C++ 新特性

不必慌张，一切知识都会随着时间的推移变得逐渐清晰

C++的发展

- C++98
  - 1998年，C++标准委员会发布了C++语言的第一个国际标准
  - 引入了异常处理、模板等
- C++03
  - 2003年，标准委员会针对98版本中存在的诸多问题进行了修订
  - 勘误、修复漏洞，核心语言规则并未变化
- C++11（大约140个新特性，600个缺陷修正）
  - 2005年，C++标准委员会发布了技术报告详细说明了引入C++新特性的计划
  - 代号C++0x （x将来被具体年份替换）
  - 标准最终在2011年才面世，就是C++11
- C++14
  - ▪2014年，标准委员会公布了C++14标准
  - 可以写0b开头的2进制常量
- C++17
  - 2017年，标准委员会公布了C++17标准
  - utf-8

C++11对98/03的显著增强

- 支持本地并行编程
  - 内存模型、线程、原子操作等
- 对泛型编程进一步加强
  - 初始化表达式、auto等
- 更好地支持系统编程
  - constexpr等
- 更好的支持库的构建
  - 内联命名空间、继承构造函数、右值引用等

LONG LONG类型的支持

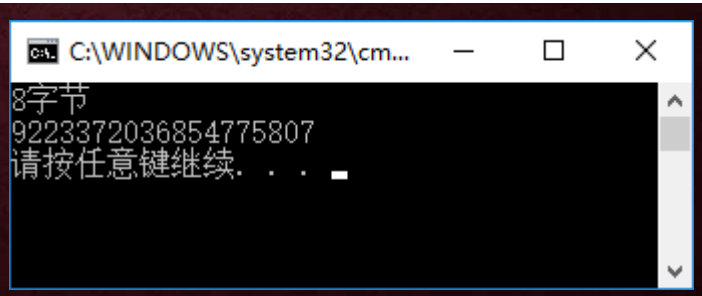
- 至少64位
- long long num=LLONG_MAX
- C++98 时被提议，但被拒绝
- C99采纳
- C++11 使用

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

int main()
{
    long long num = LLONG_MAX;

    cout << sizeof(num) << "字节\n";
    cout << num << endl;

    return 0;
}
```



变量初始化

- C语言支持隐式转换
  - int num=7.2
  - C++继续支持上述方法，但是会警告
- 用花括号初始化
  - int num{7.2}
  - 此时将引发一个错误
  - 从“double”转换到“int”需要收缩转换

推断类型

- 在定义变量时
  - 如果变量类型可以由初始化符号推断
  - 则无需指定类型
- 例如
  - **auto flag=true;**
  - **auto num=7.5;**
  - **auto res=sqrt(num);**
- 此时不可能引发错误的类型转换
- 在编译时推导，不会影响执行效率

- 使得有部分“动态类型”的感觉

模板函数的默认模板参数

```
template <typename T=int>
void test(T temp)
{
...
}
```

- 如果存在多个typename，多个默认值
  - 遵循靠右原则

类中数据成员初始化

- 之前
  - 非静态的数据成员
    - 构造函数
  - 静态数据成员
    - 定义时初始化
- 可以在定义数据成员时立刻初始化变量

空指针

- 传统用NULL表示空指针
  - # define NULL 0
  - int num=NULL;　正确
- nullptr
  - int num= nullptr; 错误
  - int *p= nullptr; 正确

CONSTEXPR

- 指值不会改变并且在编译过程中就得到计算结果的表达式

```
constexpr int Inc(int i) {
   return i + 1;
}

constexpr int a = Inc(1); // ok
constexpr int b = Inc(cin.get()); // !error
constexpr int c = a * 2 + 1; // ok
```

constexpr的好处：

- 是一种很强的约束，更好地保证程序的正确语义不被破坏
- 编译器可以在编译期对constexpr的代码进行优化
  - 比如将用到的constexpr表达式都直接替换成最终结果等
- 相比宏来说，没有预处理的开销

阻止子类再覆盖一个函数

- virtual虚函数
  - 可以实现动态多态
  - 如果在一个虚函数的最
  - 该虚函数就不可以被覆

```
class CA{
public:
virtual void print(){cout << "CA" << endl;}
};
class CB:public CA
{
public:
virtual void print() final{cout << "CB" << endl;}
};
class CC :public CB{
public:
/*virtual void print(){cout << "CC" << endl;}*/
};
int main(){
CA  *p;CC  c1;
p = &c1;p->print();
return 0;
}
```

匿名函数

- 编程范式
  - 命令式编程
  - 声明式编程
  - 函数式编程
- lambda
  - [capture](参数) mutable -> 返回值类型{函数体};
  - [capture] 用户获取父作用域的变量
  - mutable 默认lambda是const函数，用了它可以取消const

移动构造函数

- 拷贝构造函数主要解决指针悬挂问题，主要三种情况会调用
  - 函数形实结合时
  - 函数返回时，函数栈区的对象会复制一份到函数的返回
  - 用一个对象初始化另一个对象时
- 移动构造函数主要提高效率
- C++编译器已经可以区分函数参数是左值调用还是右值

```
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;
class Str{
   public:
      char *str;
      Str(char value[])
      {
         cout<<"普通构造函数..."<<endl;
         str = NULL;
         int len = strlen(value);
         str = new char[len + 1];
         strcpy(str,value);
      }
      Str(const Str &s)
      {
         cout<<"拷贝构造函数..."<<endl;
         str = NULL;
         int len = strlen(s.str);
         str = (char *)malloc(len + 1);
         memset(str,0,len + 1);
         strcpy(str,s.str);
      }
      Str(Str &&s)
      {
         cout<<"移动构造函数..."<<endl;
         str = NULL;
         str = s.str;
         s.str = NULL;
      }
      ~Str()
      {
         cout<<"析构函数"<<endl;
         if(str != NULL)
         {
            delete str;
            str = NULL;
         }
      }
};
```

联合体

- 非受限联合体
  - 之前加入一些限制不允许有些共存
  - 只能是POD类型
  - 不可以是静态类型以及引用类型

```
union T {
    string s;
    int n;
};
```

- 发现没有必要
  - 取消这些限制
  - 使用联合体的构造函数协助解决一些问题

类型的别名

- 复杂的类型名字变得简单明了、易于理解和使用
- 有助于程序员清楚地知道使用该类型的真实目的
- typedef
  - typedef unsigned int uint;
  - typedef uint uint32;
- Using
  - using uint= unsigned int;
  - 包含typedef的功能
- 对模板进一步扩展
  - template <typename T>
  - typedef std::vector<T> v;//使用typedef
  - using v = std::vector<T>;//使用using

通过USING可以导出模板类型

```
#include<iostream>
using namespace std;
template <typename T>
class ctx {
public:
    using value_type = T;
};
int main(int argc, char **argv) {
    ctx<int>::value_type a = 5;
    cout << a << endl;
    return 0;
}
```

智能指针

- 指针处理使用三问题
  - 野指针
  - 重复释放
  - 内存泄漏
- C++98有智能指针： auto_ptr
  - 可以自己相机释放
  - 是独占性的，不允许多个auto_ptr指向同一个资源
  - 被废除
- **unique_ptr**
- shared_ptr
- **weak_Ptr**

UNIQUE_PTR

- auto_ptr的升级版
  - 尝试复制unique_ptr时会编译期出错
  - auto_ptr能通过编译，在运行期埋下出错的隐患

SHARED_PTR

- 内部会有一个计数器
  - 每次复制计数+1
  - 每次释放计数-1
  - 当计数器为0就可以释放资源
- 缺陷：
  - 循环依赖(互相引用或环引用)时，计数会不正常

WEAK_PTR

- 不释放资源，只管理自己计数器
- weak_ptr没有重载 * 和 -> ，
  - 并不能直接使用资源
  - 可以使用lock()获得一个可用的shared_ptr对象，
  - 如果对象不存在，lock()会失败，返回一个空的shared_ptr

三种指针总结

- 当需要一个独占资源所有权（访问权+生命控制权）的指针，且不允许任何外界访问，使用**std::unique_ptr**
- 当需要一个共享资源所有权（访问权+生命控制权）的指针，使用**std::shared_ptr**
- 当需要一个能访问资源，但不控制其生命周期的指针，使用**std::weak_ptr**
- 推荐用法：
  - 一个**shared_ptr**和**n**个**weak_ptr**搭配使用 而不是**n**个**shared_ptr**

线程库

- 多线程
- 并行编程