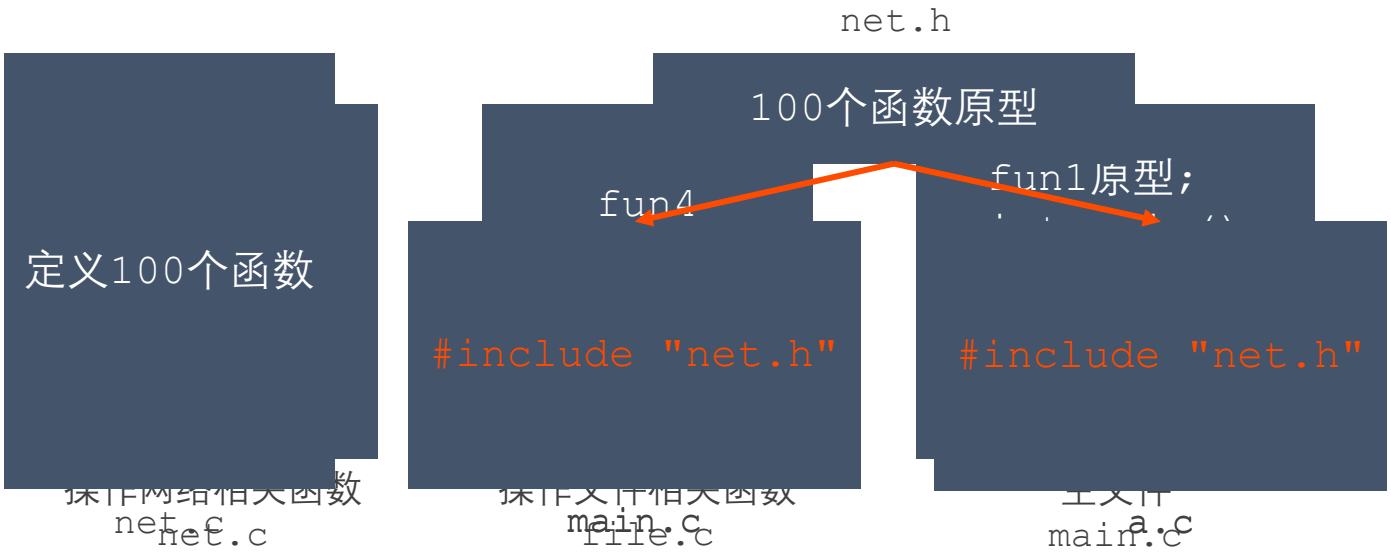
函数分文件编写

多文件编程

- 在开发 C++ 项目时，通常采用多文件编写。
- 分文件的思想：函数的定义在单独的 .c 文件中，函数的声明在对应的 .h 文件中。



Thanks

防止头文件重复引入

重复引入的问题

- 由于在不知情的原因下，有可能将头文件包含多次，如果在头文件中出现了定义（变量、函数、类），就会出现重复定义的问题。
- 一般头文件中只放声明（变量、函数、类），而声明可以多次。
- C++ 重复包含头文件，编译器拷贝和扫描需要耗费时间，降低效率。
- C++ 中处理头文件重复引入，提供了 3 种解决方案：
  - #ifndef、#define、#endif 宏定义
  - #pragma once 指令
  - _Pragma("once") 操作符

#ifndef、#define、#endif 宏定义

- 格式：

```
#ifndef _NAME_H
#define _NAME_H
//头文件内容
#endif
```

- 当程序中第一次 `#include` 该文件时，由于 `_NAME_H` 尚未定义，所以会定义 `_NAME_H` 并执行“头文件内容”部分的代码；当发生多次 `#include` 时，因为前面已经定义了 `_NAME_H`，所以不会再重复执行“头文件内容”部分的代码。

#pragma once 指令

- 格式：在头文件的最开头的位置加上 `#pragma once`
- `#ifndef` 是通过定义独一无二的宏来避免重复引入的，这意味着每次引入头文件都要进行识别，所以效率不高。
- `C` 和 `C++` 都支持宏定义，所以使用 `#ifndef` 不会影响项目的可移植性。
- 和 `#ifndef` 相比，`#pragma once` 不涉及宏定义，当编译器遇到它时就会立刻知道当前文件只引入一次，所以效率很高。
- `#pragma once` 只能作用于某个具体的文件，而无法向 `#ifndef` 那样仅作用于指定的一段代码。

_Pragma("once") 操作符

- 格式：在头文件的最开头的位置加上 _Pragma("once")
- C99 标准中新增加了一个和 #pragma 指令类似的 _Pragma 操作符，其可以看做是 #pragma 的增强版，不仅可以实现 #pragma 所有的功能，更重要的是，_Pragma 还能和宏搭配使用。
- 总结：
  - #pragma once 和 _Pragma("once")，特点是编译效率高，但可移植性差
  - #ifndef 的特点是可移植性高，编译效率差。
  - 除非对项目的编译效率有严格的要求，强烈推荐采用 #ifndef 的方式。
  - 某些场景中考虑到编译效率和可移植性，#pragma once 和 #ifndef 经常被结合使用。

Thanks

字符处理函数

字符函数库cctype

- C++ 从 C 语言继承了一个与字符相关的、非常方便的函数软件包，它可以简化诸如确定字符是否为大写字母、数字、标点符号等工作，这些函数的原型是在头文件 cctype （老式的风格中为ctype.h）中定义的。
- 例如，如果ch是一个字母，则 isalpha（ch） 函数返回一个非零值，否则返回0。（这些函数的返回类型为 int，而不是 bool）
- `if((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z'))`
- `if(isalpha(ch))`

cctype中的字符函数

函数名称	说明
isalnum()	判断是否是字母或者数字字符
isalpha()	判断是否是字母字符
isdigit()	判断是否是数字字符（0~9）
ispunct()	判断是否是标点符号字符
isspace()	判断是否是标准空白（空格、制表符、换行等）
islower()	判断是否是小写字母字符
isupper()	判断是否是大写字母字符
tolower()	转换成小写字母字符
toupper()	转换成大写字母字符

Thanks

字符串拷贝和拼接函数

strlen()

- 头文件 cstring （以前为string.h）提供了对字符串操作相关的函数。
- strlen()
  - 原型：size_t strlen(const char * str);
  - 功能：获取字符串的长度（字符的个数）
  - 参数：
    - str：字符串首地址
  - 返回值：str 字符串的长度（字符的个数）

strcpy()

- strcpy()
  - 原型：char * strcpy(char * dest, const char * src);
  - 功能：把 src 所指向的字符串复制到 dest 所指向的空间中
  - 参数：
    - dest：目的字符串首地址
    - src：源字符串首地址
  - 返回值：dest字符串的首地址

strncpy()

- strncpy()
  - 原型：char * strncpy(char * dest, const char * src, size_t n);
  - 功能：把 src 指向字符串的前 n 个字符复制到 dest 所指向的空间中
  - 参数：
    - dest：目的字符串首地址
    - src：源字符串首地址
    - n：需要拷贝字符的个数
  - 返回值：dest字符串的首地址

strcat()

- strcat()
  - 原型：char * strcat(char * dest, const char * src);
  - 功能：将 src 字符串连接到 dest 的尾部
  - 参数：
    - dest：目的字符串首地址
    - src：源字符串首地址
  - 返回值：dest字符串的首地址

strncat()

- strncat()
  - 原型：char * strncat(char * dest, const char * src, size_t n);
  - 功能：将 src 字符串前 n 个字符连接到 dest 的尾部
  - 参数：
    - dest：目的字符串首地址
    - src：源字符串首地址
    - n：指定需要追加字符串中字符的个数
  - 返回值：dest字符串的首地址

Thanks

字符串比较函数

strcmp()

- strcmp()
  - 原型：int strcmp(const char * s1, const char * s2);
  - 功能：比较 s1 和 s2 的大小，比较的是字符 ASCII 码大小。
  - 参数：
    - s1：字符串1首地址
    - s2：字符串2首地址
  - 返回值：（不同操作系统结果可能不一样，返回 ASCII 差值）
    - 相等：0
    - 大于：> 0
    - 小于：< 0

strncmp()

- strncmp()
  - 原型：int strncmp(const char * s1, const char * s2, size_t n);
  - 功能：比较 s1 和 s2 前 n 个字符的大小，比较的是字符 ASCII 码大小。
  - 参数：
    - s1：字符串1首地址
    - s2：字符串2首地址
    - n：指定比较字符串的字符数量
  - 返回值：（不同操作系统结果可能不一样，返回 ASCII 差值）
    - 相等：0
    - 大于：> 0
    - 小于：< 0

Thanks

字符查找函数

strchr()

- strchr()
  - 原型：const char * strchr(const char * s, int c);
  - 功能：在字符串 s 中查找字母 c 出现的位置
  - 参数：
    - s：字符串首地址
    - c：匹配字母(字符)
  - 返回值：返回第一次出现的 c 地址

Thanks

字符串查找函数

strstr()

- strstr()
  - 原型：char * strstr(const char * str, const char * substr);
  - 功能：在字符串 str 中查找字符串 substr 出现的位置
  - 参数：
    - src：源字符串首地址
    - substr：匹配字符串首地址
  - 返回值：返回第一次出现的 substr 地址

Thanks

字符串切割函数

strtok()

- strtok()
  - 原型：char * strtok(char * str, const char * delim);
  - 功能：将字符串切割。当 strtok() 在参数 str 的字符串中发现参数 delim 分割字符时，则会将该字符改为\0 字符，当连续出现多个时只替换第一个为\0。
  - 参数：
    - str：将要分割的字符串
    - delim：分割字符串中包含的所有字符
  - 返回值：分割后字符串首地址
- 注意：
  - 在第一次调用时：strtok() 必需给予参数 str 字符串
  - 往后调用则将参数 str 设置成空指针，每次调用成功则返回指向被分割出片段的指针

Thanks

字符串转换函数

大小写转换

- strupr()
  - 原型：char * strupr(char * str);
  - 功能：将 str 字符串转换成大写字母
  - 参数：str 将要转换的字符串的首地址
  - 返回值：str 字符串的首地址
- strlwr()
  - 原型：char * strlwr(char * str);
  - 功能：将 str 字符串转换成小写字母
  - 参数：str 将要转换的字符串的首地址
  - 返回值：str 字符串的首地址

字符串转数字

- atoi()
  - 原型：int atoi(const char * nptr);
  - 功能：atoi() 会扫描 nptr 字符串，跳过前面的空白字符，直到遇到数字或正负号才开始做转换，而遇到非数字或字符串结束符('\0')才结束转换，并将结果返回。
  - 参数：nptr 待转换的字符串
  - 返回值：成功转换后的整数，如果转换不成功则为0
- 类似的函数：
  - atof()：将一个字符串转换为一个 double 类型的数据
  - atol()：将一个字符串转换为一个 long 类型的数据
  - atoll()：将一个字符串转换为一个 long long 类型的数据

字符串转数字

- strtod()
  - 原型：double strtod(const char* str, char** endptr);
  - 功能：把参数 str 指向的字符串转换为一个浮点数（double）。如果 endptr 不为空，则指向转换中最后一个字符后的字符的指针会存储在 endptr 引用的位置。
  - 参数：str 待转换的字符串，endptr 对类型为 char* 的对象的引用，其值由函数设置为 str 中数值后的下一个字符的地址。
  - 返回值：返回转换后的双精度浮点数，如果没有执行有效的转换，则返回零（0.0）。
- 类似的函数：
  - long int strtol(const char *str, char **endptr, int base);
  - strtof()、strtoll()、strtoul()、strtoull()、strtold()

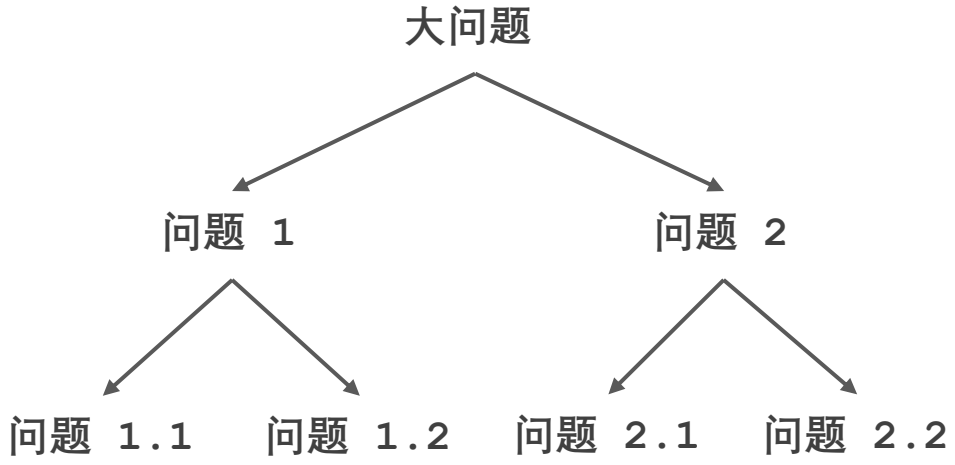
Thanks

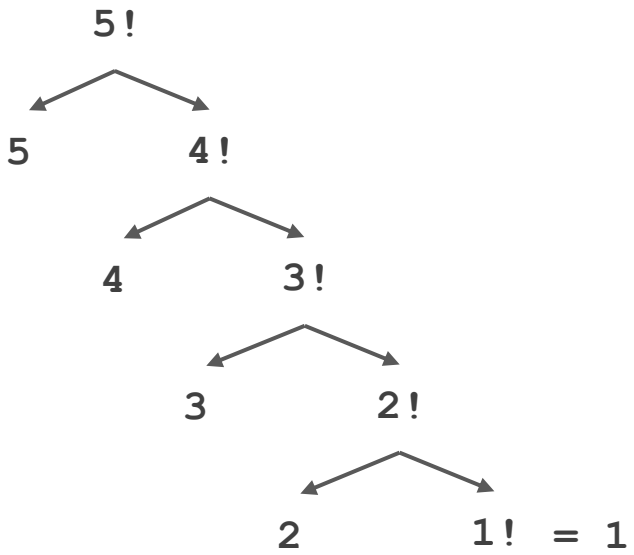
递归

什么是递归

- 函数定义中调用函数本身，被称为递归。
- 递归解决问题的思路：拆分和合并。

```
5! = 5 * 4 * 3 * 2 * 1 = 5 * 4!
n! = n * (n - 1)!
```

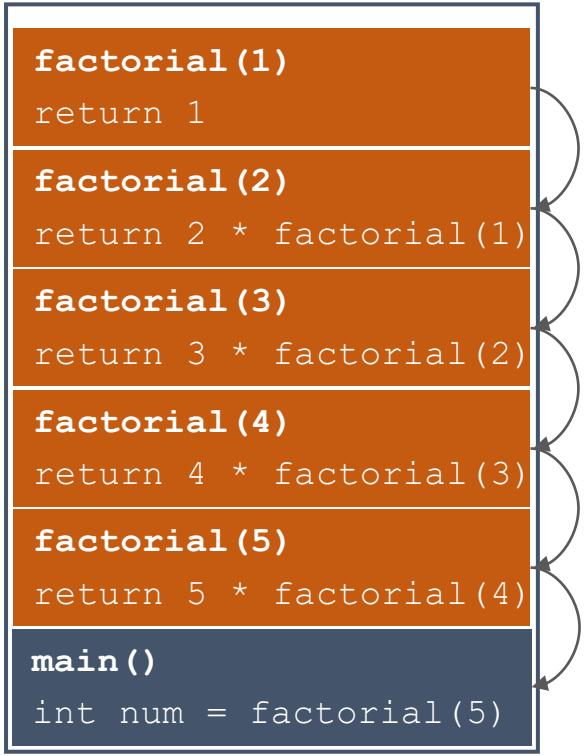




递归内存图解

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

int main() {
    int num = factorial(5);
    cout << num << endl;
}
```



Thanks

引用

引用

- C++ 新增了一种复合类型 —— 引用，引用是已经定义的变量的别名（另一个名称）。例如：如果将 rnum 作为 num 变量的引用，则可以交替使用 rnum 和 num 来表示该变量。
- 引用变量的主要用途是作为函数的参数，将引用变量用作参数，函数将使用原始数据，而不是其副本。这样除指针之外，引用也为函数处理大型结构提供了一种非常方便的途径，同时对于设计类来说，引用也是必不可少的。
- 基本语法：

```
typename & ref = varname;

int num = 20;

int & rnum = num;
```

Thanks

函数重载

什么是函数重载

- 函数重载（Overload）是 C++ 语言在 C 语言基础上新增的功能。函数重载能够在程序中使用多个同名的函数。
- 通过函数重载来设计一系列的函数，它们完成相同或者相似的功能，但使用不同的参数列表。
- 函数重载的关键是函数的参数列表（函数特征标）。如果两个函数的参数数目和类型相同，同时参数的排列顺序也相同，则它们的特征标相同，而与变量名无关。
- C++允许定义名称相同的函数，条件是它们的特征标不同。如果参数数目、参数类型、排列顺序不同，则特征标也不同。

示例

- 定义一组函数原型如下的print( )函数:

```
void print(const char * str, int width);  // #1
void print(double d, int width);          // #2
void print(long l, int width);            // #3
void print(int i, int width);             // #4
void print(const char * str);             // #5
```

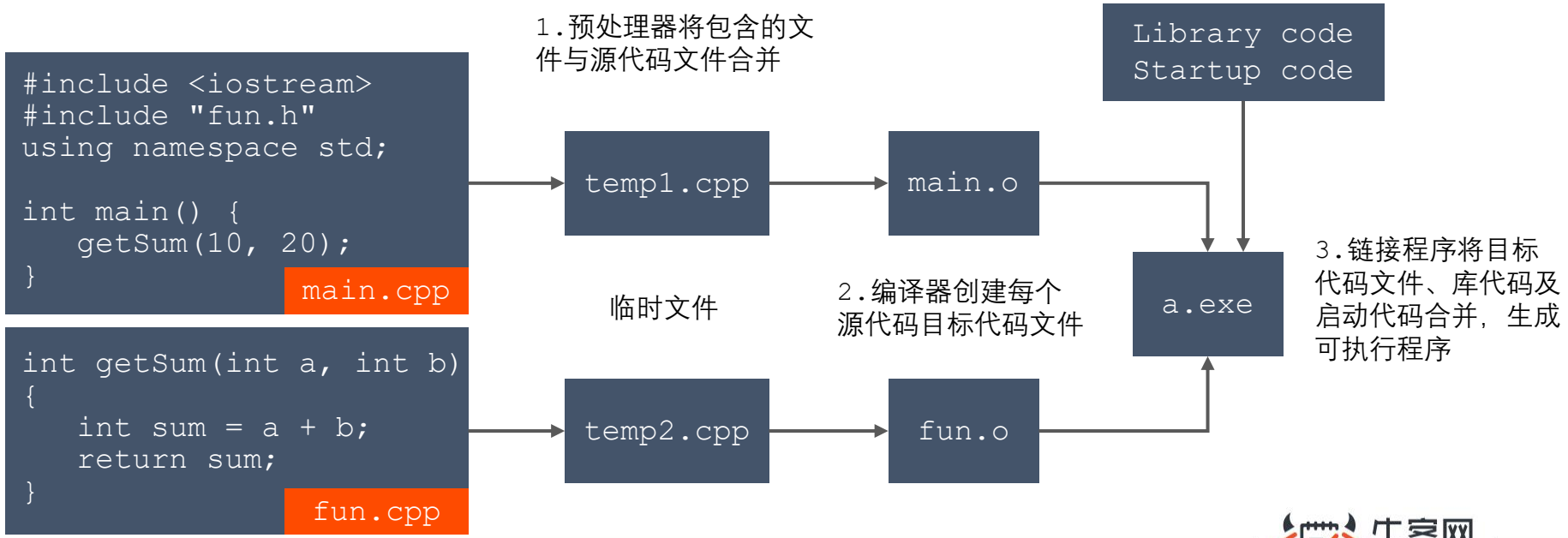
```
print("world", 15);
print("hello");
print(12.0, 10);
print(12, 10);
print(12L, 12);
```

Thanks

存储持续性、作用域、链接性

单独编译

- C++ 多文件编程中，可以单独编译这些文件，然后将它们链接成可执行程序。
- 编译多个源代码文件的命令：g++ main.cpp fun.cpp



存储持续性

- C++使用三种不同的方案来存储数据，这些方案的区别就在于数据保留在内存中的时间。

1. 自动存储持续性：在函数定义中声明的变量（包括函数参数）的存储持续性为自动的。它们在程序开始执行其所属的函数或代码块时被创建，在执行完函数或代码块时，它们使用的内存被释放。
2. 静态存储持续性：在函数定义外定义的变量和使用关键字 static 定义的变量的存储持续性都为静态，它们在程序整个运行过程中都存在。
3. 动态存储持续性：用 new 运算符分配的内存将一直存在，直到使用 delete 运算符将其释放或程序结束为止。这种内存的存储持续性为动态，有时被称为自由存储（free store）或堆（heap）。

作用域和链接性

- 作用域（scope）描述了名称在文件（翻译单元）的多大范围内可见。例如，函数中定义的变量可在该函数中使用，但不能在其他函数中使用；而在文件中的函数定义之前定义的变量则可在所有函数中使用。

- 链接性（linkage）描述了名称如何在不同单元间共享。链接性为外部的名称可在文件间共享，链接性为内部的名称只能由一个文件中的函数共享。自动变量的名称没有链接性，因为它们不能共享。

Thanks