

面向对象程序设计与实践2

交换与智能控制研究中心
北京邮电大学

C++知识简单回顾

- C++语句的基本部分：字符集、关键字、标识符、操作符
- 程序设计
 - 程序=算法+数据结构
 - 程序=对象+对象间的相互作用
- 数据
 - 数据类型：基本数据类型，自定义数据类型
 - 数据类型的转换
 - 数据的输入输出
- 算法的基本控制结构：顺序、选择、循环

字符集—C++语言的基本元素

□ 大小写的英文字母

- A~Z , a~z

□ 数字字符

- 0~9

□ 符号

- ! # % ^ & * : _ + = - ~ < > / \ ' " ; . , () [] { }

□ 其他符号

- 空格符、换行符、<TAB>

关键字--C++预定义的单词

□ 关键字是C++保留的，作为专用定义符的单词，在程序中不允许另作它用

■ auto, break, case, char, class, const, continue, default, do, default, delete, double, else, enum, explicit, extern, false, float, for, friend, if, inline, int, long, mutable, new, operator, private, protected, public, register, return, short, signed, sizeof, static, static_cast, struct, switch, this, ture, typedef, union, unsigned, virtual, void, while

标识符&操作符

- 标识符：程序员声明的单词，它命名程序正文中的一些实体
 - 函数名、变量名、对象名等
- 操作符
 - 用于实现各种运算的符号： $+$ $-$ $*$ $/$...

C++概念的深入理解——const (1)

□ 为什么要有常量定义？

- 易读性
- 正确性
- 易修改性

□ const的用法与用途

- **const int A = 2;** **// #define A 2;**
- **const char *B = “abc”;**
- **char const *B = “abc”;**
- **char * const B = “abc”;**
- ~~**const int * const A = 2;**~~

C++概念的深入理解—const (2)

❑ `const char *B = "abc";`

❑ `char const *B = "abc";`

■ 不能通过B来改变其所指的字符串的内容，但是可以使B指向其他地方，例如：

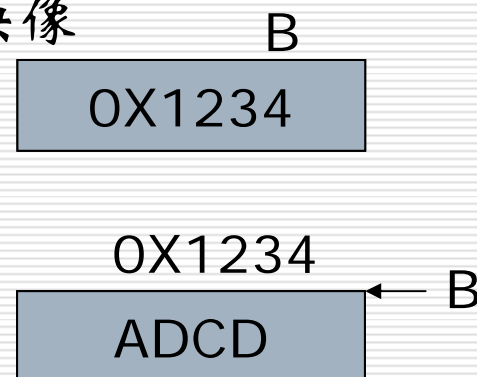
❑ `const char a, c;`

❑ `char const *B;`

❑ `B = &a;`

❑ `B = &c;`

内存映像



C++概念的深入理解—const (3)

□ char * const B;

- 此声明表示
即该指针是
方，一般该

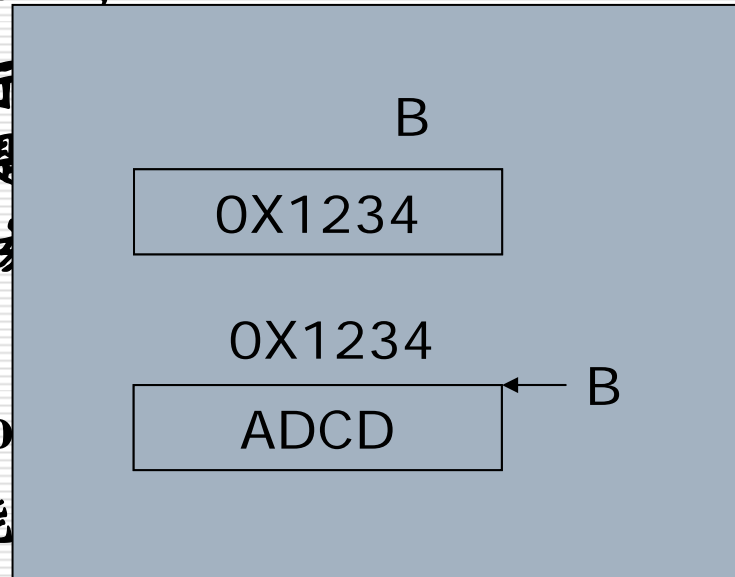
个指针常量，
向其他地
，例如：

□ char a;

□ char * co

□ 此时不能

以改变B所指向内存的值，例如：(*B) = 's';



char c;
B = &c; ✗

C++概念的深入理解—const (4)

□ 规律：从后向前读

- `const char *B = "abc";`
- `char const *B = "abc";`
- `char * const B = "abc";`
- `const int * const A = 2;`

C++概念的深入理解—const (5)

□ const用于函数声明

- 返回值
- 参数
- 函数本身

□ 函数返回值是常量

```
const Rational operator * ( const Rational& lhs, const Rational& rhs);
```

```
Rational a, b, c;  
(a * b) = c;    //如果不设置返回值为常量，谁能阻止这样的暴行！
```

C++概念的深入理解—const (6)

□ **class A{**

■ ...

■ **int f() const {};** //函数本身是const f1

■ **int f() {};** // f2, 可以修改对象的属性

■ ...

■ **};**

□ **A a;** **a.f();** // calls f2

□ **const A a;** **a.f();** // calls f1

C++概念的深入理解—const (7)

- 思考: `const char *func(char* const a) const;`
如何读?
- 思考: C++中为什么引入const代替define?
- 思考如下代码的结果
 - `const int i = 10;`
 - `int* const p2 = (int* const)&i;`
 - `*p2 = 111;`
 - `cout << *p2 << " " << i << endl;`

C++概念的深入理解——指针与数组（1）

□ 二者是等价的？

□ 指针与数组的差异

- 数组要么在静态存储区被创建（如全局数组），要么在栈上被创建。数组名对应着（而不是指向）一块内存，其地址与容量在生命期内保持不变，只有数组的内容可以改变。
- 指针可以随时改变指向的任意内存地址，它的特征是“可变”，所以我们常用指针来操作动态内存。指针远比数组灵活，但也更危险。

C++概念的深入理解——指针与数组（2）

□ 修改内容

- `char a[] = "hello";`
- `a[0] = 'X';`
- `cout << a << endl;`
- `char *p = "world";` // 注意p指向常量字符串
- `p[0] = 'X';` // 编译器不能发现该错误
- `cout << p << endl;`

`const char *p = "world";`

C++概念的深入理解——指针与数组 (3)

□ 内容复制与比较

- 不能对数组名进行直接复制与比较

□ `char a[] = "hello";` `char b[10];`

□ `b = a;` `if (b == a)` // 编译错误

□ `strcpy(b, a);` 或者 `if(strcmp(b, a) == 0)`

- 指针可以

□ `char *a = "hello"; char *b = new char[10];`

□ `b = a;` `if (b == a)` // 完全正确！

a和b的地址
是不同的！

a和b的地址
是相同的！

`strcpy(b, a);` 或者 `if(strcmp(b, a) == 0)`

C++概念的深入理解——指针与数组（4）

□ 计算内存容量

- 用运算符sizeof可计算出数组的容量（字节数）

```
void Func(char a[100])  
{  
    cout << sizeof(a) << endl; // 4字节而不是100字节  
}
```

□ cout << sizeof(p) << endl;

- 注意：数组作为函数的参数进行传递时，该数组自动退化为同类型的指针。

C++概念的深入理解——指针与数组 (5)

□ sizeof与strlen()的区别？

- sizeof是运算符，strlen是函数
 - sizeof结果类型是size_t (unsigned int)
- sizeof在编译时计算，strlen在运行时计算
- sizeof可以用类型或函数做参数，而strlen只能用char*做参数，且必须以\0结尾
- 数组做sizeof的参数不退化，而传递给strlen就退化为指针（尽管这一条本身的影响并不大）

C++概念的深入理解——指针与数组 (6)

- 数组指针和指针数组，二者有什么区别么？
- 数组指针：`int (*f)[15];`
 - 一个指针，指向一个数组；
 - 比如 `int e[15];`
 - `int (*f)[15];`
 - `f = &e;`
 - 则 `(*f)` 与 `e` 是一样的了。 `f` 是一个数组指针。

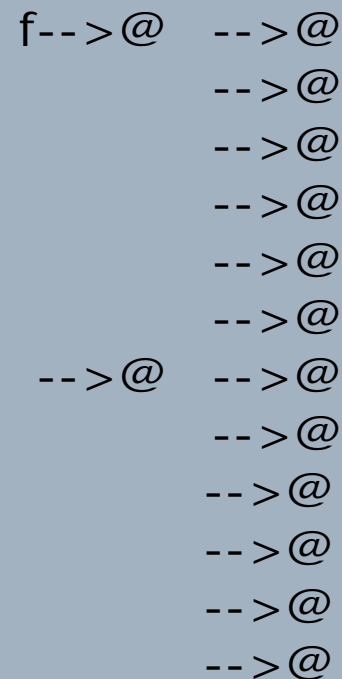
注意： `**f = (*f)[0] = e[0]`
使用 `f` 时，必须使用 `(*f)[]`，或者 `(*f) + n`

C++概念的深入理解——指针与数组 (7)

□ 指针数组

- `int *(f[15]) ; 或int *f[15];`
- `f`是一个数组，每个数组元素是一个指针

- 注意:在使用指针数组时，数组的每个成员都是指针，在使用前要先分配内存。



The diagram illustrates a pointer array in memory. It consists of a light blue rounded rectangle containing text. On the left side, the label 'f-->' is followed by a series of arrows pointing to memory locations marked with '@'. The first arrow is on the same line as the label, and the subsequent arrows are on lines below it. On the right side, there is a vertical column of memory locations, each marked with '@'. The first '@' is aligned with the first arrow, and the subsequent '@' symbols are aligned with the arrows below it. This visualizes how a pointer array stores multiple pointers to different memory locations.

C++概念的深入理解—指针与数组 (8)

□ 数组指针的用法

```
int (*d)[];  
int(*c)[5];  
int a[5];  
int b[10];
```

```
d=a;  
d=b;
```

```
c=a;  
c=b; //编译器警告
```

int (*d)[]; 则可以随便指向任

在给指针赋值时,所指向的数
时会有警告,但是运行仍然

、为15,那么(*d)[8]仍然可以正

向数组指针所赋的数组的大小
大小一定要一致。

C++概念的深入理解——指针与数组 (9)

□ 读法的规律

- 括号内先读，或者按照操作符的优先级
- 先右后左
- `int *(f[15]) ; 或int *f[15];`
- `int (*f)[15];`

□ 函数指针

- `int (*p)(char);`

- 数据指针指向数据存储区，而函数指针指向的是程序代码存储区

C++概念的深入理解——指针与数组 (10)

□ `char*(*c[10])(int **p);`

■ `c`是一个含有10个函数指针的数组，这些函数参数类型为`int**`，返回值为`char*`

□ 思考：`int (*(*f)(char *, double))[10][15];`

C++概念的深入理解——指针（1）

□ 什么是内存泄漏？

- 用动态存储分配函数动态开辟的空间，在使用完毕后未释放，结果导致一直占据该内存单元。
- 程序申请一块内存，且没有任何一个指针指向该内存。

□ 什么是野指针？

C++概念的深入理解——指针 (2)

□ 传递参数

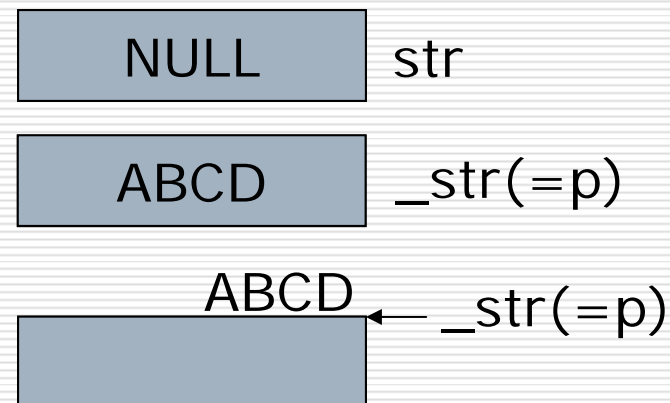
```
void Func1(int x)
{
    x = x + 10;
}
...
int n = 0;
Func1(n);
```

```
void Func2(int *x)
{
    (* x) = (* x) + 10;
}
...
int n = 0;
Func2(&n);
```


C++概念的深入理解——指针 (3)

```
void GetMemory(char *p, int num)
{
    p = new char[num];
}
void Test(void)
{
    char *str = NULL;
    GetMemory(str, 100);
    strcpy(str, "hello");
}
```

内存映像

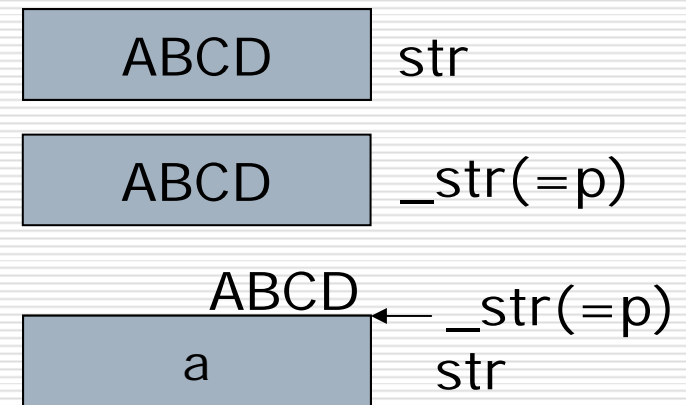


str的副本是 _str;
p == _str;

C++概念的深入理解——指针（4）

```
void ModMemory (char *p)
{
    (*p) = 'a';
}
void Test(void)
{
    char *str=NULL;
    ...
    ModMemory(str);
}
```

内存映像



`str`的副本是 `_str`;
`p == _str`;

C++概念的深入理解——指针 (5)

```
void GetMemory2(char **p, int num)
{
    *p = new char[num];
}
void Test2(void)
{
    char *str = NULL;
    GetMemory2(&str, 100); // 注意参数是 &str, 而不是str
    strcpy(str, "hello");
    cout << str << endl;
    delete str;
}
```

C++概念的深入理解——指针 (6)

```
char *GetMemory3(int num)
{
    char *p = new char[num];
    return p;
}

void Test3(void)
{
    char *str = NULL;
    str = GetMemory3(100);
    strcpy(str, "hello");
    cout<< str << endl;
    delete str;
}
```

C++概念的深入理解——指针 (7)

```
char *GetString(void)
{
    char p[] = "hello world";
    return p;           // 编译器将提出警告
}

void Test4(void)
{
    char *str = NULL;
    str = GetString();  // str 的内容是垃圾
    cout << str << endl;
}
```

C++概念的深入理解——指针 (8)

```
char *GetString2(void)
{
    char *p = "hello world"; // p[] → *p
    return p;                // 编译器将提出警告
}
void Test5(void)
{
    char *str = NULL;
    str = GetString2();
    cout << str << endl;
}
```

C++概念的深入理解——指针 (9)

□ delete把指针怎么啦？

```
char *p = new char[100];  
strcpy(p, "hello");
```

```
// p 所指的内存被释放，但是p  
    所指的地址仍然不变  
delete p;          ...
```

// 没有起到防错作用

```
if(p != NULL){  
    strcpy(p, "world"); // 出错  
}
```

执行前内存映像

ABCD p

EFGH ABCD

Delete p;

执行后内存映像

ABCD p

EFGH ABCD

C++概念的深入理解——指针 (10)

```
class A
{
public:
    void Func(void){ ... }
};
void Test(void)
{
    A *p;
    {
        A a;
        p = &a;           // 注意 a 的生命期
    }
    p->Func();           // p是"野指针"
}
```


C++概念的深入理解——指针 (11)

□ 动态内存会被自动释放吗？

```
void Func(void)
{
    char *p = new char[100];
}
```

- 指针消亡了，并不表示它所指的内存会被自动释放 --- 内存泄漏
- 内存被释放了，并不表示指针会消亡或者成了NULL指针 --- 野指针

C++概念的深入理解——指针 (12)

□ 有了malloc/free为什么还要new/delete ?

■ 0、new/delete用起来更方便、简单

➤ `Obj *a = (obj *)malloc(sizeof(obj));`

➤ `Obj *a = new Obj;`

■ 1、malloc与free是C/C++语言的标准库函数； new/delete是C++的运算符，效率更高

■ 2、new不必用sizeof()计算分配的内存字节数

`p = (char *)malloc(sizeof(char) * num);`

`P = new char[num];`

C++概念的深入理解——指针（13）

□ 有了malloc/free为什么还要new/delete？

■ 3、new不需要进行类型转换

➤ `Obj *a = (Obj *)malloc(sizeof(Obj));`

➤ `Obj *a = new Obj;`

■ 4、new可以对分配的内存进行初始化

➤ 初始化非内部数据类型，内部数据类型不初始化

■ 5、new和delete可以被重载，程序员可以借此扩展new和delete的功能，建立自定义的存储分配系统

C++概念的深入理解——指针 (14)

```
class Obj
{
public :
    Obj(void){ cout << "Initialization" << endl; }
    ~Obj(void){ cout << "Destroy" << endl; }

    void Initialize(void){ cout << "Initialization" <<
        endl; }
    void Destroy(void){ cout << "Destroy" <<
        endl; }
};
```

C++概念的深入理解——指针 (15)

```
void UseMallocFree(void)
{
    // 申请动态内存
    Obj *a = (obj *)malloc(sizeof(obj));
    a->Initialize();    // 初始化

    //...
    a->Destroy();       // 清除工作
    free(a);           // 释放内存
}
```

```
void UseNewDelete(void)
{
    // 申请动态内存并且初始化
    Obj *a = new Obj;

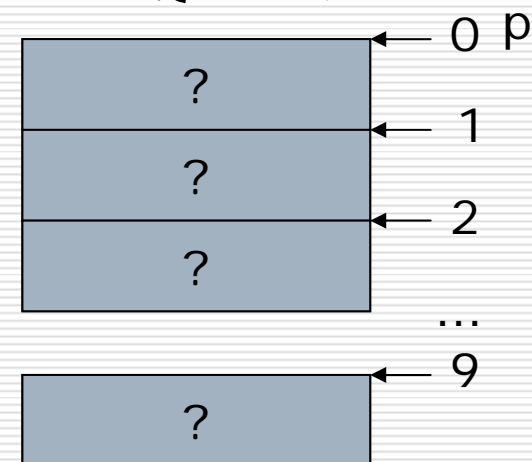
    //...
    // 清除并且释放内存
    delete a;
}
```

C++概念的深入理解——指针（16）

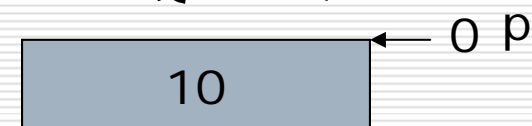
□ new/delete 注意事项

- `int *p = new int[10];`
- `int *p = new int(10);`

内存映像



内存映像



C++概念的深入理解——指针 (17)

```
class Obj
{
public:
    Obj(void);    // 无参数的构造函数
    ...
    // 创建100个动态对象
    Obj *objects = new Obj[100];
    ...
}
void ...
{
    // 创建100个动态对象的同时赋初值1
    Obj *objects = new Obj[100](1);
    Obj *a = new Obj;
    Obj *b = new Obj(1);    // 初值为1
    ...
    delete a;
    delete b;
}
}
```



C++概念的深入理解——指针 (18)

□ 删除对象

```
int *i = new int[5];  
□ Char *p = new char[6];  
  
delete i;    // delete []i;  
delete p;    // delete []p;
```


C++ 语言概念--引用

- 引用就是别名

```
int m;
```

```
int &n = m;
```

- ```
int i = 5;
```

- ```
int j = 6;
```

```
int &k = i;
```

```
k = j; // k和i的值都变成了6;  
      // i=j
```

其

引用的使用规则和与指针的比较

- 1. 不能有NULL引用，引用必须与合法的存储单元关联

- 指针则可以是NULL

- 2. 引用被创建的同时必须被初始化

- 指针则可以

- 3. 一旦引
系

- 指针则可以

```
char *pc = NULL;
```

```
char *pc;  
...  
if(pc) ...;
```

变引用的关

引用和指针的使用原则

□ 使用指针

- 有可能什么都不指向（指针设为空）
- 不同的时候指向不同的对象

□ 使用引用

- 总是会指向某一个对象
- 一旦确定指定的对象，就不再改变

引用的主要功能

- 传递函数的参数和返回值
- C++语言中，函数的参数和返回值的传递方式有三种

■ 值传递、指针传递和引用传递

```
void Func1(int x)
{
    x = x + 10;
}
...
int n = 0;
Func1(n);
```

```
void Func2(int *x)
{
    (* x) = (* x) + 10;
}
...
int n = 0;
Func2(&n);
```

引用的主要功能

```
void Func1(int x)
{
    x = x + 10;
}
...
int n = 0;
Func1(n);
```

```
void Func3(int &x)
{
    x = x + 10;
}
...
int n = 0;
Func3(n);
```

```
void Func2(int *x)
{
    (* x) = (* x) + 10;
}
...
int n = 0;
Func2(&n);
```

引用的使用场景

- 主要应用于自定义数据类型的参数传递
- 类类型的参数
 - 使用方便－相对于指针
 - 提高效率－相对于传值
 - 操作符重载时作为返回值，实现方便易懂

引用引入原因小结

□ 杀鸡焉用牛刀！

C++ 的概念——内联 (inline) (1)

- 内联：若一个函数被指定为inline函数，则它将在程序中每个调用点上被“内联地”展开
- Inline是指嵌入代码，就是在调用函数的地方不是跳转，而是把代码直接写到该处
 - 对于短小的代码，内联函数可以提高效率同时比C的宏更安全可靠

C++ 的概念——内联 (inline) (2)

□ 内联函数和普通函数相比

```
f()
{
    ...
    //使用cout << "Hello World" << endl; 替换display()
    display();
    ...
}

inline void display()
{
    cout << "Hello World" << endl;
}
```

而
被

C++ 的概念——内联 (inline) (3)

□ 内联函数和宏相比

- 宏不是函数，只是在编译前进行不加验证的简单宏体替代
- 内联函数要做参数类型检查

内联的优势

- 看起来象函数，运行起来象函数，比宏 (macro) 要好得多，还不需要函数调用的开销
- 编译器优化代码
- 如果内联函数体非常短，编译器为这个函数体生成的代码就会真的比为函数调用生成的代码要小许多

内联的劣势

- 增加整个目标代码的体积，甚至减慢程序的运行
- 大多数调试器遇上内联函数都会无能为力
 - 怎么在一个不存在的函数里设置断点？
 - 怎么单步执行到这样一个函数呢？
 - 怎么俘获对它的调用呢？

明智地使用内联 (1)

- inline指令是一种提示，而不是命令
 - 编译器拒绝内联复杂的函数（包含循环，递归）
 - 拒绝内联虚函数
 - 不要内联包含静态数据的函数
- 编译器拒绝内联时通常给出告警

明智地使用内联 (2)

- ☐ 递归函数 – 不用
- ☐ 包含循环等控制结构的函数 – 不用
- ☐ 大于10行代码的函数（当然只对变量初始化的可以忽略） – 不用
- ☐ 只有几行代码的，经常调用的函数 – 可以用

- ☐ 最后还应该注意两点：
 - (1) 内联函数的声明和定义需放在一个源代码文件中，也就是说不能让声明与实现一个在.h中一个在.cpp中。
 - (2) 类中成员函数都是隐式指示为内联函数，加inline和不加inline是相同的。

明智地使用内联 (3) – 建议原则

- 编程初期：不要内联任何函数，除非函数确实很小很简单
 - `int age() const { return personAge; }`
- 找出20%能够真正提高整个程序性能的代码
--- 如果可以内联，声明为inline
- 注意代码膨胀的问题
- 监视编译器的警告信息，看看是否有内联函数没有被编译器内联

明智地使用内联 (3)

□ 思考

■ inline 和 define 的区别及各自的优缺点？

附录--指针用法小结 (1)

1. 指针是把其他变量的地址作为其值的变量。
 2. 指针必须在使用之前声明。
 3. 声明语句 `int *Ptr;` 读法为: ptr是一个int类型的指针。
声明语句中用到的*表示该变量是一个指针。
 4. 有三种值可用来初始化一个指针, 他们是0、NULL、和一个地址。把一个指针初始化为0和初始化为NULL是等价的。
 5. 能够赋给指针的唯一的整数是0。
 6. 地址运算符(&)返回其操作数的地址。
 7. 地址运算符的操作数必须是一个变量, 不能把地址运算符用于常量、表达式。
 8. 运算符*称为“间接引用运算符”或“复引用运算符”, 它返回其操作数所指向的对象的值。这种用法称为“指针的复引用”。
 9. 在调用参数接收地址的函数时, 如果调用函数要求被调用函数修改参数的值, 应该把参数的地址传递给被调用函数, 被调用函数然后用间接引用运算符(*)修改调用函数中的参数的值。
-

附录--指针用法小结 (2)

10. 用参数接收地址的函数必须用指针作为其相应的参数。
 11. 函数原型中不一定要包含指针的名字，只需要包含指针类型。可以为了提供可读性而在函数原型中包含指针名，但是编译器会忽略这个名字。
const 限定符通知编译器：指定变量的值不应该被修改。
 12. 如果试图修改被声明为 const 的值，编译器会报错。
 13. 传递给函数的指针有四种：指向非常量数据的非常量指针，指向非常量数据的常量指针，指向常量数据的非常量指针，指向常量数据的常量指针。
 14. 因为数组名的值是数组的地址，所以数组是自动以传引用方式传递的。
 15. 以传引用方式传递一个数组元素必须传递指定数组元素的地址。
 16. 为了在编译程序时确定数组（或其他数据类型）所占用的内存字节数，提供了专门的单目运算符 sizeof。
 17. sizeof 运算符在用于数组名时以整数形式返回该数组元素所占的字节数。
 18. sizeof 运算符可用于任何变量名、变量类型和常量。
-

附录--指针用法小结 (3)

19. 指针算术运算包括自增 (++)、自减 (--)、与整数相加 (+或+=)、减去一个整数 (-或-=)、以及减去另一个指针。
20. 当一个指针加上或者减去一个整数时, 指针加上或者减去该整数与指针引用的对象的大小的乘积。
21. 指针运算符只能用于数组这种连续存储的内存区。
22. 在对指向字符数组的指针进行算术运算时, 因为每一个字符都是一个字节长, 所以计算结果与通常的算术运算是一致的。
23. 如果两个指针类型相同, 那么可以把一个指针赋给另一个指针, 否则必须用类型转换运算符把赋值运算符右边的指针转换为赋值运算符左边的指针。指向void类型的指针是例外情形, 它可以表示任何类型的指针。所有类型的指针都可以赋值给指向void类型的指针, 指向void类型的指针也可以赋给任何类型的指针。
24. 不能复引用void类型的指针。
25. 可以用相等测试运算符和关系运算符比较两个指针, 但是除非他们是指向同一个地址, 否则这种比较是没有意义的。

附录--指针用法小结 (4)

- 26. 不带下标的数组名是一个指向数组的第一个元素的指针。
 - 27. 可以象数组名那样对指针使用下标。
 - 28. 在指针/偏移量表示法中，偏移量是等同于数组下标的。
 - 29. 所有带下标的数组表达式都能够用指针（数组名或者是指向数组的单独的指针）和偏移量表示。
 - 30. 数组名是一个常量指针，它总是指向不变的内存单元。不能象常规的指针那样修改数组名。
 - 31. 指向函数的指针是驻留在内存中的函数代码地址。
 - 32. 指向函数的指针可以传递给函数，也可以从函数返回指向函数的指针，还可以把指向函数的指针存储在数组中，以及把它赋值给其他的函数指针。
-

谢谢！