C/C++ Program Design

LAB 7

CONTENTS

- Master **passing by reference**
- Learn **inline** function and **default arguments**
- Learn how to define and use **Function Templates**
- Master the definition and use of **Function Overloading**

2 Knowledge Points

2.1 Reference in Function

2.2 Inline Function

2.3 Default Arguments

2.4 Function Overloading in C++

2.5 Function Templates

2.1 Passing by Reference and Return Reference

Example : C++ program to swap two numbers using pass by reference.

amplecode > swap.cpp > ...

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

void Swap(int &x, int &y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}

int main()
{
    int a = 45, b = 35;
    cout << "Before Swap" << endl;
    cout << "a = " << a << ",b = " << b << endl;

    Swap(a, b);

    cout << "After Swap(passing by reference)" << endl;
    cout << "a = " << a << ",b = " << b << endl;

    return 0;
}
```

Only by checking the function prototype or function definition can you tell whether the function passing by value or by reference.
In the called function's body, the reference parameter actually refers to ***the original variable*** in the calling function, and the original variable can be ***modified*** directly by the called function.

The style of the arguments are like common variables

output:

```
Before Swap
a = 45,b = 35
After Swap(passing by reference)
a = 35,b = 45
```

const References

It is more efficient to pass a large object by reference than to pass it by value. Using **const** to specify a reference parameter should **not be allowed** to modify the corresponding argument. **Use const when you can**.

Reference variables **must** be initialized in the declaration and **cannot** be reassigned as aliases to other variables.

The variable **edge** is of wrong type, the compiler generates a temporary, anonymous variable and makes **ra** refer to it.

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

double refcube(const double &ra);

int main()
{
    double side = 3.0;
    double * pd = &side;
    double & rd = side;
    long edge = 5L;
    double lens[4] = { 2.0, 5.0, 10.0, 12.0};

    double c1 = refcube(side);
    double c2 = refcube(lens[2]);
    double c3 = refcube(rd);
    double c4 = refcube(*pd);
    double c5 = refcube(edge);
    double c6 = refcube(7.0);
    double c7 = refcube(side + 10.0);
    cout<< c1<<"  "<< c2<<"  "<< c3<<"  "<<c4<<"  "<<c5<<"  "<< c6<<"  "<<c7<<endl;

    return 0;
}

double refcube(const double &ra)
{
    return ra * ra * ra;
}
```

```
27  1000  27  27  125  343  2197
```

Returning a Reference

```cpp
#include <iostream>
#include <string>
struct free_throws
{
    std::string name;
    int made;
    int attempts;
    float percent;
};

void display(const free_throws & ft);
void set_pc(free_throws & ft);
free_throws & accumulate(free_throws & target, const free_throws & source);

int main()
{
    // partial initializations - remaining members set to 0
    free_throws one = {"Ifelsa Branch", 13, 14};
    free_throws team = {"Throwgoods", 0, 0};


    free_throws dup;
    dup = accumulate(team,one);     // use return value in assignment

    std::cout << "Displaying team:\n";
    display(team);
    std::cout << "Displaying dup after assignment:\n";
    display(dup);

    return 0;
}
```

return a structure reference

pass by structure references

Output:

```
Displaying team:
Name: Throwgoods
 Made: 13       Attempts: 14    Percent: 92.8571
Displaying dup after assignment:
Name: Throwgoods
 Made: 13       Attempts: 14    Percent: 92.8571
```

```cpp
void display(const free_throws & ft)
{
    using std::cout;
    cout << "Name: " << ft.name << '\n';
    cout << " Made: " << ft.made << '\t';
    cout << "Attempts: " << ft.attempts << '\t';
    cout << "Percent: " << ft.percent << '\n';
}

void set_pc(free_throws & ft)
{
    if (ft.attempts != 0)
        ft.percent = 100.0f *float(ft.made)/float(ft.attempts);
    else
        ft.percent = 0;
}

free_throws & accumulate(free_throws & target, const free_throws & source)
{
    target.attempts += source.attempts;
    target.made += source.made;
    set_pc(target);

    return target;
}
```

Do not return a reference of a local variable

return a structure reference, more efficient

```cpp
#include <iostream>
#include <string>
struct free_throws
{
    std::string name;
    int made;
    int attempts;
    float percent;
};

void display(const free_throws & ft);
const free_throws & clone2(free_throws & ft);

int main()
{
    // partial initializations - remaining members set to 0
    free_throws one = {"Ifelsa Branch", 13, 14};

    free_throws dup2;
    dup2 = clone2(one);
    std::cout << "Displaying dup2 after calling clone2:\n";
    display(dup2);
    return 0;
}

void display(const free_throws & ft)
{
    using std::cout;
    cout << "Name: " << ft.name << '\n';
    cout << " Made: " << ft.made << '\t';
    cout << "Attempts: " << ft.attempts << '\t';
    cout << "Percent: " << ft.percent << '\n';
}

const free_throws & clone2(free_throws & ft)
{
    free_throws newguy;
    newguy = ft;
    return newguy;
}
```

const means you don't want to permit behavior such as assigning a value to clone2()

return a reference of a local variable

```
returnlocalref.cpp:39:12: warning: reference to local variable ‘newguy’ returned [-Wreturn-local-addr]
   39 |     return newguy;
      |            ^~~~~~
```

```
Segmentation fault (core dumped)
```

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

//Passing by value
void change1(int n)
{
    cout << "Pass by value---the operation address of the function is:" << &n << endl;
    n++;
}

//Passing by reference
void change2(int &n)
{
    cout << "Pass by reference---the operation of the function is:" << &n << endl;
    n++;
}

//Pass by pointer
void change3(int *n)
{
    cout << "Pass by pointer---the operation of the function is:" << n << endl;
    n++;
}

int main()
{
    int n = 10;
    cout << "The address of the argument is:" << &n << endl << endl;

    change1(n);
    cout << "After change1(), n = " << n << endl << endl;

    change2(n);
    cout << "After change1(), n = " << n << endl << endl;

    change3(&n);
    cout << "After change1(), n = " << n << endl << endl;

    return 0;
}
```

Pass by value

Pass by reference

Pass by pointer

Passing by value, the address that the function operates is not that of the argument; but **passing by reference(or pointer)**, the function operates the address of argument.

```
The address of the argument is:0x7fffffb89cf4

Pass by value---the operation address of the function is:0x7fffffb89cdc
After change1(), n = 10

Pass by reference---the operation of the function is:0x7fffffb89cf4
After change1(), n = 11

Pass by pointer---the operation of the function is:0x7fffffb89cf4
After change1(), n = 11
```

different

the same

the same

Difference between reference and pointer

- The reference must be initialized when it is created; the pointer can be assigned later.
- The reference can not be initialized by NULL; the pointer can.
- Once the reference is initialized, it can not be reassigned to other variable; a pointer can be changed to point to other object.
- **sizeof(reference)** operation returns the size of the variable; **sizeof(pointer)** operation returns the size of pointer itself.

2.2 Inline Function

C++ provides **inline functions** to help reduce function-call overhead(to avoid a function call).

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

inline double cube(double side);

int main()
{
    double sideValue;
    cout << "Enter the side of your cube:";
    cin >> sideValue;

    cout << "Volume of cube with side " << sideValue << " is " << cube(sideValue) << endl;

    return 0;
}

inline double cube(double side)
{
    return side * side * side;
}
```

Place the qualifier **inline** before return type in the function prototype

The qualifier **inline** can be omitted in the function definition if it is in the function prototype.

2.3 Default Arguments

```cpp
#include <iostream>
const int ArSize = 80;

char * left(const char *str, int n = 1);

int main()
{
    using namespace std;
    char sample[ArSize];
    cout << "Enter a string:";
    cin.get(sample,ArSize);

    char *ps = left(sample,4);
    cout << ps << endl;
    delete []ps;      //free string

    ps = left(sample);
    cout << ps << endl;
    delete []ps;    //free string

    return 0;
}
// This function returns a pointer to a new string
// consisting of the first n characters in the string.
char * left(const char *str, int n)
{
    if(n < 0)
        n = 0;

    char *p = new char[n+1];
    int i;
    for(i = 0; i < n && str[i]; i++)
        p[i] = str[i];    // copy characters

    while(i <=n)
        p[i++] = '\0';     // set rest of string to '\0'

    return p;
}
```

Default arguments must be specified in the function prototype and must be **rightmost(trailing)**.

```
Enter a string:C++ is funny.
C++
C
```

2.4 Function Overloading in C++

Function overloading is a feature in C++ where two or more function can have the same name but different parameters.
Function overloading is used to create several functions of the same name that perform similar tasks, but on different data types. The C++ compiler selects the the proper function to call by examining the number, types and order of the arguments.

1.the same fuction name
2.different parameter list

Example: Illustrate function overloading

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

//overloaded function p
void add(int i, int j);
void add(int i, double j);
void add(double i, int j);
void add(int i, int j, int k);

int main()
{
    int a = 1, b = 2, c = 3;
    double d = 1.1;

    // overloaded functions with difference type
    // and number of parameters
    add(a,b);     // 1 + 2 => add prints 3
    add(a,d);    // 1 + 1.1 => add prints 2.1
    add(d,a);
    add(a, b, c);  // 1+ 2 +3 => add prints 6

    return 0;
}
```

The same function name but different parameter list

```cpp
void add(int i, int j)
{
    cout << "Result: " << i + j << endl;
}
void add(int i, double j)
{
    cout << "Result: " << i + j << endl;
}
void add(double i, int j)
{
    cout << "Result: " << i + j << endl;
}
void add(int i, int j, int k)
{
    cout << "Result: " << i + j + k << endl;
}
```

Output:

```
Result: 3
Result: 2.1
Result: 2.1
Result: 6
```

2.5 Function Templates

Overloaded functions are normally used to perform similar operations that involve different program logic on different data types. If the program logic and operations are identical for each data type, overloading may be performed more compactly and conveniently by using function templates.

```
// This function returns the first ct digits of the number num
unsigned long left(unsigned long num, unsigned ct)
{
    unsigned digits = 1;
    unsigned long n = num;
    if(ct == 0 || num ==0)
        return 0;    //return 0 if no digits

    while(n /= 10)
        digits++;

    if(digits > ct)
    {
        ct = digits -ct;
        while(ct--)
            num /= 10;
        return num;   //return left ct digits
    }
    else    // if ct >= number of digits
        return num;   //return the whole number

}
```

These two functions are overloaded functions that involve different logic different data types.

```
// This function returns a pointer to a new string
// consisting of the first n characters in the string.
char * left(const char *str, int n)
{
    if(n < 0)
        n = 0;

    char *p = new char[n+1];
    int i;
    for(i = 0; i < n && str[i]; i++)
        p[i] = str[i];    // copy characters

    while(i <=n)
        p[i++] = '\0';     // set rest of string to '\0'

    return p;
}
```

Example for function template:

1. Write a function to calculate the maximum of two integers .

```
int Max(int x, int y)
{
    return (x > y? x : y);
}
```

These two functions are overloaded functions
Their program logic and operations are identical for each data type .

2. Write a function to calculate the maximum of two doubles.

```
double Max(double x, double y)
{
    return (x > y? x : y);
}
```

Note that the code for the implementation of the double version of max() is exactly the same as for the int version of max()! !

The syntax of templates:

```
template <typename T>   // This is the template parameter declaration
T Max(T x, T y)
{
    return (x > y? x : y);
}
```

- Starts with the keyword **template**
- You can also use keyword **class** instead of **typename**
- **T** is a template argument that accepts different data types

compile internally generates and adds right code respectively.

```cpp
template <typename T>
T Max(T x, T y)
{
    return (x > y? x : y);
}

int main()
{
    cout << "Max int = " << Max<int>(3,7) << endl;
    cout << "Max char = " << Max<char>('g','e') << endl;
    cout << "Max double = " << Max<double>(3.1,7.9) << endl;

    return 0;
}
```

```cpp
int Max(int x, int y)
{
    return (x > y? x : y);
}
```

```cpp
char Max(char x, char y)
{
    return (x > y? x : y);
}
```

```cpp
double Max(double x, double y)
{
    return (x > y? x : y);
}
```

output:

```
Max int = 7
Max char = g
Max double = 7.9
```

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

// function template prototype
template <typename T>    // or class T
void Swap(T &a, T &b);

int main()
{
    int i = 10, j = 20;
    cout << "Before swap: i = " << i << ",j = " << j << endl;
    cout << "Using compiler-generated int swap:\n";
    Swap(i,j);    // generates void swap(int &, int &)
    cout << "After swap: i = " << i << ",j = " << j << endl;

    double x = 34.5, y = 78.2;
    cout << "Before swap: x = " << x << ",y = " << y << endl;
    cout << "Using compiler-generated double swap:\n";
    Swap(x,y);    // generates void swap(double &, doulbe &)
    cout << "After swap: x = " << x << ",y = " << y << endl;

    return 0;

}

// function template definition
template <typename T>
void Swap(T &a, T &b)
{
    T temp;
    temp = a;
    a = b;
    b = temp;
}
```

The function template specialization create for type **int** replaces each current of **T with int** as follows

```cpp
void Swap(int &a, int &b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

The function template specialization create for type **double** replaces each current of **T with double** as follows

```cpp
void Swap(double &a, double &b)
{
    double temp;
    temp = a;
    a = b;
    b = temp;
}
```

Output：

```
Before swap: i = 10,j = 20
Using compiler-generated int swap:
After swap: i = 20,j = 10
Before swap: x = 34.5,y = 78.2
Using compiler-generated double swap:
After swap: x = 78.2,y = 34.5
```

Overloaded template functions

```
template <typename T>
void Swap(T &a, T &b)
{
    T temp;
    temp = a;
    a = b;
    b = temp;
}

template <typename T>
void Swap(T a[], T b[], int n)
{
    T temp;
    for (int i = 0; i < n; i++)
    {
        temp = a[i];
        a[i] = b[i];
        b[i] = temp;
    }
}
void Show(int a[])
{
    using namespace std;
    cout << a[0] << a[1] << "/";
    cout << a[2] << a[3] << "/";
    for (int i = 4; i < Lim; i++)
        cout << a[i];

    cout << endl;
}
```

Overloaded template functions

```
// using overloaded template functions
#include <iostream>
template <typename T> // original template
void Swap(T &a, T &b);

template <typename T> // new template
void Swap(T *a, T *b, int n);

void Show(int a[]);
const int Lim = 8;

int main()
{
    using namespace std;
    int i = 10, j = 20;
    cout << "i, j = " << i << ", " << j << ".\n";
    cout << "Using compiler-generated int swapper:\n";
    Swap(i,j); // matches original template
    cout << "Now i, j = " << i << ", " << j << ".\n";

    int d1[Lim] = {0,7,0,4,1,7,7,6};
    int d2[Lim] = {0,7,2,0,1,9,6,9};
    cout << "Original arrays:\n";
    Show(d1);
    Show(d2);

    Swap(d1,d2,Lim); // matches new template
    cout << "Swapped arrays:\n";
    Show(d1);
    Show(d2);

    return 0;
}
```

Function prototype

Output:

```
i, j = 10, 20.
Using compiler-generated int swapper:
Now i, j = 20, 10.
Original arrays:
07/04/1776
07/20/1969
Swapped arrays:
07/20/1969
07/04/1776
```