
第一课 C 与 C++

内容概述

第一部分：C++综述

1. C++历史背景
2. Why C++
3. 推荐书籍

第二部分：C VS C++

1. C语法回顾
2. 例：C实现动态数组
3. C++的vector

第三部分：C++对C的扩展（上）

1. 命名空间
2. 输入输出
3. 基本类型转换
4. 声明、列表初始化
5. 指针和引用
6. const限定符
7. struct和class
8. string和vector
9. 练习：进制转换

C++历史背景

1982年，美国AT&T公司贝尔实验室的Bjarne Stroustrup(本贾尼)博士在C语言的基础上引入并扩充了面向对象的概念，发明了一种新的程序语言，被命名为C++。

C++发展的几个阶段：

第一阶段（从80年代到1995年）：这一阶段C++语言基本上传统类型上的**面向对象语言**，凭借着**接近C语言的效率**，在开发语言中占据了很大份额；

第二阶段（从1995年到2000年）：这一阶段由于**标准模板库(STL)**等程序库的出现，**泛型程序设计**在C++中占据了越来越多的比重性。当然，同时由于Java等语言的出现和硬件价格的大规模下降，C++受到了一定的冲击；

第三阶段从2000年至今，由于以Loki、MPL等程序库为代表的产生式编程和模板元编程的出现，C++出现了发展历史上又一个新的高峰，这些新技术的出现以及和原有技术的融合，使C++已经成为**当今主流程序设计语言中最复杂的一员**。

C与C++的区别

C语言： C语言诞生得非常早(70年代)， C语言的目标就是比汇编方便易用，同时不要损失汇编的表达能力。所以C语言可以看成是“高级的汇编”语言。C的特点，简单容易编译，灵活贴近底层。所以一直到现在，一些需要直接和硬件打交道的软件都还是用C语言写的，比如（但不限于）Linux Kernel和一些嵌入式领域。

C++语言： C++早期是基于C的。C++的目标是提高编程人员的生产效率。而提高编程人员生产率的方法有如下几种：提高抽象层次，支持模块化编程，模块内紧耦合，模块间松耦合，自动化的代码生成等等，在C++中可以比C更直接更自然地做到这些。面向对象只是C++的一部分，现代的C++的目标是支持多种编程范型，同时并不会离硬件太远。所以C++是非常适合写一些基础架构级软件的，比如编译器，GUI库等等。

C++内容

C++兼容C

++的内容:

C++对C语法的扩展

面向对象

模板和STL

应用领域(效率、建模和高度抽象)

系统层软件、服务器程序、云计算、分布式、游戏...

C++书籍

《C++ Primer 第5版》

《Effective C++》

C语法回顾

基本类型: int char float double

输入输出: scanf printf

基本语句: for, while, switch case, if else ...

数组: int arr[]

字符串: char str[]

指针: int *p

内存分配: malloc和free

函数: function

结构体: struct

文件操作: FILE

C实现动态数组

存储学生信息，要求顺序存储
可逐个添加信息，减少内存浪费

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>
struct Stu { //单个学生结构体
    int id;
    char name[20];
};
struct Stu_arr { //学生数组结构体
    struct Stu *pstu;
    int size;
};
int main() {
    struct Stu_arr ss;
    init(&ss); //初始化
    struct Stu stu1;
    stu1.id = 1; strcpy(stu1.name, "张三");
    push_back(&ss, &stu1);
    print(&ss);
    stu1.id = 5; strcpy(stu1.name, "mike");
    push_back(&ss, &stu1);
    print(&ss);
    destroy(&ss); //清理内存
    return 0;
}
```

```
<1,张三>
<1,张三> <5,mike>
请按任意键继续. . .
```

```
void init(struct Stu_arr *p_ss) {
    p_ss->size = 0;
    p_ss->pstu = NULL;
}
void destroy(struct Stu_arr *p_ss) {
    if (p_ss->pstu)
        free(p_ss->pstu);
}
void push_back(struct Stu_arr *p_ss, struct Stu *p_stu) {
    struct Stu* p_new = (struct Stu*)malloc((p_ss->size + 1)
        * sizeof(struct Stu));
    memcpy(p_new, p_ss->pstu, p_ss->size * sizeof(struct Stu));
    memcpy(p_new + p_ss->size, p_stu, sizeof(struct Stu));
    free(p_ss->pstu);
    p_ss->pstu = p_new;
    p_ss->size++;
}
void print(struct Stu_arr *p_ss) {
    for (int i = 0; i < p_ss->size; i++) {
        printf("( %d,%s)\t", (p_ss->pstu)[i].id, (p_ss->pstu)[i].name);
    }
    printf("\n");
}
```


C++的vector

使用C++中的**标准库类型vector**
可以很轻松地完成任务。

(不需要自己管理内存的分配)
(对不同的类型,都可处理)

使用C++中的**标准库类型string**
替代C中的字符数组,编程更自如。

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;
struct Stu { //单个学生结构体
    int id;
    string name;
};
int main() {
    vector<Stu> ss;
    Stu stu1;
    stu1.id = 1; stu1.name = "张三";
    ss.push_back(stu1);
    stu1.id = 5; stu1.name = "mike";
    ss.push_back(stu1);
    for (int i = 0; i < ss.size(); i++) {
        cout << "(" << ss[i].id << ", ";
        cout << ss[i].name << ")\t";
    }
    cout << endl;
    return 0;
}
```

```
<1,张三> <5,mike>
请按任意键继续. . .
```

C++对C的扩展 (命名空间:作用域)

```
#include <stdio.h>

int x = 10;

void fun() {
    int x = 20;
    printf("fun(): x = %d\n", x);
}

int main() {
    int x = 30;
    {
        int x = 40;
        printf("main(){}: x = %d\n", x);
    }
    fun();
    printf("main(): x = %d\n", x);
    return 0;
}
```

```
main(){}: x = 40
fun(): x = 20
main(): x = 30
请按任意键继续. . .
```

作用域:

变量都有其作用域

全局变量: (全局)

局部变量: (函数内部或{}内)

相同作用域中, 同名变量只可定义一次
不同作用域中, 同名变量可重复定义,
只有新定义的起作用。

实际上, 不同作用域的同名变量所占有的
内存空间是不同的。

C++对C的扩展 (命名空间: 引入原因)

lib_A.h (A库)

```
void print();  
.....
```

lib_B.h (B库)

```
void print();  
.....
```

myprint.h (开发人员自己写的)

```
void print();  
.....
```

某程序

```
#include "lib_A.h"  
#include "lib_B.h"  
#include "myprint.h"  
int main() {  
    print();  
}
```

上面程序: 编译错误

引入原因:

在大中型项目开发过程中,经常会用到多家公司提供的类库,或者协作开发的多个小组之间,可能会使用同名的函数或者全局变量,从而造成冲突。

命名空间,是为了解决这种命名冲突而引入的一种机制。

C语言中没有该机制

C++语言中引入了该机制

C++对C的扩展 (命名空间: 基本语法)

```
#include <stdio.h>
namespace A {
    int x = 10;
    void print() { printf("A::x=%d\n", x); }
}

namespace B {
    int x = 20;
    void print() { printf("B::x=%d\n", x); }
}

int x = 30;
void print() { printf("x=%d\n", x); }
int main() {
    int x = 40;
    printf("main:x=%d\n", x); //局部变量 x
    printf("全局:x=%d\n", ::x); //全局的 x
    printf("命名空间A:x=%d\n", A::x); //A中的x
    printf("命名空间B:x=%d\n", B::x); //B中的x
    A::print(); //调用命名空间A中的 print函数
    B::print(); //调用命名空间B中的 print函数
    return 0;
}
```

基本语法:

```
namespace 空间名字 {
    变量;
    函数; .....
} (注意, 此处没有分号)
```

基本用法: 空间名字::变量名/函数名

命名空间分割了全局命名空间 (::)
每个命名空间都是一个作用域

A terminal window with a black background and white text. The output of the C++ program is displayed line by line: main:x=40, 全局:x=30, 命名空间A:x=10, 命名空间B:x=20, A::x=10, B::x=20, and 请按任意键继续. . . .

C++对C的扩展(命名空间:使用方法)

```
#include <stdio.h>
namespace A {
    int x = 10;
    void print() { printf("A::x=%d\n", x); }
}

namespace B {
    int x = 20;
    void print() { printf("B::x=%d\n", x); }
}

int main() {
    {
        //打开A::x --> x 打开 A::print --> print
        using A::x; using A::print;
        printf("x=%d\n", x); //此时的x 是A::x
        print();           //此时的print是A::print
    }

    {
        //全部打开B
        using namespace B;
        printf("x=%d\n", x); //此时的x 是B::x
        print();           //此时的print是B::print
    }

    return 0;
}
```

使用用法

1. 空间名字::变量名/函数名
2. using 空间名字::变量名/函数名;
3. using namespace 空间名字;

第2种用法每次只能引入一个成员;
第3种用法会打开该空间中所有的成员(变量/函数等),谨慎使用

```
x=10
A::x=10
x=20
B::x=20
请按任意键继续. . .
```

C++对C的扩展 (命名空间:使用例子)

```
#include <stdio.h>
namespace A {
    int x = 10 ,y=20;
}
int x = 30;
int main() {
    //打开了A,A中的名字被“添加”到全局作用域
    {
        using namespace A;
        y++; //此处的y 是 A::y (正确)
        x++; //此次的x 可能是::x可能是A::x (错误)
        ::x++; //指明了是全局作用域的x (正确)
        A::x++; //指明了是A::x (正确)
        int x = 31; //当前的局部变量x (正确)
        x++; //此次的x 是上面的局部变量31 (正确)
    }
    {
        using A::x;
        x++; //此处的 x 是A中的x (正确)
    }
    return 0;
}
```

```
#include <stdio.h>
namespace A {
    int x = 10 ,y=20;
}
namespace B {
    int x = 30;
}
int main() {
    { using namespace A;
      using namespace B;
      y++; //A和B中冲突的名字没有用到 (正确)
    }
    { using namespace A;
      using namespace B;
      x++; //A和B中都有x , 二义性 (错误)
    }
    { using namespace A;
      int x; // 定义局部变量 (正确)
    }
    { using A::x;
      int x; //相当于重新定义A::x (错误)
    }
    return 0;
}
```

在实际使用中：
要避免**二义性**错误。

直接使用
空间名字::变量/函数
的写法是最保险的。

当然，出现这样的错误，编译器会提醒你。

C++对C的扩展(命名空间:嵌套、不连续)

命名空间支持**嵌套**

命名空间**可以是不连续的**，同名空间自动合并。

不要把 `#include` 放在命名空间内部。

```
#include <iostream>
using namespace std;
namespace A {
    int a1 = 1;
    int a2 = 2;
    namespace A1 {
        int x1 = 3;
        int y1 = 4;
    }
}
int main() {
    //方法1: A::A1::作用域符
    cout << A::A1::x1 << endl;
    //方法2
    using A::A1::x1; using A::A1::y1;
    cout << y1 << endl;
    //方法3
    using namespace A::A1;
    cout << x1 << endl;
    //方法4
    using namespace A;
    cout << A1::x1 << endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;
namespace A {
    int a1 = 1;
    int a2 = 2;
}
namespace A { //同名空间自动合并
    int b1 = 3;
    int b2 = 3;
}
int main() {
    return 0;
}
```

C++对C的扩展 (命名空间: 实际应用)

a.h

```
#ifndef _A_H
#define _A_H
// #include something

namespace ABC {
    // something...
}

#endif // !_A_H
```

a.cpp

```
#include "a.h"

namespace ABC {
    // something...
}
```

main.cpp

```
#include <iostream>
#include "a.h"
using namespace std;
using namespace ABC;
//using ABC::xxx 等等
int main() {
    ABC::Stu s1;
    return 0;
}
```

起别名 (方便书写)

namespace TV=myTelev....;

平时常用的命名空间成员的
using声明写在一个头文件中,
方便使用。(如: using
std::cout;)

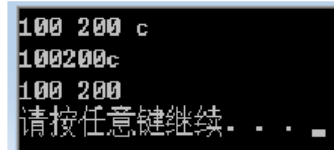
C++对C的扩展(输入输出)

C++(头文件 <code>#include <iostream></code>)		C(头文件#include <code><stdio.h></code>)	
<code>cin</code>	标准输入	键盘	<code>scanf</code>
<code>cout</code>	标准输出	屏幕	<code>printf</code>
<code>cerr</code>	标准错误输出	屏幕	<code>fprintf(stderr, ...)</code>

`cin`和`cout`是C++的标准输入流和标准输出流。
在头文件`<iostream>`中定义。
需要使用命名空间`std`

`std::cin` `std::cout` `std::endl` (推荐用法)
或者 `using std::cin; using std::cout; using std::endl;`
或者 `using namespace std;` (为了演示方便, 使用该方法)

```
#include <iostream>
using namespace std;
int main() {
    int a, b;
    char c;
    cin >> a >> b >> c;
    cout << a << b << c << endl;
    cout << a << " " << b << endl;
    return 0;
}
```



```
100 200 c
100 200 c
100 200
请按任意键继续...
```

等价于 `cin >> a; cin >> b; cin >> c;`

等价于 `cout << a; cout << " "; cout << b;`

`cin >> a;` 箭头的方向 表示 `cin` 流向 `a`(输入)

`cout << a;` 箭头的方向 表示 `a` 流向 `cout`(输出)

C++对C的扩展 (输入输出: 格式化)

C代码: 格式化输出

```
#include <stdio.h>
int main() {
    int a = 12345;
    double f = 123.4567;
    //默认输出整数
    printf("a=%d===\n", a);
    //占8格,右对齐
    printf("a=%8d===\n", a);
    //占8格,左对齐
    printf("a=%-8d===\n", a);
    //默认输出浮点数,小数显示6位
    printf("f=%f===\n", f);
    //占10格,小数显示2位,右对齐
    printf("f=%10.2f===\n", f);
    //占10格,小数显示2位,左对齐
    printf("f=%-10.2f===\n", f);
    return 0;
}
```

```
a=12345===
a= 12345===
a=12345   ===
f=123.456700===
f=  123.46===
f=123.46  ===
请按任意键继续. . .
```

C++代码:

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
    int a = 12345;
    double f = 123.4567;
    //默认输出整数
    cout << "a=" << a << "===" << endl;
    //占8格,右对齐
    cout << "a=" << setw(8) << a << "===" << endl;
    //占8格,左对齐
    cout << "a=" << setw(8) << setiosflags(ios::left)
        << a << "===" << endl;
    //默认输出浮点数,有效位数显示6位
    cout << "f=" << f << "===" << endl;
    //占10格,小数显示2位,右对齐
    cout << "f=" << setw(10) << setprecision(2)
        << setiosflags(ios::fixed) << setiosflags(ios::right)
        << f << "===" << endl;
    return 0;
}
```

```
a=12345===
a= 12345===
a=12345   ===
f=123.457===
f= 123.46===
请按任意键继续. . .
```

cout 无需关心 %d %c %f 等格式

自动识别

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
    struct mytime{
        int h;
        int m;
        int s;
    };
    mytime t1 = { 7,9,8 };
    cout << "time: " << t1.h
        << ":" << t1.m
        << ":" << t1.s
        << endl;
    cout << "time: "
        << setfill('0')
        << setw(2) << t1.h
        << ":"
        << setw(2) << t1.m
        << ":"
        << setw(2) << t1.s
        << endl;
    return 0;
}
```

```
time: 7:9:8
time: 07:09:08
请按任意键继续. . .
```

格式化输出时间

C++对C的扩展(输入输出:字符串)

C代码: 输入输出字符串

```
#include <stdio.h>
int main() {
    char str[10] = { 0 };
    scanf("%s", str);
    printf("%s\n", str);
    return 0;
}
```

```
abc de
abc
请按任意键继续
```

问题1: 输入字符串中有**空格**, 无法处理;

问题2: 输入长度超过字符数组长度, **不安全**。

```
#include <stdio.h>
#include <string.h>
int main() {
    char str[10] = { 0 };
    fgets(str, sizeof(str), stdin);
    if (str[strlen(str) - 1] == '\n')
        str[strlen(str) - 1] = '\0';
    printf("%s\n", str);
    return 0;
}
```

```
abc de
abc de
请按任意键继续
```

```
abc def ghi
abc def g
请按任意键继续
```

(解决方法)

C++代码:

```
#include <iostream>
using namespace std;
int main() {
    char str[10] = { 0 };
    cin >> str;
    cout << str << endl;
    return 0;
}
```

```
abc de
abc
请按任意键继续
```

问题1, 2 同样都存在。

```
#include <iostream>
using namespace std;
int main() {
    char str[10] = { 0 };
    cin.getline(str, sizeof(str));
    cout << str << endl;
    return 0;
}
```

```
abc de
abc de
请按任意键继续
```

```
abc def ghi
abc def g
请按任意键继续
```

(解决方法)

C++对C的扩展 (数据类型:基本内置类型)

	类型	含义	一般尺寸
整形	int	整形	4字节
	char	字符	1字节
	bool	布尔类型	1字节
浮点型	float	单精度浮点数	6位有效数字
	double	双精度浮点数	10位有效数字

```
#include <iostream>
using namespace std;
int main() {
    int i = 10;
    bool b = true;
    while (b) {
        b = i;
        cout << "b=" << b << " i=" << i << endl;
        i--;
    }
    return 0;
}
```

bool类型是C++新增的。

bool类型的取值是**true(真)**,**false(假)**

常用的数据类型:

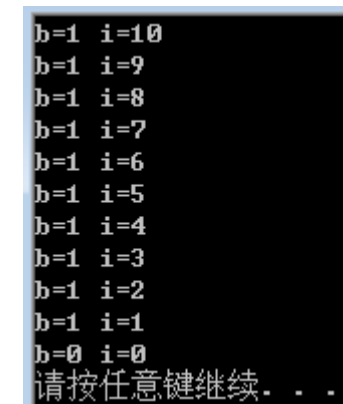
unsigned int

short, unsigned short

long, unsigned long

long long, unsigned long long

...



```
b=1 i=10
b=1 i=9
b=1 i=8
b=1 i=7
b=1 i=6
b=1 i=5
b=1 i=4
b=1 i=3
b=1 i=2
b=1 i=1
b=0 i=0
请按任意键继续. . .
```

C++对C的扩展 (数据类型:基本类型转换)

```
#include <iostream>
using namespace std;
int main() {
    //int 和 bool 转换 (bool: 1,0)
    int i = 3;
    bool b = true;
    i = b; //bool转int
    cout << "i= " << i << endl; //i=1
    cout << "b= " << b << endl; //b=1
    i = 3;
    b = i;
    cout << "b= " << b << endl; //b=1
    b = 0; //int转bool
    cout << "b= " << b << endl; //b=0

    i = 3; b = true;
    cout << i + b << endl; //4 (bool先转int再运算)

    i = 3; b = 1;
    cout << (b == i) << (i == b) << endl; //0 0
    i = 1; b = 1;
    cout << (i == b) << endl; //1

    // long, long long, 16进制, 8进制赋值
    long long_i = 123L;
    long long llong_i = 1234LL;
    int hex1 = 0xFF;
    int oct1 = 017;
    int dec1 = 18;
    cout << hex1 << " " << oct1 << " " << dec1 << endl;
    return 0;
}
```

```
i= 1
b= 1
b= 1
b= 0
4
00
1
255 15 18
请按任意键
```

```
#include <iostream>
using namespace std;
int main() {
    //int 和 char 转换 (bool: true->1,false->0)
    int i = 67;
    char c = 'A'; // 'A'的ascii码是 65
    c = i;
    cout << "c= " << c << endl; //c= C
    c = '0'; i = c;
    cout << "i= " << i << endl; //i= 48
    c = 'A'; i = c;
    cout << "i= " << i << endl; //i= 65
    c = 'A'; i = 66;
    cout << c + i << endl; //131 (运算时,char先转int)
    cout << (i > c) << endl; //1
    return 0;
}
```

```
c= C
i= 48
i= 65
131
1
请按任
```

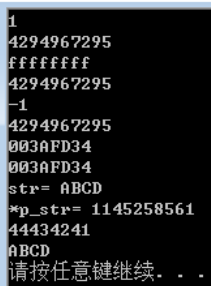
```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
    // int ,float, double的转换
    int i = 1234567890;
    float f = 1.234f;
    double fd = 3.1415926;
    f = i; // (丢失精度)
    cout << "f= " << setw(10) << setprecision(2)
        << setiosflags(ios::fixed) << f << endl;
    i = fd; //i = 3 (小数点后面丢失)
    cout << "i= " << i << endl;
    f = fd; //double转float,(精度丢失)
    cout << "f= " << setw(10) << setprecision(7)
        << setiosflags(ios::fixed) << f << endl;
}
```

```
f= 1234567936.00
i= 3
f= 3.1415925
请按任意键继续.
```

C++对C的扩展 (数据类型: unsigned 注意)

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
    //unsigned 注意点:
    unsigned int a = 2, b = 1;
    cout << a - b << endl; // 1
    cout << b - a << endl; // 4294967295
    cout << hex << b - a << dec << endl; //0xFFFFFFFF
    long long long_a = 0xFFFFFFFFLL;
    cout << long_a << endl; // 4294967295
```

补码运算



//相同的地址, 相同的内容, 只是解释方式的不同

```
unsigned int ok = b - a;
int *p_ok = (int*)&ok;
cout << *p_ok << endl; // -1
cout << ok << endl; // 4294967295
cout << &ok << endl; // 003AFD34
cout << p_ok << endl; // 003AFD34 地址与ok相同
```

*p_ok 和 ok 同一地址

内容完全相同

类型不同, 显示不同

```
char str[5] = { 'A', 'B', 'C', 'D', '\0' };
cout << "str= " << str << endl;
int *p_str = (int*)str; //小端模式(低位在低位内存)
cout << "*p_str= " << *p_str << endl; //1145258561
cout << hex << *p_str << dec << endl; //0x44434241
//反过来也是一样的
int test = 0x44434241;
char *p_char = (char*)&test;
cout << *p_char << *(p_char + 1) <<
    *(p_char + 2) << *(p_char + 3) << endl;

return 0;
}
```

低地址:
0x003AFD18

'A' (0x41) (0100 0001)
'B' (0x42) (0100 0010)
'C' (0x43) (0100 0011)
'D' (0x44) (0100 0100)

高地址:
0x003AFD1C

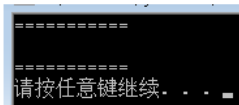
```
#include <iostream>
using namespace std;
int main() {
    //(int 和 unsigned运算时,int会先转为unsigned)
    cout << "=====" << endl;
    int j = -1;
    char arr[9] = { 1,2,3,4,5,6,7,8,9 };
    while (j < sizeof(arr)) {
        if (j == -1)
            cout << j << " ";
        else
            cout << arr[j] << " ";
        j++;
    }
    cout << endl;
    cout << "=====" << endl;

    return 0;
}
```

期望输出:

```
=====
-1 1 2 3 4 5 6 7 8 9
=====
```

实际结果



C++对C的扩展(数据类型:声明、列表初始化)

main.cpp

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

extern int g_counts;

// int g_counts; 编译报错,重定义!

int main() {
    cout << g_counts << endl;
}
```

```
int i1;           //定义
int i2 = 20;     //定义
extern int i3 = 30; //定义

extern int i4;   //声明
```

变量可多次声明,但只能定义1次。

my1.cpp

```
int g_counts = 100;

void fun(){}
```

```
#include <iostream>
using namespace std;
int main() {
    int i1 = 0;
    int i2 = { 0 };
    int i3(0);      //c++
    int i4{ 0 };   //c++
    return 0;
}
```

列表初始化: 由一组花括号括起来的初始值进行初始化

使用该方式对内置类型变量初始化时,假如存在丢失信息的风险,编译器直接**报错**。

```
#include <iostream>
using namespace std;
int main() {
    double fd = 3.14;
    int i1(fd), i2 = fd;      //ok
    int i3{ fd }, i4 = { fd }; //error
    return 0;
}
```

C++对C的扩展 (数据类型: 指针和引用)

```
#include <iostream>
using namespace std;
int main() {
    int i = 10, j = 20;
    int *pi = &i; //pi是指向i的指针
    cout << *pi << " " << i << endl; // *pi 相当于 i
    *pi = 100;
    cout << *pi << " " << i << endl; //通过指针改变值
    pi = &j; //pi指针指向其他对象了
    cout << *pi << " " << j << endl;
}
```

//空指针:

//NULL 在c中的定义是 ((void*)0)

//NULL 在c++中定义是 0

//nullptr 在c++中用来表示空指针

```
int *p = nullptr;
```

```
if (p)
```

```
    cout << "p is true" << endl;
```

```
else
```

```
    cout << "p is false" << endl;
```

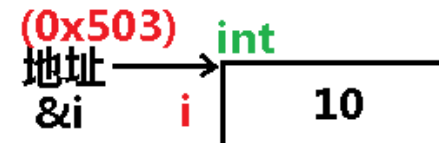
```
return 0;
```

```
}
```

```
10 10
100 100
20 20
p is false
请按任意键继续
```

指针实际上是**地址**，指针变量用来存放指针（地址）。

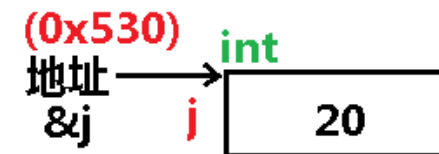
指针变量也是一种变量，同样要占用一定的存储空间。
指针的存储空间存放的是一个地址。



```
int * pi = &i;
```

```
*pi = 100;
```

```
pi = &j;
```



```
#include <iostream>
using namespace std;
int main() {
    int i = 100;
    int &ref_i = i; //ok
    cout << i << " " << ref_i << endl;
    //int &ref_i2; //错误(引用必须被初始化)
}
```

//注意书写方式:

```
int *a1 = nullptr, a2 = 0;
```

```
//上面一行: a1是指针, a2是int变量
```

```
int &r1 = a2, r2 = a2;
```

```
//上面一行: r1是a2的引用, r2是int变量
```

```
return 0;
```

```
100 100
请按任意键继续
```

} 引用是给变量或对象起一个**别名**。
定义时必须初始化，一旦绑定，终身不变。
引用并不占用存储空间。

C++对C的扩展 (数据类型: 引用)

引用是给变量或对象起一个**别名**。

定义时必须初始化，初始化是为了将该引用与它引用的变量或对象绑定，一旦绑定，该引用无法再重新绑定别的变量或对象。

引用并不占用存储空间。

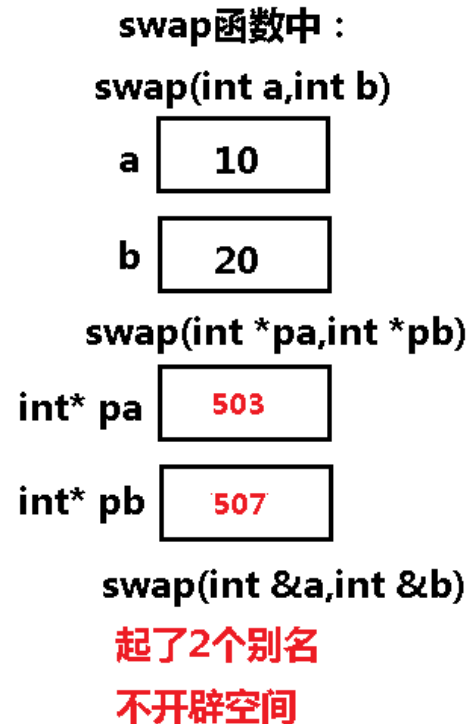
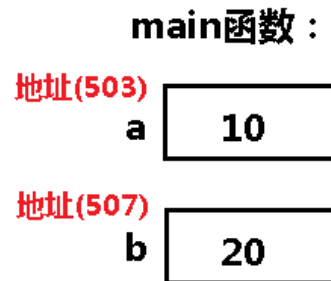
```
#include <iostream>
using namespace std;
int main() {
    int a = 10;
    int &ra = a;
    cout << &a << &ra << endl; //地址完全一样
    cout << a << ra << endl;  //值完全一样

    int &ra2 = ra; //给ra再起一个别名, ok

    int *pa = &a; //pa 是指针
    int *&rpa = pa; //给 pa 起个别名叫 rpa, ok

    return 0;
}
```

```
#include <iostream>
using namespace std;
void swap1(int *pa, int *pb) {
    int tmp = *pa;
    *pa = *pb;
    *pb = tmp;
}
void swap2(int &a, int &b) {
    int tmp = a;
    a = b;
    b = tmp;
}
int main() {
    int a = 10, b = 20;
    swap1(&a, &b);
    cout << a << b << endl; //20 10
    swap2(a, b);
    cout << a << b << endl; //10 20
    return 0;
}
```



C++对C的扩展(数据类型:引用)

数组的引用:

```
#include <iostream>
using namespace std;
int main() {
    int a = 1, b = 2, c = 3;

    //OK,指针数组
    int *p_arr[] = { &a,&b,&c };

    //错误,在数组中存放引用是不行的
    //int &r_arr[] = { a,b,c };

    int arr[3] = { a,b,c };
    int(*p1)[3] = &arr; //ok,数组指针
    int(&r1)[3] = arr; //ok,数组的引用
    return 0;
}
```

引用的本质: 从下面的代码,可以猜测,引用好像就是个指针。通过汇编代码的分析,可以推测引用实际上是通过指针来实现的。引用是指针的一种包装: 类似 `int * const p` 这样的格式的一种包装。

```
#include <iostream>
using namespace std;
struct Stu {
    int age;
    char sex;
    char name[20];
};
struct A { int &data; };
struct B { char &data; };
struct C { Stu &data; };
int main() {
    cout << sizeof(A) << endl; //4
    cout << sizeof(B) << endl; //4
    cout << sizeof(C) << endl; //4
    return 0;
}
```

C++对C的扩展 (数据类型: const 限定符1)

C 代码:

```
#include <stdio.h>
int main() {
    const int i = 10;
    //const int i; //错误, const变量必须在定义时初始化
    //i=100; //错误, const类型不能修改
    int *p = &i; //将i的地址赋值给指针p(在C中ok)
    *p = 20; //通过指针修改const int i的值
    printf("i=%d,*p=%d\n", i, *p); // 20 20
    return 0;
}
```

```
i=20,*p=20
请按任意键继续
```

C++增强了对类型的限制。

const 限定符在C++编程中会经常用到。

```
const int i = 10; //编译时初始化
const int i = get_size(); //运行时初始化
int j = 20; const int i = j; //用来初始化的值是否常量无关紧要
```

默认情况下: **const** 对象仅在文件内有效。

假如要在多个文件中生效, 则**const**变量不管是**声明**还是**定义**, 都加上**extern**关键字。

C++代码:

```
#include <iostream>
using namespace std;
int main() {
    const int i = 10;
    //int *pi = &i; 编译错误(C++中不行)
    const int *pi = &i; //ok
    //*pi = 20; 编译错误(指向常量的指针无法修改常量)

    //下面尝试, 强转转换来修改常量:
    int *pi2 = (int*)&i; //将常量i的地址强转为int *
    *pi2 = 20; //将pi2指针指向的地址内容修改为20
    //观察 *pi2 和 i 对应的内存地址是否一样:
    cout << "pi2=" << pi2 << " &i=" << &i << endl;
    //观察 *pi2 和 i 的值
    cout << "*pi2=" << *pi2 << " i=" << i << endl;
    //输出 *pi2=20 i=10
    //思考: 为什么会出现这样的结果?

    //一定要这么做, 怎么做?
    volatile const int ii = 10; //使用volatile关键字
    int *pii = (int*)&ii;
    *pii = 20;
    cout << "*pii=" << *pii << " ii=" << ii << endl;
    //输出 *pii=20 ii=20

    return 0;
}
```

```
pi2=003DF904 &i=003DF904
*pi2=20 i=10
*pii=20 ii=20
请按任意键继续...
```

C++对C的扩展 (数据类型: const 限定符2)

引用与const

```
#include <iostream>
using namespace std;
int main() {
    const int i = 10;
    //int &ri = i; 错误,非常量引用指向常量
    int ii = 20;
    const int &rii = ii; //OK
    //rii = 30; 错误, 常量引用无法修改值

    double fd = 1.23;
    //int &r = fd; 错误
    const int &r = fd; //OK
    //观察 fd 和 r 的值
    cout << fd << " " << r << endl;
    //观察 fd 和 r 的地址
    cout << &fd << endl;
    cout << &r << endl;

    return 0;
}
```

```
1.23 1
0044FA38
0044FA20
请按任意键继续
```

r 绑定了一个 **临时量**

```
const int temp = fd;
const int &r = temp;
```

指针与const

```
#include <iostream>
using namespace std;
int main() {
    int i = 10, j = 30;
    int *p1 = &i; //无const限定
    *p1 = 20; //可以改变指向变量的值(i-->20)
    p1 = &j; //可以改变指向的变量(p1指向了j)

    //指向常量的指针
    const int *p2; //const在*前面(也可写int const *p2)
    p2 = &i; //p2 的指向 可以改变 (意味着p2不是常量)
    p2 = &j;
    // *p2 = 100; 错误,*p2改不了值(意味着*p2是常量)

    //常量指针(指针本身是常量)
    int * const p3 = &i; //const在*后面
    //p3 = &j; 错误,p3的指向不能改变(p3是常量)
    *p3 = 100; //OK, *p3可以修改

    //指向常量的常量指针
    const int * const p4 = &i; //两个const
    //p4 = &j; 错误
    // *p4 = 100; 错误
    return 0;
}
```

顶层const: 指针本身是常量

底层const: 指针指向的对象是常量

C++对C的扩展 (数据类型: struct和class)

```
#include <stdio.h>
typedef void(*Train)(struct player*, int);
typedef void(*Pk)(struct player*, struct player*);
struct player {
    int level; //等级
    int hp; //hp值
    Train f_train; //函数指针(练级)
    Pk f_pk; //函数指针(PK)
};

void train_fun(struct player *p1, int nums) {
    int killnums = p1->hp > nums ? nums : p1->hp;
    p1->level += killnums; p1->hp -= killnums;
    printf("练级:长了 %d 级。", killnums);
    printf("当前: level=%d, hp=%d\n", p1->level, p1->hp);
}

void pk_fun(struct player* p1, struct player* p2) {
    int power1 = p1->level * 100 + p1->hp;
    int power2 = p2->level * 100 + p2->hp;
    if (power1 >= power2)
        printf("player1 win\n");
    else
        printf("player2 win\n");
}

int main() {
    struct player p1 =
    { .level = 1, .hp = 100, .f_train = train_fun, .f_pk = pk_fun };
    struct player p2 =
    { .level = 2, .hp = 50, .f_train = train_fun, .f_pk = pk_fun };
    p1.f_train(&p1, 6);
    p2.f_train(&p2, 10);
    p1.f_pk(&p1, &p2);
    return 0;
}
```

C代码

```
练级: 长了 6 级。当前: level=7, hp=94
练级: 长了 10 级。当前: level=12, hp=40
player2 win
请按任意键继续. . .
```

```
#include <iostream>
using namespace std;
class player {
public:
    player(int level=0, int hp=0)
        :level(level), hp(hp){}
    void train(int nums) {
        int killnums = hp > nums ? nums : hp;
        level += killnums; hp -= killnums;
        cout << "练级:长了 " << killnums << " 级。";
        cout << "当前: level=" << level << ", hp=" << hp << endl;
    }
    void pk(player &another) {
        int power1 = level * 100 + hp;
        int power2 = another.level * 100 + another.hp;
        if (power1 >= power2)
            printf("You win!\n");
        else
            printf("You loss!\n");
    }
private:
    int level; //等级
    int hp; //hp值
};

int main() {
    player p1(1, 100); player p2(2, 50);
    p1.train(6);
    p2.train(10);
    p1.pk(p2);
    return 0;
}
```

C++代码

结构体是一种自定义数据类型。
标准C中的结构体是不允许包含成员函数
C语言结构体中可通过函数指针的方式加函数
C++中的结构体对此进行了扩展

```
练级: 长了 6 级。当前: level=7, hp=94
练级: 长了 10 级。当前: level=12, hp=40
You loss!
请按任意键继续. . .
```

C++对C的扩展 (数据类型: 字符串)

```
#include <stdio.h>
#include <string.h>
int main() {
    //字符数组
    char str1[20] = "abcde"; //初始化
    char str2[20] = { 'a','b','c' }; //初始化
    //str2 = "abc"; 错误
    char str3[20];
    str3[0] = 'a'; str3[1] = 'b'; str3[2] = '\0';
    //字符指针
    char *pstr = "bcd"; //将常量字符串的地址赋给pstr
    pstr = "def";
    pstr = str1;
    pstr[0] = 'x'; //通过指针修改
    *(pstr + 1) = 'y'; //通过指针修改
    printf("str1=%s\n", str1); // 输出xycde
    //字符串长度
    printf("str1长度= %d\n", strlen(str1)); //5
    //字符串拷贝
    printf("str1=%s\n", strcpy(str1, "ddd")); //ddd
    //字符串连接
    printf("str1=%s\n", strcat(str1, str2)); //dddabc
    //字符串比较
    if (strcmp(str2, str3) > 0)
        printf("%s > %s\n", str2, str3);
    else if (strcmp(str2, str3) == 0)
        printf("%s == %s\n", str2, str3);
    else
        printf("%s < %s\n", str2, str3);
    //字符串查找
    strcpy(str2, "--ab=="); //str3: "ab"
    printf("%s\n", strstr(str2, str3)); //ab==
    return 0;
```

C代码

```
str1=xycde
str1长度= 5
str1=ddd
str1=dddabc
abc > ab
ab==
请按任意键继续.
```

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    //std::string
    std::string str1("abc"); //初始化
    string str2 = "bcd"; //初始化
    str2 = "defg"; //可以直接赋值
    str2 = str1; //可以直接赋值

    const char *pstr = str2.c_str(); //转c风格字符串
    str2[0] = 'X'; //可以直接下标访问操作
    str2.at(1) = 'Y'; //可以 at 访问操作
    cout << "str2=" << str2 << endl; //XYc

    //求字符串长度
    cout << str2.size() << endl;
    cout << str2.length() << endl;
    //strlen(str1); 错误
    cout << strlen(str2.c_str()) << endl; //正确
    //字符串连接
    str2 = str2 + str1 + "!!";
    cout << "str2=" << str2 << endl; //XYcabc!!
    //字符串比较 (str1: abc)
    cout << str2.compare(str1) << endl; //-1
    cout << (str2 < str1) << endl; //1
    //字符串查找
    cout << str2.find(str1) << endl; //3
    //字符串提取
    string str3 = str2.substr(3, 3);
    cout << str3 << endl; //abc

    return 0;
```

C++代码

```
str2=XYc
3
3
3
str2=XYcabc!!
-1
1
3
abc
请按任意键继续. . .
```

C++对C的扩展 (数据类型: vector)

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;
int main() {
    //std::vector的结构
    std::vector<int> vec11; // [ 1, 3, 9 ...]
    vector<string> vec22; // [ "abc", "play", "C++" ]
    vector<vector<int>> vec33; // [ [1,3,9..],[2,3,4..], ... ]
    vector<vector<string>> vec44; // [ ["hello","C",..],["C++","abc",..],... ]

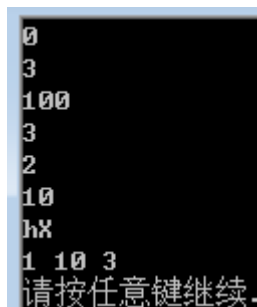
    //vector的初始化
    vector<int> vec1 = { 1,2,3 };
    vector<int> vec2{ 1,2,3 }; //列表初始化
    vector<int> vec3 = vec1; //vec1拷贝给 vec3
    vector<int> vec4(10); //初始化10个元素, 每个元素都是0
    vector<int> vec5(10, -1); //初始化10个元素, 每个元素都是-1
    vector<string> vec6(10, "hi"); //初始化10个元素, 每个元素都是 "hi"
```

```
//判断是否为空
cout << vec1.empty() << endl; //0
//元素个数
cout << vec1.size() << endl; //3
//添加元素在最后面
vec1.push_back(100);
cout << vec1[vec1.size() - 1] << endl; //100
//弹出元素在最后面
vec1.pop_back();
cout << vec1[vec1.size() - 1] << endl; //3
//直接下标访问元素
cout << vec1[1] << endl; //2
vec1[1] = 10;
cout << vec1[1] << endl; //10
// vector<string> vec6(10, "hi")
vec6[0][1] = 'X';
cout << vec6[0] << endl; //hX

//遍历 (类似遍历数组)
for (int i = 0; i < vec1.size(); i++)
    cout << vec1[i] << " "; // 1 10 3
cout << endl;

return 0;
}
```

标准库类型**vector**表示对象的集合，其中所有的对象的类型必须相同。因为**vector**“容纳着”其他对象，所以被称为“**容器**”。**vector**是一个**类模板**。



```
0
3
100
3
2
10
hX
1 10 3
请按任意键继续.
```

C++对C的扩展 (auto类型说明符)

```
//1.auto 变量必须在定义时初始化,类似于const
auto i1 = 0; auto i2 = i1;
//auto i3; //错误,必须初始化
//2.如果初始化表达式是引用,则去除引用语义
int a1 = 10;
int &a2 = a1; // a2是引用
auto a3 = a2; // a3是int类型,而不是引用
auto &a4 = a1; // a4是 引用
//3.去除顶层const
const int b1 = 100;
auto b2 = b1; // b2 是 int
const auto b3 = b1; // b3是 const int
//4.带上底层const
auto &b4 = b1; // b4 是 const int 的引用
//5.初始化表达式为数组时,推导类型为指针
int arr[3] = { 1,2,3 };
auto parr = arr; //parr 是 int * 类型
cout << typeid(parr).name() << endl;
//6.表达式为数组且auto带上&,推导类型为数组
auto &rarr = arr; //rarr 是 int [3]
cout << typeid(rarr).name() << endl;
//7.函数参数类型不能是 auto
//func(auto arg); //错误
//8.auto并不是一个真正的类型,编译时确定
//sizeof(auto); 错误
```

auto: 让编译器替我们分析表达式的类型。

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

int main() {
    vector<string> vs =
        { "all", "people", "like", "c++" };

    for (vector<string>::iterator i =
            vs.begin(); i != vs.end(); i++)
        cout << *i << " ";
    cout << endl;

    for (auto i = vs.begin(); i != vs.end(); i++)
        cout << *i << " ";
    cout << endl;

    for (auto &s : vs)
        cout << s << " ";
    cout << endl;

    return 0;
}
```

```
all people like c++
all people like c++
all people like c++
请按任意键继续. . .
```


C++对C的扩展 (练习: 进制转换, 填空)

进制转换:

输入: **整数**(如: 256)

输出: **10进制的字符串**("256")

16进制的字符串("FF")

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;
char* num2dec(int num, char* res, int length);
string num2hex(int num);

int main() {
    char res[12];
    int test = 8267100;
    cout << "10进制: " <<
        num2dec(test, res, sizeof(res)) << endl;
    string res2;
    res2 = num2hex(test);
    cout << "16进制: " << res2 << endl;
    return 0;
}
```

```
char* num2dec(int num, char* res, int length) {
    if (num == 0) {
        res[0] = '0'; res[1] = '\0'; return res;
    }
    memset(res, 0, length);
    int idx = 0;
    while (num) {
        char c = num % 10 +  ;
        res[idx++] = c;
        num /= 10;
    }
    //0 和 idx-1 的位置 对换 (数组前后交换)
    for (int i = 0; i <= (idx - 1) / 2; i++) {
        char tmp = res[i];
        res[i] = res[idx - 1 - i];
        res[idx - 1 - i] = tmp;
    }
    return res;
}
```

```
string num2hex(int num) {
    if (num == 0) return "0";
    vector<char> vec;

    while (num) {
        int yu = num % 16;
        vec.push_back("0123456789ABCDEF"[yu]);
        num /= 16;
    }
    string tmp;
    for (int i = vec.size() - 1; i >= 0; i--)
        tmp += vec[i];
    return tmp;
}
```

思路:

通过**取余操作**逐个得到num的个位、十位、百位.....存入数组
然后反向取出

C++对C的扩展 (练习: 进制转换)

进制转换:

输入: **整数**(如: 256)

输出: **10进制的字符串**("256")

16进制的字符串("FF")

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;
char* num2dec(int num, char* res, int length);
string num2hex(int num);

int main() {
    char res[12];
    int test = 8267100;
    cout << "10进制: " <<
        num2dec(test, res, sizeof(res)) << endl;
    string res2;
    res2 = num2hex(test);
    cout << "16进制: " << res2 << endl;
    return 0;
}
```

```
char* num2dec(int num, char* res, int length) {
    if (num == 0) {
        res[0] = '0'; res[1] = '\0'; return res;
    }
    memset(res, 0, length);
    int idx = 0;
    while (num) {
        char c = num % 10 + '0';
        res[idx++] = c;
        num /= 10;
    }
    //0 和 idx-1 的位置 对换 (数组前后交换)
    for (int i = 0; i <= (idx - 1) / 2; i++) {
        char tmp = res[i];
        res[i] = res[idx - 1 - i];
        res[idx - 1 - i] = tmp;
    }
    return res;
}
```

```
10进制: 8267100
16进制: 7E255C
请按任意键继续.
```

```
string num2hex(int num) {
    if (num == 0) return "0";
    vector<char> vec;

    while (num) {
        int yu = num % 16;
        vec.push_back("0123456789ABCDEF"[yu]);
        num /= 16;
    }
    string tmp;
    for (int i = vec.size() - 1; i >= 0; i--)
        tmp.push_back(vec[i]);
    return tmp;
}
```