Advanced Programming

Lab 07

1 CONTENTS

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

2 Knowledge Points

2.1 Reference in Function

2.2 Inline Function

2.3 Default Arguments

2.4 Function Overloading

2.5 Function Templates

2.1 Reference in function

A **reference** defines an alternative name (or **alias**) for an object. A reference type “refer to“ another variable. Using “**&**” to declare a reference.

```
int ival = 1024;
int &refVal = ival; // refVal refers to (is another name for) ival
int &refVal2;       // error: a reference must be initialized
```

Once initialized, a reference remains bound to its initial object. There is no way to rebind a reference to refer to a different object.

After a reference has been defined, all operations on that reference are actually operations on the object to which the reference is bound.

```
refVal = 2;        // assign 2 to the object to which refVal refers, i.e., to ival
int ii = refVal;   // same as ii = ival
```

```
int &refVal2 = 10; // error: initializer must be an object
double dval = 3.14;
int &refVal3 = dval; // error: initializer must be an int object
```

Reference as function parameters –passing by reference

```
#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:" << 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

Result:

```
Before swap:
a = 45, b = 35
After swap:
a = 35, b = 45
```

6 differences between pointer and reference.

Use references when you can,
and pointers when you have to.

```
#include <iostream>
using namespace std;
struct demo
{
    int a;
};
int main()
{
    int x = 5;
    int y = 6;
    demo d;

    int *p;
    p = &x; // 1. Pointer reinitialization allowed
    p = &y;
    int &r = x;
    // &r = y; //2. Compile Error
    r = y; // 2. x value becomes 6

    p = NULL;
    // &r = NULL; //3. Compile error
    p++;                                 // 3. Points to next memory location
    x++;                                 // 3. x value becomes 7
    cout << &p << " " << &x << endl;  // 4. Different address
    cout << &r << " " << &x << endl; // 4. Same address

    demo *q = &d;
    demo &qq = d;
    q->a = 8;
    // q.a = 8;  //5. Compile Error
    qq.a = 8;
    // qq->a = 8; // 5. Compile Error

    cout << p << endl; // 6. Print the address
    cout << r << endl; // 6. Print the value of x
    return 0;
}
```

const reference: reference that refers to a const type. A reference to const cannot be used to change the object to which the reference is bound.

```
const int ci = 100;
const int &r1 = ci; // OK: both reference and underlying object are const
r1 = 42;            // error: r1 is a const reference
int &r2 = ci;       // error: non const reference to a const object
```

The type of a reference must match the type of the object to which it refers. But there are **two exceptions** to the rule. The one is that a const reference can refer to an non const object. The other is that we can initialize a const reference from any expression that can be converted to the type of the reference.(**Note**: The first exception also applies to **pointers to const**.)

```
int i = 42;  // OK:rl bound to i
int &r1 =i;
const int &r2 = 1;     //OK:r21s const reterence,bound to non const object
const int &r3 = 99;     //OK:r3 is const reference
const int &r4 = r1 *2;  //OK:r4 is const reference
//int &r5 =rl *3;    //error:r5 is non const reference
r1 = 0;              // OK:r1 is non const reference,it might be changed
r2 = 5;  //error:r2 is const reference,it can not be used to change i
```

const References

```cpp
#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;
}
```

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.

For the **const reference** parameters, the arguments can be literals or expressions.

Result:

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

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

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

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

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

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

    passByVal(n);
    cout << "After calling passByVal(), n = " << n << endl << endl;

    passByPoint(&n);
    cout << "After calling passByPoint(), n = " << n << endl << endl;

    passByRef(n);
    cout << "After calling passByRef(), n = "<< n << endl << endl;

    return 0;
}
```

Pass by value

Pass by pointer

Pass by reference

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.

Result:

```
The address of the argument is:0x7fff18d72044

Pass by value---the operation address of the function is: 0x7fff18d7202c
After calling passByVal(), n = 10

Pass by pointer---the operation address of the function is: 0x7fff18d72044
After calling passByPoint(), n = 11

Pass by reference---the operation address of the function is: 0x7fff18d72044
After calling passByRef(), n = 12
```

Return a Reference

```cpp
#include <iostream>
using namespace std;
struct point
{
    double x;
    double y;
};
point mid1(const point &, const point &);
point* mid2(const point &, const point &);
void mid3(const point &, const point &, point &);
point& mid4(const point &, const point &);
```

```cpp
int main()
{
    point p1{1, 1};
    point p2{3, 3};
    point pv, pr, prr;
    point *pp = NULL;

    pv = mid1(p1, p2);
    pp = mid2(p1, p2);
    mid3(p1, p2, pr);
    prr = mid4(p1, p2);

    cout << "Calling mid1, midpoint is:(" << pv.x << "," << pv.y << ")" << endl;
    cout << "Calling mid2, midpoint is:(" << pp->x << "," << pp->y << ")" << endl;
    cout << "Calling mid3, midpoint is:(" << pr.x << "," << pr.y << ")" << endl;
    cout << "Calling mid4, midpoint is:(" << prr.x << "," << prr.y << ")" << endl;

    delete pp;

    return 0;
}
```

```cpp
point& mid4(const point &p1, const point &p2)
{
    point p;
    p.x = (p1.x + p2.x) / 2;
    p.y = (p1.y + p2.y) / 2;

    return p;
}
```

Result:

```
cs@DESKTOP-L61ETB1:/mnt/h/CS219_2024F/code/week07$ g++ _10_pointStruct.cpp
_10_pointStruct.cpp: In function 'point& mid4(const point&, const point&)':
_10_pointStruct.cpp:69:12: warning: reference to local variable 'p' returned [-Wreturn-local-addr]
   69 |     return p;
      |            ^
_10_pointStruct.cpp:65:11: note: declared here
   65 |     point p;
      |           ^
^[[Acs@DESKTOP-L61ETB1:/mnt/h/CS219_2024F/code/week07$ ./a.out
Segmentation fault
```

Do not return a reference of a local variable.You can return a reference parameter.

```
point mid1(const point &p1, const point &p2)
{
    point pv;
    pv.x = (p1.x + p2.x) / 2;
    pv.y = (p1.y + p2.y) / 2;

    return pv;
}
```

return a local structure variable is ok, but less efficient

```
point* mid2(const point &p1, const point &p2)
{
    point* pp = new point;
    pp->x = (p1.x + p2.x) / 2;
    pp->y = (p1.y + p2.y) / 2;

    return pp;
}
```

return a local structure pointer which is allocated memory by new, is ok.

```
void mid3(const point &p1, const point &p2, point &pr)
{
    pr.x = (p1.x + p2.x) / 2;
    pr.y = (p1.y + p2.y) / 2;
}
```

Result:

```
Calling mid1, midpoint is:(2,2)
Calling mid2, midpoint is:(2,2)
Calling mid3, midpoint is:(2,2)
```

The function does not return anything.
The third parameter is a reference parameter, modifying the value of the parameter is exactly changing that of the argument.

Return a Reference

```
#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);

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

    return 0;
}
```

pass by structure references

const means the value of the reference can not be modified

return a structure reference

```
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;
}
```

return a structure reference, more efficient

Do not return a reference of a local variable
Return the reference parameter

Return a Reference

Whether a function call is an **lvalue** depends on the return type of the function. Calls to functions that return references are **lvalues**; other return types yield rvalues. We can assign to the result of a function that returns a reference to **non-const**.

The return value is a reference, so the call is an lvalue. Like any other lvalue, it may appear as the left-hand operand of the assignment operator.

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

char &get_val(string &str, string::size_type ix)
{
    return str[ix]; // get_val assumes the given index is valid
}


int main()
{
    string s("a value");
    cout << s << endl; // prints a value

    get_val(s, 0) = 'A'; // changes s[0] to A

    cout << s << endl; // prints A value

    return 0;
}
```

Result:

```
a value
A value
```

```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 & clone(free_throws &ft);
```

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

Such as:

clone(dup2) = three;   // not allowed for const
                       // reference return

```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';
}
const free_throws & clone(free_throws &ft)
{
    ft.percent = 5;
    return ft;
}
```

return a reference parameter

```cpp
int main()
{
    // partial initializations - remaining members set to 0
    free_throws one = {"Ifelsa Branch", 13, 14};
    free_throws two = {"Andor Knott", 10, 16};
    free_throws three = {"Minnie Max", 7, 9};

    std::cout << "The original one is: " << std::endl;
    display(one);

    free_throws dup1 = clone(one);

    std::cout << "The dup1 is: " << std::endl;
    display(dup1);
    std::cout << "After calling clone(), the one is: " << std::endl;
    display(one);

    free_throws dup2 = clone(two);

    std::cout << "The dup2 is: " << std::endl;
    display(dup2);
    std::cout << "After calling clone(), the two is: " << std::endl;
    display(two);

    return 0;
}
```

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;
}
```

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

```cpp
// 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;
}
```

```
Enter a string: hello world
hell
h
```

2.4 Function Overloading

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: function overloading

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

// overloaded function prototypes
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 different
types 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);     // 1.1 + 1 => add prints
2.1
    add(a, b, c);  // 1 + 2 + 3 => add prints
6

    return 0;
}
```

```
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;
}
```

Result:

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

We can overload based on whether the parameter is a reference (or pointer) to the **const** or **non-const** version of a given type.

```
Record lookup(Account & );           // function that takes a reference to Account
Record lookup(const Account & );     // new function that takes a const reference

Record lookup(Account * );           // new function, takes a pointer to Account
Record lookup(const Account * );     // new function, takes a pointer to const
```

In these cases, the compiler can use the constness of the argument to distinguish which function to call.

2.5 Function Templates

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

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

When we call a function template, the compiler (ordinarily) uses the arguments of the call to deduce the template argument(s) for us. These compiler-generated functions are generally referred to as an **instantiation of the template**.

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

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;
}
```

implicit instantiation
<int>,<char>,<double>
can be omitted.

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

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

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

Result:

```
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);

template void Swap<int>(int &, int &);
template void Swap<double>(double &, double &);

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 &, double &)
    cout << "After swap: x = " << x << ", y = " << y << endl;

    return 0;
}

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

explicit instantiation

The function template instantiation creates 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 instantiation creates 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;
}
```

Result:

```
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

```
#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);
    // cin.get();

    return 0;
}
```

Function prototype

```
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;
}
```

Result:

```
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
```

Function template specialization

If a function template for the general definition is not appropriate for a particular type, you can define a **specialized version** of the function template.

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

template <typename T>
T sum(T x, T y)
{
    return x + y;
}

struct Point
{
    int x;
    int y;
};

// Specialization for sum Point
template <>
Point sum<Point>(Point pt1, Point pt2)
{
    Point pt;
    pt.x = pt1.x + pt2.x;
    pt.y = pt1.y + pt2.y;
    return pt;
}
```

```cpp
int main()
{
    // Implicit instantiated functions
    cout << "sum = " << sum(1, 2) << endl;
    cout << "sum = " << sum(1.1, 2.2) << endl;

    Point pt1 {1, 2};
    Point pt2 {2, 3};

    // template specialization
    Point pt = sum(pt1, pt2);

    cout << "pt = (" << pt.x << ", " << pt.y << ")" << endl;

    return 0;
}
```

Result:

```
sum = 3
sum = 3.3
pt = (3, 5)
```

3 Exercises

1. The following is a program skeleton:

```
#include <iostream>
#include <cstring>    //for strlen(), strcpy()

struct stringy
{
    char * str;   // points to a string
    int ct;       // length of string(not counting '\0')
};

// prototypes for set() and two overloading functions show() with default arguments

int main()
{
    stringy beany;
    char testing[] = "Reality isn't what it used to be.";

    set(beany,testing);  // first argument is a reference,
         // allocates space to hold copy of testing,
         // sets str member of beany to point to the
         // new block, copies testing to the new block,
         // and sets ct member of beany

    show(beany);    //prints member string once
    show(beany, 2);  //prints member string twice

    testing[0] = 'D';
    testing[1] = 'u';
    show(testing);     //prints testing string once
    show(testing, 3);   //prints test string thrice
    show("Done!");    // prints "Done" on the screen

    // free the memory
    return 0;
}


// defines the three functions
```

Complete this skeleton by providing the described functions and prototypes. Note that there should be two **show()** functions, each using default arguments. Use **const** arguments when appropriate. Note that **set()** should use **new** to allocate sufficient space to hold the designated string.

A sample runs might look like this:

```
Reality isn't what it used to be.
Reality isn't what it used to be.
Reality isn't what it used to be.
Duality isn't what it used to be.
Duality isn't what it used to be.
Duality isn't what it used to be.
Duality isn't what it used to be.
Done!
```

2. Write a template function **maxn()** that takes as its arguments an array of items of type T and an integer representing the number of elements in the array and that returns the largest item in the array. Test it in a program that uses the function template with an array of five int values({1,2,3,4,5}) and an array of four double values({1.1,2.7,-3.5,-2}). The program should also include a **specialization** that takes **an array of pointers-to-char** as an argument and **the number of pointers** as a second argument and that returns the address of the longest string. If multiple strings are tied for having the longest length, the function should return the address of the first one tied for longest. Test the specialization with an array of the five string pointers({“this”,”no body”,”morning”,”birds”,”sky”}).

A sample runs might look like this:

```
Max int is: 5
Max double is: 2.7
Longest string is: no body
```