Structural-Programming

Function

原则

- 定义不允许嵌套
- 先定义后使用

函数的执行机制

- 建立被调用函数的栈空间
- 参数传递
  - 值传递 (call by value)
  - 引用传递 (call by reference)
- 保存调用函数的运行状态
- 将控制转交被调函数

函数重载

- 原则
  - 函数名相同，参数不同(类型，个数，顺序)
  - 返回值不作为区分的标准
- 匹配原则
  - 严格匹配(参数)
  - 内部转换(参数)(隐式)
  - 用户自定义的转换

sample:

```
void f(int x){
    cout << "int" << endl;
}
void f(long x){
    cout << "long" << endl;
}
void g(double x){
    cout <<"double" << endl;
}
void g(string s){
    cout << "string" << endl;
}
void h(char c){
    cout << "char" << endl;
}
void h(double x){
    cout << "double" << endl;
}
int main(){
```

```
    f(1);//int, 这是严格匹配
    g(1);//double, 这是内部转换
    f(1L);//long, 这是用户定义的转换
    h(1);//error, ambiguous call
    return 0;
}
```

默认参数

- 语法 : 默认参数放在非默认参数后面(why ? 非默认参数需要值，要先匹配完)
- 默认参数可能导致模糊的函数重载

```
void f(int);
void f(int, int=2);
//ambiguous
```

内联函数 inline

- 目的 : 提高可读性，提高效率(函数调用需要开销)
- 实现方法 : 编译系统将为 inline 函数创建一段代码，在调用点，以相应的代码替换
- 限制 : 不能递归
- 适用 : 使用频率高，简单，小段代码
- 注意:
    - 只是请求 inline，编译系统不一定会采纳
- 缺点:
    - 增大目标代码
    - 病态的换页
    - 降低 cache命中率

Program organization

原则

- 分成头文件(.h)按和源文件(.cpp)
- 头文件中放: 常量定义，变量、函数声明，类型定义，编译预处理，内联函数

使用 namespace

两种使用方式

declaration

```
namespace L{
    int k;
    void f();
}
using namespace L::k;
using namespace L::f;
k = 0;
f();
```

directive

```
using namespace L;
k = 0;
f();
```

不建议多次使用 using-directive, 如果没用到很多尽量 using-declaration

Array

特征

- 存储相同类型的元素
- 存储空间连续

一维数组

- 类型定义 `T a[]`
- 函数接口 `void f(T a[], int n)` 数组大小需要当作参数传递，不能使用 `sizeof` (数组类型变量参与表达式就退化成指针类型)

多维数组

- 定义: `T a[][]`
- 存储组织 : 不管是一维还是多维，在内存中都表现为一片连续的空间，如 `int a[2][2]` 在内存中排列顺序为: `a[0][0],a[0][1],a[1][0],a[1][1]`
- 参数传递 : 缺省第一维 `void f(T a[][m], int n)`
- 升降维处理 : 多维当作一维使用/一维当作多维使用，与指针有关。

Struct

struct

- 赋值： 同类型才可以赋值
- 内存分布：成员按声明顺序从高地址向低地址排列， 考虑对齐
- 参数传递 : 传递指针或者引用 (方便施加副作用/节省空间) `void f(T* s), void g(T& s)`

Union

union

- 存储分布：所有成员共享存储空间，占的大小和对齐以最大的成员为准
- 使用场景 : 对同一个位串做不同解释、实现某种多态行为

Pointer

管理地址信息

- 管理数据 : 指向各种类型变量的指针
- 调用代码 : 函数指针

指针定义

- 格式 `<base type> * <pointer variable>` : `T * p;`
- 使用 typedef 来定义一个指针类型 : `typedef int* Pointer; Pointer p, q;`, p,q 都是指针类型变量
- 赋值 : `int* p = 0x8048900; int * q = (int*)0x8048900`

操作符

- 取地址 `&` : `&x` 取出变量 x 的地址
- 间接引用 `*` : `*x` 简介引用指针变量 x 指向的内存空间，`*x = 1` 将 x 指向的变量赋值为 1 (假设 x 是指向整数类型的指针)

初始化

指针变量要初始化，如果没有合适的初始值要置为 `nullptr`, 防止访问到不该访问的内存。

指针运算

- 赋值 : 同类型赋值
- 加减运算(与整型变量) : 基类型不变，改变的值为 `sizeof(base type) * 整型数值`， `int* p; p ++;` p 的值改变了 4
- 同类型指针相加减 : 结果是整型，数值是偏移量 (指针值差/`sizeof(base type)`)
- 比较运算 : 只有 `!= , ==`

输出

```
int x = 1;
int * p = &x;
cout << p << endl; // p 的值 (x 的地址)
cout << *p << endl; //p 指向的整数的值
//特例 : 字符串
char* s = "abcd";
cout << s << endl; // 指针指向的字符串
cout << *s << endl;// 指向的字符，即 a
cout << (int*)s << endl;//地址值
```

void*

只管理地址信息，不知道这块内存存的是什么

- 指针类型的公共接口 :

  ```
  void *p;
  int x;
  double y;
  p = &x;
  p = &y;
  ```

- 要做任何操作必须强制类型转换

```
*p = 1; //error

*((double*) p) = 1;//OK
*((int*) p) = 1;//OK
//example : memset(void*, unsigned)
void memset(void* p, unsigned n){
    char* q = (char*)p;
    for(unsigned i = 0; i < n; i ++){
        *q = 0;
        q ++;
    }
}
```

常量指针与指针常量

- 常量指针: 指向常量的指针

`const <type> * <pointer_variable>`

```
//sample
const int c = 0;
int y = 0;
const int * cp = &c;//ok
int * p = &y;//ok
cp = &y;//ok
*cp = 1 //error
*p = 1;//ok
p = &c;//error without const_cast<int*>
```

使用常量指针可以消除函数副作用，当不准备产生副作用的时候尽可能使用。

```
void f(const int* p);// p read only
void g(int * p);//p read and write
```

a sample about const cast

```
int x=10;
int *p = &x;
cout << " x  " << &x  << x << endl;
cout << " p  " << &p << p << endl;
cout << "*p " << p <<  *p << endl;


const int c=128;
int * q = const_cast<int *>(&c);
*q = 111;
cout << " c " << &c << c << endl; //c      0012FF74    128
cout << " q " << &q << q << endl;// q      0012FF70    0012FF74
cout << "*q " << q << *q << endl;//*q      0012FF74    111
```

为什么这里 c 的值打印出来仍然是 128？

因为用了 const int 会把代码里出现 c 的地方直接用常量 128 代替，即使在实际上那块空间的值已经变化。

以下我写了一小段代码来验证：

```
#include <iostream>

using namespace std;

int main(){
        const int c = 128;
        int a = 10;
        int b = a + c;
        return 0;
}
```

这时反汇编中main函数的部分

```
0000000000001169 <main>:
    1169:       f3 0f 1e fa             endbr64
    116d:       55                      push   %rbp
    116e:       48 89 e5                mov    %rsp,%rbp
    1171:       c7 45 f4 80 00 00 00    movl   $0x80,-0xc(%rbp)
    1178:       c7 45 f8 0a 00 00 00    movl   $0xa,-0x8(%rbp)
    117f:       8b 45 f8                mov    -0x8(%rbp),%eax
    1182:       83 e8 80                sub    $0xffffff80,%eax
    1185:       89 45 fc                mov    %eax,-0x4(%rbp)
    1188:       b8 00 00 00 00          mov    $0x0,%eax
    118d:       5d                      pop    %rbp
    118e:       c3                      retq
```

看 1182 这一行, 在上一行中把变量 a 中的值存到了 eax 里面, 然后在这行直接与常量进行计算, 减去 0xffffff80 就是加上 0x80, 也就是常量 c 的值，这里不是先把 c 的值拷贝过来，而是直接用常量参与运算，说明其实这里的 c 被替换成了常量 128

- 指针常量： 这个指针本身是个常量

必须在定义的时候初始化，之后不能指向别的对象

`<type> * const <variable> = &x;`

```
int x = 0;
int y = 1;
int * const p = &x;//ok
p = &y;//error
*p = 1;//ok
```

指针作为形参

- 提高传输效率： 比传入整个对象一般消耗的空间和时间资源少
- 函数副作用： 需要产生副作用的时候传入非常量指针
- 常量指针： 防止副作用

函数指针

语法 : `<return type> (*<variable_name>)(parameter list)`

sample :

```
//...
double f(int x);
int main(){
    double (*pf)(int) = f;
    double y = f(1);
    return 0;
}
```

指针与数组

- 数组元素操作: 下标表达式， random access `O(1)`
- 数组元素的指针表示法

sample

```
int a[10];
a[1] = 1;// 和 *(a + 1) = 1 等价，数组参与表达式退化成指针
int b[10][10];
//b[i][j] <-> *(*(b + i) + j)
```

不管几维的数组，在内存中都是连续分布的，据此可以进行数组的升维降维操作。

- 指针数组 : 元素是指针的数组

指针与结构

- 函数传参传递结构体的时候经常传递指向结构体的指针，减少大块传输，提高效率
- 如果有可能尽量使用 const 指针

Dynamic Variable

动态变量

- 动态在哪?
    - 大小
    - 生命周期
- 非编译时刻确定
- 生存在堆 (heap) 上

申请

语法:

`new <typename>`

`new <typename>[integer expression]`

和 malloc 的区别，malloc 不会调用构造函数

归还

语法:

`delete <pointer variable>`

`delete[]  <pointer variable>`

和 free 的区别: free 不会调用析构函数

应用

- 数据结构 : 链表，树，图...

Reference

引用

- 为已有的一块内存空间取一个别名

sample:

```
int x = 0;
int &r = x;
r = 1;
cout << x << endl;//1
```

- 引用变量必须指向同类型变量
- 引用变量必须初始化

应用

- 函数参数传递
- 动态变量命名

```
void f(int& x){
    x = 2;
}
int main(){
    int y = 1;
    f(y);
    cout << y << endl;//2
    return 0;
}
```

使用注意事项

- 传参数的时候有可能的话尽量用 const 限定引用(不需要副作用的话)
- 引用一旦定义无法改变
- 如何释放引用堆上的对象

```
int *p = new int(100);
int &x = *p;
delete &x;
```