

C++第一天课堂笔记

一、C++概述

1.1 C++简介

c++起初也叫“c with class”.通过名称表明，c++是对C的扩展，因此c++是C语言的超集。

任何有效的C程序都是有效的c++程序。c++程序可以使用已有的C程序库。

c++语言在C语言的基础上添加了面向对象编程和泛型编程的支持。

c++继承了C语言高效、简洁、快速和可移植的传统（特性）。

C++编程方式：

- C的面向过程编程
- C++的面向对象编程 【重点】
- c++模板的泛型编程 【重点】

C语言和C++语言的关系：

C++语言是在C语言的基础上，添加了面向对象、模板等现代程序设计语言的特性而发展起来的。两者无论是从语法规则上，还是从运算符的数量和使用上，都非常相似，所以我们常常将这两门语言统称为“C/C++”

C语言和C++并不是对立的竞争关系：

- 1.C++是C语言的加强，是一种更好的C语言。
- 2.C++是以C语言为基础的，并且完全兼容C语言的特性

1.2 可移植性和标准

可移植性：开发的C/C++的源代码,可以在任何安装C/C++编译器的操作系统，可以重新编译，即可运行。

为什么存在C++编译标准？

程序是否可移植性有两个问题需要解决。
第一是硬件，针对特定硬件编程的程序是不可移植的；
第二，语言的实现，windows xp c++ 和 Redhat Linux 或 Mac OS X 对C++的实现不一定相同。虽然我们希望C++版本与其他版本兼容，但是如果没有一个公开的标准，很难做到。

标准：

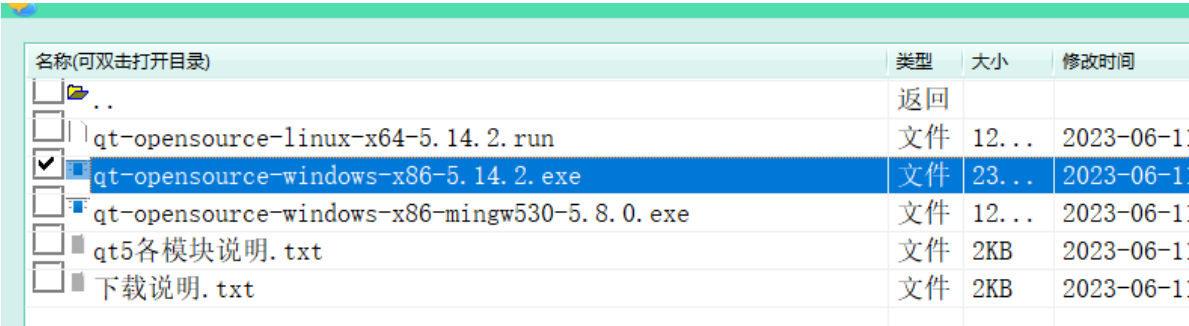
- ANSI/ISO C++
- ISO/IEC 14882:1998，简称C++98
 - 已有的C++特性
 - 对语言进行了扩展，添加了异常、运行阶段类型识别(RTTI)、模板和标准模板库(STL)

- ISO/IEC 14882:2003：只是修复C++98的bug（错误），没有大的改动。因此 C++98也称之为 C++98/c++2003。
- ISO/IEC 14882:2011： c++11

二、c++初始

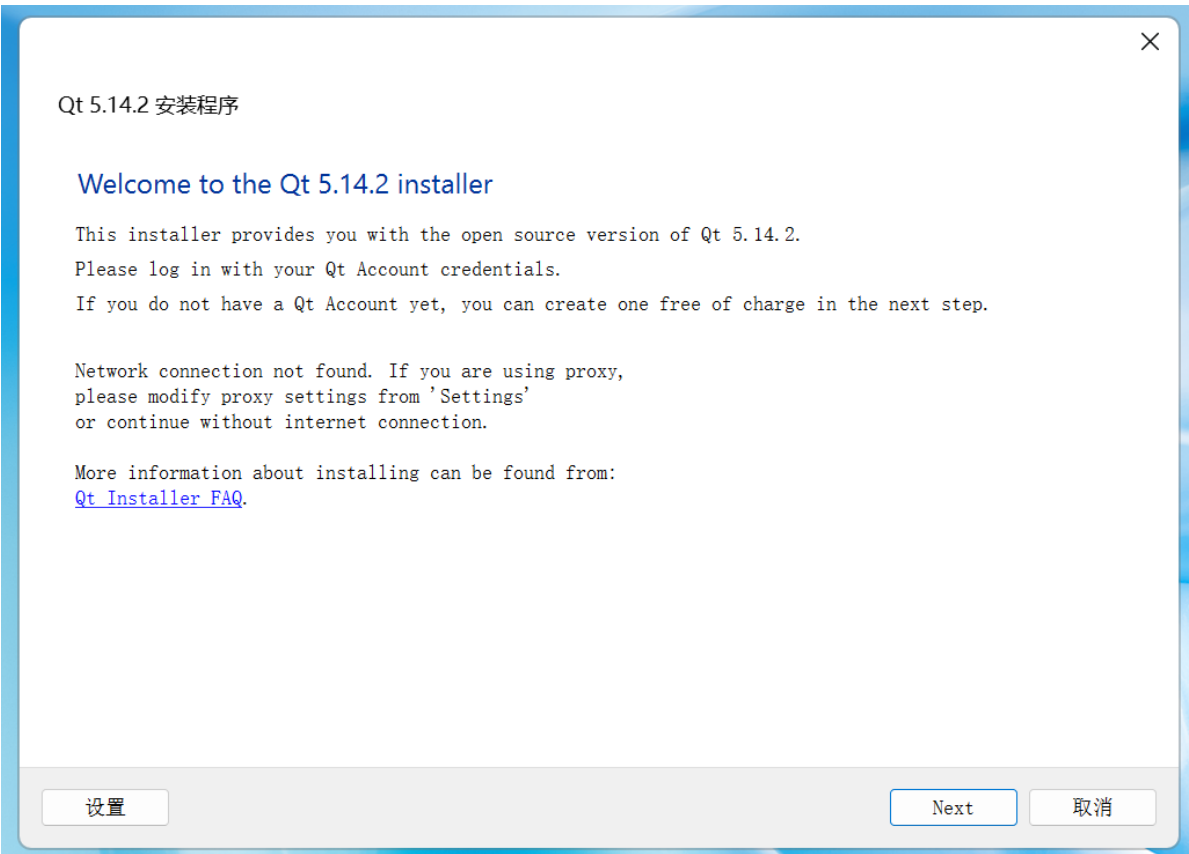
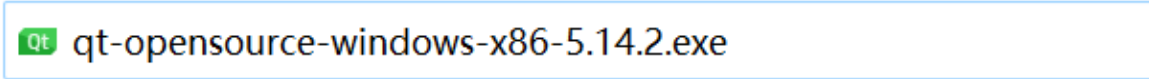
2.1 Qt安装

下载位置： 飞秋共享-> 01-应用开发工具 -> Qt



| 名称(可双击打开目录) | 类型 | 大小 | 修改时间 |
|--|----|-------|------------|
| .. | 返回 | | |
| qt-opensource-linux-x64-5.14.2.run | 文件 | 12... | 2023-06-10 |
| qt-opensource-windows-x86-5.14.2.exe | 文件 | 23... | 2023-06-10 |
| qt-opensource-windows-x86-mingw530-5.8.0.exe | 文件 | 12... | 2023-06-10 |
| qt5各模块说明.txt | 文件 | 2KB | 2023-06-10 |
| 下载说明.txt | 文件 | 2KB | 2023-06-10 |

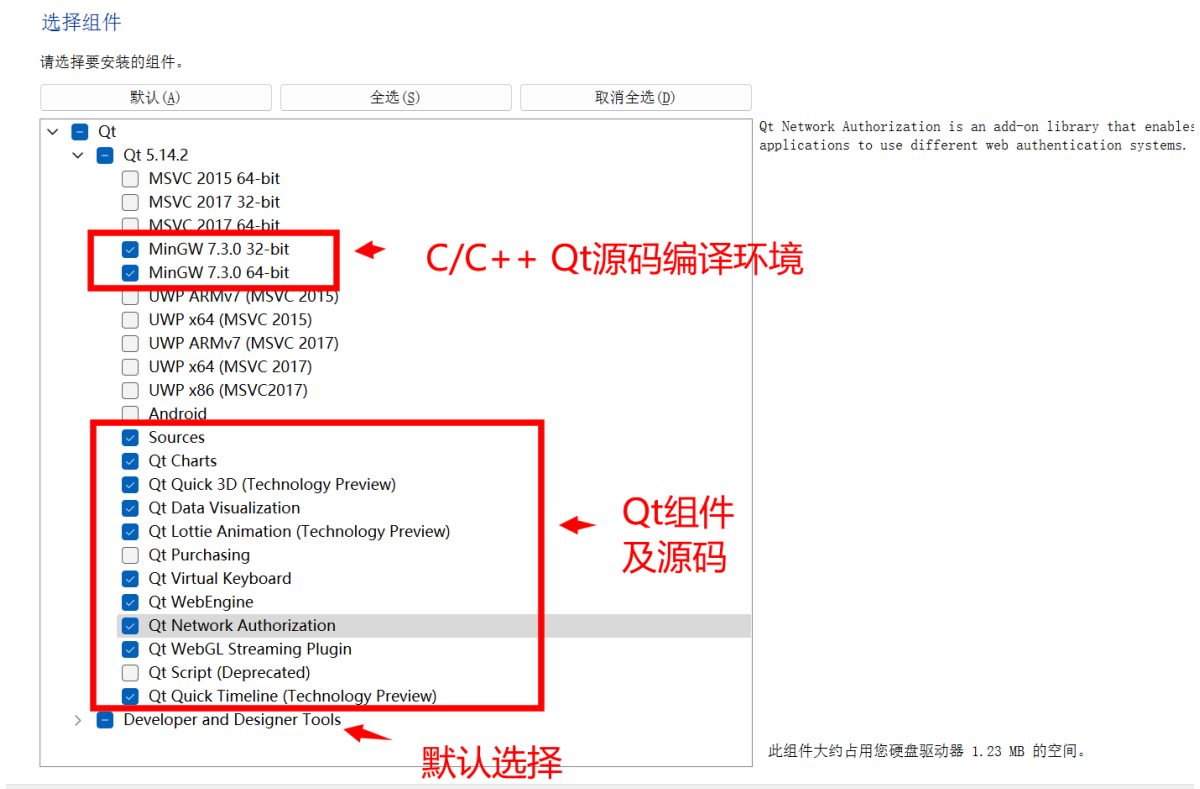
开始安装之前，需要关闭网络连接，再双击"qt-opensource-windows-x86-5.14.2.exe"



出现了上面的欢迎安装界面时，就可以恢复网络了。点击【下一步】，再点击【下一步】



确认安装路径之后，点击【下一步】



点击【下一步】

许可协议

请阅读以下许可协议。 您必须接受这些协议中的条款才能继续安装。

Qt Installer LGPL License Agreement

GPLv3 with Qt Company GPL Exception License Agreement

Python Software Foundation License Version 2

GENERAL

Qt is available under a commercial license with various pricing models and packages that meet a variety of needs. Commercial Qt license keeps your code proprietary where only you can control and monetize on your end product's development, user experience and distribution. You also get great perks like additional functionality, productivity enhancing tools, world-class support and a close strategic relationship with The Qt Company to make sure your product and development goals are met.

Qt has been created under the belief of open development and providing freedom and choice to developers. To support that, The Qt Company also licenses Qt under open source licenses, where most of the functionality is available under LGPLv3. It should be noted that the tools as well as some add-on components are available only under GPLv3. In order to preserve the true meaning of open development and uphold the spirit of free software, it is imperative that the rules and regulations of open source licenses are followed. If you use Qt under open-source licenses, you need to make sure that you comply with all the licenses of the components you use.

Qt also contains some 3rd party components that are available under different open-source licenses. Please refer to the documentation for more details on 3rd party licenses used in Qt.

GPLv3 and LGPLv3

GNU LESSER GENERAL PUBLIC LICENSE

The Qt Toolkit is Copyright (C) 2017 The Qt Company Ltd.
Contact: <https://www.qt.io/licensing>

You may use, distribute and copy the Qt GUI Toolkit under the terms of GNU Lesser General Public License version 3, which supplements GNU General

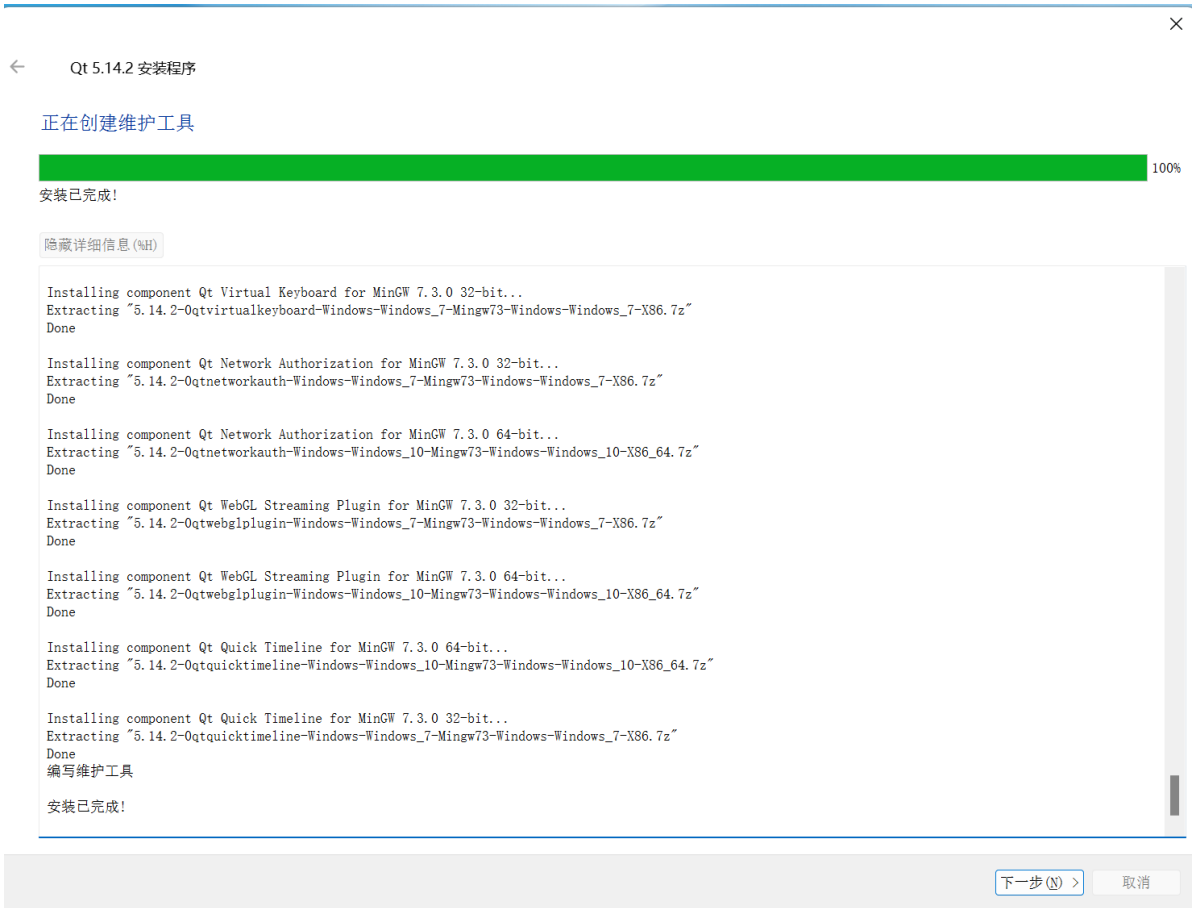
- ☒ I have read and agree to the terms contained in the license agreements. **同意**
- ☐ I do not accept the terms and conditions of the above license agreements.

点击【下一步】，再点击【下一步】，最后点击【安装】

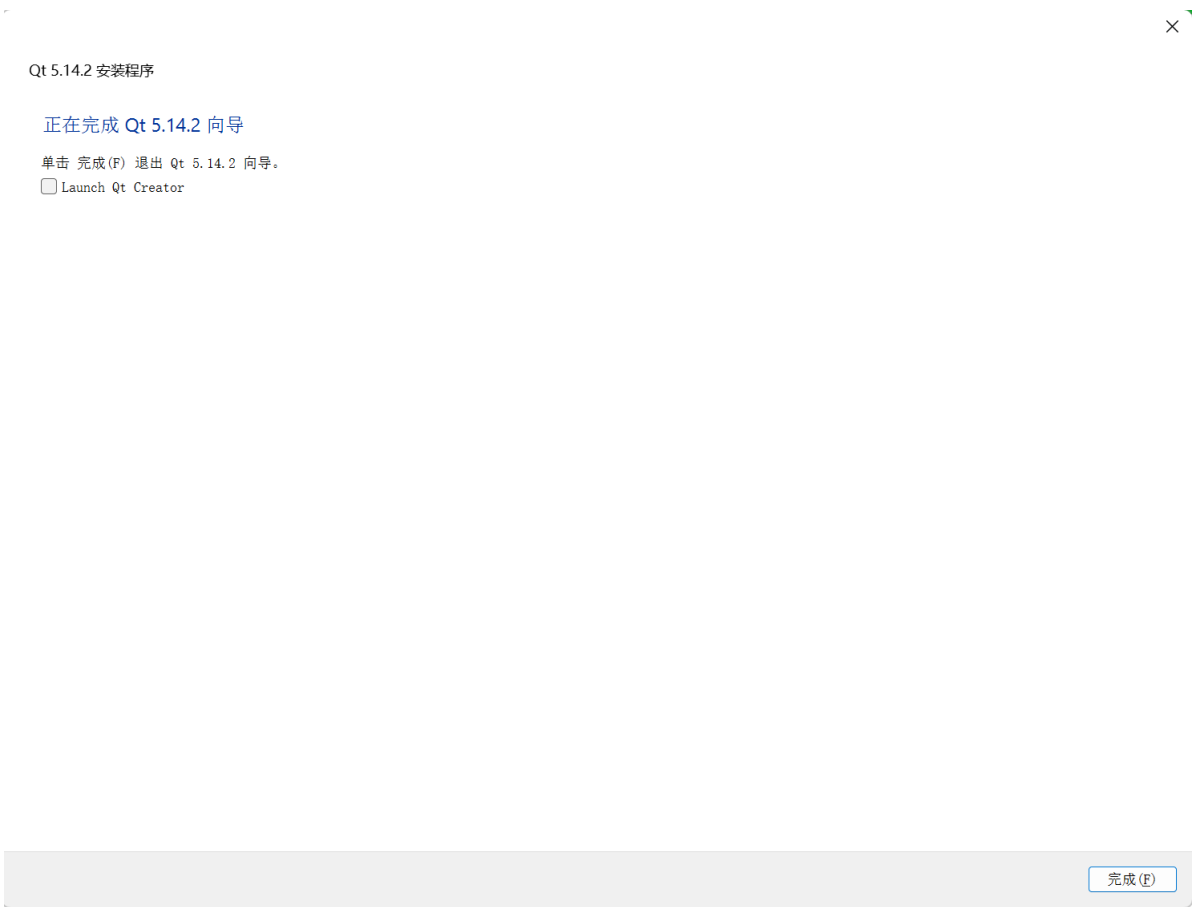
正在安装 Qt 5.14.2

正在准备安装...

显示详细信息(%S)



安装完成，点击【下一步】

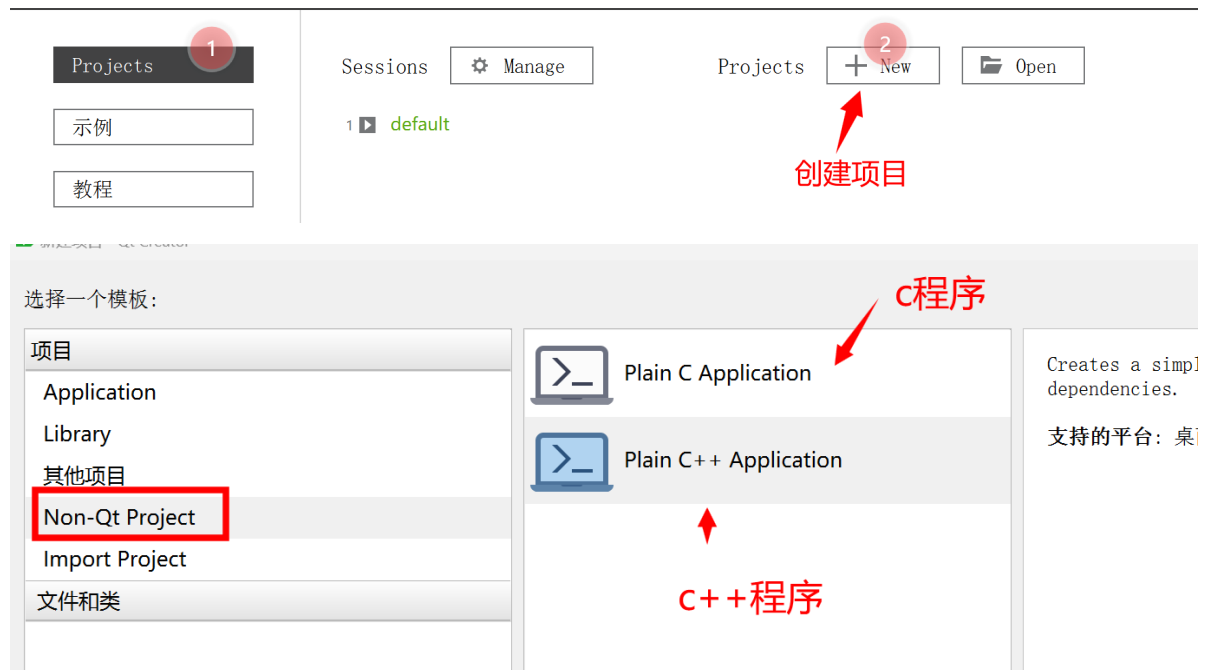


最后，点击【完成】关闭Qt安装向导程序。

2.2 Qt创建HelloWorld

打开Qt Creator工具（按Win 键，搜索Qt，找到Qt Creator4.11.1）

创建工程（项目）：

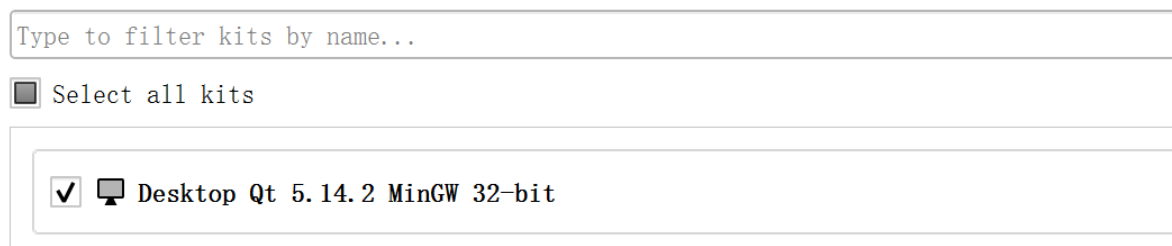


Define Build System

Build system: qmake

Kit Selection

The following kits can be used for project **weekldemo**:



Project Management

作为子项目添加到项目中: <None>

添加到版本控制系统(V): <None> Configure...

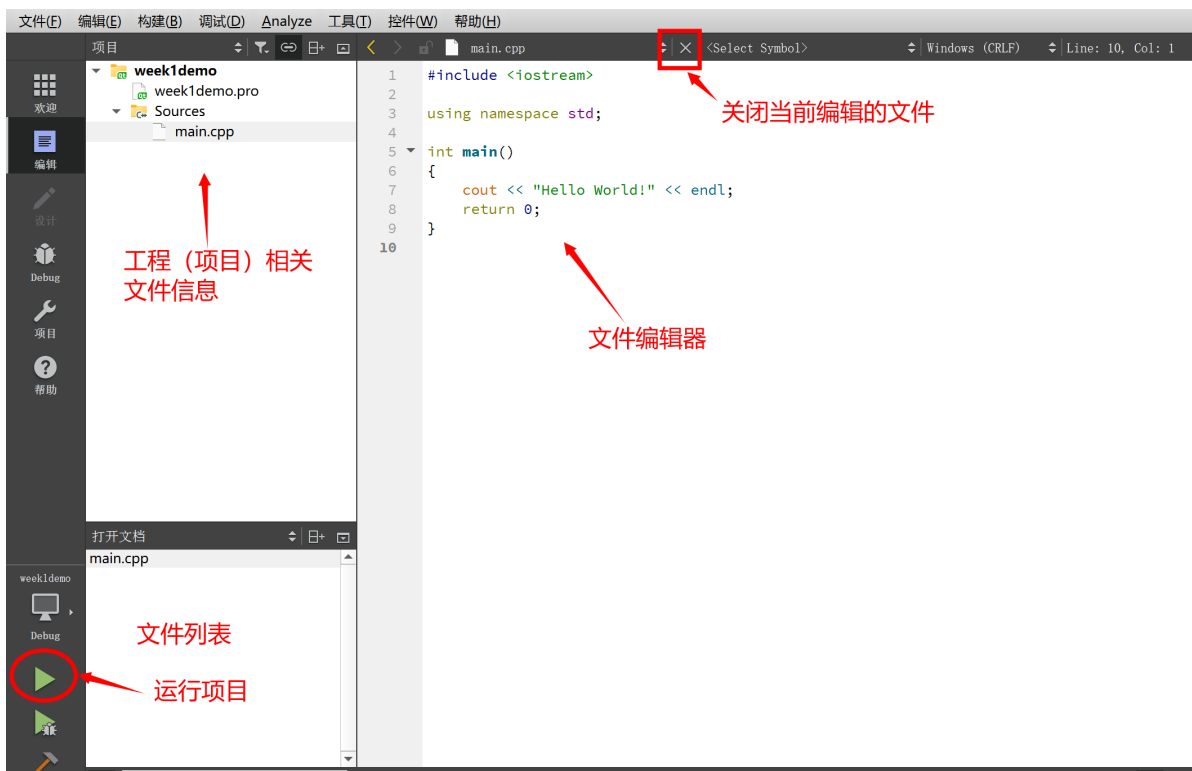
要添加的文件

D:\iot\XA2301\code2\week1demo:

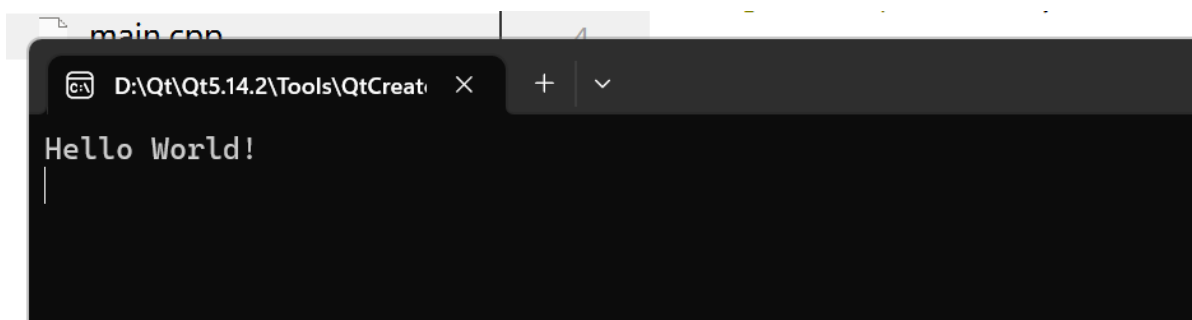
main.cpp
week1demo.pro

完成(F)

取消



点击 运行项目的按钮, 运行整个工程(项目)。



2.3 分析C++的helloworld文件

```
#include <iostream>

using namespace std;

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

第一行：引入 `<iostream>` 头文件, 此文件包含std命名空间, 和标准的输入和输出。

第二行： `using namespace std;` 使用标准的命名空间, 命名空间中定义标准的变量。

第四行： `cout` 输出对象, 是std命名空间的, 也可以写成 `std::cout` , `cout` 和 `<<` 运算符组合使用, 表示输出什么内容。如"Hello World!", `endl` 表示换行符。

| 头文件类型 | 约定 | 示例 | 说明 |
|---------|--------------|------------|-------------------------------------|
| c++旧式风格 | 以.h 结尾 | iostream.h | c++程序可用 |
| c 旧式风格 | 以.h 结尾 | math.h | c/c++程序可用 |
| c++新式风格 | 无扩展名 | iostream | c++程序可用, 使用 namespace std |
| 转换后的 c | 加上前缀 c, 无扩展名 | cmath | c++程序可用, 可使用非 c 特性, 如 namespace std |

将main.cpp的文件内容, 修改如下:

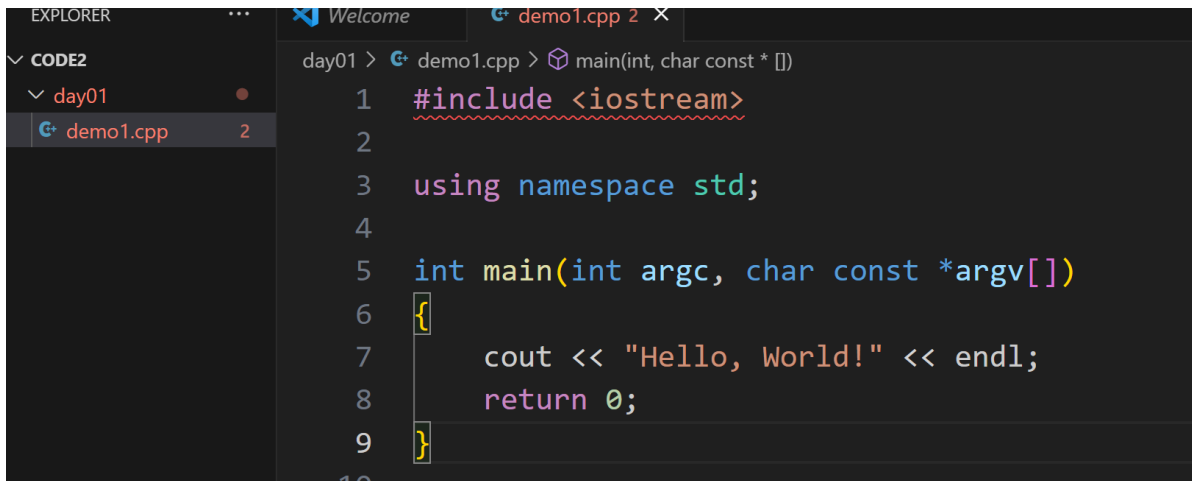
```
#include <stdio.h>

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

代码可以正常运行, 说明 C++兼容C的语法。

2.4 Linux下开发C++程序

VSCode 和 g++编译器



```
1 #include <iostream>
2
3 using namespace std;
4
5 int main(int argc, char const *argv[])
6 {
7     cout << "Hello, World!" << endl;
8     return 0;
9 }
```

打开vscode的终端 (ctrl+` 或 ctrl+j) : 检查g++是否存在, 以及它的版本号

```
disen@qfxa:~/code2/day01$ which g++
/usr/bin/g++
disen@qfxa:~/code2/day01$ g++ --version
g++ (Ubuntu 5.3.1-14ubuntu2) 5.3.1 20160413
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

如果没有g++编译器, 则需要手动安装: `sudo apt install g++ -y`

编译cpp源文件, 并执行目标程序

```
disen@qfxa:~/code2/day01$ g++ demo1.cpp
disen@qfxa:~/code2/day01$ ls
a.out  demo1.cpp
disen@qfxa:~/code2/day01$ ./a.out
Hello, World!
disen@qfxa:~/code2/day01$
```

2.5 面向过程和面向对象

2.5.1 面向过程

面向过程编程思想的核心: 功能分解, 自顶向下, 逐层细化 (程序=数据结构+算法)

面向过程编程语言存在的主要缺点是不符合人的思维习惯, 而是要用计算机的思维
方式去处理问题, 而且面向过程编程语言重用性低、维护困难。

2.5.2 面向对象

面向对象编程 (Object-Oriented Programming) 以下简称 OOP。

在面向对象中, 算法与数据结构被看做是一个整体, 称作对象, 现实世界中任何类的对象都具有一定的属性和操作, 也总能用数据结构与算法两者合一地来描述, 所以可以用下面的等式来定义对象和程序:

对象 = 算法（方法或功能函数） + 数据结构（数组、结构体...）
程序 = 对象 + 对象 +

面向对象编程思想的核心：应对变化、提高复用。

2.5.3 面向对象的三大特征 【重要】

- 1) 封装性： 可以将客观的事务抽象成类，类中封装一些数据（属性-特征）和方法（行为），可以根据需要可以设置数据或方法的访问权限，对于一些重要的数据设置不可访问。
- 2) 继承性： 继承所表达的是类之间相关的关系，这种关系使得对象可以继承另外一类对象的特征和能力。**继承的作用**：避免公用代码的重复开发，减少代码和数据冗余。
- 3) 多态性： 多态性可以简单地概括为“一个接口，多种方法”，字面意思为多种形态。多用于描述一个对象的多种形态，使用多态性必须具有继承和重写方法的前提。

三、C++对C的扩展

3.1 ::作用域运算符

可用::对被屏蔽的同名的全局变量进行访问。

如：

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

int x = 200;
int main(int argc, const char **argv)
{
    int x = 100;           // 局部变量
    cout << "x=" << x << endl; // 优先全局变量，屏蔽了同名的全局变量
    cout << "全局的x=" << ::x << endl;
    return 0;
}
```

```
disen@qfxa:~/code2/day01$ ./a.out
x=100
全局的x=200
disen@qfxa:~/code2/day01$
```

3.2 名字控制

在大的工程中，存在多个源文件或类，所以可能存在重名的情况。在C++提供不同方式，使得同一个名字可用具有不同的数据或功能。

3.2.1 C++命名空间(namespace)

为了避免，在大规模程序的设计中，以及在程序员使用各种各样的 C/C++ 库时，这些标识符的命名发生冲突，标准 C++ 引入关键字 namespace（命名空间/名字空间/名称空间），可以更好地控制标识符的作用域。

3.2.2 namespace用法

【注意】

- 1) 命名空间只能全局范围内定义，不能在局部定义。
- 2) 命名空间可嵌套命名空间
- 3) 声明和实现可分离
- 4) 无名的命名空间只能在当前文件中使用。
- 5) 命名空间可以存在别名。

定义namespace的语法：

```
namespace [名称] {  
    成员的声明，可以具有初始化的值；  
    [  
        内部嵌套 namespace {};  
    ]  
    [声明成员函数]  
    [定义成员函数]  
}  
  
// 实现namespace中声明的函数  
函数返回值类型 命名空间的名称::声明的函数名(形参列表){  
  
}
```

【无命名空间】

无名命名空间，意味着命名空间中的标识符只能在本文件内访问，相当于给这个标识符加上了 `static`，使得其可以作为内部连接。

如1：

```
#include <iostream>  
using namespace std;  
  
namespace A  
{  
    // 声明变量成员，可以指定初始化。  
    int x = 10;  
}  
  
namespace B  
{  
    int x = 20;  
}  
  
namespace C
```

```

{
    namespace A
    {
        int x = 100;
    }
    namespace B
    {
        int x = 200;
    }
}

namespace D
{
    int x = 1;
    void show_x() // 定义函数功能
    {
        cout << "D namespace x: " << x << endl;
    }
    int add(int y); // 声明函数
}

int D::add(int y)
{
    // 可以访问当前命名空间的所有成员
    return x + y;
}

int main(int argc, char const *argv[])
{
    cout << "A x=" << A::x << endl;
    cout << "B x=" << B::x << endl;
    cout << "C-A x=" << C::A::x << endl;
    cout << "C-B x=" << C::B::x << endl;
    // 调用D的show_x()函数
    D::show_x();
    // 调用D的add(int) 函数
    int ret = D::add(100);
    cout << "D::add(100)=" << ret << endl;
    return 0;
}

```

```

disen@qfxa:~/code2/day01$ g++ demo03.cpp
disen@qfxa:~/code2/day01$ ./a.out
A x=10
B x=20
C-A x=100
C-B x=200
D namespace x: 1
D::add(100)=101

```

如2：无名的命名空间

```
#include <iostream>
```

```
using namespace std;

namespace
{
    int x = 10;
}

int main(int argc, char const *argv[])
{
    cout << "x=" << x << endl; // 当前文件的任一位置都可以访问的
    return 0;
}
```

3.2.3 using 声明

using 声明可使得指定的标识符（函数、变量、结构体、枚举、类）可用。

语法：

using 命名空间的名称::成员名; 当前位置向下，可以直接使用成员名，即为命名空间的成员。

【注意】使用了命名空间之后，可能存在名称冲突。平时使用时注意using和其它同名标识符先后顺序。

如：

```
#include <iostream>

using namespace std;

namespace A
{
    int a = 10;
}
namespace B
{
    int a = 20;
}
namespace C
{
    int a = 30;
}

int main(int argc, char const *argv[])
{
    using namespace A;
    cout << "A a=" << a << endl; // 搜索a变量，先从局部，再从引入的空间中
    using B::a; // 将B空间的变量a，引入到当前区域
    cout << "B a=" << a << endl;
    // // 名称冲突
    // int a = 100; // 当前区域已存在了 a
    // cout << "a=" << a << endl;
    using namespace C;
    cout << "C a=" << C::a << endl;

    return 0;
}
```

```
}
```

using 声明碰到函数重载

using 一次函数名，即将所有的相同名的重载函数都引入。

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

// 函数的重载，只和函数名、参数列表相关。
// 函数名相同，参数列表不同（个数、类型、顺序），构成函数的重载
namespace A
{
    int add(int a, int b) { return a + b; }
    float add(int a, float b) { return a * 2.0f + b; }
    double add(int a, double b) { return a * 3.0 + b; }
    int add(int a, int b, int c) { return a + b + c; }
}

int main(int argc, char const *argv[])
{
    using A::add;
    // 调用重载的函数时，依据传入数据的类型确认调用的是哪一个函数
    cout << "add(1, 1.5f)=" << add(1, 1.5f) << endl;
    cout << "add(1, 1.5)=" << add(1, 1.5) << endl;
    cout << "add(1, 3)=" << add(1, 3) << endl;
    return 0;
}
```

```
disen@qfxa:~/code2/day01$ ./a.out
add(1, 1.5f)=3.5
add(1, 1.5)=4.5
add(1, 3)=4
```

3.2.4 using 编译指令

using 编译指令使整个命名空间标识符可用

【注意】使用 using 声明或 using 编译指令会增加命名冲突的可能性。也就是说，如果有名称空间，并在代码中使用作用域解析运算符，则不会出现二义性。

语法：

```
using namespace 命名空间的名称;    当前位置向下，可以直接访问命名空间中所有的成员。
```

3.2.5 命名空间使用

需要记住的关键问题是当引入一个全局的 using 编译指令时，就为该文件打开了该命名空间，它不会影响任何其他文件，所以可以在每一个实现文件中调整对命名空间的控制。

比如，如果发现某一个实现文件中有太多的 using 指令或声明而产生的命名冲突，就要对该文件做个简单的改变，通过明确的限定或者 using 声明来消除名字冲突，这样不需要修改其他的实现文件。

3.3 全局变量检测增强

在C++中全局变量无论是声明还是定义，只能出现一次。

因此，在c++中定义全局变量时，无初始化值时，默认值为0。

c 语言代码：

```
int a = 10; //赋值, 当做定义
int a; //没有赋值, 当做声明

int main(){
    printf("a:%d\n",a);
    return EXIT_SUCCESS;
}
```

此代码在 c++ 下编译失败,在 c 下编译通过.

3.4 C++中所有的变量和函数都必须有类型

C的自定义函数时，形参变量可以没有数据类型，即为任意类型

c++中，函数的形参变量必须指定类型。

如：C的程序

```
#include <stdio.h>

void show(x)
{
    printf("x=%d\n", (int)x);
}

int main(int argc, char const *argv[])
{
    show(20);
    show(15.0);
    return 0;
}
```

```
disen@qfxa:~/code2/day01$ gcc demo7.c
demo7.c: In function 'show':
demo7.c:3:6: warning: type of 'x' defaults to 'int' [-Wimplicit-int]
void show(x)
    ^
disen@qfxa:~/code2/day01$ ./a.out
x=20
x=1
disen@qfxa:~/code2/day01$
```

如：c++的程序

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

```

void show(x)
{
    cout << "x=" << (int)x << endl;
}

int main(int argc, char const *argv[])
{
    show(20);
    show(15.0);
    return 0;
}

```

PROBLEMS 16 OUTPUT DEBUG CONSOLE TERMINAL

```

demo7.cpp:4:12: error: variable or field 'show' declared void
void show(x)
    ^
demo7.cpp:4:11: error: 'x' was not declared in this scope
void show(x)
    ^
demo7.cpp: In function 'int main(int, const char**)':
demo7.cpp:11:12: error: 'show' was not declared in this scope
    show(20);
    ^

```

修改cpp内容:

```

#include <iostream>
using namespace std;

void show(int x)
{
    cout << "x=" << x << endl;
}

int main(int argc, char const *argv[])
{
    show(20);
    show(15.0);
    return 0;
}

```

```

disen@qfxa:~/code2/day01$ g++ demo7.cpp
disen@qfxa:~/code2/day01$ ./a.out
x=20
x=15

```


3.5 更严格的类型转换

在C++中，不同类型的变量之间赋值时，需要明确的类型转换。

如：demo8.cpp

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

int main(int argc, char const *argv[])
{
    // 按c的方式自动转换
    int a = 129;
    char b = a;
    cout << "b=" << (int)b << endl;

    // c++必须强转（明确数据类型）
    // 在C中没有问题
    char *p = malloc(32); // 报错， 修改为 char *p=(char *)malloc(32); 即可
    strcpy(p, "disen");
    cout << "p=" << p << endl;
    return 0;
}
```

demo8.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char const *argv[])
{
    int a = 129;
    char b = a;
    printf("b = %d\n", b);

    char *p = malloc(32);
    strcpy(p, "disen");
    printf("%s\n", p);
    return 0;
}
```

3.6 struct 类型加强

在C++中，定义结构体变量时，不用加struct。

如：c

```

struct S {
    int id;
    char name[30];
};

void test1(){
    struct S s1={1, "disen"};
}

```

在c++中

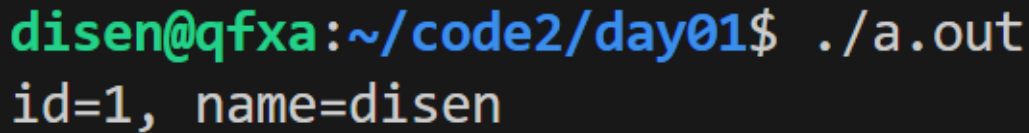
```

#include <iostream>
using namespace std;

struct S
{
    int id;
    char name[30];
};

int main(int argc, char const *argv[])
{
    // struct S s1 = {1, "disen"};
    S s1 = {1, "disen"};
    cout << "id=" << s1.id << ", name=" << s1.name << endl;
    return 0;
}

```



```

disen@qfxa:~/code2/day01$ ./a.out
id=1, name=disen

```

3.7 新增"bool 类型关键字

bool 类型只有两个值，true(非0值)，false(0 值)

bool 类型占 1 个字节大小(操作系统最小处理数据单位是字节)。

如：

```

#include <iostream>

using namespace std;

int main(int argc, char const *argv[])
{
    // bool flag = true;
    // bool flag = false;
    bool flag = -1; // 非0都为真(true)
    if (flag)
    {
        cout << "Disen 666" << endl;
    }
    else
    {

```

```

    cout << "小马, 666" << endl;
}
return 0;
}

```

```

disen@qfxa:~/code2/day01$ ./a.out
Disen 666

```

3.8 三目运算符功能增强

C中三目运算表达返回是变量的值，而c++中，三目运算表达式返回是变量。

如：

```

int a=10,b=20;
(a>b?a:b) = 100;
cout << "a=" << a << " b=" << b << endl;

```

```

disen@qfxa:~/code2/day01$ gcc demo11.c
demo11.c: In function 'main':
demo11.c:4:21: error: lvalue required as left operand of assignment
    (a > b ? a : b) = 100;
                    ^
disen@qfxa:~/code2/day01$

```

```

disen@qfxa:~/code2/day01$ g++ demo11.cpp
disen@qfxa:~/code2/day01$ ./a.out
a=10 b=100
disen@qfxa:~/code2/day01$

```

3.9 C/C++中的 const

const修饰的变量，只能读取，不能修改其值。

C/C++中 const 的区别：

原则上：c中的const变量，不能做为数组的个数使用，因为数组分配的空间可能发生变化（扩大，缩小）。

gcc 99/g++11编译器：const变量可以作为数组的个数使用。

C中的const变量，在定义时，则会在内存中创建空间，而c++不会创建，只有取一个 const 地址，或者把它定义为 extern,则会为该 const 创建内存空间。

如：

```

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

int main(int argc, char const *argv[])
{
    const int n = 10; // n 常量标识名，初始化定义时，不会在栈中创建空间
    const int m = 10;
    printf("n addr = %p\n", &n); // 一量取const变量的地址，则会在内存中开辟空间
    printf("m addr = %p\n", &m);
    return 0;
}

```

在c++中，出现在所有函数之外的 const 作用于整个文件(也就是说它在该文件外不可见)，默认为内部连接，c++中其他的标识符一般默认为外部连接（指向是常量区）。

如：默认情况下，const初始化值为常量，不会开辟空间，只是存储在符号表中。

```

#include <iostream>
using namespace std;

int main(int argc, char const *argv[])
{
    const int x = 10; // 在符号表存储，即没有在栈中开辟空间
    int *p = (int *)&x; // 在栈中开辟空间，p指针指向，空间的值是x符号表的数据
    *p = 100; // 修改的是栈空间的数据，不影响符号表
    cout << "x=" << x << endl;
    cout << "*p=" << *p << endl;
    return 0;
}

```

如2： 如果const变量初始化值为一个变量时，则会在内存中开辟空间

```

#include <iostream>
using namespace std;

int main(int argc, char const *argv[])
{
    int x = 20;
    const int X = x; // 在栈中开辟空间
    int *p = (int *)&x; // p指针指向的x开辟的空间地址
    *p = 100; // 修改的是栈空间的数据，x 也被修改了
    cout << "x=" << x << endl;
    cout << "X=" << X << endl;
    cout << "*p=" << *p << endl;
    return 0;
}

```

```
disen@qfxa:~/code2/day01$ ./a.out
x=20
X=100
*p=100
disen@qfxa:~/code2/day01$
```

对于自定数据类型，比如类对象，那么也会分配内存。

```
const Person person; //未初始化 age
//person.age = 50; //不可修改
Person* pPerson = (Person*)&person;
//指针间接修改
pPerson->age = 100;
cout << "pPerson->age: " << pPerson->age << endl;
pPerson->age = 200;
cout << "pPerson->age: " << pPerson->age << endl;
```

尽量以 `const` 替换 `#define`

`const` 和 `#define` 区别总结:

1. `const` 有类型，可进行编译器类型安全检查。
`#define` 无类型，不可进行类型检查。
2. `const` 有作用域，而 `#define` 不重视作用域，默认定义处到文件结尾。
如果定义在指定作用域下有效的常量，那么 `#define` 就不能。