



復旦大學

COMP130135

# 面向对象程序语言C++ 标准库

王雪平

wangxp@fudan.edu.cn

2020/2/15



## 主要内容

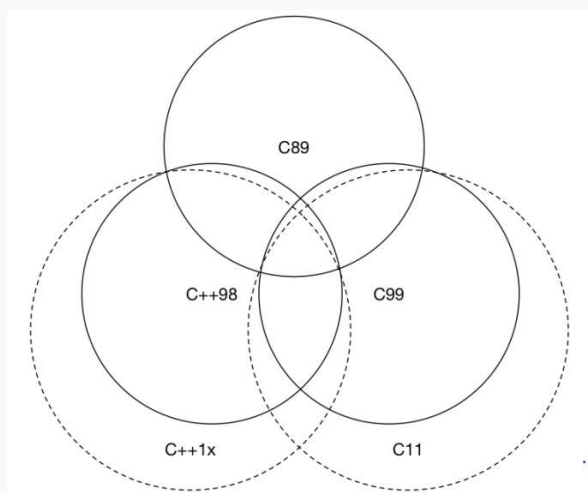
1. C与C++
2. 使用Hello World程序
3. C++从代码到执行
4. 使用std::string
5. 计算机基础



## 相互关系

### 1. C与C++

- ❑ 不同：各有标准；各有编译器；
- ❑ 类似：C和C++有相当一部分是重叠的，但是还有相当一部分是不同的。  
随着时间的推移，两者之间的差异会越来越大！
- ❑ 可以使用韦恩图表示如下



## 注意

## 1. C与C++

- ❑ C++可以使用C的标准库，但是不建议使用，除非必要，特别是关于输入输出方面的内容，强烈建议不要在C++中不要使用C的输入输出函数！
- ❑ C++和C使用不同的编译命令。例如在GCC编译套件中，编译C++代码，需要使用命令`g++`；而编译C代码，需要使用命令`gcc`！
- ❑ 要清楚各种标准中的特性：这样在编译C++代码时可以指定相应的标准。例如在GCC编译套件中，如果使用了C++17的标准，在编译时需要使用命令参数`g++ -std=c++17`。



## C++的Hello World程序

## 2. 使用Hello World程序

```
//chapt01/hello.cpp
```

单行注释

```
#include <iostream>
```

标准头文件

```
int main()
```

main函数

```
{
```

```
    std::cout << "Hello, world!" << std::endl;
```

```
    return 0; //返回0表示运行成功
```

标准输出流

输出  
操作符结束当前  
输出行

## 操作符<<

```
std::cout << "Hello, world!" << std::endl;
```

操作符：两元操作符号，需要两个操作数。

运 算： 按照左结合的顺序。

表达式：除计算外，表达式本身还会返回值。

第二个<<的  
第一个操作  
数是什么？



## 名字空间

- 名字空间： namespace
- 也称名字空间、名称空间等，表示着一个标识符（identifier）的可见范围。
- 标识符在不同命名空间中是互不相干的。

- C++名字空间： namespace
- 在一个名字前加上std::表明这个名字属于名字空间std;
- 标准库使用std来包含它定义的所有名字;
- “::” 是名字空间操作符。



## 简化重复的std::

```
//chapt01/hello.cpp
```

```
#include <iostream>
```

```
using std::cout; using std::endl;
```

```
int main()
```

```
{
```

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

```
    return 0;
```

```
}
```

使用using  
关键词

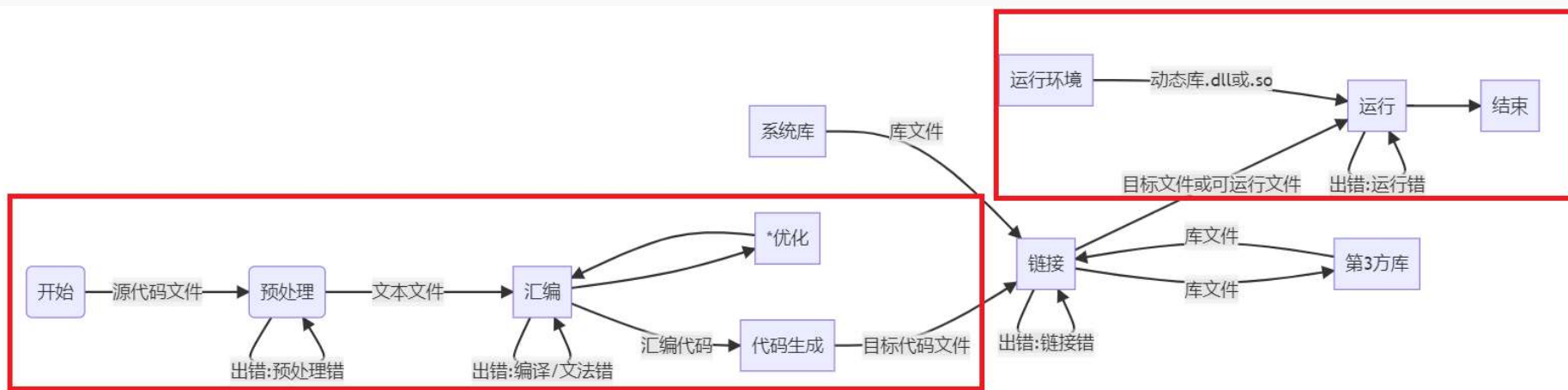




## 从代码到执行

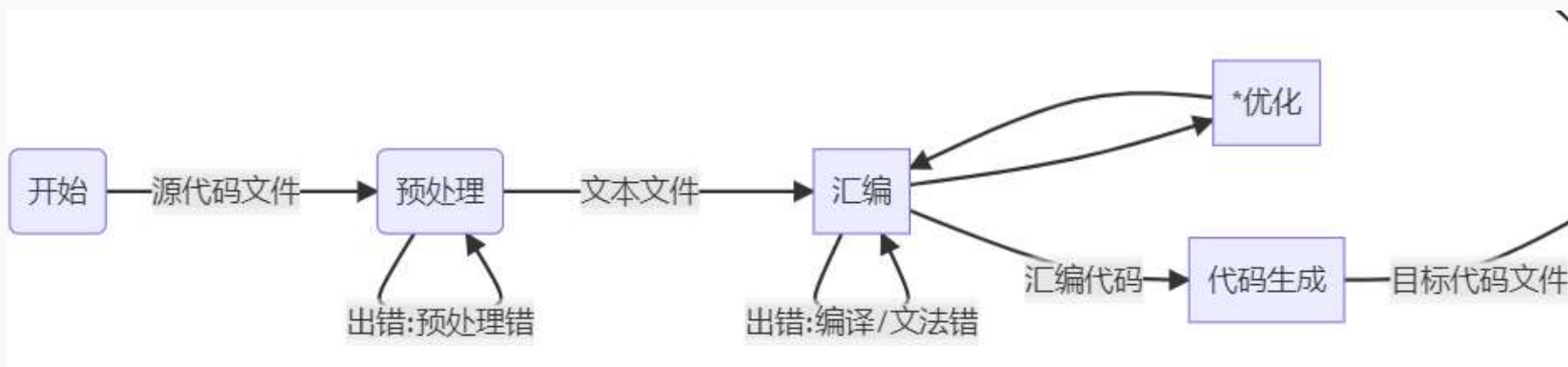
### 3. C++从代码到执行

- 编译性的语言：从一种代码变成另外一种代码，最终执行的是机器码。
- 可以分为：编译、链接、执行三个阶段
- 编译阶段最为复杂：下图左下侧的框为编译阶段；右上侧框图为运行阶段；除此以外为链接阶段。



## 编译阶段

- ❑ 编译阶段：对每一个编译单元进行处理，得到目标代码文件。
- ❑ 编译单元：C/C++的源文件，包括.c、.cpp或.cc文件；C/C++的头文件不是编译单元，无法生成目标代码，但是可以通过#include成为源文件的一部分。
- ❑ 编译阶段比较复杂，分为：预处理、文法分析、汇编、优化、代码生成。其中优化是可选步骤。
- ❑ GCC可以让我们分解每一个阶段：预处理、汇编、代码生成
- ❑ 以下的例子使用代码function.cpp, function.h, main.cpp



## 预处理

- ❑ 预处理阶段：把源文件和头文件中的#指令进行处理，典型的包括：
  - ❑ #include指令，把相应的头文件插入；
  - ❑ #define 宏定义，直接对相应的内容进行替换；
  - ❑ #ifdef #else 根据条件，包含(去除)部分代码；
- ❑ 对上面的代码进行预处理，需要使用如下命令
  - ❑ `g++ -std=c++17 -E -P function.cpp -o function.i`
  - ❑ `g++ -std=c++17 -E -P lcrmain.cpp -o main.i`
- ❑ 注意：只有function.cpp和lcrmain.cpp是编译单元，function.h是头文件不是编译单元。**对每一个编译单元都需要进行编译过程。**



## 预处理

## □ 预处理实例

```
// function.h
#pragma once
```

```
#define FIRST_OPTION
#ifdef FIRST_OPTION
#define MULTIPLIER (3.0)
#else
#define MULTIPLIER (2.0)
#endif
```

```
float add_and_multiply(float x, float y);
```

```
// function.cpp
#include "function.h"
```

```
int nCompletionStatus = 0;
float add(float x, float y)
{
    float z = x + y;
    return z;
}
float add_and_multiply(float x, float y)
{
    float z = add(x, y);
    z *= MULTIPLIER;
    return z;
}
```

命令: `g++ -std=c++17 -E -P function.cpp -o function.i`

```
// function.i 预处理的结果
float add_and_multiply(float x, float y);
int nCompletionStatus = 0;
float add(float x, float y)
{
    float z = x + y;
    return z;
}
float add_and_multiply(float x, float y)
{
    float z = add(x, y);
    z *= (3.0);
    return z;
}
```

注意:1)function.h的内容现在只剩下函数声明这一行;

2) function.h中宏定义MULTIPLIER在最终的代码中不出现, 已经被替换;

3) function.h中的**#ifdef #else #endif**已经在预处理阶段执行完毕, 在预处理之后我们只看到结果, 就是MULTIPLIER 被定义为(3.0)。



## 汇编

- ❑ 汇编：经过前面的预处理、文法分析、语法分析，在没有错误的情况下，会生成汇编语言代码。
- ❑ 从语言上讲，汇编语言更加贴近计算机硬件，是一门专门的语言。
- ❑ 对上面的代码进行汇编，需要使用如下命令
  - ❑ `g++ -S function.i -o function.s`
  - ❑ `g++ -S clrmain.i -o clrmain.s`
- ❑ 注意：对每个编译单元都要独立进行处理；而且可以从上一阶段产生的结果开始进行。即每个阶段都有相应的输入，同时每个阶段都有相应的输出；而下一个阶段的输入就是上一个阶段的输出；这样就构成了一个类似管道的操作。



## 汇编

## □ 汇编实例(部分)

```

        .seh_proc    _Z16add_and_multiplyff
_Z16add_and_multiplyff:
.LFB1:
        .seh_endprologue
        addss    %xmm1, %xmm0
        mulss    .LC0(%rip), %xmm0
        ret
        .seh_endproc
        .globl    nCompletionStatus
        .bss
        .align 4
nCompletionStatus:
        .space 4
        .section .rdata,"dr"
        .align 4

```

命令: g++ -S -O1 function.i -o function.s

## □ 分析.s文件, 可以看到

- 函数和数据在汇编代码中有不同名字、形式和位置;
- 函数add\_and\_multiply名字为\_Z16add\_and\_multiplyff;
- 全局变量nCompletionStatus的名字没有变化;
- 函数和数据是放在不同的部分的。



## 代码生成

- ❑ 代码生成：生成目标代码文件(object file)，使用-c选项，同时可以从.cpp文件，或.i文件，或.s文件生成，可以指定代码优化-Ox选项；
- ❑ 生成的代码是二进制形式，可以说是前面生成的汇编码的二进制形式
- ❑ 对上面的代码进行汇编，可以使用如下命令
  - ❑ `g++ -c function.s -o function.o`
  - ❑ `g++ -c -O1 function.i -o function.o`
  - ❑ `g++ -c -O1 function.cpp -o function.o`
- ❑ 注意：可以从.cpp文件，.i文件或.s文件直接生成目标代码文件。一个编译单元对应一个目标代码文件。
- ❑ 目标代码文件是二进制的，不能直接使用vscode打开查看；但是可以使用工具nm, objdump, readelf查看其内容—这些工具都是GCC套件的一部分，可以在shell下直接使用。



## 代码生成

### □ 目标文件

```
objdump -d function.o
```

```
Disassembly of section .text:
```

```
0000000000000000 <_Z3addff>:
```

```
0: f3 0f 58 c1      addss  %xmm1,%xmm0
4: c3              ret
```

```
0000000000000005 <_Z16add_and_multiplyff>:
```

```
5: f3 0f 58 c1      addss  %xmm1,%xmm0
9: f3 0f 59 05 00 00 00 mulss  0x0(%rip),%xmm0
# 11 <_Z16add_and_multiplyff+0xc>
10: 00
11: c3              ret
```

```
nm function.o
```

```
0000000000000005 T _Z16add_and_multiplyff
0000000000000000 T _Z3addff
0000000000000000 B nCompletionStatus
```

命令: `g++ -c -O1 function.s -o function.o`

### □ 分析.o文件，可以看到

□ 可以使用`objdump -d`命令查看里面的函数: `_Z16add_and_multiplyff`和前面汇编代码中是一致的;

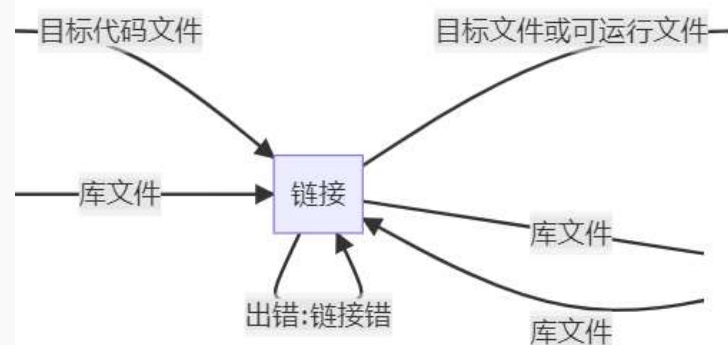
□ 可以使用`nm`命令查看里面的符号: 例如`function`中定义了三个符号: 特别注意函数的名称有变化。 什么变化?





## 链接阶段

- ❑ 链接阶段：把编译阶段生成的（多个）目标文件和库文件，经过处理，生成一个可运行的程序或者动态库的过程。经过上述处理：function.o中包含两个函数的二进制代码；main.o中包含main函数的代码，同时main函数需要调用function.o中的两个函数。这样就需要把这两个目标文件以某种形式合成一个；同时代码的最终运行还需要一些系统函数的支持（这部分代码在C++库中），有时还需要第3方库的支持。
- ❑ 链接阶段可能发生错误：1）根本无法找到某个函数的实现；2）根本无法找到需要的库文件；3）库文件不匹配。
- ❑ 生成的可执行程序：可以使用nm、objdump -d、readelf等工具来查看—实际上.exe文件和编译阶段产生的文件，其格式大同小异。



# 链接

objdump -d main.exe

nm main.exe

## □ 可运行程序

0000000140001850 <\_Z3addff>:

140001850: f3 0f 58 c1

140001854: c3

addss %xmm1,%xmm0

ret

0000000140001855 <\_Z16add\_and\_multiplyff>:

140001855: f3 0f 58 c1

140001859: f3 0f 59 05 ff 37

00 mulss 0x37ff(%rip),%xmm0

140001860: 00

140001861: c3

addss %xmm1,%xmm0

# 140005060 <.rdata>

ret

命令: g++ function.o clrmain.o -o main.exe

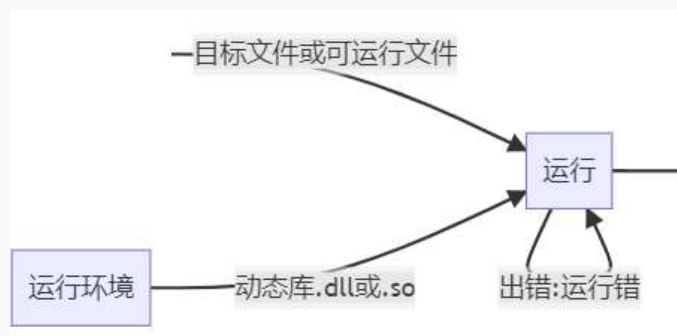
## □ 分析.exe文件，可以看到

- 和.o文件不一样的是，函数和数据前面的地址发生了变化；名称还是一样的；
- 在.exe中包含更多的函数和符号：有很多是在c语言中熟悉的



## 运行阶段

- ❑ 这运行阶段：把生成的.exe加载到内存中并开始程序执行的过程。整个过程比较复杂，本课程不详细描述。
- ❑ 在链接时：可以把所有需要的代码都一并放在最终的.exe中，这样.exe在运行时就不需要其他的代码了—这种链接方式称为静态链接；缺点是最终的.exe很大；还可以只把部分代码放在.exe中，公用的部分（例如输入输出、函数计算等），放在共享动态库中（windows中的dll或linux下的so文件），这样就可以减少.exe的文件大小，这种称为动态链接。



## 运行阶段

- ❑ 通常使用动态链接的方式。这样会产生很多问题，比如到哪里去找这些动态库；如果有多个同名的动态库，到底取哪一个？
- ❑ 无论在windows下还是linux下，都可以设置Path环境变量：当输入一个命令(例如main.exe)时，shell就会按照path中的设置，在每一个目录中去找对应的命令，当找到时就结束这个查找的过程；如果没有找到，就找下一个目录，依次类推。如果Path变量中的所有目录查找完毕，都没有找到的话，就报错：windows下的报错
  - ❑ windows **xxx不是内部或外部命令，也不是可运行的程序或批处理文件。**
  - ❑ Linux **Command 'xxx' not found, did you mean:**
- ❑ 对于程序需要的动态库(.dll和.so)进行类似的查找。
- ❑ 当使用手动方式安装程序时，要特别主要添加Path环境变量。



## 使用字符串-输入

```
// chapt01/getname.cpp
//ask for a person's name, and greet the person

#include <iostream>
#include <string>
int main()
{
    //ask for the person's name
    std::cout << "Please enter your first name:";

    // read the name;
    std::string name;
    std::cin >> name;

    //write greeting
    std::cout << "Hello, " << name << "!" << std::endl;
    return 0;
}
```

输入  
操作符

变量



## 改写代码getname.cpp

- ❑ 为什么需要改写：main函数中包含所有的细节性的代码！不利于理解代码
- ❑ 可以把其中的语句拆成函数：完成上述任务需要做两件事情1) 读取名称；2) 输出名称！
- ❑ 改写后的代码思路更加清楚！

```
int main()
{
    std::string name = getUserName();
    greetUser(name);
    return 0;
}
```



原则1：每个函数只完成一个任务；不要把多个任务放在一个函数中！  
SRP: Single Responsibility Principle

# 实例1：为名字装框输出-运行结果(frame.cpp)

```
Please enter your first name: wang

*****
*                               *
* Hello, wang! *
*                               *
*****
```



## string 类型

### ■ String类型

- 定义在标准头文件<string>中，是C++标准库之一；
- 一个string类型的对象包含零个或多个字符的序列；
- 下表列出了string类型的部分操作。

操作	说明
<code>std::string s</code>	定义一个变量，初始化为空
<code>std::string t=s</code>	定义一个变量t，初始化为s
<code>std::string z(n, c)</code>	定义一个变量z，用字符c的n份复制来初始化为z
<code>os &lt;&lt; s</code>	把s中含有的所有字符，写入到os代表的输出流中
<code>is &gt;&gt; s</code>	从is表示的流中读取非空白符的字符，把从is成功读取的字符存入s中。
<code>s + t</code>	包含s中所有字符的复制并且在后面紧接着t中所有字符的复制
<code>s.size()</code>	s中含有的所有字符的个数





## 实例1: 为名字装框输出-代码

```
// chapt01/frame.cpp
// frame name

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

int main()
{
    cout << "Please enter your first name:";
    string name;
    cin >> name;

    //build the message that we intend to write
    const string greeting = "Hello, " + name + "!";
    const string spaces(greeting.size(), ' ');
    const string second = "* " + spaces + " *";
    const string first(second.size(), '*');

    //write it all
    cout << endl;
    cout << first << endl;
    cout << second << endl;
    cout << " * " << greeting << " * " << endl;
    cout << second << endl;
    cout << first << endl;
    return 0;
}
```



## 初始化、重载和常量

```
const string greeting = "Hello, " + name + "!";
```

初始化

常量

重载



## 构造函数和成员函数

```
const string spaces(greeting.size(), ' ');
```



## String变量的定义与使用

```
std::string hello="Hello"; //构造字符串，其包含字符序列Hello
```

```
std::string first(10, '*'); //使用构造函数，构造一个字符串，该字符串包含10个*号。
```

```
std::string name; //构造一个空字符串对象：注意该对象是合法的，只是长度为0
```



## 改写代码frame.cpp

- ❑ 根据SRP, frame.cpp代码太糟糕了! (Code smells)
- ❑ 改写后的代码如下, 思路更加清楚!

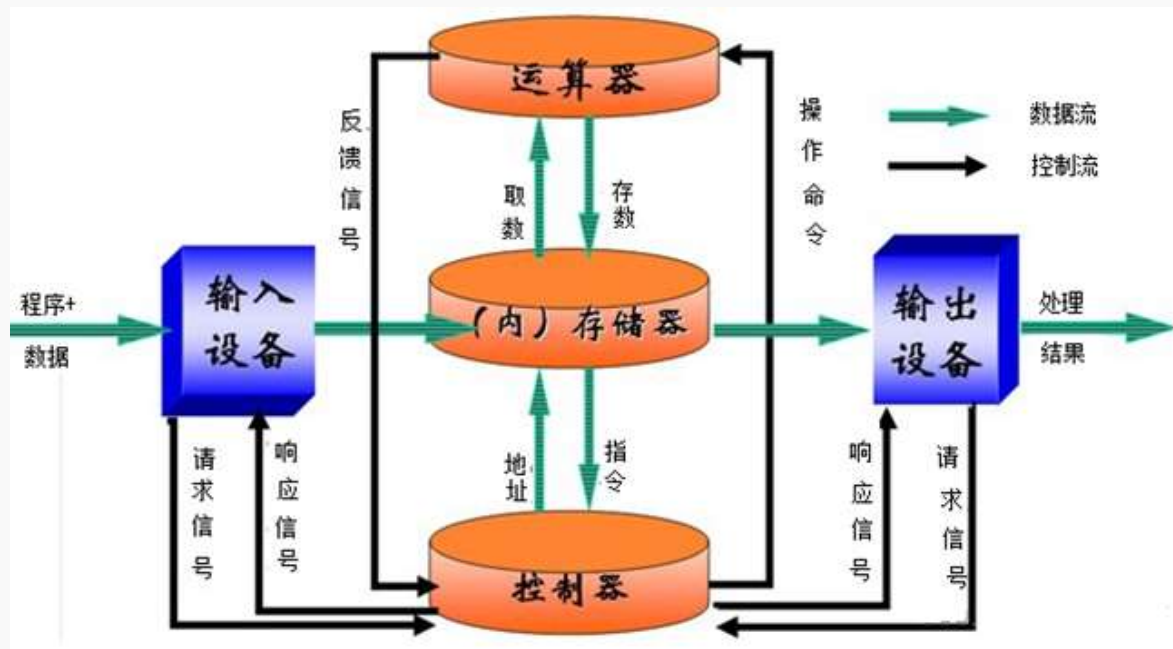
```
// chapt01/frame_v2.cpp
int main()
{
    std::string name = getUsername();
    std::cout << frameUserName(name);

    return 0;
}
```



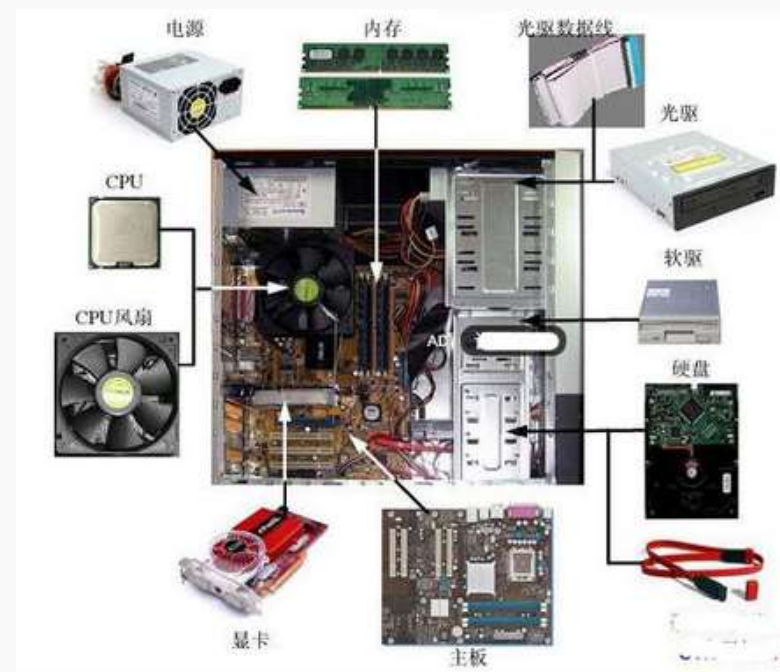
完整的代码在frame\_v2.cpp中

## 计算机体系结构



冯·诺伊曼体系结构

## 5. 计算机基础



计算机的物理构成图

- 1) CPU( Central Processing Unit ) =运算器+ 控制器;
- 2) 辅存: 包括硬盘、闪存 (U盘、TF卡、SD卡等)、DVD;
- 3) 数据和指令首先从外存读到内存后才能被CPU使用

# 存储层次

- 计算机内部采用二进制表示 1) 一个字节=8个比特 (1B=8b) ; 2) 一个字节可以表示0到255的整数或表示一个ASCII字符; 3) 二进制00101011对应十进制43。

存储容量(单位:字节)

1~2K

640K~16M

1G~512G

>256G

>256G



访问时间(单位:  $T_0$ )

1

10~200

1000

100000

>100000

$T_0$ : 在纳秒级

1K字节= $2^{10}$ 字节

1M字节= $2^{20}$ 字节

1G字节= $2^{30}$ 字节

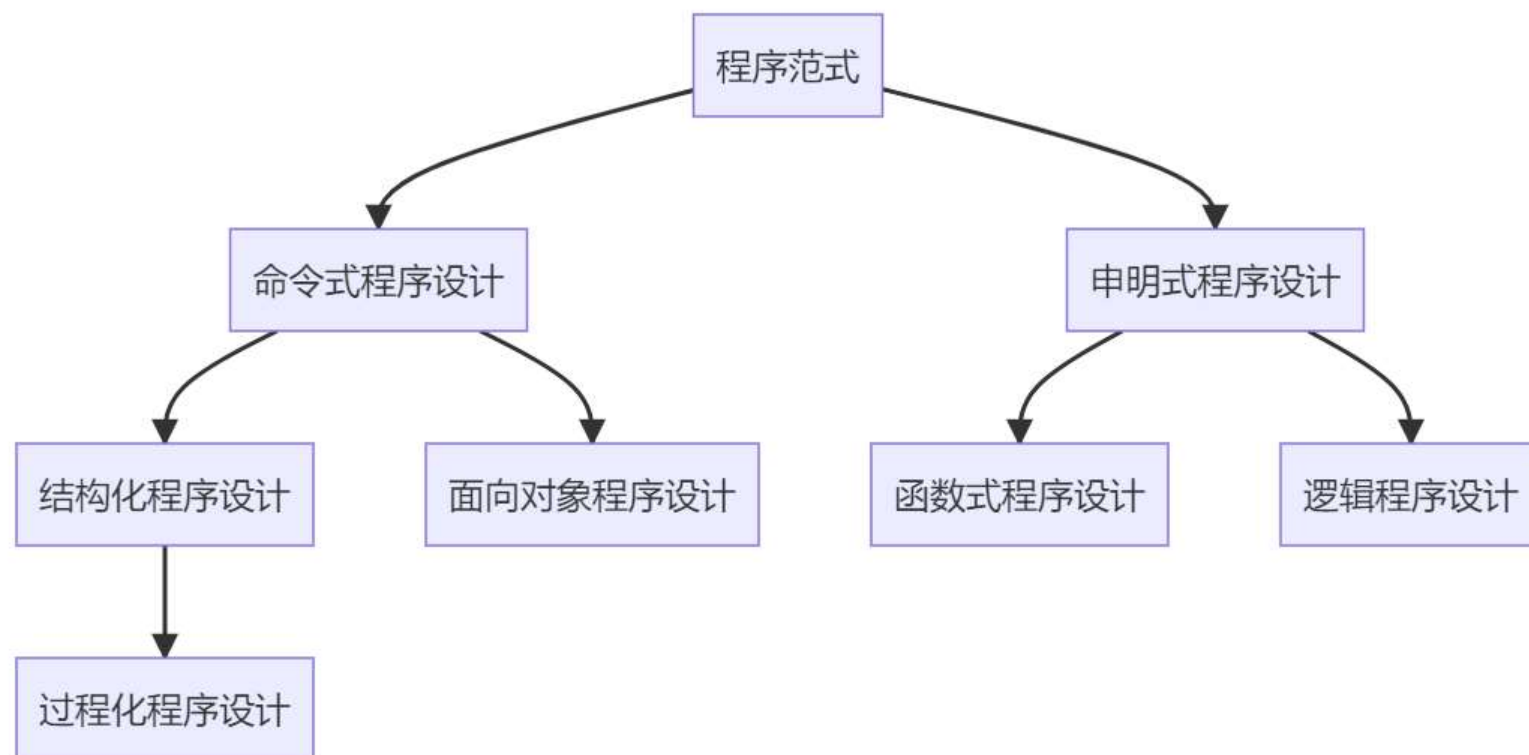
1T字节= $2^{40}$ 字节

1P字节= $2^{50}$ 字节

1E字节= $2^{60}$ 字节

## 编程范式(programming paradigm)

## 6. 编程范式





## 命令式程序设计(imperative programming)

- ❑ 在命令式程序设计中，控制流程是显式的，程序中的每一条语句表示了一条命令；
- ❑ 而其先后关系则指明了执行的顺序关系。

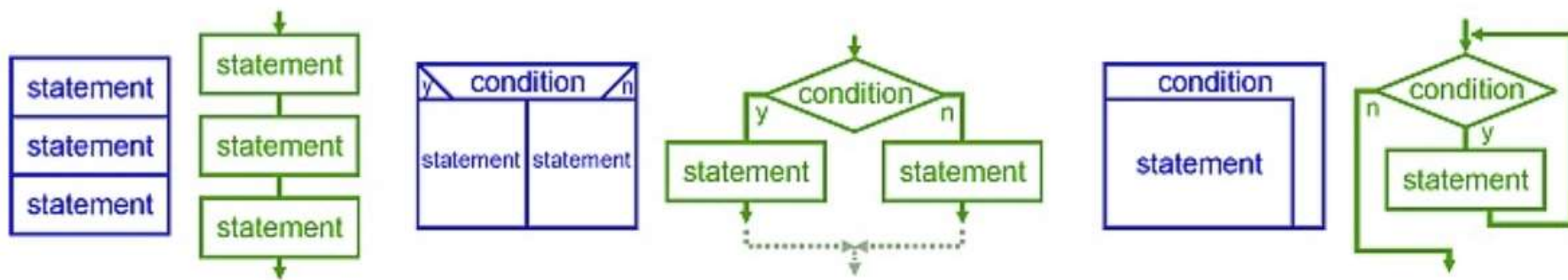
```
    result = []
    i = 0
start:
    numPeople = length(people)
    if i >= numPeople goto finished
    p = people[i]
    nameLength = length(p.name)
    if nameLength <= 5 goto nextOne
    upperName = toUpper(p.name)
    addToList(result, upperName)
nextOne:
    i = i + 1
    goto start
finished:
    return sort(result)
```



## 结构化程序设计(SP, structured programming)

- 进行了增强;
- 构成部分包括: 包含顺序、选择、循环以及递归四种形式的控制结构

```
result = [];  
for i = 0; i < length(people); i++ {  
    p = people[i];  
    if length(p.name) > 5 {  
        addToList(result, toUpper(p.name));  
    }  
}  
return sort(result);
```



## 过程化程序设计(PP, procedural programming)

- ❑ 强调程序的模块化，有时也把过程化程序设计称为模块化程序设计。
- ❑ 面向大型复杂的系统
  - ❑ 过程化程序设计和自上而下的程序设计哲学是契合的。
  - ❑ 在不同的语言中，模块的定义是不一样的：例如函数、包、类、对象等。



## 面向对象程序设计(OOP, object-oriented programming)

- ❑ 认为现实世界中的问题，实际上是现实世界中的实体(Entity，即对象)的相互作用产生的结果。
- ❑ 这个与现实世界是类似的。
- ❑ 面向对象程序设计涉及到类、类的对象、方法、消息，

```
result = []  
for p in people {  
    if p.name.length > 5 {  
        result.add(p.name.toUpperCase);  
    }  
}  
return result.sort;
```



## 申明式设计(Declarative Programming)

- ❑ 在申明式程序设计中，控制流是隐式的
- ❑ 但是程序员需要说明最终的结果。
  - ❑ 没有赋值语句、没有循环，只有SELECT、FROM、WHERE和ORDER等特定的子句(subclause)

```
```SQL
select upper(name)
from people
where length(name) > 5
order by name
```
```



## 函数式程序设计(FP, functional programming)

- 下面是函数式编程的实例
- 很麻烦?

```
sort(
  fix(λf. λp.
    if(equals(p, emptylist),
      emptylist,
      if(greater(length(name(head(p))), 5),
        append(to_upper(name(head(p))), f(tail(p))),
        f(tail(people)))))(people))
```

```
let
  fun uppercasedLongNames [] = []
    | uppercasedLongNames (p :: ps) =
      if length(name p) > 5 then (to_upper(name
p))::(uppercasedLongNames ps)
      else (uppercasedLongNames ps)
in
  sort(uppercasedLongNames(people))s
```

```
people |> map (to_upper o name) |> filter (λs. length s > 5) |> sort
```

