Lecture 3.2

Topics

---

1. Functions with Default Argument – Polymorphism

- In C++, a given default argument will assign a default value when no corresponding argument with specific value is specified as the function is called.
- To give the default value to a parameter, each parameter must be followed by an equal sign and the value to be defaulted to.

It is important to remember that the default arguments should take **constant values** or values from some **global variables** and they **cannot** take values from local variables or other parameters.

Furthermore, the following rules must be observed:

a. When a function with one or more default arguments is created, these arguments and values must be specified all at once. This can be done either in function prototype or in function heading but not both even if the values are duplicates.

b. All default values, if any, must be given to the right of the last parameter that does not have default value.

c. Once beginning to define default parameters, there may not be any parameters to the right of the last defaulted parameter that have no defaults.

Example

```
/**
 *Program Name: cis25L0321.cpp
 *Discussion:   Default Arguments
 */

#include <iostream>
using namespace std;

int product( int = 1, int = 1 );

int main( void ) {
  int x;

  x = product();
  cout << "\nValue of x = param1 * param2 : " << x << "\n";
  x = product( 5 );
  cout << "\nValue of x = param1 * param2 : " << x << "\n";
  x = product( 5, 6 );
  cout << "\nValue of x = param1 * param2 : " << x << "\n";

  return 0;
}

/**
 *Function Name:  product()
 *Description:    Computing the product of two integers
 *Pre:            Two integers with default values
 *Post:           returning the product
 */
int product( int a, int b ) {
  cout << "\nValue of the first parameter is " << a
       << "\nValue of the second parameter is " << b;
  return ( a * b );
}
```

OUTPUT

```
Value of the first parameter is 1
Value of the second parameter is 1
Value of x = param1 * param2 : 1

Value of the first parameter is 5
Value of the second parameter is 1
Value of x = param1 * param2 : 5

Value of the first parameter is 5
Value of the second parameter is 6
Value of x = param1 * param2 : 30
```

Note!

1) The default values are specified (given) starting from right to left for each function prototype (or function definition but not both).
2) There cannot be any unspecified (parameter) values to the right of a specified one.

And also, when making a call to a function with arguments,

a. The actual arguments will be assigned to the formal arguments starting from left to right.
b. The default values (if there are) will be used if there are not enough argument values being specified in the function call.

This default approach gives the user flexible options in acquiring the function execution. However, overusing default arguments can cause ambiguity and errors to compilation.

2. Function Polymorphism – Overloading

Recall that a function in C++ should be declared before its definition is given. The declaration will be as follows,

```
ReturnType functionName( Type1 arg1, Type2 arg2 );
```

where

`ReturnType` is the type of the value to be returned by the function.
`functionName` is the name of the function.
`Type1` is the type of the argument by the name of `arg1`.
`Type2` is the type of the argument by the name of `arg2`.

Again, every function is said to have a function signature, which includes the function name and its argument list. The function signature does not include the return type.

Function overloading is one of the polymorphic forms where a function name can be reused in several implementations. This means that different definitions are defined and available under the same name. Having many such definitions is great but how can one use these versions? What are the rules for distinguishing each version of this overloaded function?

Using the functions is up to the user and based on its specific definition. However, it is important to be able to call the correct version of the overloaded function and not the others. Here are the rules for doing just that; each version of the overloaded function will be distinguished by:

(i) Number of arguments from the argument list.

(ii) Types of arguments.

Remember that **FUNCTION RETURN TYPE** is not a deciding factor and should not be used to select a version of the overloaded function.

Examples of how to avoid ambiguity in function overloading will clarify these rules and will be shown shortly.

3. Function Polymorphism – Ambiguity

There are times where a version of the overloaded functions may be called in incorrectly. The error is not intentional but instead of ambiguous function calls resulted from the arguments of function and how the function called.

This problem is termed as ambiguity in function overloading (this ambiguity may hang the system!).

Definition

Ambiguity in function overloading occurs when the compiler does not know which version of an overloaded function to call.

Recall that the overloaded function **should NOT rely** just on the return type for correct call. Instead, the number of arguments and the argument types are the deciding factors in making a correct call of one version of the overloaded function.

But in many cases, the use of default arguments may also introduce additional ambiguity.

Regarding default arguments of function, although they are powerful and convenient, they can be misused. Some general rules may be helpful in deciding whether default argument needed as follows,

- Default arguments tend to mislead anyone who has to use the function.
- Default arguments are used in specific order and allow variations in function calls.
- Only use default argument if the default makes sense. If the value has no real meaning or no preference over any other value then it may NOT be a good choice to have default value at all.
- If a particular argument is needed frequently each time the function to be called, then it may be desirable to give it a default value.
- DO NOT OVERUSE default arguments.

The following example will demonstrate some possible ambiguities resulted from function overloading. In this example, the automatic type (data) conversion rules are applied when the function is called with an argument that is compatible but not of the same type. Refer to the type promotion or data conversion rules in earlier lectures.

```
/**
 *Program Name: cis25L0322.cpp
 *Discussion:   Default Arguments
 *              Automatic Type Conversion (Type promotion)
 */

#include <iostream>
using namespace std;

float foo( float x ) {
  return ( x * 2.0 );
}

double foo( double x ) {
  return ( x * 3.0 );
}

int main( void ) {
  float z = 5.0;
  double y = 10.0;

  cout << foo( z );         //The float version of foo() is called
  cout << foo( y );         //The double version of foo() is called
  cout << foo( 15 );        //Which conversion should take place?

  return 0;
}
```

```
1>Compiling...
1>cis25L0322.cpp
1>e:\...\cis25\code\cis25l0322.cpp(11) : warning C4244: 'return' : conversion from
'double' to 'float', possible loss of data
1>e:\ ...\cis25\code\cis25l0322.cpp(24) : error C2668: 'foo' : ambiguous call to
overloaded function
1>        e:\ ...\cis25\code\cis25l0322.cpp(14): could be 'double foo(double)'
1>        e:\ ...\cis25\code\cis25l0322.cpp(10): or       'float foo(float)'
1>        while trying to match the argument list '(int)'
1>Build log was saved at "file://e:\
...\cis25\code\cis25Project001\cis25Project001\Debug\BuildLog.htm"
1>cis25Project001 - 1 error(s), 1 warning(s).
```

There is no ambiguity in the definitions of the two overloaded versions of the function `foo()`. However, this does not prevent problem at hand in converting a value as argument for a function.

Clearly, the function call to `foo( 15 )` produces ambiguity. The constant `15` must be converted to one of higher order types that, in this case, is either `float` or `double`. Which one?

So there is the problem. This problem can be easily avoided by typecasting as,

```
cout << foo( static_cast< float >( 15 ) );
cout << foo( static_cast< double >( 15 ) );
```

or by sending the variable as argument in the above example.

We will visit the topics of function overloading and polymorphism frequently throughout the semester.

4. Derived Data Type: References

4.1 Definition

What is a **reference variable** (or just **reference**)? A reference variable is a derived-type variable that just simply acts like another name for a **previously defined variable**.

4.2 Kinds of References

There are three ways that a reference can be created for:

(i) An independent reference,
(ii) A reference being passed to a function,
(iii) A reference being returned from a function.

A reference must be initialized as soon as it is created unless it is one of the following:

(i) Member of class, or
(ii) Return value from function, or
(iii) Parameter reference for functions.

Using an independent reference with its ordinary way in a program is generally discouraged and SHOULD BE AVOIDED.

However, using references in functions (arguments and return value) are more suggestive and preferred. It is important to understand passing references to functions. The discussion on returning a reference from a function will be given in later lectures (when overloading I/O operators).

For now, let’s consider a brief discussion of “Independence Reference Variable”.

4.3 Independent Reference Variable

Again, even though an independent reference variable can be defined, its use in “straight” coding is not recommended because there is no need to create two names for the same location; it only adds more confusion.

The following example elaborates on this point.

Example

```
/**
 *Program Name: cis25L0323.cpp
 *Discussion:   Reference
 */
#include <iostream>
using namespace std;

int main( void ) {
  int iX = 1;

  int& iXAlias = iX;

  cout << "\nAddress of :"
       << "\n\t\tiX is " << &iX
       << "\n\t\tiXAlias is " << &iXAlias
       << "\n\nValue of :"
       << "\n\t\tiX is " << iX
       << "\n\t\tiXAlias is " << iXAlias << endl;
  cout << "\nAssigning new value to iXAlias." << endl;
  iXAlias = 10;
  cout << "\nAddress of :"
       << "\n\t\tiX is " << &iX
       << "\n\t\tiXAlias is " << &iXAlias
       << "\n\nValue of :"
       << "\n\t\tiX is " << iX
       << "\n\t\tiXAlias is " << iXAlias << endl;
  return 0;
}
```

OUTPUT

```
Address of :
                iX is 0012FF7C
                iXAlias is 0012FF7C

Value of :
                iX is 1
                iXAlias is 1

Assigning new value to iXAlias.

Address of :
                iX is 0012FF7C
                iXAlias is 0012FF7C

Value of :
                iX is 10
                iXAlias is 10
```

Note that in this example `iX` and `iXAlias` have, for all purposes, the same values. If one is modified so is the other.

In addition, the following restrictions should be observed:

(1) You may not obtain address of a reference -- **Using reference address is at your own risk**.

(2) Array of references is not allowed.

5. Functions with Pointers and References (Brief)

There are several terms relevant to functions as

Call-By-Value (CBV) (or Pass-By-Value -- PBV) and Call-By-Reference (CBR) (or Pass-By-Reference -- PBR)

Functions may be designed to receive either a value or a reference through calls. The prototypes for these cases were discussed previously. It is worth to mention here several facts about functions.

- A function may not return a value; in this case it is called a "`void` function".
- A function may have several **`return`** statements but ONLY one **`return`** statement will be executed for each function call.

- A function may produce side effects if its execution alters any system states or objects besides its own local objects.
- A function name can be reused several times for different purposes and implementations.

The first three facts are easy to understand. Function overloading is more interesting and requires a bit more work and implementation.

5.1 Functions – Passing-by-Reference

One of the important uses of pointers is in functions. Using pointers, one can send **data-information** to function for manipulation.

Note!

> The goal of the program is to manipulate (actual) data values, which are given or obtained; while memory addresses (of existing variables) are execution specific and dependent, which the programming logic and its (running/execution) system may assign different memory locations for different executions.

Let's look at ways to send data (and data information) to function. The function prototypes are given below.

```
ReturnType functionName1( Type1 *ptr1 );
ReturnType functionName2( Type1 *ptr1, Type4 *ptr4 );
ReturnType functionName3( Type1 *ptr1, Type2 old2 );
ReturnType functionName4( Type3 old3, Type4 *ptr4 );
```

Note that **`ptr1`** and **`ptr4`** are explicitly written as pointers. These functions have copies of the existing data sent to them through `old2` and `old3`; and the locations of the data (i.e., addresses) are sent through `ptr1` and `ptr4`..

In the above prototypes, note that

(1) **`ptr1`** and **`ptr4`** indicate that data are passed by references – PBR, and

(2) **`old2`** and **`old3`** indicate that data are passed by values (which are their copies) – PBV

The following example would illustrate the points.

Example

```
/**
 *Program Name: cis25L0324.cpp
 *Discussion:   Pointer Accessing
 */

#include <iostream>
using namespace std;

void printClassInfo( void );

int addTwoInt( int iA, int iB );
int addTwoIntPtr( int *iPtr1, int *iPtr2 );

int main ( void ) {
  int iVar1;
  int iVar2;
  int iVar3;
  iVar1 = 10;
  iVar2 = 20;

  iVar3 = addTwoInt( iVar1, iVar2 );
  cout << "\nThe sum is " << iVar3 << endl;

  iVar1 = 100;
  iVar2 = 200;

  iVar3 = addTwoIntPtr( &iVar1, &iVar2 );
  cout << "\nThe sum is " << iVar3 << endl;

  return 0;
}

/**
 *Function Name: addTwoInt()
 *Description:   To return the sum of two int's
 *Pre         :   Sending two int's
 *Post:       :   returning the sum of two int's
 */
int addTwoInt( int iA, int iB ) {
  int iResult;

  iResult = iA + iB;

  return iResult;
}

/**
 *Function Name: addTwoIntPtr()
 *Description:   To return the sum of two int's
 *Pre         :   Sending in the addresses of two int's
 *Post:       :   Returning the sum of two int's
 */
int addTwoIntPtr( int* iPtr1, int* iPtr2 ) {
  int iResult;

  iResult = *iPtr1 + *iPtr2;

  return iResult;
}

/**
 *Function Name: printClassInfo()
 *Description:   To print class information
 *Pre         :   Nothing (no argument required)
 *Post:       :   None
 */
void printClassInfo( void ) {
  cout << "\tCIS 25 -- C++ Programming"
       << "\n\n\tObject Orientation\n\n\t- Inheritance"
       << "\n\t- Polymorphism\n\t- Template" << endl;

  return;
}
```

OUTPUT

```
The sum is 30

The sum is 300
```

The results are as expected. Let’s look into the details of its execution flow.

Next, what about returning addresses from functions?

```
int* addTwoInt2( int, int );
int* addTwoIntPtr2( int *, int * );
```

5.2 Pointer – Return by Function

Recall that a function may return a value. This return value may also be an address as indicated with a pointer. This address is being passed from one environment (within the called function, i.e., the called environment) to another environment (back to the calling function, i.e., the calling environment).

Since data used in the called function might be out of scope (i.e., not available, may not exist), passing the address from any called function to another function must be done carefully.

Let’s consider the example below.

Example

```
/**
 *Program Name: cis25L0325.cpp
 *Discussion:   Returning pointers from functions
 */
#include <iostream>
using namespace std;

int addTwoInt( int iA, int iB );
int addTwoIntPtr( int* iPtr1, int* iPtr2 );

int* addTwoInt2( int, int );
int* addTwoIntPtr2( int*, int* );


int main ( void ) {
  int iVar1;
  int iVar2;

  int* iPtr1;
  int * iPtr2;

  iVar1 = 10;
  iVar2 = 20;

  iPtr1 = addTwoInt2( iVar1, iVar2 ); /*What is this?*/

  iVar1 = 100;
  iVar2 = 200;

  iPtr2 = addTwoIntPtr2( &iVar1, &iVar2 );
  cout << "\nThe sum is " << *iPtr2;

  return 0;
}
```

```
/**
 *Function Name: addTwoInt2()
 *Description:   To return the sum of two int's
 *Pre        :   Sending in two int's
 *Post:      :   Returning address of the storage where
 *               the sum of two int's is at
 */
int* addTwoInt2( int iA, int iB ) {
  int iResult;

  iResult = iA + iB;

  return &iResult;
}

/**
 *Function Name: addTwoIntPtr2()
 *Description:   To return the sum of two int's
 *Pre        :   Sending in the addresses of two int's
 *Post:      :   Returning address of the storage where
 *               the sum of two int's is at
 */
int* addTwoIntPtr2( int* iPtr1, int* iPtr2 ) {
  int* iPtrTemp;

  iPtrTemp = new int;

  *iPtrTemp = *iPtr1 + *iPtr2;

  return iPtrTemp;
}
```

OUTPUT

```
The sum is 300
```

How would the program work?

6. Passing References (of Variables/Objects) to Functions – Introduction

Recall that a reference behaves like a pointer but not a pointer. When a reference is passed to a function the following syntax is used,

```
ReturnType functionName( ArgType & aRef );
```

The **aRef** now refers to the original variable given in the calling function and this original variable can have its value changed inside the function being called. The sample calls are given in the following example.

Example

```
/**
 *Program Name:  cis25L0326.cpp
 *Discussion:    Passing References
 *                 - Pointer
 *                 - Reference
 */
#include <iostream>
using namespace std;
```

```cpp
void printValue( int& );
void printValue( int* );

int main( void ) {
  int iX = 5;

  cout << "\nValue of iX before calling printValue( ) : "
       << iX << endl;

  printValue( iX );

  cout << "\nValue of iX after calling printValue( iX ) : "
       << iX << endl;

  printValue( &iX );

  cout << "\nValue of iX after calling printValue( &iX ) : "
       << iX << endl;

  return 0;
}

/**
 *Function Name:  printValue()
 *Description:    Printing an integer
 *Pre:            The reference of the integer value
 *Post:           None
 */
void printValue( int& iRef ) {
  cout << "\nValue being passed : " << iRef << endl;
  cout << "\niRef will change the value in calling function!"
       << endl;

  iRef = iRef + 9;

  return;
}

/**
 *Function Name:  printValue()
 *Description:    Printing an integer
 *Pre:            The address of the integer value
 *Post:           None
 */
void printValue( int* iPtr ) {
  cout << "\nValue being passed : " << *iPtr << endl;
  cout << "\niPtr will change the value in calling function!"
       << endl;

  *iPtr = *iPtr + 18;

  return;
}
```

OUTPUT

```
Value of iX before calling printValue( ) : 5

Value being passed : 5
```

```
iRef will change the value in calling function!

Value of iX after calling printValue( iX ) : 14

Value being passed : 14

iPtr will change the value in calling function!

Value of iX after calling printValue( &iX ) : 32
```

The output values would speak for themselves regarding what values changed and which function would cause such change.