第二节课

If条件判断的使用

```
if (condition) {
    // 如果条件为真，执行这里的代码
} else if (condition) {
    // 如果上面的条件为假，但这个条件为真，执行这里的代码
} else {
    // 如果上面的条件都为假，执行这里的代码
}
```

在这里有几点可以关注一下：

- 可以只写if而不写else，如果你不需要else这个条件的话
- 如果有多个if-else结构，判断的顺序是从上往下的，也就是全部不满足后再执行最后的else的内容
- 如果condition为整数，那么`非零的整数都认为是真`，不过还是不建议将整数作为判断真假的方式，建议条件写全可读性更强，比如`conditon==0`或者`conditon!=0`

如果不写`{}`包含代码块，就只会默认执行紧跟的后面的第一句话，后面便不会包含在里面

例如：

```
if (condition)
    cout << "hello";
    cout << " world!" << endl; // 这句话不会被if所约束，永远会执行
```

在上面的示例中，只有`cout << "hello";`和if相关

---

while循环的使用

```
while (condition) {
    // 循环体内的代码
}
```

1. `while`关键字后面的括号中是条件表达式。条件表达式的结果应该是一个布尔值（`true`或`false`）。只要条件为`true`，循环体内的代码将会一直执行。当条件为`false`时，循环将结束，程序将继续执行循环之后的代码。
2. 循环体内的代码是要重复执行的代码块。你可以在这里执行任意数量的语句，包括条件语句（如`if`语句）、输入输出语句、变量操作等。

下面是一个示例，演示了`while`循环的用法：

```cpp
#include <iostream>
int main() {
    int count = 0;

    while (count < 5) {
        std::cout << "当前计数值为：" << count << std::endl;
        count++;
    }

    std::cout << "循环结束" << std::endl;

    return 0;
}
```

---

do-while循环的使用

```cpp
do {
    // 循环体内的代码
} while (condition);
```

注意do-while和while的区别：

- 会先执行do里面的内容再判断while的condition，也就是**循环体内的代码至少会执行一次**

do-while循环平时几乎不常用，平时最常用的是**while和for循环**，不过，**do-while有一个非常重要的使用地方，配合宏定义#define使用，在这里卖一个坑，后面讲到宏定义再讲**

---

for循环的使用

```cpp
for（初始化表达式；条件表达式；更新表达式）{
    // 循环体内的代码
}
```

1. 初始化表达式：在进入循环之前执行一次的表达式。通常用于初始化循环计数器或设置其他变量的初始值。
2. 条件表达式：在每次循环迭代开始前进行检查的表达式。只要条件为`true`，循环体内的代码将会执行。如果条件为`false`，循环将结束，程序将继续执行循环之后的代码。
3. 更新表达式：在每次循环迭代结束后执行的表达式。通常用于更新循环计数器或进行其他的变量操作。

示例：

```
for (int i = 0; i < 5; i++) {
        std::cout << "当前计数值为：" << i << std::endl;
}
```

循环结构总结

注意区分for循环和while循环和do-while循环的相似处：

- while会先判断条件再执行结构体
- do-while 一定会先执行一遍结构体再和while相同
- for会先执行一遍初始化的代码，再和while相同，每次循环后会执行一遍结束的代码

---

break打断循环的操作

```
int count = 0;

while (true) {
    std::cout << "当前计数值为：" << count << std::endl;
    count++;

    if (count == 5) {
        break;  // 当计数值达到5时，打断循环
    }
}
```

比如上面的代码，while处是一个死循环，使用break可以跳出这个while循环

🤔：如果有多个while循环嵌套，再使用break会发生什么？自己试一下？

- [ ] 退出到最外层的循环
- [ ] 只退出离break最近的一个循环

---

continue 继续完成当前循环

你可以使用continue跳过当前循环迭代的剩余部分，继续下一次循环迭代。

`continue`语句用于提前结束当前迭代，程序将跳过当前迭代的代码，并继续执行下一次迭代。

示例：

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) {
        continue;  // 当 i 等于 3 时，跳过当前迭代
    }
    std::cout << "当前计数值为：" << i << std::endl;
}
```

在这个示例中，**i==3时不会打印输出**

---

switch-case使用

语法：

```
switch (表达式) {
    case 值1:
        // 当表达式的值等于值1时执行的代码
        break;
    case 值2:
        // 当表达式的值等于值2时执行的代码
        break;
    // 可以有更多的case语句
    default:
        // 当表达式的值与之前的所有case不匹配时执行的代码
        break;
}
```

1. 程序会将`表达式`的值与每个case后面的值进行比较，如果找到匹配的值，就执行相应的代码块。**如果没有找到匹配的值，将会执行default语句（如果有的话）。**
2. 执行匹配的代码块时，程序会从匹配的case处开始执行，直到遇到`break`语句或switch-case结尾。`break`语句用于跳出switch-case结构，避免执行其他case的代码。
3. 如果没有在匹配的代码块中遇到`break`语句，程序将会继续执行后续的case代码块，直到遇到`break`语句或switch-case结尾。

示例：

```
switch ('A') {
    case 64 :
        cout << "64" << endl; break;
    case 65 :
        cout << "65" << endl; break;
    case 66 :
        cout << "66" << endl; break;
    case 67 :
        cout << "67" << endl; break;
    default: cout << "default" << endl; break;
}
```

💡：这里的表达式能用哪些类型？int和char可以进行对比吗？string可以进行对比吗？如果不行的话为什么不行，那么又应该用什么方式进行对比呢？

> 比较字符串大小的函数 strcmp 在string.h文件中
>
> ```
> cout << strcmp("apple","apple") << endl;
> cout << strcmp("apple","apple1") << endl;
> cout << strcmp("apple","appl") << endl;
> cout << strcmp("apple","aaa") << endl;
> cout << strcmp("apple","zz") << endl;
> ```

宏定义的使用

语法：

```
#define 宏名 替换文本
```

示例：

```
#define PI 3.14159
```

1. 宏定义是**简单的文本替换**，没有类型检查，所以要确保在替换时不会引发错误。
2. 宏定义的替换文本中如果包含多个语句，请使用花括号将它们括起来，以确保语句块的完整性。
3. 宏定义的作用域从定义的位置开始，一直到文件末尾或遇到

对于宏定义，同样**使用g++ -E获取预处理后的内容**

使用宏定义替换函数的效果

```
#define MAX(a, b)  a>b?a:b
```

🤔一下这样会有什么问题，需要做什么改进

宏定义进阶

```
#define VAR(x) #x<<"="<<x
```

使用这个可以更方便的打印输出结果。

比如 `cout << VAR(i) << endl;` 会输出 `i=*`

使用宏定义替代多个操作

```
#define debug  cout << __FILE__ << ": " ; \
        cout << __LINE__ << endl;
```

比如现在想输出debug打印当前文件名和行数，像这样是没有问题的。

但是！如果碰到if这种，debug是两句话，那么不就是只会约束到第一句话，`cout << __LINE__ << endl;` 永远都会被执行到了

这种的解决方法就是需要用到之前提到的do-while,改进代码为

```
#define debug  do{cout << __FILE__ << ": " ; \
      cout << __LINE__ << endl;}while(false);
```

函数使用

在C++中，函数由函数声明和函数定义两部分组成。

函数声明用于向编译器提供有关函数名称、参数类型和返回类型的信息。函数声明通常放在头文件（.h）中，以便在多个源文件中共享。

函数定义包括实际的函数代码，用于定义函数的行为。函数定义通常放在源文件（.cpp）中。

```
// 函数声明
int add(int a, int b);

// 函数定义
int add(int a, int b)
{
    return a + b;
}
```

函数的声明和定义应该相同

参数传递

函数可以接受参数，用于向函数传递数据。在C++中，参数可以通过值传递、引用传递或指针传递。

- 值传递：参数的值被复制给函数中的形式参数，**对形式参数的修改不会影响实际参数。**
- 引用传递：通过引用传递参数，函数可以直接访问并修改实际参数的值。
- 指针传递：**通过指针传递参数，函数可以通过指针访问并修改实际参数的值。**

使用函数传递数组

写一个二分查找的示例：

```
int BinarySearch(int nums[], int n,int target) {
  int left{0};
  int right = n - 1;
  int mid = (right - left) / 2 + left;

  while (left <= right) {
    mid = (right - left) / 2 + left;
    if (nums[mid] > target) {
      right = mid - 1;
    } else if (nums[mid] < target) {
      left = mid + 1;
    } else {
      return mid;
    }
  }
  return mid;
};
```

多文件操作

在多文件包含中，常见的做法是将函数的声明放在头文件中（通常使用`.h`扩展名），将函数的定义和其他实现放在源文件中（通常使用`.cpp`或`.cxx`扩展名）。头文件中包含了函数的声明和可能需要使用的其他头文件，源文件中包含了函数的实现。

头文件示例`add.h`

```
#ifndef ADD_H
#define ADD_H

int add(int a, int b);

#endif
```

源文件示例`add.cpp`

```
#include "add.h"

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

主文件示例`main.cpp`

```
#include <iostream>
#include "add.h"

int main()
{
    int a = 5;
    int b = 3;
    int result = add(a, b);

    std::cout << "The sum is: " << result << std::endl;

    return 0;
}
```

小任务：尝试将之前的二分查找内容写到多个文件中去。