Lecture15

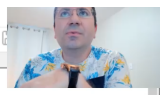
1. marco → pre-processor (#define)
2. Function Template (specified & non specified, typecasting)
3. Function overloading (How to match)
4. Stack (reusable code, Data structure)
5. operator overloading (non member function & member function)

marco.

pre-processor (before compiler)

- We can define some function in pre-processor

- Apply name in main(). it will substitute with it actual value



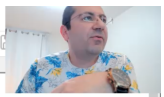
```
1
2  #define PI 3.1416
3  #define MIN1(a, b) (a < b) ? a : b
4  #define MIN2(a, b) (((a) < (b)) ? (a) : (b))
5  #define Rounding(x) (static_cast<int>((x) + 0.5))
6  #include <iostream>
7
8  using namespace std;
9
10 int main()
11 {
12     int a=3, b = 4;
13     int z;
14
15     int c = (a > b) ? 8 : 7; //If true 8, otherwise 7
16
17     z = MIN1(a*2, b +5);
18     cout << "z = " << z << endl;
20     int x = 2, y = 3;
21     z = MIN1(x, y) + 7; // NO!
22     cout << "z = " << z << endl;
23
24     z = MIN2(x, y) + 7; // Yes!
25     cout << "z = " << z << endl;
26
27     double v = 5.4;
28     cout << "v = " << Rounding(v) << endl;
29
30     double w = 5.6;
31     cout << "w = " << Rounding(w) << endl;
32
```

name, actual value.

Different? if there have "()", then it can use to do calculation

if. else.

② Function Template



```cpp
#include <iostream>
#include <string.h> // C library

template<typename T>

T* find(T* start, T* finish, const T& v) {
    while (start != finish && *start != v)
        ++start;
    return start;
}

template<typename T>
T min(T g, T d) {
    return ((g<d)?g:d);
}

// there must be a template for which this
// template is a special case
template<>
char* min<char*>(char*g, char*d) {
    if (strcmp(g, d) < 0) return g;
     else return d;
}

int main()
{
    int v[10] = {1, 4, 7, 9, 14, 36, 42, 51, 56, 65};

    int *p = find (v, v+5, 7);
    if (p != v+5)
        std::cout << "It was found." << std::endl;
    else
        std::cout << "We didn't find it." << std::endl;

    int *q = find (v, v+7, 56);
    if (q != v+7)
        std::cout << "It was found." << std::endl;
    else
        std::cout << "We didn't find it." << std::endl;

    int x = 5, y = 7;
    double a = 4.2, b = 6.8;
    char c[] = "bonjour", d[] = "salut";

    int z = min(x, y);
    double w = min(a, b);

     // does not work without the static_cast;
    // all T types must match the same type.
    int u = min(static_cast<int>(a), y);

    // calls the second function template min;
    // a call to the first function would return
    // the char* array with the smallest address
    char* s = min(c, d);

    std::cout << "z = " << z << std::endl;
    std::cout << "w = " << w << std::endl;
    std::cout << "u = " << u << std::endl;
    std::cout << "s = " << s << std::endl;


    /*----------------- results ----------------------*\
    It was found.
    We didn't find it.
    z = 5
    w = 4.2
    u = 4
    s = bonjour
    \*------------------------------------------------*/

}
```

automatic deduce type
↳ find <int>

1. Implicit : promotion
   allow convert between numerical type
2. Explicit "( )"
   (new_type) expression
   new_type (expression)
   C_style Casting
   static_cast : safe conversion
   ↳ provided type check

What's the difference between implicit & explicit ?

implicit : needed for compiler

explicit : request by programmer

(3) Function Overloading

```
#include <iostream>

class Fraction {

  long num;
  long den;

  public:

    Fraction(long d=0, long n=1) : den(d), num(n) {} // note: conversion
    long->Fraction

    double getValue() { return static_cast<double>(num)/den; }
};
```

Conversion Constructor

If we don't provide any initialization value to constructor, 0,1 will automatically assign to the constructor

```
#include <iostream>
#include "3-fraction.h"

using namespace std;

/*-------------------------------------------------------------*\
Function calling rules (determined at compile time)
    1. Exact match
    2. Trivial match
        array name -> pointer
        int -> const int
    3. A template function
    4. Match with promotion
        char -> int
        short -> int
        float -> double
    5. Match with standard conversion
        int -> float
        float -> int
        Derived* -> Base*
        T* -> void
    6. Match with conversion defined by the use of
        one argument constructor
\*-------------------------------------------------------------*/

void fct(char) {
    cout << "fct(char) called" << endl;
}

void fct(double) {
    cout << "fct(double) called" << endl;
}

void fct(int, float) {
    cout << "fct(int, float) called" << endl;
}

void fct(int, double) {
    cout << "fct(int, double) called" << endl;
}

void add(Fraction &f, long l) {
    cout << "add(Fraction &f, long l) called" << endl;
}

void add(long l, Fraction &f) {
    cout << "add(long l, Fraction &f) called" << endl;
}
```

How to determine which function to call

Same function name but different arguments.

```
52
53  void add(long l, Fraction &f) {
54      cout << "add(long l, Fraction &f) called" << endl;
55  }
56
57  void add(Fraction &f1, Fraction &f2) {
58      cout << "add(Fraction &f1, Fraction &f2) called" << endl;
59  }
60
61  int main()
62  {
63      // fct(1);
64      // -> compilation error:
65      // ambiguity between fct(double) and fct(Char)
66
67      fct(9.9); // exact match
68
69      double d = 2.0;
70      char x = 2;
71
72      fct(x, d);
73      // arg1 -> fct(int, float) or fct(int, double)
74      // arg2 -> fct(int, double)
75      // fct(int, double) is the intersection between the two
76
77      //fct(x, x);
78      // arg1 -> fct(int, float) or fct(int, double)
79      // arg2 -> fct(int, float) or fct(int, double)
80      // -> compilation error: ambiguity
81
82      // Note: For the next conversions, a
83      // conversion is defined from long to fraction
84
85      Fraction f(2, 3);
86      add(f, 23);
87      // arg1 -> add(Fraction, long) or add(Fraction, Fraction)
88      // arg2 -> add(Fraction, long)
89      // add(Fraction, long) is the intersection between the two
89      // arg2 -> add(Fraction, long)
90      // add(Fraction, long) is the intersection between the two
91
92      // add(21, 23);
93      // arg1 -> add(long, Fraction)
94      // arg2 -> add(Fraction, long)
95      // -> compilation error: no match found
96
```

④ Stack

```cpp
#include "4-stack.h"
#include <iostream>
#include <stdlib.h>
using namespace std;

int main() {

    Stack<int> stack(5);

    cout << "The stack is empty: " << stack.isEmpty() << endl;

    stack.push(1);
    cout << "Push: 1" << endl;

    stack.push(2);
    cout << "Push: 2" << endl;

    stack.push(3);
    cout << "Push: 3" << endl;

    stack.push(4);
    cout << "Push: 4" << endl;

    stack.push(5);
    cout << "Push: 5" << endl;

    cout << "The stack is empty: " << stack.isEmpty() << endl;
    cout << "The stack is full: " << stack.isFull() << endl;

    cout << "Pull: " << stack.pull() << endl;
    cout << "Pull: " << stack.pull() << endl;
    cout << "Pull: " << stack.pull() << endl;

    cout << "The stack is empty: " << stack.isEmpty() << endl;
    cout << "The stack is full: " << stack.isFull() << endl;
    cout << "The top object in the stack is: " << stack.top()
         << endl;

    cout << "Pull: " << stack.pull() << endl;
    cout << "Pull: " << stack.pull() << endl;

    cout << "The stack is empty: " << stack.isEmpty() << endl;
    cout << "The stack is full: " << stack.isFull() << endl;

    Stack<float> p2(9);

    return 0;
}

/*
The stack is empty: 1
Push: 1
```

C:\Users\aeama\Desktop\C++\CSI 2372_August\Lectures\Lec 8 Templates\3-functionOverloading.cpp

C++ source file length : 1,520 lines : 71 Ln : 9 Col : 14 Sel : 3 | 1 Windows (CR

2021-11-08 16:54:04

```cpp
#ifndef STACK_H
#define STACK_H

template<typename T>
class Stack
{
public:
   Stack(int s): size(s),current(-1),stackPointer(new T[size]){}

   ~Stack()
   {
     delete [] stackPointer;
   }

    bool push(const T &valueToPush)
    {
       if (!isFull())
       {
           stackPointer[++current] = valueToPush;
           return true;
        }

       return false;
    }

    T pull()
    {
       return stackPointer[current--];
    }

    T top(){
        if (isEmpty()) return T(); // stack is empty!
        return stackPointer[current];
    }

    bool isEmpty() const
    {
       return current == -1;
    }

    bool isFull() const
    {
       return current == size - 1;
    }

    private:
       int size;
       int current;
       T *stackPointer;
};
```

using heap to store

⑤ operator overloading

+ , − , ++ , −− , != , << , >> , ... []

primitive data type.

When if you do with object datatype?

obj1 + obj2 } ?
cout << obj1 }

Solution = operator (=) { ..... } programmer defined.

assignment

Example ①

```
#include "1_Clock.h"

const Clock operator+(const Clock &left, const Clock &right)
{
    // return an anonymous reference
    return Clock(left) += right;
    // declaration of anonymous variable with Clock (left)
}

const Clock operator-(const Clock &left, const Clock &right)
{

    return Clock(left) -= right;
}
```

object

keyword

↑ Not class function.
non member function

```
#ifndef CLOCK3_H
#define CLOCK3_H

#include <iostream>
#include <string>
#include <sstream>
#include <stdio.h>
using namespace std;

class Clock {

    private:

        long seconds;

    public:

        Clock(long sec=0) : seconds(sec) { }
        Clock(int d, int hrs, int min, long sec) {
                seconds= sec + min*60 + hrs*3600 + d*86400; }
        Clock(const Clock &t) : seconds(t.seconds) { }

        long sec() const {return seconds%60;}
        int min() const {return (seconds/60)%60;}
        int hr()  const {return (seconds/3600)%24;}
        int day() const {return seconds/86400;}



        // a += b is equivalent to a.operator+=(b);
        // operator : keyword when redefining operators
        // binary : 2 operands *this and right
        Clock& operator+=(const Clock &right) {
                seconds+= right.seconds;
                // return the result by reference
                return *this; }

        Clock& operator-=(const Clock &right) {
                seconds-= right.seconds; return *this; }

        // a =~ b is equivalent to a= b.operator~();
        // unary: 1 operands *this
        // const so does not change the state of the object
        const Clock operator~() const {
                Clock t(*this);
                if (t.sec()<30) t.seconds-= t.sec();
                else t.seconds+= 60-t.sec();
                return t; }


        // pre-increment
        // int differentiates post and pre-increment,
        Clock& operator++() { *this+= 1; return *this; }

        // post-increment
        // const to avoid calling a non-const method/fct
        const Clock operator++(int) { // int => dummy operand
                Clock tk(*this);
                ++(*this);
                // return the result by value
                return tk;
                // option alternative: return Clock(seconds++);
            }

        const string toString() const {
            stringstream ss (stringstream::in | stringstream::out
            ss << day() << "day" << (day()>1?"s ":" ");
            ss << hr() << "hr" << (hr()>1?"s ":" ");
            ss << min() << "min" << (min()>1?"s ":" ");
            ss << sec() << "sec";
            return ss.str();}
};

// cout is an output stream
// so we use an ostream
    inline ostream&
operator<<(ostream &left, const Clock &right) { // function

        left << right.toString();
        return left;
        // important to return the ostream to chain couts
    }

// implicit conversion from int to Clock
const Clock operator+
                (const Clock &left, const Clock &right);
const Clock operator-
                (const Clock &left, const Clock &right);


#endif // CLOCK3_H
```

member function

class member function

obj1.second;

obj2

obj1+=obj2

return object

return copy

What is member function?

→ You can use "this" keyword to implies the object.

When to return copy of object. when to return object it self.?

→ When we are changing the object. usually return itself reference