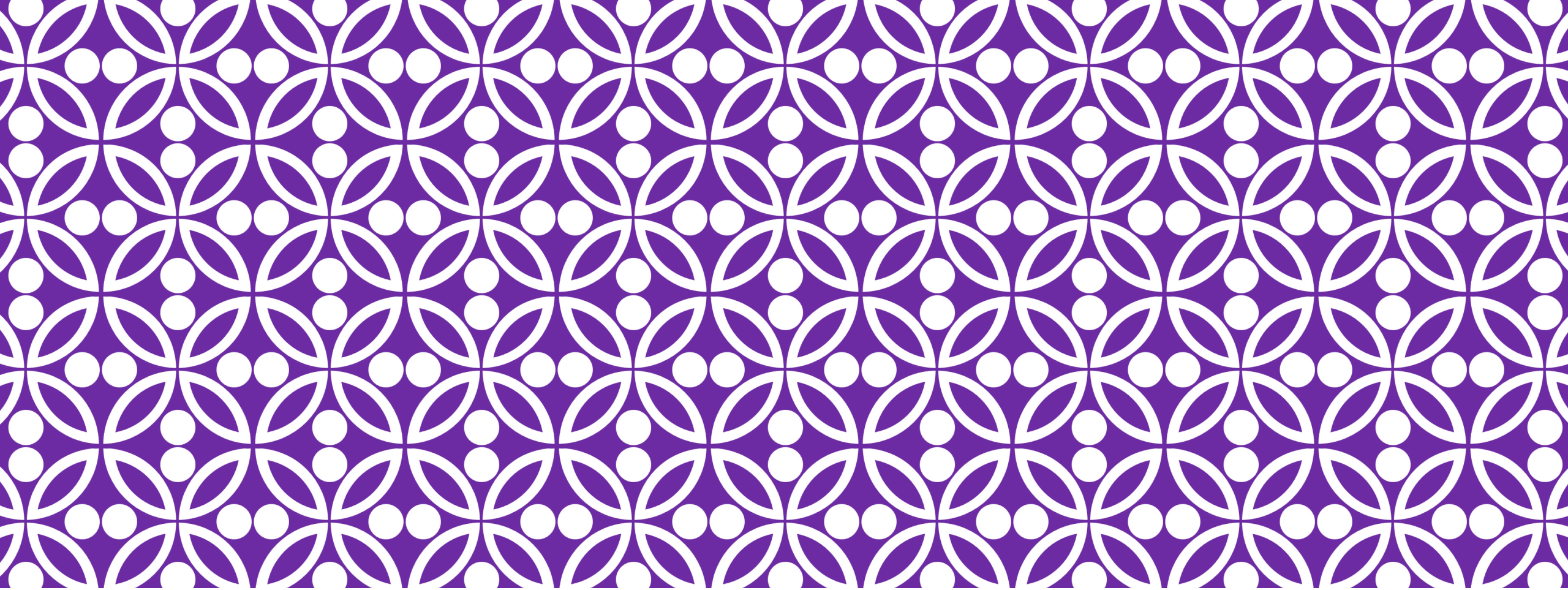
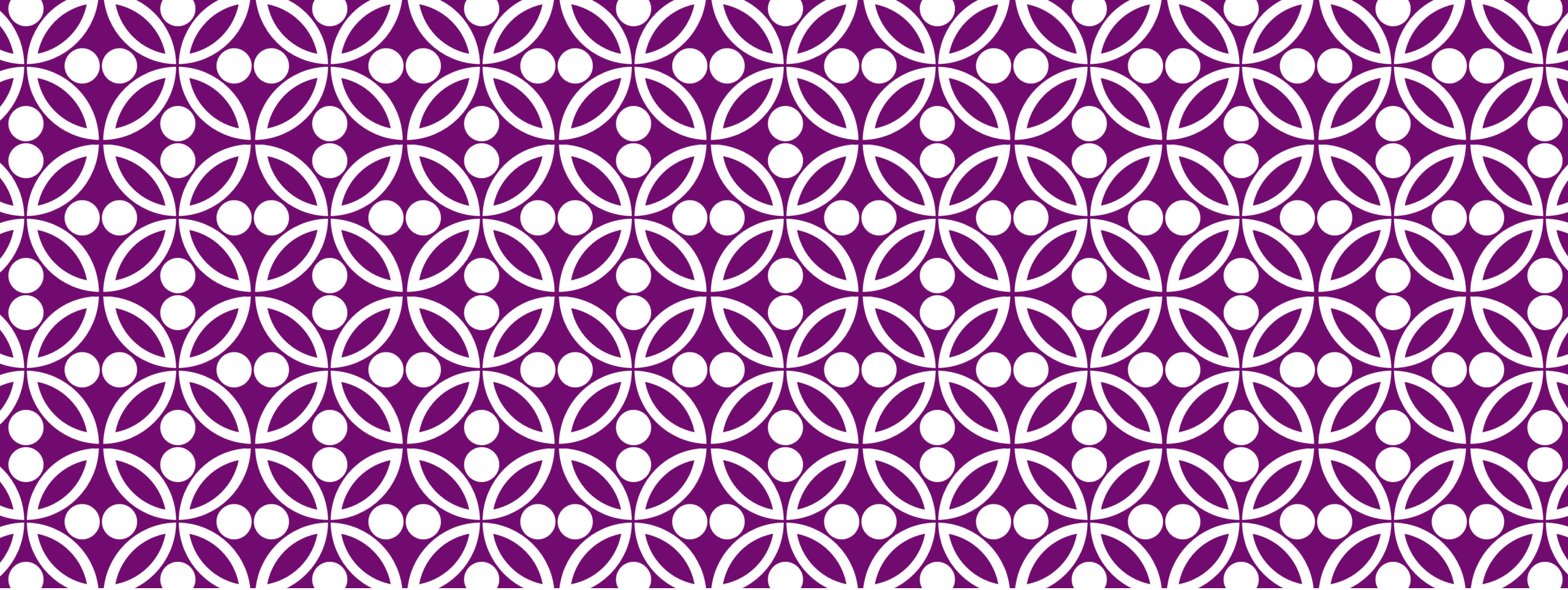


C++ 程序设计入门

HKU Astar
2022年9月24日



开发环境

清单

- g++
- make
- cmake
- 文本编辑器/IDE
- 命令行终端

查看现存安装版本

```
$ g++ --version
g++ (GCC) 12.1.0
...
$ make --version
GNU Make 4.3
...
$ cmake --version
cmake version 3.23.2
...
```

Windows

- TDM-GCC: https://jmeubank.github.io/tdm-gcc/
- Nuwen's MinGW Distro: https://nuwen.net/mingw.html

- CMake: https://cmake.org/download/

Linux

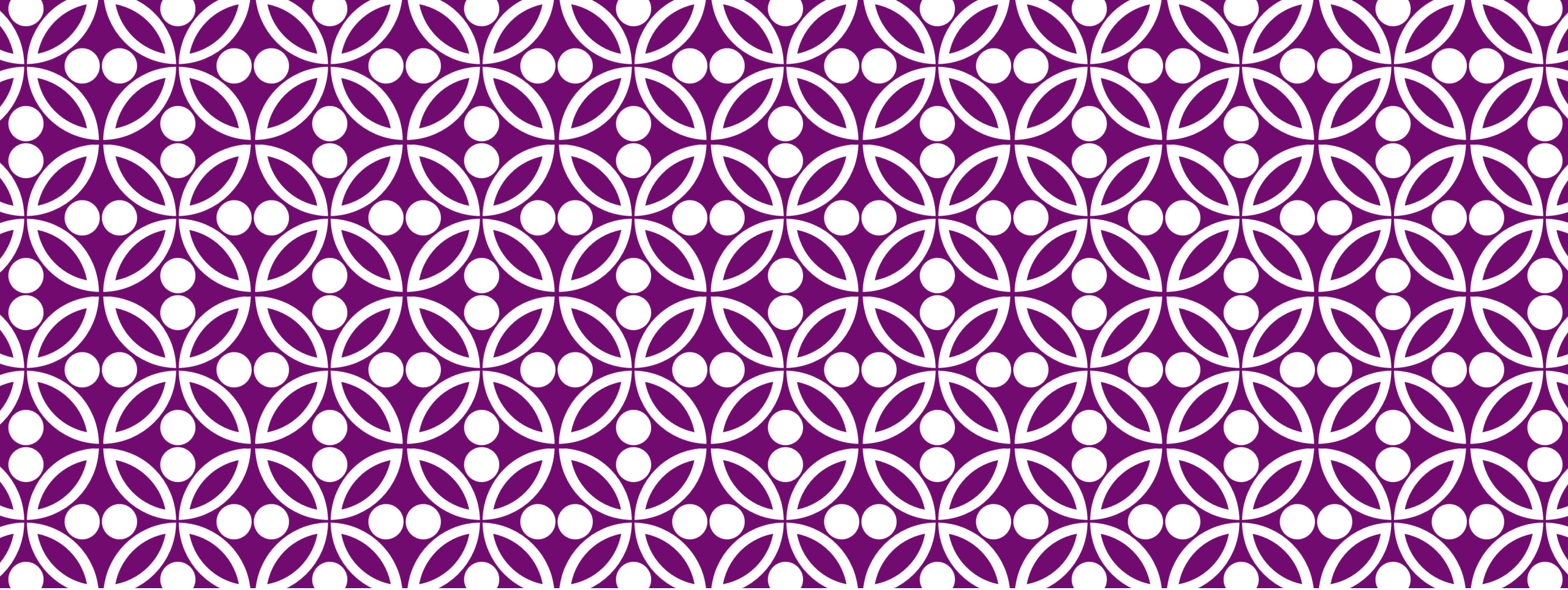
安装方式取决于具体发行版及包管理器:

- `sudo apt install gcc make cmake`
- `sudo pacman -S gcc make cmake`
- `sudo dnf install gcc gcc-c++ make cmake`
- ……

MacOS

MacOS 通常自带一个用 clang++ 冒名顶替的 g++，直接用那个就好了。

- `brew install make cmake`



C++ 基础

Hello World

```cpp
// Hello World
#include <iostream>
int main() {
    std::cout << "Hello, world !" << std::endl;
}
```

单行注释

头文件

主函数

输出

```
$ g++ test.cpp -o test
$ ./test
```

编译

运行

```
$ test
```

运行

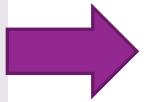
(Windows 命令提示符)

A+B Problem

```cpp
/* A+B
Problem */
#include <iostream>
int main() {
    int a, b;
    std::cin >> a >> b;
    std::cout << a + b << std::endl;
}
```

多行注释

变量声明



输入

数据类型

C++ 的六种基本数据类型如下：

类型	布尔型	字符型	整型	浮点型	双浮点型	无类型
关键字	bool	char	int	float	double	void

int 关键字可用以下关键字修饰：

符号性：

- signed ：带符号（默认）
- unsigned：无符号

位宽：

- short ：**至少** 16 位
- long ：**至少** 32 位
- long long：**至少** 64 位

数据类型

一般情况下，各整数类型的位宽及表示范围：

类型名	位宽	表示范围
short int	16	-2^{15}~$2^{15}-1$
unsigned short int		0~$2^{16}-1$
(long) int	32	-2^{31}~$2^{31}-1$
unsigned (long) int		0~$2^{32}-1$
long long int	64	-2^{63}~$2^{63}-1$
unsigned long long int		0~$2^{64}-1$

类型转换：数值提升

- char 与 short 在算术运算时自动提升为 int，对应无符号类型同理。
- 位宽较小的变量与位宽较大的变量进行算术运算时，前者自动提升。
- bool 类型可以提升为整型，false 对应 0， true 对应 1。

➢ **数值提升过程中，值本身保持不变。** 😊

类型转换：数值转换

- 目标类型为位宽为 x 的无符号整型，转换结果可视作原值 $\mod 2^x$ 的结果。
- 目标类型为位宽为 x 的带符号整型，**一般而言**，转换结果可视作原值 $\mod 2^x$ 的结果。[1]
- 整数转换为浮点数，或位宽较大的浮点数转换为位宽较小的浮点数，会将该数舍入到目标类型下最接近的值。
- 浮点数转换为整数时，会舍弃浮点数的全部小数部分。
- 将其他类型转换为 `bool` 类型时，零值转换为 `false`，非零值转换为 `true`。

➢ **数值转换过程中，值可能会发生改变。** 😵

1 自 C++20 起生效。C++20 前结果是实现定义的。

变量定义

定义变量时，需包含类型说明符及变量名。

常量定义与普通变量类似，只需于类型说明符前添加 const 关键字即可。

```
int hku;
bool astar = true;
const double PI = 3.14;
```

变量作用域

C++ 中，由一对大括号括起来的若干语句构成一个代码块。作用域是变量可以发挥作用的代码块。

全局变量的作用域，自其定义之处开始，至文件结束为止。

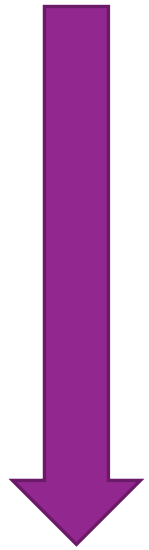
局部变量的作用域，自其定义之处开始，至代码块结束为止。

```
int g = 20;             // 定义全局变量
int main() {
  int g = 10;           // 定义局部变量
  std::cout << g;       // 输出: 10
}
```

运算符

运算符类型	运算符	结合性
单元运算符	+，-，++，--，!	-
二元算术运算符	*，/，%	从左至右
	+，-	
关系运算符	<，<=，>，>=	
	==，!=	
逻辑运算符	&&	
	||	
赋值运算符	=，+=，-=，*=，/=，%=	从右至左

高优先级



低优先级

自增/自减运算符

自增/自减运算符 ++ 及 -- 既可前缀使用，亦可后缀使用。作前缀时，返回自增（减）后的结果；作后缀时，返回自增（减）前的结果。

```
i = 100;
op1 = i++;  // op1 = 100, 先 op1 = i, 然后 i = i + 1
i = 100;
op2 = ++i;  // op2 = 101, 先 i = i + 1, 然后赋值 op2
i = 100;
op3 = i--;  // op3 = 100, 先赋值 op3, 然后 i = i - 1
i = 100;
op4 = --i;  // op4 = 99, 先 i = i - 1, 然后赋值 op4
```

流程控制：分支

```
if (条件) {
  主体;
}
```

```
if (条件) {
  主体1;
} else {
  主体2;
}
```

```
if (条件1) {
  主体1;
} else if (条件2) {
  主体2;
} else if (条件3) {
  主体3;
} else {
  主体4;
}
```

```
switch (选择句) {
  case 标签1:
    主体1;
    break;
  case 标签2:
    主体2;
    break;
  default:
    主体3;
}
```

流程控制：循环

```
for (初始化; 判断条件; 更新) {
  循环体;
}
```

```
while (判断条件) {
  循环体;
}
```

```
do {
  循环体;
} while (判断条件);
```

数组

```cpp
int arr[1001];  // 数组 arr 的下标范围是 [0, 1001)
int main() {
  int n;
  std::cin >> n;
  for (int i = 1; i <= n; i++) {
    std::cin >> arr[i];
  }
}
```

```cpp
int arr[101][101];
int main() {
  for (int i = 1; i <= 100; i++)
    for (int j = 1; j <= 100; j++)
      std::cin >> arr[i][j];
}
```

函数

声明函数，需指明返回值类型、函数名及参数列表。实现可置声明之后。

```
int some_function(int, int);  // 声明
/* some other code here... */
int some_function(int x, int y) {  // 实现
  int result = 2 * x + y;
  return result;
  result = 3;  // 这条语句不会被执行
}
```

在同一文件中，亦可将两者合并。

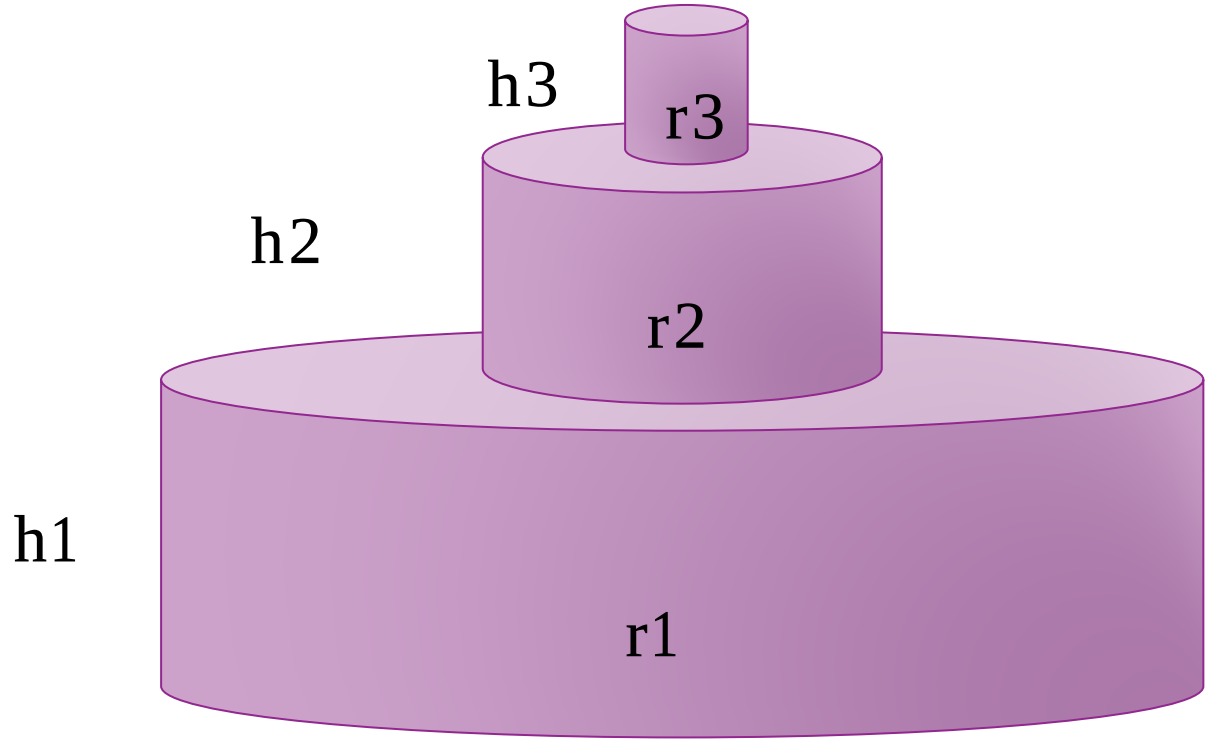
```
int some_function(int x, int y) {
  return 2 * x + y;
}
```

函数

若需于函数中修改变量的值，则需采用“传引用”的方式。

```
void foo(int &x, int &y) {
  x = x * 2;
  y = y + 3;
}
/* ... */
a = 1;
b = 1;
// 调用前: a = 1, b = 1
foo(a, b);  // 调用 foo
// 调用后: a = 2, b = 4
```

小练习1：计算器



h1 = h2 = h3 = 5 m

r1 = 10 m

r2 = 6 m

r3 = 2 m

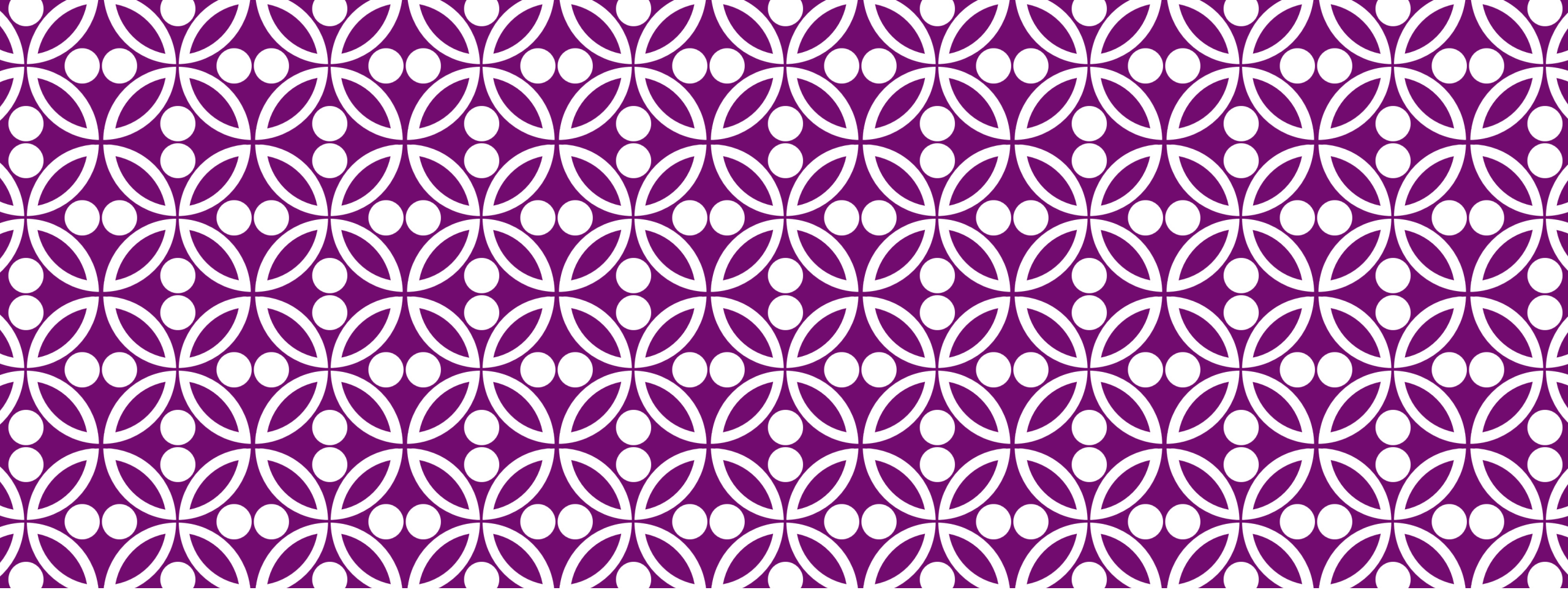
Question: V = ?

参考程序

```cpp
#include <cmath>
#include <iostream>

double volume(const int &h, const int &r) {
    return M_PI * r * r * h;
}

int main() {
    const int h1 = 5, h2 = 5, h3 = 5;
    const int r1 = 10, r2 = 6, r3 = 2;
    const double v1 = volume(h1, r1), v2 =
volume(h2, r2), v3 = volume(h3, r3);
    std::cout << v1 + v2 + v3 << std::endl;
}
```



C++ 面向对象

结构体与类

结构体与类是一系列成员元素及成员函数的组合体。

```
struct Node {
  int weight;
  int value;
  void fun();
};
Node a;
```

```
struct Node {
  int weight;
  int value;
} a;
```

```
class Node {
 public:
  int weight;
  int value;
  void fun();
};
Node a;
```

可以使用 . 访问成员元素或成员函数。

```
std::cout << a.value;
```

访问说明符

- public　：其后成员无论类内或类外皆可访问。
- protected：其后成员可被类内、派生类或友元的成员访问，但类外不能访问。
- private　：其后成员可被类内或友元的成员访问，但类外或派生类不能访问。

➢struct 的所有成员默认 public；class 的所有成员默认 private。

构造函数

```
class Node {
 public:
  int weight, value;
  Node() {
    weight = 0;
    value = 0;
  }
  Node(int _weight = 0, int _value = 0) {
    weight = _weight;
    value = _value;
  }
  // Node(int _weight,int _value):weight(_weight),value(_value) {}
};
Node A;         // ok
Node B(1, 2);   // ok
Node C{1, 2};   // ok, (C++11)
```

析构函数

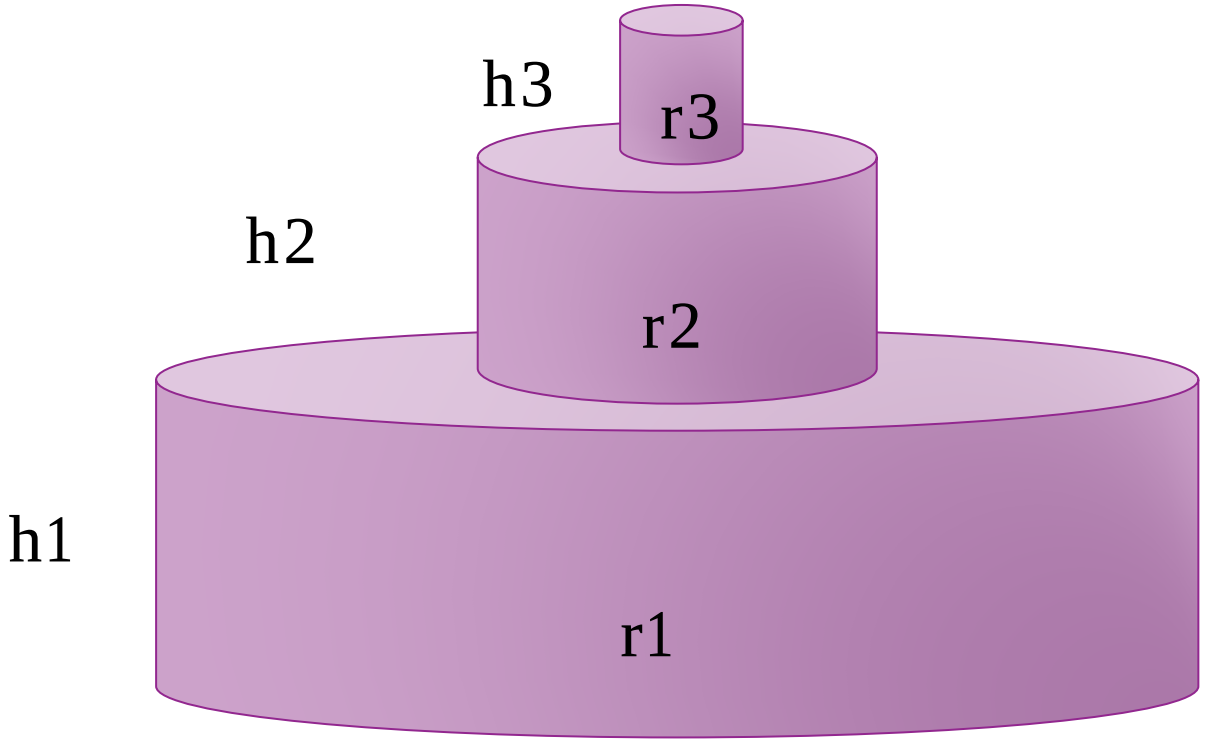
当成员元素包含指针时，需用析构函数手动释放内存，否则易造成内存泄漏。

```
class Node {
 public:
  int weight, value, *ned;
  ~Node() { delete ned; }
};
```

静态成员

```
class Object {
 public:
  Object() { counter++; }
  static int getCounter() { return counter; }
 private:
  static int counter;
};
int Object::counter = 0;
int main() {
  Object obj1, obj2, obj3;
  cout << Object::getCounter() << endl;
}
```

小练习2：计算器



h1 = h2 = h3 = 5 m

r1 = 10 m

r2 = 6 m

r3 = 2 m

Question: V = ?

参考程序

```cpp
#include <cmath>
#include <iostream>
class Cylinder {
 private:
  int h, r;
  static int counter;
 public:
  Cylinder(const int &_h, const int &_r):
    h(_h), r(_r) { counter++; }
  double getVolume() const { return M_PI * r * r * h; }
  static double getCounter() { return counter; }
};
int Cylinder::counter = 0;
int main() {
  const auto c1 = Cylinder(5, 10), c2 = Cylinder(5, 6), c3 = Cylinder(5, 2);
  std::cout << c1.getVolume() + c2.getVolume() + c3.getVolume() << std::endl;
  std::cout << Cylinder::getCounter() << std::endl;
}
```

命名空间

命名空间机制可以用来解决复杂项目中名字冲突的问题。

```
namespace A {
int cnt;
void f(int x) { cnt = x; }
}  // namespace A
```

在该命名空间外，可使用 A::f(x) 访问命名空间 A 内的 f 函数。

命名空间

命名空间的声明可嵌套：

```
namespace A {
namespace B {
void f() { ... }
}  // namespace B
void f() {
  B::f();  // 实际访问的是 A::B::f()，由于当前位于命名空间 A 内，所以可以省略前面的 A::
}
}  // namespace A
void f() { // 这里定义的是全局命名空间的 f 函数，与 A::f 和 A::B::f 都不会产生冲突
  A::f();
  A::B::f();
}
```

命名空间

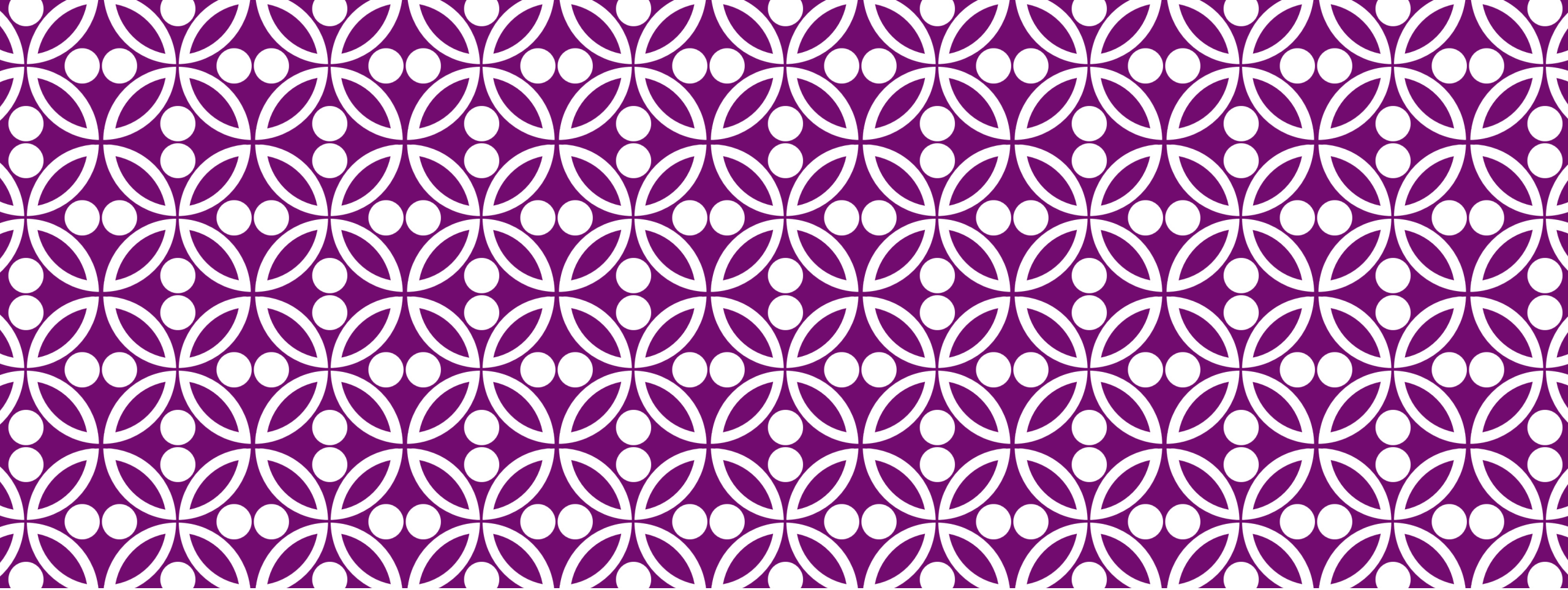
如果懒得使用 :: 访问命名空间内部成员，可以使用 using 语句。

- using 命名空间::成员名； ：将某成员导入当前作用域。
- using namespace 命名空间;：将某命名空间所有成员导入当前作用域。

- 工程中不推荐使用 using namespace 命名空间;，因为这易导致命名冲突。

命名空间

```cpp
#include <iostream>
using std::cin;
using std::cout;
using std::endl;
// using namespace std;
// using std::cin, std::cout, std::endl; (C++17)
int main() {
  int x, y;
  cin >> x >> y;
  cout << y << x <<endl;
}
```



C++ 工程管理

多文件工程

实际工程开发中，人们通常不会在一个文件中写完所有代码，而是将整个工程项目拆分成多个 .h 和 .cpp 文件。两者通常成对出现，前者用于声明，后者用于实现。

可是这样一来，编译就不能像 g++ test.cpp -o test 那样简单了。

于是，make 与 CMake 等自动化建构系统便应运而生。

make

若想使用 make 来建构我们的工程，我们需要将我们的工程划分成若干个“目标”，然后用 Makefile 文件描述各个目标的生成方式及依赖关系。

COMP2113/ENGG1340 的 Group Project 需要使用 make 建构：

- https://github.com/skylee03/HKU-ENGG1340-230

CMake

CMake 是另一个自动化建构系统。与 make 不同，CMake 并不直接建构出最终的软件，而是通过 CMakeLists.txt 产生标准的建构文件（如 Makefile），然后再依一般的建构方式使用。

我们需要修改的代码库 RoboRTS 就是用 CMake 建构的：

- https://github.com/RoboMaster/RoboRTS

通常无需记忆 CMake 具体使用方式，要用时现场查教程即可：

- https://aiden-dong.gitee.io/2019/07/20/CMake教程之CMake从入门到应用/

作业

实现一个面向对象的数据结构，要求如下：

- 支持以下三种操作：
  - 加入一个数 x；
  - 回退到第 k 次操作之后；
  - 删除一个数 x。
- 将该数据结构封装在 namespace 与 class 中；
- 在 .h 和 .cpp 文件内分别编写声明与实现；
- 用 CMake 及 make 合并编译。