
第二课 C++对C的扩展

内容概述

1. 函数重载:定义
2. 函数重载:二义性
3. 函数重载:原理
4. 运算符重载初步
5. 默认实参
6. 函数重载:找错
7. 练习:日期运算
8. 内联函数
9. 内存四区
10. new/delete基本用法
11. 练习:动态创建二维数组
12. 表达式
13. 左值右值
14. 显式转换
15. 函数参数传递
16. 函数返回类型

C++对C的扩展(函数重载:定义)

C语言中, 函数名是不能相同的。

C++中, 引入了函数**重载(overload)**的概念。

函数同名, 参数列表不同形成重载。

重载规则:

1. 函数名相同
2. 参数**个数不同/参数类型不同/参数顺序不同**
3. **返回值类型**不同不构成重载

函数匹配(重载确定):

1. 参数完全匹配
2. 通过隐式转换后再匹配

**每个实参都不比其他的匹配差
至少有一个实参的匹配优于其他**

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

int max_val(int a, int b) { return a > b ? a : b; }
double max_val(double a, double b) { return a > b ? a : b; }
float max_val(float a, float b) { return a > b ? a : b; }

void f1(int a) { }
void f1(int a, int b) { }

void f2(int a, char b) { }
void f2(char b, int a) { }

int main() {
    max_val(10, 20);    //完全匹配第1个
    max_val(1.2, 1.3); //完全匹配第2个
    max_val(1.2f, 1.3f); //完全匹配第3个
    max_val('a', 'b'); // char提升为int后匹配第1个
    return 0;
}
```

C++对C的扩展(函数重载:二义性、const)

函数调用时,根据函数名以及参数的匹配来确定调用哪个函数。

实参在匹配时,没有找到类型完全匹配的,则会尝试隐式转换实参类型来匹配

若隐式转换有多个函数可能与之匹配,则发生错误,称为“**二义性调用**”

```
#include <iostream>
using namespace std;
void f1(int a){}
void f1(float a){}
void f2(long a) {}
void f2(double a){}
int main() {
    //f1(1.2);
    //二义性: 1.2是double类型,向int和float都可转换
    //f2(1);
    //二义性: 1是整数,向long和double都可转换
    return 0;
}
```

```
#include <iostream>
using namespace std;
void f1(const int a){}
//void f1(int a){} 错误,顶层const
void f2(int * const p) {}
//void f2(int *p){} 错误,顶层const
void f3(const int *p){}
void f3(int *p){} //OK,底层const
void f4(const int &r){}
void f4(int &r){} //OK,底层const
```

```
int main() {
    int i = 10;
    int *p1 = &i;
    f3(p1); //调用 非const
    const int *p2 = &i;
    f3(p2); //调用 const
    int &r1 = i;
    f4(r1); //调用 非const
    const int &r2 = i;
    f4(r2); //调用 const
    return 0;
}
```

底层const可形成重载

顶层const不形成重载。

底层const和非const都可匹配底层const

C++对C的扩展(函数重载:原理)

编译器在编译C++文件中当前使用的作用域里的同名函数时,根据函数形参的类型和顺序会对函数进行**重命名**(不同的编译器在编译时对函数的重命名标准不一样)

为了兼容C语言,函数声明加上 **extern "C"**

```
文件a.cpp
#include <iostream>
using namespace std;
extern "C" void f1(int a); //单行
extern "C" {              //多行
    void f2(int a);
    void f3(int a);
}
int main() {
    f1(10);
    f2(20);
    f3(20);
    return 0;
}
```

```
文件b.cpp
extern "C" void f1(int a);
extern "C" void f2(int a);
extern "C" void f3(int a);

void f1(int a) {}
void f2(int a) {}
void f3(int a) {}
```

在stdio.h、stdlib.h等C头文件中,都能找到 **extern "C"**

C++对C的扩展(函数重载:运算符重载初步)

运算符重载:

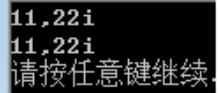
C++中预定义的运算符的操作对象只能是基本数据类型。

实际上,对于很多用户自定义类型,也需要有类似的运算操作。

运算符重载的实质是函数重载。

C++中的运算符除了少数几个之外,全部可以重载。

```
#include <iostream>
using namespace std;
struct Comp {
    int real;
    int image;
};
Comp operator+(const Comp &a, const Comp &b) {
    Comp res = { a.real + b.real, a.image + b.image };
    return res;
}
int main() {
    Comp a = { 1, 2 }, b = { 10, 20 };
    Comp c = a + b; //相当与 operator+(a,b)
    Comp d = operator+(a, b); //相当于 a + b
    cout << c.real << "," << c.image << "i" << endl;
    cout << d.real << "," << d.image << "i" << endl;
    return 0;
}
```



C++对C的扩展(函数重载:默认实参)

在函数的多次调用中,某些实参都被赋予同一个值,此时可以设置**默认实参**。

1. 默认的顺序,必须是**从右向左**,不能间隔。
2. 函数声明和定义分开时,默认实参写在**第一次声明**处,定义处不能再写。
3. 有函数重载的时候,要注意**二义性**。

```
#include <iostream>
using namespace std;
void f1(int a, int b = 0); //第一次声明
//void f1(int a, int b = 0); //错误
void f1(int a, int b); //OK
void f1(int a = 0, int b); //OK

//void f1(int a, int b = 0) {} //错误
void f1(int a, int b) {} //OK
```

```
#include <iostream>
using namespace std;
void f1(int a = 0) {}
void f2(double fd,int i=3,char* ps="abc"){ }
int main() {
    f1(); //相当于 f1(0)
    f1(10);
    f2(3.14); //相当于 f2(3.14,3,"abc")
    f2(3.14, 10); //相当于f2(3.14,10,"abc")
    f2(3.14, 10, "bcd");
    return 0;
}
```

```
#include <iostream>
using namespace std;
void f1(int a,int b=0){ }
void f1(int a){ }

int main() {
    //f1(1); //二义性错误
    return 0;
}
```

C++对C的扩展(函数重载:找错)

例子1:下面2个函数构成重载吗?

```
int main() { return 0; }
int main(int argc, char* argv[]) { return 0; }
```

例子2:下面2个函数构成重载吗?

```
typedef int int_32;
using int32 = int;
void f(int32 a, int32 *const p);
int_32 f(int_32 a, int *p);
```

例子3:下面输出是多少?

```
void f(int a, double b) { cout << 1; }
void f(double a, double b) { cout << 2; }
int main() {
    f(10, 10); //输出是多少?
    return 0;
}
```

例子4:下面输出是多少?

```
void f(int a) { cout << 1; }
void f(short a) { cout << 2; }
int main() {
    f('a'); //输出是多少?
    return 0;
}
```

例子5:下面2个函数构成重载吗?

```
void f(int a, char* p = NULL) { cout << 1; }
void f(int a, int b = 0) { cout << 2; }
int main() {
    f('a'); //正确吗?
    f('a', 0); //输出多少?
    f(10.2, NULL); //输出多少?
    f(1, nullptr); //输出多少?
    return 0;
}
```

例子6:下面2个函数构成重载吗?

```
void f(const int* p) { cout << 1; }
void f(int* p) { cout << 2; }
int main() {
    int i = 1;
    const int j = 2;
    int *pi = &i;
    const int *pj = &j;
    f(pi); //输出多少
    f(pj); //输出多少
    //f(nullptr); //正确吗?
    return 0;
}
```


C++对C的扩展(函数重载:找错答案)

例子1:下面2个函数构成重载吗?

```
int main() { return 0; }  
int main(int argc, char* argv[]) { return 0; }
```

错, main函数不能重载。

例子2:下面2个函数构成重载吗?

```
typedef int int_32;  
using int32 = int;  
void f(int32 a, int32 *const p);  
int_32 f(int_32 a, int *p);
```

错, 注意typedef

顶层const不构成重载

例子3:下面输出是多少?

```
void f(int a, double b) { cout << 1; }  
void f(double a, double b) { cout << 2; }  
int main() {  
    f(10, 10); //输出是多少? 1, 至少有一个参数的匹配比别的好  
    return 0;  
} (寻找最佳匹配)
```

例子4:下面输出是多少?

```
void f(int a) { cout << 1; }  
void f(short a) { cout << 2; }  
int main() {  
    f('a'); //输出是多少? 1, char 到 int 属于类型提升  
    return 0;  
} 类型提升 优先于 类型转换
```

例子5:下面2个函数构成重载吗? **构成重载**

```
void f(int a, char* p = NULL) { cout << 1; }  
void f(int a, int b = 0) { cout << 2; }  
int main() {  
    f('a'); //正确吗? 错  
    f('a', 0); //输出多少? 2  
    f(10.2, NULL); //输出多少? 2  
    f(1, nullptr); //输出多少? 1  
    return 0;  
}
```

例子6:下面2个函数构成重载吗? **构成重载, 底层const**

```
void f(const int* p) { cout << 1; }  
void f(int* p) { cout << 2; }  
int main() {  
    int i = 1;  
    const int j = 2;  
    int *pi = &i;  
    const int *pj = &j;  
    f(pi); //输出多少 2  
    f(pj); //输出多少 1  
    //f(nullptr); //正确吗? 错误  
    return 0;  
}
```

C++对C的扩展(函数重载:找错2)

```
#include <iostream>
using namespace std;
namespace A {
    void f1(int a, int b) {}
}
namespace B {
    void f1(int a) {}
}
using namespace A;
using namespace B;
int main() {
    //A中的f1和B中的f1构成重载吗?
    f1(10);
    f1(10, 20);
    return 0;
}
```

答案：构成重载

```
#include <iostream>
using namespace std;
namespace A {
    void f1(int a, int b) {}
}
namespace B {
    void f1(int a) {}
}
using A::f1;
using B::f1;
int main() {
    //A中的f1和B中的f1构成重载吗?
    f1(10);
    f1(10, 20);
    return 0;
}
```

答案：构成重载

```
#include <iostream>
using namespace std;
void f1(int a, int b) {}
void f1(int a) {}
int main() {
    void f1(int a); //在该作用域中声明函数
    // f1构成重载吗?
    f1(10);
    f1(10, 20);
    return 0;
}
```

答案：只有作用域内的声明起作用。

f1(10) -- ok

f1(10,20) --错误

C++对C的扩展(函数重载:日期运算练习)

例: 日期的简单运算

日期结构体:

```
struct Date{  
    int y; int m; int d;};
```

问题1: 闰年判断

传入年份, 判断是否闰年,
传入Date结构体, 判断是否闰年。

问题2: 加法运算

两个或多个Date相加, 返回Date;
Date加天数(int), 返回Date;

问题3: 打印

输入Date, 打印结果
输入天数(int), 打印结果

```
#include <iostream>  
using namespace std;  
struct Date{  
    int y; //年  
    int m; //月  
    int d; //日  
};  
int tab[2][12] = {  
    { 31,28,31,30,31,30,31,31,30,31,30,31 },  
    { 31,29,31,30,31,31,30,31,30,31,30,31 } };  
int d2n(const Date &date) {  
    int days = 0;  
    for (int y = 1; y < date.y; y++)  
        days = is_leap(y) ?  
            days + 366 : days + 365;  
    for (int m = 1; m < date.m; m++)  
        days = is_leap(date.y) ? days +  
            tab[1][m - 1] : days + tab[0][m - 1];  
    days += date.d;  
    return days;  
}
```

函数: d2n, 将Date类型转化为天数

函数: n2d, 将天数转化为Date类型

```
Date n2d(int days) {  
    int y = 1, m = 1;  
    while (1) {  
        if (is_leap(y)) {  
            if (days > 366) days -= 366;  
            else break;  
        }  
        else {  
            if (days > 365) days -= 365;  
            else break;  
        }  
        y++;  
    }  
    while (1) {  
        if (is_leap(y)) {  
            if (days > tab[1][m - 1])  
                days -= tab[1][m - 1];  
            else break;  
        }  
        else {  
            if (days > tab[0][m - 1])  
                days -= tab[0][m - 1];  
            else break;  
        }  
        m++;  
    }  
    return Date{ y,m,days };  
}
```

C++对C的扩展(函数重载:日期运算填空)

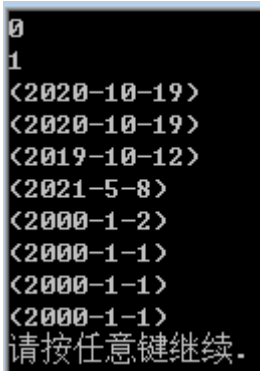
```
bool is_leap(int year) {
    if ((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0))
        return true;
    return false;
}
bool is_leap(const Date &date) {
    return is_leap(date.y);
}
Date operator+(const Date &d1, int days) {
    return n2d(d2n(d1) + days);
}
Date operator+(int days, const Date &d1) {
    return operator+(d1, days);
}
Date operator+(, ) {
    return n2d(d2n(d1) + d2n(d2));
}
void print(const Date &date) {
    cout << "(" << date.y << "-" << date.m
        << "-" << date.d << ")" << endl;
}
void print(int days) {
    print(n2d(days));
}
```

```
#include <iostream>
using namespace std;
struct Date{
    int y; //年
    int m; //月
    int d; //日
};
int d2n(const Date &date);
Date n2d(int days);
函数: d2n, 将Date类型转化为天数
函数: n2d, 将天数转化为Date类型
int tab[2][12] = {
    { 31,28,31,30,31,30,31,31,30,31,30,31 },
    { 31,29,31,30,31,31,30,31,30,31,30,31 } };
int main() {
    Date d1 = { 2019,10,10 };
    Date d2 = { 2,1,10 };
    cout << is_leap(d1) << endl; //0
    cout << is_leap(2000) << endl; //1
    Date d3 = d1 + d2;
    Date d4 = operator+(d2, d1);
    print(d3); //2020-10-19
    print(d4); //2020-10-19
    d3 = d1 + 2;
    print(d3); //2019-10-12
    d3 = d1 + 200 + d2 + Date{ 1,1,1 };
    print(d3); //2021-5-8
    print(Date{ 2000,1,2 }); //2000-1-1
    print(n2d(730120)); //2000-1-1
    print(730120); //2000-1-1
    print(Date{ 1999, 1, 1 } +365); //2000-1-1
    return 0;
}
```

C++对C的扩展(函数重载:日期运算答案)

```
bool is_leap(int year) {
    if ((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0))
        return true;
    return false;
}
bool is_leap(const Date &date) {
    return is_leap(date.y);
}
Date operator+(const Date &d1, int days) {
    return n2d(d2n(d1) + days);
}
Date operator+(int days, const Date &d1) {
    return operator+(d1, days);
}
Date operator+(const Date &d1, const Date &d2) {
    return n2d(d2n(d1) + d2n(d2));
}
void print(const Date &date) {
    cout << "(" << date.y << "-" << date.m
        << "-" << date.d << ")" << endl;
}
void print(int days) {
    print(n2d(days));
}
```

```
#include <iostream>
using namespace std;
struct Date{
    int y; //年
    int m; //月
    int d; //日
};
int d2n(const Date &date);
Date n2d(int days);
//函数: d2n, 将Date类型转化为天数
//函数: n2d, 将天数转化为Date类型
int tab[2][12] = {
    { 31,28,31,30,31,30,31,31,30,31,30,31 },
    { 31,29,31,30,31,31,30,31,30,31,30,31 } };
int main() {
    Date d1 = { 2019,10,10 };
    Date d2 = { 2,1,10 };
    cout << is_leap(d1) << endl; //0
    cout << is_leap(2000) << endl; //1
    Date d3 = d1 + d2;
    Date d4 = operator+(d2, d1);
    print(d3); //2020-10-19
    print(d4); //2020-10-19
    d3 = d1 + 2;
    print(d3); //2019-10-12
    d3 = d1 + 200 + d2 + Date{ 1,1,1 };
    print(d3); //2021-5-8
    print(Date{ 2000,1,2 }); //2000-1-1
    print(n2d(730120)); //2000-1-1
    print(730120); //2000-1-1
    print(Date{ 1999, 1, 1 } +365); //2000-1-1
    return 0;
}
```



```
0
1
<2020-10-19>
<2020-10-19>
<2019-10-12>
<2021-5-8>
<2000-1-2>
<2000-1-1>
<2000-1-1>
<2000-1-1>
请按任意键继续.
```

内联函数

语法：在函数声明前加上 **inline**

注意：内联说明只是向编译器发出的一个**请求**，编译器可以忽略这个请求；

使用场景：一般来说，内联机制用于优化**规模较小、流程直接、频繁调用**的函数。比如行数太多、有递归等，一般编译器都不会真正内联。

特点：内联函数可以在程序中多次定义，但是多次定义必须一致；所以，一般内联函数都写在头文件中。（因为在编译的时候要展开）。

函数调用过程：调用前先要保存寄存器，返回时恢复；还可能需拷贝实参；程序转向一个新的位置继续执行。

```
#include <iostream>
using namespace std;
//C语言做法
#define max_val_c(a,b) (((a) > (b)) ? (a) : (b))
//C++做法,inline内联函数
inline int max_val(int a, int b) {
    return a > b ? a : b;
}
int main() {
    int i = 10, j = 20;
    cout << max_val_c(i, j) << endl; //预编译时替换
    cout << max_val(i, j) << endl; //编译时展开
    return 0;
}
```


内联函数：宏和内联的区别

使用**宏**和**内联函数**都可以节省在函数调用方面所带来的时间和空间开销。二者都采用了空间换时间的方式，在其调用处进行展开：

- (1) 在**预编译时期**，宏定义在调用处执行字符串的原样替换。在**编译时期**，内联函数在调用处展开，同时进行参数类型检查。
- (2) 内联函数首先是函数，可以像调用普通函数一样调用内联函数。而宏定义往往需要添加很多括号防止歧义，编写更加复杂。
- (3) 内联函数可以作为某个类的成员函数，这样可以使使用类的保护成员和私有成员。而当一个表达式涉及到类保护成员或私有成员时，宏就不能实现了(无法将this指针放在合适位置)。

可以用内联函数完全替代宏。

在编写内联函数时，函数体应该短小而简洁，不应该包含循环等较复杂结构，否则编译器不会将其当作内联函数看待，而是把它决议成为一个静态函数。

有些编译器甚至会优化内联函数，通常为避免一些不必要拷贝和构造，提高工作效率。

频繁的调用内联函数和宏定义容易造成**代码膨胀**，消耗更大的内存而造成过多的换页操作。

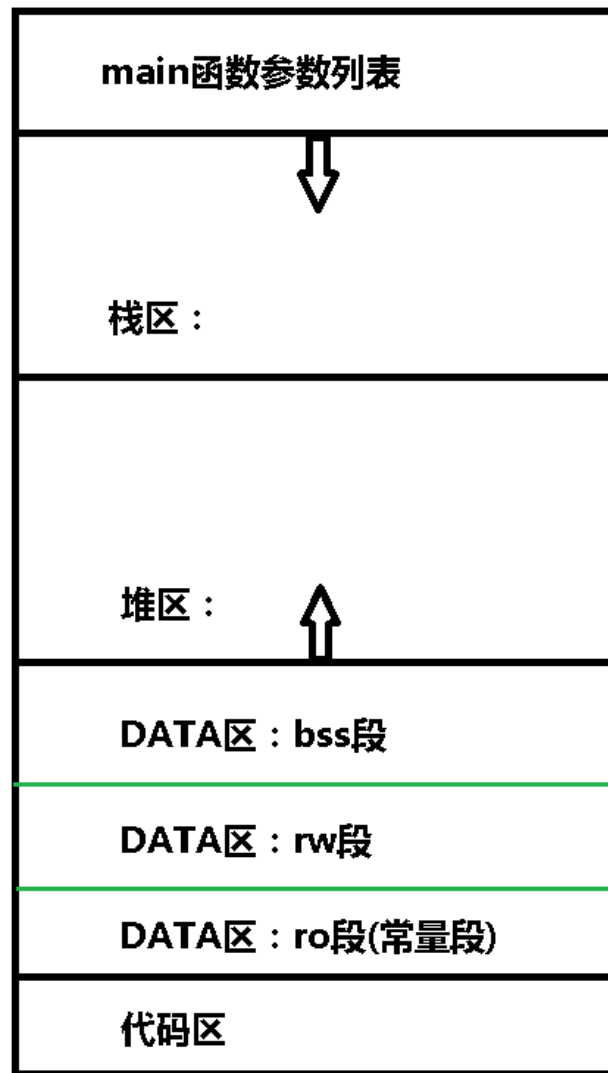
内存四区

```
#include <stdio.h>
void fun() {
    static int k = 10; //初始化的静态局部变量(data区的rw段)
    static int kk;     //未初始化静态局部变量(data区的bss段) 默认初始化为0
    printf("data: static= %p(rw),%p(bss)\n", &k, &kk);
}
int g_int1, g_int2;   //未初始化全局变量(data区bss段)
int g_int3 = 10;     //初始化的全局变量(data区rw段)
char *g_pstr1 = "abc"; //g_pstr1 初始化的全局变量(data区rw段)
                    //"abc" 字符串常量(data区常量区)

int main() {
    printf("data: global= %p(bss),%p(bss)\n", &g_int1, &g_int2);
    printf("data: global= %p(rw)\n", &g_int3);
    printf("data: global= %p(rw)\n", &g_pstr1);
    printf("data:p_str1指向的地址: %p(常量段)\n", g_pstr1);
    printf("data: abc:          %p(常量段)\n", &"abc");
    printf("code: fun= %p(code)\n", fun);
    fun();
    int i = 10, j = 20; //栈区
    printf("栈区: %p(栈区),%p(栈区)\n", &i, &j);
    int *pi = new int(10); //pi在栈区, pi指向的内容(*pi)在堆区
    printf("堆区: pi %p(堆区)\n", pi);
    delete pi;
    return 0;
}
```

```
data: global= 00401360(bss),00401364(bss)
data: global= 00401004(rw)
data: global= 00401008(rw)
data:p_str1指向的地址: 003FEB58<常量段>
data: abc:          003FEB58<常量段>
code: fun= 003F14B0(code)
data: static= 00401000(rw),00401368(bss)
栈区: 0018FD18<栈区>,0018FD0C<栈区>
堆区: pi 00514EE0<堆区>
请按任意键继续. . .
```

高地址



程序运行时的内存空间分布
(内存四区图)

局部变量等

malloc或new分配的空间

没有初始化或初始化为0的全局变量
或static变量。

初始化不为0的全局变量
或static变量。

常量段 (如字符串常量)

存放函数体的二进制代码

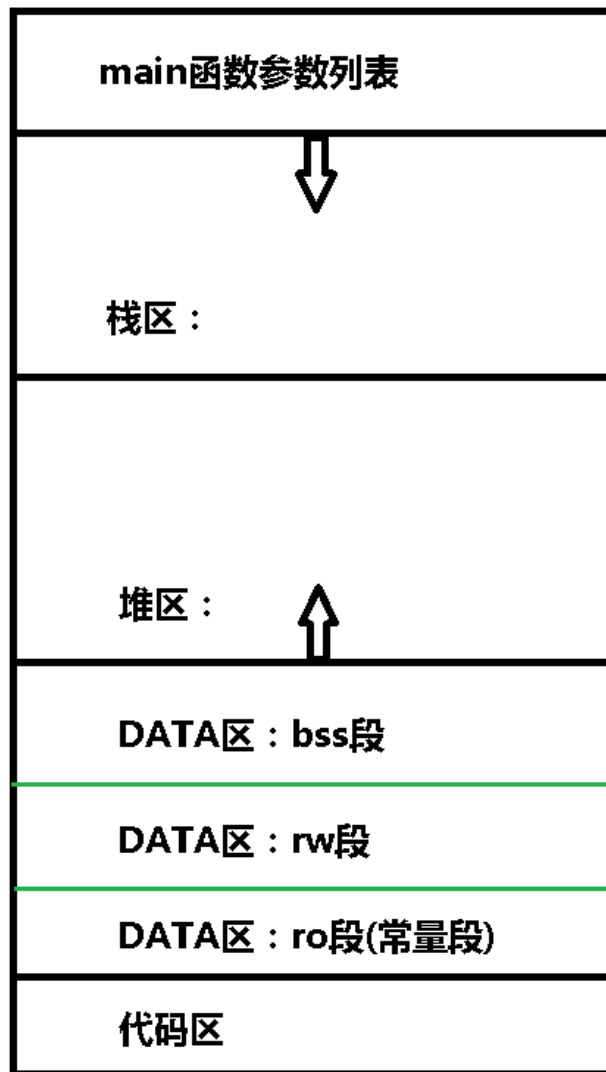
低地址

内存四区:练习

```
#include <stdio.h>
#include <stdlib.h>
int a = 0, b, c = 10;
extern int x;
static char ch1, ch2 = 'o';
struct A {
    int data;
    int *p_data;
};
int main() {
    char d = 'x';
    static int e;
    char *p = (char *)malloc(20);
    A a1;
    a1.p_data = &a1.data;
    A* pa2 = (A*)malloc(sizeof(A));
    pa2->p_data = (int*)malloc(10 * sizeof(int));
    free(pa2->p_data);
    free(pa2);
    return 0;
}
```

指出程序中
各变量的位置

高地址



程序运行时的内存空间分布
(内存四区图)

局部变量等

malloc或new分配的空间

没有初始化或初始化为0的全局变量
或static变量。

初始化不为0的全局变量
或static变量。

常量段 (如字符串常量)

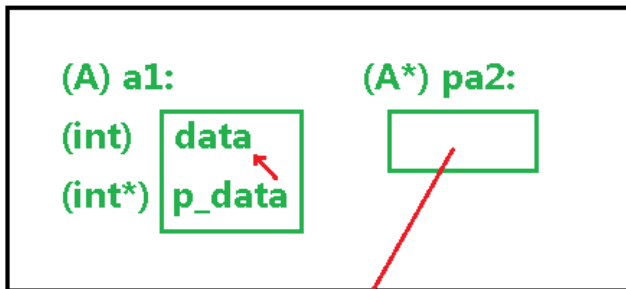
存放函数体的二进制代码

低地址

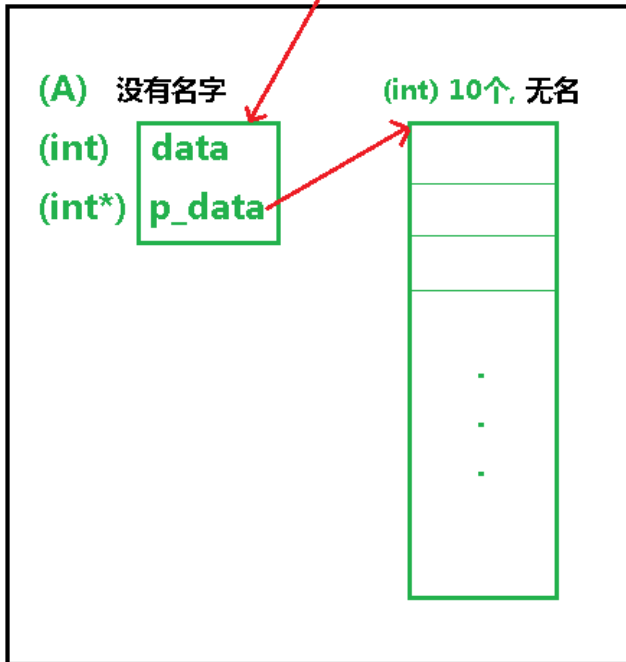
内存四区:练习答案

```
#include <stdio.h>
#include <stdlib.h>
int a = 0, b, c = 10; // a,b 是data区bss段, c是data区rw段
extern int x; //变量的声明, 没有分配内存
static char ch1, ch2 = 'o'; //ch1是data区bss段, ch2是data区rw段
struct A { //结构体类型定义, 不分配内存
    int data;
    int *p_data;
};
int main() {
    char d = 'x'; //栈区
    static int e; //data区bss段
    char *p = (char *)malloc(20); //p在栈区, p指向的内存在堆区
    A a1; //栈区
    a1.p_data = &a1.data; //指针本身 和 它指向的内存 都在 栈区
    A* pa2 = (A*)malloc(sizeof(A)); //pa2在栈区, *pa2在堆区
    pa2->p_data = (int*)malloc(10 * sizeof(int)); //都在堆区
    free(pa2->p_data);
    free(pa2);
    return 0;
}
```

栈区:



堆区:



new/delete基本应用

堆内存的申请和释放:

C: malloc 和 free 两个函数

C++: new/delete 两个关键字, 并扩充了功能
(new/delete在创建对象时会调用构造器和析构器)

注意事项

- 1, 配对使用, new--delete,new[]—delete[]
- 2, 有申请就要有释放, 避免内存泄漏
- 3, 防止多重释放
- 4, 避免交叉使用

比如 malloc 申请的空间用 delete,
new 出的空间用 free

new/delete, 重点用在类对象的申请与释放。

申请时会调用构造器完成初始化

释放时会调用析构器完成内存的清理

```
#include <iostream>
#include <stdlib.h>
using namespace std;
int main() {
    int *p1 = (int*)malloc(sizeof(int));
    free(p1); //C的malloc和free
    int *p2 = new int; //C++:不要类型转换
    delete p2; //C++:不要sizeof(int)
    //new 和 new[]
    //开辟单变量空间
    int *p3 = new int; //没初始化,随机数
    p3 = new int(10); //初始化为 10
    int *p4 = new int{100}; //初始化为 100
    //开辟数组空间
    p4 = new int[10]; //开辟10个int空间(没初始化)
    p4 = new int[10]{ 1,2 }; //初始化,后面用0初始化
    int **pp = new int*[10]{NULL}; //10个存放int*的空间
    //delete 和 delete []
    delete p3; // delete 对应 new
    delete[] p4; // delete [] 对应 new []
    delete[] pp;
    return 0;
}
```

new申请内存失败的处理

new申请失败:

默认: 失败**抛异常**

加nothrow: 失败**返回NULL**

set_new_handler
处理函数

```
#include <iostream>
using namespace std;
int * p1, *p2;
void my_new_heandler() {
    cout << "haha" << endl;
    delete[] p1;
    p1 = NULL;
}
int main() {
    set_new_handler(my_new_heandler);
    p1 = new int[336870912];
    cout << "p1 ok" << endl;
    p2 = new int[336870912];
    cout << "p2 ok" << endl;
    getchar(); //等待观察
    delete[] p1;
    delete[] p2;
    return 0;
}
```

C++:

set_new_handler

```
p1 ok
haha
p2 ok
请按任
```

```
#include <stdlib.h>
```

```
#include <assert.h>
```

```
int main() {
    int *p = (int*)malloc(sizeof(int));
    assert(p!=NULL); //assert(p);
    if (p == NULL) exit(-1);
    if (p) free(p);
    return 0;
}
```

C语言:

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

```
int main() {
    int *p1 = new int[336870912];
    int *p2 = new int[336870912];
    //申请失败, 抛出异常, 不处理就中止程序
    int *p3 = new (std::nothrow)int[336870912];
    //申请失败, p3==NULL, 类似C的处理方式
    return 0;
}
```

C++: **默认抛异常**

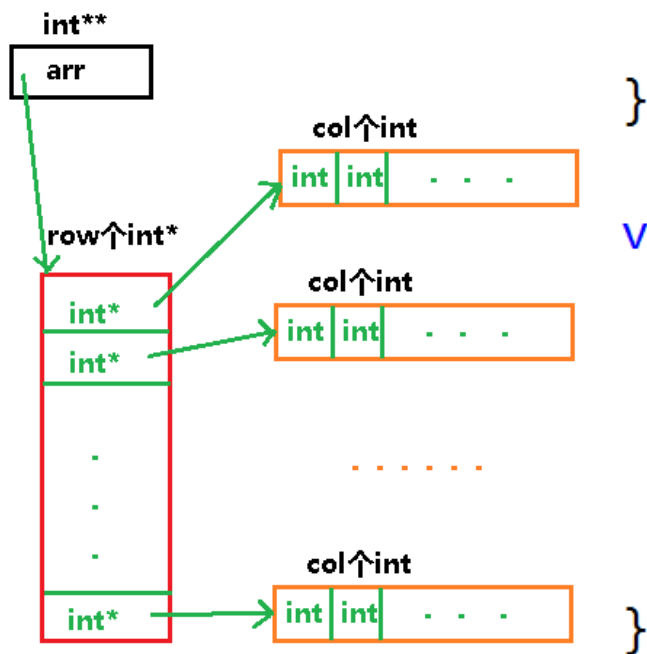
加nothrow返回NULL

new/delete: 动态创建二维数组, 练习填空

根据输入的 row, col 值
创建二维数组:

实现create和destroy函数:

```
Type** create(int row, int col);  
void destroy(Type** p, int row);
```



```
Type** create(int row, int col) {  
    if (row <= 0 || col <= 0)  
        return NULL;  
    Type **p = ;  
    for (int i = 0; i < row; i++)  
        p[i] = ;  
    return p;  
}
```

```
void destroy(Type** p, int row) {  
    if (p == NULL || row <= 0)  
        return;  
    for (int i = 0; i < row; i++)  
        ;  
    ;  
}
```

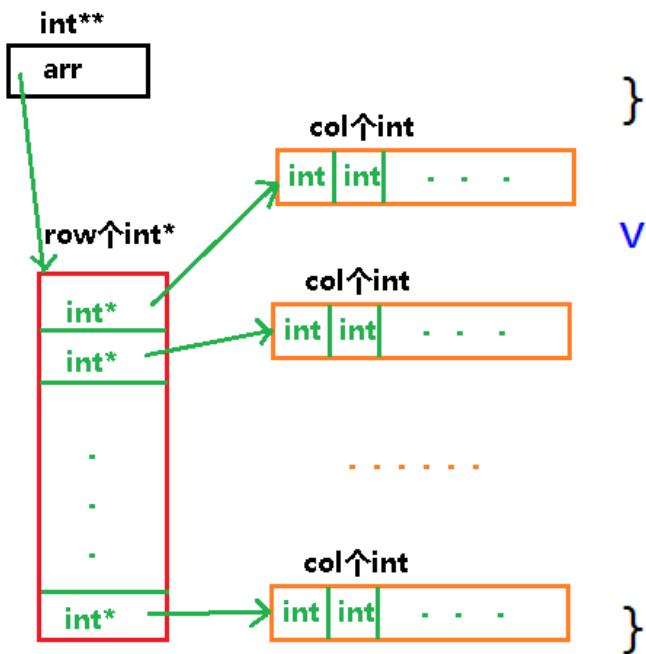
```
#include <iostream>  
using namespace std;  
typedef int Type; //using Type = int;  
Type** create(int row, int col);  
void destroy(Type** p, int row);  
int main() {  
    int row, col;  
    cin >> row >> col;  
    Type ** arr = create(row, col);  
    for (int i = 0; i < row; i++) {  
        for (int j = 0; j < col; j++) {  
            arr[i][j] = i * 10 + j + 1;  
            cout << arr[i][j] << "\t";  
        }  
        cout << endl;  
    }  
    destroy(arr, row);  
    return 0;  
}
```

new/delete: 动态创建二维数组, 练习答案

根据输入的 row, col 值
创建二维数组:

实现create和destroy函数:

```
Type** create(int row, int col);  
void destroy(Type** p, int row);
```



```
Type** create(int row, int col) {  
    if (row <= 0 || col <= 0)  
        return NULL;  
    Type **p = new Type*[row];  
    for (int i = 0; i < row; i++)  
        p[i] = new Type[col];  
    return p;  
}
```

```
void destroy(Type** p, int row) {  
    if (p == NULL || row <= 0)  
        return;  
    for (int i = 0; i < row; i++)  
        delete[] p[i];  
    delete[] p;  
}
```

```
#include <iostream>  
using namespace std;  
typedef int Type; //using Type = int;  
Type** create(int row, int col);  
void destroy(Type** p, int row);  
int main() {  
    int row, col;  
    cin >> row >> col;  
    Type ** arr = create(row, col);  
    for (int i = 0; i < row; i++) {  
        for (int j = 0; j < col; j++) {  
            arr[i][j] = i * 10 + j + 1;  
            cout << arr[i][j] << "\t";  
        }  
        cout << endl;  
    }  
    destroy(arr, row);  
    return 0;  
}
```

```
3 4  
1    2    3    4  
11   12   13   14  
21   22   23   24  
请按任意键继续. . .
```

表达式

求值的顺序:

$f1() + f2() * f3() - f4()$

优先级: $f2()$ 的返回值和 $f3()$ 的返回值先相乘

结合律: $f1()$ 的返回值先后 $f2() * f3()$ 的乘积相加,再减去 $f4()$

但是: $f1(), f2(), f3(), f4()$ 的调用顺序没有明确规定,取决于编译器

所以: $i1 = 0;$

```
cout << (i1++ + ++i1 - 1) << endl;
```

//不确定,不同编译器不同解释,不要这样写

赋值表达式:

```
int a,b;
```

```
a = b = 1; //c c++都可以
```

```
(a = b) = 1; //c不可以,c++可以,表达式可以被赋值
```

$(a = b)$ 先执行,返回 a ,然后 $a = 1$

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

```
int main() {
    //优先级和结合律
    int n = -10 * 2 + 20 / 2 * 3 - 10;
    ((((-10) * 2) + ((20 / 2) * 3)) - 10);
    int arr[] = { 1,4,5,8 };
    int i1 = *arr + 2; //3
    int i2 = *(arr + 2); //5
```

//常用写法:

```
int *p = arr;
*p++; // *(p++), *p 然后 p = p + 1
*++p; // *(++p) 先 p = p + 1, 然后 *p
```

//避免下面这样的写法

```
i1 = 0;
cout << (i1++ + ++i1 - 1) << endl;
```

```
return 0;
```

```
}
```


表达式

逻辑运算符 && ||:

短路求值:

&&: 左侧表达式为真才运行右侧表达式

||: 左侧表达式为假才运行右侧表达式

```
int arr[]={1,2,3,0,4};
int idx=0;
while(idx<sizeof(arr) && arr[idx] != 0){
    cout <<arr[idx++];
}
```

上面代码不会出现下标越界。

```
#include <iostream>
using namespace std;
int main() {
    int a = 1, b = 1;
    (a = 0) && (b = 100); //&&左侧表达式为假, 右侧不运算
    cout << a << b << endl; // a=0 b=1
    int i = 1, j = 1;
    (i = 10) || (j = 100); //||左侧表达式为真, 右侧不运算
    cout << i << j << endl; //i=10 j=1

    //注意次序:
    int c = 1;
    //a<b先比, 返回bool类型, 然后bool类型提升为int(0或1)和c比较
    bool b1 = a < b < c;

    //注意优先级: 先b<c比较得到bool类型, 提升为int再和a==判断
    a == b < c;

    //注意bool类型:
    a = 10;
    while (a) //循环10次
        a--;
    a = 10;
    while (a == true) //循环0次
        a--;
    return 0;
}
```


左值右值

左值右值:

形式区分(语法区分): 能否用取地址&运算符;

语义区分(本质涵义): 表达式代表的是持久对象还是临时对象。

左值和右值的区分标准在于能否获取地址。

当一个对象被用作右值的时候, 用的是对象的值(内容), 当对象被用作左值的时候, 用的是对象的身份(在内存中的位置)

临时量:

临时变量实际上就是一个没有名字的变量而已。

临时变量和它的引用具有相同的生命周期。

内置类型临时量的const属性

函数返回值、类型转换 产生临时量

```
#include <iostream>
using namespace std;
int main() {
    int a; //a 的含义
    a = 10;
    int b;
    b = a + 20; //临时量
    b = a + 20.1;
    int *p = &b;

    //内置类型的临时量有const属性
    //(a + 2) = 10; //为啥不行?
    int &ra = a;
    //int &ra1 = a + 2; //为啥不行?
    const int &cra = a + 2; //ok
    int &&rta = a + 2; //右值引用

    ++++a; //ok
    //++a++; //不行

    return 0;
}
```

显式转换

static_cast:

具有明确定义的类型转换（不能转换底层const）

const_cast:

改变运算对象的底层const

reinterpret_cast:

通常为运算对象的位模式提供较低层次上的重新解释。

dynamic_cast:

支持运行时类型识别，常用于基类指针转向派生类指针。

```
#include <iostream>
using namespace std;
int main() {
    //static_cast<转换后类型>(需要转换的数据)
    double fd1 = 1.2;
    int i1 = fd1; //隐式转换,精度丢失,编译器会警告
    i1 = static_cast<int>(fd1); //明确转换,不警告
    void *vi1 = &i1;
    int* pi1 = (int*)vi1; //c写法 void* --> int*
    pi1 = static_cast<int*>(vi1); //c++写法
    //注意:int* <--> char* 不能static_cast
    char *pc1 = (char*)pi1; //c写法 强转int*-->char*
    //pc1 = static_cast<char*>(pi1); 错误

    //const_cast:
    const int i2 = 10;
    const int *cpi2 = &i2;
    int *pi2 = (int*)cpi2; //c写法
    pi2 = const_cast<int*>(cpi2); //去掉底层const
    int &ri2 = const_cast<int&>(i2);

    //reinterpret_cast
    int a = 0x00434241;
    int* p = &a;
    char* pc = (char*)p; //c写法
    pc = reinterpret_cast<char*>(p);
    int *pa = (int*)a; //c写法,int-->int*
    pa = reinterpret_cast<int*>(a);

    return 0;
}
```

函数参数传递

函数调用时

用传入的**实参初始化形参**

传值参数

非引用类型形参，
实参值拷贝给形参
指针形参也是如此。

传引用参数

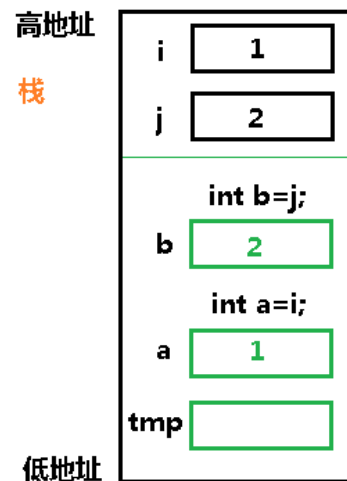
引用形参，通过引用绑定实参

使用引用避免拷贝。

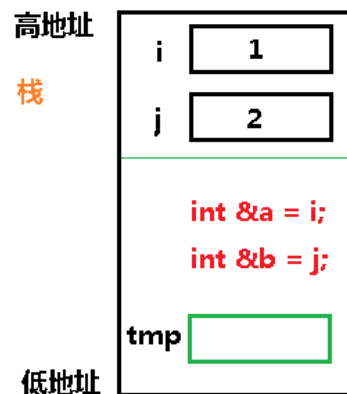
使用引用或指针作为形参，可
传回额外的信息。

```
int main() {  
    int i = 1, j = 2;  
    swap(i, j);  
    swap(&i, &j);  
    swap1(i, j);  
    return 0;  
}  
  
void swap(int a, int b) {  
    int tmp = a;  
    a = b;  
    b = tmp;  
}  
  
void swap(int *pa, int *pb) {  
    int tmp = *pa;  
    *pa = *pb;  
    *pb = tmp;  
}  
  
void swap1(int &a, int &b) {  
    int tmp = a;  
    a = b;  
    b = tmp;  
}
```

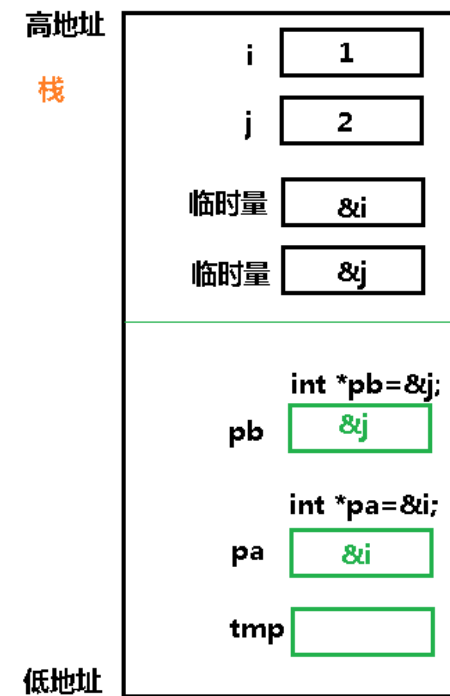
```
swap(int a, int b)  
swap(i, j);
```



```
swap1(int &a, int &b)  
swap1(i, j);
```



```
swap(int *pa, int *pb)  
swap(&i, &j);
```



函数参数传递: const形参和实参

形参是顶层const:
用实参初始化形参时
忽略顶层const
const int i = 常量 或 变量 都一样

形参类型	实参类型	例子
int *	int *	Fun(&i); //OK
int *	const int *	Fun(&ci); //ERROR
const int *	int *	Fun(&i); //OK
const int *	const int *	Fun(&ci); //OK
int &	int &	Fun(i); //OK
int &	const int &	Fun(ci); //ERROR Fun(30); //ERROR
const int &	int &	Fun(i); //OK
const int &	const int &	Fun(ci); //OK Fun(30); //OK

尽量使用常引用:

```
void fun(int &a) {}  
void func(const int &a) {}  
bool find(string &s, char c) {  
    return true; //s能在这里修改  
}  
bool findc(const string &s, char c) {  
    return true; //保证s在这里不能修改  
}  
int main() {  
    //fun(42); 不行  
    func(42); //ok  
    const string s1 = "abc";  
    //find(s1, 'c'); //不行  
    findc(s1, 'c'); //ok  
    return 0;  
}
```

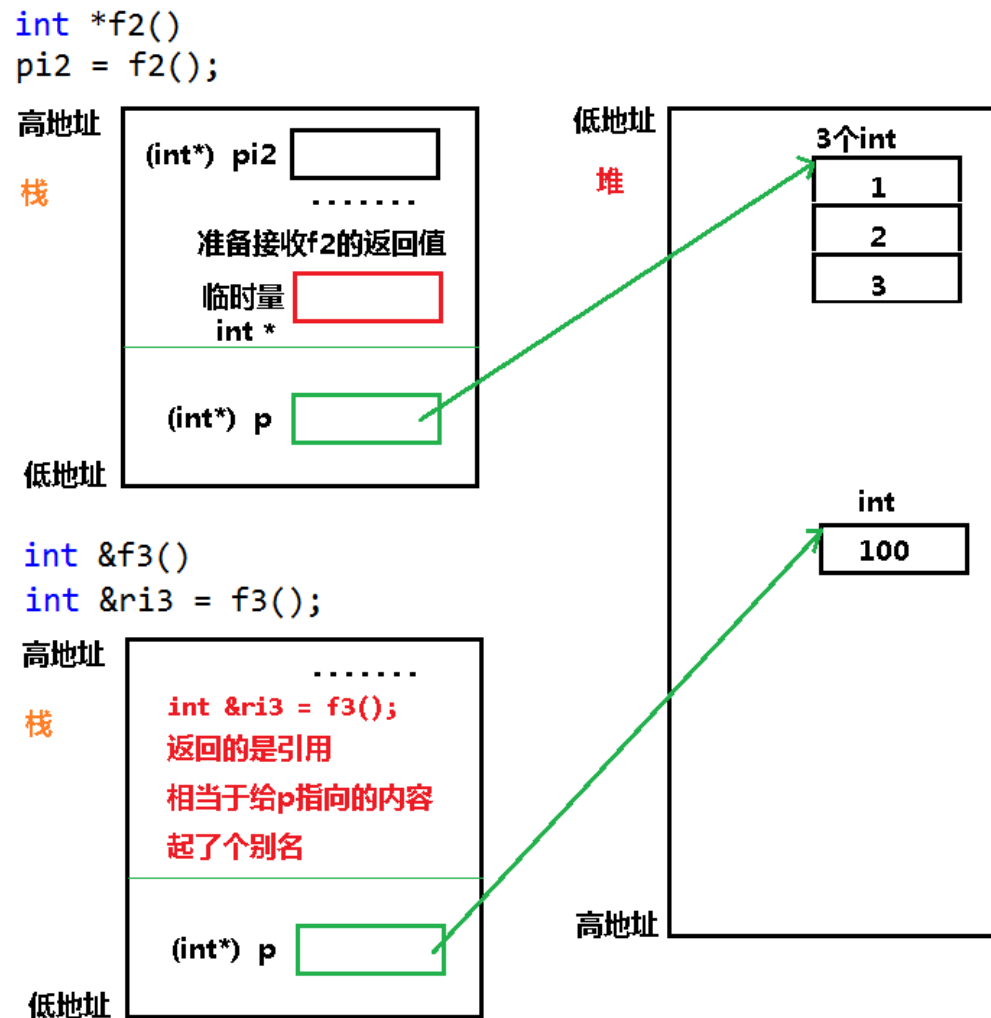
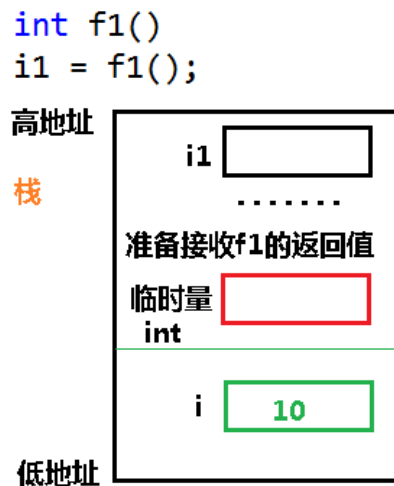
```
void fun(int *i)  
    { cout << 1 << endl; } //1  
void fun(const int *i)  
    { cout << 2 << endl; } //2  
void fun(int &i)  
    { cout << 3 << endl; } //3  
void fun(const int &i)  
    { cout << 4 << endl; } //4  
int main() {  
    int i = 10;  
    int &ri = i;  
    const int &cri = i;  
    //&ri, &cri, ri, cri为实参调用fun  
    fun(&ri); //1  
    fun(&cri); //2  
    fun(ri); //3  
    fun(cri); //4  
    fun(42); //4  
    fun(1.23); //4  
    fun('a'); //4  
    return 0;  
}
```

函数返回类型

函数返回的值：
返回的值用于
初始化调用点
的一个临时量，
该临时量就是
函数的返回值。

不要返回局部
对象的引用或
指针

```
int f1() {
    int i = 10;
    return i;
}
int *f2() {
    int *p = new int[3]{ 1,2,3 };
    return p;
}
int &f3() {
    int *p = new int(100);
    return *p;
}
int main() {
    int i1;
    i1 = f1();
    cout << i1 << endl; //10
    int *pi2;
    pi2 = f2();
    cout << *++pi2 << endl; //2
    int &ri3 = f3();
    cout << ri3 << endl; //100
    delete[--pi2];
    delete &ri3;
    return 0;
}
```



函数返回类型与const

```
#include <iostream>
using namespace std;
int f1() { return 33; } //ok
//和上面是一样的,调用时 int i = f2(); ok
const int f2() { int i = 10; return i; }
int *f3() {
    int *p = new int[3]{ 1,2,3 };
    return p;
}
const int *f4() {
    int *p = new int[3]{ 1,2,3 };
    return p;
}
int &f5() {
    int *p = new int(100);
    return *p;
}
const int &f6() {
    int *p = new int(100);
    cout <<"f6():"<< p << endl;
    return *p;
}
//int &f7() { return 10; } //错误
int &f7() { int i = 101; return i; }//编译ok,但是不能这样用
const int &f8() { return 102; } //编译ok,但是不能这样用
```

```
int main() {
    //int *p = f4(); //错误
    const int *p = f4(); //必须用const接收
    //int &i = f6(); //错误
    const int &i = f6(); //必须用const接收
    const double &pd = f6(); //pd在栈空间了
    cout << &pd << endl; //上面的转换后,地址不同了
    cout << &p << endl;
    cout << "-----" << endl;
    cout << f7() << endl; //输出101 貌似ok
    int &rf7 = f7();
    cout << rf7 << endl; //输出101 貌似也ok
    f1(); //随便调用了一个函数
    cout << rf7 << endl; //输出-858.... 出错!
    //所以不能返回局部变量的引用
    return 0;
}
```

```
f6<>:007B6470
f6<>:007B62E8
003EFBBC
003EFBE4
-----
101
101
-858993460
请按任意键继续
```