7 Foundations

这一讲是 recap 以前学过的内容并介绍一些更加 rigorous 的概念.

7.1 Machine Model 机器模型

一个 **name(名称)** 指 some entity such as a variable, function, or type.

一个 **varibale(变量)** 是一个指代 memory 中的一个 object 的 name. **(variable 就是一种 name！)**

一个 name 具有它的 **scope(作用域)**，scope 决定了什么区域的 code 能够使用该 name 来指代它表示的 entity.

一个 **declaration(声明)** 就是引入一个 name 的行为，并且 declaration 处是该 name 的 scope 的起点.

一个 **object(对象)** 就是 memory 中的 a piece of data.

object is locateda at some **address(地址)** in the memory.

object 的 **lifetime(生命周期)** 就是 legal to use this object 的时期. 一个 object 在某个时刻被 create，在某个时刻被 destroy，从被 create 到被 destroy 的这段时间就是我们能够 legallu use it 的时间. 它取决于该 object 的 **storage duration**.

为了学习 memory 中 objects 和用来指代它们的 variables 的变化，我们引入 **machine model**.

Machine Model 就是对于 codes 如何 relate to program execution 的一个描述.

最基础的 machine model 就是把 memory 看做一个像 array 的东西的图. 如下. 之后会有更复杂的 machine model

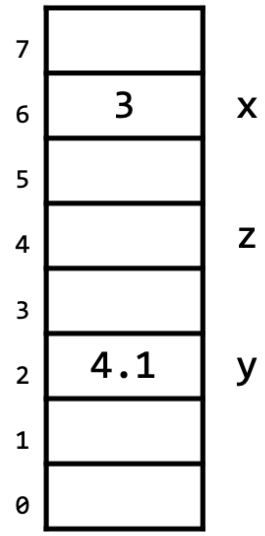


Figure 2.1: Basic machine model, with memory as a linear structure with slots for different objects.

现在我们开始讲 object 的储存周期.

7.1.1 Storage Duration

有三种 Storage Duration.

1. **static (也叫 Global 全局对象)**: 这种 object 的 lifetime 是 the whole program.
2. **automatic (也叫 Local 局部对象)**: 这种 object 的 lifetime 是被绑定于一个特定的 scope，比如某一段代码块.
3. **dynamic (动态)**: static 和 automatic durations 都是由 compiler 控制的，而 dynamic duration 却是由 programmer explicitly create and control 的.

本课程将只讲 static 和 automatic 的 object.

Variable 和 memory object 并不等同. Variable 是和 source code 相挂钩的概念，而 memory object 则是和 runtime 相挂钩的概念. **同一个 variable 可以在不同时间指代不同的 object.** 比如 a local variable in a functon that is called more than once. **Dynamic storage duration 的 objects 不会和 一个 variable 关联**.

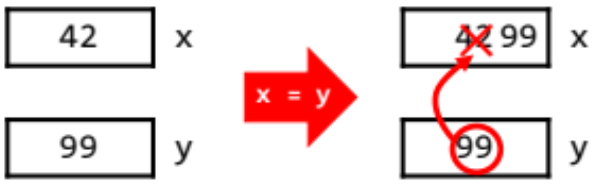
7.1.2 Reference sematics and Value sematics 引用语义与值语义

我们都已经学过了 pass by reference 和 pass by value 的区别. 而我们知道，C++ 中 `x = y` 仅表示 modify the value of object `x` 而不是 change which object `x` is referring to，

如果一个行为中 `x = y` 表示前者，那么这个地方就叫做 **value semantics(值语义)**；而如果表示后者，那么就叫做 **reference semantics(引用语义)**. 因而 C++ 这个语言的默认是 value sematics.

而 **C++ 仅在 intializating new variable 的时候支持 reference semantics**.

```
int x = 42; // initial x to refer to an object, and sets the initial value to 42.
int y = 99;
y = x; // assign the value of x:42 to y.
x = 55; // y is not becoming 55, because they are still pointing at different objects.
```

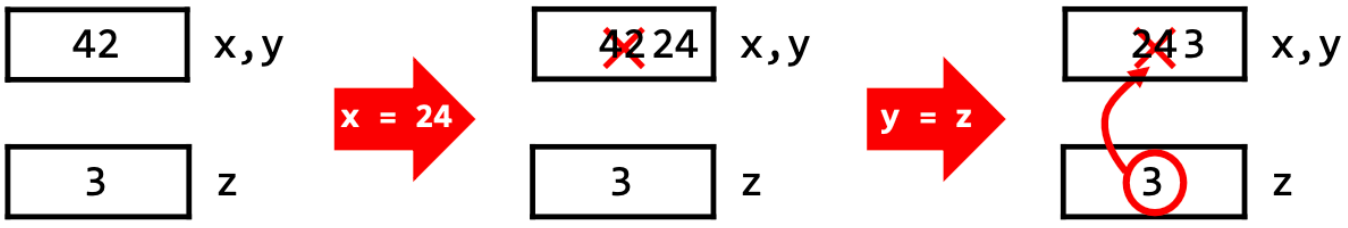


但是如果在 **new variable name** 的左侧加上一个 **ampersand (&)**, 那么就能够把该 name 关联到一个现有的 object.

我们都知道，这个符号叫做 reference operator（取地址运算符），放在一个代表某个 object 的 variable name 左侧，可以提取它所代表的 object 的 memory address 来存放在指针变量里；但是在 initialization 的时候如果在新 varible name 的左侧加上 &，则并不是表示提取 address 的运算，而是 "将我们左边这个新的 varibale name关联到现有的右边这个 variable name 代表的 object 上"，通俗而言就是给对象取个新名字. 这就是 reference operator 的另一个用法. 这个用法和写函数时 pass by reference 的用法是一样的，即直接指向这块 address 上的 object 对其进行赋名/运算.

```
int x = 42;
int &y = x;
//y和x是同一个object, 值都是42, y是给它起的新名字
x = 24;
cout << y; //24, 值被改变了
z = 3;
y = z; //pass by value
cout << x; //3, 值被改变了
```

这里，y 和 x 是两个 variable，但是他们都和同一个 object associate了. 因而改变其中任何一个的值，另一个也会改变，不如说 y 和 x 是称呼该 object 的两个名字.



并且**由于C++只支持在 initialization 时的 reference semantics, a variable 和 a memory object 之间的 association can never be broken! (except when variable goes out of scope).** 也就是说你一旦 declare 了一个 variable，那么它就和你 declare 时让它 refer 的 object 绑定了，**你无法重新给这个 variable 绑定新的 object**！你修改它就是修改这个 object.

而对于当出了 scope 之后，这个 name 就无效了，我们的 object 就少了一个名字. 比如一个函数的 pass by reference 的参数，就是对一个变量取的一个新名字，而在 `main()` 函数运行过了这个函数之后，这个名字就失去了. 因而 **pass-by-reference parameters 也叫做原 object 的一个 alias (别名)，可以理解为我们在 Windows 桌面创建的快捷方式.**

7.1.3 Compile time 和 Runtime

一个程序首先要 compile，其次再 execute (run).

Compile 一个程序是需要时间的. 这段时间里 compiler 会做一下的事情:

1. observe variable declarations: 观察代码里 variables 的 name, type, scope 和 value.
2. 确定需要给每个 type 的 variables 的 size.

   比如：

   int 需要 4 bytes

   double 需要 8 bytes

   更复杂的 ADT: n bytes

   (**注意：一个 byte 就是 8 个 bits，一个 bit 实际上就是 binary digits，就是一个 0/1 的二进制数**)

Compile time 就是 Compiler compile 你的 codes 的这段时间.

早在 compile time，compiler 就会决定好这段代码**需要多大的 memory.**

而 **runtime** 就是程序运行的时间. 这段时间 compiled code 被执行，variables 会以 objects 的形式被储存在 memory (RAM) 里，其大小就是这个 variable type 的大小.

（当然 variable 自身也会占据一定大小，就比如两个 int variable 都是同一个 object 的名字，那么这个 object 就只有 4 bytes，但是其实这两个名字也占据了一点 memory 使得它们能被识别。但这种东西我们就不管了。）

所以我们要注意的概念是：**object 只在 runtime 产生，variable 是 compile time 就被识别了。**

7.2 Function Calls and the Call Stack

7.2.1 Activation Record and stack 激活记录与栈

刚才我们介绍了最基本的把 memory 中的 object 看作类似 array 元素的 machine model，而现在我们介绍一个更加 structured 的 machine model，即被多种高级语言的 implementations 使用的：**Stack-based memory management （基于栈的内存管理）.**

在大多数高级语言的 implementations 中，函数调用的数据都集中存储在一个 **activation record（激活记录）**中，**activation record 就是给函数分配的 memory，分配给一个函数的 memory 就叫做 它的activation record** 其中包含函数的所有 parameters 和 local variables，temporary objects 即 return address等，而本课程中我们只考虑 activation record 中的 parameters 和 local variables，忽略其他数据.

在 stack-based memory management 中，activation record 被储存在一个叫做 **stack（栈）**的 data structure 中. 当创建一条 activation record 后，它就会被放在 **stack 的顶部**，而**第一个被销毁的 activation record 就是这条最新被创建出来的 activation record**. 这就是著名的 **last-in, first-out Behavior (LIFO，后进先出行为)**.

由于 activation record (more commonly)也叫做 **stack frame（栈帧）**.

所以知道了 activation record 的概念后，我们就大概知道高级语言程序的一些粗略的数据存储方式了. 比如我们的 C++ 程序总是从 `main()` 函数开始的，而所有在 `main()` 函数中的 local objects 都存在 `main()` 的 activation record 中，而 `main()` 又调用其他函数，运行时在栈顶为这些函数创造 activation record，而创造的**这个 activation record 的大小取决于它可能存储的局部变量数量. 就是说，这个 activation 的大小并不是自由改变的，而是从它被创建时就固定好的，根据里面的 local variable 数量决定（我们可能会觉得是 object 数量，但是并不是，而是 local variable 的数量，因为我们的理解基于简化模型，实际上作为 reference 的 variable 也占 memory，但是是 user 不能接触的.）**

7.2.2 Function Call 的过程

我们知道，C++ 的程序就是运行函数. 最开始的函数就是 `main()`； `main()` 会 call 其他函数，而其他函数又会 call 其他函数；运行完 `main()`，程序就结束了.

每一个 function call 的流程按照下面这个过程：

1. 当 function 运行到嵌套在里面的其他 function 时，evaluate actual **arguments** to the function.
2. Make a new and unique invocation of the called function.

(1) Push a stack frame
(2) Pass parameters

3. 暂停 original function 的运行
4. 运行 called function，可能会返回一些值 (optional)
5. 运行 called function 结束后，重新在 original function left off 的地方 resume.
6. destroy called function 的 stack frame.

注意，这里从 3 到 4 的过渡被称为一种 **active flow**，因为这个过程是 code explictly 告诉 computer 去运行哪个 function，而从第 4 步到第 5 步的过渡被称为一种 **passive flow**，因为这个过程的 transfer of control 并不是 code 告诉 computer, 而是 computer 在运行 stack frame 的过程中自主的.

7.2.3 Activation Record (Stack Frame) 的运行例

为了理解程序运行的流程，我们考虑这一个完整的程序作为第一个例子：

```
void bar(){
}

void foo(){
  bar();
}

int main(){
  foo();
}
```

这一程序都看得出来，是 `main()` 函数调用 `foo()` 这个函数，而 `foo()` 这个函数内部再调用 `bar()` 这个函数.

这个过程在 stack 中的样子是这样的：（不论哪个方向都一样的）

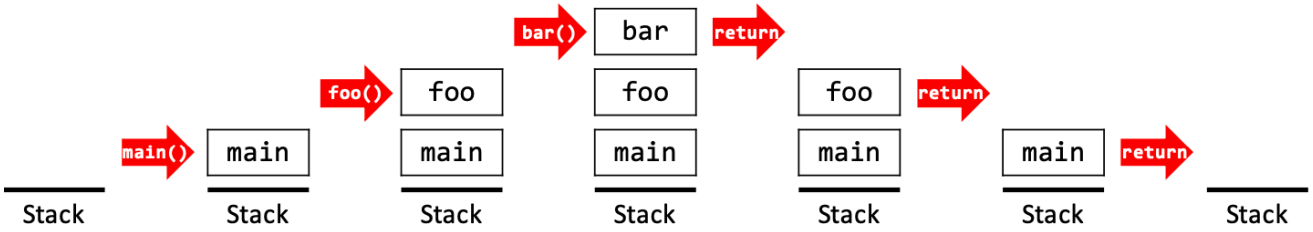


Figure 3.1: A stack that stores activation records.

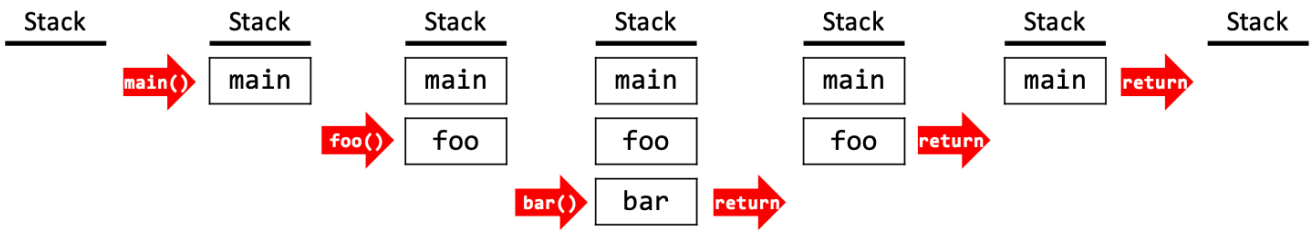


Figure 3.2: A stack that grows downward rather than upward.

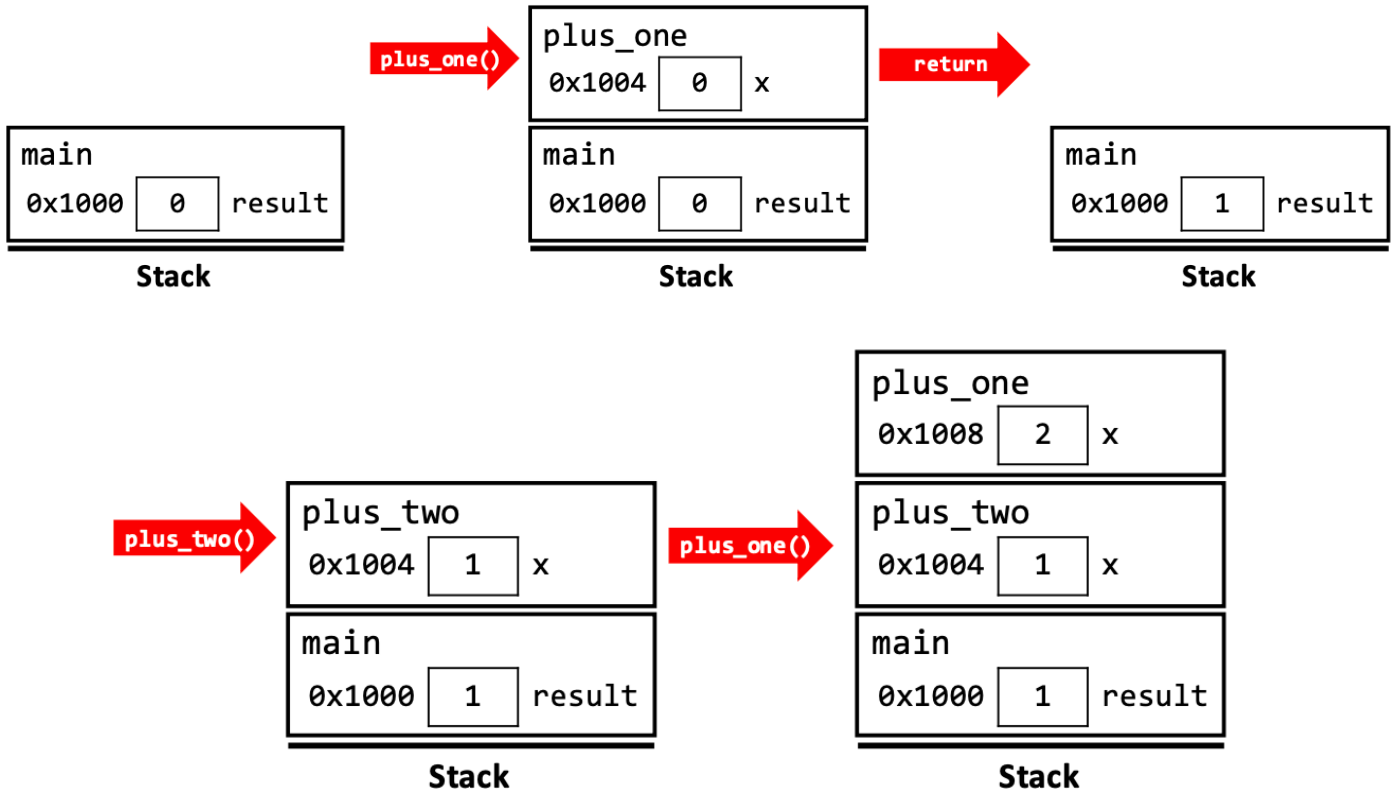
1. program 一开始就会调用 `main()` 函数，因而创建一条 `main()` 函数的 activation record 并放入栈顶；
2. 而后 `main()` 函数调用 `foo()` 函数，因而创造一条 `foo()` 函数的 activation record 并入栈，放在栈顶，`main()` 函数的上面；
3. 而后 `foo()` 函数调用 `bar()` 函数，同样创造 activation record 并入栈顶.
4. 最里层的 `bar()` 函数运行结束，从栈顶先移除它的 activation record；
5. 而后 `foo()` 函数运行完成，从栈顶移除 activation record；
6. `main()` 函数运行结束，栈被清空.

这是一个比较简单的过程，现在我们加入 variables 来进一步理解，看第二个稍微复杂一些的例子.

```
int plus_one(int x){
  return x + 1;
}
int plus_two(int x){
  return plus_one(x + 1);
}
int main(){
  int result = 0;
  result = plus_one(0);
  result = plus_two(result);
  cout << result;
}
```

program 首先调用 `main()` 函数，并先声明一个 local variable `result`（对应的 object 同时被创造，就是我们刚才说的仅在 variable 的 initialization 时的 reference semantics）；而后`main()` 函数调用 `plus_one()` 函数，`plus_one()` 生成其 scope 下的 local variable `x`，将调用这个函数的 `main()` 函数中的 local variable `result` 的值 pass 给自己的 local variable `x`；而后 `plus_one()` 运行结束返回 `x` 的值加上 1 并 assign 给 `result`，`result` 的值更新；而后 `main()` 函数又运行下一个函数 `plus_two()`.

而这一过程在 stack 中是这样的一个流程：



1. `main()` 函数被调用，`main()` 的 activation record 被创建.
2. `main()` 函数的执行过程中 local variable `result` 被声明，留出 memory 中的位置存放对应的 object，具体数据为 int 型的 1.
3. `plus_one()` 函数被 `main()` 函数调用，创建 `plus_one()` 的 activation record. 在被调用的同时，`plus_one()` 函数生成在它的 scope 中的 local variable `x`，并通过函数的 pass by value，获取调用它的 `main()` 函数里的 local variable `result` 的值，并 pass 给自己的 local variable `x`.
4. `plus_one()` 对自己的 local variable `x` 进行加一的操作，运行结束时把它的值作为函数的 return value，return 给调用自己的 `main()` 函数. `main()` 函数获取 `plus_one()` 的 return 值的同时，把它 assign 给 `main()` 的 local variable `result`.
5. 在完成 return 的过程后，`plus_one()` 运行结束并清除自己的 activation record，即出 stack. 现在 stack 中只剩下在栈底的 `main()` 函数的 activation record.

这就是从 program 开始运行一直到 `result = plus_one(0);` 这一行运行结束的全过程. 现在来到下一行 `result = plus_two(result);`. 从刚才的状态开始.

6. `main()` 调用 `plus_two()` 函数，`plus_two()` 的 activation record 被创建. **注意到，`plus_two()` 的 activation record 在 memory 中的位置会被安排到刚才被清除的 `plus_one()` 的 activation record 的位置上！** `plus_two()` 在创建 activation record 的同时也一样从 `main()` 中获取 `result` 的值并赋给自己初始的 local variable `x`.
7. `plus_two()` 调用 `plus_one()` 函数，创建 activation record，传参，把自己的local variable `x` 传给 `plus_one` 的 local variable `x`. 这里注意的是，我们可以发现**这两个函数的变量名都叫做 `x`，但是这并不会冲突，因为它们储存在不同的 activation record 中. 在 `plus_one` 的 scope 中 `x` 指代 `plus_one` 中的 object，而在 `plus_two` 的 scope 中指代 `plus_two` 中的 object. 所以程序识别 local variable name 其实是以一个 activaion record 为单位的，因由 scope 决定！多个 activation record 同时存在时，里面可以有相同的局部变量名，这也就是我们之前说的一个 variable name 绑定了一个 object 就不能绑定其他 object了，除非出 scope. 当然我们自然而然地想到 global variable 肯定不能这样.**
8. 之后各函数依次运行结束并清除 activation record 出栈.

7.2.4 pass-by-reference function 的 stack frame 运行例

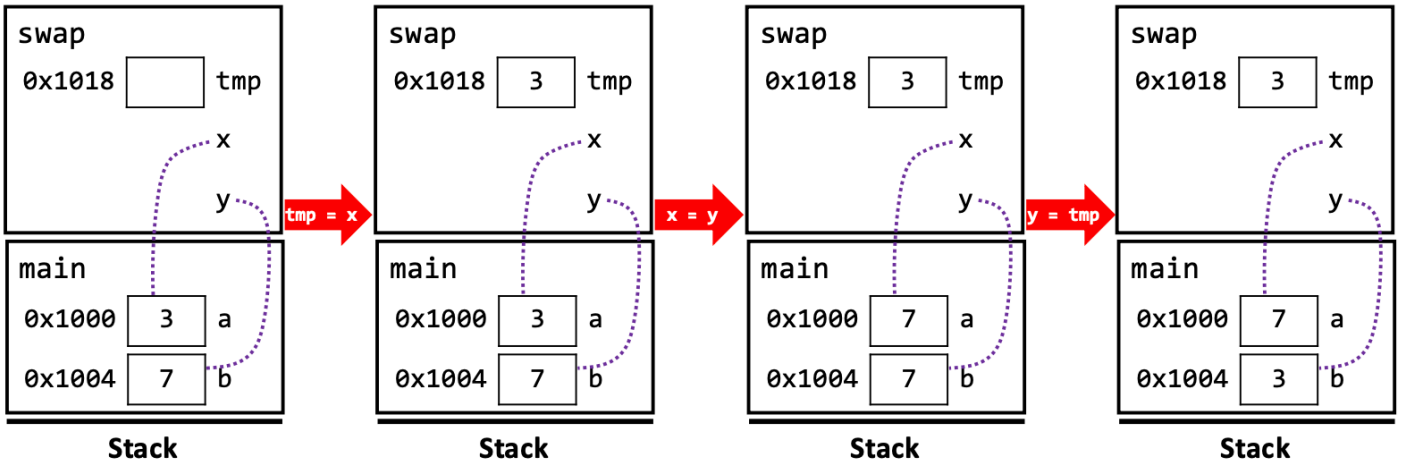
上面都是只有 pass by value 的 parameter 的函数，现在看第三个例子：一个有 pass by reference 的 parameter 的函数.

Pass-by-reference 就是在创建初始 variable 时用了 reference semantics. 实际上，**函数中传参就是用等号赋值！只不过这一步可以跨域 activation record 到调用本函数的另一个函数的 activation record 中去. 比如 `void swap(int &x, int &y);` 而我们在 `main()` 函数里 call 它，`swap(x,y);`. 这就是等于 `int &x = x;`, `int &y = y`, 由于是跨 scope 的所以这里同一个 variable `x` 可以指代两个不同的 object，完成这个赋值行为.**

按照我们之前学习的，`int &a = b` 就是给 `b` 指代的 object 取另一个名字叫做 `a`，因而 pass-by-reference 的 local variable 就是给调用自己的函数传过来的它的作为参数的 local variable 所指代的 object 取的一个新名字. 这一个传参过程是跨 activation record 的. 我们看下面这个例子.

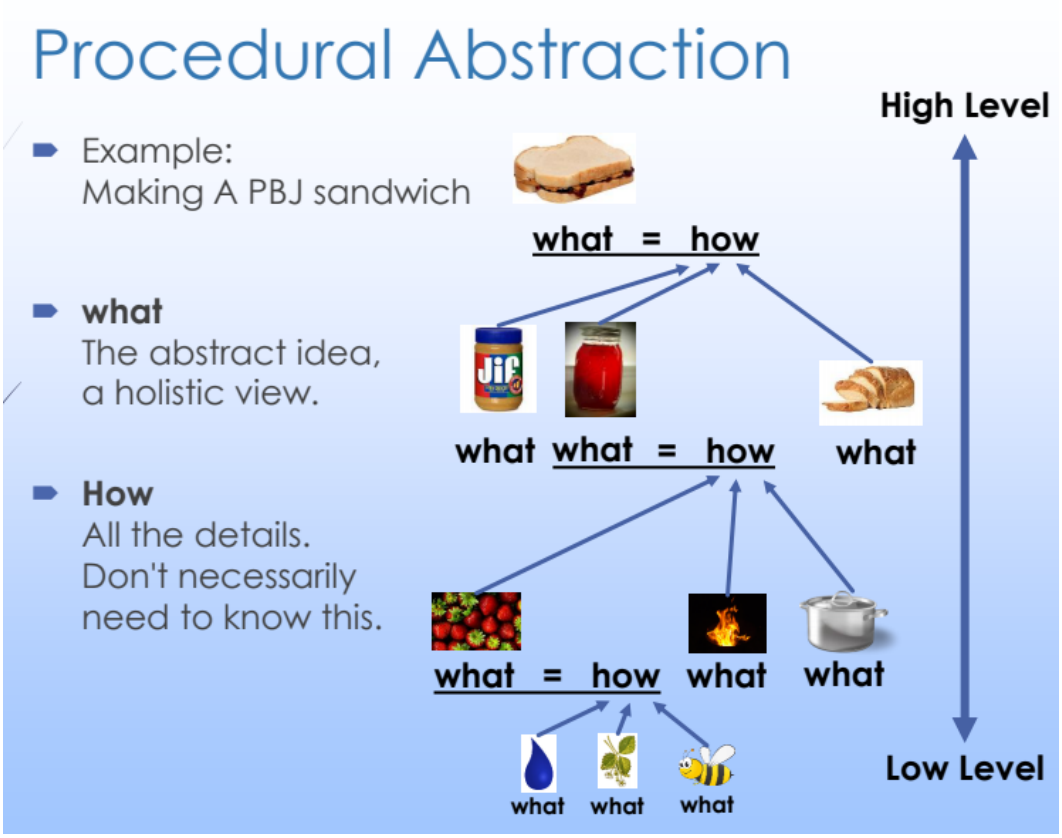
```
void swap(int &x, int &y){
  int tmp = x;
  x = y;
  y = tmp;
}

int main(){
  int a = 3;
  int b = 7;
  cout << a << "," << b << endl; //3,7
  swap(a, b);
  cout << a << "," << b << endl; //7,3
}
```

我们来综合刚才对 `&` 的学习以及对 call stack 的学习来看这一流程：



1. 程序创建 `main()` 函数的 activation record，大小包含两个 local variable的 space. `main()` 函数运行.
2. 运行到 `swap()` 时为其创建 activation record，大小包含三个 local variable，其中两个是 `main()` 的 local variable `a` 和 `b` 的 reference，另外一个 `tmp` refer to 自己的一个 temporary object.
3. `swap()` 进行我们熟悉的变量交换操作. 这里 `x` 和 `y` 都是引用参数，但我们要记得我们说的：reference semantics 仅在 initialization 时，而这里是 value semantics. 这里的 `x = y` 把 `y` (`b`) 所指代的 object 的值赋给 `x` (`a`) 所指代的 object. 自然地，这一过程在两个 activation record 中是同步进行的，因为动的是同两个 object.
4. 结束清除 activation record 而出栈.

7.3 Procedural Abstraction 过程抽象化



procedural abstraction 把 what code does 和 how it works 给分开了. **Functions 就是 C++ 中过程抽象化的主要工**

procedural abstraction 把 what code does 和 how it works 给分开了. **Functions 就是 C++ 中过程抽象化的主要工具.**

7.3.1 Interfaces 和 implementations

CS terminology 中，**interface** 指用来 specify what something does 的东西，而 **implementation** specifes how it works.

```
#include <iostream>
#include <vector>
using namespace std;
double mean(vector<double> v) //interface
{
  // implementations
}
int main() {
  vector<double> v;
  double m = mean(v); // only care about the interface here
  cout << m;
}
```

7.4 Unit Testing

7.4.1 Testing 的类型

Testing 有以下几个类型

1. Unit testing: One piece at a time (e.g., a function)
   它的好处是

   (1) Find and fix bugs early
   (2) Test smaller, less complex, easier to understand units.
2. System Testing: Test Entire project (code base)

   我们在 unit testing 之后应该进行 system testing 来确保代码的衔接正确
3. Regression testing: 在更改代码之后，Automatically run all unit and system tests.

这一节课提供了 `unit_test_framework.hpp`, 其中定义了一个 **macro(宏)** 来方便写 Unit Tests. 因而写 project 只需要把这个文件 include 进 testing 的 cpp 文件就可以了

```
#include "unit_test_framework.hpp"
```

7.4.2 Test Cases 的类型

7.4.2 Test Cases 的类型

1. Type prehibited: 类型不一致可能会导致 compiler error 的 Testing，**不需要写.**
2. Requires prohibited: 违反 requirements 的 tests. **不需要写.**
3. Simple: 一般的 test cases.
4. Edge (Special): 一些边缘问题，比如只有一个元素的 array 和 vector，测试众数的时候设置多个众数(应当取最小的)等.
5. Stress: 大型的数据，如几万个元素的vector. **这节课不需要写.**

7.4.3 Unit-Test Framework 教程

比如测试一个叫做 `arrays.cpp` 的文件，我们应该新建一个文件叫 `arrays_tests.cpp`. 在其中引用 framework.

如果其中包括两个叫做 `slideRight()` 的函数和 `flip()` 的函数，那么就在 `arrays_tests.cpp` 中写一些叫做 `TEST(test_slide_right_123)` 和 `` `TEST(test_flip_123) `` 这样的函数.

```
#include "arrays.hpp"
#include "unit_test_framework.hpp"

// A helper function for comparing arrays. Returns true if the
// arrays are the same size and contain the same values.
bool compare_arrays(int arr1[], int size1, int arr2[], int size2) {
  if (size1 != size2) {
    return false;
  }

  for (int i = 0; i < size1; ++i) {
    if (arr1[i] != arr2[i]) {
      return false;
    }
  }

  return true;
}

// We define a test case with the TEST(<test_name>) macro.
// <test_name> can be any valid C++ function name.
TEST(test_slide_right_1) {
  int testing[] = { 4, 0, 1, 3, 3 };
  int correct[] = { 3, 4, 0, 1, 3 };
  slideRight(testing, 5);
  ASSERT_TRUE(compare_arrays(testing, 5, correct, 5));
}

TEST(test_flip_1) {
  int testing[] = { 4, 0, 1, 3, 3 };

  int correct[] = { 3, 3, 1, 0, 4 };
```

```
  int correct[] = { 5, 3, 1, 0, 4 };
  flip(testing, 5);
  ASSERT_TRUE(compare_arrays(testing, 5, correct, 5));
}

TEST_MAIN() // No semicolon! 这就是宏中定义的语法
```

在写完之后，用编译器制作出 .exe 文件

```
g++ -Wall -Werror -pedantic -g -std=c++11 arrays.cpp arrays_tests.cpp -o arrays_tests.exe

./arrays_tests.exe

# Running test: numbers_are_equal
# PASS
# Running test: true_is_true
# PASS

# *** Results ***
# ** Test case 'numbers_are_equal': PASS
# ** Test case 'true_is_true': PASS
# *** Summary ***
# Out of 2 tests run:
# 0 failure(s), 0 error(s)
```

这个 framework macro 最大的特点是：还可以用制作出来的 .exe 只跑其中的一个函数.

```
./arrays_tests.exe numbers_are_equal
# Running test: numbers_are_equal
# PASS

# *** Results ***
# ** Test case 'numbers_are_equal': PASS
# *** Summary ***
# Out of 1 tests run:
# 0 failure(s), 0 error(s)
```

7.4.4 Framework macro提供的特殊 Assert 函数

Assertion	Description
`ASSERT_EQUAL(first, second)`	相等
`ASSERT_NOT_EQUAL(first, second)`	不等
`ASSERT_TRUE(bool value)`	true
`ASSERT_FALSE(bool value)`	false
`ASSERT_ALMOST_EQUAL(double first, double second, double precision)`	$\|a-b\| \leq \epsilon$
`ASSERT_SEQUENCE_EQUAL(first, second)`	arrays, vectors, lists 相等