Lecture14 名称、作用域、绑定

- Name：名称就是你想的那样，通常会想到标识符，但也可以更笼统一些
- Binding：绑定是两个事物之间的关联，比如名称和它所命名的事物
- Scope：绑定的作用域是程序中激活绑定的部分（从文本上讲）

1. 绑定 Binding

绑定无处不在

- 域名 -> IP 地址 -> NIC 和 MAC 地址
- 变量 -> 变量的值或引用（指针）
- 多态：通过方法查找表进行静态 / 动态绑定

```
SuperClass v = new SubClass();
v.methodA();
```

- `SuperClass` 可以是类或者接口
- 编译时，只检查 `SuperType` 对 `methodA()` 的声明
- 运行时，通过查找表从 `SubClass` 逐层往上进行查找

绑定的时间

绑定时间 Binding Time 是创建绑定的时间点

- **language design time**
  - 程序接口，可能的类型
- **language implementation time**
  - I/O，算术溢出，堆栈大小，类型相等
- **program writing time**
  - 算法，名称
- **compile time**
  - 数据布局规划
- **link time**
  - 整个程序在内存中的布局
- **load time**
  - 物理地址的选择
- **run time**
  - 值 / 变量绑定，字符串的大小
  - 包含
    - program start-up time
    - module entry time
    - elaboration time (point a which a declaration is first "seen")
    - procedure entry time

        - block entry time
        - statement execution time

静态绑定和动态绑定

- 术语**静态 static** 和**动态 dynamic** 通常分别用来指运行时之前和运行时绑定的内容
- 绑定时间与程序的设计实现密切相关
- 通常，绑定时间越早，效率越高，绑定时间越晚，灵活性越大
    - 编译语言往往有较早的绑定时间
    - 解释型语言往往具有较晚的绑定时间
- 我们主要讲的是根据变量的**名称**进行的绑定
    - 并不是所有的数据都被命名!例如，C 或 Pascal 中的动态存储是由指针引用的，而不是名称

生命周期

- 核心的事件
    - 对象的创建
    - 绑定的创建
    - 对变量的引用（使用绑定）
    - （临时）取消绑定
    - 绑定的重激活
    - 绑定的摧毁
    - 对象的摧毁
- 从**绑定的创建**到**绑定的摧毁**的这段时间叫做绑定的**生命周期 life time**
    - 如果对象离开了绑定，它就是一个**垃圾 garbage**
    - 如果绑定离开了对象，它就是一个**悬挂的引用 dangling reference**
- 程序中激活绑定的文本区域是它的**作用域 scope**

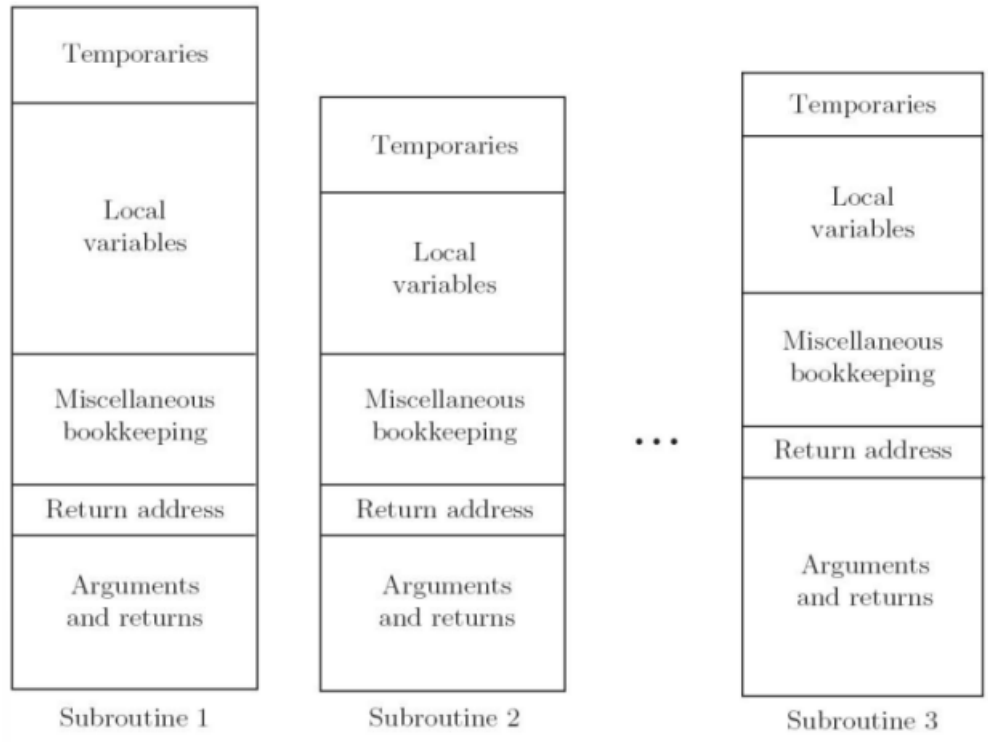
2. 存储分配机制

静态 Static

静态空间分配给

- code：代码
- globals：全局
- static：静态
- explicit constants：明确的常量 String, set 等
- scalars：可能存储在指令中的标量

在没有递归的语言或程序中，子程序的静态空间分配



栈空间 Stack

栈空间分配给

- parameters：参数
- local variables：局部变量
- temporaries：临时变量

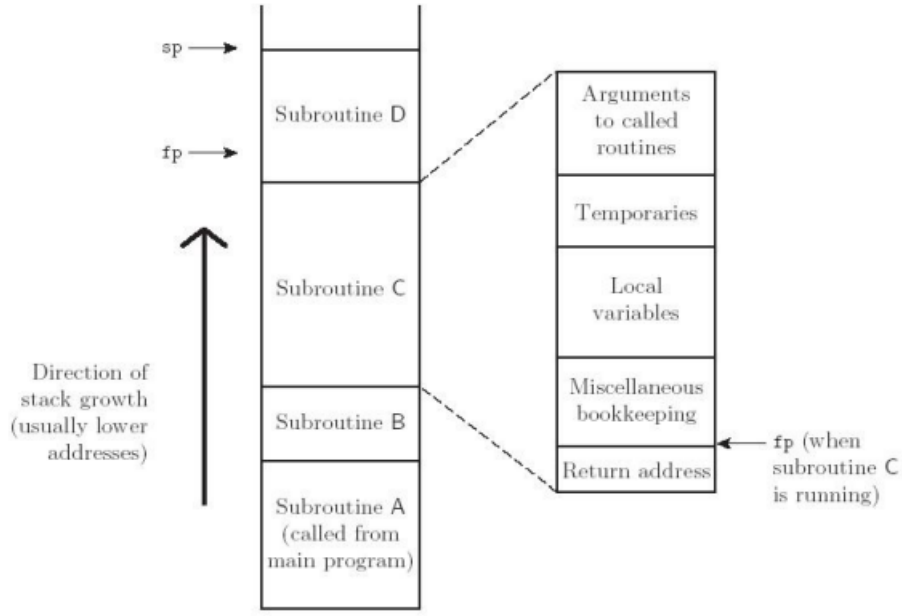
为什么要用栈?

- 为递归例程分配空间
  - 在旧的 FORTRAN 中不可能——没有递归
- 空间复用
  - 在所有编程语言中都有这个目的

栈帧 stack frame 的内容

- 参数和返回值
- 局部变量
- 临时变量
- 保存的寄存器、行号静态链接等

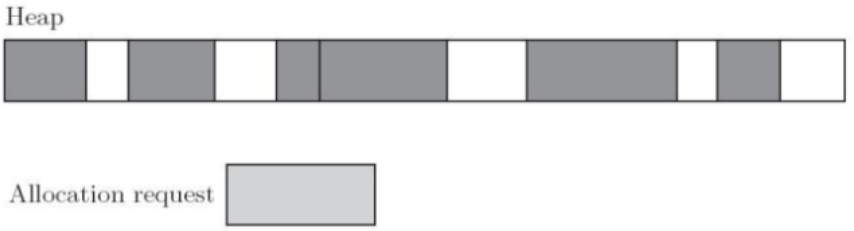
局部变量和参数在编译时从栈指针或帧指针被分配固定偏移量 offset



- **栈指针 sp** 指向栈区域中第一个没有使用的位置（再某些机器中，可能是最后使用的位置）
- **帧指针 fp** 指向当前子例程的帧（激活的记录）中的一个已知位置

堆空间 Heap

- **动态分配**和释放子块的内存区域
- 更加的非结构化
- 分配和再分配可以以任意顺序发生
    - 内存可能会碎片化
    - 需要**垃圾回收 garbage collection**
- 堆的结构
    - 通常使用一个单独的**链表**（空闲链表）来管理，这些链表是没有使用的块



- 如图，黑色的块是正在使用的，白色的块是没有使用的，尽管没有使用的总空间可以满足分配指定的块（灰色），但是没有一个单一的块（白色）是足够大的

3. 作用域 Scope

作用域介绍

- 作用域是**不允许更改绑定**或至少**不允许重新声明**的程序段的最大范围
- 在大多数有**子例程**的语言中，我们在子例程项上打开一个新的作用域
    - 为新的局部变量创建绑定
    - 对重新声明的全局变量禁用绑定（这些变量被称为： have a "hole" in their scope）

    - 引用变量
- 在**子例程中创建的绑定会在子例程退出时销毁**，当离开子例程的时候
    - 销毁局部变量的绑定
    - 重新激活已禁用的全局变量的绑定
- Modula, Ada 等模块，为您提供了**不受子例程生命周期限制的封闭的作用域**
    - 对模块中声明的变量的绑定在模块外部是不活动的，不会销毁
    - 同样的效果可以在许多语言中使用自己的术语来实现
        - C: 使用 static 声明的变量
        - Algol: 使用 own 声明的变量

静态作用域规则 Static Scope Rules

使用**静态static（词法 lexical ）作用域规则**，作用域是根据程序的**物理（词法）结构**定义的

- 作用域的确定可以由**编译器**来完成
- 标识符的所有绑定都可以通过检查程序来解析
- 通常，我们选择在编译时进行的**最近的，激活的**绑定
- 大多数编译语言，包括 C++ 和 Java，都使用静态作用域规则

静态作用域规则的经典例子是块结构语言中最紧密嵌套的规则，例如 Algol60 和 Pascal

- 标识符在声明它的作用域中和每个封闭作用域中都是已知的，除非它在封闭作用域中被重新声明
- 为了解析对标识符的引用，我们检查局部作用域和静态封闭作用域，直到找到绑定

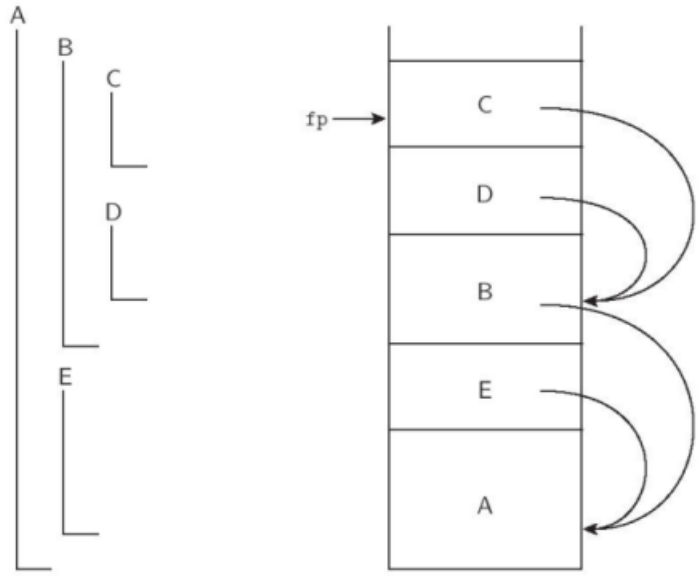
访问非局部变量 STATIC LINKS

使用静态链接 static links 访问非局部变量

如何进行访问

- 每一帧指针都指向声明它的例程（correct instance）所在的帧
- 你想访问一个在作用域 k 的变量，通过 k 个静态的链接找到，然后使用已知的在帧中的偏移量中找到

示例如下



- 子例程A、B、C、D 和 E 如左侧所示嵌套

- 如果运行时嵌套调用的序列是A, E, B, D 和 C，那么堆栈中的静态链接将如图所示
- 子例程 C 的代码可以在帧指针的已知偏移量处找到局部变量
- 它可以通过一次解引用它的静态链，然后应用一个偏移量来找到周围作用域 B 的局部对象
- 它可以找到 A 作用域内的局部对象，通过两次解引用它的静态链，然后应用一个偏移量

动态作用域规则 Dynamic Scope Rules

使用动态作用域规则，绑定依赖于程序执行的**当前状态**

- 它们不能总是通过检查程序来解决，因为它们依赖于**调用序列**
- 为了解析引用，我们使用在**运行时**进行的最近的激活的绑定

动态作用域规则通常在**解释语言 interpreted languages** 中遇到

这类语言通常不会在编译时进行类型检查，因为当动态作用域规则生效时，类型确定并不总是可能的

访问动态作用域下的变量

- 为所有激活变量维护一个**栈（关联列表）**
  - 当需要查找一个变量时，从栈顶部向下查找
  - 这相当于在动态链上搜索激活的记录
- 对于每一个变量的名称，维护一个槽 slot，保留一个中心表
  - 如果名称不是在运行时创建，表布局（以及每个槽的位置）可以在编译时固定
  - 否则，将需要一个哈希函数或其他东西来进行查找
  - 每个子例程在进入和退出（栈上的push / pop）时为其局部变量更改表项

别名 Aliasing

- 在给定作用域中引用单个对象的两个或多个名称称为别名
- 别名有什么好处?
  - 节省空间 —— 现代数据分配方法更好
  - 多重表达
- 此外，在参数传递时也会出现别名

```
1  public static void foo(MyObject x) {
2      x.val = 10;
3  }
4  public static void main(String[] args) {
5      MyObject o = new MyObject(1);
6      foo(o);
7  }
8
```

重载 Overloading

- 几乎所有语言都有一些重载
- 在符号表的帮助下处理
  - 查找请求名称的可能含义列表，语义分析器根据上下文选择最合适的一个
- 有些语言会在很大程度上重载

- Ada，C++

重载的类型

- **重载函数** —— 两个具有相同名称的不同函数
- **泛型函数** —— 一个可以在编译时以多种方式实例化的语法模板
  - 也称为显式参数多态性
  - 通过 C++ 中的宏处理器

单独编译 Separate Compilation

- 由于大多数大型程序都是增量地构造和测试的，有些程序可能非常大，因此语言必须支持单独的编译
- 编译单元通常是一个“模块 module”
  - Java / C++：Class
  - C++ 中的命名空间可以链接单独的类
  - C 语言更随意
  - Java 和 C#首先打破了所有方法/类都需要带有头信息的文件的标准