Adventures in C++ Functions

Prototype

- 函数原型中的变量名不需要和定义中的变量名一样，而且可以省略：

```
void n_chars( char, int );      // valid
```

- 函数调用时如果类型不匹配，C++会自动将传递的值转换成原型中指定的类型，条件是两者都是**算式类型**：
  - 如果函数原型中需要 `int` 数据，但是传递时给了 `double`，则函数只会检查64位中的前16位，并试图将它解释成一个 `int` 值；
  - 当且仅当有意义时，原型化才会导致类型转换（`eg.` 不会将整型转换成结构或指针）

Functions and Arrays

在大多数情况下，C++将数组名视为指针，但是有三个情况例外：

1. 数组声明使用数组名来标记存储位置；
2. 对数组名使用 `sizeof` 将得到整个数组的长度(以字节为单位)；
3. 将地址运算符 `&` 用于数组名时，将返回整个数组的地址。

The Implications of Using Arrays as Arguments

```
/* In function prototype */
int sum_arr( int arr[], int num );  // 两者等价
int sum_arr( int* arr, int num );   // 两者等价
```

将数组作为参数：

- 传递数组时：传递数组的地址，包含的元素类型以及元素数目，函数**使用原来的数组**；
- 传递常规变量：函数将使用**拷贝的变量**，而不是原变量。

Functions and Two-Dimensional Arrays

```
int data[3][4] = {{1,2,3,4}, {9,8,7,6}, {2,4,6,8}};
int total = sum(data, 3);
/* Prototype of sum */
int sum( int (*arr)[4], int size );     // 两者等价
int sum( int arr[][4], int size );      // 两者等价
int sum( int *arr[4], int size );       // 错误的❌
```

- `data` 的类型是指向 4 个 `int` 组成的数组的指针(数组指针)——`data` 是一个数组名，该数组有 3 个元素，每一个元素都是一个有 4 个 `int` 值得数组；
- 指针类型指出：它指向由 4 个 `int` 值组成的数组，其指定了列数，所以这就是没有将列数作为单独参数进行传递的原因；

- ```
  arr             // pointer to first row of an array of 4 int.
  arr+r           // pointer to r^th row (an array of 4 int).
  *(arr+r)        // a pointer to the first int in row r(i.e. arr[r])
  *(arr+r)+c      // pointer int number c in row r(i.e. arr[r]+c)
  *(*(arr+r)+c)   // value of int nmber c in row r(i.e. arr[r][c])
  ```

Pointers and `const`

> Const applies to the thing left to it. If there is nothing on the left, then it applies to the thing right of it.——const 默认作用于其**左边的东西**，否则作用于其右边的东西**(最靠近的那个)**

> Const 关键字的作用以 `*` 为界，在左是指针指向的值不能被修改，在右是指针指向的地址不能修改——C primer plus

主要需要知道的就是 `const int x` 和 `int const x` 是表达的同一个意思 

1. `const int *`: a pointer to a const `int`.
   - `const` 只有右边有东西，所以 `const` 修饰 `int` 成为常量整型，然后 `*` 再作用于常量整型
   - `(const int) (*)`
2. `int const *`: a pointer to a const `int`.
   - `const` 左边有东西，所以 `const` 修饰 `int`，然后 `*` 再作用于 `int const`.
   - `(int const) (*)`
3. `int* const`: a const pointer to an `int`.
   - `const` 左边有东西，所以 `const` 作用于指针，不可以改变指向的地址.
   - `(int) (*<-const)`
4. `const int* const`: a const pointer to a const integer.
   - 左边的 `const` 左边没东西，所以作用于右边的 `int`；右边的 `const` 作用于 `*`.
   - `(const int) (*<-const)`
5. `int const * const`: a const pointer to a const integer.
   - 两个 `const` 左边右边都有东西，左边的作用于 `int`，右边的作用于 `*`.
   - `(int const) (*<-const)`
6. `int const * const * const`: a const pointer to a const pointer to a const integer.

```
int age = 39;
const int* pt = &age;
*pt += 1;    cin >> *pt;      // Both are invalid.
```

- 注意：`pt` 的声明并不意味着它指向的 `age` 实际上就是一个常量，而只是意味着对 `pt` 而言这个值是常量；我们可以通过直接修改 `age` 变量来修改 `age` 的值.

```
const float g_earth = 9.80;
const float* pe = &g_earth;     // Valid.

const float g_moon = 1.63;
float* pm = &g_moon;            // Invalid.
```

- 将 `const` 变量的地址赋值给指向 `const` 的指针是**对的**;
- 将 `const` 变量的地址赋值给常规指针是**错的**.
```

```
const int months[12] = {...};
int sum(int arr[], int n);          // Invalid.
int sum(const int arr[], int n);    // Valid.
```

- 如果有一个由 `const` 数据组成的数组，那么将**不能将数组名作为参数传递给使用非常量形参的函数**，这就是为什么上述第二行是错误的原因.

```
const int **pp2;
int *p1;
const int n = 13;
pp2 = &p1;          // Not allowed, but suppose it were
*pp2 = &n;          // valid, both const, but set p1 to point at n
*p1 = 10;           // valid, but change const to n
```

- 两级间接关系时：将 `const` 和非 `const` 混合的指针赋值方式将不再安全.

Pointers to Functions

假设要设计一个 `estimate()` 函数来估算编写的代码所需的时间，并且希望不同的程序员都将使用该函数，对于所有的用户来说，`estimate()` 中一部分代码都是相同的，但该函数允许每个人提供自己的算法来估算时间。

1. **获取函数的地址：**不跟参数的函数名就是地址(`eg.` 函数 `think()` 的地址就是 `think`)
    - 要将函数函数作为参数进行传递，必须传递函数名：

    ```
    process(think);     // passes address of think() to process()
    ```

2. **声明函数指针：**声明应指定函数的返回类型以及函数的参数列表，像函数原型一样指出有关信息
    - ```
      double pam(int);     // Prototype
      double (*pf)(int);   // pf points to a function that takes one int
                           // argument and that returns type double.
      ```

      和 `pam()` 声明类似，只是将 `pam` 换成了 `(*pf)`，由于 `pam` 是函数，因此 `(*pf)` 也是函数，`pf` 是函数指针.
        - **要声明指向特定类型的函数的指针，首先编写这种函数的原型，然后用 `(*pf)` 替换函数名，这样 `pf` 就是这类函数的指针 (注意一定要括号 `()`)**

        ```
        double (*pf)(int);  // pf points to a function that returns
        double.
        double *pf (int);   // pf() a function that returns a pointer-
        to-double, same as double* pf(int);
        ```

3. **使用函数指针来调用函数：** `(*pf)` 和函数名等价，因此使用 `(*pf)` 时只需要将他看作函数名即可，但是直接使用 `pf` 也可以
        ```

```
double pam(int);        // prototype
double (*pf)(int);      // pointer to function
pf = pam;               // pf now points to the pam() function
double x = pam(4);      // call pam() using the function name
double y = (*pf)(5);    // call pam() using the pointer pf
double z = pf(6);       // also call pam() using the pointer pf
```

上述第5，6行的调用方法的逻辑完全不一致，但是C++都是支持的.

4. 具体代码实例在 `ubuntu` 里面.

Advanced pointer to functions

```
const double* f1(const double ar[], int n); // ar[] <==> *ar
const double* f2(const double [], int);     // 函数原型中可以省略标识符
const double* f3(const double *, int);      // 这三个函数原型都是等价的
```

可以在声明函数指针的同时初始化：

```
const double* (*p1)(const double *, int) = f1;
```

也可以使用 `auto` **自动类型推断**功能，但要注意**只能用于单值初始化**不能用于数组之类的：

```
auto p2 = f2;        // c++ automatic type deduction
cout << (*p1)(av, 3) << *(*p1)(av, 3) << endl;
cout << p2(av, 3) << *p2(av, 3) << endl;
```

- `(*p1)(av, 3)` 和 `p2(av, 3)` 都指向被调用的函数 `f1()` 和 `f2()` 并将 `av, 3` 作为参数，因此两句前半部分显示的都是两个函数的返回值，类型为 `const double*`.
- `*(*p1)(av, 3)` 和 `*p2(av, 3)` 都对 `const double*` 指针做了解引用，即查看这些地址处的实际值.

An array of function pointers

```
const double* (*pa[3])(const double *, int) = {f1, f2, f3};
```

运算符 `[]` 的优先级高于 `*`，因此 `*pa[3]` 表明 `pa` 是一个包含三个指针的指针数组，上述声明的其他部分指出了每个指针都是函数指针指向：参数列表为 `const double*, int`，返回类型为 `const double*` 的函数 (这里不能用 `auto` 因为其只能用于单值初始化).

```
const double* px = pa[0](av, 3);        // use it.
const double* py = (*pa[1])(av, 3);     // use it.
```

A pointer to the whole array

```
auto pc = &pa;      // C++11 automatic type deduction
const double* (*(*pd)[3])(const double *, int) = &pa;
```

- `pd` 是一个指针，指向一个包含三个元素的数组；
- `pd` 指向数组，`*pd` 就是数组，`(*pd)[i]` 是数组中的元素即函数指针，所以要调用函数：
  - `(*pd)[i](av, 3)` 是调用函数，`*(*pd)[i](av, 3)` 是函数返回指针指向的那个值；

- `(*(*pd)[i])(av, 3)` 是调用函数，`*(*(*pd)[i])(av, 3)` 是那个值。
- ```
  **&pa == *pa == pa[0]
  ```
  - `pa` 是数组名，是第一个元素的地址，即 `&pa[0]` 是单个指针的地址；`&pa` 是整个数组的地址，虽然它们的值一样，但是它们的类型不一样.

Inline function

An inline function is **expanded in place**, like a pre-processor macro, so the overhead of the function call is eliminated(消除了函数调用的开销).

```
inline int plusOne(int x) {return ++x;}
```

- The "definition" of an inline function should be **put in a header file**. Then `#include` it where the function is needed.
  - Never be afraid of multi-definition of inline functions, since they have no body at all.
- An inline function definition may **not generate any code in `.obj` file**.
- It is **declaration** rather than definition.
- Any function you define inside a class declaration is automatically an inline.
- 通常的做法是省略原型，将整个函数的定义放在原本提供原型的地方；
- 在**函数声明或者函数定义**前加上关键字 `inline`.
- C++ 的内联功能远胜于 C 语言的宏定义；

Reference Variables

引用是已定义的变量的别名，其主要用途是作为函数的形参，将引用变量用作参数使得函数将使用原始变量，而不是副本；除指针之外，引用也是函数处理大型结构提供了一种非常方便的途径.

Create Reference Variables

```
int rats;
int & rodents = rats;   // rodents a reference, an alias(别名) for rats
int *prats = &rats;     // prats a pointer
```

- `int &` **是一种类型声明**，即指向 `int` 变量的引用；
- `rats` 和 `rodents` 的**值和地址都相同**，对其中的一个改变都将影响两个变量；
- 引用和指针的区别：
  - 必须在**声明**引用变量时进行**初始化**：

    ```
    int rats;
    int & rodents;
    rodents = rats;     // Invalid, we can't do this
    ```
    ```

    - 引用更接近 `const` 指针，必须在创建时进行初始化；
  - when a reference pledges its allegiance to a particular variable, it sticks to its pledge. **一旦与某一个变量关联起来，就将一直效忠于它.**

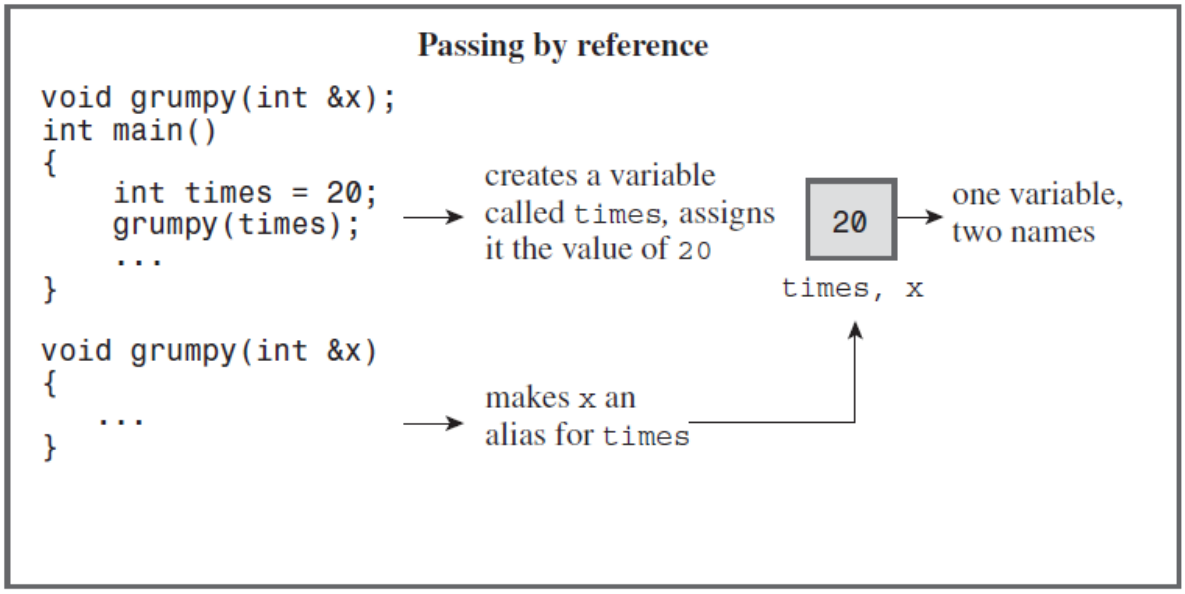
```
int rats = 101, bunnies = 50;
int & rodents = rats;
rodents = bunnies;      // Can not change the reference.
```

`rodents = bunnies` 等价于 `rats = bunnies`，所以现在 `rats = 50`.

  - **可以通过初始化来设置引用，但不能通过赋值来设置.**

References as Function Parameters

引用常被用作**函数参数**，使得**函数中的变量名称**为调用**程序中的变量的别名**，按引用传递允许被调用的函数能够访问调用函数中的变量.



Temporary Variables, Reference Arguments, and `const`

```
double square( double &ra ) {
    ra *= ra;
    return ra;
}
square( x+1.0 );        // Invalid, 因为表达式 x+1.0 不是变量，所以会报错
```

如果实参和引用参数**不匹配**，C++将生成**临时变量**，但当且仅当**参数为 `const` 引用**时才成立，如果参数是 `const` 引用，只有在下面两种情况才会生成临时变量：

1. 参数的类型匹配正确，但不是**左值 (lvalue)**：
   - 左值参数：可以被引用的数据对象，**非左值对象**包括**字面常量**(引号括起的字符串除外)和包含**多项的表达式** (`eg. x+1.0`).
   - 常规变量和 `const` 变量都可以视为左值，因为可以通过地址访问它们.
2. 实参的类型匹配不正确，但可以转换成正确的类型：
   - `eg. long egde = 5L` 然后调用 `double c = square( edge )`，类型不正确但是可转换

- 在上面的两种情况下编译器都将生成一个**临时匿名变量，**并让 `ra` 指向它，这些临时变量只在函数调用期间存在，此后编译器便将其随意删除.
- 对于形参为 `const` 引用的 C++ 函数，如果参数不匹配，则其行为类似于按值传递，为确保原始数据不被修改，将使用临时变量来存储值.

返回引用时需要注意的问题：

```
const free_throws & clone( free_throws & ft ) { // free_throws 是一个结构
    free_throws newguy;
    newguy = ft;
    return newguy;
}
free_throws n = clone(ft);      // wrong.
```

因为函数中的临时变量 `newguy` 在函数结束后将不复存在，这个内存单元不存在引用。

rvalue reference

使用 `&&` 声明指向右值的引用，字面常量和含多项的表达式：

```
double && rref = std::sqrt(36.00);  // not allowed for double &.
double j = 15.0;
double && jref = 2.0 * j + 18.5;    // not allowed for double &.
std::cout << rref << std::endl;     // display 6.0
std::cout << jref << std::endl;     // display 48.5
```

Using References with a Class Object

将**类对象**传递给函数时，C++ 通常的做法是**使用引用**，把类的对象作为参数.

- 将C-风格字符串用作 string 类对象引用参数：
  - string 类的属性：string 类定义了一种 `char*` 到 `string` 的转换功能，可以使用C-风格字符串来初始化 `string` 对象；
  - `const` 引用的形参的属性：假设实参类型和引用参数类型不匹配，但可被转换为引用类型，程序将创建一个正确类型的临时变量，使用转换后的实参值来初始化它，然后传递一个指向该临时变量的引用；
    - 所以可以将实参 `char*` 或 `const char*` 传递给形参 `const string &`.
    - 如果形参类型为 `const string &`，在调用函数时，使用的对象可以是 `string` 对象或C-风格字符串：引号括起的字符串字面量、以空字符结尾的 `char` 数组、指向 `char` 的指针变量.
  -

Objects, Inheritance, and References

Inheritance: the language feature that makes it possible to pass features from one class to another is called inheritance——`eg.` `ostream` is termed a **base class**, `ofstream` is termed a **derived class**.

- Derived class 继承了所有 base class 的方法，这意味着派生类的对象可以使用基类的方法.
- A base class reference can refer to a derived class object without requiring a type cast(基类引用可以指向派生类对象)
  - 所以可以定义一个接受基类引用作为参数的函数，而在调用函数时，既可以将基类对象作为参数，也可以将派生类对象作为参数——`eg.` 参数类型为 `ostream &` 的函数可以接受

`ofstream` 对象.

-

`ostream` formatting methods: `setf()`

- 设置之后保持不变，直到再次调用相应的方法重新设置：
  - `setf(ios_base::fixed)`：将对象置于使用定点表示法的模式；
  - `setf(ios_base::showpoint)`：将对象置于显示小数点的模式，即使小数部分为零；
  - `precision()`：指定显示多少位小数(假定对象处于定点模式下)；
- 只在下一次显示有效的设置：
  - `width()`：设置下一次输出操作使用的字段宽度，默认的字段宽度为 0，刚好容纳下要显示的内容；

Default Arguments

A default argument is a value given in the declaration that the compiler automatically inserts if you don't provide a value in the function call.

- To define a function with an argument list,defaults must be **added from right to left.**

```
int harpo(int n, int m = 4, int j = 5);
int chico(int n, int m = 6, int j);             //illegal
int groucho(int k = 1, int m = 2, int n = 3);
```

- 实参按照从左到右的顺序依次被赋给相应的形参，而不能跳过任何参数：

```
beeps = harpo(3, , 8);      // 错的
```

Function Overloading

函数重载的关键是函数的**参数列表**——也称为**函数特征标 (function signature)**：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，变量名是无关紧要的.

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**参数数目和 / 或参数类型**不同，则特征标也不同.

- 编译器在检查函数特征标时，将把类型引用和类型本身视为同一个特征标：

```
double cube( double x );
double cube( double & x );
cout << cube(x);            // 引起混乱
```

- 匹配函数时，并不区分 `const` 和非`const` 变量：

```
void dribble(char * bits);          // overloaded
void dribble (const char *cbits);   // overloaded
void dabble(char * bits);           // not overloaded
void drivel(const char * bits);     // not overloaded
```

Here's what various function calls would match:

```
const char p1[20] = "How's the weather?";
char p2[20] = "How's business?";
dribble(p1);       // dribble(const char *);
dribble(p2);       // dribble(char *);
dabble(p1);        // no match
dabble(p2);        // dabble(char *);
drivel(p1);        // drivel(const char *);
drivel(p2);        // drivel(const char *);
```

- **特征标不同**是函数可以重载的**必要条件**:

```
double sam(int n, float m);     // same signatures,
long sam(int n, float m);       // hence not allowed.
```

  - 返回类型可以不同，但是特征标必须也不同才可以算是重载，只有返回类型不同不可以算作重载》

Function Templates

它使用泛型来定义函数，其中的**泛型可用具体的类型替换**，通过将类型作为参数传递给模板，可使编译器生成该类型的函数；由于类型是参数表示的，因此模板特性也称为**参数化类型。**

```
/* Declaration */
template <typename AnyType>     // <class AnyType> 也是可以的
void swap( AnyType &a, AnyType &b );

/* Definition */
template <typename AnyType>     // <class AnyType> 也是可以的
void swap( AnyType &a, AnyType &b ) {
    AnyType temp;
    temp = a;
    a = b;
    b = temp;
}
```

- 关键字 `template` 和 `typename` 是必需的，其中 `typename` 可以和关键字 `class` 互换两者等价;
- 类型名(`AnyType`)可以任意选择，只要遵守C++命名规则就可以，但是尖括号 `< >` 是必需的;
- 在调用的时候无需显式说明要用什么类型，只要将对应类型的实参传递给函数就可以了;
- 函数模板并不能缩短可执行程序，只是缩短了源代码，常见的情形是将模板放在头文件中，并在需要使用模板的文件中包含头文件;

Overloaded Templates

类似常规函数重载一样，重载的模板的函数特征标必须不同：

```
template <typename T>
void swap( T &a, T &b );        // Original template
template <typename T>
void swap( T *a, T *b, int n ); // new template
```

Explicit Specializations

如果我们现在有一个结构，而我们只想交换结构中的某一个成员，于是就无法使用上面的 `swap()` 模板了，然而我们可以提供一个具体化函数定义——**显式具体化(explicit specializations)**：

- 对于给定的函数名，可以有非模板函数、模板函数和显式具体化模板函数以及它们的重载版本；
- 显式具体化的原型和定义应以 `template<>` 打头，并通过名称来指出类型；
- 具体化优先于常规模板，而非模板函数优先于具体化和常规模板；

```
struct job {
    string name;
    double salary;
}
// non template function prototype
void swap( job &a, job &b );

// template prototype
template <typename T>
void swap( T &a, T &b );

// explicit specialization for the job type
template <> void swap<job>( job &a, job &b );

// Definition of swap just the salary field of job structure
template <> void swap<job>( job &a, job &b ) {
    double temp;
    temp = a.salary;
    a.salary = b.salary;
    b.salary = temp;
}
```

```
template <> void swap<job>( job &a, job &b );
template <> void swap( job &a, job &b );        // simple form.
```

Instantiations and Specializations

代码中包含的函数模板本身并不会生成函数定义，它只是一个用于生成函数定义的方案。

- **隐式实例化 (implicit instantiation)**：编译器使用模板为特定类型生成函数定义时，得到的是模板实例 (Instantiation)，在调用模板函数的时候提供了特定的类型编译器就生成对应类型的函数定义.
- **显式实例化 (explicit instantiation)**：直接命令编译器创建特定的实例，如 `swap<int>()`，其语法是声明所需的种类——用 `<>` 符号指定类型，并在声明前加上关键字 `template`.

```
template void swap<int>( int, int );    // explicit instantiation
                                        // 使用 swap() 模板生成一个使用 int 类型实例
```

- **显式具体化 (explicit sppecialization)**：不使用模板来生成函数定义，而应使用专门为某一类型显式地定义地函数定义

```
template <> void swap<int>( int &, int & );    // explicit sppecialization
template <> void swap( int &, int & );          // explicit sppecialization
```

  - 显式具体化声明在关键字 `template` 后面包含 `<>`，而显式实例化没有；
  - 显式具体化地声明必须有相应的函数定义；