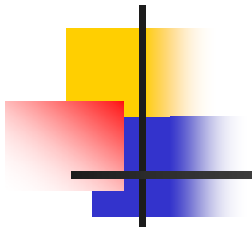
第二章 C++初探:创建和使用对象

面向对象程序设计(C++)

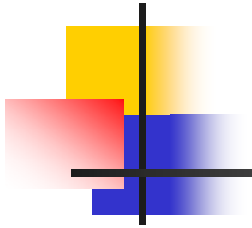


2.1 概述

This chapter explains key differences between C and C++, and takes you through three essential C++ features:

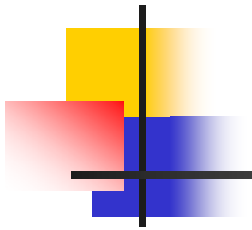
- **Type safety (类型安全性)**
- **Classes (类：抽象、封装、多态性等)**
- **Templates (模板)**

Also covered are IOStreams and the *free store* operators new and delete.



2.1 概述(cont.)

- History
- Changes to C subset
- IOStreams
- new and delete
- Objects
- Templates

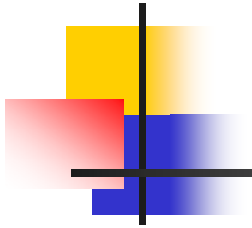


2.2 C++的历史

- **Bjarne Stroustrup, Bell Labs (1980s)**
- **C with Classes**
  - **Add objects to C**
  - **Leverage C's efficiency(效率), portability(轻巧性), availability(可用性)**
- **ANSI Committee, 1991**
  - **ISO Standard, July 1998**

2.3 对C子集的某些改进

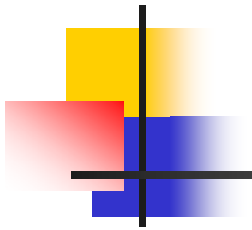
- **Motivated mostly by type safety**
- **'a' is char, not int (@c)**
- **"a" is const char*, not char* (@c)**
- **f() is the same as f(void)**
  - 而在C中等同于参数个数不确定的函数
- **const integers can be used as array dimensions**
  - 常量在C++中通常是一个符号表中的条目；而在C中是一个变量。
- **Structure tags are type names**



2.4 对C的扩展

- Abstract (抽象)
- Encapsulation (封装)
  - Access Control(Public,Private,Protected)
  - Friends(友元)：允许友元破坏封装性
- Inheritance (继承)
- Virtual Function (多态性)
  - Later Binding (晚绑定)
- Overloading (重载)

  -- 允许函数名和运算符重载
- Template (模板)



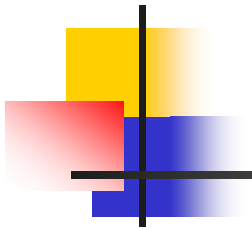
2.5 C++的输入输出:IOStreams初探

```
// hello.cpp
#include <iostream>
using namespace std;
int main() {
    cout << "Hello, world" << endl;
}
```

Hello, world

- cout是一个预定义的对象；
- cout<<“Hello,world” 等价于：
  cout.operator<<(“Hello,world”)
- C++的输入输出是类型安全的输入输出. (取代printf)

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

int main() {
    int a=10;
    cout << "a";
    cout << a;
    return 0;
}
```



2.6 new and delete 运算符

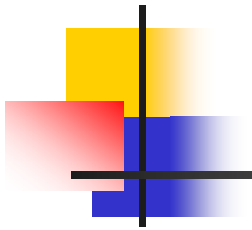
- **Replacement for malloc/free**

```
int* ip = new int(7);
delete ip;
int* iap = new int[10];
delete [] iap;
```

- **Compile-time operators**

  --- Calculate size of objects for you

- **Automatic initialization & cleanup**



2.7 对象(Object)概述

- **Based on Classes**
  - **structs with member functions**
  - **Always have an associated object**
- **Improved support for information hiding**
  - **private, protected keywords**
- **Automatic initialization/cleanup(构造和析构函数)**

2.7.1 例：Class Stack

```
// intstack.h: A Stack class for ints
class StackOfInt {
public:
      StackOfInt(int); //构造函数，生成对象时自动执行
      void push(int);
      int pop();
      int top() const;
      int size() const;
      ~StackOfInt();  //析构函数，撤销对象时自动执行
private:
      int *data;
      int length;
      int ptr;
};
```

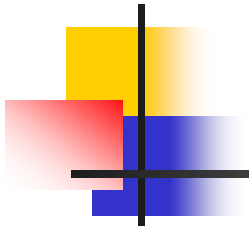
```
// intstack.cpp
#include "intstack.h"
StackOfInt::StackOfInt(int stk_size) {
    data = new int[length = stk_size];
    ptr = 0;
};
void StackOfInt::push(int x) {
    if (ptr < length)
        data[ptr++] = x;
    else
        throw "overflow";
}
int StackOfInt::pop() {
    if (ptr > 0)
        return data[--ptr];
    else
        throw "underflow";
}
```

```
// (intstack.cpp continued)
int StackOfInt::top() const {
    if (ptr > 0)
        return data[ptr-1];
    else
        throw "underflow";
}
int StackOfInt::size() const {
    return ptr;
}
StackOfInt::~StackOfInt() {
    delete [] data;
}
```

```
// tintstack.cpp: Tests StackOfInt
#include "intstack.h"
#include <iostream>
using namespace std;
int main() {
    const int N = 5;
    StackOfInt stk(N);

    for (int i = 0; i < N; ++i)
        stk.push(i);
    while (stk.size() > 0)
        cout << stk.pop() << ' ';
    cout << endl;
}
```

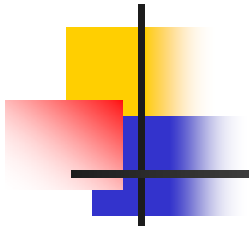
```
// tintstack.cpp: Tests StackOfInt
#include "intstack.h"
#include <iostream>
using namespace std;
int main() {
    const int N = 5;
    StackOfInt stk(N);

    for (int i = 0; i < N; ++i)
        stk.push(i);
    while (stk.size() > 0)
        cout << stk.pop() << ' ';
    cout << endl;
}
```

输出：*4 3 2 1 0*



2.8 Templates(模板：参数化的类)

- Support *generic programming*
- Ideal for *containers*
  - Logic is independent from contained objects
- You write the template code once
- Compiler generates versions on demand



例：通用栈模板

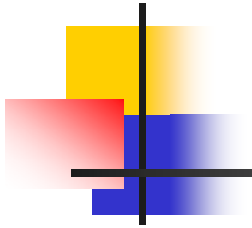
```
// stack9b.h: A Stack template
template<class T>
class Stack {
public:
    Stack(int);
    void push(T);
    T pop();
    T top() const;
    int size() const;
    ~Stack();
private:
    T *data;
    int length;
    int ptr;
};
```

```
template<class T>
Stack<T>::Stack(int stk_size) {
    data = new T[length = stk_size];
    ptr = 0;
};
…
template<class T>
void Stack<T>::push(T x) {
    if (ptr < length)     data[ptr++] = x;
    else         throw "overflow";
}
template<class T>
T Stack<T>::pop() {
    if (ptr > 0)    return data[--ptr];
    else      throw "underflow";
}
```

```
template<class T>
T Stack<T>::top() const {
    if (ptr > 0)
        return data[ptr-1];
    else
        throw "underflow";
}
…
// tstack9b.cpp: Tests the Stack template
#include "stack9b.h"
#include <iostream>
using namespace std;
int main() {
    const int N = 5;
    Stack<float> stk(N);
```

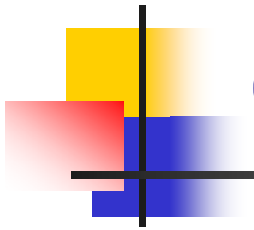
```
    for (int i = 0; i < N; ++i)
        stk.push(i + 0.5);
    while (stk.size() > 0)
        cout << stk.pop() << ' ';
    cout << endl;
}
```

输出：***4.5 3.5 2.5 1.5 0.5***



2.9 小结:*A First Look at C++*

- C++ emphasizes type safety
- IOStreams provide type-safe I/O
- new and delete provide safe heap management
- Classes are like structs
  - They allow member functions
  - They support explicit access control
- Templates support generic programming



Congratulations!

You're ready to tackle C++

补充1：函数模板

- 函数模板是函数重载的一种特殊方式

- 模板编译时难以检查参数相关的错误（实际中必须配合模板说明使用）

- 函数模板调用时才会生成可执行代码

```
template<typename T>
T max(T a, T  b)
{
    if (a>b)
        return a;
};

std::cout << max(3,5)<<endl;
```

模板隐式实例化

补充1：函数模板

- **数组模板的求和**

```
template<typename T>
T sum(const T* array, int n)
{
    T sum(0);
    for (int i=0;i<n;++i)
        sum+=array[i];
    return sum;
};
```

如何避免参数传递时需要人为确定n?

补充1：函数模板

```
template<typename T, unsigned N> //让编译器推断N


T sum(const T(&array)[N])
{
    T sum(0);
    for (int i=0;i<N;++i)
        sum+=array[i];
    return sum;
};
```

- 更具一般性的解决方法是通过拟函数

补充2：拟函数

- **元编程技术的重要基础**
- **实际是能够像函数一样调用的类**

函数求导的例子：

```
double fin_diff(double f(double), double x, double h)
{
    return (f(x+h)-f(x))/h;
};

double  sin_plus_cos(double x)
{
    return sin(x)+cos(x);
}
```

补充2：拟函数

```
int main(){
    cout<<fin_diff(sin_plus_cos,1,0.001)<<endl;
    cout<<fin_diff(sin_plus_cos,0,0.001)<<endl;
}
```

- 如果我们希望迭代调用fin_diff计算二阶导数会遇到什么问题?

  fin_diff(fin_diff)?

- 如何处理与函数功能密切相关的可调参数?

补充2：拟函数

- **拟函数实现**

```
class psc_f
{
public:
    psc_f(double alpha): alpha(alpha) {}

    double operator() (double x) const
    {
        return sin(alpha*x)+cos(x);
    }                          //重载运算符()
private:
    double alpha;   // 重要参数的内部保留
};
```

补充2：拟函数

- **求导拟函数模板**

```
template <typename F, typename T>
class derivative
{
  public:
    derivative(const F& f, const T& h):f(f),h(h){}

    T operator() (const T& x) const
    {
       return (f(x+h)-f(x))/h;  // 调用时只需传入求导位置x
    }
  private:
    const F& f;
    T            h;   //步长成为内部变量
}
```

补充2：拟函数

```
using d_psc_f=derivative<psc_f, double>;

psc_f       psc_o(1.0);     // 初始化需要被求导的拟函数
d_psc_f     d_psc_o(psc_o, 0.001); //初始化计算导数的拟函数

using dd_psc_f=derivative<d_psc_f, double>;

dd_psc_f   dd_psc_o(d_psc_o, 0.001);

求x=0时的二阶导数：
cout<<dd_psc_o(0)<<endl;
```

- **从上面这个例子可以看出，拟函数可以用于非常复杂的函数设计，特别是基于已有的拟函数继续构造新的拟函数（元编程）**

补充3：Blackjack(21点)游戏的框架设计

- 点数总和尽量接近**21**点（超过**21**点爆牌，输）
- **J，Q，K**点数**=10**，**A=1**或者**11**；没有大小王
- （**J**或者**Q**或者**K**）**+A=Blackjack**，获胜玩家赢得**1.5**倍筹码
- 游戏开始，向每位玩家和庄家发两张牌，庄家一张牌不可见
- 玩家可以任意添牌（此阶段玩家爆牌即输掉筹码）
- 玩家添牌结束后，庄家亮出隐藏牌。庄家点数**<17**必须添牌，**>=17**必须停牌（庄家爆牌，输；否则，比大小）

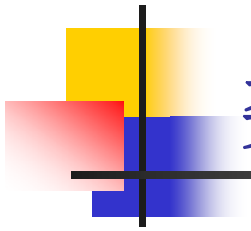
问题：如何基于面向对象的设计思想进行程序架构?

类的设计

类	基类
Card	
Hand	
Deck	Hand
GenericPlayer	Hand
Player	GenericPlayer
House	GenericPlayer
Game	

- 其它设计方法？

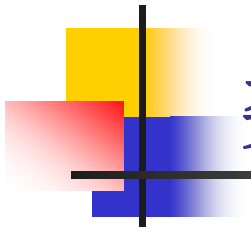
- **Card**对象对应真实扑克牌（不可复制）
- **Hand**类包含**Card**对象的指针（某个对象手牌的集合）
- **Deck**类：洗牌和发牌（多态）



类的描述

Card类

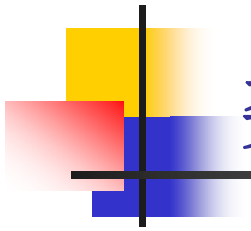
成员	描述
rank m_Rank	rank: 枚举类型
suit m_Suit	suit: 枚举类型
bool m_IsFaceUp	
int GetValue()	
void Flip()	



类的描述

Hand类

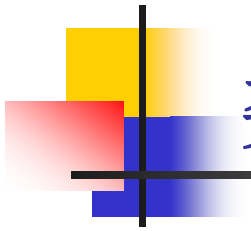
成员	描述
vector<Card*> m_Cards	
void Add(Card* pCard)	
void Clear()	
int GetTotal() const	



类的描述

GenericPlayer类

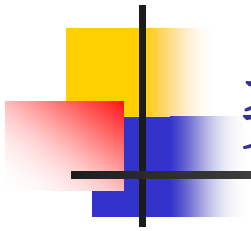
成员	描述
string m_Name	
virtual bool IsHitting() const=0	是否添牌？
bool IsBusted() const	是否爆牌（自动计算）？
void Bust() const	显示



类的描述

Player类

成员	描述
string m_Name	
virtual bool IsHitting() const=0	
bool IsBusted() const	
void Bust() const	显示
void Win() const	Blackjack
void Lose() const	
void Draw() const	
m_Bet	余额



类的描述

House类

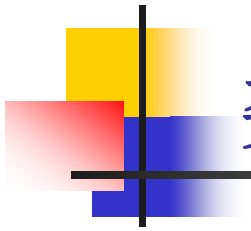
成员	描述
string m_Name	
virtual bool IsHitting() const=0	
bool IsBusted() const	
void Bust() const	显示
void FlipFirstCard()	?

类的描述

Deck类

成员	描述
◦ ◦ ◦	
◦ ◦ ◦	
◦ ◦ ◦	
◦ ◦ ◦	
void Populate()	
void Shuffle()	?
void Deal(Hand& aHand)	
void AdditionalCards(GenericPlayer* aGenericPlayer)	

- 可能出现的其它情况？



类的描述

Game类

成员	描述
Deck m_Deck	
House m_House	
vector<Player> m_Players	
void Play()	?

每轮游戏流程

- **Deal Players&House** 两张牌
- 隐藏**House**第一张牌
- 显示**Players&House Hand**
- **Deal Players** 添牌
- 显示**House**第一张牌
- 判断**House**点数，添牌

- 如果 **House Busted**
  - 没有爆牌的**Players**获胜

  否则
  - 遍历**Players**
    - 如果 **Player**点数大
      - 获胜
    - 否则 **Player**点数小（爆牌的如何处理？）
      - 失败
    - 否则 **Player**点数相同
      - 打平