Lecture 11

Advanced Functions

Shibo Li
shiboli@cs.fsu.edu

Department of Computer Science
Florida State University

The slides are mainly from Sharanya Jayaraman

Function Overloading

The term **function overloading** refers to the way C++ allows more than one function in the same scope to share the same name—as long as they have **different parameter lists**

- The rationale is that the compiler must be able to look at any function call and decide exactly which function is being invoked
- Overloading allows intuitive function names to be used inmultiple contexts

Function Overloading

- The parameter list can differ in number of parameters, ortypes of parameters, or both
- **Name mangling/Name Decoration**. The basic idea is that the compiler encodes the function's name along with its parameter types into a unique name, making it distinct from other overloaded functions.

```
int Process(double num); // function 1
int Process(char letter); // function 2
int Process(double num, int position); // function 3
```

Function Overloading

- The parameter list can differ in number of parameters, ortypes of parameters, or both
- **Name mangling/Name Decoration**. The basic idea is that the compiler encodes the function's name along with its parameter types into a unique name, making it distinct from other overloaded functions.

```cpp
int Process(double num); // function 1
int Process(char letter); // function 2
int Process(double num, int position); // function 3
```

Sample calls, based on the above declarations

```
int x;
float y = 12.34;

x = Process(3.45, 12); // invokes function 3
x = Process('f'); // invokes function 2
x = Process(y); // invokes function 1
//(automatic type conversion applies)
```

Function Polymorphism

Function Overloading $\neq$ Function Overidding

- multiple functions with similar functionality
- Functions are in the same class/scope
- params MUST differ
- compile-time
- handle different types or amounts of data

- a specific implementation in a subclass
- Functions are in base and derived classes
- params Must be identical
- Runtime
- To modify or extend base class behavior

Default Parameters

Allows a function to have default values for parameters

- Specify default values in function declaration.
- Rules
  - Default values must be provided from right to left.
  - Once a parameter has a default value, all subsequent parameters must have defaults.

```
void display(int x, int y = 10, int z = 20) {
   cout << x << " " << y << " " << z << endl;
}

display(1);       // Output: 1 10 20
display(1, 2);    // Output: 1 2 20
```

Ambiguities of Function Overloading

Even with legally overloaded functions, it's possible to make ambiguous function calls, largely due to

- Automatic type conversion
- default paramters

Avoiding Ambiguity

Example 0:

```
// Overloaded functions
void func(int x) {
    cout << "func(int) called." << endl;
}

void func(double x) {
    cout << "func(double) called." << endl;
}

int main() {
    func(0);
    return 0;
}
```

Avoiding Ambiguity

Example 0:

```
// Overloaded functions
void func(int x) {
    cout << "func(int) called." << endl;
}

void func(double x) {
    cout << "func(double) called." << endl;
}

int main() {
    func(10.0);
    return 0;
}
```

Avoiding Ambiguity

Example 1:

```
// Overloaded functions
void func(int x) {
    cout << "func(int) called." << endl;
}

void func(double x) {
    cout << "func(double) called." << endl;
}

int main() {
    func('A');
    return 0;
}
```

Avoiding Ambiguity

Example 1: Ambiguity due to type promotion

```
// Overloaded functions
void func(int x) {
    cout << "func(int) called." << endl;
}

void func(double x) {
    cout << "func(double) called." << endl;
}

int main() {
    func('A'); // No errors, first one will be called.
    return 0;
}
```

Avoiding Ambiguity

Example 2:

```
// Overloaded functions
void func(int x) {
    cout << "func(int) called." << endl;
}

void func(float x) {
    cout << "func(float) called." << endl;
}

int main() {
    func(10.5);
    return 0;
}
```

Example 2: Ambiguity due to type conversion

```
// Overloaded functions
void func(int x) {
    cout << "func(int) called." << endl;
}

void func(float x) {
    cout << "func(float) called." << endl;
}

int main() {
    func(10.5); // Error: Ambiguous call
    return 0;
}
```

Example 3:

```
void func(long x) {
    cout << "func(long) called." << endl;
}

void func(double x) {
    cout << "func(double) called." << endl;
}

int main() {
    func(100);
    return 0;
}
```

Avoiding Ambiguity

Example 3: Ambiguity due to type promotion

```
void func(long x) {
    cout << "func(long) called." << endl;
}

void func(double x) {
    cout << "func(double) called." << endl;
}

int main() {
    func(100); // Error: Ambiguous call
    return 0;
}
```

Avoiding Ambiguity

Example 4: due to default parameters

```cpp
// Overloaded functions
void func(int x, float y = 3.14) {
    cout << "func(int, float) called" << endl;
}

void func(int x) {
    cout << "func(int) called" << endl;
}

int main() {
    func(5);
    return 0;
}
```

Example 4:

```
// Overloaded functions with default parameters
void func(int x, float y = 3.14) {
    cout << "func(int, float) called" << endl;
}

void func(int x) {
    cout << "func(int) called" << endl;
}

int main() {
    func(5); // Ambiguous call: which overload should be
        called?
    return 0;
}
```

Avoiding Ambiguity

Example 5: Ambiguity Due to Promotion and Conversion of Mixed Data Types

```cpp
// Overloaded functions with different parameter types
void func(float x, double y) {
    cout << "func(float, double) called." << endl;
}

void func(double x, float y) {
    cout << "func(double, float) called." << endl;
}

int main() {
    func(1, 2); // Error: Ambiguous call
    return 0;
}
```

General Rule to Avoid Ambiguity

To avoid ambiguity:

- Use function overloading with **clearly distinct parameter types** that do not require implicit type conversions.
- Be cautious when using default parameters in conjunction with overloading.
- Use **type casting in the function call** to explicitly specify which overloaded function you want to invoke:

```
func(static_cast<int>(10.5)); // Calls func(int)
func(static_cast<float>(100)); // Calls func(float)
```

Reference Variables

- A reference is an alias(nickname) for another variable. It is created using the & symbol.
  - Must be initialized at the time of declaration.

```
int x = 10;
int &ref = x; // ref is a reference to x
// x, ref are both referring to the SAME
    storage location
cout << x << endl;
cout << ref << endl;
```

```
10
10
```

Reference Variables

- No separate memory is allocated for references.

```
int x = 10;
int &ref = x; // ref is a reference to x
x += 1;
cout << x << endl;
cout << ref << endl;
```

```
11
11
```

Reference Variables

- No separate memory is allocated for references.

```
int x = 10;
int &ref = x; // ref is a reference to x
x += 1;
cout << &x << endl; // When & is not used
    after a data type, it means address-of
    operator
cout << &ref << endl;
```

```
0x7ffc37b3bd14
0x7ffc37b3bd14
```

Reference Variables

- Note: The notation can become confusing when different sources place the & differently. The following three declarations are equivalent:

```
int &r = n;
int& r = n;
int & r = n;
```

The spacing between the "int" and the "r" is irrelevant. All three of these declare r as a reference variable that refers to n.

Why Reference?

```
int x = 10;
int &ref = x; // ref is a reference to x
```

- While the above code example shows what a reference variable is, you will not likely use it this way!
- In this example, the regular variable and the reference are in the same scope, so it seems silly. ("Why do I need to call it r when I can call it x ?")
- So when are references useful?

Why Reference?

- Avoids copying large structures
  - Passing function parameters by reference to avoid unnecessary copies.
- Allows modification of variables passed to a function.
  - Two variables are in different scopes (this means functions)!

Pass by Value

- Recall that the variables in the formal parameter list are always local variables of a function
- This is known as Pass By Value - function parameters receive **copies** of the data sent in.

```
int addOne(int a) {
   return a+=1;
}

int main() {
   int a = 1;
   cout << addOne(a) << endl; // ?
   cout << a << endl; // ?
}
```

Pass by Value

- Recall that the variables in the formal parameter list are always local variables of a function
- This is known as Pass By Value - function parameters receive **copies** of the data sent in.

```
int addOne(int a) {
   return a+=1; // will not affect the caller
}

int main() {
   int a = 1;
   cout << addOne(a) << endl; // 2
   cout << a << endl; // 1
}
```

```
int addOne(int &a) {
   return a+=1;
}

int main() {
   int a = 1;
   cout << addOne(a) << endl; // ?
   cout << a << endl; // ?
}
```

Pass by Reference

```
int addOne(int &a) {
   return a+=1; // DO change the caller!
}

int main() {
   int a = 1;
   cout << addOne(a) << endl; // 2
   cout << a << endl; // 2
}
```

- When reference variables are used as formal parameters, this is known as Pass By Reference
- Parameters passed by are still local to the function, but they are **reference variables** (i.e., oringal variables)
- `int& func()` return reference also possible

Comparing: Value vs. Reference

- Pass By Value
  - The local parameters are copies of the original argumentspassed in
  - Changes made in the function to these variables do not affect originals
- Pass By Reference
  - The local parameters are references to the storage locations ofthe original arguments passed in.
  - Changes to these variables in the function will affect the originals
  - No copy is made, so overhead of copying (time, storage) is saved

const Reference Parameters

- The keyword const can be used on reference parameters.

```
void func(const int& x)
```

- This will prevent x from being changed in the function body
- This would be used to avoid the overhead of making a copy,but still prevent the data from being changed

Exercise: Estimate π through harmonic series

$$\frac{\pi}{4} = \sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{2k-1} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots$$

- Prompt to the user for entering k.
- You should validate the user's inputs to ensure $n \geq 1$. You prompt the user indefinitely until the user enters valid inputs.
- Print out the approximated π with order k.

```
Enter the order you want to approximate PI: 0
ERROR: invalid n! try again!
Enter the order you want to approximate PI: 10
Approximate PI with order of 10 is: 3.04184
```

```
Enter the order you want to approximate PI: 10000
Approximate PI with order of 10000 is: 3.14149
```