表达式和异常处理

动态内存分配

new delete

异常处理

try/catch

动态内存分配：

在C中我们这样分配内存：

```
int *p = (int*)malloc(100 * sizeof(int));
```

malloc 主要做了两件事情：

1.向系统申请分配内存空间

2.返回该内存空间的首地址

这里注意几点：

1.sizeof（int）申请任何类型变量都采用 sizeof，这是一种跨平台的做法。

2.malloc 返回的是 void*类型的指针，所以这里需要进行强制类型转换。

申请的内存在使用完毕后要进行释放：

```
free(p);
```

free 函数就做了一件事情：

释放 p 指向的内存。

表达式和异常处理 郭春阳

前面说了 malloc 的一些注意点，使用 sizeof 和对结果进行强制类型转换，这也是它的缺点。

C++引入了新的内存分配运算符：new 和 delete

new 的使用方法有三种：

```
int *p1 = new int;
int *p2 = new int(99);
int *p3 = new int[22];
```

下面我们来逐个解释：

1. int *p1 = new int;

申请一个 int 空间，值为默认值

如果这样写 string *ps = new string; 那么就是申请了一个 string，值为空。

2. int *p2 = new int(99);

申请一个 int 空间，值为指定的 99。

string *ps = new string("Beijing");就是动态申请了一个值为 Beijing 的 string。

3. int *p3 = new int[22];

申请一个长度为 22 的 int 数组，值为默认值通常为 0，具体与实现平台相关。

string *ps = new string[10];

ps 指向一个长度为 10 的 string 数组。

表达式和异常处理 郭春阳

new 运算符做了如下的工作：

1.申请内存

2.执行某些类型的构造函数（例如 string）

3.返回这段内存的首地址

那么 new 和 malloc 的区别在哪里？

主要是两点：

1.new 是运算符，malloc 是函数

2.new 会执行某些类型的构造函数，而 malloc 仅仅申请内存。

所以对于 string 类型，只可以使用 new，绝对不可以使用 malloc!!

对应 delete 操作符的使用方法如下：

1.delete ps;

释放 ps 指向的单个变量

所以上面的 p1 和 p2 都应该使用 delete p 的形式

2.delete[] ps;

删除 ps 所指向的数组。所以上面的 ps3 应该用 delete[] 释放

这里我们解释下，**为什么 new 和 delete 的操作要配合使用**，在 C 中我们调用 malloc 申请内存的时候，我们得到一个内存块，在该内存块的**前面**有一处小的区域记录着**这块内存的大小**，所以我们 free 的时

候，系统知道该回收多少内存。同样的道理，我们调用 delete[]，系统也是先去读取前面的区域，获取究竟应该释放多少内存，如果调用 delete，就省略了这一步骤，因为 delete 只需要释放一个单位。这也是 **delete 和 delete[]的本质区别**!

所以，应该使用后者的场合使用了 delete，就造成了内存泄漏。

动态二维数组的申请和释放：

```
int **p = new int*[5];
for (int i = 0; i != 5; ++i) {
    p[i] = new int[4];
}
```

这个数组的特点是：子数组之间连续，数组之间不是连续的。

这个数组仍然支持下标操作，可以自行尝试。

数组的释放过程如下：

```
for (int i = 0; i != 5; ++i) {
    delete[] p[i];
}
delete[] p;
```

注意：

new 和 delete 的使用必须要配对。

前面采用的是 new int[]，后面必须用 delete[]

new delete malloc 和 free 不要混用

C 语言中的错误处理：

在 C 语言中，我们处理错误最常用的方式是去获取函数调用的**返回值**。在 linux 中，一般用-1 表示函数调用失败，同时 linux 提供一个**全局变量 errno（多线程里面 errno 是线程私有的一个全局变量）**，来记录**错误码**。返回值为 0 或者正数通常表示调用成功。

例如：

```
int fd = socket(...);
if(fd == -1){
    perror("socket:");
    exit(EXIT_FAILURE);
}
```

我们可以**写一个宏**：

```
#define ERR_EXIT(m) \
    do\
    {\
        perror(m);\
        exit(EXIT_FAILURE);\
    }while(0)
```

然后我们就可以这样使用：

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
if (fd < 0) {
    ERR_EXIT("socket");
}
```

当然也可以不定义宏，但是定义宏确实减少了代码函数。

异常处理：

表达式和异常处理 郭春阳

看一段代码：

```
int fd = socket(...);
if(fd == -1){
    perror("socket:");
    exit(EXIT_FAILURE);
}
```

这段代码用来获取一个 socket，判断获取成功与否，用 fd 是否等于-1 来判断，这种处理错误的方式成为 error code。

C++引入一种**现代编程风格**的错误处理方式：异常处理。

什么是异常处理？首先，异常就是程序运行中的不正常行为。异常处理是采用一种特殊的代码结构对错误进行处理。

例如：

```
int fd;
try {
    fd = socket(...);
    if(fd == -1){
        throw std::runtime_error("socket create error!");
    }

    // .................
} catch (Exception &e) {
    cerr << e.what() << endl; //打印异常信息
}
```

这段代码就是一个异常处理的流程。它的执行流程如下：

1. 创建一个 socket

2. 检查 socket 的返回值为-1，如果不是，一切正常，执行完其他代码，然后直接绕过 catch 块

3. 如果 socket 的返回值为-1，说明出现了错误，这时程序抛出一个异常，这时，程序不再去执行 try 里面其余的代码，而是进入

catch 块，执行里面的错误处理代码。

4. catch 块执行完，程序恢复正常。

从上面的分析可以看出，通常把可能发生异常的代码放入 try 块，把对错误的处理放入到 catch 中。

throw 表达式：

系统通过 throw 抛出异常，如果不对异常进行处理，默认的方式是终止程序。

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

int main() {

    cout << "before" << endl;
    throw std::runtime_error("test!");

    cout << "after" << endl;

}
```

观察程序的输出可以看到，程序没有输出 after，因为抛出异常时就终止了。

用 try/catch 块捕获异常

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

```
int main() {

    try {
        int num;
        cin >> num;
        if(num == 1)
            throw exception("test1");
        if(num == 2)
            throw exception("test2");

        cout << num << endl;
    } catch (...) {
        cout << "catch a exception" << endl;
    }

}
```

上面的代码中，catch(...)的含义是捕获所有异常。

常见的异常有：

1.exception 最常见的问题

2.out_of_range 越界错误

3.invalid_argument 非法参数

将上面的代码做修改如下：

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

int main() {

    try {
        int num;
        cin >> num;
        if(num == 1)
            throw out_of_range("test1");
        if(num == 2)
            throw invalid_argument("test2");

        cout << num << endl;
    } catch (...) {
```

```
            cout << "catch a exception" << endl;
        }

    }
```

catch 块仍然正常工作。

用 catch 捕获特性类型的异常

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

int main() {

    try {
        int num;
        cin >> num;
        if(num == 1)
            throw out_of_range("test1");
        if(num == 2)
            throw invalid_argument("test2");

        cout << num << endl;
    } catch (out_of_range &e) {
        cout << "catch a out_of_range" << endl;
    }

}
```

运行上面的代码，可以看到 catch 块只可以捕获特定类型的异常。如果 try 可能产生不同的类型，可以这样写：

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

int main() {

    try {
        int num;
        cin >> num;
        if (num == 1)
```

```
            throw out_of_range("test1");
        if (num == 2)
            throw invalid_argument("test2");

        cout << num << endl;
    } catch (out_of_range &e) {
        cout << "catch a out_of_range" << endl;
    } catch (invalid_argument &e) {
        cout << "catch a invalid_argument" << endl;
    }

}
```

在以前的 switch case 语句中，我们通常处理几种特殊情况，其余的采用默认处理，这里也可以这样：

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

int main() {

    try {
        int num;
        cin >> num;
        if (num == 1)
            throw out_of_range("test1");
        else if (num == 2)
            throw invalid_argument("test2");
        else
            throw exception();
    } catch (out_of_range &e) {
        cout << "catch a out_of_range" << endl;
    } catch (invalid_argument &e) {
        cout << "catch a invalid_argument" << endl;
    } catch (...) {
        cout << "default process" << endl;
    }

}
```

这里有一处值得注意的地方：

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

int main() {

    try {
        int num;
        cin >> num;
        if (num == 1)
            throw out_of_range("test1");
        else if (num == 2)
            throw invalid_argument("test2");
        else
            throw exception();
    } catch (...) {
        cout << "default process" << endl;
    } catch (out_of_range &e) {
        cout << "catch a out_of_range" << endl;
    } catch (invalid_argument &e) {
        cout << "catch a invalid_argument" << endl;
    }

}
```

上面的程序编译错误，错误信息如下：

error: ‘...’ handler must be the last handler for its try block

原因是 catch(...)可以捕获任意类型的异常，而 catch 块捕获的顺序是从上到下的，一个异常只能被捕获一次。所以后面的 catch 代码均无效。

类型转化运算符

C++仍然支持 C 风格的强制类型转换，但是在 C++中，我们更加推荐使用类型安全的转换方式。

C++提供了四种用于类型转换的运算符：static_cast、dynamic_cast、reinterpret_cast、const_cast。

1.static_cast

static_cast 用于在相关联的指针类型之间进行转换，还可以显式执行标准数据类型的类型转换。

对 static_cast 类型转换的时机是在编译期间（所以它的名字中含有一个 static），确保指针被转换成相关类型，如果类型不匹配会编译产生错误。这跟 C 显然不一样，C 中的指针可以转化为完全不相关的类型，编译器无法检查出错误。

在 C++中，static_cast 可以用于将派生类的指针向上转型为基类指针，也可以把基类指针向下转换为派生类类型。

使用如下：

```
#include <iostream>
#include <string>

using namespace std;

int main(int argc, char **argv) {

    double pi = 3.1415;
    int val = static_cast<int>(pi);
    cout << val << endl;
```

```
    void *p = new int[8];
    int *ptr = static_cast<int*>(p);
}
```

void*是可以转换为 int*的。但是如果我们在代码末尾加上一句：

```
char *pc = static_cast<char*>(ptr);
```

程序编译便有了问题：error: invalid static_cast from type ‘int*’ to type ‘char*’，这在 C 语言中是检查不出来的。

涉及到基类和派生类的部分我们后面讲解。

2.dynamic_cast

通过名字我们就可以看出来，dynamic_cast 是一种发生在运行期间的强制类型转换。与刚才的 static_cast 不同，前者的检查发生在编译期间，如果转化不合法，那么编译报错。dynamic_cast 如果类型转换不成功，会返回空指针。

注意一点：dynamic_cast 针对的是 class 的类型转换。

3.reinterpret_cast

reinterpret_cast 是 C++中与 C 风格类型转换最接近的类型转换运算符，它让程序员能够将一种对象类型转换为另一种，不管它们是否相关。

实际上，reinterpret_cast 常用来转换 static_cast 不允许的转换风格。例如：

```
#include <iostream>
#include <string>
```

```
using namespace std;

int main(int argc, char **argv) {

    void *p = new int[8];
    int *ptr = reinterpret_cast<int*>(p);

    char *pc = reinterpret_cast<char*>(ptr);
}
```

这段代码是通过编译的。

注意点： reinterpret_cast 与 C 风格类型转换还是有区别的，reinterpret_cast 只改变对指针的操作，但是并不进行**对齐操作**，也就是指针的值绝对不会更改，但是 C 风格类型转换可能会改变指针的值。

4.const_cast

const_cast 可以去掉对象的 const 访问属性。不到万不得已，不要使用它。可以用于转化指针，但是可能导致不可预料的后果。