
中国科学技术大学
University of Science and Technology of China
程序设计 II
Programming Design II
C++概述
主讲：吴锋

目录
CONTENTS

C++的发展历史

- C++由同是贝尔实验室的 Bjarne Stroustrup 在C语言的基础上于在20世纪80年代发明并实现
- C++继承了C语言的高效性、可移植性和可用性
- 起初，C++被称作 “C with Classes” ，作为C语言的增强版出现，随后不断增加了新特性
  - 1980 - 1995：面向对象语言，性能接近C（±5%）
  - 1995 - 2000：泛型程序设计，STL和Boost库
  - 2000 - 至今 ：产生式编程和模板元编程
- 1998年，国际标准组织（ISO）颁布了C++的第一个国际标准C++98，目前最新标准为C++20

C++的编程范式

- C++支持多重范式的编程
  - **C**：除了C的语句、预处理、内置数据类型、数组、指针外，C++还改进了数据类型安全
    - C 的库 xxx.h 对应于 C++ 中的 cxxx，如 stdio.h -> cstdio，stdlib.h -> cstdlib，math.h -> cmath，string.h -> cstring
    - 同时还要使用标准名字空间：using namespace std;
  - **Object-Oriented C++**：也就是面向对象设计在C++中的实现，包括类、封装、继承、多态，虚函数等
  - **Template C++**：也就是C++的泛型编程部分，包括模板编程以及模板元编程等，功能强大，难度高
  - **STL & Boost**：也就是C++的标准模板库和程序库，容器、迭代器、算法和函数对象自成一体

C++的数据类型

- 数据类型：定义使用存储空间(内存)的方式
- 通过定义数据类型，告诉编译器怎样创建一片特定的存储空间，以及怎样操纵这片存储空间
- C++的数据类型包括：
  - 语言本身内置类型（Built-In Types）
    - 如：bool, char, int, float, double 等
  - 用户自定义类型（User-Defined Types）
    - 如：enum, struct, class 等
  - 标准库定义类型
    - 如：string，vector，list 等

C++的类型安全

- 类型安全：每个数据(常量/变量)必须有对应的类型，只有符合声明类型的操作才能够在数据上执行
  - 静态类型安全：编译器检查程序有无违反类型安全规则，如果不符合类型安全，则编译器报错
  - 动态类型安全：在运行阶段能够探测编写的程序是否违反类型规则，如果不符合类型安全，则抛出异常
- C++虽然对C进行了类型安全方面的改进，但依然不是完全静态类型安全，也不是完全动态类型安全
  - 完全静态类型安全限制程序的表达力
  - 完全动态类型安全则影响程序的效率

C++的变量引用

- 引用是通过别名直接访问某个变量，对引用的操作就是对被引用的变量的操作
  - 相对应的，指针是通过地址间接访问某个变量
  - 在大多数场合可以替代指针，且比指针安全（不会出现空指针、野指针的问题），比如函数传参

```
int a = 10;
int &b = a;      // b是a的别名;
b = b + 10;      // 对b的改变，实际上改变的是a;
cout << a;       //输出：20
void set(int &c, int d) { c = d; }
set(a, 30);      // 对参数的改变，也改变引用对象
cout << a;       //输出：30
```

C++的变量引用

- 引用在定义时必须初始化，从而避免空指针问题；初始化后，不能改变引用的“指向”，从而避免野指针问题

```
int a =10; int &b = a;
int &c; // error,未初始化。
```

- 可以用某个引用的地址值赋给一个指针，而指针则指向被引用的变量，实现引用和指针的转换

```
int a = 10; int &b = a;
int *p;  p = &b;     // 则p指向a
```

- 可以用某个引用初始化另一个引用

```
int a;
int &r1 = a;
int &r2 = r1;
r2 = 10;          // a的值被改变。
```

C++的常量修饰符

- const 常量修饰符将一个对象变为常量，程序中对这个值的任何修改将导致编译错误
  - 如： const int bufSize = 5; bufSize = 3; //错误
  - 常量必须在定义时初始化，如： const int bufSize; //错误
  - 整形常量可以作为数组维度定义，如： int a[bufSize];
  - const int *cptr; // 一个指向常量的指针变量
    - 即指针本身的地址可变，但指向的对象的值不可变（常量）
  - int *const cptr2; // 一个指向变量的常量指针
    - 即指针本身的地址不可变，但指向对象的值可变（变量）
- 编程过程中尽可能用 const 常量来代替C中的宏，因为常量能够被编译器检查，而宏在预处理时即被替换

C++的输入输出

- C++的输入输出由iostream库提供，是标准库的一部分，提供了三个标准流
  - cin：标准输入，从用户终端读入数据
  - cout：标准输出，向用户终端写数据
  - cerr：标准错误，导出程序错误消息
- 输出操作通过重载左移操作符 << 实现，输出操作通过重载右移操作符 >> 实现
  - cout << “Hello,world”; 等价于：

    cout.operator<<(“Hello,world”);

C++的输入输出

```
#include <iostream>  // 使用标准流需要包含的头文件
using namespace std;  // 使用标准流需要使用的名字空间
int main()
{
    int v1, v2;
    // 流式输入和输出，支持所有的内置类型
    if (cin >> v1 >> v2) // 根据返回值判断是否结束或失败
    // endl 是预定义的操作符
    // 用于在输出流中插入一个换行符 \n
    // 同时刷新输出缓冲区。
    cout << "v1 + v2 = " << v1 + v2 << endl;
    return 0;
}                                    // 输入：7 3 输出：v1 + v2 = 10
```

C++的动态内存分配

- new: 从“堆”中动态分配存储块
- delete: 从“堆”中动态删除存储块

```
int  *pInt;
 pInt = new int ;          // 从堆中动态分配内存
 if (pInt ==0)
        exit(-1);          // 申请失败
 else { …;}                // 申请成功
  …
 delete pInt;              //释放空间
```

C++的动态内存分配

- 为数组分配空间/撤销空间

```
int size ;
cin >> size;
int * parrayInt = new int[size]; //动态申请
char * string = new char[20];
…… //使用parrayInt

//释放空间，[ ]表示撤销整个数组，
delete [ ] parrayInt;
delete [ ] string;
```

C++的动态内存分配

- 编译时，new 可以根据对象的类型，自动决定对象的大小；而 malloc 需要显式指定需要分配空间的大小（通常调用sizeof来实现）。
- new 返回指向正确类型的指针，不必进行强制类型转换；而 malloc 返回 void *，必须进行强制类型转换，可能带来类型错误。
- 用 new 生成对象时,会调用构造函数, 用malloc则不会;同样,用 delete 删除对象时,会调用析构函数, 用free则不会。
- new/delete 必须配对使用，malloc/free 也一样。

C++的动态内存分配

- 如果用free释放“new创建的动态对象”，那么该对象因无法执行析构函数而可能导致程序出错
- 如果用delete释放“malloc申请的动态内存”，理论上讲程序不会出错，但是该程序的可读性很差，还可能内存泄露

```
Time * pint = new Time;     // 正确地构造
delete pint;                // 正确地析构

Time * pint = new Time;     // 正确地构造
free(pint) ;                // 可以释放pint,但不调用析构函数,未能完
                            // 整释放空间.

Time *pint = (Time *) malloc(sizeof(Time));
                            // 仅分配空间，未调用构造函数
delete pint;                // 正确地析构
```

C++的类

- 类可以简单的理解为：带成员函数和访问控制的结构体，具有自动执行的构造和析构函数。

```
// intstack.h: 定义一个整型栈类
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;
};
```

C++的类

```
// intstack.cpp 用于实现类中的成员函数
#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;
}
```

C++的类

```
// 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;
    return 0;
}
                                    // 输出: 4 3 2 1 0
```

C++的字符串类

- 标准C++中提供了方便字符串存储和操作的string类

```
#include <string> // 包含头文件
using namespace std; // 导入名字空间

// 初始化
string s1();  // 初始化一个空字符串
string s2("Hello");  // 字符串初始化

// 赋值，相当于 strcpy
s1 = "Hello";  // s1 = "Hello"
s2 = 'K';  // s2 = "K"

// 取得字符串长度，相当于 strlen
int len1 = s1.length();
int len2 = s2.size();

// 两个字符串相连，相当于 strcat
string s3 = s1 + s2 + '\n';
```

```
// 两个字符串的比较，相当于 strcmp
int n = s1.compare(s2);

// 查找子串，相当于 strstr
int k = s2.find("el");

// 取得字符串中的第i个字符
int i = 1;
int c1 = s1[i];
s1[i] = 'a';

// 输入 / 输出字符串
cin >> s1;
cout << s2;

// 转化为 C 风格的字符串
const char * cstr = s1.c_str();
```

C++的模板

- 模板可以简单的理解为带参数的类
- 数据结构可以理解为数据的一种容器，即数据结构与其存储的数据类型可以分离
- 程序员根据数据结构的操作逻辑编写模板
- 编译器在编译阶段决定具体存储的数据类型
- 有效的提高程序的可复用性，提高编码效率
- 类似的思想：C库函数使用函数指针，如qsort

C++的模板

```
template<class T> // stack.h: 数据类型为模板参数 T
class Stack {
public:
    Stack(int);
    void push(T);
    T pop();
    T top() const;
    int size() const;
    ~Stack();
private:
    T *data; // 实现时无需考虑具体的数据类型，用T代替
    int length;
    int ptr;
};
```

C++的模板

```
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";
}
…
```

C++的模板

```
#include "stack.h"
#include <iostream>
using namespace std;
int main() {
    const int N = 5;
    Stack<float> stk(N); // 使用时决定数据的类型，如 float

    for (int i = 0; i < N; ++i)
        stk.push(i + 0.5);
    while (stk.size() > 0)
        cout << stk.pop() << ' ';
    cout << endl;
    return 0;
}                                   // 输出：4.5 3.5 2.5 1.5 0.5
```

C++的vector容器

- 标准C++中的vector<T>容器为数组提供了一种替代表示

```
#include <vector> // 包含头文件
using namespace std; // 导入名字空间

vector<int> ivec1; // 定义一个空向量
vector<int> ivec2(10); // 长度为10的向量

for (int i = 0; i < 100; ++i) {
  ivec1.push_back(i); // 添加元素
}

for （int i = 0; i < ivec1.size(); ++i） {
  ivec1[i] += 1; // 修改元素
}

for (int i = 0; i < ivec1.size(); ++i) {
  cout << ivec1[i] << endl; // 输出元素
}
```

需要结合标准模板库STL中的泛型算法使用，才能发挥容器的最大功效。

◎ 从 C 快速过渡到 C++

- 用 引用 & 代替 指针 *
- 用 常量 const 代替 宏 #define
- 用 cout / cin 代替 printf / scanf
- 用 new / delete 代替 malloc / free
- 用 类 class 代替 结构体 struct
- 用 模板 template 代替 代码拷贝
- 用 字符串类 string 代替 字符数组
- 用 向量模板 vector<T> 代替 数组

得到了一个更安全、更好用的C语言改进版。