实验 2 - C++ 程序设计之函数篇

> 智能 212 史胤隆 2006010529
> 指导老师：杨伟杰

一、实验目的

1. 熟悉 Visual C++ 6.0 的开发环境与特点。
2. 熟悉 Visual C++ 6.0 开发环境下的源程序编辑、调试等功能。
3. 通过程序设计学习 C++ 中函数的声明、定义、调用、重载
4. 学习 c++ 多文件结构

二、实验内容及结果

练习 1

写一个程序将 24 小时制的时间转换为 12 小时制的时间

> 本例应用标准开发风格; 详见源码

本练习考察的主要是函数的引用传参。使用引用传参，我们可以直接修改传入的参数，而不需要返回值。这样，我们可以直接在原参数上修改，而不需要额外的内存开销；同时，引用传参使得信息流可以从参数流出函数，实现信息的双向传递。

转换逻辑

首先我们需要明白时间格式转换不是一个简单的取余过程：

- 1: 至 11: 之间不需要转换
- 13: 至 23: 之间需要减去 12
- 0: 需要显示为 12: AM
- 12: 需要显示为 12: PM

因此，我们需要：

- 先根据时间判断是 AM 还是 PM

  ```
  ampm = hours < 12 ? 'A' : 'P';
  ```

- 24 制时间对 12 取余

  ```
  ... (hours %= 12) ...
  ```

- **如果为 0，显示为 12**

  ```
  hours = (hours %= 12) == 0 ? 12 : hours;
  ```

运行结果

```
请输入 24 小时制时间 (H:m): 03:27
12 小时制时间 (h:m tt): 3:27 A.M.
```

```
请输入 24 小时制时间 (H:m): 13:14
12 小时制时间 (h:m tt): 1:14 P.M.
```

```
请输入 24 小时制时间 (H:m): 00:01
12 小时制时间 (h:m tt): 12:1 A.M.
```

```
请输入 24 小时制时间 (H:m): 12:25
12 小时制时间 (h:m tt): 12:25 P.M.
```

练习 2

编一个程序，用同一个函数名对圆、矩形、梯形求面积

本练习主要考察函数的重载。函数重载是 C++ 的一个重要特性，它允许我们定义多个同名函数，只要它们的参数列表不同即可。这样，我们可以使用同一个函数名对不同的数据类型进行操作，提高代码的复用性。

```
你希望计算 (1.圆 / 2.矩形 / 3.梯形): 1
请输入
    半径: 2
圆的面积为: 12.56
```

```
你希望计算 (1.圆 / 2.矩形 / 3.梯形): 2
请输入
    长: 114
    宽: 514
矩形的面积为: 58596
```

```
你希望计算 (1.圆 / 2.矩形 / 3.梯形): 3
请输入
    上底: 1.5
    下底: 2.5
    高: 2.7
梯形的面积为: 5.4
```

练习 3

设计多文件工程用于求圆面积和矩形面积

本练习主要考察多文件结构。它可以将一个大型程序分解为多个小文件，提高代码的可读性和可维护性。

```
你希望计算 (1.圆 / 2.矩形): 1
请输入
    半径: 5
圆的面积为: 78.5
```

```
你希望计算 (1.圆 / 2.矩形): 2
请输入
    长: 1.14
    宽: 5.14
矩形的面积为: 5.8596
```

三、实验总结

1. 友好性、鲁棒性与输入检查

我们永远不要期待用户完全按照我们构想的方式输入。因此，无论是实际开发还是课程学习中，应该时刻注意：

- 输入提示
- 输入检查
- 错误处理

事实上，这些工作只需要我们多一句 `if`，多一个 `default`。

```
你希望计算 (1.圆 / 2.矩形 / 3.梯形): 0
输入错误!
```

```
你希望计算 (1.圆 / 2.矩形): 3
输入错误!
```

知识点实践：条件编译

在练习 1 的代码实践了本节课提到的条件编译，可以通过对宏定义的注释与否来控制是否进行输入检查，并避免了额外的运行时开销。

```
...

#define CHECK

...

void input(int& hours24, int& minutes)
{
    //  [out]   int&    hours24
    //  [out]   int&    minutes
    std::cout << "请输入 24 小时制时间 (H:m): ";
    char _;
    std::cin >> hours24 >> _ >> minutes;
#ifdef CHECK
    if (hours24<0 || hours24>23 || minutes<0 || minutes>59 || _!=':') {
        std::cerr << "输入错误!" << std::endl;
        exit(1);
    }
#endif
}

...
```

```
请输入 24 小时制时间 (H:m): 114:514
输入错误!
```

```
请输入 24 小时制时间 (H:m): 11-40
输入错误!
```

2. 引用与指针

在本课中，我们使用了与之前学习的指针传参不同的引用传参。因此，我们有必要思考引用与指针的异同。为梳理思路，我们进行实验。

我们设计如下实验程序：

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

void fpointer(int* p)
{
    *p += 1;
}

void freference(int& r)
{
    r += 1;
}

int main()
{
    int value = 10;
    cout << "value = " << value << endl;
    fpointer(&value);
    cout << "value = " << value << endl;
    freference(value);
    cout << "value = " << value << endl;
    return 0;
}
```

```
value = 10
value = 11
value = 12
```

获取这组代码的汇编：

> 汇编代码由 Compiler Explorer 生成

- ***fpointer(int*):***

```
push    rbp
mov     rbp, rsp
mov     QWORD PTR [rbp-8], rdi
mov     rax, QWORD PTR [rbp-8]
mov     eax, DWORD PTR [rax]
lea     edx, [rax+1]
mov     rax, QWORD PTR [rbp-8]
mov     DWORD PTR [rax], edx
nop
pop     rbp
ret
```

- ***freference(int&):***

```
push    rbp
mov     rbp, rsp
mov     QWORD PTR [rbp-8], rdi
mov     rax, QWORD PTR [rbp-8]
mov     eax, DWORD PTR [rax]
lea     edx, [rax+1]
mov     rax, QWORD PTR [rbp-8]
mov     DWORD PTR [rax], edx
nop
pop     rbp
ret
```

- ***main:***

```
push    rbp
mov     rbp, rsp
sub     rsp, 16
mov     DWORD PTR [rbp-4], 10
...
lea     rax, [rbp-4]
mov     rdi, rax
call    fpointer(int*)
...
lea     rax, [rbp-4]
mov     rdi, rax
call    freference(int&)
...
mov     eax, 0
leave
ret
```

我们不难发现，无论是传参方式，还是函数内部操作，指针传参与引用传参的汇编代码是完全一致的。这是因为引用传参本质上是指针传参的语法糖，编译器会在编译时将引用传参转换为指针传参。

从语法层面讲，两种都是地址的概念的实现，但指针保存的是所指对象的地址 (实体)，引用仅仅是对象的别名 (非实体)，指针需要通过解指针间接访问，而引用是直接访问。另外：

- 引用不能为空，指针可以为空；
- 引用必须在定义时就初始化并且不能改变所指的对象，而指针可以改变地址，从而改变所指的对象。

当然，引用传参的存在，大大简化了代码的书写，提高了代码的可读性，减少了指针传参的错误。因此，我们在 C++ 中更推荐使用引用传参。

3. 有关多文件结构的若干问题

a) 多文件结构的本质

多文件编译的本质是将多个文件编译为多个目标文件，再链接为一个可执行文件，而声明-定义原则是 C / C++ 实现多文件结构的基本前提。

C / C++ 仅要求函数需要在调用前声明，而不强求在调用前定义。因此，我们只要事先确保函数在定义和调用前已被定义过，而不必关注不同文件的编译顺序。我们只需要将一个可执行程序的若干源码文件分为两组，一组是声明，一组是定义和调用 (一定程度上，调用的本质也是定义——我们可以嵌套地追溯到 `main` 函数)，这便形成了我们常用的头文件 `.h` 和源文件 `.cpp` 的组合。

b) 不同编译环境对多文件结构的处理差异

在不同的编译环境下，多文件结构的处理方式可能有所不同。在 Microsoft Visual Studio 体系中，标准的多文件编译方式是将需要参与编译的文件全部加入到项目中；而在 GCC 等编译器中，我们需要在编译的时候统一指定诸如 `g++ main.cpp func.cpp -o main` 的命令；在大多第三方 IDE 中，我们需要利用 Makefile 或 CMake 等工具来操作前述环境实现多文件编译的管理。

c) 使用 `#include <***.cpp>` 不是主流处理方法

这是本次实验中遇到的一个真实的错误。由于采用的基于 MinGW 的 Jetbrains CLion IDE 并不像 Visual Studio 一样自动将所有文件加入到项目中，期初没能正确的配置 CMakeLists.txt 文件，导致了无法正确编译的问题，而我尝试了将 `#include <***.cpp>` 的方式来解决这个问题。

但是事实上，这种做法是错误的。`#include` 本质上是将文件的内容直接插入到当前文件中，而不是将文件编译为目标文件。这种操作不仅不是真正意义上的多文件结构，而且会导致重复定义的问题。

> **正确的 CMakeLists.txt**
>
> ```
> cmake_minimum_required(VERSION 3.27)
> project(exp02exer3)
>
> set(CMAKE_CXX_STANDARD 11)
>
> add_executable(exp02exer3 my_area.h my_circle.cpp my_rect.cpp my_main.cpp)
> ```

d) inline 函数不适用于单独的多文件结构文件

期初，由于没正确理解多文件结构，使用了 `#include <***.cpp>` 编译的代码反而能够正确运行；而在正确配置 CMakeLists.txt 之后出现了无法构建的问题：

```
====================[ 构建 | exp02exer3 | Debug ]================================
%CLion%\bin\cmake\win\x64\bin\cmake.exe --build "OOP-class\exp02\exer3\cmake-
build-debug" --target exp02exer3 -j 10
Error: could not load cache
```

经查，这是由于在多文件结构练习中，画蛇添足地引入了内联函数。内联函数的定义不适用于单独的 `.cpp` 文件。错误来自以下函数：

```
inline double circle(double r);
```

```
inline double rect(double l, double w);
```

尝试两种解决解决方案，均成功：

- 将内联函数的定义直接放在头文件中 (但这不符合本题目的要求)
- 取消内联函数的定义，改为正常的普通函数。

这进一步说明了：

- `inline` 函数是不同于标准函数的，它的含义是将函数的定义直接插入到调用处，而不是将函数编译为目标文件。因此，`inline` 函数不能生成独立的目标文件，也不能被链接。或者说，`inline` 严格意义上并不是标准的函数，而是一种编译器的优化手段，更近似于以函数写法表示的宏定义。
- 多文件结构并不是简单的把多个文件的代码拼接在一起，而是将不同的文件编译为不同的目标文件，再链接为一个可执行文件。