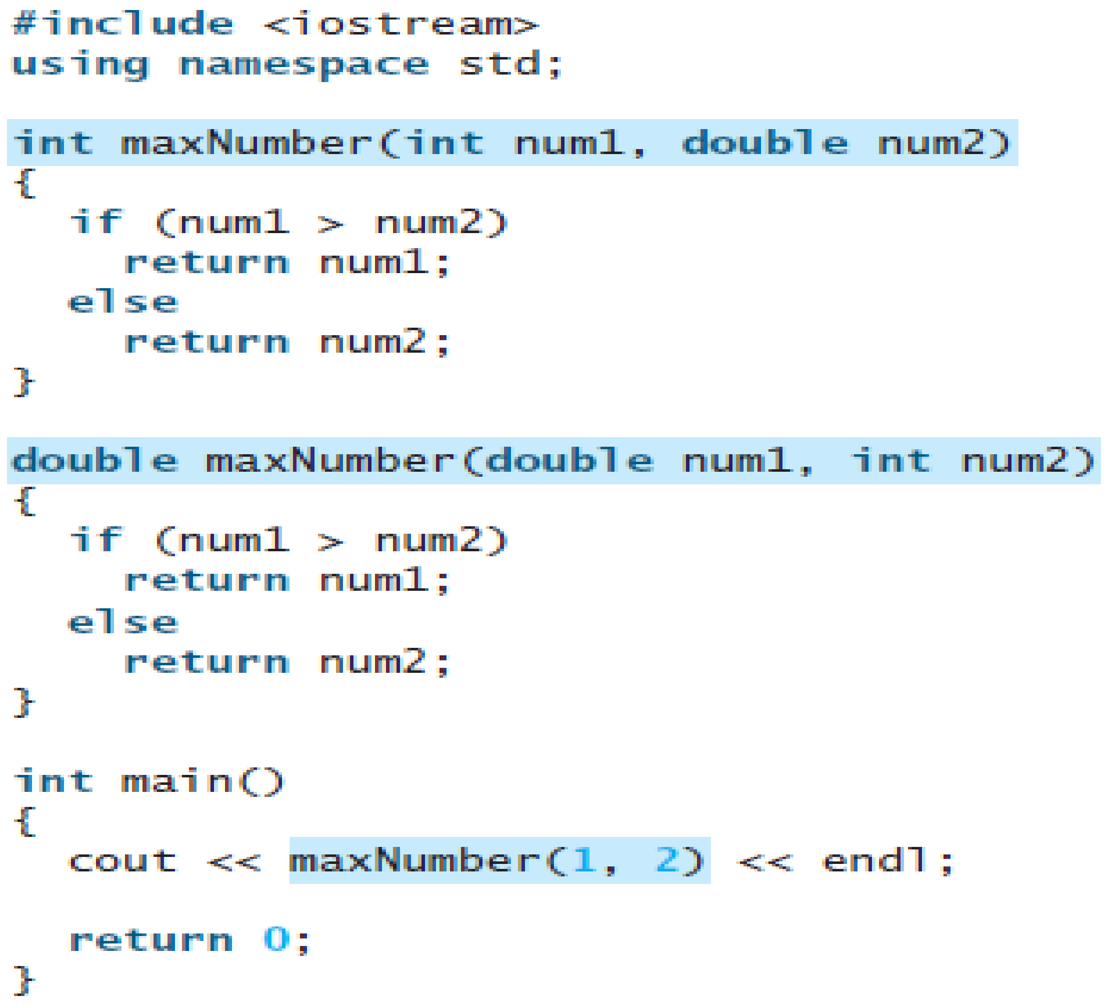
```cpp
#include <iostream>
using namespace std;

// Return the max between two int values
int max(int num1, int num2)
{
  if (num1 > num2)
    return num1;
  else
    return num2;
}

// Find the max between two double values
double max(double num1, double num2)
{
  if (num1 > num2)
    return num1;
  else
    return num2;
}

// Return the max among three double values
double max(double num1, double num2, double num3)
{
  return max(max(num1, num2), num3);
}

```

```
int main()
{
  // Invoke the max function with int parameters
  cout << "The maximum between 3 and 4 is " << max(3, 4) << endl;

  // Invoke the max function with the double parameters
  cout << "The maximum between 3.0 and 5.4 is "
    << max(3.0, 5.4) << endl;

  // Invoke the max function with three double parameters
  cout << "The maximum between 3.0, 5.4, and 10.14 is "
    << max(3.0, 5.4, 10.14) << endl;

  return 0;
}
```

Ambiguous invocation

- Sometimes there are two or more possible matches for an invocation of a function, and the compiler cannot determine the most specific match. This is referred to as *ambiguous invocation*.
- Ambiguous invocation causes a compile error. Consider the following code:

Ambiguous invocation



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

int maxNumber(int num1, double num2)
{
  if (num1 > num2)
    return num1;
  else
    return num2;
}

double maxNumber(double num1, int num2)
{
  if (num1 > num2)
    return num1;
  else
    return num2;
}

int main()
{
  cout << maxNumber(1, 2) << endl;

  return 0;
}
```

Activity

What is wrong in the following program?

```
void p(int i)
{
  cout << i << endl;
}

int p(int j)
{
  cout << j << endl;
}
```

Given two function definitions,

```
double m(double x, double y)
double m(int x, double y)
```

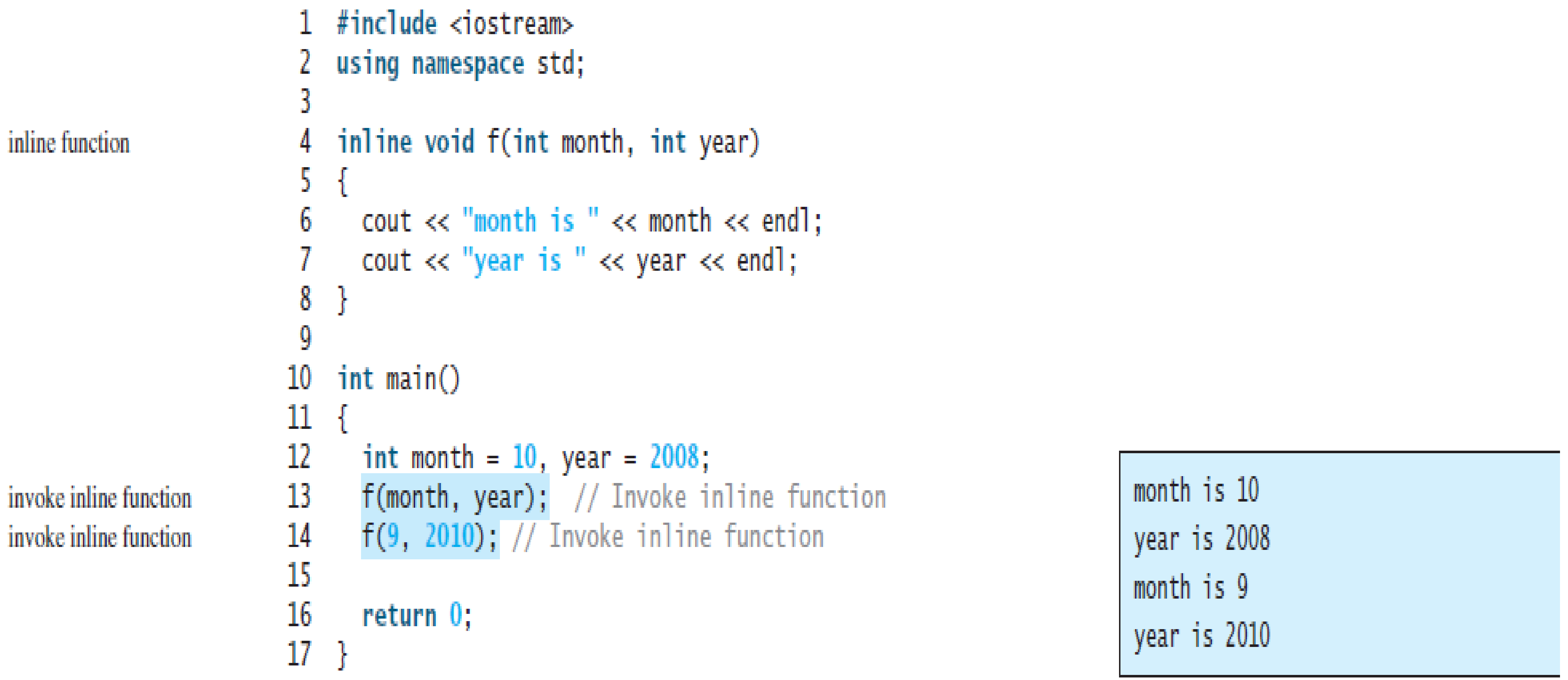
answer the following questions:

a. Which of the two functions is invoked for
```
   double z = m(4, 5);
```
b. Which of the two functions is invoked for
```
   double z = m(4, 5.4);
```
c. Which of the two functions is invoked for
```
   double z = m(4.5, 5.4);
```

Inline Functions

- *C++ provides inline functions for improving performance for short functions.*
- Implementing a program using functions makes the program easy to read and easy to maintain, but function calls involve runtime overhead (i.e., pushing arguments and CPU registers into the stack and transferring control to and from a function).
- C++ provides *inline functions* to avoid function calls. Inline functions are not called; rather, the compiler copies the function code *in line* at the point of each invocation. To specify an inline function, precede the function declaration with the inline keyword,

Inline Functions



inline function

invoke inline function

invoke inline function

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

inline void f(int month, int year)
{
  cout << "month is " << month << endl;
  cout << "year is " << year << endl;
}

int main()
{
  int month = 10, year = 2008;
  f(month, year);  // Invoke inline function
  f(9, 2010); // Invoke inline function

  return 0;
}
```

```
month is 10
year is 2008
month is 9
year is 2010
```

Inline Functions

- As far as programming is concerned, inline functions are the same as regular functions, except they are preceded with the inline keyword.
- However, behind the scenes, the C++ compiler expands the inline function call by copying the inline function code.
- Inline functions are desirable for short functions but not for long ones that are called in multiple places in a program,
- Making multiple copies will dramatically increase the executable code size. For this reason, C++ allows the compilers to ignore the inline keyword if the function is too long.
- The inline keyword is merely a request; it is up to the compiler to decide whether to honor or ignore it.