从今天开始我们进入 C++学习

1、 C 和 C++有什么不同

2、 C++新增特性

（1） 命名空间

（2） 引用

（3） 函数的默认参数

（4） 内联函数

（5） 函数的重载

（6） 输入输出流

（7） 强制类型转换

（8） 变量的说明可以放在程序任何位置

（9） const 关键字

（10） new delete(delete[])

（11） 码段、数据段、BSS 段讲解

1、C 和 C++有什么不同？

1、从机制上：c 是面向过程的（但 c 也可以编写面向对象的程序）；c++是面向对象的，提供了类。但是，c++编写面向对象的程序比 c 容易
2、从适用的方向：c 适合要求代码体积小的，效率高的场合，如嵌入式；c++适合更上层的，复杂的； linux 核心大部分是 c 写的，因为它是系统软件，效率要求极高。
3、从名称上也可以看出，c++比 c 多了+，说明 c++是 c 的超集；那为什么不叫 c+而叫 c++呢，是因为 c++比 c 来说扩充的东西太多了，所以就在 c 后面放上两个+；于是就成了 c++
4、C 语言是结构化编程语言，C++是面向对象编程语言。C++侧重于对象而不是过程，侧重于类的设计而不是逻辑的设计。

1、c++概述

由 AT&T 贝尔实验室的 Bjarne Stroustrup 开发
从 C 语言派生的，与 C 语言是兼容的
新增关键字：class friend virtual inline private public protected const this string 等
新增运算符：new delete operator ：：等

1.1 命名空间

C++中引入命名空间的主要是为了处理程序中常见的命名冲突，它是由 ANSI C++引入的可以由用户命名的作用域。

所谓命名空间，实际上就是一个由程序设计者命名的内存区域 ，程序设计者可以根据需要指定一些有名字的空间域，把一些全局实体分别放在各个命名空间中，从而与其它全局实体分隔开来。如：

```
namespace ns            //指定命名空间 ns
{
  int a;
  double b;
}
```

namespace 是定义命名空间所必须写的关键字，ns 是用户自己指定的命名空间的名字（可以用任意的合法标识符），在花括号内是声明块，在其中声明的实体称为命名空间成员（namespace member）。现在命名空间成员包括变量 a 和 b，注意 a 和 b 仍然是全局变量，仅仅是把他们隐藏在指定的命名空间中而已。如果在程序中要使用变量 a 和 b，必须加上命名空间名和作用域分辨符“::”，如 ns::a，ns::b。这种用法称为命名空间限定，这些名字（如 ns::a）称为被限定名。

命名空间的作用是建立一些相互分隔的作用域，把一些全局实体分隔开来，以免产生名字冲突。

可以根据需要设置多个命名空间，每个命名空间名代表一个不同的命名空间域，不同的命名空间不能同名。这样可以把不同的库中的实体放到不同的命名空间中，或者说，用不同的命名空间把不同的实体隐蔽起来。过去我们用的全局变量可以理解为全局命名空间，独立于所有有名的命名空间之外，它是不需要用 namespace 声明的，实际上是由系统隐式声明的，存在于每个程序中。

在标准 C++以前，都是用#include<iostream.h>这样的写法的，因为要包含进来的头文件名就是 iostream.h。标准 C++引入了名字空间的概念，并把 iostream 等标准库封装到了 **std** 名字空间中，同时为了不与原来的头文件混淆，规定标准 C++使用一套新的头文件，这套头文件的文件名后不加.h 扩展名，如 iostream、string 等等，并且把原来 C 标准库的头文件也重新命名，如原来的 string.h 就改成 cstring(就是把.h 去掉，前面加上字母 c)，所以头文件包含的写法也就变成了#include <iostream>。

并不是写了#include<iostream>就必须用 using namespace std;我们通常这样的写的原因是为了一下子把 std 名字空间的东西全部暴露到全局域中（就像是直接包含了 iostream.h 这种没有名字空间的头文件一样），使标准 C++库用起来与传统的 iostream.h 一样方便。如果不用 using namespace std;使用标准库时就得时时带上名字空间的全名，如 std::cout << "hello" << std::endl;

```
#include <iostream>
int main(void)
{
    std::cout<<"Hello World"<<std::endl;
    return 0;
}
```

2、c++新增特性

- 引用
- 函数的默认参数
- 内联函数
- 函数重载
- 输入/输出流
- **string 类型**

2.0、 string 类型

在 C 语言中定义字符串要用字符数组或者字符指针（构造数据类型）。如：

char szStr[10] = “hello”;　或　 char *szStr = “hello”;

但是在做字符串的相关复制和加法和比较等等运算时，要用到字符串相关的函数。

在 C++中用到 strcpy 等函数，包含头文件:

#include <string>　或　#include <string.h>

用到数学函数，如 sqrt()等时，包含头文件：

#include <cmath>　或　#include <math.h>

C++中对此进行了改进，引用了 string 类型。如：

string szStr1 = “hello”;　 string szStr2 = “world”;

string szStr = szStr1 + szStr2;　可以直接相加，而不用借助于 strcat 或 strcpy 等函数。

需要包含头文件：#include <string> using namespace std;

例程：

以前用 C 语言写的：

```c
#include <stdio.h>
#include <string.h>

int main(void)
{
    char buf1[] = {"string111"};
    char buf2[] = {"string222"};

    if(strcmp(buf1,buf2))
    {
        printf("不相等\n");
    }
    else
        printf("相等\n");

    return 0;
}
```

C++添加 string 后

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

//增加string类型

int main()
{
    string szStr1 = "hello";
    string szStr2 = "world";
    string szStr = szStr1 + szStr2;

    if(szStr1>szStr2)
        cout<<"ture"<<endl;
    else
        cout<<"flase"<<endl;
    cout<<szStr1<<endl;
    cout<<szStr2<<endl;
    cout<<szStr<<endl;
    return 0;
}
```

2.1、输入输出流：

流是字符集合或数据流的源或目的地

有两种流

输出流,在类库里称为 ostream

输入流,在类库里称为 istream

同时处理输入和输出,由 istream 和 ostream 派生双向流 iostream.

C++名字	对应设备	C 对应名字	默认设备
cin	标准输入流	stdin	键盘
cout	标准输出流	stdout	屏幕
cerr	标准错误流（非缓冲）	stderr	屏幕
clog	标准错误流（缓冲)	stdprn	打印机

C:　printf(”Welcome!”);
C++: cout<<”Welcome”;
C:　scanf(“%d”,&a);
C++: cin>>a;
C:　include<stdio.h>
C++: include <iostream>

2.2、引用：

引用就是变量的别名，对引用的操作与对变量直接操作是完全一样的。

引用的声明：　数据类型　&　引用名　=　初始值;　（初始值是变量名）

引用在定义的时候必须要初始化。注意：此处的’&’并不是取地址符，而是“引用说明符”。声明一个引用并不是定义了一个新的变量，只表示给变量取了一个别名，**不能再把该引用名作为其他变量名的别名。**编译器会给它分配内存空间，因此引用本身占据存储单元，但是引用表现出来给用户看到的，不是引用自身的地址，而是目标变量的地址。也就是说对引用取地址就是目标变量的内存地址。

c++的函数允许利用引用进行参数传递，具有高效性和安全性

引用作为函数参数的特点如下：

1. 在进行实参和形参的结合时，不会为形参分配内存空间，而是将形参作为实参的一个别名。使用引用传递函数的参数，在内存中并没有产生实参的副本，它是直接对实参操作；而使用一般变量传递函数的参数，当发生函数调用时，需要给形参分配存储单元，形参变量是实参变量的副本；因此，当参数传递的数据较大时，用引用比用一般变量传递参数的效率和所占空间都好。

2. 用引用，能达到同指针传递同样的效果，则函数内对形参的操作相当于直接对实参的操作，即形参的变化会影响实参。但是引用相对指针的优点如下：利用指针传递时，在被调函数中同样要给形参分配存储单元，且需要重复使用"*指针变量名"的形式进行运算，这很容易产生错误且程序的阅读性较差；另一方面，在主调函数的调用点处，必须用变量的地址作为实参。而引用更容易使用，更清晰。

```
void swap(int & i, int &j)
{
    int tmp = i;
    i = j;
    j = tmp;
}
int main(void)
{
    int a=3,b=4;
    swap(a,b);
}
```

返回引用:

```
int &fn(int &num)
{
    return(num);
}
void main()
```

```
{
    int n1, n2;
    n1 = fn(n2);
}
```

2.3、函数的默认参数

```
void func(int num1,int num2 = 3,char ch = '*'){}
void func(int num1=2,int num2,char ch='+');//错误
```

注意点：

1. 一旦给一个参数赋了默认值，则它右面的所有参数也都必须有默认值。
2. 默认值的类型必须正确。
3. 默认值可以在原型声明或者函数定义中给出，但不能在两个位置同时给出，建议在原型声明中指定默认值。
4. 使用函数默认参数时：一旦给一个参数不采用默认值，则它左边的所有参数也都必须不采用默认值

```
func(2,13,'+'); //都不采用默认值
func(1);        //第二个和第三个参数采用默认值
func(2,25);     //第三个参数采用默认值
func();         //所有这三个参数都采用默认值
func(2,,'+'); //错误！
```

优点:

如果要使用的参数在函数中几乎总是采用相同的值，则默认参数非常方便

通过添加参数来增加函数的功能时，默认参数也非常有用

2.4、内联函数节省短函数的执行时间 (拿空间换时间)

一、inline 关键字用来定义一个类的内联函数，引入它的主要原因是用它替代 C 中表达式形式的宏定义。

表达式形式的宏定义一例：

#define ExpressionName(Var1,Var2) ((Var1)+(Var2))*((Var1)-(Var2))为什么要取代这种形式呢：

1． 首先谈一下在 C 中使用这种形式宏定义的原因，C 语言是一个效率很高的语言，这种宏定义在形式及使用上像一个函数，但它使用预处理器实现，没有了参数压栈，代码生成 等一系列的操作,因此，效率很高，这是它在 C 中被使用的一个主要原因。

2． 这种宏定义在形式上类似于一个函数，但在使用它时，仅仅只是做预处理器符号表中的简单替换，因此它不能进行参数有效性的检测，也就不能享受 C++编译器严格类型检查的好处，另外它的返回值也不能被强制转换为可转换的合适的类型，这样，它的使用就存在着一系列的隐患和局限性。

3． 在 C++中引入了类及类的访问控制，这样，如果一个操作或者说一个表达式涉及到类的保护成员或私有成员，你就不可能使用这种宏定义来实现(因为无法将 this 指针放在合适的位置)。

4． inline 推出的目的，也正是为了取代这种表达式形式的宏定义，它消除了它的缺点，同时又很好地继承了它的优点。

二、为什么 inline 能很好地取代预定义呢？
对应于上面的 1-3 点，阐述如下：
1． inline 定义的类的内联函数，函数的代码被放入符号表中，在使用时直接进行替换，（像宏一样展开），没有了调用的开销，效率也很高。
2． 很明显，类的内联函数也是一个真正的函数，编译器在调用一个内联函数时，会首先检查它的参数的类型，保证调用正确。然后进行一系列的相关检查，就像对待任何一个真正的函数一样。这样就消除了它的隐患和局限性。
3． inline 可以作为某个类的成员函数，当然就可以在其中使用所在类的保护成员及私有成员。
三、在何时使用 inline 函数：
1、首先，你可以使用 inline 函数完全取代表达式形式的宏定义。
2、另外要注意，内联函数一般只会用在函数内容非常简单的时候，这是因为，内联函数的代码会在任何调用它的地方展开，如果函数太复杂，代码膨胀带来的恶果很可能会大于效率的提高带来的益处。内联函数最重要的使用地方是用于类的存取函数。
四、如何使用类的 inline 函数：
inline float converter(float dollars); //只需在函数的开头加上关键字 inline

注意：inline 说明对编译器来说只是一种建议，编译器可以选择忽略这个建议。比如，你将一个长达 1000 多行的函数指定为 inline，编译器就会忽略这个 inline，将这个函数还原成普通函数。

从 inline 的原理，我们可以看出，inline 的原理，是用空间换取时间的做法，是以代码膨胀（复制）为代价，仅仅省去了函数调用的开销，从而提高函数的执行效率。如果执行函数体内代码的时间，相比于函数调用的开销较大，那么效率的收获会很少。所以，如果函数体代码过长或者函数体重有循环语句，if 语句或 switch 语句或递归时，不宜用内联

.关键字 inline 必须与函数定义体放在一起才能使函数成为内联，仅将 inline 放在函数声明前面不起任何作用。内联函数调用前必须声明。

```
//以下代码不能成为内联函数
inline void Foo(int x, int y); // inline 仅与函数声明放在一起
void Foo(int x, int y)
{
    ...
}
//以下代码可以成为内联函数
void Foo(int x, int y);
inline void Foo(int x, int y) // inline 与函数定义体放在一起
{
    ...
}
```

所以说，inline 是一种“用于实现的关键字”，而不是一种“用于声明的关键字”。
注意点：
非常短的函数适合于内联
在链接时函数体会插入到发生函数调用的地方
预处理 ➔ 编译 ➔ 汇编 ➔链接 ➔ 运行

2.5、函数重载:相同名称,不同参数(参数个数,类型,排列顺序)

编译器通过调用时参数的个数和类型以及顺序来确定调用重载函数的哪个定义，不能仅仅通过函数的返回值类型的不同实现函数重载。
只有对不同的数据集完成基本相同任务的函数才应重载
void display();
void display(const char*);
void display(int one, int two);
void display(float number);
优点:
1、不必使用不同的函数名
2、有助于理解和调试代码
3、易于维护代码

2.6、强制类型转换：
C 语言的强制类型转换有两种：
1、自动数据类型转换（bit->char->int->long->float;unsigned->signed）
2、强制数据类型转换
C++的强制类型转换相对 C 语言略有不同。
C 语言中的强制类型换成常常采用：int b; float a = (float)b;
C++中的强制类型转换常常采用：int b; float a = float(b); //这种方式类似于函数

```
#include <iostream>

using namespace std;

int main()

{

    char b=0xaa;

    int a=0xaaaaaaaa;

    printf("sizeof(a)=%x\r\n",sizeof(a));

    printf("sizeof(b)=%x\r\n",sizeof(b));

    printf("sizeof(char(a))=%x\r\n",sizeof(char(a)));

    printf("sizeof(int(b))=%x\r\n",sizeof(int(b)));

    return 0;

}
```

注意：强制类型转换不改变任何变量的数据类型，它只是改用另一种类型读取这个变量，并不是改变变量本身的类型。

电脑变量赋值都是寻址的，int 占 4 字节，char 就一个字节，如果允许不强制赋值，按照字节对齐赋值，那么这个 char 类型数据就会存在 int 类型的最高字节，而且后三个字节

读的是未知数据，那这个数就增大了几百万倍，所以强制类型转换是必须的，但他不能改变变量的的类型，编程语言也没这个功能，因为可以这样的话，只能造成混乱，需要给变量重新分配内存，因为原内存扩展长度，腿就伸到别人家了。

2.7、变量的说明可以放在程序任何位置

例如：

```
for(int i=0;i<100;i++);
```

注意：

标准 C++规定的是 i 的作用域是此 for 语句

但 VC6.0 对标准支持不是很好，默认把它扩大到文件结束。

我们在用时不要在 for 循环外再调用循环内定义的变量，也符合 c++要用的时候再定义的思想

```
#include <iostream>
using namespace std;
int main()
{
    for(int i=0;i<1;i++)
    {
        cout<<i;
    }
    cout<<i;                    //不要这样用，有的编译器会报错
    return 0;
}
```

2.8、const 关键字：

c++中利用 const 来定义只读量，与 C 语言中的#define 类似,用 const 修饰的不能修改,常用来定义一个符号常量。

```
const   int   Number = 1;
```

例子：

```
#include <iostream>
using namespace std;
const int Number = 1,n = 10;          //定义只读变量
int main(void)
{
    /*常量指针*/
    int a ;
    const int * p;                    //常量指针 （指针 p 所指向的常量不能修改，但是可以改变指针 p 的指向）
    p = &Number;
    p = &n;
    //*p = 5;     //错误
    //n = 5;      //错误
```

```
    /*指针常量*/
    int n1 = 3;
    int const n2 = 5;                    //注意（int const  、const int 的写法是一样的）
    int * const pn = &n1;                //指针常量（指针 pn 所指向的变量是可以更新的，不可更新的是指针 pn 所指向的方向）
    //pn = &n2;     //错误
    //n2 = 6;       //错误
    *pn = 6;

    return 0;
}
```

使用 const 修饰指针时，由于 const 的位置不同而含义不同。

（1） 若声明指针常量，则指针 pn 所指向的变量是可以更新的，不可更新的是指针 pn 所指向的方向。

（2） 若声明为指向常量指针，则指针 p 所指向的常量不能修改，但是可以改变指针 p 的指向

2.9、new delete(delete[])

new 运算符动态开辟内存

例程：

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

const int Number = 1,n = 10;          //定义只读变量

int main(void)
{
    int *a=new int[100];              //[n]申请内存大小
    int *p=new int(1);                //（n）申请一块内存并初始化为 n

    cout<<*p<<endl;

   delete p;                          //delete    删除指针 p 指向的内存
    delete[] a;                       //delete[]  删除一个数组.
    return 0;
}
```

3、码段、数据段、BSS 段讲解

程序的运行中内存布局方式:
1> 代码段 2> 数据段 > bss 段 3> 堆区 4> 栈区

3.1 代码段

代码段：代码段（codesegment/textsegment）通常是指用来存放程序执行代码的一块内存区域。**这部分区域的大小在程序运行前就已经确定，并且内存区域通常属于只读。**在执行程序的过程中，CPU 的程序计数器 PC 指向代码段的每一条机器代码，并由处理器依次运行

3.2 只读数据段（RO data）

程序中使用的一些不会被更改的数据，所在的内存区域也是属于只读。比如：**只读全局量、const** char a[100]={"ABCDEF"};大小为 100 字节的只读数据区；只读局部量 const int a = 11；**程序中使用的常量** printf("hello world\n");其中包括字符串常量，编译器就会把它放到只读数据区。

3.3 数据段

数据段（datasegment）通常是指用来存放程序中已初始化的全局变量的一块内存区域。也被称为读写数据段或是已初始化数据段。这部分区域的大小在程序运行前就已经确定，属于静态区域，但是具有可写的特点。

例如：**已初始化的全局变量** **已初始化的局部静态变量（static）**

3.4 BSS 段

BSS 段（bsssegment）通常是指用来存放程序中未初始化数据的一块内存区域。BSS 是英文 BlockStartedbySymbol 的简称。BSS 段属于静态内存分配。它只会在目标文件中被标识，该段将会在程序运行时产生，在运行之前不需要占用存储器空间。

3.5 堆（heap）

堆是用于存放进程运行中被动态分配的内存段，它的大小并不固定，可动态扩张或缩减。当进程调用 malloc 等函数分配内存时，新分配的内存就被动态添加到堆上（堆被扩张）；当利用 free 等函数释放内存时，被释放的内存从堆中被剔除（堆被缩减）一般由程序员分配和释放

3.6 栈(stack)

栈又称堆栈，**是用户存放程序临时创建的局部变量**，也就是说我们函数括弧“{}”中定义的变量（但不包括 static 声明的变量，static 意味着在数据段中存放变量）。除此以外，在函数被调用时，其参数也会被压入发起调用的进程栈中，并且待到调用结束后，函数的返回值也会被存放回栈中。由于栈的先进后出特点，所以栈特别方便用来保存/恢复调用现场。从这个意义上讲，我们可以把堆栈看成一个寄存、交换临时数据的内存区。一般由编译器自动分配释放。

注意：代码段、只读数据段、读写数据段、未初始化数据段属于静态区域，而堆和栈属于动态区域。**代码段、只读数据段和读写数据段将在连接之后产生，未初始化数据段将在程序初始化的时候开辟，而堆和栈将在程序的运行中分配和释放。**

所以 C 语言的程序在编译和连接后，将生成代码段（Code）只读数据段（RO-data）和读写数据段（对于 keil：RW-data 和 ZI-data））,在初始化的时候生成 BSS 段,运行时动态形成堆区（Heap）和栈区（Stack）

例如 keil 软件生成可执行文件提示
linking...
Program Size: Code=16800 RO-data=548 RW-data=32 ZI-data=21856

作业

复习今天讲的以下内容，通过练习的方式记住各个知识点如何运用

（1） 命名空间 ：using namespace std;
（2） 引用:&
（3） 函数的默认参数:
（4） 内联函数:inline
（5） 函数的重载:
（6） 输入输出流:cout cin <iostream>
（7） 强制类型转换 b=(int)a b=int(a)
（8） 变量的说明可以放在程序任何位置:for(int i=0;i<100;i++);
（9） const 关键字:常量指针 int const *p、指针常量 const int *p
（10） new delete(delete[]):区别 malloc free
（11） 码段、数据段、BSS 段讲解