C++对C函数的扩展

1. 内联函数（关键字：inline）

> - **定义：C++编译器可以将一个函数进行内联编译，被C++编译器内联编译的函数叫做 内联函数**
> - **作用： C++中推荐使用内联函数 替代宏代码片段： #define FUN(a, b) ((a) > (b) ? (a) : (b))**
> - **优点：省去了函数被调用时压栈、跳转、返回 的开销**

1.1 基本形式

```
// inline void myFun(int a, int b); // XXX!!! 内联函数必须跟函数体写到一块：不能做声明
inline void function(int a, int b){ // 内联函数
    return a>b?a:b;
}

int main(int argc, const char *argv[]){
    // 调用该内联函数时，编译器 可能会 将该函数替换为以下内容
    // 优点：直接插入代码片段 避免压栈 出栈 返回的 额外开销
    function(1, 2); // 直接替换代码片段：  { a>b?a:b; }
    return 0;
}
```

> - **宏代码片段 由预处理器处理， 进行简单的文本替换，没有任何编译过程**
> - **C++编译器直接将 内联函数的函数体 替换在 内联函数 被调用的位置**

1.2 限制条件

> - **不能对函数进行取址操作 （因为是直接替换代码块）**
> - **函数内联声明必须在调用语句之前， 如： inline void function(int a, int b)**
> - **不能存在任何形式的循环语句 和 过多的条件判断语句，且函数体不能过大：当函数体的执行开销远大于压栈、跳转、返回所用的开销时，那么内联将无意义**

1.3 内联总结

> 1. **内联函数在编译时直接将函数体 插入函数被调用的位置**
> 2. **关键字inline 只是一种请求，因此编译器 也可能拒绝 这种请求**
> 3. **现代C++编译器能够进行编译优化，因此一些函数即使没有关键字inline声明，也可能被编译器内联编译**
> 4. **C++编译器提供了扩展语法，能够对函数进行强制内联，如：g++中的 __attribute__((always_inline)) 属性**

1.4 内联函数 与 宏定义 区别举例

```
#include "iostream"
using namespace std;
#define MYFUNC(a, b) ((a) < (b) ? (a) : (b))

inline int myfunc(int a, int b) {
    return a < b ? a : b;
}

int main(int argc, const char *argv[]){
    /* 内联函数调用 */
    int a = 1, b = 3;
    int c = myfunc(++a, b);
    printf("a = %d\n b = %d\n", a);     // a = 2  b = 3  c = 2

    /* 宏定义调用 */
    a = 1; b = 3;
    c = MYFUNC(++a, b);                 // 陷阱!!! 展开宏定义 ((++a) < (b) ? (++a) : (b))
    printf("a = %d\n b = %d\n", a);     // a = 3  b = 3  c = 3
    return 0;
}
```

2. 默认参数

> **C++中可以在函数声明时为参数提供一个默认值，当函数调用时没有指定这个参数的值，编译器会自动用默认值代替**

2.1 举例

```
void printInfo(int a = 3){
    printf("a = %d \n", a);
}

int main(int argc, const char *argv[]){
    printInfo(4); // 打印 4
    printInfo();  // c++编译通过，打印 默认值 3
}
```

2.2 规则

> - **只有参数列表后面部分的参数才可以提供默认参数值**
> - **一旦在函数调用中开始使用默认参数值，那这个参数后的所有参数都 必须使用默认值**

```
void printInfo1(int a = 3, int b = 2, int c){}  // c++编译器报错，不同通过
void printInfo2(int c, int a = 3, int b = 2){}  // 正常
```

3. 函数占位参数

> - **占位参数只有参数类型声明，而没有参数名声明**
> - **必须写全参数**

```
int function(int a, int b, int) {
    return a + b;
}

int main(int argc, const char *argv[]){
    // function(1, 2);                                  // 编译不通过，提示需要传入三个参数
    printf("func(1, 2, 3) = %d\n", function(1, 2, 3));  // 可以正常打印
    return 0;
}
```

4. 函数的默认参数 和 占位参数

> - **可以将占位参数与默认参数结合起来使用**
> - **意义：为以后程序的扩展留下线索，兼容C语言程序中可能出现的不规范写法**

```
int function(int a, int b, int = 0) {
    cout << "a = " << a << "b = " << b << endl;
}

int main(int argc, const char *argv[]){
    function(1, 2);     // 编译通过，没问题
    function(1, 2, 3);  // 编译通过，没问题
    return 0;
}
```

5 函数重载(Function Overload)

5.1 概念

> - **定义：用同一个函数名定义不同的函数，当函数名和不同的参数搭配时函数的含义不同**
> - **本质： 重载函数本质上是相互独立的不同函数**

> - **函数重载的判断标准：**
>
> ☑ **函数重载至少满足下面的一个条件：（1）参数个数不同 （2）参数类型不同 （3）参数顺序不同**
>
> ☑ **函数返回值不是函数重载的判断标准**

```
void function(int a){
    printf("a:%d \n", a);
}

void function(char *p){
    printf("%s \n", p);
}

void function(int a, int b){
    printf("a:%d ", a);
    printf("b:%d \n", b);
}

// 调用重载函数时候， C++编译器会按照输入匹配格式自动查找并区分是哪个函数
```

5.2 编译器调用准则

> 1. 将所有同名函数作为候选者
> 2. 尝试寻找可行的候选函数
> 3. 精确匹配实参
> 4. 通过默认参数能够匹配实参
> 5. 通过默认类型转换匹配实参
> 6. 匹配失败
> 7. 最终寻找到的可行候选函数不唯一，则出现二义性，编译失败
> 8. 无法匹配所有候选者，函数未定义，编译失败

5.3 与函数指针结合 举例

```
int fufunctionc(int a, int b, int c = 0){
    printf("a = %d, b = %d \n", a, b);
    printf("fun c\n");
    return 0;
}

int  (int a, int b){
    printf("a = %d, b = %d \n", a, b);
    return 0;
}

typedef int(*PFUNC)(int a, int b);

int main(int argc, const char *argv[]){
    PFUNC p = function;
    p(1,2); // 打印 下边那个 function()
    return 0;
}

// 根据重载规则挑选与函数指针参数列表一致的候选者 严格匹配候选者的函数类型与函数指针的函数类型
```
