

C++ Tutorial

<https://www.sololearn.com/Play/Cplusplus>

翻译： Vincent

2018/11/10

linqinghong@email.szu.edu.cn

1. 基本概念.....	3
1.1. 什么是 C++.....	3
1.2. Hello, World.....	3
1.3. 采用工具.....	6
1.4. 输出文本.....	8
1.5. 注释.....	12
1.6. 变量.....	13
1.7. 变量的使用.....	15
1.8. 更多关于变量.....	17
1.9. 基本算术.....	17
1.10. 赋值和自增运算符.....	20
2. 条件和循环.....	23
2.1. if 语句.....	23
2.2. else 语句.....	24
2.3. while 循环.....	27
2.4. 使用 while 循环.....	29
2.5. for 循环.....	30
2.6. do... while 循环.....	32
2.7. switch 语句.....	34
2.8. 逻辑运算符.....	36
3. 数据类型、数组、指针.....	40
3.1. 数据类型简介.....	40
3.2. 整型、单浮点型、双浮点型.....	41
3.3. 字符串、字符型、布尔型.....	42
3.4. 变量命名规则.....	43
3.5. 数组.....	44
3.6. 在循环中使用数组.....	45
3.7. 计算中的数组.....	46
3.8. 多维数组.....	47
3.9. 指针简介.....	48
3.10. 更多关于指针.....	50
3.11. 动态内存.....	51
3.12. sizeof() 运算符.....	53
4. 函数.....	55
4.1. 函数简介.....	55
4.2. 函数参数.....	57
4.3. 多参数函数.....	59
4.4. rand() 函数.....	60
4.5. 默认参数.....	63
4.6. 函数重载.....	64
4.7. 递归.....	66
4.8. 传递数组给函数.....	67
4.9. 用指针传递引用.....	68
5. 类和对象.....	70

5.1. 什么是对象.....	70
5.2. 什么是类.....	71
5.3. 类的例子.....	72
5.4. 抽象化.....	73
5.5. 封装.....	75
5.6. 封装的例子.....	75
5.7. 构造函数.....	78
6. 更多关于类.....	81
6.1. 单独的类文件.....	81
6.2. 析构函数.....	84
6.3. 选择操作符.....	86
6.4. 常量对象.....	88
6.5. 成员初始化.....	90
6.6. 组合 1.....	92
6.7. 组合 2.....	94
6.8. Friend 关键字.....	95
6.9. This 关键字.....	97
6.10. 运算符重载.....	99
7. 继承与多态.....	102
7.1. 继承.....	102
7.2. 受保护的成员.....	104
7.3. 派生类的构造函数和析构函数.....	105
7.4. 多态.....	107
7.5. 虚函数.....	110
7.6. 抽象类.....	111
8. 模板、异常和文件.....	115
8.1. 函数模板.....	115
8.2. 多参数函数模板.....	117
8.3. 类模板.....	118
8.4. 模板特化.....	121
8.5. 异常.....	122
8.6. 更多关于异常.....	124
8.7. 使用文件.....	126
8.8. 更多关于文件.....	127

1.基本概念

1.1.什么是 C ++

✧ 欢迎使用 C ++

- C ++是一种通用编程语言。
- C ++用于创建计算机程序。艺术应用，音乐播放器甚至电子游戏！

C ++源自 C，主要基于它。

1.2.Hello,World

✧ 你的第一个 C ++程序

- C ++程序是命令或语句的集合。
- 下面是一个简单的代码，其中包含"Hello world!"作为其输出。

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

int main()
{
    cout << "Hello world!";
    return 0;
}
```

让我们分解代码的各个部分。

```
#include <iostream>
```

• C ++提供了各种标题，每个标题包含程序正常工作所需的信息。此特定程序调用标头<iostream>。

• 行开头的**数字符号(#)**以编译器的预处理器为目标。在这种情况下，**#include** 告诉预处理器包含<iostream>头。

<iostream>标头定义了输入和输出数据的标准流对象。

✧ 你的第一个 C++ 程序

- C++ 编译器忽略空行。
- 通常，空行用于提高代码的可读性和结构。

空格，例如空格，制表符和换行符也会被忽略，尽管它用于增强程序的关注度。

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

int main()
{
    cout << "Hello world!";
    return 0;
}
```

- 在我们的代码中，**using namespace std;** 告诉编译器使用 **std**（标准）命名空间。**std** 命名空间包含 C++ 标准库的功能。

✧ Main

- 程序执行从 main 函数 **int main()** 开始。

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

int main()
{
    cout << "Hello world!";
    return 0;
}
```

- 花括号 {} 表示函数的开头和结尾，也可以称为函数体。括号内的信息表示执行时函数的作用。

无论程序做什么，每个 C++ 程序的入口点都是 **main()**。

✧ 你的第一个 C++ 程序

- 下一行，**cout << "Hello world!";** 结果显示“Hello world!” 到屏幕。

```
#include <iostream>
```

```
using namespace std;

int main()
{
    cout << "Hello world!";
    return 0;
}
```

- 在 C++ 中，**流**用于执行输入和输出操作。
- 在大多数程序环境中，标准默认输出目标是屏幕。在 C++ 中，**cout** 是用于访问它的流对象。
- **cout** 与插入操作符结合使用。将插入运算符写为<<以将之后的数据插入到之前的流中。

在 C++ 中，**分号**用于终止语句。每个语句必须以**分号**结尾。它表示一个逻辑表达式的结束。

✧ 声明

- **块**是一组逻辑连接的语句，由花括号括起。

例如：

```
{
    cout << "Hello world!";
    return 0;
}
```

只要记住用**分号**结束每个语句，就可以在一行上有多个语句。如果不这样做将导致错误。

✧ 返回

- 程序中的最后一条指令是 **return** 语句。该行 **return 0;** 终止 **main()** 函数并使其将值 0 返回给调用进程。非零值（通常为 1）表示异常终止。

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

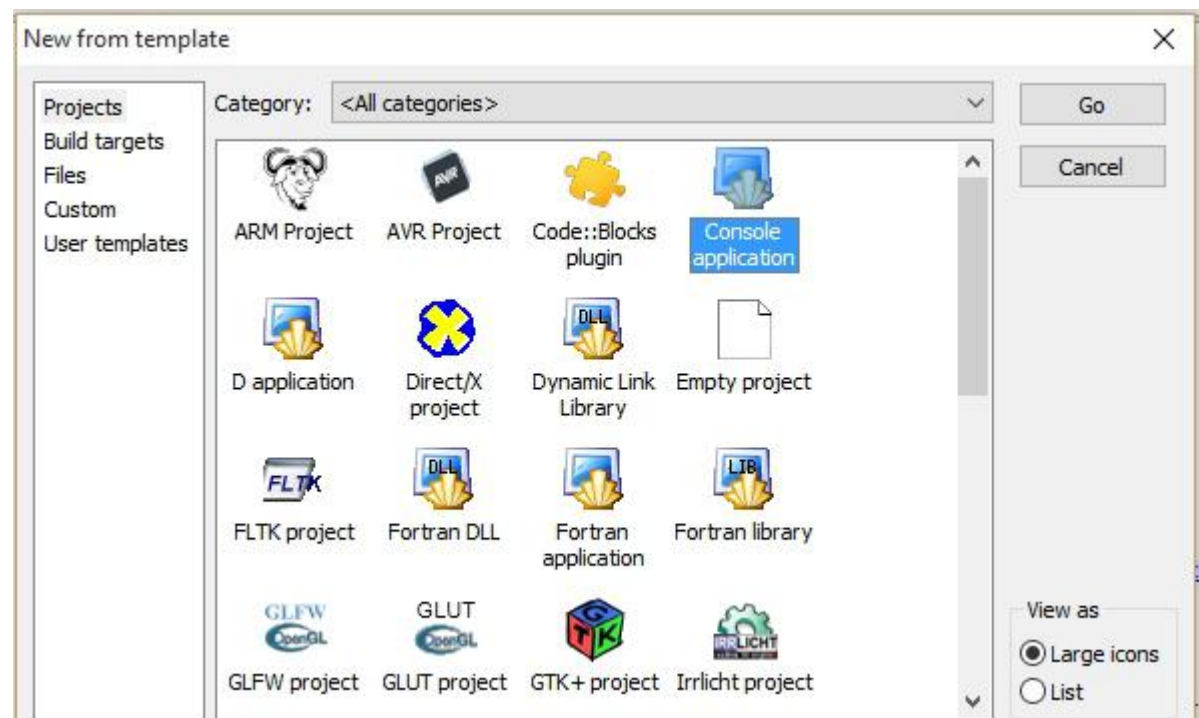
int main()
{
    cout << "Hello world!";
    return 0;
}
```

如果遗忘 `return` 语句，C++编译器会隐式插入"`return 0`"; 到 `main()`函数的末尾。

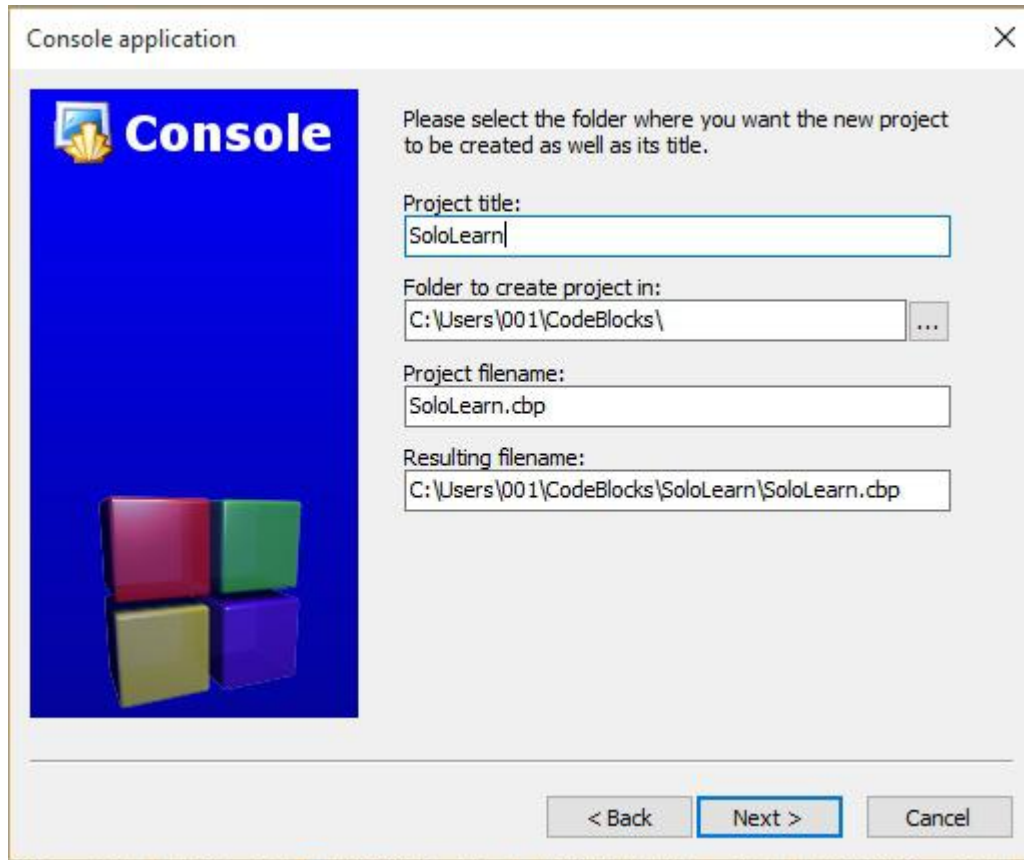
1.3.采用工具

✧ 采用工具

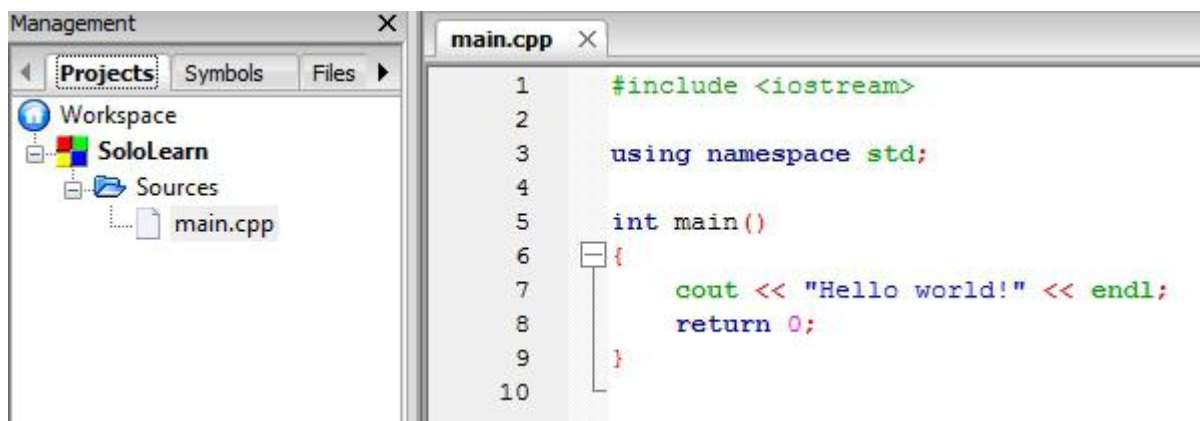
•要创建项目，请打开 `Code :: Blocks` 并单击“**创建新项目**”(或 `File-> New-> Project`)。这将打开项目模板的对话框。选择 **Console 应用程序**，然后单击 **Go**。



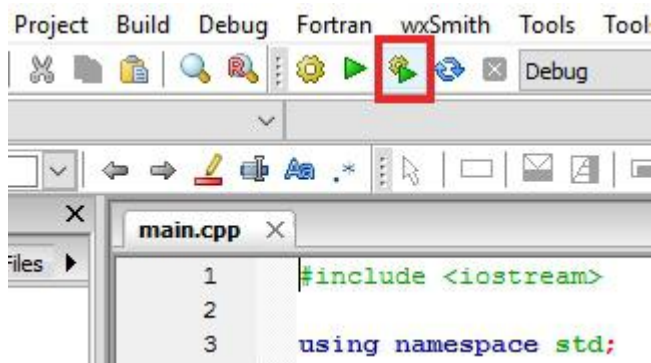
- 完成向导，确保选择 C++ 作为语言。
- 为项目命名并指定要将其保存到的文件夹。



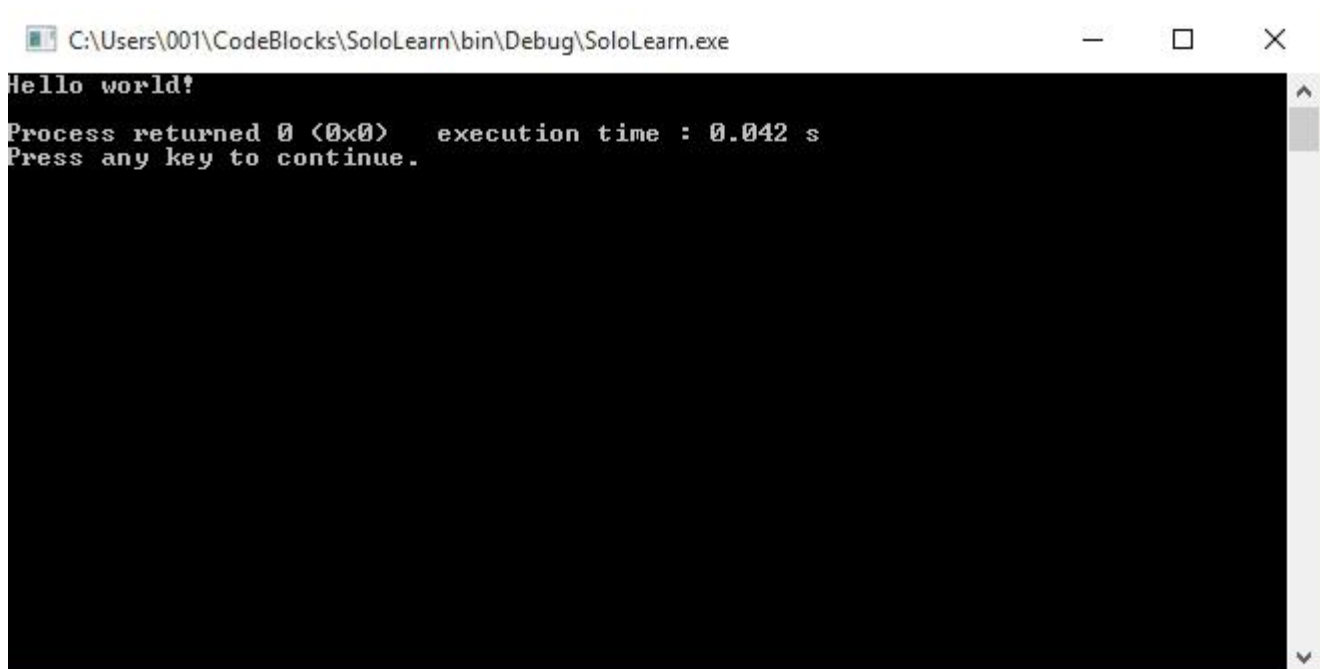
- 确保选中“编译器”，然后单击“完成”。
- **GNU GCC** 是 Code :: Blocks 的流行编译器之一。
- 在左侧边栏上，展开“**Sources**”。您将看到您的项目及其源文件。Code :: Blocks 自动创建了一个 **main.cpp** 文件，其中包含一个基本的 Hello World 程序（C++ 源文件具有 .cpp、.cp 或 .c 扩展名）。



- 单击工具栏中的“构建并运行”图标以编译并运行该程序。



- 控制台窗口将打开并显示程序输出。



- 恭喜！你刚刚编译并运行了你的第一个 C++ 程序！

您可以在我们的 **Code Playground** 上运行，保存和共享 C++ 代码，而无需安装任何其他软件。

如果需要在计算机上安装软件，请参阅本课程。

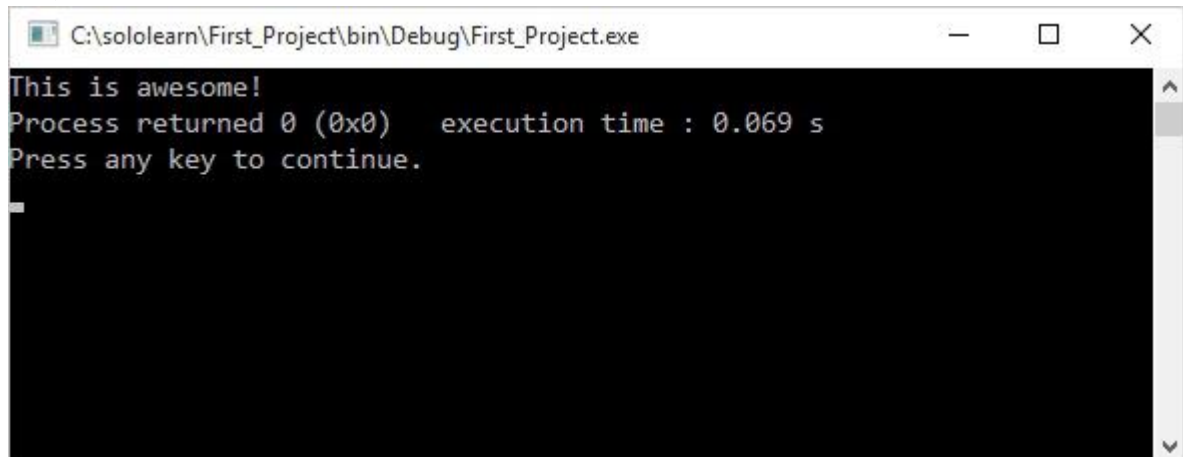
1.4. 输出文本

✧ 你的第一个 C++ 程序

- 您可以在 **cout** 之后添加多个插入运算符。

```
cout << "This " << "is " << "awesome!";
```

结果:



```
C:\sololearn\First_Project\bin\Debug\First_Project.exe
This is awesome!
Process returned 0 (0x0) execution time : 0.069 s
Press any key to continue.
```

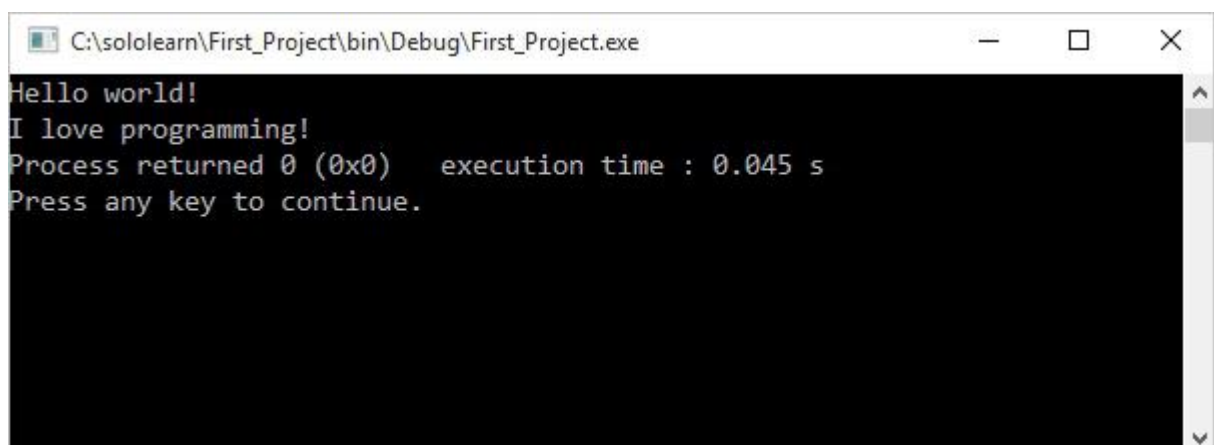
✧ 新行

- `cout` 运算符不会在输出结尾处插入换行符。
- 打印两行的一种方法是使用 `endl` 操纵符，它将放入换行符。

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

int main()
{
    cout << "Hello world!" << endl;
    cout << "I love programming!";
    return 0;
}
```

- `endl` 操纵符向下移动到新行以打印第二个文本。



```
C:\sololearn\First_Project\bin\Debug\First_Project.exe
Hello world!
I love programming!
Process returned 0 (0x0) execution time : 0.045 s
Press any key to continue.
```

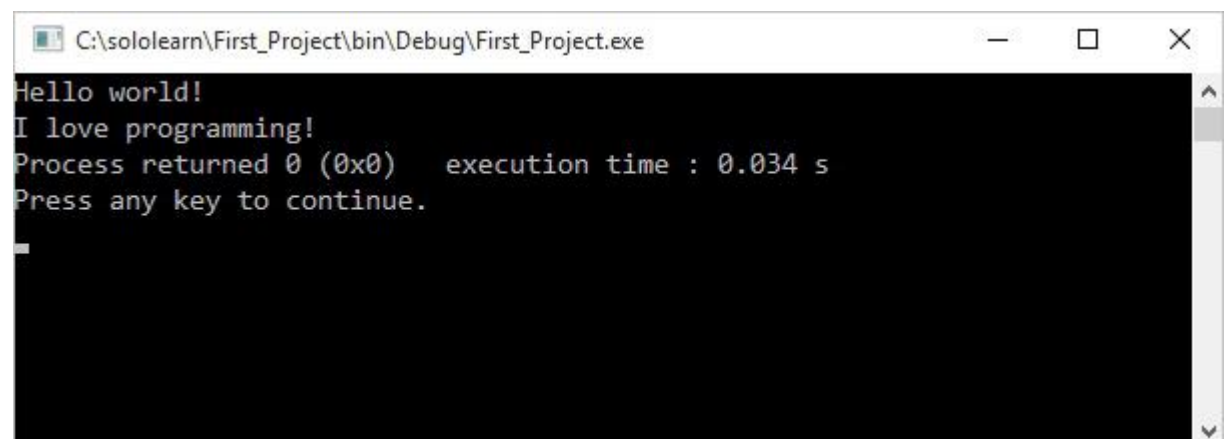
◇ 新行

- 新行字符 `\n` 可以用作 `endl` 的替代。
- 反斜杠(`\`)称为**转义字符**，表示“特殊”字符。

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

int main()
{
    cout << "Hello world! \n";
    cout << "I love programming!";
    return 0;
}
```

结果：



```
C:\sololearn\First_Project\bin\Debug\First_Project.exe
Hello world!
I love programming!
Process returned 0 (0x0)   execution time : 0.034 s
Press any key to continue.
```

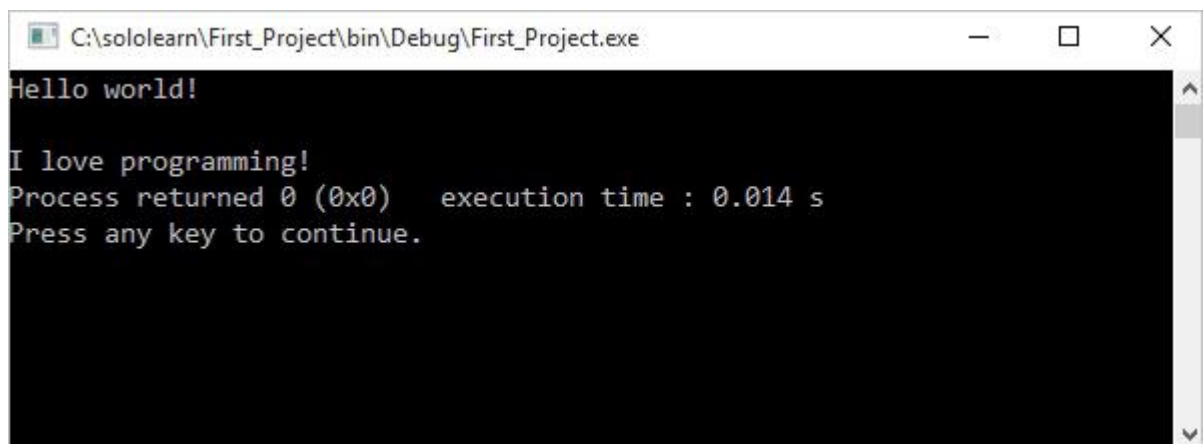
◇ 新行

- 两个换行符放在一起会产生一个空行。

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

int main()
{
    cout << "Hello world! \n\n";
    cout << "I love programming!";
    return 0;
}
```

结果:



```
C:\sololearn\First_Project\bin\Debug\First_Project.exe
Hello world!
I love programming!
Process returned 0 (0x0) execution time : 0.014 s
Press any key to continue.
```

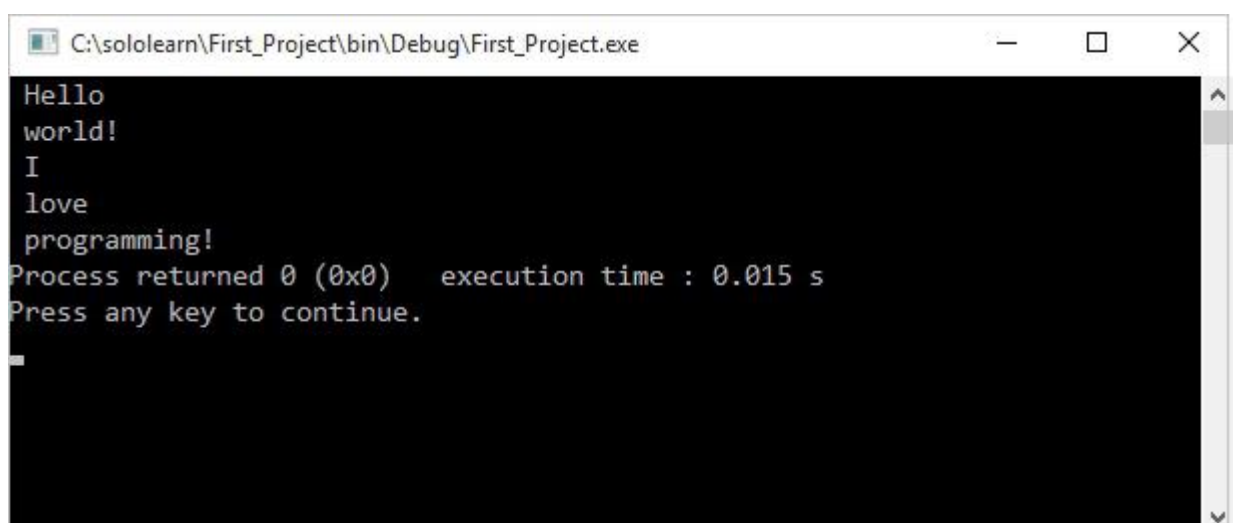
✧ 多个新行

- 使用单个 **cout** 语句，其中包含与程序一样多的 **\n** 实例，将打印出多行文本。

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

int main()
{
    cout << " Hello \n world! \n I \n love \n programming!";
    return 0;
}
```

结果:



```
C:\sololearn\First_Project\bin\Debug\First_Project.exe
Hello
world!
I
love
programming!
Process returned 0 (0x0) execution time : 0.015 s
Press any key to continue.
```

1.5.注释

✧ 注释

- **注释**是可以包含在 C++ 代码中的解释性语句，用于解释代码正在执行的操作。
- 编译器会忽略注释中出现的所有内容，因此结果中不会显示任何信息。

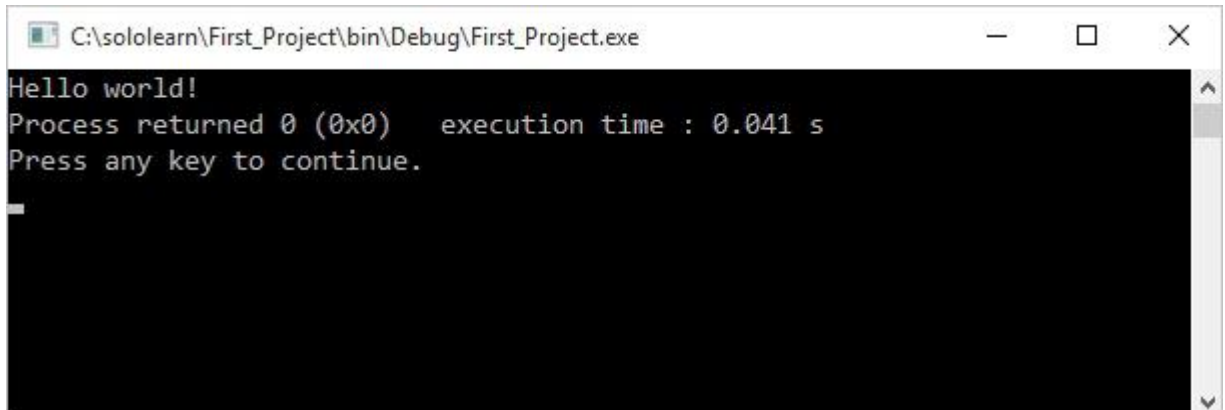
• 以**两斜杠(//)**开头的注释称为单行注释。斜杠告诉编译器忽略后面的所有内容，直到行尾。

例如：

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

int main()
{
    // prints "Hello world"
    cout << "Hello world!";
    return 0;
}
```

- 编译上面的代码时，它会忽略**//prints "Hello world"**语句并产生以下结果：



注释使您的代码对其他人更具可读性。

✧ 多行注释

- 需要多行的注释以**/***开头，以***/**结尾
- 您可以将它们放在同一条线上或在它们之间插入一条或多条线。

```
/* This is a comment */

/* C++ comments can
   span multiple lines
  */
```

如果您编写错误的代码段，请不要立即删除它。将其放入多行注释中，然后在找到正确的解决方案时将其删除。

◇ 使用注释

- 注释可以在任何地方写入，并且可以在整个代码中重复多次。
- 在标有 `/*` 和 `*/` 的注释中，`//` 字符没有特殊含义，反之亦然。这允许您在另一种注释中“嵌套”。

```
/* Comment out printing of Hello world!

cout << "Hello world!"; // prints Hello world!

*/
```

在代码中添加注释是一种很好的做法。它有助于您和其他阅读它的人清楚地理解代码。

1.6. 变量

◇ 变量

- 创建**变量**会保留内存位置或内存中用于存储值的空间。编译器要求您为声明的每个变量提供**数据类型**。
- C++ 提供丰富的内置和用户定义的数据类型。
- **整数**，内置类型，表示整数值。使用关键字 **int** 定义整数。
- C++ 要求您为定义的所有变量指定**类型**和**标识符**。
- **标识符**是变量、函数、类、模块或任何其他用户定义项的名称。标识符以字母 (A-Z 或 a-z) 或下划线 (`_`) 开头，后跟附加字母，下划线和数字 (0 到 9)。
- 例如，定义一个名为 **myVariable** 的变量，它可以保存**整数值**，如下所示：

```
int myVariable = 10;
```

不同的操作系统可以为相同的数据类型保留不同大小的内存。

✧ 变量

- 现在，让我们为变量赋值并打印它。

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

int main()
{
    int myVariable = 10;
    cout << myVariable;
    return 0;
}
// Outputs 10
```

C++编程语言区分大小写，因此 **myVariable** 和 **myvariable** 是两个不同的标识符。

✧ 变量

- 在程序中使用之前，使用**名称**和**数据类型**定义所有变量。如果您有多个相同类型的变量，则可以在一个声明中定义它们，用**逗号**分隔它们。

```
int a, b;
// defines two variables of type int
```

- 可以为变量分配值，并可以使用该变量执行操作。
- 例如，我们可以创建一个名为 **sum** 的附加变量，并将两个变量一起添加。

```
int a = 30;
int b = 15;
int sum = a + b;
// Now sum equals 45
```

使用+运算符添加两个数字。

✧ 变量

- 让我们创建一个程序来计算和打印两个整数的和。

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

int main()
{
    int a = 30;
    int b = 12;
    int sum = a + b;

    cout << sum;

    return 0;
}

//Outputs 42
```

请始终牢记，必须先使用**名称和数据类型**定义所有变量，然后才能使用它们。

1.7.变量的使用

✧ 接受用户输入

- 以下程序提示用户输入一个数字并将其存储在变量 **a** 中：

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

int main()
{
    int a;
    cout << "Please enter a number \n";
    cin >> a;

    return 0;
}
```


- 程序运行时，会显示消息“请输入一个数字”，然后等待用户输入一个数字，然后按 Enter 或 Return 键。
 - 输入的数字存储在变量 **a** 中。
- 只要用户需要输入数字，程序就会等待。

✧ 接受用户输入

- 您可以在整个程序中多次接受用户输入：

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

int main()
{
    int a, b;
    cout << "Enter a number \n";
    cin >> a;
    cout << "Enter another number \n";
    cin >> b;

    return 0;
}
```

✧ 接受用户输入

- 让我们创建一个程序，接受两个数字的输入并打印它们的总和。

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

int main()
{
    int a, b;
    int sum;
    cout << "Enter a number \n";
    cin >> a;
    cout << "Enter another number \n";
    cin >> b;
    sum = a + b;
    cout << "Sum is: " << sum << endl;
}
```

```
return 0;  
}
```

1.8.更多关于变量

✧ 变量

- 在声明变量时，只需要指定一次数据类型。
- 之后，可以在不参考数据类型的情况下使用该变量。

```
int a;  
a = 10;
```

多次指定给定变量的数据类型会导致语法错误。

✧ 变量

变量的值可以在整个程序中根据需要多次更改。

例如：

```
int a = 100;  
a = 50;  
cout << a;  
  
// Outputs 50
```

1.9.基本算术

✧ 算术运算符

- C++支持这些算术运算符。

Operator	Symbol	Form
Addition	+	$x + y$
Subtraction	-	$x - y$
Multiplication	*	$x * y$
Division	/	x / y
Modulus	%	$x \% y$

- 加法运算符将其操作数加在一起。

```
int x = 40 + 60;
cout << x;

// Outputs 100
```

您可以在一行中使用多个算术运算符。

✧ 减法

- 减法运算符从另一个减去一个操作数。

```
int x = 100 - 60;
cout << x;

//Outputs 40
```

✧ 乘法

- 乘法运算符乘以其操作数。

```
int x = 5 * 6;
cout << x;

//Outputs 30
```

✧ 除法

- 除法运算符将第一个操作数除以第二个操作数。删除任何余数以返回整数值。

例：

```
int x = 10 / 3;  
cout << x;  
  
// Outputs 3
```

- 如果一个或两个操作数是浮点值，则除法运算符执行浮点除法。

除以 0 将导致程序崩溃。

✧ 取余

- 模数运算符(%)非正式地称为余数运算符，因为它在整数除法后返回余数。

例如：

```
int x = 25 % 7;  
cout << x;  
  
// Outputs 4
```

✧ 运算符优先级

- 运算符**优先级**确定表达式中的元素分组，这会影响表达式的计算方式。某些运算符优先于其他运算符；例如，乘法运算符优先于加法运算符。

例如：

```
int x = 5+2*2;  
cout << x;  
// Outputs 9
```

- 上面的程序首先计算 $2 * 2$ ，然后将结果添加到 5。

- 与数学一样，使用**括号**会改变运算符优先级。

```
int x = (5 + 2) * 2;  
cout << x;  
  
// Outputs 14
```

◇ 运算符优先级

- 括号使得运算符具有更高的优先级。如果存在彼此嵌套的括号表达式，则首先计算最里面括号内的表达式。

如果没有表达式在括号中，则在**加法**（加法、减法）运算符之前将计算**乘法**（乘法、除法、模数）运算符。

1.10.赋值和自增运算符

◇ 赋值运算符

- 简单赋值运算符(=)将右侧分配给左侧。
 - C++提供了缩写运算符，它们能够同时执行操作和赋值。
- 例如：

```
int x = 10;  
x += 4; // equivalent to x = x + 4  
x -= 5; // equivalent to x = x - 5
```

赋值运算符(=)将右侧分配给左侧。

◇ 赋值运算符

- 相同的简写语法适用于乘法，除法和模数运算符。

```
x *= 3; // equivalent to x = x * 3  
x /= 2; // equivalent to x = x / 2  
x %= 4; // equivalent to x = x % 4
```

相同的简写语法适用于乘法，除法和模数运算符。

◇ 自增运算符

- **increment** 运算符用于将整数的值增加 1，并且是常用的 C++运算符。

```
x++; //equivalent to x = x + 1
```

increment 运算符用于将整数的值增加 1。

◇ 自增运算符

例如：

```
int x = 11;
```

```
x++;
```

```
cout << x;
```

```
// Outputs 12
```

◇ 自增运算符

- 增量运算符有两种形式，前缀和后缀。

```
++x; // prefix
```

```
x++; // postfix
```

- 前缀递增值，然后执行表达式。
- 后缀先执行表达式，然后执行递增。

前缀示例：

```
x = 5;
```

```
y = ++x;
```

```
// x is 6, y is 6
```

后缀示例：

```
x = 5;
```

```
y = x++;
```

```
// x is 6, y is 5
```

- 前缀示例增加 x 的值，然后将其赋给 y。
- 后缀示例将 x 的值赋给 y，然后递增它。

◇ 自减运算符

- 自减运算符(--)的工作方式与自增运算符的工作方式大致相同，但不是增加值，而是将其减 1。

```
--x; // prefix  
x--; // postfix
```

自减运算符(--)的工作方式与自增运算符的工作方式大致相同。

2.条件和循环

2.1.if 语句

✧ if 语句

- 使用关系运算符来衡量条件。

例如：

```
if (7 > 4) {  
    cout << "Yes";  
}  
  
// Outputs "Yes"
```

- if 语句判断条件(7> 4)，发现它为 **true**，然后执行 **cout** 语句。
- 如果我们将更大的运算符更改为小于运算符(7<4)，则不会执行该语句，也不会打印任何内容。

if 语句中指定的条件不需要分号。

✧ 关系运算符

- 其他关系运算符：

Operator	Description	Example
>=	Greater than or equal to	7 >= 4 True
<=	Less than or equal to	7 <= 4 False
==	Equal to	7 == 4 False
!=	Not equal to	7 != 4 True

例如：

```
if (10 == 10) {  
    cout << "Yes";  
}  
  
// Outputs "Yes"
```


✧ 关系运算符

- **不等于**运算符判断操作数，确定它们是否相等。如果操作数不相等，则将条件评估为 **true**。

例如：

```
if (10 != 10) {  
    cout << "Yes";  
}
```

上述条件的计算结果为 **false**，并且不执行代码块。

✧ 关系运算符

- 您可以使用关系运算符来比较 **if** 语句中的变量。

例如：

```
int a = 55;  
int b = 33;  
if (a > b) {  
    cout << "a is greater than b";  
}  
  
// Outputs "a is greater than b"
```

2.2.else 语句

✧ else 语句

- **if** 语句后面可以跟一个可选的 **else** 语句，该语句在条件为 **false** 时执行。

语法：

```
if (condition) {  
    //statements  
}  
else {  
    //statements
```

```
}
```

上面的代码将测试条件：

- 如果计算结果为 **true**，则执行 **if** 语句中的代码。
- 如果计算结果为 **false**，则执行 **else** 语句中的代码。

当在 **if/else** 中只使用一个语句时，可以省略花括号。

✧ else 语句

例如：

```
int mark = 90;

if (mark < 50) {
    cout << "You failed." << endl;
}
else {
    cout << "You passed." << endl;
}

// Outputs "You passed."
```

✧ else 语句

• 在前面的所有示例中，**if/else** 语句中只使用了一个语句，但您可以根据需要包含任意数量的语句。

例如：

```
int mark = 90;

if (mark < 50) {
    cout << "You failed." << endl;
    cout << "Sorry" << endl;
}
else {
    cout << "Congratulations!" << endl;
    cout << "You passed." << endl;
    cout << "You are awesome!" << endl;
}
```

```
/* Outputs
Congratulations!
You passed.
You are awesome!
*/
```

✧ 嵌套 if 语句

- 您还可以在另一个 if 语句中包含或嵌套 if 语句。

例如：

```
int mark = 100;

if (mark >= 50) {
    cout << "You passed." << endl;
    if (mark == 100) {
        cout << "Perfect!" << endl;
    }
}
else {
    cout << "You failed." << endl;
}

/*Outputs
You passed.
Perfect!
*/
```

✧ 嵌套 if 语句

- C++ 提供了嵌套无数的 if / else 语句的选项。

例如：

```
int age = 18;
if (age > 14) {
    if (age >= 18) {
        cout << "Adult";
    }
    else {
        cout << "Teenager";
    }
}
```

```

}
else {
    if (age > 0) {
        cout << "Child";
    }
    else {
        cout << "Something's wrong";
    }
}
}

```

请记住，所有 **else** 语句必须具有相应的 **if** 语句。

✧ if else 语句

- 在 if / else 语句中，可以包含单个语句而不将其括在花括号中。

```

int a = 10;
if (a > 4)
    cout << "Yes";
else
    cout << "No";

```

无论如何，包括花括号是一个很好的做法，因为它们澄清了代码并使其更容易阅读。

2.3.while 循环

✧ 循环

- 循环重复执行一组语句，直到满足特定条件。
- 只要给定条件保持为真，**while** 循环语句就会重复执行目标语句。

语法：

```

while (condition) {
    statement(s);
}

```

当条件为真时，循环迭代。

在条件变为假的时刻，程序控制转移到紧跟循环的行。

✧ while 循环

- 循环体是大括号内的语句块。

例如：

```
int num = 1;
while (num < 6) {
    cout << "Number: " << num << endl;
    num = num + 1;
}

/* Outputs
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
*/
```

- 上面的示例声明了一个等于 1 的变量（`int num = 1`）。
- **while** 循环检查条件(`num < 6`)，并执行其正文中的语句，每次循环运行时，`num` 的值增加 1。

在第 5 次迭代之后，**num** 变为 6，并且条件被评估为 **false**，并且循环停止运行。

✧ while 循环

- 增量值可以更改。如果更改，循环运行的次数也将更改。

```
int num = 1;
while (num < 6) {
    cout << "Number: " << num << endl;
    num = num + 3;
}

/* Outputs
Number: 1
Number: 4
*/
```

如果没有最终将循环条件判断为 **false** 的语句，循环将无限地运行。

2.4.使用 while 循环

✧ 使用增量或减量

- 增量或减量运算符可用于更改循环中的值。

例如：

```
int num = 1;
while (num < 6) {
    cout << "Number: " << num << endl;
    num++;
}

/* Outputs
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
*/
```

num ++相当于 num = num + 1。

✧ 使用 while 循环

- 循环可用于从用户获得多个输入。
- 让我们创建一个程序，允许用户输入数字 5 次，每次都将在一个变量中。

```
int num = 1;
int number;

while (num <= 5) {
    cin >> number;
    num++;
}
```

上面的代码要求用户输入 5 次，每次都将在数字变量中。

✧ 使用 while 循环

- 现在让我们修改我们的代码来计算用户输入的数字之和。

```
int num = 1;
int number;
int total = 0;

while (num <= 5) {
    cin >> number;
    total += number;
    num++;
}
cout << total << endl;
```

- 上面的代码将每次循环迭代时用户输入的数字添加到 **total** 变量中。
- 循环停止执行后，将打印 **total** 的值。该值是用用户输入的所有数字的总和。

请注意，变量 **total** 的初始值为 0。

2.5.for 循环

✧ for 循环

- **for** 循环是一种重复控制结构，允许您有效地编写执行特定次数的循环。

语法：

```
for ( init; condition; increment ) {
    statement(s);
}
```

- 首先执行 **init** 步骤，不重复。
- 接下来，判断 **condition**，如果条件为真，则执行循环体。
- 在下一步中，**increment** 语句更新循环控制变量。
- 然后，循环体重复自身，仅在条件变为假时停止。

例如：

```
for (int x = 1; x < 10; x++) {
    // some code
}
```

如果不需要，可以省略 **init** 和 **increment** 语句，但请记住分号是必需的。

✧ for 循环

- 下面的示例使用 **for** 循环打印 0 到 9 之间的数字。

```
for (int a = 0; a < 10; a++) {  
    cout << a << endl;  
}
```

```
/* Outputs
```

```
0
```

```
1
```

```
2
```

```
3
```

```
4
```

```
5
```

```
6
```

```
7
```

```
8
```

```
9
```

```
*/
```

- 在 **init** 步骤中，我们声明了一个变量 **a** 并将其设置为 0。
- **a < 10** 是条件。
- 每次迭代后，执行 **a ++** 自增语句。

当 **a** 增到 10 时，条件计算结果为 **false**，循环停止。

✧ for 循环

- 可以更改增量语句。

```
for (int a = 0; a < 50; a+=10) {  
    cout << a << endl;  
}
```

```
/* Outputs
```

```
0
```

```
10
```

```
20
```

```
30
```

```
40
```

```
*/
```

- 您还可以在语句中使用递减。

```
for (int a = 10; a >= 0; a -= 3) {
```



```

    cout << a << endl;
}

/* Outputs
10
7
4
1
*/

```

使用 for 循环时，不要忘记 **init** 和 **condition** 语句后面的分号。

2.6.do... while 循环

✧ do...while 循环

- 与在头部测试循环条件的 **for** 和 **while** 循环不同，**do ... while** 循环在循环底部检查其条件。
- **do ... while** 循环类似于 **while** 循环。一个区别是 **do ... while** 循环保证至少执行一次。

语法：

```

do {
    statement(s);
} while (condition);

```

例如，您可以从用户那里获取输入，然后进行检查。如果输入错误，您可以再次输入。

✧ do...while 循环

这是一个例子：

```

int a = 0;
do {
    cout << a << endl;
    a++;
} while(a < 5);

/* Outputs
0

```

```
1
2
3
4
*/
```

不要忘记 **while** 语句后面的分号。

✧ **while v.s.do...while** 循环

- 如果条件判断为 **false**，则 **do** 中的语句仍将运行一次：

```
int a = 42;
do {
    cout << a << endl;
    a++;
} while(a < 5);

// Outputs 42
```

- **do ... while** 循环至少执行一次语句，然后判断条件。
- **while** 循环在判断条件之后执行语句。

✧ **while v.s.do...while** 循环

- 与其他循环一样，如果循环中的条件永远不会计算为 **false**，则循环将永远运行。

例如：

```
int a = 42;
do {
    cout << a << endl;
} while (a > 0);
```

- 这将**永远**打印到屏幕 42。

始终测试您的循环，以便您知道它们以您期望的方式运行。

2.7.swith 语句

✧ 多条件

• 有时需要判断变量是否与多个值相等。这可以使用多个 if 语句来实现。
例如：

```
int age = 42;
if (age == 16) {
    cout << "Too young";
}
if (age == 42) {
    cout << "Adult";
}
if (age == 70) {
    cout << "Senior";
}
```

在这种情况下，**switch** 语句是一种更优雅的方案。

✧ switch 语句

• **switch** 语句根据值列表（称为 **case**）判断变量，以确定它是否等于其中的一个。

```
switch (expression) {
    case value1:
        statement(s);
        break;
    case value2:
        statement(s);
        break;
    ...
    case valueN:
        statement(s);
        break;
}
```

• Switch 计算表达式以确定它是否等于 **case** 语句中的值。如果找到匹配项，则在该情况下执行语句。

一个开关可以包含任意数量的 **case** 语句，后面跟着相应的值和冒号。

✧ switch 语句

- 以下是使用单个 **switch** 语句编写的上一个示例：

```
int age = 42;
switch (age) {
    case 16:
        cout << "Too young";
        break;
    case 42:
        cout << "Adult";
        break;
    case 70:
        cout << "Senior";
        break;
}
```

- 上面的代码相当于三个 **if** 语句。

注意在每个情况之后的关键词 **break**。这将很快介绍。

✧ 默认情况

- 在 **switch** 语句中，当没有确定任何情况为真时，可以使用可选的 **default** 情况来执行任务。

例：

```
int age = 25;
switch (age) {
    case 16:
        cout << "Too young";
        break;
    case 42:
        cout << "Adult";
        break;
    case 70:
        cout << "Senior";
        break;
    default:
        cout << "This is the default case";
}

// Outputs "This is the default case"
```

- 当没有任何案例与 **switch** 表达式匹配时，将执行 **default** 语句的代码。

default 情况必须出现在 **switch** 的末尾。

✧ **break** 语句

- **break** 语句的作用是终止 **switch** 语句。
- 在变量属于某一值的情况下，之后的语句将继续执行，直到遇到 **break** 语句。换句话说，省略 **break** 语句会导致之后的所有语句被执行，甚至是那些与表达式不匹配的语句。

例如：

```
int age = 42;
switch (age) {
    case 16:
        cout << "Too young" << endl;
    case 42:
        cout << "Adult" << endl;
    case 70:
        cout << "Senior" << endl;
    default:
        cout << "This is the default case" << endl;
}
/* Outputs
Adult
Senior
This is the default case
*/
```

- 如您所见，程序执行匹配的 **case** 语句，将“**Adult**”打印到屏幕上。如果没有指定的 **break** 语句，语句将在匹配的大小写后继续运行。因此，打印了所有其他案例陈述。这种行为称为“直达”。

作为 **switch** 语句的最后一种情况，**default** 情况下不需要 **break** 语句。

break 语句也可用于打破循环。

2.8.逻辑运算符

✧ 逻辑运算符

- 使用逻辑运算符组合条件语句并返回 **true** 或 **false**。

Operator	Name of Operator	Form
&&	AND Operator	y && y
	OR Operator	x y
!	NOT Operator	! x

- **AND** 运算符以下列方式工作：

Left Operand	Right Operand	Result
false	false	false
false	true	false
true	false	false
true	true	true

在 **AND** 运算符中，两个操作数必须为 **true** 才能使整个表达式为 **true**。

✧ **AND** 运算符

例如：

```
int age = 20;
if (age > 16 && age < 60) {
    cout << "Accepted!" << endl;
}

// Outputs "Accepted"
```

- 在上面的示例中，逻辑 **AND** 运算符用于组合两个表达式。
仅当两个表达式都为 **true** 时，if 语句中的表达式才会求值为 **true**。

✧ **AND** 运算符

- 在单个 if 语句中，逻辑运算符可用于组合多个条件。

```
int age = 20;
int grade = 80;

if (age > 16 && age < 60 && grade > 50) {
    cout << "Accepted!" << endl;
}
```

仅当所有条件都为 **true** 时，整个表达式的计算结果为 **true**。

✧ OR 运算符

- 如果任何一个操作数为 **true**，则 **OR(||)**运算符返回 **true**。

Left Operand	Right Operand	Result
false	false	false
false	true	true
true	false	true
true	true	true

例：

```
int age = 16;
int score = 90;
if (age > 20 || score > 50) {
    cout << "Accepted!" << endl;
}

// Outputs "Accepted!"
```

您可以组合任意数量的逻辑 **OR** 语句。
此外，多个 **OR** 和 **AND** 语句可以组合在一起。

✧ 逻辑非

- 逻辑非(!)运算符只使用一个操作数，反转其逻辑状态。因此，如果条件为**真**，则 **NOT** 运算符使其为**假**，反之亦然。

Right Operand	Result
true	false
false	true

```
int age = 10;  
if ( !(age > 16) ) {  
    cout << "Your age is less than 16" << endl;  
}  
  
// Outputs "Your age is less than 16"
```

小心使用它，因为!false 意味着真。

3.数据类型、数组、指针

3.1.数据类型简介

◇ 数据类型

- 操作系统根据变量的**数据类型**分配内存并选择将存储在内存中的内容。
- 数据类型定义标识符的正确使用，可以存储什么类型的数据以及可以执行哪些类型的操作。

C++中有许多内置类型。

◇ 表达式

- 以下示例显示了合法和非法的 C++表达式。

```
55+15 // legal C++ expression
//Both operands of the + operator are integers

55 + "John" // illegal
// The + operator is not defined for integer and string
```

您可以通过重载运算符为非法表达式实现一些逻辑。你稍后会了解它。

◇ 数值数据类型

- 数字数据类型包括：
- **整数**（整数），例如-7,42。
- **浮点数**，例如 3.14，-42.67。

◇ 字符串和字符

- **字符串**由数字，字符或符号组成。字符串文字放在**双引号**中；一些例子是“你好”，“我的名字是大卫”，类似。

- **字符**是单个字母或符号，必须用**单引号**括起来，如“a”，“b”等。

在 C++ 中，单引号表示一个字符；双引号创建一个字符串文字。虽然 'a' 是单个字符文字，但 "a" 是字符串文字。

◇ 布尔型

- 布尔数据类型只返回两个可能的值：**true(1)**和 **false(0)**。
条件表达式是布尔数据类型的示例。

3.2. 整型、单浮点型、双浮点型

◇ 整形

- **整数**类型包含非小数，可以是正数或负数。整数的例子包括 42，-42 和类似的数字。
整数类型的大小根据程序运行系统的体系结构而变化，尽管在大多数现代系统体系结构中 4 个字节是最小大小。

◇ 整形

- 使用 **int** 关键字定义整数数据类型。

```
int a = 42;
```

- 可以使用以下一个或多个类型**修饰符**修改几种基本类型（包括整数）：
signed: 有符号整数可以包含负数和正数。
unsigned: 无符号整数只能保存正值。
short: 默认长度的一半。
long: 默认长度的两倍。

例如：

```
unsigned long int a;
```

整数数据类型保留 4-8 个字节，具体取决于操作系统。

◇ 浮点数

- 浮点类型变量可以保存实数，例如 420.0，-3.33 或 0.03325。
- 浮点指的是小数点之前和之后可以出现不同数量的数字。你可以说十进制具有“浮点”的属性。

- 有三种不同的浮点数据类型：**float**，**double** 和 **long double**。

- 在大多数现代架构中，**float** 是 4 个字节，**double** 是 8，**long double** 可以等于 **double**（8 个字节）或 16 个字节。

例如：

```
double temp = 4.21;
```

浮点数据类型始终是有符号的，这意味着它们能够同时保存正值和负值。

3.3.字符串、字符型、布尔型

◇ 字符串

- 字符串是一个有序的字符序列，用双引号括起来。
- 它是标准库的一部分。
- 您需要包含<string>库以使用字符串数据类型。或者，您可以使用包含字符串库的库。

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

int main() {
    string a = "I am learning C++";
    return 0;
}
```

<string>库包含在<iostream>库中，因此如果已经使用<iostream>，则不需要单独包含<string>。

✧ 字符

- **char** 变量保存 1 字节整数。但是，**char** 变量的值通常被解释为 ASCII 字符，而不是将 **char** 的值解释为整数。

- 单引号之间包含一个字符（例如 'a'，'b' 等）。

例如：

```
char test = 'S';
```

美国信息交换标准码(ASCII)是一种字符编码方案，用于表示计算机中的文本。

✧ 布尔

- 布尔变量只有两个可能的值：**true**(1)和 **false**(0)。

要声明一个布尔变量，我们使用关键字 **bool**。

```
bool online = false;
```

```
bool logged_in = true;
```

如果将布尔值赋给整数，则 **true** 变为 1，**false** 变为 0。

如果为布尔值分配整数值，则 0 变为 **false**，并且具有非零值的任何值都变为 **true**。

3.4.变量命名规则

✧ 变量命名规则

- 在命名变量时使用以下规则：

- 所有变量名称必须以字母或下划线（_）开头。
- 在首字母后面，变量名称可以包含其他字母和数字。变量名中不允许使用空格或特殊字符。

有两种已知的命名约定：

Pascal case: 标识符中的第一个字母和每个后续连接词的首字母大写。例如：

BackColor

Camel case: 标识符的第一个字母是小写，每个后续连接词的首字母大写。例如：

backColor

✧ 区分大小写

• C++区分大小写，这意味着以大写形式编写的标识符不等同于以小写形式具有相同名称的另一个标识符。

例如，*myvariable* 与 *MYVARIABLE* 不同，与 *MyVariable* 不同。

这是三个不同的变量。

选择建议用法的变量名称，例如：firstName，lastName。

✧ 变量命名规则

• C++关键字（保留字）不能用作变量名。

例如，**int**，**float**，**double**，**cout** 不能用作变量名。

变量名称的长度没有实际限制（取决于环境），但要尽量保持变量名称的实用性和有意义。

3.5.数组

✧ 数组

• 数组用于存储数据集合，将数组视为相同类型的变量集合将很有用。

• 您可以声明一个数组来存储所有值，而不是声明多个变量并存储单个值。

• 声明数组时，请指定其元素类型以及它将保留的元素数。

例如：

```
int a[5];
```

• 在上面的示例中，变量 **a** 被声明为五个整数型的数组[在方括号中指定]。

• 您可以通过指定它包含的值来初始化数组：

```
int b[5] = {11, 45, 62, 70, 88};
```

• 这些值以逗号分隔的列表提供，括在{花括号}中。

大括号{}之间的值的数量不得超过方括号[]中声明的元素的数量。

✧ 初始化数组

• 如果省略数组的大小，则会创建一个足以容纳初始化的数组。

例如：

```
int b[] = {11, 45, 62, 70, 88};
```

- 这将创建与上一示例中创建的数组相同的数组。
- 数组的每个元素都有一个索引，可以精确定位元素的特定位置。
- 数组的第一个元素的索引为 0，第二个元素的索引为 1。
- 因此，对于我们在上面声明的数组 b：

11	45	62	70	88
[0]	[1]	[2]	[3]	[4]

• 要访问数组元素，请通过将元素的索引放在数组名称后面的方括号中来索引数组名称。

例如：

```
int b[] = {11, 45, 62, 70, 88};

cout << b[0] << endl;
// Outputs 11

cout << b[3] << endl;
// Outputs 70
```

✧ 分配数组元素

- 索引号也可用于为元素分配新值。

```
int b[] = {11, 45, 62, 70, 88};
b[2] = 42;
```

- 这为数组的第三个元素赋值 42。

永远记住，元素列表始终以索引 0 开头。

3.6.在循环中使用数组

✧ 循环中的数组

- 偶尔需要迭代数组的元素，根据某些计算分配元素值。

通常，这是使用循环完成的。

◇ 循环中的数组

- 让我们声明一个数组，它将存储 5 个整数，并使用 **for** 循环为每个元素赋值：

```
int myArr[5];
```

```
for(int x=0; x<5; x++) {  
    myArr[x] = 42;  
}
```

- 数组中的每个元素都赋值为 42。
- **for** 循环中的 **x** 变量用作数组的索引。

数组中的最后一个索引是 4，因此 **for** 循环条件是 $x < 5$ 。

◇ 循环中的数组

- 让我们输出数组中的每个索引和相应的值。

```
int myArr[5];
```

```
for(int x=0; x<5; x++) {  
    myArr[x] = 42;  
  
    cout << x << ": " << myArr[x] << endl;  
}
```

```
/* Outputs
```

```
0: 42
```

```
1: 42
```

```
2: 42
```

```
3: 42
```

```
4: 42
```

```
*/
```

3.7.计算中的数组

◇ 计算中的数组

- 以下代码创建一个程序，该程序使用 **for** 循环来计算数组中所有元素的总和。

```
int arr[] = {11, 35, 62, 555, 989};
```

```
int sum = 0;
```

```
for (int x = 0; x < 5; x++) {
    sum += arr[x];
}

cout << sum << endl;
//Outputs 1652
```

- 为了回顾，我们声明了一个数组和一个变量 **sum**，它将保存元素的总和。
- 接下来，我们使用 **for** 循环遍历数组的每个元素，并将相应元素的值添加到 **sum** 变量中。

在数组中，第一个元素的索引是 0，因此 **for** 循环将 **x** 变量初始化为 0。

3.8.多维数组

✧ 多维数组

- 多维数组包含一个或多个数组。声明一个多维数组，如下所示。

```
type name[size1][size2]...[sizeN];
```

- 在这里，我们创建了一个二维 3x4 整数数组：

```
int x[3][4];
```

将此数组可视化为由 3 行和 4 列组成的表。

	Column 1	Column 2	Column 3	Column 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

记住元素计数总是从 0 开始。

✧ 二维数组

- 可以通过指定每行的括号值来初始化多维数组。
- 以下是一个包含 2 行和 3 列的数组：

```
int x[2][3] = {  
    {2, 3, 4}, // 1st row  
    {8, 9, 10} // 2nd row  
};
```

- 您也可以只使用一行编写相同的初始化。

```
int x[2][3] = {{2, 3, 4}, {8, 9, 10}};
```

- 通过使用数组的行索引和列索引来访问元素。

例如：

```
int x[2][3] = {{2, 3, 4}, {8, 9, 10}};  
cout << x[0][2] << endl;  
  
//Outputs 4
```

第一个索引 0 指的是第一行。第二个索引 2 指的是第一行的第 3 个元素，即 4。

✧ 多维数组

- 数组可以包含无限数量的维度。

```
string threeD[42][8][3];
```

- 上面的例子声明了一个三维的字符串数组。正如我们之前所做的那样，可以使用索引号来访问和修改元素。

超过三维的数组更难管理。

3.9. 指针简介

✧ 指针

- 每个变量都是一个内存位置，其地址已定义。
- 可以使用符号(&)运算符（也称为地址运算符）来访问该地址，该运算符表示内存中的地址。

例如：

```
int score = 5;
cout << &score << endl;

//Outputs "0x29fee8"
```

这输出存储器地址，存储变量得分。

✧ 指针

- 指针是一个变量，另一个变量的地址作为其值。
- 在 C++ 中，指针有助于使某些任务更容易执行。如果不使用指针，则无法执行其他任务（如动态内存分配）。

- 所有指针共享相同的数据类型 - 表示内存地址的长十六进制数。

不同数据类型的指针之间的唯一区别是指针指向的变量的数据类型。

✧ 指针

- 指针是一个变量，与任何其他变量一样，必须在使用它之前声明它。
- 星号用于声明指针（与用于乘法的星号相同），但是，在此语句中，星号用于将变量指定为指针。
- 以下是有效的指针声明：

```
int *ip; // pointer to an integer
double *dp; // pointer to a double
float *fp; // pointer to a float
char *ch; // pointer to a character
```

- 就像变量一样，我们给指针指定一个名称并定义指针所指向的类型。

星号可以放在数据类型或变量名称旁边，也可以放在中间。

✧ 使用指针

- 在这里，我们将变量的地址分配给指针。

```
int score = 5;
int *scorePtr;
scorePtr = &score;

cout << scorePtr << endl;
```

```
//Outputs "0x29fee8"
```

- 上面的代码声明了一个指向一个名为 **scorePtr** 的整数的指针，并使用**&**符号 (address-of)运算符为其分配 **score** 变量的内存位置。

现在，**scorePtr** 的值是 **score** 的内存位置。

3.10.更多关于指针

◇ 指针操作

指针有两个运算符：

取地址运算符(&)：返回其操作数的内存地址。

取内容（或解引用）运算符(*)：返回位于其操作数指定的地址的变量的值。

例如：

```
int var = 50;
int *p;
p = &var;

cout << var << endl;
// Outputs 50 (the value of var)

cout << p << endl;
// Outputs 0x29fee8 (var's memory location)

cout << *p << endl;
/* Outputs 50 (the value of the variable
   stored in the pointer p) */
```

星号(*)用于声明指针，仅用于指示它是指针（星号是其类型复合说明符的一部分）。不要将此与 **dereference** 运算符混淆，后者用于获取位于指定地址的值。它们只是用相同符号表示的两种不同的东西。

◇ 取消引用

- **解引用运算符(*)**基本上是指针指向的变量的别名。

例如：

```
int x = 5;
int *p = &x;
```

```
x = x + 4;  
x = *p + 4;  
*p = *p + 4;
```

• 前面的三个语句都是等价的，并返回相同的结果。我们可以通过取消引用变量的指针来访问变量。

当 **p** 指向变量 **x** 时，解引用指针(***p**)表示与变量 **x** 完全相同。

3.11.动态内存

✧ 静态与动态内存

- 要想成为 C++ 程序员，必须要充分了解**动态内存**的工作原理。
- 在 C++ 程序中，内存分为两部分：

堆栈：所有局部变量都从堆栈中占用内存。

堆：程序运行时可以使用的未用程序存储器，用于**动态**分配存储器。

• 很多时候，您事先并不知道将特定信息存储在已定义变量中需要多少内存，并且可以在运行时确定所需内存的大小。

• 您可以使用 **new** 运算符运行时为**堆**内给定类型的变量分配内存，该运算符返回分配的空间的地址。

```
new int;
```

这将分配在**堆**上存储**整数**所需的内存大小，并返回该地址。

✧ 动态内存

- 分配的地址可以存储在**指针**中，然后可以取消引用该指针以访问变量。

例：

```
int *p = new int;  
*p = 5;
```

- 我们为整数动态分配了内存，并为其赋值为 5。

指针 **p** 作为局部变量存储在**堆栈**中，并将**堆**的分配地址保存为其值。值 **5** 存储在堆中的该地址。

◇ 动态内存

- 对于堆栈中的局部变量，自动执行内存管理。
- 在堆上，需要手动处理动态分配的内存，并在不再需要时使用 **delete** 运算符释放内存。

```
delete pointer;
```

此语句释放指针指向的内存。

例如：

```
int *p = new int; // request memory
*p = 5; // store value

cout << *p << endl; // use value

delete p; // free up the memory
```

忘记释放已使用 **new** 关键字分配的内存将导致内存泄漏，因为该内存将保持分配状态，直到程序关闭为止。

◇ 悬空指针

- **delete** 运算符释放为变量分配的内存，但不会删除指针本身，因为指针存储在堆栈中。

- 指向不存在的内存位置的指针称为**悬空指针**。

例如：

```
int *p = new int; // request memory
*p = 5; // store value

delete p; // free up the memory
// now p is a dangling pointer

p = new int; // reuse for a new address
```

- **NULL** 指针是一个值为零的常量，它在几个标准库中定义，包括 **iostream**。
- 如果您没有要分配的确切地址，那么在声明指针变量时将它指定给指针变量是一个好习惯。指定为 **NULL** 的指针称为**空指针**。例如：`int * ptr = NULL;`

◇ 动态内存

- 动态内存也可以分配给数组。

例如：

```
int *p = NULL; // Pointer initialized with null
p = new int[20]; // Request memory
delete [] p; // Delete array pointed to by p
```

- 请注意语法中的**括号**。

动态内存分配在许多情况下都很有用，例如当程序依赖于输入时。例如，当您的程序需要读取图像文件时，它不会事先知道图像文件的大小和存储图像所需的内存。

3.12.sizeof()运算符

◇ sizeof

- 虽然为不同数据类型分配的大小取决于用于运行程序的计算机的体系结构，但C++确实保证了基本数据类型的最小大小：

Category	Type	Minimum Size
boolean	bool	1 byte
character	char	1 byte
integer	short	2 bytes
	int	2 bytes
	long	4 bytes
	long long	8 bytes
floating point	float	4 bytes
	double	8 bytes
	long double	8 bytes

sizeof 运算符可用于获取变量或数据类型的大小（以字节为单位）。

语法：

sizeof (data type)

sizeof 运算符确定并返回类型或变量的大小（以字节为单位）。

例如：

```
cout << "char: " << sizeof(char) << endl;
cout << "int: " << sizeof(int) << endl;
cout << "float: " << sizeof(float) << endl;
cout << "double: " << sizeof(double) << endl;
int var = 50;
cout << "var: " << sizeof(var) << endl;

/* Outputs
char: 1
int: 4
float: 4
double: 8
var: 4
*/
```

根据使用的计算机和编译器，输出值可能会有所不同。

◇ 数组的大小

- C++ **sizeof** 运算符还用于确定**数组**的大小。

例如：

```
double myArr[10];
cout << sizeof(myArr) << endl;

//Outputs 80
```

- 在我们的机器上，**double** 占用 8 个字节。该阵列存储 10 个双精度数，因此整个数组在内存中占用 80(8 * 10)个字节。

- 此外，将**数组**中的总字节数除以单个元素中的字节数，以了解数组中有多少个元素。

例如：

```
int numbers[100];
cout << sizeof(numbers) / sizeof(numbers[0]);

// Outputs 100
```

4.函数

4.1.函数简介

✧ 函数

- **函数**是执行特定任务的一组语句。
- 您可以在 C++ 中定义自己的函数。
- 使用函数可以有許多优点，包括：
 - 您可以在函数中重用代码。
 - 您可以轻松测试各个函数。
 - 如果需要进行任何代码修改，您可以在单个函数中进行修改，而无需更改程序结构。
 - 您可以对不同的输入使用相同的函数。

每个合法的 C++ 程序至少有一个函数 - **main()** 函数。

✧ 返回类型

- 主要功能采用以下一般形式：

```
int main()
{
    // some code
    return 0;
}
```

- 函数的**返回类型**在其名称之前声明。在上面的示例中，返回类型为 **int**，表示函数返回整数值。
- 有时，函数将执行所需的操作而不返回值。这些函数使用关键字 **void** 定义。**void** 是定义无值状态的基本数据类型。

✧ 定义一个函数

- 使用以下语法定义 C++ 函数：

```
return_type function_name( parameter list )
{
    body of the function
}
```



```
}
```

返回类型：函数返回的值的数据类型。

函数名称：函数的名称。

参数：调用函数时，将值传递给参数。该值称为实际参数或参数。参数列表是指函数参数的类型，顺序和数量。

函数体：定义函数功能的语句集合。

参数是可选的；也就是说，你可以拥有一个没有参数的函数。

✧ 定义一个函数

- 例如，让我们定义一个不返回值的函数，只打印一行文本到屏幕。

```
void printSomething()
{
    cout << "Hi there!";
}
```

- 我们的函数名为 **printSomething**，返回 **void**，没有参数。
- 现在，我们可以在 **main()**中使用我们的函数。

```
int main()
{
    printSomething();

    return 0;
}
```

要调用函数，只需要传递必需的参数和函数名称。

✧ 函数

- 您必须在调用之前声明一个函数。

例如：

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

void printSomething() {
    cout << "Hi there!";
}

int main() {
    printSomething();
}
```

```
return 0;  
}
```

在 **main()** 函数之后放置声明会导致错误。

✧ 函数

• 函数**声明**或**函数原型**告诉编译器函数名称以及如何调用函数。函数的实际主体可以单独定义。

例如：

```
#include <iostream>  
using namespace std;  
  
//Function declaration  
void printSomething();  
  
int main() {  
    printSomething();  
  
    return 0;  
}  
  
//Function definition  
void printSomething() {  
    cout << "Hi there!";  
}
```

在一个源文件中定义函数并在另一个文件中调用该函数时，需要函数声明。 这种情况下，您应该在调用该函数的文件顶部声明该函数。

4.2.函数参数

✧ 函数

• 对于使用**参数**的函数，它必须声明形式**参数**，这些参数是接受参数值的变量。

例如：

```
void printSomething(int x)  
{  
    cout << x;
```

```
}
```

- 这定义了一个函数，它接受一个**整数**参数并打印其值。

形式参数在函数内的行为与其他局部变量类似。它们是在进入函数时创建的，并在退出函数时被销毁。

✧ 函数参数

- 一旦定义了参数，就可以在调用函数时传递相应的参数。

例如：

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

void printSomething(int x) {
    cout << x;
}

int main() {
    printSomething(42);
}

// Outputs 42
```

- 42 作为**参数**传递给函数，并分配给函数的形式**参数**：**x**。

在函数内更改参数不会改变参数。

✧ 函数参数

- 您可以将不同的参数传递给同一个函数。

例如：

```
int timesTwo(int x) {
    return x*2;
}
```

- 上面定义的函数接受一个整数参数并返回其乘以 2 的值。

- 我们现在可以使用具有不同参数的函数。

```
int main() {
    cout << timesTwo(8);
    // Outputs 16
}
```

```

cout << timesTwo(5);
// Outputs 10

cout << timesTwo(42);
// Outputs 84
}

```

4.3. 多参数函数

✧ 多参数

- 您可以通过用逗号分隔它们来为函数定义任意数量的参数。
- 让我们创建一个返回两个参数之和的简单函数。

```

int addNumbers(int x, int y) {
    // code goes here
}

```

- 根据定义，**addNumbers** 函数接受两个 **int** 类型的参数，并返回 **int**。
应为每个参数定义数据类型和名称。

✧ 多参数

- 现在让我们计算两个参数的总和并返回结果：

```

int addNumbers(int x, int y) {
    int result = x + y;
    return result;
}

```

✧ 多参数

- 现在我们可以调用该函数。

```

int addNumbers(int x, int y) {
    int result = x + y;
    return result;
}

```

```
int main() {
    cout << addNumbers(50, 25);
    // Outputs 75
}
```

- 您还可以将返回的值分配给变量。

```
int main() {
    int x = addNumbers(35, 7);
    cout << x;
    // Outputs 42
}
```

✧ 多参数

- 您可以根据需要为单个函数添加任意数量的参数。

例如：

```
int addNumbers(int x, int y, int z, int a) {
    int result = x + y + z + a;
    return result;
}
```

如果您有多个参数，请记住在声明它们和传递参数时用逗号分隔它们。

4.4.rand()函数

✧ 随机数

• 生成**随机数**在许多情况下都很有用，包括创建游戏，统计建模程序和类似的最终产品。

• 在 C++ 标准库中，您可以访问称为 **rand()** 的伪随机数生成器函数。使用时，我们需要包含头 **<cstdlib>**。

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

int main() {
    cout << rand();
}
```

这将输出一个随机数。

✧ 随机数

- **for** 循环可用于生成多个随机数。

```
int main() {  
    for (int x = 1; x <= 10; x++) {  
        cout << rand() << endl;  
    }  
}
```

/* Output:

41

18467

6334

26500

19169

15724

11478

29358

26962

24464

*/

✧ 随机数

- 使用 **modulo(%)** 运算符生成特定范围内的随机数。
- 以下示例生成 1 到 6 范围内的整数。

```
int main () {  
    for (int x = 1; x <= 10; x++) {  
        cout << 1 + (rand() % 6) << endl;  
    }  
}
```

/* Output:

6

6

5

5

6

5

```
1
1
5
3
*/
```

但是，**rand()**函数只返回一个伪随机数。这意味着每次运行代码时，它都会生成相同的数字。

✧ srand()函数

- **srand()**函数用于生成真正的随机数。
- 此函数允许将种子值指定为其参数，该参数用于 **rand()**函数的算法。

```
int main () {
    srand(98);

    for (int x = 1; x <= 10; x++) {
        cout << 1 + (rand() % 6) << endl;
    }
}
```

更改种子值会更改 **rand()**的返回值。但是，相同的参数将导致相同的输出。

✧ 真正的随机数

- 生成真正随机数的解决方案是使用当前时间作为 **srand()**函数的种子值。
- 此示例使用 **time()**函数获取系统时间的秒数，并随机播种 **rand()**函数（我们需要包含**<ctime>**函数头）：

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

int main () {
    srand(time(0));

    for (int x = 1; x <= 10; x++) {
        cout << 1 + (rand() % 6) << endl;
    }
}
```

- **time(0)**将返回当前的第二个计数，提示 **srand()**函数在每次程序运行时为 **rand()**函数设置不同的种子。

每次运行程序时，使用此种子值将创建不同的输出。

4.5.默认参数

✧ 参数默认值

- 定义函数时，可以为每个最后一个参数指定**默认值**。如果在调用函数时缺少相应的参数，则使用默认值。

- 为此，请使用赋值运算符将值分配给函数定义中的参数，如此示例所示。

```
int sum(int a, int b=42) {  
    int result = a + b;  
    return (result);  
}
```

- 这将为 **b** 参数指定默认值 **42**。如果我们在不传递 **b** 参数值的情况下调用函数，则将使用默认值。

```
int main() {  
    int x = 24;  
    int y = 36;  
  
    //calling the function with both parameters  
    int result = sum(x, y);  
    cout << result << endl;  
    //Outputs 60  
  
    //calling the function without b  
    result = sum(x);  
    cout << result << endl;  
    //Outputs 66  
  
    return 0;  
}
```

对函数的第二次调用不传递第二个参数的值，而是使用默认值 **42**。

◇ 使用默认参数

另一个例子：

```
int volume(int l=1, int w=1, int h=1) {
    return l*w*h;
}

int main() {
    cout << volume() << endl;
    cout << volume(5) << endl;
    cout << volume(2, 3) << endl;
    cout << volume(3, 7, 6) << endl;
}

/* Output
1
5
6
126
*/
```

如您所见，默认参数值可用于在不同情况下（一个或多个参数没有被使用时）调用相同的函数，。

4.6.函数重载

◇ 重载

- 函数**重载**允许创建具有**相同名称**的多个函数，只要它们具有不同的参数即可。
- 例如，您可能需要 **printNumber()**函数来打印其参数的值。

```
void printNumber(int a) {
    cout << a;
}
```

- 这仅对**整型**参数有效。重载它将使其可用于其他类型，例如 **floats**。

```
void printNumber(float a) {
    cout << a;
}
```

现在，相同的 **printNumber()** 函数名称将适用于整数和浮点数。

✧ 函数重载

- 当重载函数时，函数的定义必须区别于参数列表中的（参数类型/数量）。

例如：

```
void printNumber(int x) {  
    cout << "Prints an integer: " << x << endl;  
}  
void printNumber(float x) {  
    cout << "Prints a float: " << x << endl;  
}  
int main() {  
    int a = 16;  
    float b = 54.541;  
    printNumber(a);  
    printNumber(b);  
}  
  
/* Output:  
Prints an integer: 16  
Prints a float: 54.541  
*/
```

如您所见，函数调用基于提供的参数。**整形**参数将采用带有**整形**参数的函数。**float**参数将调用采用 **float** 参数的函数。

✧ 函数重载

- 您**不能**重载仅由返回类型不同的**函数**声明。

以下声明会导致错误。

```
int printName(int a) {}  
float printName(int b) {}  
double printName(int c) {}
```

虽然每个函数使用相同的名称，但唯一的区别是**返回类型**，这是不允许的。

4.7.递归

✧ 递归

- C++中的**递归函数**是一个调用自身的函数。

为避免无限期地运行递归，必须包含终止条件。

✧ 递归

- 为了演示递归，让我们创建一个程序来计算一个数字的**阶乘**。
- 在数学中，术语阶乘是指所有小于或等于特定的非负整数(**n**)的乘积。**n** 的阶乘表示为 **n!**

例如：

$$4! = 4 * 3 * 2 * 1 = 24$$

递归是一种解决相同问题而对象为较小实例的方法。

✧ 递归

- 让我们定义我们的函数：

```
int factorial(int n) {  
    if (n==1) {  
        return 1;  
    }  
    else {  
        return n * factorial(n-1);  
    }  
}
```

- if 语句定义退出条件。在这种情况下，当 **n** 等于 **1** 时，返回 **1**(**1** 的阶乘为 **1**)。
- 我们在 **else** 语句中放置了递归函数调用，它返回 **n** 乘以 **n-1** 的阶乘。
- 例如，如果使用参数 **4** 调用阶乘函数，它将按如下方式执行：
返回 **4 * factorial(3)**，即 **4 * 3 * factorial(2)**，即 **4 * 3 * 2 * factorial(1)**，即 **4 * 3 * 2 * 1**。

factorial 函数调用自身，然后继续这样做，直到参数等于 **1**。

✧ 递归

- 我们现在处于可以调用阶乘函数的位置。

```
int factorial(int n) {
    if (n==1) {
        return 1;
    }
    else {
        return n * factorial(n-1);
    }
}
int main() {
    cout << factorial(5);
}

//Outputs 120
```

- 退出条件的另一个名称是基本情况。

请记住，基本情况对于实际递归是必需的。没有它，递归将永远运行。

4.8.传递数组给函数

✧ 数组和函数

- 数组也可以作为参数传递给函数。
- 在声明函数时，应使用方括号将参数定义为数组。

例如：

```
void printArray(int arr[], int size) {
    for(int x=0; x<size; x++) {
        cout << arr[x];
    }
}
```

✧ 数组和函数

- 我们可以在 main()中使用我们的函数，并用示例数组调用它：

```
void printArray(int arr[], int size) {
```

```

    for(int x=0; x<size; x++) {
        cout <<arr[x]<< endl;
    }
}
int main() {
    int myArr[3]= {42, 33, 88};
    printArray(myArr, 3);
}

```

- **printArray** 函数将数组作为参数(int arr []), 并使用 **for** 循环遍历数组。
- 我们在 main()中调用该函数, 我们将 **myArr** 数组传递给函数, 该函数打印其元素。

将它作为参数传递给函数时, 请记住指定**不带方括号的数组名称**。

4.9.用指针传递引用

✧ 函数参数

- 在调用函数时, 有两种方法可以将参数传递给函数。

按值: 此方法将参数的实际值复制到函数的形式参数中。在这里, 我们可以对函数中的参数进行更改, 而不会对参数产生任何影响。

通过引用: 此方法将参数的引用复制到形式参数中。在函数内, 引用用于访问调用中使用的实际参数。这意味着对参数所做的任何更改都会影响参数。

默认情况下, C++使用 **call 按值**来传递参数。

✧ 按值传递

- 默认情况下, C++中的参数**按值**传递。
- 当按值传递时, 参数的副本将传递给函数。

例:

```

void myFunc(int x) {
    x = 100;
}

int main() {
    int var = 20;
}

```

```
myFunc(var);
cout << var;
}
// Outputs 20
```

因为参数的副本传递给函数，所以函数不会修改原始参数。

✧ 按引用传递

- **传递引用**将参数的地址复制到形式参数中。在函数内部，该地址用于访问调用中使用的实际参数。这意味着对参数所做的更改会影响参数。
- 要通过引用传递值，参数**指针**将像任何其他值一样传递给函数。

```
void myFunc(int *x) {
    *x = 100;
}

int main() {
    int var = 20;
    myFunc(&var);
    cout << var;
}
// Outputs 100
```

- 如您所见，我们使用 **address-of 运算符&**将变量直接传递给函数。
 - 函数声明表示该函数将指针作为其参数（使用***运算符**定义）。
- 结果，该函数实际上已经改变了参数的值，因为它通过指针访问它。

✧ 总结

- **按值传递**：此方法将参数的实际值复制到函数的形式参数中。在这种情况下，对函数内部参数所做的更改不会对参数产生影响。
 - **引用传递**：此方法将参数的引用复制到形式参数中。在函数内部，引用用于访问调用中使用的实际参数。因此，对参数所做的更改也会影响参数。
- 通常，通过值传递更快更有效。当您的函数需要修改参数时，或者当您需要传递使用大量内存并且复制成本较大的数据类型时，通过引用传递。

5.类和对象

5.1.什么是对象

✧ 什么是对象

- 面向对象编程是一种编程风格，旨在使编程更接近于思考现实世界。
- 在编程中，**对象**是独立的单元，每个对象都有自己的**标识**，就像现实世界中的对象一样。
苹果是一个对象；一个杯子也是。每个人都有其独特的**身份**。可能有两个看起来完全相同的杯子，但它们仍然是独立的，独特的物体。

✧ 对象

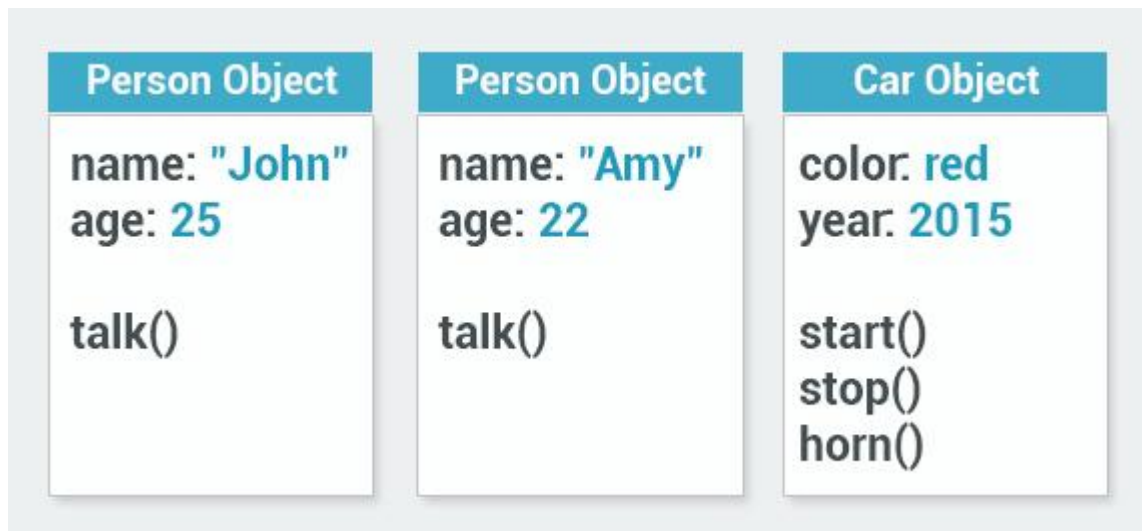
- 对象可能包含其他对象，但它们仍然是不同的对象。
 - 对象还具有用于描述它们的**特征**。例如，汽车可以是红色或蓝色，杯子可以是满的或空的，等等。这些特征也称为**属性**。属性描述对象的当前状态。
 - 对象可以有多个属性（杯子可以是**空的**，**红色**和**大的**）。
- 对象的**状态**与其类型无关；一个杯子可能装满水，另一个可能是空的。

✧ 对象

- 在现实世界中，每个对象都以自己的方式**运行**。汽车**移动**，电话**响了**，等等。这同样适用于对象 - 行为特定于对象的类型。
- 因此，以下三个维度描述了面向对象编程中的任何对象：**身份**，**属性**，**行为**。

✧ 对象

- 在编程中，对象是**自包含**的，具有自己的**标识**。它与其他对象分开。
- 每个对象都有自己的**属性**，用于描述其当前状态。每个人都表现出自己的**行为**，这表明他们可以做些什么。



- 在计算时，对象并不总是代表实物个体。
- 例如，编程对象可以表示日期，时间，银行账户。银行账户不是有形的；你无法看到或触摸它，但它仍然是一个定义良好的对象 - 它有自己的**标识**，**属性**和**行为**。

5.2.什么是类

✧ 什么是类

- 使用**类**创建对象，这实际上是 OOP 的焦点。
- 该类**描述**了对象的内容，但与对象本身是分开的。
- 换句话说，类可以被描述为对象的**蓝图**，描述或定义。
- 您可以使用相同的类作为蓝图来创建多个不同的对象。例如，在准备创建新建筑时，建筑师会创建一个蓝图，用作实际构建结构的基础。同样的蓝图可用于创建多个建筑物。
- 编程以相同的方式工作。我们首先定义一个类，它成为创建对象的蓝图。
- 每个类都有一个**名称**，并描述**属性**和**行为**。

在编程中，术语**类型**用于指代类名：我们正在创建特定**类型**的对象。
属性也称为 **properties** 或 **data**。

✧ 方法

- 方法是类行为的另一个术语。方法基本上是属于类的**函数**。

方法类似于函数 - 它们是被调用的代码块，它们还可以执行操作和返回值。

✧ 一个类的例子

- 例如，如果我们正在创建银行业务计划，我们可以为我们的班级提供以下特征：

name: BankAccount

attributes: accountNumber, balance, dateOpened

behavior: open(), close(), deposit()

- 该类指定每个对象应具有已定义的属性和行为。但是，它没有指定实际数据是什么；它只提供一个**定义**。

- 一旦我们编写了类，我们就可以继续创建基于该类的对象。
- 每个对象称为一个类的**实例**。创建对象的过程称为**实例化**。

每个对象都有自己的标识，数据和行为。

5.3.类的例子

✧ 声明一个类

- 使用关键字 **class** 开始您的类定义。使用包含在一组花括号中的类名和类主体来跟随关键字。
- 以下代码声明了一个名为 **BankAccount** 的类：

```
class BankAccount {  
  
};
```

类定义必须后跟分号。

✧ 声明一个类

- 在花括号内定义类主体中的所有**属性和行为**（或成员）。
- 您还可以为类的成员定义**访问说明符**。

- 使用 **public** 关键字定义的成员可以从类外部访问，只要它在类对象范围内的任何位置即可。

您还可以将类的成员指定为 **private** 或 **protected**。这将在本课程后面更详细地讨论。

◇ 创建一个类

- 让我们用一个公共方法创建一个类，然后打印出“Hi”。

```
class BankAccount {  
    public:  
        void sayHi() {  
            cout << "Hi" << endl;  
        }  
};
```

- 下一步是实例化 **BankAccount** 类的对象，就像我们定义一个类型的变量一样，区别在于我们的对象的类型是 **BankAccount**。

```
int main()  
{  
    BankAccount test;  
    test.sayHi();  
}
```

- 我们的名为 **test** 的对象定义了该类的所有成员。

- 注意用于访问和调用对象方法的点分隔符(.)。

我们必须在使用它之前声明一个类，就像我们使用函数一样。

5.4.抽象化

◇ 抽象化

- 数据**抽象**是仅向外界提供基本信息的概念。这是一个表示基本功能而不包含实现细节的过程。

- 一个好的现实世界的例子是一本书：当你听到术语书时，你不知道确切的细节，即：页数，颜色，大小，但你理解书的概念 - 书的抽象。

抽象的概念是我们关注其品质，而不是一个特定例子的具体特征。

✧ 抽象化

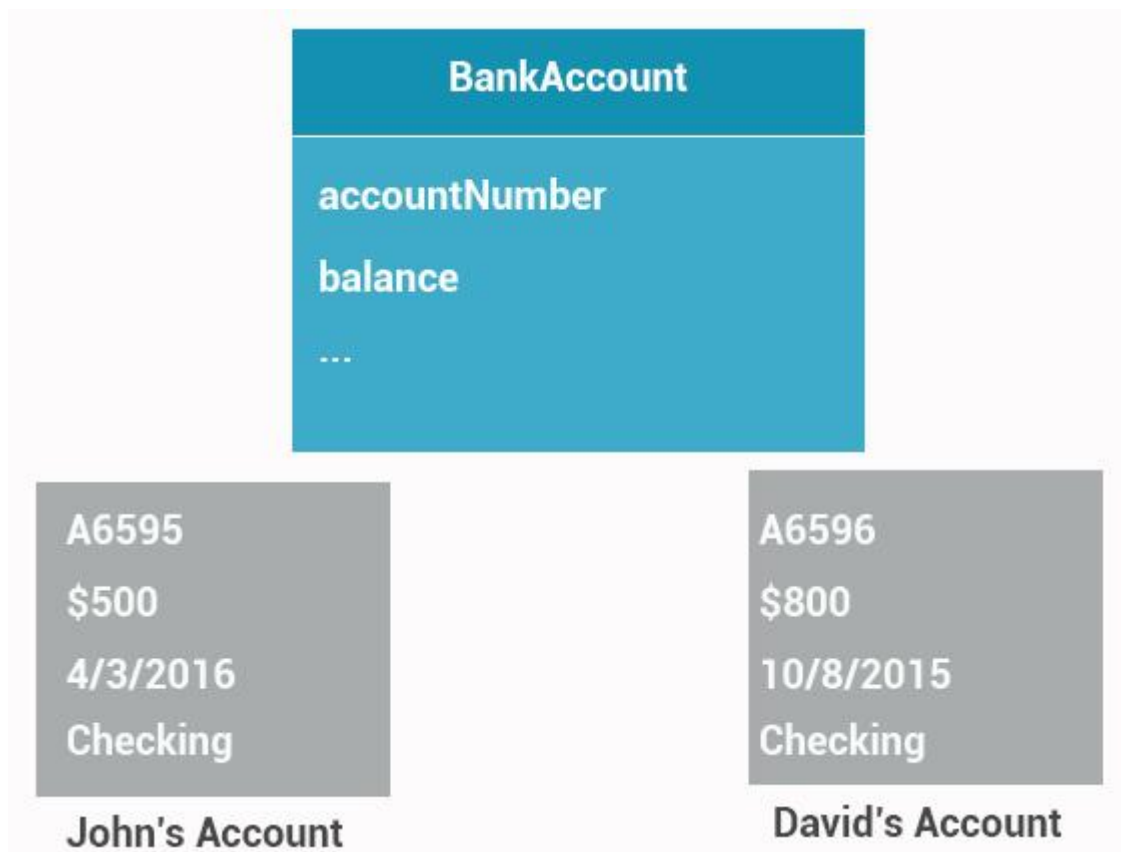
- **抽象**意味着，我们可以拥有一个与任何特定实例完全分离的想法或概念。
- 它是面向对象编程的基本构建块之一。
- 例如，当您使用 **cout** 时，您实际上正在使用类 **ostream** 的 **cout** 对象。 这会将数据流式传输到标准输出。

```
cout << "Hello!" << endl;
```

在此示例中，无需了解 **cout** 将如何在用户屏幕上显示文本。您需要知道的唯一能够使用它的是公共接口。

✧ 抽象化

- 抽象允许我们编写单个银行帐户类，然后根据类创建不同的对象，用于单个银行帐户，而不是为每个银行帐户创建单独的类。



抽象是其他面向对象基础的基础，例如**继承**和**多态**。这些将在后面的课程中讨论。

5.5.封装

✧ 封装

- “封装”这个词的部分含义是“围绕”一个实体的想法，不仅仅是为了保持内在的东西，而且还要**保护**它。
- 在面向对象中，封装不仅仅意味着在类中简单地将属性和行为组合在一起；它还意味着限制对该类内部工作的访问。

- 这里的关键原则是对象仅显示其他应用程序组件有效运行应用程序所需的内容。其他一切都被排除在外。

这称为**数据隐藏**。

✧ 封装

- 例如，如果我们使用 **BankAccount** 类，我们不希望程序的其他部分进入并更改任何对象的 **balance**，而不通过 **deposit()**或 **withdraw()**行为。
- 我们应该**隐藏**该属性，控制对它的访问，因此只能由对象本身访问。
- 这样，**balance** 不能直接从对象外部更改，只能使用其方法访问。
- 这也被称为“**黑箱**”，它指的是关闭对象的内部工作区域，除了我们想要公开的部分。
- 这允许我们在不改变整个程序的情况下改变方法的属性和实现。例如，我们可以稍后返回并更改 **balance** 属性的数据类型。

总之，封装的好处是：

- 控制访问或修改数据的方式。
- 代码更灵活，更易于根据新要求进行修改。
- 更改代码的一部分而不影响代码的其他部分。

5.6.封装的例子

✧ 访问说明符

- 访问说明符用于设置对类的特定成员的访问级别。
- 访问说明符的三个级别是 **public**，**protected** 和 **private**。
- **public** 成员可以从类外部访问，也可以在类对象范围内的任何位置访问。

例如：

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

class myClass {
    public:
        string name;
};

int main() {
    myClass myObj;
    myObj.name = "SoloLearn";
    cout << myObj.name;
    return 0;
}

//Outputs "SoloLearn"
```

name 属性是 **public**；它可以从代码外部访问和修改。

访问修饰符只需要声明一次；多个成员可以遵循单个访问修饰符。

请注意 **public** 关键字后面的冒号(:)。

✧ Private

- **私有**成员无法从课外访问，甚至无法查看；它只能在课堂上被访问。
- 可以使用**公有**成员函数来访问**私有**成员。例如：

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

class myClass {
    public:
        void setName(string x) {
            name = x;
        }
    private:
        string name;
};

int main() {
```

```
myClass myObj;
myObj.setName("John");

return 0;
}
```

- **name** 属性是私有的，无法从外部访问。
- **public setName()**方法用于设置 **name** 属性。

如果未定义访问说明符，则默认情况下，类的所有成员都设置为 **private**。

✧ 访问说明符

- 我们可以添加另一个公共方法以获取属性的值。

```
class myClass {
public:
    void setName(string x) {
        name = x;
    }
    string getName() {
        return name;
    }
private:
    string name;
};
```

- **getName()**方法返回私有名称属性的值。

✧ 访问说明符

- 把它们放在一起：

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

class myClass {
public:
    void setName(string x) {
        name = x;
    }
    string getName() {
        return name;
    }
private:
    string name;
};
```

```

    }
    private:
        string name;
};

int main() {
    myClass myObj;
    myObj.setName("John");
    cout << myObj.getName();

    return 0;
}

//Outputs "John"

```

- 我们使用封装来隐藏外部代码中的 **name** 属性。然后我们使用公共方法提供对它的访问。我们的类数据只能通过这些方法读取和修改。
这允许更改方法和属性的实现，而不会影响外部代码。

5.7.构造函数

✧ 构造函数

- 类**构造函数**是类的特殊成员函数。只要在该类中创建新对象，就会执行它们。
- 构造函数的名称与类的名称相同。它没有返回类型，甚至无效。

例如：

```

class myClass {
    public:
        myClass() {
            cout << "Hey";
        }
        void setName(string x) {
            name = x;
        }
        string getName() {
            return name;
        }
    private:

```

```

    string name;
};

int main() {
    myClass myObj;

    return 0;
}

//Outputs "Hey"

```

现在，在创建 **myClass** 类型的对象时，会自动调用构造函数。

✧ 构造函数

- 构造函数对于为某些成员变量设置初始值非常有用。
- 默认构造函数没有参数。但是，在需要时，可以将参数添加到构造函数中。这使得可以在创建对象时为其分配初始值，如以下示例所示：

```

class myClass {
public:
    myClass(string nm) {
        setName(nm);
    }
    void setName(string x) {
        name = x;
    }
    string getName() {
        return name;
    }
private:
    string name;
};

```

我们定义了一个构造函数，它使用一个参数，并使用 **setName()** 方法将其赋值给 **name** 属性。

✧ 构造函数

- 在创建对象时，您现在需要传递构造函数的参数，就像调用函数时一样：

```

class myClass {

```



```

public:
    myClass(string nm) {
        setName(nm);
    }
    void setName(string x) {
        name = x;
    }
    string getName() {
        return name;
    }
private:
    string name;
};

int main() {
    myClass ob1("David");
    myClass ob2("Amy");
    cout << ob1.getName();
}
//Outputs "David"

```

- 我们定义了两个对象，并使用构造函数为每个对象传递 **name** 属性的初始值。
有多个构造函数可以使用不同数量的参数。

6.更多关于类

6.1.单独类文件

✧ 创建一个新的类

- 在单独的文件中定义新的类通常是一个好的习惯。这使得维护和读取代码变得更容易。

- 要做到这一点，在代码块中使用以下步骤：

- 单击 **File->New->Class...**

给新类起一个名字，取消选中“Has destructor”并选中“Header and implementation files shall be in same folder”，然后单击“**Create**”按钮。

Class definition

Class name:

Arguments:

☐ Has destructor ☐ Has copy ctor

☒ Virtual destructor ☐ Has assignment op.

Inheritance

☐ Inherits another class

Ancestor:

Ancestor's include filename:

Scope:

Member variables

Add new:

☒ Add "Getter" method

☒ Add "Setter" method

☒ Remove prefix:

Documentation

☐ Add documentation where appropriate

File policy

☒ Add paths to project ☐ Use relative path

☒ Header and implementation file shall be in same folder

Folder:

☐ Header and implementation file shall always be lower case

Header file

Folder:

Filename:

☒ Add guard block in header file

Guard block:

Implementation file

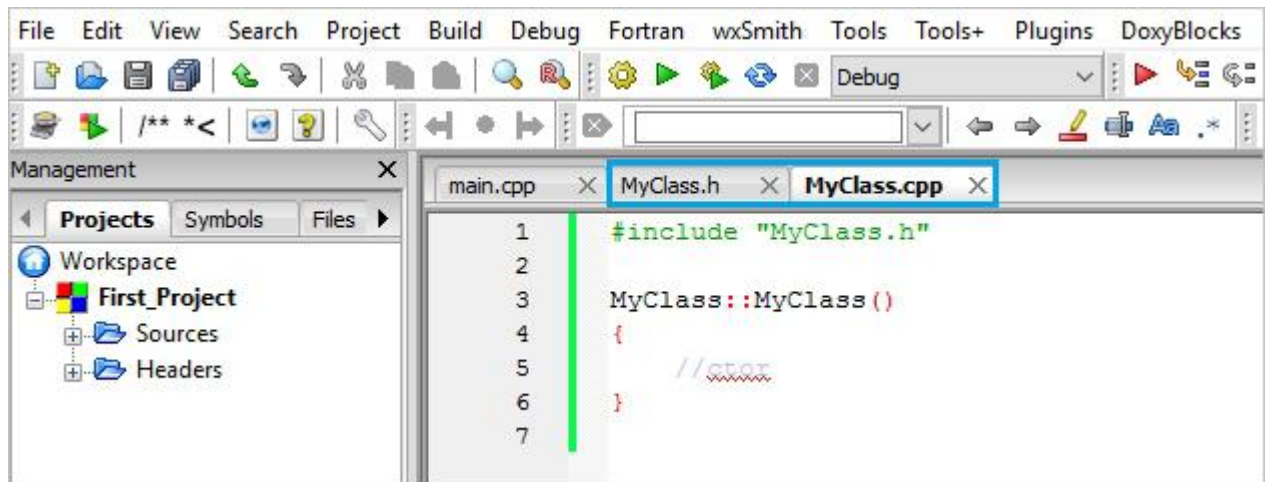
☒ Generate implementation file

Folder:

Filename:

Header include:

- 注意到两个新文件被添加到项目中：



新文件作为我们新类的模板。

- **MyClass.h** 是头文件。
- **MyClass.cpp** 是源文件。

✧ 源文件和头文件

- 头文件(.h)包含函数声明（原型）和变量声明。
- 它目前包含一个新的 **MyClass** 类的模板，有一个默认构造函数。

MyClass.h

```
#ifndef MYCLASS_H
#define MYCLASS_H

class MyClass
{
public:
    MyClass();
protected:
private:
};

#endif // MYCLASS_H
```

- 类及其方法的实现在源文件中(.cpp)。
- 目前它只包含一个空构造函数。

MyClass.cpp

```
#include "MyClass.h"

MyClass::MyClass()
{
```

```
//ctor  
}
```

头文件中的 `#ifndef` 和 `#define` 语句将在即将到来的课程中讨论。

✧ 作用域解析符

- 源文件(.cpp)中的双冒号被称为作用域解析运算符，它用于构造函数定义：

```
#include "MyClass.h"  
  
MyClass::MyClass()  
{  
    //ctor  
}
```

- 作用域解析运算符用于定义已经声明的特定类的成员函数。记住，我们在头文件中定义了构造函数原型。

因此，基本上，`MyClass::MyClass()` 引用 `MyClass()` 成员函数——或者，在这种情况下，是 `MyClass` 类的构造函数。

✧ 源文件和头文件

- 要在 main 中使用我们的类，我们需要包含头文件。
- 例如，要在 main 中使用我们新创建的 `MyClass`：

```
#include <iostream>  
#include "MyClass.h"  
using namespace std;  
  
int main() {  
    MyClass obj;  
}
```

头声明“什么”类（或正在实现的任何内容）将执行，而 `cpp` 源文件定义“如何”它将执行这些功能。

6.2.析构函数

✧ 析构函数

- 还记得构造函数吗？它们是在创建对象时自动调用的特殊成员函数。
- **析构函数**也是特殊函数。在销毁或删除对象时调用它们。

当对象超出范围时，或者只要将**删除**表达式应用于指向类对象的指针，对象就会被销毁。

✧ 析构函数

- **析构函数**的名称与类完全相同，仅以**波浪号(~)**为前缀。析构函数不能返回值或接受任何参数。

```
class MyClass {  
    public:  
        ~MyClass() {  
            // some code  
        }  
};
```

在退出程序之前，析构函数对于释放资源非常有用。这可以包括关闭文件，释放内存等。

✧ 析构函数

- 例如，让我们在其头文件 **MyClass.h** 中为 **MyClass** 类声明一个**析构函数**：

```
class MyClass  
{  
    public:  
        MyClass();  
        ~MyClass();  
};
```

声明 **MyClass** 类的析构函数。

✧ 析构函数

- 在头文件中声明析构函数后，我们可以在源文件 **MyClass.cpp** 中编写实现：

```
#include "MyClass.h"
#include <iostream>
using namespace std;

MyClass::MyClass()
{
    cout<<"Constructor"<<endl;
}

MyClass::~MyClass()
{
    cout<<"Destructor"<<endl;
}
```

请注意，我们包含了<iostream>标头，因此我们可以使用 **cout**。

✧ 析构函数

- 由于析构函数不能获取参数，因此它们也不能超载。
- 每个类只有一个析构函数。

定义析构函数不是强制性的；如果你不需要，你不必定义一个。

✧ 析构函数

- 让我们回到我们的主函数。

```
#include <iostream>
#include "MyClass.h"
using namespace std;

int main() {
    MyClass obj;

    return 0;
}
```

- 我们包含了类的头文件，然后创建了该类型的对象。
- 这将返回以下输出：

```
Constructor
Destructor
```

- 程序运行时，它首先创建对象并调用构造函数。删除该对象，并在程序执行完成时调用析构函数。

请记住，我们从构造函数中打印了“Construcor”，并从析构函数中打印了“Destructor”。

6.3.选择操作符

✧ #ifndef 和 #define

- 我们为类的头文件和源文件，这形成了这个头文件。

```
#ifndef MYCLASS_H
#define MYCLASS_H

class MyClass
{
public:
    MyClass();
protected:
private:
};

#endif // MYCLASS_H
```

ifndef 代表“if not defined”。第一对语句告诉程序定义 **MyClass** 头文件（如果尚未定义）。

endif 结束条件。

这可以防止头文件在一个文件中被多次包含。

✧ 成员函数

- 让我们在我们的类中创建一个名为 **myPrint()** 的示例函数。

```
MyClass.h
class MyClass
{
public:
    MyClass();
    void myPrint();
};
```

MyClass.cpp

```
#include "MyClass.h"
#include <iostream>
using namespace std;

MyClass::MyClass() {
}

void MyClass::myPrint() {
    cout << "Hello" << endl;
}
```

由于 **myPrint()** 是常规成员函数，因此必须在声明和定义中指定其返回类型。

✧ 点运算符

- 接下来，我们将创建一个 **MyClass** 类型的对象，并使用点(.)运算符调用其 **myPrint()** 函数：

```
#include "MyClass.h"

int main() {
    MyClass obj;
    obj.myPrint();
}

// Outputs "Hello"
```

✧ 指针

- 我们还可以使用指针来访问对象的成员。
- 以下指针指向 **obj** 对象：

```
MyClass obj;
MyClass *ptr = &obj;
```

指针的类型是 **MyClass**，因为它指向该类型的对象。

◇ 选择运算符

- 箭头成员选择运算符(**->**)用于使用指针访问对象的成员。

```
MyClass obj;  
MyClass *ptr = &obj;  
ptr->myPrint();
```

使用对象时，请使用**点成员选择运算符(.)**。

使用指向对象的指针时，请使用**箭头成员选择运算符(->)**。

6.4.常量对象

◇ 常量

- **常量**是具有固定值的表达式。程序运行时无法更改。
- 使用 **const** 关键字定义常量变量。

```
const int x = 42;
```

所有常量变量**必须在创建时初始化**。

◇ 常量对象

- 与内置数据类型一样，我们可以使用 **const** 关键字使类对象保持不变。

```
const MyClass obj;
```

- 所有 **const** 变量必须在创建时初始化。在类的情况下，这种初始化是通过构造函数完成的。如果未使用参数化构造函数初始化类，则必须提供公共默认构造函数 - 如果未提供公共默认构造函数，则会发生编译器错误。

- 一旦通过构造函数初始化了 **const** 类对象，就无法修改对象的成员变量。这包括直接更改公共成员变量和调用成员函数来设置成员变量的值。

当您使用 **const** 声明对象时，您无法在对象的生命周期内更改其数据成员。

✧ 常量对象

- 只有非 **const** 对象才能调用非 **const** 函数。
- 常量对象不能调用常规函数。因此，要使常量对象起作用，您需要一个常量函数。

• 要将函数指定为 **const** 成员，**const** 关键字必须遵循函数原型，在其参数的右括号之外。对于在类定义之外定义的 **const** 成员函数，必须在函数原型和定义上使用 **const** 关键字。例如：

MyClass.h

```
class MyClass
{
    public:
        void myPrint() const;
};
```

MyClass.cpp

```
#include "MyClass.h"
#include <iostream>
using namespace std;

void MyClass::myPrint() const {
    cout << "Hello" << endl;
}
```

- 现在 **myPrint()** 函数是一个常量成员函数。因此，它可由我们的常量对象调用：

```
int main() {
    const MyClass obj;
    obj.myPrint();
}
// Outputs "Hello"
```

✧ 常量对象

- 尝试从常量对象调用常规函数会导致错误。
- 此外，当任何 **const** 成员函数尝试更改成员变量或调用非 **const** 成员函数时，都会生成编译器错误。

定义常量对象和函数可确保相应的数据成员不会被意外修改。

6.5.成员初始化

✧ 成员初始化

- 回想一下，**常量**是无法更改的变量，并且必须在创建时初始化所有 **const** 变量。
- C++提供了一种方便的语法来初始化类的成员，称为**成员初始化列表**（也称为**构造函数初始化程序**）。

✧ 成员初始化

- 考虑以下类：

```
class MyClass {  
    public:  
        MyClass(int a, int b) {  
            regVar = a;  
            constVar = b;  
        }  
    private:  
        int regVar;  
        const int constVar;  
};
```

- 该类有两个成员变量，**regVar** 和 **constVar**。它还有一个构造函数，它接受两个参数，用于初始化成员变量。
- 运行此代码会返回**错误**，因为其成员变量之一是**常量**，在声明后无法为其分配值。
- 在这种情况下，可以使用**成员初始化列表**为成员变量赋值。

```
class MyClass {  
    public:  
        MyClass(int a, int b)  
        : regVar(a), constVar(b)  
        {  
        }  
    private:  
        int regVar;  
        const int constVar;  
};
```

- 请注意，在语法中，初始化列表遵循构造函数参数。该列表以**冒号(:)**开头，然后列出要初始化的每个变量以及该变量的值，并使用逗号分隔它们。
- 使用语法**变量（值）**来指定值。

初始化列表消除了**在构造函数体中放置显式赋值的需要**。此外，初始化列表不以分号结束。

✧ 成员初始化

- 让我们使用单独的头文件和源文件编写前面的示例。

MyClass.h

```
class MyClass {  
    public:  
        MyClass(int a, int b);  
    private:  
        int regVar;  
        const int constVar;  
};
```

MyClass.cpp

```
MyClass::MyClass(int a, int b)  
: regVar(a), constVar(b)  
{  
    cout << regVar << endl;  
    cout << constVar << endl;  
}
```

- 我们在构造函数中添加了 **cout** 语句来打印成员变量的值。
- 我们的下一步是在 **main** 中创建类的对象，并使用构造函数来赋值。

```
#include "MyClass.h"  
  
int main() {  
    MyClass obj(42, 33);  
}  
  
/*Outputs  
42  
33  
*/
```

构造函数用于创建对象，通过成员初始化列表为成员变量分配两个参数。

✧ 成员初始化

- 成员初始化列表可用于常规变量，并且必须用于常量变量。

即使在成员变量不是常量的情况下，使用成员初始化也是有意义的。

6.6.组合 1

✧ 组合

• 在现实世界中，复杂对象通常使用更小，更简单的对象构建。例如，使用金属框架，发动机，轮胎和许多其他部件组装汽车。这个过程叫做组合。

- 在 C++ 中，对象组合涉及在其他类中使用类作为成员变量。
- 此示例程序演示了组合的实际操作。它包含 **Person** 和 **Birthday** 类，每个 **Person** 都有一个 **Birthday** 对象作为其成员。

Birthday:

```
class Birthday {
public:
    Birthday(int m, int d, int y)
        : month(m), day(d), year(y)
    {
    }
private:
    int month;
    int day;
    int year;
};
```

• 我们的 **Birthday** 类有三个成员变量。它还有一个构造函数，使用成员初始化列表初始化成员。

为简单起见，该类在单个文件中声明。或者，您可以使用标头和源文件。

✧ 组合

- 我们还在我们的 **Birthday** 类中添加一个 **printDate()** 函数：

```
class Birthday {
public:
    Birthday(int m, int d, int y)
```

```

: month(m), day(d), year(y)
{
}
void printDate()
{
    cout<<month<<"/"<<day
    <<"/"<<year<<endl;
}
private:
    int month;
    int day;
    int year;
};

```

将 `printDate()` 函数添加到 `Birthday` 类。

✧ 组合

- 接下来，我们可以创建 `Person` 类，其中包括 `Birthday` 类。

```

#include <string>
#include "Birthday.h"

class Person {
public:
    Person(string n, Birthday b)
    : name(n),
      bd(b)
    {
    }
private:
    string name;
    Birthday bd;
};

```

- `Person` 类有一个 `name` 和一个 `Birthday` 成员，以及一个用于初始化它们的构造函数。
- 确保包含相应的头文件。

6.7.组合 2

✧ 组合

- 现在，我们的 **Person** 类有一个类型为 **Birthday** 的成员：

```
class Person {  
public:  
    Person(string n, Birthday b)  
    : name(n),  
      bd(b)  
    {  
    }  
private:  
    string name;  
    Birthday bd;  
};
```

组合用于共享 **has-a** 关系的对象，如“A **Person** has a **Birthday**”。

✧ 组合

- 让我们在 **Person** 类中添加一个 **printInfo()** 函数，它打印对象的数据：

```
class Person {  
public:  
    Person(string n, Birthday b)  
    : name(n),  
      bd(b)  
    {  
    }  
    void printInfo()  
    {  
        cout << name << endl;  
        bd.printDate();  
    }  
private:  
    string name;  
    Birthday bd;  
};
```

请注意，我们可以调用 **bd** 成员的 **printDate()** 函数，因为它的类型为 **Birthday**，它定义了该函数。

◇ 组合

• 现在我们已经定义了我们的 **Birthday** 和 **Person** 类，我们可以转到主函数，创建一个 **Birthday** 对象，然后将它传递给 **Person** 对象。

```
int main() {  
    Birthday bd(2, 21, 1985);  
    Person p("David", bd);  
    p.printInfo();  
}  
  
/*Outputs  
David  
2/21/1985  
*/
```

• 我们为 1985/2/21 创建了一个 **Birthday** 对象。接下来，我们创建了一个 **Person** 对象，并将 **Birthday** 对象传递给它的构造函数。最后，我们使用 **Person** 对象的 **printInfo()** 函数来打印其数据。

通常，组合用于保持每个单独的类相对简单，直接，并专注于执行一个任务。它还使每个子对象都是自包含的，允许可重用性（我们可以在各种其他类中使用 **Birthday** 类）。

6.8.Friend 关键字

◇ 友元函数

- 通常，无法从该类外部访问类的私有成员。
- 但是，将非成员函数声明为类的朋友允许它访问类的私有成员。这是通过在类中包含此外部函数的声明，并在其前面加上关键字 **friend** 来实现的。
- 在下面的示例中，**someFunc()**（它不是该类的成员函数）是 **MyClass** 的朋友，可以访问其私有成员。

```
class MyClass {  
    public:  
        MyClass() {  
            regVar = 0;  
        }  
    private:
```



```
int regVar;

friend void someFunc(MyClass &obj);
};
```

请注意，在将对象传递给函数时，我们需要使用**&**运算符**通过引用**传递它。

✧ 友元函数

- 函数 **someFunc()** 被定义为类外的常规函数。它采用 **MyClass** 类型的对象作为其参数，并且能够访问该对象的私有数据成员。

```
class MyClass {
public:
    MyClass() {
        regVar = 0;
    }
private:
    int regVar;

    friend void someFunc(MyClass &obj);
};

void someFunc(MyClass &obj) {
    obj.regVar = 42;
    cout << obj.regVar;
}
```

- someFunc()** 函数更改对象的私有成员并打印其值。

要使其成员可访问，该类必须在其定义中将该函数声明为 **friend**。你不能把一个函数“成为”一个类的朋友，而不让这个类“放弃”对该函数的友谊。

✧ 友元函数

- 现在我们可以 **在 main 中创建一个对象并调用 someFunc() 函数**：

```
int main() {
    MyClass obj;
    someFunc(obj);
}

//Outputs 42
```

- **someFunc()**能够修改对象的私有成员并打印其值。
- 友元函数的典型用例是在访问两个不同类私有成员之间进行的操作。

您可以在任意数量的类中声明函数朋友。

与友元函数类似，您可以定义友元类，该类可以访问另一个类的私有成员。

6.9.This 关键字

✧ This

- C++中的每个对象都可以通过一个名为 **this** 指针的重要指针访问自己的地址。
- 在成员函数内部，**this** 可以用于引用调用对象。
- 让我们创建一个示例类：

```
class MyClass {
public:
    MyClass(int a) : var(a)
    {}
private:
    int var;
};
```

友元函数没有 **this** 指针，因为朋友不是类的成员。

✧ This

- **printInfo()**方法提供了三种打印类的成员变量的替代方法。

```
class MyClass {
public:
    MyClass(int a) : var(a)
    {}
    void printInfo() {
        cout << var<<endl;
        cout << this->var<<endl;
        cout << (*this).var<<endl;
    }
private:
    int var;
};
```

- 所有三种选择都会产生相同的结果。

this 是指向对象的**指针**，因此箭头选择运算符用于选择成员变量。

✧ This

- 要查看结果，我们可以创建类的对象并调用成员函数。

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

class MyClass {
public:
    MyClass(int a) : var(a)
    {}
    void printInfo() {
        cout << var << endl;
        cout << this->var << endl;
        cout << (*this).var << endl;
    }
private:
    int var;
};

int main() {
    MyClass obj(42);
    obj.printInfo();
}

/* Outputs
42
42
42
*/
```

- 访问成员变量的所有三种方法都有效。

请注意，只有成员函数具有 **this** 指针。

✧ This

- 当您可以选择直接指定变量时，您可能想知道为什么必须使用 **this** 关键字。

- **this** 关键字在**运算符重载**中起着重要作用，将在下一课中介绍。

6.10.运算符重载

✧ 运算符重载

- 大多数 C++ 内置运算符都可以重新定义或**重载**。
- 因此，运算符也可以与用户定义的类型一起使用（例，允许您将两个对象**相加**）。
- 此图表显示可以重载的运算符。

+	-	*	/	%	^
&		~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	
+=	-=	/=	%=	^=	&=
=	*=	<<=	>>=	[]	()
->	->*	new	new[]	delete	delete[]

无法重载的运算符包括 :: | .* | . | ?:

✧ 运算符重载

- 让我们声明一个示例类来演示运算符重载：

```
class MyClass {
public:
    int var;
    MyClass() {}
    MyClass(int a)
    : var(a) {}
};
```

- 我们的类有两个构造函数和一个成员变量。

我们将重载+运算符，以便能够将我们类的两个对象一起添加。

✧ 运算符重载

- 重载运算符是函数，由关键字**运算符**定义，后跟定义的运算符的符号。
- 重载运算符与其他函数类似，因为它具有**返回类型**和**参数列表**。
- 在我们的例子中，我们将重载**+**运算符。它将**返回**我们类的一个对象，并将我们类的一个对象作为其**参数**。

```
class MyClass {  
public:  
    int var;  
    MyClass() {}  
    MyClass(int a)  
    : var(a) {}  
  
    MyClass operator+(MyClass &obj) {  
    }  
};
```

现在，我们需要定义函数的功能。

✧ 运算符重载

- 我们需要我们的**+**运算符返回一个新的 **MyClass** 对象，其成员变量等于两个对象的成员变量之和。

```
class MyClass {  
public:  
    int var;  
    MyClass() {}  
    MyClass(int a)  
    : var(a) {}  
  
    MyClass operator+(MyClass &obj) {  
        MyClass res;  
        res.var= this->var+obj.var;  
        return res;  
    }  
};
```

- 在这里，我们声明了一个新的 **res** 对象。然后，我们将当前对象(**this**)和参数对象(**obj**)的成员变量的总和分配给 **res** 对象的 **var** 成员变量。结果返回 **res** 对象。

- 这使我们能够在 `main` 中创建对象，并使用重载`+`运算符将它们添加到一起。

```
int main() {  
    MyClass obj1(12), obj2(55);  
    MyClass res = obj1+obj2;  
  
    cout << res.var;  
}  
  
//Outputs 67
```

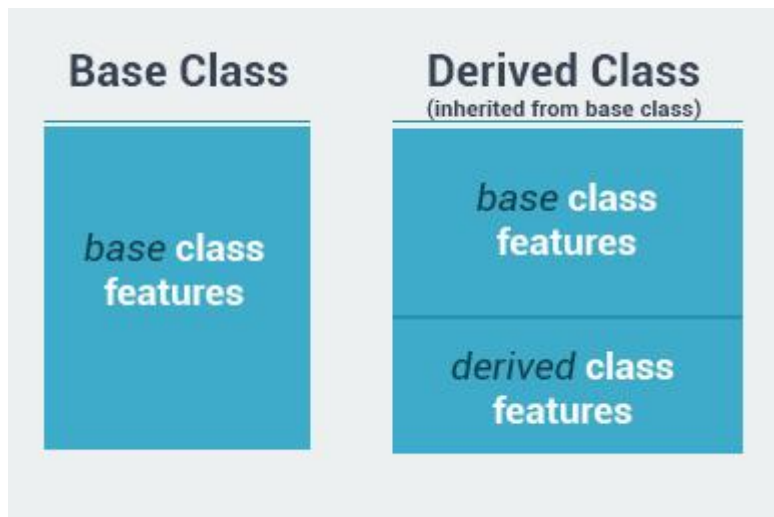
使用重载运算符，您可以使用任何所需的自定义逻辑。但是，不可能改变运算符的优先级，分组或操作数。

7.继承与多态

7.1.继承

◇ 继承

- **继承**是面向对象编程最重要的概念之一。
- 继承允许我们基于另一个类定义一个类。这有助于更轻松地创建和维护应用程序。
- 其属性由另一个类继承的类称为**基类**。继承属性的类称为**派生类**。例如，**Daughter** 类（派生）可以从 **Mother** 类（基）继承。
- 派生类继承了基类的所有功能，并且可以拥有自己的附加功能。



继承的想法实现了这种关系。例如，哺乳动物**是一种**动物，狗**是一种**哺乳动物，因此也是狗**是一种**动物。

◇ 继承

- 为了演示继承，让我们创建一个 **Mother** 类和一个 **Daughter** 类：

```
class Mother
{
public:
    Mother() {}
    void sayHi() {
        cout << "Hi";
    }
}
```

```

    }
};

class Daughter
{
public:
    Daughter() {};
};

```

- **Mother** 类有一个名为 **sayHi()** 的公共方法。

下一步是**继承**（派生）母亲的女儿。

✧ 继承

- 此语法从 **Mother** 类派生 **Daughter** 类。

```

class Daughter : public Mother
{
public:
    Daughter() {};
};

```

- **Base** 类是使用冒号和访问说明符指定的：**public** 表示基类的所有公共成员在派生类中都是公共的。

换句话说，**Mother** 类的所有公共成员都成为 **Daughter** 的公共成员。

✧ 继承

- 由于 **Mother** 类的所有公共成员都成为了 **Daughter** 类的公共成员，我们可以创建一个类型为 **Daughter** 的对象，并为该对象调用 **Mother** 类的 **sayHi()** 函数：

```

#include <iostream>
using namespace std;

class Mother
{
public:
    Mother() {};
    void sayHi() {
        cout << "Hi";
    }
};

```



```

class Daughter: public Mother
{
    public:
        Daughter() {};
};

int main() {
    Daughter d;
    d.sayHi();
}
//Outputs "Hi"

```

- 派生类继承所有基类方法，但以下情况除外：
 - 构造函数，析构函数
 - 重载运算符
 - 友元函数

通过在逗号分隔列表中指定基类，可以从多个类派生类。例如： **class Daughter:public Mother, public Father**

7.2.受保护的成员

✧ 访问说明符

- 到目前为止，我们专门使用 **public** 和 **private** 访问说明符。
- 公共成员可以从类外的任何地方访问，而私人成员的访问仅限于他们的类和朋友功能。

正如我们之前看到的，使用公共方法访问私有类变量是一种很好的做法。

✧ 受保护

- 还有一个访问说明符 - **受保护**。
- **受保护**的成员变量或函数与私有成员非常相似，但有一点不同 - 可以在派生类中访问它。

```

class Mother {
    public:
        void sayHi() {
            cout << var;
        }
}

```

```
private:
    int var=0;

protected:
    int someVar;
};
```

现在 **someVar** 可以被任何派生自 **Mother** 类的类访问。

✧ 继承类型

- 访问说明符还用于指定**继承的类型**。
- 记住，我们使用 **public** 来继承子类：

```
class Daughter: public Mother
```

- **私有和受保护的**访问说明符也可以在这里使用。
- **公有继承**：基类的公共成员成为派生类的公共成员，基类的受保护成员成为派生类的受保护成员。基类的私有成员永远不能直接从派生类访问，但可以通过调用基类的公共成员和受保护成员来访问。
- **受保护的继承**：基类的公共成员和受保护成员成为派生类的受保护成员。
- **私有继承**：基类的公共成员和受保护成员成为派生类的私有成员。

公共继承是最常用的继承类型。

如果在继承类时未使用访问说明符，则默认情况下该类型将变为**私有**。

7.3.派生类的构造函数和析构函数

✧ 继承

- 继承类时，不会继承基类的构造函数和析构函数。
- 但是，在创建或删除派生类的对象时会调用它们。
- 为了进一步解释这种行为，让我们创建一个包含构造函数和析构函数的示例类：

```
class Mother {
public:
    Mother()
```

```

{
    cout <<"Mother ctor"<<endl;
}
~Mother()
{
    cout <<"Mother dtor"<<endl;
}
};

```

- 在 main 中创建对象会产生以下输出：

```

int main() {
    Mother m;
}
/* Outputs
Mother ctor
Mother dtor
*/

```

程序完成运行后，将创建并删除该对象。

✧ 继承

- 接下来，让我们创建一个具有自己的构造函数和析构函数的 **Daughter** 类，并使其成为 **Mother** 的派生类：

```

class Daughter: public Mother {
public:
    Daughter()
    {
        cout <<"Daughter ctor"<<endl;
    }
    ~Daughter()
    {
        cout <<"Daughter dtor"<<endl;
    }
};

```

使用自己的构造函数和析构函数创建一个 **Daughter** 类。

◇ 继承

- 现在，当我们创建一个 **Daughter** 对象时会发生什么？

```
int main() {  
    Daughter m;  
}  
  
/*Outputs  
Mother ctor  
Daughter ctor  
Daughter dtor  
Mother dtor  
*/
```

- 请注意，首先调用基类的构造函数，然后调用派生类的构造函数。
- 当对象被销毁时，调用派生类的析构函数，然后调用基类的析构函数。

您可以将其视为以下内容：派生类需要其基类才能工作 - 这就是首先设置基类的原因。

◇ 总结

构造函数

- 首先调用基类构造函数。

析构函数

- 首先调用派生类析构函数，然后调用基类析构函数。

此序列使您可以为派生类指定初始化和取消初始化方案。

7.4.多态

◇ 多态

- **多态**这个词的意思是“有很多形式”。
- 通常，当存在类的层次结构并且它们通过继承相关时，会发生多态性。

• C++多态意味着对成员函数的调用将导致执行不同的实现，具体取决于调用该函数的对象的**类型**。

简单地说，多态性意味着单个函数可以具有许多不同的实现。

◇ 多态

- 使用一个例子可以更清楚地证明多态性：
- 假设你想制作一个简单的游戏，其中包括不同的敌人：怪物，忍者等。所有敌人都有一个共同的功能：**攻击**函数。但是，他们每个人都以不同的方式进行攻击。在这种情况下，多态允许在不同的对象上调用相同的**攻击**函数，但会导致不同的行为。

第一步是创建**敌人类**。

```
class Enemy {  
protected:  
    int attackPower;  
public:  
    void setAttackPower(int a){  
        attackPower = a;  
    }  
};
```

我们的 **Enemy** 类有一个名为 **setAttackPower** 的公共方法，它设置受保护的 **attackPower** 成员变量。

◇ 多态

- 我们的第二步是为两种不同类型的敌人 **Ninjas** 和 **Monsters** 创建类。这两个新类都继承自 **Enemy** 类，因此每个类都具有攻击力。同时，每个都有特定的**攻击**函数。

```
class Ninja: public Enemy {  
public:  
    void attack() {  
        cout << "Ninja! - "<< attackPower << endl;  
    }  
};  
  
class Monster: public Enemy {  
public:  
    void attack() {  
        cout << "Monster! - "<< attackPower << endl;  
    }  
};
```

```
};
```

- 如您所见，他们各自的**攻击**函数各不相同。
- 现在我们可以 **在 main 中创建我们的 Ninja 和 Monster 对象。**

```
int main() {  
    Ninja n;  
    Monster m;  
}
```

Ninja 和 **Monster** 继承自 **Enemy**，因此所有 **Ninja** 和 **Monster** 对象都是 **Enemy** 对象。这允许我们执行以下操作：

```
Enemy *e1 = &n;  
Enemy *e2 = &m;
```

我们现在创建了两个类型为 **Enemy** 的指针，指向 **Ninja** 和 **Monster** 对象。

✧ 多态

- 现在，我们可以调用相应的函数：

```
int main() {  
    Ninja n;  
    Monster m;  
    Enemy *e1 = &n;  
    Enemy *e2 = &m;  
  
    e1->setAttackPower(20);  
    e2->setAttackPower(80);  
  
    n.attack();  
    m.attack();  
}  
  
/* Output:  
Ninja! - 20  
Monster! - 80  
*/
```

- 通过直接在对象上调用函数，我们可以获得相同的结果。但是，使用指针更快更有效。
- 此外，指针表明你可以使用 **Enemy** 指针，而不知道它包含子类的对象。

7.5.虚函数

✧ 虚函数

- 前面的示例演示了如何使用基类指针来派生类。为什么这有用？继续我们的游戏示例，我们希望每个敌人都有一个 **attack()** 函数。
- 为了能够使用 **Enemy** 指针为每个派生类调用相应的 **attack()** 函数，我们需要将基类函数声明为 **virtual**。
- 在基类中定义虚函数，在派生类中使用相应的版本，允许多态使用 **Enemy** 指针来调用派生类的函数。
- 每个派生类都将覆盖 **attack()** 函数并具有单独的实现：

```
class Enemy {
public:
    virtual void attack() {
    }
};

class Ninja: public Enemy {
public:
    void attack() {
        cout << "Ninja!"<<endl;
    }
};

class Monster: public Enemy {
public:
    void attack() {
        cout << "Monster!"<<endl;
    }
};
```

虚函数是使用关键字 **virtual** 声明的基类函数。

✧ 虚函数

- 现在，我们可以使用 **Enemy** 指针来调用 **attack()** 函数。

```
int main() {
    Ninja n;
    Monster m;
```

```

Enemy *e1 = &n;
Enemy *e2 = &m;

e1->attack();
e2->attack();
}

/* Output:
Ninja!
Monster!
*/

```

由于 `attack()` 函数被声明为 `virtual`，它就像一个模板，告诉派生类可能有自己的 `attack()` 函数。

✧ 虚函数

- 我们的游戏示例用于演示多态的概念；我们使用 **Enemy** 指针调用相同的 `attack()` 函数，并生成不同的结果。

```

e1->attack();
e2->attack();

```

- 如果基类中的函数是**虚函数**，则根据引用的对象的实际类型调用派生类中函数的实现，而不管指针的声明类型如何。

声明或继承虚函数的类称为**多态类**。

7.6.抽象类

✧ 虚函数

- 虚函数也可以在基类中实现：

```

class Enemy {
public:
    virtual void attack() {
        cout << "Enemy!"<<endl;
    }
};

class Ninja: public Enemy {
public:

```



```

void attack() {
    cout << "Ninja!"<<endl;
}
};

class Monster: public Enemy {
public:
    void attack() {
        cout << "Monster!"<<endl;
    }
};

```

• 现在，当您创建一个 **Enemy** 指针并调用 **attack()** 函数时，编译器将调用该指针指向的函数，该函数对应于对象的类型：

```

int main() {
    Ninja n;
    Monster m;
    Enemy e;

    Enemy *e1 = &n;
    Enemy *e2 = &m;
    Enemy *e3 = &e;

    e1->attack();
    // Outputs "Ninja!"

    e2->attack();
    // Outputs "Monster!"

    e3->attack();
    // Outputs "Enemy!"
}

```

这就是通常使用**多态**的方式。不同的类具有相同名称的函数，甚至相同的参数，但具有不同的实现。

✧ 纯虚函数

- 在某些情况下，您希望在基类中包含一个虚函数，以便可以在派生类中重新定义它以适合该类的对象，但是你没有为基类中的函数提供有意义的定义。
- 没有定义的虚拟成员函数称为**纯虚函数**。它们基本上指定派生类自己定义该函

数。

- 语法是将它们的定义替换为 `= 0`（等号和零）：

```
class Enemy {  
public:  
    virtual void attack() = 0;  
};
```

`= 0` 告诉编译器该函数没有正文。

◇ 纯虚函数

- 纯虚函数基本上定义了派生类将自己定义的函数。
- 从具有纯虚函数的类继承的每个派生类都必须覆盖该函数。

如果未在派生类中重写纯虚函数，则在尝试实例化派生类的对象时，代码无法编译并导致错误。

◇ 纯虚函数

- 必须在派生类中覆盖 **Enemy** 类中的纯虚函数。

```
class Enemy {  
public:  
    virtual void attack() = 0;  
};  
  
class Ninja: public Enemy {  
public:  
    void attack() {  
        cout << "Ninja!"<<endl;  
    }  
};  
  
class Monster: public Enemy {  
public:  
    void attack() {  
        cout << "Monster!"<<endl;  
    }  
};
```

◇ 抽象类

- 您**无法**使用纯虚函数创建基类的对象。
- 运行以下代码将返回错误：

```
Enemy e; // Error
```

- 这些类称为**抽象**。它们是只能用作基类的类，因此可以使用纯虚函数。
- 您可能认为抽象基类是无用的，但事实并非如此。它可用于创建指针并利用其所有多态性功能。
- 例如，你可以写：

```
Ninja n;  
Monster m;  
Enemy *e1 = &n;  
Enemy *e2 = &m;  
  
e1->attack();  
e2->attack();
```

在此示例中，使用唯一类型的指针（**Enemy ***）引用不同但相关类型的对象，并且每次调用适当的成员函数，因为它们是虚拟的。

8.模板、异常和文件

8.1.函数模板

✧ 函数模板

- 函数和类有助于使程序更易于编写，更安全，更易于维护。
- 但是，虽然函数和类确实具有所有这些优点，但在某些情况下，它们也可能受到 C++ 的要求的限制，即您需要为所有参数指定类型。

- 例如，您可能想要编写一个计算两个数字之和的函数，类似于：

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

您可以使用模板来定义函数和类。让我们看看它们是如何工作的。

✧ 函数模板

- 我们现在可以在 main 中调用两个整数的函数。

```
int sum(int a, int b) {  
    return a+b;  
}  
  
int main () {  
    int x=7, y=15;  
    cout << sum(x, y) << endl;  
}  
// Outputs 22
```

该函数按预期工作，但仅限于**整数**。

✧ 函数模板

- 有必要为每个新类型编写一个新函数，例如双精度。

```
double sum(double a, double b) {  
    return a+b;
```

```
}
```

- 能够编写一个版本的 `sum()` 来处理任何类型的参数会不会更有效率?
- **函数模板**使我们能够做到这一点!
- 使用**函数模板**, 基本思想是避免为每个变量指定精确类型的必要性。相反, C++ 为我们提供了使用占位符类型定义函数的功能, 称为**模板类型参数**。

- 要定义函数模板, 请使用关键字**模板**, 然后使用模板类型定义:

```
template <class T>
```

我们将模板类型命名为 **T**, 这是一种通用数据类型。

✧ 函数模板

- 现在我们可以使用我们的通用数据类型 **T**:

```
template <class T>
T sum(T a, T b) {
    return a+b;
}

int main () {
    int x=7, y=15;
    cout << sum(x, y) << endl;
}

// Outputs 22
```

- 该函数返回泛型类型 **T** 的值, 取两个参数, 类型为 **T**.

我们的新函数与前一个函数完全相同。

✧ 函数模板

- 相同的函数可以与其他数据类型一起使用, 例如双精度数:

```
template <class T>
T sum(T a, T b) {
    return a+b;
}

int main () {
    double x=7.15, y=15.54;
```

```
cout << sum(x, y) << endl;
}
// Outputs 22.69
```

- 编译器自动调用相应类型的函数。

创建模板类型参数时，关键字 **typename** 可用作关键字 **class** 的替代：**template <typename T>**。

在这种情况下，关键字是相同的，但在本课程中，我们将使用关键字 **class**。

✧ 函数模板

- 模板函数可以节省大量时间，因为它们只编写一次，并且可以使用不同类型。
- 模板函数减少了代码维护，因为重复的代码显着减少。

增强安全性是使用模板功能的另一个优点，因为不需要手动复制功能和更改类型。

8.2.多参数函数模板

✧ 函数模板

- 函数模板还可以使用多种通用数据类型。使用逗号分隔列表定义数据类型。
- 让我们创建一个函数来比较不同数据类型（**int** 和 **double**）的参数，并打印较小的一个。

```
template <class T, class U>
```

如您所见，此模板声明了两种不同的通用数据类型，**T** 和 **U**。

✧ 函数模板

- 现在我们可以继续我们的函数声明：

```
template <class T, class U>
T smaller(T a, U b) {
    return (a < b ? a : b);
}
```

三元运算符检查 **a < b** 条件并返回相应的结果。表达式 **(a < b ? a : b)** 等于表达式，如果 **a** 小于 **b**，则返回 **a**，否则返回 **b**。

✧ 函数模板

- 在我们的主函数中，我们可以将函数用于不同的数据类型：

```
template <class T, class U>
T smaller(T a, U b) {
    return (a < b ? a : b);
}

int main () {
    int x=72;
    double y=15.34;
    cout << smaller(x, y) << endl;
}

// Outputs 15
```

输出转换为**整数**，因为我们将函数模板的返回类型指定为与第一个参数（T）相同的类型，即整数。

✧ 函数模板

- T 是 Type 的缩写，是类型参数的广泛使用名称。
- 但是，没有必要使用 T；您可以使用适合您的任何标识符声明您的类型参数。您需要避免的唯一术语是 C++ 关键字。

请记住，当您声明模板参数时，绝对**必须**在函数定义中使用它。否则，编译器会抱怨！

8.3. 类模板

✧ 类模板

- 正如我们可以定义函数模板一样，我们也可以定义**类模板**，允许类具有使用模板参数作为类型的成员。
- 相同的语法用于定义类模板：

```
template <class T>
class MyClass {
```

```
};
```

与函数模板一样，您可以使用逗号分隔列表定义多个通用数据类型。

✧ 类模板

- 作为一个例子，让我们创建一个类 **Pair**，它将持有一对泛型类型的值。

```
template <class T>
class Pair {
private:
    T first, second;
public:
    Pair (T a, T b):
        first(a), second(b) {
    }
};
```

上面的代码声明了一个类模板 **Pair**，它有两个泛型类型的私有变量，以及一个用于初始化变量的构造函数。

✧ 类模板

- 如果您在类之外定义成员函数，则需要特定的语法-例如，在单独的源文件中。您需要在类名后面的尖括号中指定泛型类型。

例如，要在类外部定义成员函数 **greater()**，请使用以下语法：

```
template <class T>
class Pair {
private:
    T first, second;
public:
    Pair (T a, T b):
        first(a), second(b){
    }
    T bigger();
};

template <class T>
T Pair<T>::bigger() {
    // some code
}
```


如果您在类之外定义成员函数，则需要特定语法。

✧ 类模板

- 较大的函数返回两个成员变量的较大值。

```
template <class T>
class Pair {
private:
    T first, second;
public:
    Pair (T a, T b):
        first(a), second(b){
    }
    T bigger();
};

template <class T>
T Pair<T>::bigger() {
    return (first>second ? first : second);
}
```

三元运算符比较两个变量，返回较大的变量。

✧ 类模板

- 要为不同类型创建模板类的对象，请在尖括号中指定数据类型，就像在类外定义函数时所做的那样。
- 在这里，我们为整数创建一个 **Pair** 对象。

```
Pair <int> obj(11, 22);
cout << obj.bigger();
// Outputs 22
```

- 我们可以使用相同的类来创建存储任何其他类型的对象。

```
Pair <double> obj(23.43, 5.68);
cout << obj.bigger();
// Outputs 23.43
```

8.4.模板特化

✧ 模板特化

- 对于常规类模板，类处理不同数据类型的方式是相同的；所有数据类型都运行相同的代码。

- **模板特化**允许将特定类型作为模板参数传递时模板有不同的实现。

- 例如，我们可能需要以与数值数据类型不同的方式处理字符数据类型。

- 为了演示这是如何工作的，我们可以先创建一个常规模板。

```
template <class T>
class MyClass {
public:
    MyClass (T x) {
        cout <<x<<" - not a char"<<endl;
    }
};
```

作为常规类模板，**MyClass** 以相同的方式处理所有各种数据类型。

✧ 模板特化

- 要为数据类型 **char** 指定不同的行为，我们将创建模板特化。

```
template <class T>
class MyClass {
public:
    MyClass (T x) {
        cout <<x<<" - not a char"<<endl;
    }
};

template <>
class MyClass<char> {
public:
    MyClass (char x) {
        cout <<x<<" is a char!"<<endl;
    }
};
```

- 首先，请注意我们在类名前加上 **template<>**，包括一个空参数列表。这是因为所有类型都是已知的，并且此特化不需要模板参数，但仍然是类模板的特化，因

此需要注意这样。

- 但是比这个前缀更重要的是类模板名后面的**<char>**特化参数。 此特化参数本身标识模板类专用的类型(**char**)。

在上面的示例中，第一个类是通用模板，第二个类是特化。

如有必要，你的特化可以表明与通用模板的行为完全不同的行为。

✧ 模板特化

- 下一步是声明不同类型的对象并检查结果：

```
int main () {  
    MyClass<int> ob1(42);  
    MyClass<double> ob2(5.47);  
    MyClass<char> ob3('s');  
}  
/* Output:  
42 - not a char  
5.47 - not a char  
s is a char!  
*/
```

- 如您所见，通用模板适用于 **int** 和 **double**。但是，我们为 **char** 数据类型调用了模板特化。

请记住，从通用模板到特化没有成员“继承”，因此模板特化的所有成员必须自己定义。

8.5.异常

✧ 异常

- 程序执行期间发生的问题称为**异常**。
- 在 C++ 中，异常是对程序运行时出现的异常的响应，例如尝试除以零。

✧ 抛出异常

- C++异常处理基于三个关键字：**try**，**catch** 和 **throw**。
- **throw** 用于在出现问题时抛出异常。

例如：

```
int motherAge = 29;
int sonAge = 36;
if (sonAge > motherAge) {
    throw "Wrong age values";
}
```

- 代码查看 **sonAge** 和 **motherAge**，如果发现 **sonAge** 是两者中的较大者，则抛出异常。

在 **throw** 语句中，操作数确定异常的类型。这可以是任何表达。表达式结果的类型将决定抛出的异常的类型。

✧ 捕获异常

- **try** 块标识将激活特定异常的代码块。接下来是一个或多个 **catch** 块。**catch** 关键字表示在抛出特定异常时执行的代码块。
- 可以生成异常的代码被 **try / catch** 块包围。
- 您可以通过关键字 **catch** 后面的括号中显示的异常声明来指定要捕获的异常类型。

例如：

```
try {
    int motherAge = 29;
    int sonAge = 36;
    if (sonAge > motherAge) {
        throw 99;
    }
}
catch (int x) {
    cout<<"Wrong age values - Error "<<x;
}

//Outputs "Wrong age values - Error 99"
```

- **try** 块抛出异常，然后 **catch** 块处理它。
- 错误代码 99 是一个整数，出现在 **throw** 语句中，因此它会导致 **int** 类型的异

常。

如果 `try` 块抛出多个异常，可以列出多个 `catch` 语句来处理各种异常。

8.6.更多关于异常

✧ 异常处理

- 处理用户输入时，异常处理特别有用。
- 例如，对于请求用户输入两个数字，然后输出它们的除法的程序，请确保处理除零，以防用户输入 0 作为第二个数字。

```
int main() {  
    int num1;  
    cout << "Enter the first number:";  
    cin >> num1;  
  
    int num2;  
    cout << "Enter the second number:";  
    cin >> num2;  
  
    cout << "Result:" << num1 / num2;  
}
```

- 如果用户输入 0 以外的任何数字，该程序将完美运行。

如果为 0，程序崩溃，所以我们需要处理该输入。

✧ 异常处理

- 如果第二个数字等于 0，我们需要抛出异常。

```
int main() {  
    int num1;  
    cout << "Enter the first number:";  
    cin >> num1;  
  
    int num2;  
    cout << "Enter the second number:";  
    cin >> num2;  
  
    if(num2 == 0) {
```

```

    throw 0;
}

cout << "Result:" << num1 / num2;
}

```

- 此代码抛出类型为 `integer` 的代码 `0` 的异常。

✧ 异常处理

- 现在我们需要使用 `try / catch` 块来处理抛出的异常。

```

int main() {
    try {
        int num1;
        cout << "Enter the first number:";
        cin >> num1;

        int num2;
        cout << "Enter the second number:";
        cin >> num2;

        if(num2 == 0) {
            throw 0;
        }

        cout << "Result:" << num1 / num2;
    }
    catch(int x) {
        cout << "Division by zero!";
    }
}

```

- 当输入 `0` 作为第二个数字时，将导致输出“Division by zero!”替代程序崩溃。
- 在我们的例子中，我们只捕获类型为 `整数` 的异常。可以指定 `catch` 块处理 `try` 块中抛出的任何类型的异常。要实现此目的，请在 `catch` 的括号之间添加 **省略号 (...)**：

```

try {
    // code
} catch(...) {
    // code to handle exceptions
}

```

```
}
```

8.7.使用文件

✧ 使用文件

- 另一个有用的 C++ 功能是读取和写入文件的能力。这需要名为 **fstream** 的标准 C++ 库。

- **fstream** 中定义了三种新数据类型：

ofstream：输出文件流，用于创建信息并将信息写入文件。

ifstream：从文件中读取信息的输入文件流。

fstream：通用文件流，具有允许其创建，读取和写入文件信息的 **ofstream** 和 **ifstream** 功能。

- 要在 C++ 中执行文件处理，头文件 **<iostream>** 和 **<fstream>** 必须包含在 C++ 源文件中。

```
#include <iostream>
```

```
#include <fstream>
```

这些类直接或间接地从类 **istream** 和 **ostream** 派生。我们已经使用了类型为这些类的对象：**cin** 是类 **istream** 的对象，**cout** 是类 **ostream** 的对象。

✧ 打开文件

- 文件在打开后，才能从中读取或写入文件。
- 可以使用 **ofstream** 或 **fstream** 对象来打开文件进行写入。
- 让我们打开一个名为“**test.txt**”的文件并写一些内容：

```
#include <iostream>
```

```
#include <fstream>
```

```
using namespace std;
```

```
int main() {
```

```
    ofstream MyFile;
```

```
    MyFile.open("test.txt");
```

```
    MyFile << "Some text. \n";
```

```
}
```

- 上面的代码创建了一个名为 **MyFile** 的 **ofstream** 对象，并使用 **open()** 函数打开文件系统上的“test.txt”文件。如您所见，使用相同的流输出运算符写入文件。如果指定的文件不存在，**open** 函数将自动创建它。

✧ 打开文件

- 使用完文件后，使用成员函数 **close()** 关闭它。

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

int main() {
    ofstream MyFile;
    MyFile.open("test.txt");

    MyFile << "Some text! \n";
    MyFile.close();
}
```

- 运行此代码将导致在项目目录中使用“Some text!”创建“test.txt”文件。写在里面。您还可以选择在 **open** 函数中指定文件的路径，因为它可以位于项目之外的位置。

8.8.更多关于文件

✧ 使用文件

- 您还可以使用 **ofstream** 对象构造函数提供文件的路径，而不是调用 **open** 函数。

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

int main() {
    ofstream MyFile("test.txt");

    MyFile << "This is awesome! \n";
    MyFile.close();
}
```

与 **open** 函数一样，您可以提供位于不同目录中的文件的完整路径。

✧ 使用文件

- 在某些情况下，例如当您没有文件权限时，**open** 函数可能会失败。
- **is_open()**成员函数检查文件是否已打开并准备好进行访问。

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

int main() {
    ofstream MyFile("test.txt");

    if (MyFile.is_open()) {
        MyFile << "This is awesome! \n";
    }
    else {
        cout << "Something went wrong";
    }
    MyFile.close();
}
```

is_open()成员函数检查文件是否已打开并准备好进行访问。

✧ 文件打开模式

• **open** 函数的可选第二个参数定义了打开文件的模式。此列表显示支持的模式。

Mode Parameter	Meaning
ios::app	append to end of file
ios::ate	go to end of file on opening
ios::binary	file open in binary mode
ios::in	open file for reading only
ios::out	open file for writing only
ios::trunc	delete the contents of the file if it exists

- 可以使用按位运算符 **OR(|)**组合所有这些标志。
- 例如，要以写入模式打开文件并截断它，如果它已经存在，请使用以下语法：

```
ofstream outfile;
outfile.open("file.dat", ios::out | ios::trunc );
```

从文件中读取

- 您可以使用 **ifstream** 或 **fstream** 对象从文件中读取信息。

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

int main () {
    string line;
    ifstream MyFile("test.txt");
    while ( getline (MyFile, line) ) {
        cout << line << '\n';
    }
    MyFile.close();
}
```

- **getline** 函数从输入流中读取字符并将它们放入字符串中。

上面的示例读取文本文件并将内容打印到屏幕上。
我们的 **while** 循环使用 **getline** 函数逐行读取文件。

CERTIFICATE

Issued 15 November, 2017

This is to certify that

Vincent

has successfully completed the

C++ Tutorial course



Yeva Hyusyan
Chief Executive Officer