

2023

# 面向对象程序设计

## C++ 高级主题

李际军    [lijijun@cs.zju.edu.cn](mailto:lijijun@cs.zju.edu.cn)





# 第一部分

强制类型转换运算符





# 1. 强制类型转换运算符

•C++ 引入了四种功能不同的强制类型转换运算符以进行强制类型转换：

`static_cast`、`reinterpret_cast`、`const_cast` 和 `dynamic_cast`。

强制类型转换是有一定风险的，有的转换并不一定安全。C++ 引入新的强制类型转换机制，主要是为了克服 C 语言强制类型转换的以下三个缺点。

1) 没有从形式上体现转换功能和风险的不同。

例如，将 `int` 强制转换成 `double` 是没有风险的，而将常量指针转换成非常量指针，将基类指针转换成派生类指针都是高风险的。

2) 将多态基类指针转换成派生类指针时不检查安全性，即无法判断转换后的指针是否确实指向一个派生类对象。

3) 难以在程序中寻找到底什么地方进行了强制类型转换。

强制类型转换是引发程序运行时错误的一个原因，因此在程序出错时，可能就会想到是不是有哪些强制类型转换出了问题。



# 1. 强制类型转换运算符

- C++ 强制类型转换运算符的用法如下：

强制类型转换运算符 < 要转换到的类型 > ( 待转换的表达式 )

- 例如：

- `double d = static_cast <double> (3*5);` // 将 3\*5 的值转换成实数



## 2. 四种强制类型转换运算符 :static\_cast

- static\_cast 用于进行比较“自然”和低风险的转换，如整型和浮点型、字符型之间的互相转换。另外，如果对象所属的类重载了强制类型转换运算符 T（如 T 是 int、int\* 或其他类型名），则 static\_cast 也能用来进行对象到 T 类型的转换。
- static\_cast 不能用于在不同类型的指针之间互相转换，也不能用于整型和指针之间的互相转换，当然也不能用于不同类型的引用之间的转换。因为这些属于风险比较高的转换。



## 2. 四种强制类型转换运算符 :static\_cast

•static\_cast 用法示例如下：

```
#include <iostream>
using namespace std;
class A
{
public:
    operator int() { return 1; }
    operator char*() { return NULL; }
};
int main()
{
    A a;
    int n;
    char* p = "New Dragon Inn";
    n = static_cast <int> (3.14); // n 的值变为 3
    n = static_cast <int> (a); // 调用 a.operator int , n 的值变为 1
    p = static_cast <char*> (a); // 调用 a.operator char* , p 的值变为 NULL
    n = static_cast <int> (p); // 编译错误 , static_cast 不能将指针转换成整型
    p = static_cast <char*> (n); // 编译错误 , static_cast 不能将整型转换成指针
    return 0;
}
```





## 2. 四种强制类型转换运算

### 符 `:reinterpret_cast`

- `reinterpret_cast` 用于进行各种不同类型的指针之间、不同类型的引用之间以及指针和能容纳指针的整数类型之间的转换。转换时，执行的是逐个比特复制的操作。
- 这种转换提供了很强的灵活性，但转换的安全性只能由程序员的细心来保证了。例如，程序员执意要把一个 `int*` 指针、函数指针或其他类型的指针转换成 `string*` 类型的指针也是可以的，至于以后用转换后的指针调用 `string` 类的成员函数引发错误，程序员也只能自行承担查找错误的烦琐工作：（ C++ 标准不允许将函数指针转换成对象指针，但有些编译器，如 Visual Studio 2010，则支持这种转换 ）。



## 2. 四种强制类型转换运算

### 符 :reinterpret\_cast

#### • reinterpret\_cast 用法示例如下：

```
#include <iostream>
using namespace std;
class A
{
public:
    int i;
    int j;
    A(int n):i(n),j(n) { }
};
int main()
{
    A a(100);
    int &r = reinterpret_cast<int&>(a); // 强行让 r 引用 a
    r = 200; // 把 a.i 变成了 200
    cout << a.i << "," << a.j << endl; // 输出 200,100
    int n = 300;
```





## 2. 四种强制类型转换运算

符: **reinterpret\_cast**

```
A *pa = reinterpret_cast<A*> ( & n); // 强行让 pa 指向 n
```

```
pa->i = 400;    // n 变成 400
```

```
pa->j = 500;    // 此条语句不安全，很可能导致程序崩溃
```

```
cout << n << endl; // 输出 400
```

```
long long la = 0x12345678abcdLL;
```

```
pa = reinterpret_cast<A*>(la); //la 太长，只取低 32 位 0x5678abcd 拷贝给 pa
```

```
unsigned int u = reinterpret_cast<unsigned int>(pa); //pa 逐个比特拷贝到 u
```

```
cout << hex << u << endl; // 输出 5678abcd
```

```
typedef void (* PF1) (int);
```

```
typedef int (* PF2) (int,char *);
```

```
PF1 pf1; PF2 pf2;
```

```
pf2 = reinterpret_cast<PF2>(pf1); // 两个不同类型的函数指针之间可以互相转换
```

```
}
```

程序的输出结果是：

200, 100

400

5678abed



## 2. 四种强制类型转换运算

### 符 :reinterpret\_cast

•第 19 行的代码不安全，因为在编译器看来，pa->j 的存放位置就是 n 后面的 4 个字节。本条语句会向这 4 个字节中写入 500。但这 4 个字节不知道是用来存放什么的，贸然向其中写入可能会导致程序错误甚至崩溃。

上面程序中的各种转换都没有实际意义，只是为了演示 reinterpret\_cast 的用法而已。在编写黑客程序、病毒或反病毒程序时，也许会用到这样怪异的转换。

reinterpret\_cast 体现了 C++ 语言的设计思想：用户可以做任何操作，但要为自己的行为负责。



## 2. 四种强制类型转换运算符 :const\_cast

•const\_cast 运算符仅用于进行去除 const 属性的转换，它也是四个强制类型转换运算符中唯一能够去除 const 属性的运算符。

将 const 引用转换为同类型的非 const 引用，将 const 指针转换为同类型的非 const 指针时可以使用 const\_cast 运算符。例如：

```
const string s = "Inception";
```

```
string& p = const_cast <string&> (s);
```

```
string* ps = const_cast <string*> (&s); // &s 的类型是 const string*
```





## 2. 四种强制类型转换运算符：`dynamic_cast`

- 用 `reinterpret_cast` 可以将多态基类（包含虚函数的基类）的指针强制转换为派生类的指针，但是这种转换不检查安全性，即不检查转换后的指针是否确实指向一个派生类对象。`dynamic_cast` 专门用于将多态基类的指针或引用强制转换为派生类的指针或引用，而且能够检查转换的安全性。对于不安全的指针转换，转换结果返回 `NULL` 指针。
- `dynamic_cast` 是通过“运行时类型检查”来保证安全性的。`dynamic_cast` 不能用于将非多态基类的指针或引用强制转换为派生类的指针或引用——这种转换没法保证安全性，只好用 `reinterpret_cast` 来完成。



## 2. 四种强制类型转换运算符 :dynamic\_cast

### •dynamic\_cast 示例程序如下：

```
#include <iostream>
#include <string>
using namespace std;
class Base
{ // 有虚函数，因此是多态基类
public:
    virtual ~Base() {}
};
class Derived : public Base {};
int main()
{
    Base b;
    Derived d;
    Derived* pd;
    pd = reinterpret_cast <Derived*> (&b);
```

```
    if (pd == NULL)
        // 此处 pd 不会为 NULL。reinterpret_cast 不检查安全性，总是进行转换
        cout << "unsafe reinterpret_cast" << endl; // 不会执行
    pd = dynamic_cast <Derived*> (&b);
    if (pd == NULL)
        // 结果会是 NULL，因为 &b 不指向派生类对象，此转换不安全
        cout << "unsafe dynamic_cast1" << endl; // 会执行
    pd = dynamic_cast <Derived*> (&d); // 安全的转换
    if (pd == NULL) // 此处 pd 不会为 NULL
        cout << "unsafe dynamic_cast2" << endl; // 不会执行
    return 0;
}
```

程序的输出结果是：  
unsafe dynamic\_cast1



## 2. 四种强制类型转换运算符 :dynamic\_cast

- 第 20 行，通过判断 pd 的值是否为 NULL，就能知道第 19 行进行的转换是否是安全的。第 23 行同理。

如果上面的程序中出现了下面的语句：

**Derived & r = dynamic\_cast <Derived &> (b);**

- 那该如何判断该转换是否安全呢？不存在空引用，因此不能通过返回值来判断转换是否安全。C++ 的解决办法是：dynamic\_cast 在进行引用的强制转换时，如果发现转换不安全，就会抛出一个异常，通过处理异常，就能发现不安全的转换。





## 第二部分

异常处理



# 1. 异常处理基本语法



简要说明

C++ 通过 throw 语句和 try...catch 语句实现对异常的处理。throw 语句的语法如下：

**throw 表达式；**

该语句抛出一个异常。异常是一个表达式，其值的类型可以是基本类型，也可以是类。

try...catch 语句的语法如下：

```
try {  
    语句组  
}  
catch( 异常类型 ) {  
    异常处理代码  
}  
...  
catch( 异常类型 ) {  
    异常处理代码  
}
```

catch 可以有多个，但至少要有有一个。

try...catch 语句的执行过程是：

- 执行 try 块中的语句，如果执行的过程中没有异常抛出，那么执行完后就执行最后一个 catch 块后面的语句，所有 catch 块中的语句都不会被执行；
- 如果 try 块执行的过程中抛出了异常，那么抛出异常后立即跳转到第一个“异常类型”和抛出的异常类型匹配的 catch 块中执行（称作异常被该 catch 块“捕获”），执行完后再跳转到最后一个 catch 块后面继续执行。



# 1. 异常处理基本语法



简要说明

```
#include <iostream>
using namespace std;
int main()
{
    double m ,n;
    cin >> m >> n;
    try {
        cout << "before dividing." << endl;
        if( n == 0)
            throw -1; // 抛出 int 类型异常
        else
            cout << m / n << endl;
        cout << "after dividing." << endl;
    }
    catch(double d) {
        cout << "catch(double) " << d << endl;
    }
    catch(int e) {
        cout << "catch(int) " << e << endl;
    }
    cout << "finished" << endl;
    return 0;
}
```

程序的运行结果如下：

```
9 6 ✓
before dividing.
1.5
after dividing.
Finished
```





# 1. 异常处理基本语法



简要说明

- 说明当 `n` 不为 0 时，`try` 块中不会抛出异常。因此程序在 `try` 块正常执行完后，越过所有的 `catch` 块继续执行，`catch` 块一个也不会执行。

程序的运行结果也可能如下：

9 0 ✓

before dividing.

`catch(int)` -1

finished

- 当 `n` 为 0 时，`try` 块中会抛出一个整型异常。抛出异常后，`try` 块立即停止执行。该整型异常会被类型匹配的第一个 `catch` 块捕获，即进入 `catch(int e)` 块执行，该 `catch` 块执行完毕后，程序继续往后执行，直到正常结束。
- 如果抛出的异常没有被 `catch` 块捕获，例如，将 `catch(char e)`，当输入的 `n` 为 0 时，抛出的整型异常就没有 `catch` 块能捕获，这个异常也就得不到处理，那么程序就会立即中止，`try...catch` 后面的内容都不会被执行。



## 2. 能够捕获任何异常的 catch 语句



简要说明

- 如果希望不论抛出哪种类型的异常都能捕获，可以编写如下 catch 块：

```
catch(...) {
```

```
    ...
```

```
}
```

这样的 catch 块能够捕获任何还没有被捕获的异常。



## 2. 能够捕获任何异常的 catch 语句



简要说明

```
#include <iostream>
using namespace std;
int main()
{
    double m, n;
    cin >> m >> n;
    try {
        cout << "before dividing." << endl;
        if (n == 0)
            throw - 1; // 抛出整型异常
        else if (m == 0)
            throw - 1.0; // 抛出 double 型异常
        else
            cout << m / n << endl;
        cout << "after dividing." << endl;
    }
```

```
catch (double d) {
    cout << "catch (double)" << d << endl;
}
catch (...) {
    cout << "catch (...)" << endl;
}
cout << "finished" << endl;
return 0;
}
```

程序的运行结果如下：

```
9 0 ✓
before dividing.
catch (...)
finished
```





## 2. 能够捕获任何异常的 catch 语句



简要说明

- 当 `n` 为 0 时，抛出的整型异常被 `catchy(...)` 捕获。
- 程序的运行结果也可能如下：  
0 6 ✓  
before dividing.  
catch (double) -1  
finished
- 当 `m` 为 0 时，抛出一个 `double` 类型的异常。虽然 `catch (double)` 和 `catch(...)` 都能匹配该异常，但是 `catch(double)` 是第一个能匹配的 `catch` 块，因此会执行它，而不会执行 `catch(...)` 块。
- 由于 `catch(...)` 能匹配任何类型的异常，它后面的 `catch` 块实际上就不起作用，因此不要将它写在其他 `catch` 块前面。



### 3. 异常的再抛出



简要说明

- 如果一个函数在执行过程中抛出的异常在本函数内就被 `catch` 块捕获并处理，那么该异常就不会抛给这个函数的调用者（也称为“上一层的函数”）；如果异常在本函数中没有被处理，则它就会被抛给上一层的函数。例如下面的程序：

```
#include <iostream>
#include <string>
using namespace std;
class CException
{
public:
    string msg;
    CException(string s) : msg(s) {}
};
double Devide(double x, double y)
{
    if (y == 0)
        throw CException("devided by zero");
    cout << "in Devide" << endl;
    return x / y;
}
```



### 3. 异常的再抛出



简要说明

```
int CountTax(int salary)
{
    try {
        if (salary < 0)
            throw - 1;
        cout << "counting tax" << endl;
    }
    catch (int) {
        cout << "salary < 0" << endl;
    }
    cout << "tax counted" << endl;
    return salary * 0.15;
}
```

```
int main()
{
    double f = 1.2;
    try {
        CountTax(-1);
        f = Devide(3, 0);
        cout << "end of try block" << endl;
    }
    catch (CException e) {
        cout << e.msg << endl;
    }
    cout << "f = " << f << endl;
    cout << "finished" << endl;
    return 0;
}
```





### 3. 异常的再抛出



#### 简要说明

- 程序的输出结果如下：  
salary < 0  
tax counted  
devided by zero  
f=1.2  
finished
- CountTa 函数抛出异常后自行处理，这个异常就不会继续被抛给调用者，即 main 函数。因此在 main 函数的 try 块中，CountTax 之后的语句还能正常执行，即会执行 f = Devide(3, 0);
- 第 35 行，Devide 函数抛出了异常却不处理，该异常就会被抛给 Devide 函数的调用者，即 main 函数。抛出此异常后，Devide 函数立即结束，第 14 行不会被执行，函数也不会返回一个值，这从第 35 行 f 的值不会被修改可以看出。
- Devide 函数中抛出的异常被 main 函数中类型匹配的 catch 块捕获。第 38 行中的 e 对象是用复制构造函数初始化的。
- 如果抛出的异常是派生类的对象，而 catch 块的异常类型是基类，那么这两者也能够匹配，因为派生类对象也是基类对象。



### 3. 异常的再抛出



简要说明

```
#include <iostream>
#include <string>
using namespace std;
int CountTax(int salary)
{
    try {
        if( salary < 0 )
            throw string("zero salary");
        cout << "counting tax" << endl;
    }
    catch (string s ) {
        cout << "CountTax error : " << s << endl;
        throw; // 继续抛出捕获的异常
    }
    cout << "tax counted" << endl;
    return salary * 0.15;
}
```

```
int main()
{
    double f = 1.2;
    try {
        CountTax(-1);
        cout << "end of try block" << endl;
    }
    catch(string s) {
        cout << s << endl;
    }
    cout << "finished" << endl;
    return 0;
}
```



### 3. 异常的再抛出



简要说明

- 程序的输出结果如下：  
CountTax error:zero salary  
zero salary  
finished
- 第 14 行的 `throw;` 没有指明抛出什么样的异常，因此抛出的就是 `catch` 块捕获到的异常，即 `string("zero salary")`。这个异常会被 `main` 函数中的 `catch` 块捕获。





## 4. 函数的异常声明列表



### 简要说明

- 为了增强程序的可读性和可维护性，使程序员在使用一个函数时就能看出这个函数可能会抛出哪些异常，C++ 允许在函数声明和定义时，加上它所能抛出的异常的列表，具体写法如下：  
`void func() throw (int, double, A, B, C);`
- 或  
`void func() throw (int, double, A, B, C){...}`
- 上面的写法表明 func 可能抛出 int 型、double 型以及 A、B、C 三种类型的异常。异常声明列表可以在函数声明时写，也可以在函数定义时写。如果两处都写，则两处应一致。
- 如果异常声明列表如下编写：  
`void func() throw ();`
- 则说明 func 函数不会抛出任何异常。
- 一个函数如果不交待能抛出哪些类型的异常，就可以抛出任何类型的异常。
- 函数如果抛出了其异常声明列表中没有的异常，在编译时不会引发错误，但在运行时，Dev C++ 编译出来的程序会出错；用 Visual Studio 2010 编译出来的程序则不会出错，异常声明列表不起实际作用。

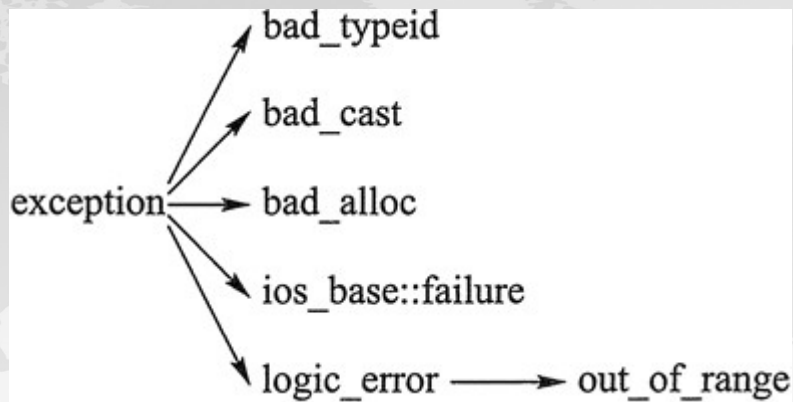


## 5. C++ 标准异常类



简要说明

- C++ 标准库中有一些类代表异常，这些类都是从 `exception` 类派生而来的。常用的几个异常类如图所示。



- `bad_typeid`、`bad_cast`、`bad_alloc`、`ios_base::failure`、`out_of_range` 都是 `exception` 类的派生类。C++ 程序在碰到某些异常时，即使程序中没有写 `throw` 语句，也会自动抛出上述异常类的对象。这些异常类还都有名为 `what` 的成员函数，返回字符串形式的异常描述信息。使用这些异常类需要包含头文件 `stdexcept`。



## 5. C++ 标准异常类



简要说明

下面分别介绍以上几个异常类。本节程序的输出以 Visual Studio 2010 为准，Dev C++ 编译的程序输出有所不同。

### 1) `bad_typeid`

使用 `typeid` 运算符时，如果其操作数是一个多态类的指针，而该指针的值为 `NULL`，则会抛出此异常。

### 2) `bad_cast`

在用 `dynamic_cast` 进行从多态基类对象（或引用）到派生类的引用的强制类型转换时，如果转换是不安全的，则会抛出此异常。





## 5. C++ 标准异常类



### 简要说明

```
#include <iostream>
#include <stdexcept>
using namespace std;
class Base
{
    virtual void func() {}
};
class Derived : public Base
{
public:
    void Print() {}
};
void PrintObj(Base & b)
{
    try {
        Derived & rd = dynamic_cast <Derived &>(b);
        // 此转换若不安全, 会抛出 bad_cast 异常
        rd.Print();
    }
}
```

```
        catch (bad_cast & e) {
            cerr << e.what() << endl;
        }
    }
}
int main()
{
    Base b;
    PrintObj(b);
    return 0;
}
```

程序的输出结果如下：  
Bad dynamic\_cast!

在 PrintObj 函数中，通过 dynamic\_cast 检测 b 是否引用的是一个 Derived 对象，如果是，就调用其 Print 成员函数；如果不是，就抛出异常，不会调用 Derived::Print。



## 5. C++ 标准异常类



简要说明

### 3) `bad_alloc`

在用 `new` 运算符进行动态内存分配时，如果没有足够的内存，则会引发此异常。程序示例如下：

```
#include <iostream>
#include <stdexcept>
using namespace std;
int main()
{
    try {
        char * p = new char[0x7fffffff]; // 无法分配这么多空间，会抛出异常
    }
    catch (bad_alloc & e) {
        cerr << e.what() << endl;
    }
    return 0;
}
```

程序的输出结果如下：

bad allocation

ios\_base::failure

在默认状态下，输入输出流对象不会抛出此异常。如果用流对象的 `exceptions` 成员函数设置了一些标志位，则在出现打开文件出错、读到输入流的文件尾等情况时会抛出此异常。



## 5. C++ 标准异常类



### 简要说明

#### 4) out\_of\_range

用 `vector` 或 `string` 的 `at` 成员函数根据下标访问元素时，如果下标越界，则会抛出此异常。例如：

```
#include <iostream>
#include <stdexcept>
#include <vector>
#include <string>
using namespace std;
int main()
{
    vector<int> v(10);
    try {
        v.at(100) = 100; // 抛出 out_of_range 异常
    }
    catch (out_of_range & e) {
        cerr << e.what() << endl;
    }
    string s = "hello";
    try {
        char c = s.at(100); // 抛出 out_of_range 异常
    }
    catch (out_of_range & e) {
        cerr << e.what() << endl;
    }
    return 0;
}
```

程序的输出结果如下：

```
invalid vector <T> subscript
invalid string position
```

如果将 `v.at(100)` 换成 `v[100]`，将 `s.at(100)` 换成 `s[100]`，程序就不会引发异常（但可能导致程序崩溃）。因为 `at` 成员函数会检测下标越界并抛出异常，而 `operator[]` 则不会。`operator[]` 相比 `at` 的好处就是不用判断下标是否越界，因此执行速度更快。





## 第三部分

shared\_ptr ( 智能指针 )



# 1. shared\_ptr (智能指针)

- C++ 11 模板库的 `<memory>` 头文件中定义的智能指针，即 `shared_ptr` 模板。
- 只要将 `new` 运算符返回的指针 `p` 交给一个 `shared_ptr` 对象“托管”，就不必担心在哪里写 `delete p` 语句——实际上根本不需要编写这条语句，托管 `p` 的 `shared_ptr` 对象在消亡时会自动执行 `delete p`。而且，该 `shared_ptr` 对象能像指针 `p` 一样使用，即假设托管 `p` 的 `shared_ptr` 对象叫作 `ptr`，那么 `*ptr` 就是 `p` 指向的对象。
- 通过 `shared_ptr` 的构造函数，可以让 `shared_ptr` 对象托管一个 `new` 运算符返回的指针，写法如下：  

```
shared_ptr<T> ptr(new T);
```

 // `T` 可以是 `int`、`char`、类等各种类型
- 此后，`ptr` 就可以像 `T*` 类型的指针一样使用，即 `*ptr` 就是用 `new` 动态分配的那个对象。
- 多个 `shared_ptr` 对象可以共同托管一个指针 `p`，当所有曾经托管 `p` 的 `shared_ptr` 对象都解除了对其的托管时，就会执行 `delete p`。

# 1. shared\_ptr (智能指针)

```
#include <iostream>
#include <memory>
using namespace std;
class A
{
public:
    int i;
    A(int n):i(n) { };
    ~A() { cout << i << " " << "destroyed" << endl; }
};
int main()
{
    shared_ptr<A> sp1(new A(2)); //A(2) 由 sp1 托管 ,
    shared_ptr<A> sp2(sp1); //A(2) 同时交由 sp2 托管
    shared_ptr<A> sp3;
    sp3 = sp2; //A(2) 同时交由 sp3 托管
    cout << sp1->i << ", " << sp2->i << ", " << sp3->i << endl;
    A * p = sp3.get(); // get 返回托管的指针 , p 指向 A(2)
    cout << p->i << endl; // 输出 2
    sp1.reset(new A(3)); // reset 导致托管新的指针 , 此时 sp1 托管 A(3)
    sp2.reset(new A(4)); // sp2 托管 A(4)
    cout << sp1->i << endl; // 输出 3
    sp3.reset(new A(5)); // sp3 托管 A(5), A(2) 无人托管 , 被 delete
    cout << "end" << endl;
    return 0;
}
```

程序的输出结果如下 :

2,2,2

2

3

2 destroyed

end

5 destroyed

4 destroyed

3 destroyed





# 1. shared\_ptr (智能指针)

• 可以用第 14 行及第 16 行的形式让多个 `shared_ptr` 对象托管同一个指针。这多个 `shared_ptr` 对象会共享一个对共同托管的指针的“托管计数”。有  $n$  个 `shared_ptr` 对象托管同一个指针  $p$ ，则  $p$  的托管计数就是  $n$ 。当一个指针的托管计数减为 0 时，该指针会被释放。`shared_ptr` 对象消亡或托管了新的指针，都会导致其原托管指针的托管计数减 1。

第 20、21 行，`shared_ptr` 的 `reset` 成员函数可以使得对象解除对原托管指针的托管（如果有的话），并托管新的指针。原指针的托管计数会减 1。

输出的第 4 行说明，用 `new` 创建的动态对象 `A(2)` 被释放了。程序中没有写 `delete` 语句，而 `A(2)` 被释放，是因为程序的第 23 行执行后，已经没有 `shared_ptr` 对象托管 `A(2)`，于是 `A(2)` 的托管计数变为 0。最后一个解除对 `A(2)` 托管的 `shared_ptr` 对象会释放 `A(2)`。

`main` 函数结束时，`sp1`、`sp2`、`sp3` 对象消亡，各自将其托管的指针的托管计数减为 0，并且释放其托管的指针，于是会有以下输出：

5 destructed

4 destructed

3 destructed

只有指向动态分配的对象的指针才能交给 `shared_ptr` 对象托管。将指向普通局部变量、全局变量的指针交给 `shared_ptr` 托管，编译时不会有问題，但程序运行时会出错，因为不能析构一个并没有指向动态分配的内存空间的指针。



# 第四部分

Lambda 表达式 ( 匿名函数 )

## 1. Lambda 表达式 ( 匿名函数 )

- 使用 STL 时，往往会大量用到函数对象，为此要编写很多函数对象类。有的函数对象类只用来定义了一个对象，而且这个对象也只使用了一次，编写这样的函数对象类就有点浪费。而且，定义函数对象类的地方和使用函数对象的地方可能相隔较远，看到函数对象，想要查看其 `operator()` 成员函数到底是做什么的也会比较麻烦。对于只使用一次的函数对象类，能否直接在使用它的地方定义呢？Lambda 表达式能够解决这个问题。使用 Lambda 表达式可以减少程序中函数对象类的数量，使得程序更加优雅。

Lambda 表达式的定义形式如下：

```
[ 外部变量访问方式说明符 ] ( 参数表 ) -> 返回值类型  
{  
    语句块  
}
```

下面是一个合法的 Lambda 表达式：

```
[=] (int x, int y) -> bool {return x%10 < y%10; }
```



## 1. Lambda 表达式 ( 匿名函数 )

- Lambda 表达式实际上是一个函数，只是它没有名字。下面的程序段使用了上面的 Lambda 表达式：

```
int a[4] = {11, 2, 33, 4};  
sort(a, a+4, [=](int x, int y) -> bool { return x%10 < y%10; } );  
for_each(a, a+4, [=](int x) { cout << x << " "; } );
```

- 这段程的输出结果是：

11 2 33 4

程序第 2 行使得数组 a 按个位数从小到大排序。具体的原理是：sort 在执行过程中，需要判断两个元素 x、y 的大小时，会以 x、y 作为参数，调用 Lambda 表达式所代表的函数，并根据返回值来判断 x、y 的大小。这样，就不用专门编写一个函数对象类了。

第 3 行，for\_each 的第 3 个参数是一个 Lambda 表达式。for\_each 执行过程中会依次以每个元素作为参数调用它，因此每个元素都被输出。

# 1. Lambda 表达式 ( 匿名函数 )

- 下面是用到了外部变量的 Lambda 表达式的程序：

```
#include <iostream>
#include <algorithm>
using namespace std;
int main()
{
    int a[4] = { 1, 2, 3, 4 };
    int total = 0;
    for_each(a, a + 4, [&](int & x) { total += x; x *= 2; });
    cout << total << endl; // 输出 10
    for_each(a, a + 4, [=](int x) { cout << x << " "; });
    return 0;
}
```

程序的输出结果如下：

```
10
2 4 6 8
```

第 8 行，[&] 表示该 Lambda 表达式中用到的外部变量 total 是传引用的，其值可以在表达式执行过程中被改变（如果使用 [=]，编译无法通过）。该 Lambda 表达式每次被 for\_each 执行时，都将 a 中的一个元素累加到 total 上，然后将该元素加倍。

# 1. Lambda 表达式 ( 匿名函数 )

```
#include <iostream>
using namespace std;
int main()
{
    int x = 100,y=200,z=300;
    auto ff = [=,&y,&z](int n) {
        cout <<x << endl;
        y++; z++;
        return n*n;
    };
    cout << ff(15) << endl;
    cout << y << "," << z << endl;
}
```

程序的输出结果如下：

```
100
225
201, 301
```

第 6 行定义了一个变量 `ff`，`ff` 的类型是 `auto`，表示由编译器自动判断其类型（这也是 C++11 的新特性）。本行将一个 Lambda 表达式赋值给 `ff`，以后就可以通过 `ff` 来调用该 Lambda 表达式了。

第 11 行通过 `ff`，以 15 作为参数 `n` 调用上面的 Lambda 表达式。该 Lambda 表达式指明，对于外部变量 `y`、`z`，可以修改其值；对于其他外部变量，例如 `x`，不能修改其值。因此在该表达式执行时，可以修改外部变量 `y`、`z` 的值，但如果出现试图修改 `x` 值的语句，就会编译出错。





# 第五部分

auto 和 decltype 关键字

## 1. auto 和 decltype 关键字

- 可以用 auto 关键字定义变量，编译器会自动判断变量的类型。例如：

**auto** i=100; // i 是 int

**auto** p = **new** A(); // p 是 A\*

**auto** k = 34343LL; // k 是 long long

- 有时，变量的类型名特别长，使用 auto 就会很方便。例如：

```
map <string, int, greater <string> >mp;
```

```
for( auto i = mp.begin(); i != mp.end(); ++i)
```

```
cout << i -> first << ", " << i -> second;
```

- 编译器会自动判断出 i 的类型是 map<string, int, greater<string>>::iterator。

## 1. auto 和 decltype 关键字

- decltype 关键字可以用于求表达式的类型。例如：

```
int i;  
double t;  
struct A { double x; };  
const A* a = new A();  
decltype(a) x1; //x1 是 A*  
decltype(i) x2; //x2 是 int  
decltype(a -> x) x3; // x3 是 double
```

编译器自动将 `decltype (a)` 等价于 `A*`，因为编译器知道 `a` 的类型是 `A*`。





# 1. auto 和 decltype 关键字

- auto 和 decltype 可以一起使用。例如：

```
#include <iostream>
using namespace std;
struct A {
    int i;
    A(int ii) : i(ii) {}
};
A operator + (int n, const A & a)
{
    return A(a.i + n);
}
template <class T1, class T2>
auto add(T1 x, T2 y) -> decltype(x + y) {
    return x + y;
}
int main() {
    auto d = add(100, 1.5); // d 是 double 类型，d = 101.5
    auto k = add(100, A(1)); // k 是 A 类型，因为表达式“100+A(1)”是 A 类型的
    cout << d << endl;
    cout << k.i << endl;
    return 0;
}
```

- 程序的输出结果如下：

101.5  
101

第 12 行告诉编译器，add 的返回值类型是 decltype(x+y)，即返回值的类型和 x+y 这个表达式的类型一致。编译器将 add 实例化时，会自动推断出 x+y 的类型。



## 1. auto 和 decltype 关键字

- 在 C++11 中，函数返回值若为 `auto`，则需要和 `decltype` 配合使用。在 `C++14` 中，则可以不用 `decltype`，例如下面的程序没有问题：

```
auto add (int a, int b)
{
    int i = a + b;
    return i;
}
```



# 第六部分

右值引用



## 1. C++11 右值引用详解

- 能出现在赋值号左边的表达式称为“左值”，不能出现在赋值号左边的表达式称为“右值”。一般来说，左值是可以取地址的，右值则不可以。
- 非 `const` 的变量都是左值。函数调用的返回值若不是引用，则该函数调用就是右值。前面所学的“引用”都是引用变量的，而变量是左值，因此它们都是“左值引用”。
- C++11 新增了一种引用，可以引用右值，因而称为“右值引用”。无名的临时变量不能出现在赋值号左边，因而是右值。右值引用就可以引用无名的临时变量。定义右值引用的格式如下：  
类型 && 引用名 = 右值表达式；

- 例如：

```
class A{};
```

`A & r1 = A();` // 错误，无名临时变量 `A()` 是右值，因此不能初始化左值引用 `r1`

`A && r2 = A();` // 正确，因 `r2` 是右值引用



# 1. C++11 右值引用详解

- 引入右值引用的主要目的是提高程序运行的效率。有些对象在复制时需要进行深复制，深复制往往非常耗时。合理使用右值引用可以避免没有必要的深复制操作。例如下面的程序：

```
#include <iostream>
#include <string>
#include <cstring>
using namespace std;
class String
{
public:
char* str;
String() : str(new char[1]) { str[0] = 0; }
String(const char* s) {
str = new char[strlen(s) + 1];
strcpy(str, s);
}
String(const String & s) { // 复制构造函数
cout << "copy constructor called" << endl;
str = new char[strlen(s.str) + 1];
strcpy(str, s.str);
}
```



# 1. C++11 右值引用详解

```
String & operator = (const String & s) { // 复制赋值号
    cout << "copy operator = called" << endl;
    if (str != s.str) {
        delete[] str;
        str = new char[strlen(s.str) + 1];
        strcpy(str, s.str);
    }
    return *this;
}

String(String && s) : str(s.str) { // 移动构造函数
    cout << "move constructor called" << endl;
    s.str = new char[1];
    s.str[0] = 0;
}

String & operator = (String && s) { // 移动赋值号
    cout << "move operator = called" << endl;
    if (str != s.str) {
        str = s.str;
        s.str = new char[1];
        s.str[0] = 0;
    }
    return *this;
}

~String() { delete[] str; }
};
```





# 1. C++11 右值引用详解

```
template <class T>
void MoveSwap(T & a, T & b) {
    T tmp(move(a)); //std::move(a) 为右值，这里会调用移动构造函数
    a = move(b); //move(b) 为右值，因此这里会调用移动赋值号
    b = move(tmp); //move(tmp) 为右值，因此这里会调用移动赋值号
}

int main()
{
    String s;
    s = String("this"); // 调用移动赋值号
    cout << " * * * " << endl;
    cout << s.str << endl;
    String s1 = "hello", s2 = "world";
    MoveSwap(s1, s2); // 调用一次移动构造函数和两次移动赋值号
    cout << s2.str << endl;
    return 0;
}
```



## 1. C++11 右值引用详解

- 程序的输出结果如下：

```
move operator = called
```

```
****
```

```
this
```

```
move constructor called
```

```
move operator = called
```

```
move operator = called
```

```
hello
```

第 33 行重载了一个移动赋值号。它和第 19 行的复制赋值号的区别在于，其参数是右值引用。在移动赋值号函数中没有执行深复制操作，而是直接将对象的 `str` 指向了参数 `s` 的成员变量 `str` 指向的地方，然后修改 `s.str` 让它指向别处，以免 `s.str` 原来指向的空间被释放两次。

该移动赋值号函数修改了参数，这会不会带来麻烦呢？答案是不会。因为移动赋值号函数的形参是一个右值引用，则调用该函数时，实参一定是右值。右值一般是无名临时变量，而无名临时变量在使用它的语句结束后就不再有用，因此其值即使被修改也没有关系。

## 1. C++11 右值引用详解

- 第 53 行，如果没有定义移动赋值号，则会导致复制赋值号被调用，引发深复制操作。临时无名变量 `String("this")` 是右值，因此在定义了移动赋值号的情况下，会导致移动赋值号被调用。移动赋值号使得 `s` 的内容和 `String("this")` 一致，然而却不用执行深复制操作，因而效率比复制赋值号高。

虽然移动赋值号修改了临时变量 `String("this")`，但该变量在后面已无用处，因此这样的修改不会导致错误。

第 46 行使用了 C++ 11 中的标准模板 `move`。`move` 能接受一个左值作为参数，返回该左值的右值引用。因此本行会用定义于第 28 行、以右值引用作为参数的移动构造函数来初始化 `tmp`。该移动构造函数没有执行深复制，将 `tmp` 的内容变成和 `a` 相同，然后修改 `a`。由于调用 `MoveSwap` 本来就会修改 `a`，所以 `a` 的值在此处被修改不会产生问题。

第 47 行和第 48 行调用了移动赋值号，在没有进行深复制的情况下完成了 `a` 和 `b` 内容的互换。对比 `Swap` 函数的以下写法：



## 1. C++11 右值引用详解

```
template <class T>
void Swap(T & a, T & b) {
    T tmp(a); // 调用复制构造函数
    a=b; // 调用复制赋值号
    b=tmp; // 调用复制赋值号
}
```

Swap 函数执行期间会调用一次复制构造函数，两次复制赋值号，即一共会进行三次深复制操作。而利用右值引用，使用 MoveSwap，则可以在无须进行深复制的情况下达到相同的目的，从而提高了程序的运行效率。



# 第七部分

Signal Handling



# 1. C++ Signal Handling

- There are signals which can not be caught by the program but there is a following list of signals which you can catch in your program and can take appropriate actions based on the signal. These signals are defined in C++ header file `<csignal>`.
- The `signal()` Function  
`void (*signal (int sig, void (*func)(int)))(int);`

Keeping it simple, this function receives two arguments: first argument as an integer which represents signal number and second argument as a pointer to the signal-handling function.





# 1. C++ Signal Handling

```
#include <iostream>
#include <csignal>
using namespace std;
void signalHandler( int signum )
{
    cout << "Interrupt signal (" << signum << ") received.\n";

    // cleanup and close up stuff here
    // terminate program
    exit(signum);
}
int main ()
{
    // register signal SIGINT and signal handler
    signal(SIGINT, signalHandler);
    while(1)
    {
        cout << "Going to sleep...." << endl;
        sleep(1);
    }
    return 0;
}
```



## 1. C++ Signal Handling

- When the above code is compiled and executed, it produces the following result –

Going to sleep....

Going to sleep....

Going to sleep....

Interrupt signal (2) received.



## 1. C++ Signal Handling

- The `raise()` Function

You can generate signals by function **`raise()`**, which takes an integer signal number as an argument and has the following syntax.

```
int raise (signal sig);
```

- Here, **`sig`** is the signal number to send any of the signals: `SIGINT`, `SIGABRT`, `SIGFPE`, `SIGILL`, `SIGSEGV`, `SIGTERM`, `SIGHUP`. Following is the example where we raise a signal internally using `raise()` function as follows –



# 1. C++ Signal Handling

```
#include <iostream>
#include <csignal>
using namespace std;
void signalHandler( int signum )
{
    cout << "Interrupt signal (" << signum << ") received.\n";

    // cleanup and close up stuff here
    // terminate program
    exit(signum);
}
int main ()
{
    int i = 0;
    // register signal SIGINT and signal handler
    signal(SIGINT, signalHandler);
    while(++i)
    {
        cout << "Going to sleep...." << endl;
        if( i == 3 )
        {
            raise( SIGINT);
        }
        sleep(1);
    }
    return 0;
}
```

When the above code is compiled and executed, it produces the following result and would come out automatically –

```
Going to sleep....
Going to sleep....
Going to sleep....
Interrupt signal (2) received.
```





# 第八部分

C++ Multithreading



# 1. C++ Multithreading

- Multithreading is a specialized form of multitasking and a multitasking is the feature that allows your computer to run two or more programs concurrently. In general, there are two types of multitasking: process-based and thread-based .
- A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.
- C++ does not contain any built-in support for multithreaded applications. Instead, it relies entirely upon the operating system to provide this feature.
- Creating Threads

```
#include <pthread.h>
```

```
pthread_create (thread, attr, start_routine, arg)
```

Here, **pthread\_create** creates a new thread and makes it executable.



# 1. C++ Multithreading

- Terminating Threads

```
#include <pthread.h>  
pthread_exit (status)
```

Here **pthread\_exit** is used to explicitly exit a thread. Typically, the `pthread_exit()` routine is called after a thread has completed its work and is no longer required to exist.

If `main()` finishes before the threads it has created, and exits with `pthread_exit()`, the other threads will continue to execute. Otherwise, they will be automatically terminated when `main()` finishes.





# 1. C++ Multithreading

- This simple example code creates 5 threads with the `pthread_create()` routine. Each thread prints a "Hello World!" message, and then terminates with a call to `pthread_exit()`.

```
#include <iostream>
#include <cstdlib>
#include <pthread.h>
using namespace std;
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
    long tid;
    tid = (long)threadid;
    cout << "Hello World! Thread ID, " << tid << endl;
    pthread_exit(NULL);
}

int main ()
{
    pthread_t threads[NUM_THREADS];
    int rc;
    int i;
    for( i = 0; i < NUM_THREADS; i++ )
    {
        cout << "main() : creating thread, " << i << endl;
        rc = pthread_create(&threads[i], NULL, PrintHello, (void *)i);

        if (rc)
        {
            cout << "Error:unable to create thread," << rc << endl;

            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

Compile the following program using -  
lpthread library as follows -

```
$gcc test.cpp -lpthread
```

Now, execute your program which gives the  
following output -

```
main() : creating thread, 0
main() : creating thread, 1
main() : creating thread, 2
main() : creating thread, 3
main() : creating thread, 4
Hello World! Thread ID, 0
Hello World! Thread ID, 1
Hello World! Thread ID, 2
Hello World! Thread ID, 3
Hello World! Thread ID, 4
```



## 2. Passing Arguments to Threads

- his example shows how to pass multiple arguments via a structure. You can pass any data type in a thread callback because it points to void as explained in the following example –

```
#include <iostream>
#include <cstdlib>
#include <pthread.h>
using namespace std;

#define NUM_THREADS 5

struct thread_data
{
    int thread_id;
    char *message;
};
```

```
void *PrintHello(void *threadarg)
{
    struct thread_data *my_data;
    my_data = (struct thread_data *) threadarg;
    cout << "Thread ID : " << my_data->thread_id ;
    cout << " Message : " << my_data->message << endl;
    pthread_exit(NULL);
}

int main ()
{
    pthread_t threads[NUM_THREADS];
    struct thread_data td[NUM_THREADS];
    int rc;
    int i;
    for( i = 0; i < NUM_THREADS; i++ )
    {
        cout <<"main() : creating thread, " << i << endl;
        td[i].thread_id = i;
        td[i].message = "This is message";
        rc = pthread_create(&threads[i], NULL, PrintHello, (void *)&td[i]);
        if (rc)
        {
            cout << "Error:unable to create thread," << rc << endl;
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```



## 2. Passing Arguments to Threads

- When the above code is compiled and executed, it produces the following result –

```
main() : creating thread, 0
main() : creating thread, 1
main() : creating thread, 2
main() : creating thread, 3
main() : creating thread, 4
Thread ID : 3 Message : This is message
Thread ID : 2 Message : This is message
Thread ID : 0 Message : This is message
Thread ID : 1 Message : This is message
Thread ID : 4 Message : This is message
```



### 3. Joining and Detaching Threads

- There are following two routines which we can use to join or detach threads –

`pthread_join (threadid, status)`

`pthread_detach (threadid)`

- The `pthread_join()` subroutine blocks the calling thread until the specified 'threadid' thread terminates. When a thread is created, one of its attributes defines whether it is joinable or detached. Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.





### 3. Joining and Detaching Threads

```
#include <iostream>
#include <cstdlib>
#include <pthread.h>
#include <unistd.h>
using namespace std;
#define NUM_THREADS 5
void *wait(void *t)
{
    int i;
    long tid;
    tid = (long)t;
    sleep(1);
    cout << "Sleeping in thread " << endl;
    cout << "Thread with id : " << tid << " ...exiting " << endl;
    pthread_exit(NULL);
}
int main ()
{
    int rc;
    int i;
    pthread_t threads[NUM_THREADS];
    pthread_attr_t attr;
    void *status;
    // Initialize and set thread joinable
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
```



### 3. Joining and Detaching Threads

```
for( i = 0; i < NUM_THREADS; i++ )
{
    cout << "main() : creating thread, " << i << endl;
    rc = pthread_create(&threads[i], &attr, wait, (void *)i );
    if (rc)
    {
        cout << "Error:unable to create thread," << rc << endl;
        exit(-1);
    }
}

// free attribute and wait for the other threads
pthread_attr_destroy(&attr);
for( i = 0; i < NUM_THREADS; i++ )
{
    rc = pthread_join(threads[i], &status);
    if (rc)
    {
        cout << "Error:unable to join," << rc << endl;
        exit(-1);
    }
    cout << "Main: completed thread id :" << i ;
    cout << "   exiting with status :" << status << endl;
}
cout << "Main: program exiting." << endl;
pthread_exit(NULL);
}
```



### 3. Joining and Detaching Threads

When the above code is compiled and executed, it produces the following result –

```
main() : creating thread, 0
main() : creating thread, 1
main() : creating thread, 2
main() : creating thread, 3
main() : creating thread, 4
Sleeping in thread
Thread with id : 0 .... Exiting
Sleeping in thread
Thread with id : 1 .... Exiting
Sleeping in thread
Thread with id : 2 .... Exiting
Sleeping in thread
Thread with id : 3 .... Exiting
Sleeping in thread
Thread with id : 4 .... Exiting
Main: completed thread id :0 exiting with status :0
Main: completed thread id :1 exiting with status :0
Main: completed thread id :2 exiting with status :0
Main: completed thread id :3 exiting with status :0
Main: completed thread id :4 exiting with status :0
Main: program exiting.
```



# 第九部分

Web Programming





## 1. What is CGI

- The Common Gateway Interface, or CGI, is a set of standards that define how information is exchanged between the web server and a custom script.
- The CGI specs are currently maintained by the NCSA and NCSA defines CGI is as follows –
- The Common Gateway Interface, or CGI, is a standard for external gateway programs to interface with information servers such as HTTP servers.
- The current version is CGI/1.1 and CGI/1.2 is under progress.

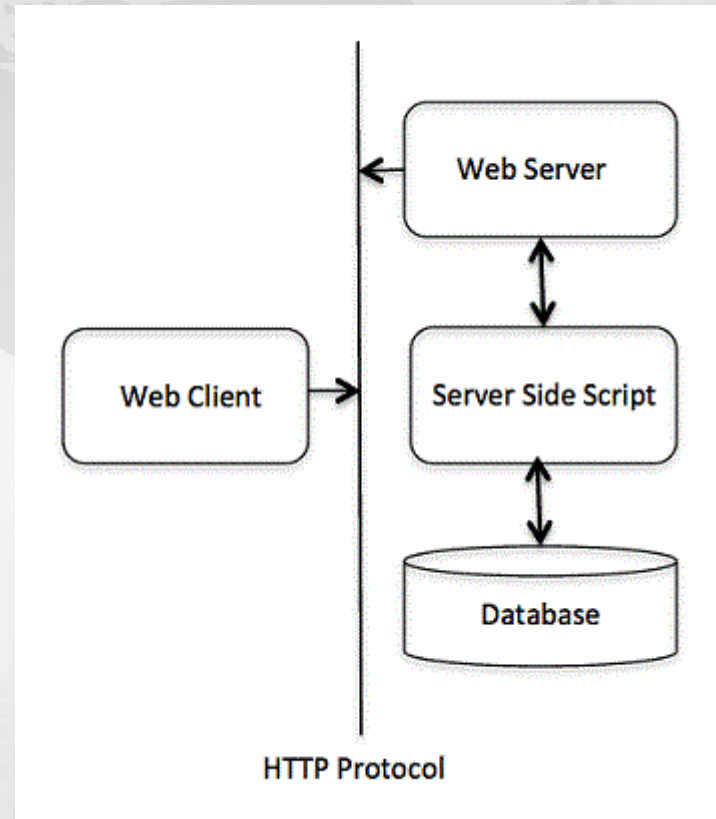


## 2. Web Browsing

- To understand the concept of CGI, let's see what happens when we click a hyperlink to browse a particular web page or URL.
  - Your browser contacts the HTTP web server and demand for the URL ie. filename.
  - Web Server will parse the URL and will look for the filename. If it finds requested file then web server sends that file back to the browser otherwise sends an error message indicating that you have requested a wrong file.
  - Web browser takes response from web server and displays either the received file or error message based on the received response.
- However, it is possible to set up the HTTP server in such a way that whenever a file in a certain directory is requested, that file is not sent back; instead it is executed as a program, and produced output from the program is sent back to your browser to display.
- The Common Gateway Interface (CGI) is a standard protocol for enabling applications (called CGI programs or CGI scripts) to interact with Web servers and with clients. These CGI programs can be a written in Python, PERL, Shell, C or C++ etc.

### 3. CGI Architecture Diagram

- The following simple program shows a simple architecture of CGI –



## 4. Web Server Configuration

- Before you proceed with CGI Programming, make sure that your Web Server supports CGI and it is configured to handle CGI Programs. All the CGI Programs to be executed by the HTTP server are kept in a pre-configured directory. This directory is called CGI directory and by convention it is named as `/var/www/cgi-bin`. By convention CGI files will have extension as **.cgi**, though they are C++ executable.
- By default, Apache Web Server is configured to run CGI programs in `/var/www/cgi-bin`. If you want to specify any other directory to run your CGI scripts, you can modify the following section in the `httpd.conf` file –





## 4. Web Server Configuration

```
Directory "/var/www/cgi-bin">
```

```
    AllowOverride None
```

```
    Options ExecCGI
```

```
    Order allow,deny
```

```
    Allow from all
```

```
</Directory>
```

```
<Directory "/var/www/cgi-bin">
```

```
    Options All
```

```
</Directory>
```

Here, I assume that you have Web Server up and running successfully and you are able to run any other CGI program like Perl or Shell etc.



## 5. First CGI Program

```
#include <iostream>
using namespace std;
int main ()
{
    cout << "Content-type:text/html\r\n\r\n";
    cout << "<html>\n";
    cout << "<head>\n";
    cout << "<title>Hello World - First CGI Program</title>\n";
    cout << "</head>\n";
    cout << "<body>\n";
    cout << "<h2>Hello World! This is my first CGI program</h2>\n";
    cout << "</body>\n";
    cout << "</html>\n";
    return 0;
}
```

- Compile above code and name the executable as `cplusplus.cgi`. This file is being kept in `/var/www/cgi-bin` directory and it has following content. Before running your CGI program make sure you have change mode of file using **`chmod 755 cplusplus.cgi`** UNIX command to make file executable.



## 5. First CGI Program

- Here is small CGI program to list out all the CGI variables.

```
#include <iostream>
#include <stdlib.h>
using namespace std;
const string ENV[ 24 ] =
{
    "COMSPEC", "DOCUMENT_ROOT", "GATEWAY_INTERFACE",
    "HTTP_ACCEPT", "HTTP_ACCEPT_ENCODING",
    "HTTP_ACCEPT_LANGUAGE", "HTTP_CONNECTION",
    "HTTP_HOST", "HTTP_USER_AGENT", "PATH",
    "QUERY_STRING", "REMOTE_ADDR", "REMOTE_PORT",
    "REQUEST_METHOD", "REQUEST_URI", "SCRIPT_FILENAME",
    "SCRIPT_NAME", "SERVER_ADDR", "SERVER_ADMIN",
    "SERVER_NAME", "SERVER_PORT", "SERVER_PROTOCOL",
    "SERVER_SIGNATURE", "SERVER_SOFTWARE" };
```



## 5. First CGI Program

```
int main ()
{
    cout << "Content-type:text/html\r\n\r\n";
    cout << "<html>\n";
    cout << "<head>\n";
    cout << "<title>CGI Environment Variables</title>\n";
    cout << "</head>\n";
    cout << "<body>\n";
    cout << "<table border = \"0\" cellspacing = \"2\">";
    for ( int i = 0; i < 24; i++ )
    {
        cout << "<tr><td>" << ENV[ i ] << "</td><td>";
        // attempt to retrieve value of environment variable
        char *value = getenv( ENV[ i ].c_str() );
        if ( value != 0 )
        {
            cout << value;
        }
        else
        {
            cout << "Environment variable does not exist.";
        }
        cout << "</td></tr>\n";
    }
    cout << "</table><\n";
    cout << "</body>\n";
    cout << "</html>\n";
    return 0;
}
```





## 6. C++ CGI Library

- For real examples, you would need to do many operations by your CGI program. There is a CGI library written for C++ program which you can download from <ftp://ftp.gnu.org/gnu/cgicc/> and follow the steps to install the library –

```
$tar xzf cgicc-X.X.X.tar.gz  
$cd cgicc-X.X.X/  
$./configure --prefix=/usr  
$make  
$make install
```

## 7. First CGI Program

- GET and POST Methods
- You must have come across many situations when you need to pass some information from your browser to web server and ultimately to your CGI Program. Most frequently browser uses two methods to pass this information to web server. These methods are GET Method and POST Method.
- Passing Information Using GET Method
- The GET method sends the encoded user information appended to the page request. The page and the encoded information are separated by the ? character as follows –  
<http://www.test.com/cgi-bin/cpp.cgi?key1=value1&key2=value2>
- Simple URL Example: Get Method  
[/cgi-bin/cpp\\_get.cgi?first\\_name=ZARA&last\\_name=ALI](/cgi-bin/cpp_get.cgi?first_name=ZARA&last_name=ALI)



## 7. First CGI Program

- Below is a program to generate **cpp\_get.cgi** CGI program to handle input given by web browser. We are going to use C++ CGI library which makes it very easy to access passed information –

```
#include <iostream>
#include <vector>
#include <string>
#include <stdio.h>
#include <stdlib.h>
#include <cgicc/CgiDefs.h>
#include <cgicc/Cgicc.h>
#include <cgicc/HTTPHTMLHeader.h>
#include <cgicc/HTMLClasses.h>

using namespace std;
using namespace cgicc;
```



## 7. First CGI Program

```
int main ()
{
    Cgicc formData;
    cout << "Content-type:text/html\r\n\r\n";
    cout << "<html>\n";
    cout << "<head>\n";
    cout << "<title>Using GET and POST Methods</title>\n";
    cout << "</head>\n";
    cout << "<body>\n";
    form_iterator fi = formData.getElement("first_name");
    if( !fi->isEmpty() && fi != (*formData).end())
    {
        cout << "First name: " << **fi << endl;
    }
    else
    {
        cout << "No text entered for first name" << endl;
    }
    cout << "<br/>\n";
    fi = formData.getElement("last_name");
    if( !fi->isEmpty() && fi != (*formData).end())
    {
        cout << "Last name: " << **fi << endl;
    }
    else
    {
        cout << "No text entered for last name" << endl;
    }
    cout << "<br/>\n";
    cout << "</body>\n";
    cout << "</html>\n";
    return 0;
}
```



## 7. First CGI Program

- Now, compile the above program as follows –

```
$g++ -o cpp_get.cgi cpp_get.cpp -lcgicc
```

- Generate cpp\_get.cgi and put it in your CGI directory and try to access using following link –

[/cgi-bin/cpp\\_get.cgi?first\\_name=ZARA&last\\_name=ALI](/cgi-bin/cpp_get.cgi?first_name=ZARA&last_name=ALI)

- This would generate following result –

First name: ZARA

Last name: ALI

## 7. Simple FORM Example: GET Method

Here is a simple example which passes two values using HTML FORM and submit button. We are going to use same CGI script `cpp_get.cgi` to handle this input.

```
<form action = "/cgi-bin/cpp_get.cgi" method = "get">  
First Name: <input type = "text" name = "first_name"> <br />  
Last Name: <input type = "text" name = "last_name" />  
<input type = "submit" value = "Submit" />  
</form>
```

Here is the actual output of the above form. You enter First and Last Name and then click submit button to see the result.

First Name:

Last Name:

Submit

## 8. Passing Information Using POST Method

A generally more reliable method of passing information to a CGI program is the POST method. This packages the information in exactly the same way as GET methods, but instead of sending it as a text string after a ? in the URL it sends it as a separate message. This message comes into the CGI script in the form of the standard input. The same `cpp_get.cgi` program will handle POST method as well. Let us take same example as above, which passes two values using HTML FORM and submit button but this time with POST method as follows –

```
<form action = "/cgi-bin/cpp_get.cgi" method = "post">  
First Name: <input type = "text" name = "first_name"><br />  
Last Name: <input type = "text" name = "last_name" />  
<input type = "submit" value = "Submit" />  
</form>
```

Here is the actual output of the above form. You enter First and Last Name and then click submit button to see the result.

First Name:

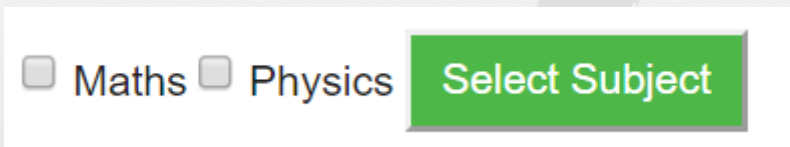
Last Name:

## 9. Passing Checkbox Data to CGI Program

Checkboxes are used when more than one option is required to be selected. Here is example HTML code for a form with two checkboxes –

```
<form action = "/cgi-bin/cpp_checkbox.cgi" method = "POST" target = "_blank">
<input type = "checkbox" name = "maths" value = "on" /> Maths
<input type = "checkbox" name = "physics" value = "on" /> Physics
<input type = "submit" value = "Select Subject" />
</form>
```

The result of this code is the following form –



☐ Maths ☐ Physics





# 例子

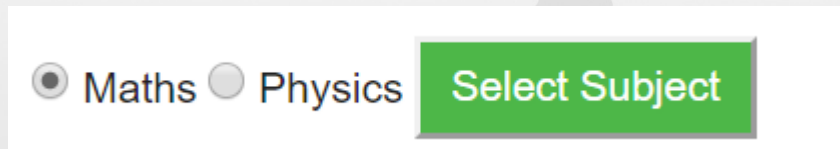
```
#include <iostream>
#include <vector>
#include <string>
#include <stdio.h>
#include <stdlib.h>
#include <cgicc/CgiDefs.h>
#include <cgicc/Cgicc.h>
#include <cgicc/HTTPHTMLHeader.h>
#include <cgicc/HTMLClasses.h>
using namespace std;
using namespace cgicc;
int main () {
    Cgicc formData;
    bool maths_flag, physics_flag;
    cout << "Content-type:text/html\r\n\r\n";
    cout << "<html>\n";
    cout << "<head>\n";
    cout << "<title>Checkbox Data to CGI</title>\n";
    cout << "</head>\n";
    cout << "<body>\n";
    maths_flag = formData.queryCheckbox("maths");
    if( maths_flag ) {
        cout << "Maths Flag: ON " << endl;    }
    else {
        cout << "Maths Flag: OFF " << endl;    }
    cout << "<br/>\n";
    physics_flag = formData.queryCheckbox("physics");
    if( physics_flag ) {
        cout << "Physics Flag: ON " << endl;    }
    else {
        cout << "Physics Flag: OFF " << endl;    }
    cout << "<br/>\n";
    cout << "</body>\n";
    cout << "</html>\n";
    return 0;
}
```

## 10. Passing Radio Button Data to CGI Program

Radio Buttons are used when only one option is required to be selected.  
Here is example HTML code for a form with two radio button –

```
<form action = "/cgi-bin/cpp_radiobutton.cgi" method = "post" target = "_blank">  
<input type = "radio" name = "subject" value = "maths" checked = "checked"/> Maths  
  
<input type = "radio" name = "subject" value = "physics" /> Physics  
<input type = "submit" value = "Select Subject" />  
</form>
```

The result of this code is the following form –



☒ Maths ☐ Physics



# 例子

```
#include <iostream>
#include <vector>
#include <string>
#include <stdio.h>
#include <stdlib.h>
#include <cgicc/CgiDefs.h>
#include <cgicc/Cgicc.h>
#include <cgicc/HTTPHTMLHeader.h>
#include <cgicc/HTMLClasses.h>
using namespace std;
using namespace cgicc;
int main () {
    Cgicc formData;
    cout << "Content-type:text/html\r\n\r\n";
    cout << "<html>\n";
    cout << "<head>\n";
    cout << "<title>Radio Button Data to CGI</title>\n";
    cout << "</head>\n";
    cout << "<body>\n";
    form_iterator fi = formData.getElement("subject");
    if( !fi->isEmpty() && fi != (*formData).end()) {
        cout << "Radio box selected: " << **fi << endl;
    }
    cout << "<br/>\n";
    cout << "</body>\n";
    cout << "</html>\n";
    return 0;
}
```

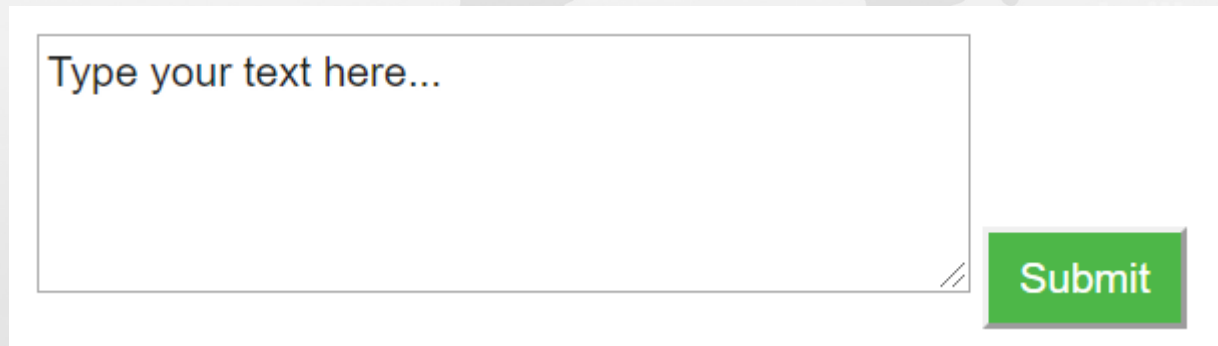
## 11. Passing Text Area Data to CGI Program

TEXTAREA element is used when multiline text has to be passed to the CGI Program.

Here is example HTML code for a form with a TEXTAREA box –

```
<form action = "/cgi-bin/cpp_textarea.cgi" method = "post" target = "_blank">  
<textarea name = "textcontent" cols = "40" rows = "4">  
Type your text here...  
</textarea>  
<input type = "submit" value = "Submit" />  
</form>
```

The result of this code is the following form –



The image shows a rendered HTML form. On the left is a text area with a light gray border and a small diagonal icon in the bottom right corner. Inside the text area, the text "Type your text here..." is displayed in a light gray font. To the right of the text area is a green rectangular button with the word "Submit" in white text.





# 例子

```
#include <iostream>
#include <vector>
#include <string>
#include <stdio.h>
#include <stdlib.h>
#include <cgicc/CgiDefs.h>
#include <cgicc/Cgicc.h>
#include <cgicc/HTTPHTMLHeader.h>
#include <cgicc/HTMLClasses.h>
using namespace std;
using namespace cgicc;
int main () {
    Cgicc formData;
    cout << "Content-type:text/html\r\n\r\n";
    cout << "<html>\n";
    cout << "<head>\n";
    cout << "<title>Text Area Data to CGI</title>\n";
    cout << "</head>\n";
    cout << "<body>\n";
    form_iterator fi = formData.getElement("textcontent");
    if( !fi->isEmpty() && fi != (*formData).end()) {
        cout << "Text Content: " << **fi << endl;
    }
    else {
        cout << "No text entered" << endl;
    }
    cout << "<br/>\n";
    cout << "</body>\n";
    cout << "</html>\n";
    return 0;
}
```

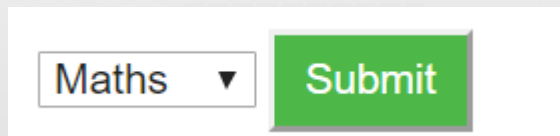
## 12. Passing Drop down Box Data to CGI Program

Drop down Box is used when we have many options available but only one or two will be selected.

Here is example HTML code for a form with one drop down box –

```
<form action = "/cgi-bin/cpp_dropdown.cgi" method = "post" target = "_blank">
<select name = "dropdown">
<option value = "Maths" selected>Maths</option>
<option value = "Physics">Physics</option>
</select>
<input type = "submit" value = "Submit"/>
</form>
```

The result of this code is the following form –



The screenshot shows a web form with a dropdown menu and a submit button. The dropdown menu is labeled 'Maths' with a downward arrow, indicating it is the selected option. To the right of the dropdown is a green button with the text 'Submit' in white.



# 例子

```
#include <iostream>
#include <vector>
#include <string>
#include <stdio.h>
#include <stdlib.h>
#include <cgicc/CgiDefs.h>
#include <cgicc/Cgicc.h>
#include <cgicc/HTTPHTMLHeader.h>
#include <cgicc/HTMLClasses.h>
using namespace std;
using namespace cgicc;
int main () {
    Cgicc formData;
    cout << "Content-type:text/html\r\n\r\n";
    cout << "<html>\n";
    cout << "<head>\n";
    cout << "<title>Drop Down Box Data to CGI</title>\n";
    cout << "</head>\n";
    cout << "<body>\n";
    form_iterator fi = formData.getElement("dropdown");
    if( !fi->isEmpty() && fi != (*formData).end()) {
        cout << "Value Selected: " << **fi << endl;
        cout << "<br/>\n";
        cout << "</body>\n";
        cout << "</html>\n";
        return 0;
    }
}
```



## 12. Setting up Cookies

It is very easy to send cookies to browser. These cookies will be sent along with HTTP Header before the Content-type filed. Assuming you want to set UserID and Password as cookies. So cookies setting will be done as follows

```
#include <iostream>
using namespace std;
int main () {
cout << "Set-Cookie:UserID = XYZ;\r\n";
cout << "Set-Cookie:Password = XYZ123;\r\n";
cout << "Set-Cookie:Domain = www.tutorialspoint.com;\r\n";
cout << "Set-Cookie:Path = /perl;\n";
cout << "Content-type:text/html\r\n\r\n";
cout << "<html>\n";
cout << "<head>\n";
cout << "<title>Cookies in CGI</title>\n";
cout << "</head>\n";
cout << "<body>\n";
cout << "Setting cookies" << endl;
cout << "<br/>\n";
cout << "</body>\n";
cout << "</html>\n";
return 0;
}
```





## 13. Retrieving Cookies

It is easy to retrieve all the set cookies. Cookies are stored in CGI environment variable HTTP\_COOKIE and they will have following form.

key1 = value1; key2 = value2; key3 = value3.....

```
#include <iostream>
#include <vector>
#include <string>
#include <stdio.h>
#include <stdlib.h>
#include <cgicc/CgiDefs.h>
#include <cgicc/Cgicc.h>
#include <cgicc/HTTPHTMLHeader.h>
#include <cgicc/HTMLClasses.h>
using namespace std;
using namespace cgicc;
int main () {
    Cgicc cgi;
    const_cookie_iterator cci;
    cout << "Content-type:text/html\r\n\r\n";
    cout << "<html>\n";
    cout << "<head>\n";
    cout << "<title>Cookies in CGI</title>\n";
    cout << "</head>\n";
    cout << "<body>\n";
    cout << "<table border = \"0\" cellpadding = \"2\">";           // get environment variables
    const CgiEnvironment& env = cgi.getEnvironment();
    for( cci = env.getCookieList().begin(); cci != env.getCookieList().end(); ++cci ) {
        cout << "<tr><td>" << cci->getName() << "</td><td>";
        cout << cci->getValue();
        cout << "</td></tr>\n";    }
    cout << "</table>\n";
    cout << "<br/>\n";
    cout << "</body>\n";
    cout << "</html>\n";
    return 0;
}
```

Now, compile above program to produce getcookies.cgi, and try to get a list of all the cookies available at your computer –

/cgi-bin/getcookies.cgi

This will produce a list of all the four cookies set in previous section and all other cookies set in your computer –

UserID XYZ Password XYZ123 Domain  
www.tutorialspoint.com Path /perl



## 14. File Upload Example

To upload a file the HTML form must have the enctype attribute set to **multipart/form-data**. The input tag with the file type will create a "Browse" button.

```
<html>
<body>
  <form enctype = "multipart/form-data" action = "/cgi-bin/cpp_uploadfile.cgi"
    method = "post">
    <p>File: <input type = "file" name = "userfile" /></p>
    <p><input type = "submit" value = "Upload" /></p>
  </form>
</body>
</html>
```

The result of this code is the following form –

File:  未选择任何文件



# 代码

```
#include <iostream>
#include <vector>
#include <string>
#include <stdio.h>
#include <stdlib.h>
#include <cgicc/CgiDefs.h>
#include <cgicc/Cgicc.h>
#include <cgicc/HTTPHTMLHeader.h>
#include <cgicc/HTMLClasses.h>
using namespace std;
using namespace cgicc;
int main () {
    Cgicc cgi;
    cout << "Content-type:text/html\r\n\r\n";
    cout << "<html>\n";
    cout << "<head>\n";
    cout << "<title>File Upload in CGI</title>\n";
    cout << "</head>\n";
    cout << "<body>\n";    // get list of files to be uploaded
    const_file_iterator file = cgi.getFile("userfile");
    if(file != cgi.GetFiles().end()) {    // send data type at cout.
        cout << HTTPContentHeader(file->getDataType());    // write content at cout.
        file->writeToStream(cout);    }
    cout << "<File uploaded successfully>\n";
    cout << "</body>\n";
    cout << "</html>\n";
    return 0;
}
```





THANKS !