

知识点1【c++的概述】（了解）

知识点2【第一个c++程序】（了解）

1、安装Qtcreator集成开发环境

知识点3【c++的初始】（了解）

1、第一个c++程序

2、c++面向对象的三大特性（重要）

知识点4【::作用域运算符】（了解）

1、可以优先使用全局变量

知识点5【命名空间】（了解）

1、创建一个命名空间

2、命名空间 只能定义在全局（重要）

3、命名空间可以嵌套

4、可以随时将新的成员加入命名空间

5、命名空间中 函数的声明和实现分开

6、无名命名空间

7、命名空间取别名

8、使用using申明命名空间中的某几个成员 可用

using 申明某个成员 容易造成名字冲突

using 申明制定成员函数 遇到函数重载

9、using申明整个命名空间 可以直接通过成员名 使用

知识点6【类型增强】（了解）

1、全局变量检测增强

2、c++的函数形参必须有类型

3、如果函数没有参数，建议写void

4、更严格的类型转换

5、结构体类型增强（重要）

6、c++新增bool类型

7、三目运算符增强

8、左值和右值（c++ c共有）

知识点7【c++的const】（重要）

1、c++和c中的const都是修饰变量为 只读。

2、c语言 严格准许 const修饰的是只读变量。

3、c++的const 会对变量 优化

1、如果以常量 初始化const修饰的变量 编译器会将变量的值 放入符号常量表中，不会立即给变量开辟空间

2、只有当对a 取地址时 编译器才会给a开辟空间（只读变量）

3、如果以变量 初始化const修饰的只读变量，没有符号常量表，立即开辟空间

4、如果以const修饰的是自定义类型的变量 也不会有符号常量表，立即开辟空间

5、c++中尽量使用const代替define

知识点8【引用】（重要）

1、引用的定义

案例1:给普通变量取别名

案例2：给数组取别名

案例3：给指针变量取别名

案例4：给函数取别名

注意：

2、引用作为函数的参数

3、引用作为函数的返回值类型（链式操作）

4、常引用

5、引用的本质：常量指针

知识点9【内联函数】（了解）

1、宏函数和内联函数的区别（重要背）

2、内联函数的注意事项

知识点10【函数重载】（重要）

1、函数重载的条件：（背）

2、函数重载的底层实现原理：

知识点11【函数的缺省参数】（重要）

注意点：

案例：一下默认参数正确的是

知识点12【占位参数】（了解）重载++运算符

知识点1【c++的概述】（了解）

c++的编程思想：面向对象、泛型编程。

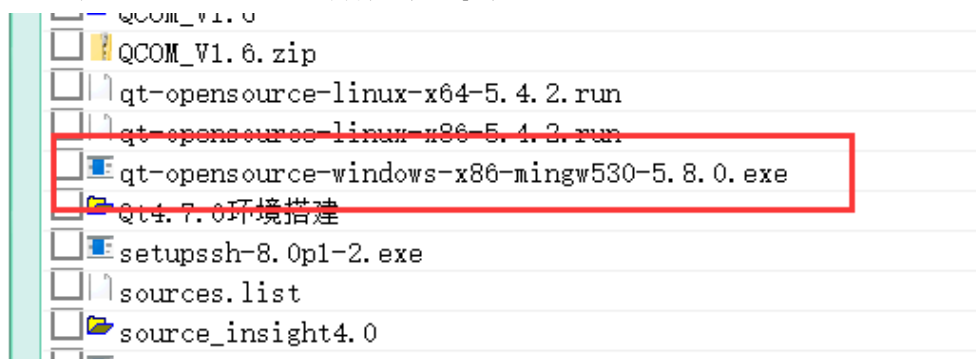
面向过程（c）：通过分析出解决问题所需要的步骤，然后用函数把这些步骤一步一步实现,并调用。

面向对象(c++)：算法与数据结构被看做是一个整体(对象)，程序=对象+对象+对象+对象

c++标准：c++98、c++11

知识点2【第一个c++程序】（了解）

1、安装Qtcreator集成开发环境



文档：[windows安装qt_creator5.8.0.note](#)

链接：[http://note.youdao.com/noteshare?](http://note.youdao.com/noteshare?id=cd415b0f7e7922b9c09d08a693f448b9&sub=FD1C9DE02A9E4365B75968807A14D906)

[id=cd415b0f7e7922b9c09d08a693f448b9&sub=FD1C9DE02A9E4365B75968807A14D906](http://note.youdao.com/noteshare?id=cd415b0f7e7922b9c09d08a693f448b9&sub=FD1C9DE02A9E4365B75968807A14D906)

知识点3【c++的初始】（了解）

1、第一个c++程序

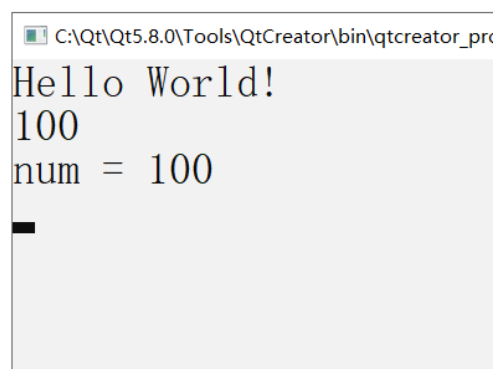
```
#include <iostream> //标准输入输出流

//使用std命名空间
using namespace std;

int main(int argc, char *argv[])
{
    //cout输出设备（终端） endl 换行符
    cout << "Hello World!" << endl;

    //
    int num = 0;
    //cin输入设备（键盘）
    cin >> num;
    cout<<"num = "<< num<<endl;
    //cin>>num1>>num2;

    return 0; //返回值 以及 结束函数
}
```



C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator_pr
Hello World!
100
num = 100

2、c++面向对象的三大特性（重要）

封装:将相同属性的数据和方法封装在一起，加权限区分，用户只能借助公共方法操作 私有数据。

继承:体现在类和类之间的关系，如果A类继承于B类，那么A类直接拥有B类的数据和方法。

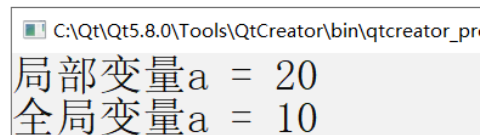
多态:一个接口（函数），多种功能。

知识点4【::作用域运算符】（了解）

::解决归属问题（谁是谁的谁）

1、可以优先使用全局变量

```
using namespace std;
int a = 10; //全局区
void test01()
{
    int a = 20; //栈
    cout<<"局部变量a = "<<a<<endl;
    cout<<"全局变量a = "<<::a<<endl;
}
```



C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator_pr
局部变量a = 20
全局变量a = 10

知识点5【命名空间】（了解）

使用关键字namespace，控制标名称的作用域。

命名空间的本质：对符号常量、变量、函数、结构、枚举、类和对象等等进行封装

1、创建一个命名空间

```
namespace A {  
    //定义变量、函数、类型等  
    int data = 10;  
}  
  
namespace B {  
    int data = 20;  
}  
void test02()  
{  
    cout<<"A::data = "<<A::data<<endl;  
    cout<<"B::data = "<<B::data<<endl;  
}
```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator.

A::data = 10
B::data = 20

2、命名空间 只能定义在全局（重要）

```
13 void test02()  
14 {  
15     namespace A {  
16         //定义变量、函数、类型等  
17         int data = 10;  
18     }  
19  
20     namespace B {  
21         int data = 20;  
22     }  
23     cout<<"A::data = "<<A::data<<endl;  
24     cout<<"B::data = "<<B::data<<endl;  
25 }  
26
```

局部命名空间 错误

3、命名空间可以嵌套

```
namespace A {  
    //定义变量、函数、类型等  
    int data = 10;  
    namespace B {命名空间 嵌套 命名空间  
        int data = 20;  
    }  
}  
  
void test02()  
{  
    cout<<"A::data = "<<A::data<<endl;  
    cout<<"A::B::data = "<<A::B::data<<endl;  
}
```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator_process_s

A::data = 10
A::B::data = 20

4、可以随时将新的成员加入命名空间

```

namespace C {
    int data= 10;
}
//1000行代码
namespace C {
    void func(void)
    {
        cout<<"C中的func函数"<<endl;
    }
}
void test03()
{
    cout<<"data = "<<C::data<<endl;
    C::func();
}

```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcra

data = 10
C中的func函数

5、命名空间中 函数的声明和实现分开

```

namespace D {
    //在命名空间内申明:说明fun01 fun02属于D命名空间
    void fun01();
    void fun02();
}

//命名空间的成员函数 外部实现 必须加作用域
void D::fun01()
{
    cout<<"D中的fun01"<<endl;
}
void D::fun02()
{
    cout<<"D中的fun02"<<endl;
}

void test04()
{
    D::fun01();
    D::fun02();
}

```

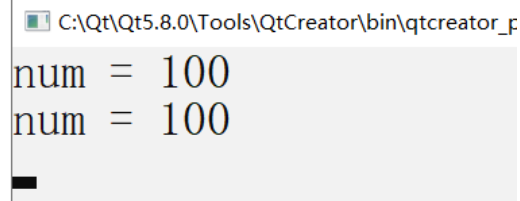
C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcra

D中的fun01
D中的fun02

6、无名命名空间

无名命名空间 只能在 本源文件使用。

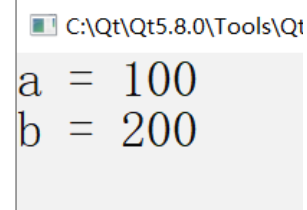
```
//static int num = 100;只能在当前源文件使用
namespace{
    int num = 100;
}
void test05()
{
    cout<<"num = "<<num<<endl;
    cout<<"num = "<<::num<<endl;
}
```



```
C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator_p
num = 100
num = 100
```

7、命名空间取别名

```
namespace veryLongName {
    int a =100;
    int b = 200;
}
//给veryLongName取别名shortName
namespace shortName = veryLongName;
void test06()
{
    cout<<"a = "<<shortName::a<<endl;
    cout<<"b = "<<veryLongName::b<<endl;
}
```

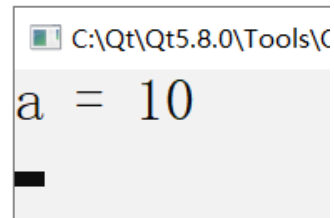


```
C:\Qt\Qt5.8.0\Tools\Qt
a = 100
b = 200
```

8、使用using申明命名空间中的某几个成员 可用

```
namespace A {
    int a = 10;
    int b = 20;
    int c = 30;
}

void test01()
{
    //只能说明A::a可以直接使用变量名a访问
    using A::a;
    cout<<"a = "<<a<<endl;
}
```



```
C:\Qt\Qt5.8.0\Tools\Qt
a = 10
```

using 申明某个成员 容易造成名字冲突

```

5  namespace A {
6      int a = 10;
7      int b = 20;
8      int c = 30;
9  }
10
11  void test01()
12  {
13      //只能说明A::a可以直接使用变量名a访问
14      using A::a; //会和普通变量产生名字冲突
15      int a = 100;
16
17      cout<<"a = "<<a<<endl;
18  }
19

```

using 申明制定成员函数 遇到函数重载

```

1  namespace B {
2      void fun01(void)
3      {
4          cout<<"B中的fun01 void"<<endl;
5      }
6      void fun01(int a)
7      {
8          cout<<"B中的fun01 int"<<endl;
9      }
10     void fun01(int a, int b)
11     {
12         cout<<"B中的fun01 int int"<<endl;
13     }
14 }
15 void test02()
16 {
17     //函数重载 命名空间的所有同名函数 都被申明 可用
18     using B::fun01;
19
20     fun01();
21     fun01(10);
22     fun01(10,20);
23 }

```


B中的fun01 void
B中的fun01 int
B中的fun01 int int

9、using申明整个命名空间 可以直接通过成员名 使用

```
namespace C {  
    int a = 10;  
    int b = 20;  
    int c = 30;  
}  
//int a = 100; //冲突  
void test03()  
{  
    //使用整个命名空间  
    //不会和同名局部变量冲突 会和全局同名变量冲突  
    using namespace C;  
    int a = 100;  
    //变量名a 先寻找局部变量a,如果找不到 再到申明的命名空间中找  
    std::cout<<"a = "<<a<<std::endl;  
    cout<<"b = "<<b<<endl;  
    cout<<"c = "<<c<<endl;  
}
```

C:\Qt\Qt5.8.0\Tools\QtCreator\b
a = 100
b = 20
c = 30

加作用域解决冲突

```
namespace C {  
    int a = 10;  
    int b = 20;  
    int c = 30;  
}  
int a = 200;  
void test03()  
{  
    using namespace C;  
    int a = 100;  
  
    std::cout<<"命名空间中的a = "<<C::a<<std::endl;  
    cout<<"全部变量a = "<<::a<<endl;  
    cout<<"局部变量a = "<<a<<endl;  
}
```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtc
命名空间中的a = 10
全部变量a = 200
局部变量a = 100

知识点6【类型增强】（了解）

1、全局变量检测增强

c 语言代码: ←

```
int a = 10; //赋值, 当做定义↓
int a; //没有赋值, 当做声明↓
↓
int main(){↓
    printf("a:%d\n",a);↓
    return EXIT_SUCCESS;↓
}←
```

此代码在 c++ 下编译失败,在 c 下编译通过.←

2、c++的函数形参必须有类型

c语言: 允许函数形参无类型 (可以传任意参数)

```
1 //i没有写类型, 可以是任意类型
2 int fun1(i){
3     printf("%d\n", i);
4     return 0;
5 }
6 //i没有写类型, 可以是任意类型
7 int fun2(i){
8     printf("%s\n", i);
9     return 0;
10 }
```

c++不允许

3、如果函数没有参数, 建议写void

c语言: 可以 c++不可以

```
1 //没有写参数, 代表可以传任何类型的实参
2 int fun3(){
3     printf("fun3333333333333333\n");
4     return 0;
5 }
6
7
8 int main(int argc, char *argv[])
9 {
10     fun3(10);
11     fun3(10, 20);
```

```

12  fun3("hello");
13
14
15  return 0;
16  }

```

4、更严格的类型转换

```

enum COLOR{ GREEN, RED, YELLOW };
int main(int argc, char *argv[])
{
    enum COLOR num;
    num = 100;
    return 0;
}

```

枚举变量c语言允许赋其他int类型的值

c++不允许

5、结构体类型增强（重要）

```

struct stu
{
    int num;
    char name[32];

    //c++允许函数 作为结构体的成员
    void fun()
    {
        cout<<"stu 中的fun"<<endl;
    }
};

void test04()
{
    //c++可以不写struct c语言必须写
    stu lucy={100,"lucy"};
    cout<<"num = "<<lucy.num<<" "<<"name="<<lucy.name<<endl;
    lucy.fun();
}

```

```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreato
num = 100 name=lucy
stu 中的fun

```

6、c++新增bool类型

bool类型拥有两个值， true false

```

void test04()
{
    bool bl;
    printf("%d\n", sizeof(bool)); //占1 字节
    bl = false;
    if(bl)
    {
        cout<<"为真"<<endl;
    }
    else
    {
        cout<<"为假"<<endl;
    }
}

```



C:\Qt\Qt5.8.0\Tools\QtC

1
为假

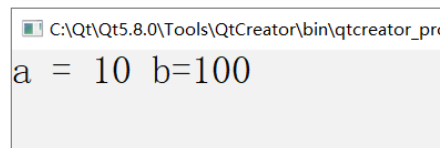
7、三目运算符增强

```

void test04()
{
    int a= 10;
    int b = 20;

    //c++三目运算符 返回值是引用 而c语言返回的是值
    a>b?a:b = 100; //c++可以赋值 c语言不可以赋值
    cout<<"a = "<<a<<" "<<"b="<<b<<endl;
}

```



C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator_pr

a = 10 b=100

8、左值和右值 (c++ c共有)

左值：能放在=左边，（能被赋值的值 就是左值）

右值：只能放在=右边 （不能被赋值的值 就是右值）

知识点7 【c++的const】（重要）

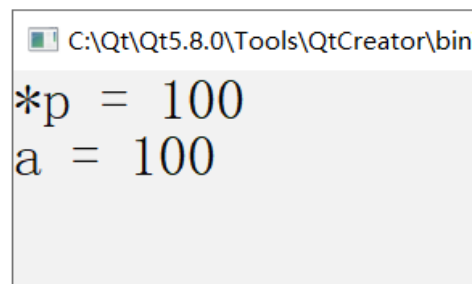
1、c++和c中的const都是修饰变量为 只读。

2、c语言 严格准许 const修饰的是只读变量。

```

int main(int argc, char *argv[])
{
    const int a= 10;
    int *p = (int *)&a;
    *p = 100;
    printf("*p = %d\n", *p);
    printf("a = %d\n", a);
    return 0;
}

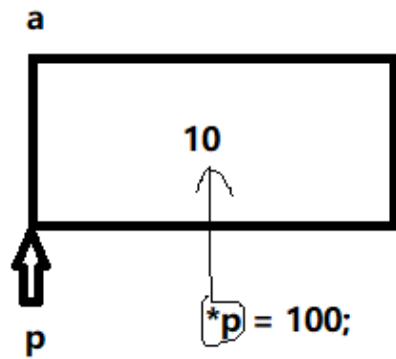
```



C:\Qt\Qt5.8.0\Tools\QtCreator\bin

*p = 100
a = 100

`const int a = 10;`



3、c++的const 会对变量 优化

1、如果以**常量** 初始化const修饰的变量 编译器会将变量的**值** 放入符号常量表中，不会**立即**给变量开辟空间

符号常量表

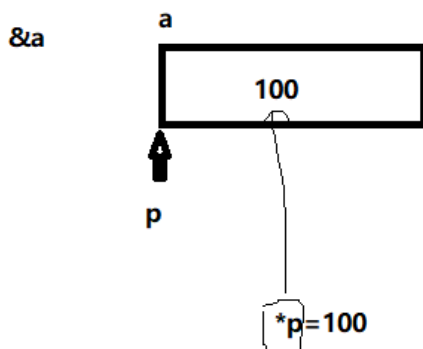
`const int a = 10;`

符号	值
a	10

2、只有当对a **取地址**时 编译器才会给a开辟空间（只读变量）

符号常量表

`const int a = 10;`



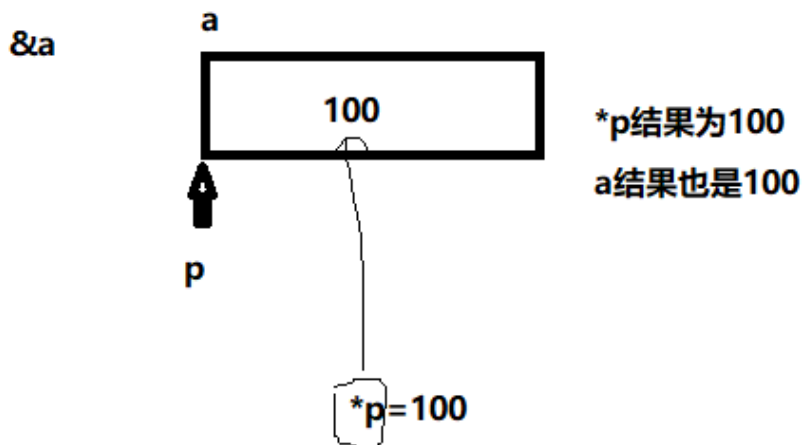
符号	值
a	10

通过指针变量p 访问空间内容*p 取的是空间的值
通过变量名a 访问的是符号常量表中的值

3、如果以**变量** 初始化const修饰的只读变量，**没有**符号常量表，立即开辟空间

```
int b = 10;
```

```
const int a = b;
```



4、如果以const修饰的是自定义类型的变量 也不会有符号常量表，立即开辟空间

```
struct STU
{
    int num;
    char name[32];
};
//#include<string.h>
#include<cstring>
void test04()
{
    const STU l Lucy={100,"lucy"};
    STU *p = (STU *)&lucy;
    p->num = 200;
    strcpy(p->name, "bob");

    cout<<p->num<<" "<<p->name<<endl;
    cout<<lucy.num<<" "<<lucy.name<<endl;
}
```

```
C:\Qt\Qt5.8.0\Tools\QtC
200 bob
200 bob
```

5、c++中尽量使用const代替define

```
1 #define A 10
2 const int A=10;
```

- 1、const有类型，可进行编译器类型安全检查。#define无类型,不可进行类型检查
- 2、const有作用域，而#define不重视作用域，宏不能作为命名空间、结构体、类的成员，而const可以

```

namespace A {
//    #define N 10 不能 作为命名空间、结构体、类的成员
    const int N=10;//可以 作为命名空间、结构体、类的成员
}
void test05()
{
    cout<<A::N<<endl;
}

```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator_process_stub.exe

10

知识点8【引用】（重要）

1、引用的定义

引用的本质：就是给变量名取个**别名**。

引用定义的步骤：

- 1 &别名
- 2 给哪个变量取别名 就定义该变量
- 3 从上往下整体替换

```
int a = 10;
```

a b代表同一空间内容

a b



```
int &b = a;
```

案例1:给普通变量取别名

```

void test01()
{
    int a = 10;
    //需求: 给变量a 取个别名叫b
    //在定义的时候 &修饰变量为引用    b就是a的别名
    //系统不会为引用 开辟独立空间
    int &b = a;//引用必须初始化

    cout<<"a = "<<a<<endl;
    cout<<"b = "<<b<<endl;

    b = 100;
    cout<<"a = "<<a<<endl;
    cout<<"b = "<<b<<endl;

    cout<<"&a = "<<&a<<endl;
    cout<<"&b = "<<&b<<endl;
}

```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator_process_stub.exe

```

a = 10
b = 10
a = 100
b = 100
&a = 0x61fe88
&b = 0x61fe88

```

案例2: 给数组取别名

```

1 void test02()
2 {
3     int arr[5]={10,20,30,40,50};
4     int n = sizeof(arr)/sizeof(arr[0]);
5
6     int (&myArr)[5] = arr;
7     int i=0;
8     for(i=0;i<n;i++)
9     {
10         cout<<myArr[i]<<" ";
11     }
12     cout<<endl;
13 }

```

C:\Qt\Qt5.8.0\Tools\QtCreator\b

10 20 30 40 50

案例3：给指针变量取别名

```

✓ void test03()
{
    int num = 10;
    int *p = &num;

    int* &my_p = p;
    cout<<"*p = "<<*p<<endl;
    cout<<"*my_p = "<<*my_p<<endl;
}

```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcrea

*p = 10
*my_p = 10

案例4：给函数取别名

```

void fun01(void)
{
    cout<<"fun01"<<endl;
}

```

```

void test04()
{
    void (&my_fun)(void) = fun01;
    my_fun();
}

```

C:\Qt\Qt5.8.0\Tools\C

fun01

注意：


```

void test05()
{
    int a = 10;
    int b = 20;

    //a的别名为num
    int &num = a;

    //将b的值赋值给num 也就是a的空间
    num = b;
    cout<<"a = "<<a<<endl;
    cout<<"num = "<<num<<endl;
}

```

```

C:\Qt\Qt5.8.0\Tools\QtCreator
a = 20
num = 20

```

2、引用作为函数的参数

函数内部可以通过 **引用** 操作外部变量。

```

✓ void swap01(int *p1, int *p2)
{
    int tmp = *p1;
    *p1 = *p2;
    *p2 = tmp;
}
✓ void swap02(int &x, int &y)//int &x=a; int &y=b;
{
    int tmp = x;
    x = y;
    y=tmp;
}
✓ void test06()
{
    int a = 10;
    int b = 20;
    cout<<"a = "<<a<<" "<<" b = "<<b<<endl;
    //swap01(&a, &b);
    swap02(a, b);
    cout<<"a = "<<a<<" "<<" b = "<<b<<endl;
}

```

```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator_process_stub.exe
a = 10 , b = 20
a = 20 , b = 10

```

节约空间

```

struct STU
{
    int num;
    char name[32];
};
void printStu01(STU tmp)//tmp占 空间
{
    cout<<tmp.num<<" "<<tmp.name<<endl;
}
void printStu02(STU &tmp)//tmp不占 空间
{
    cout<<tmp.num<<" "<<tmp.name<<endl;
}
void test07()
{
    STU lucy={100, "lucy"};
    printStu02(lucy);
}

```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator_proces

100 lucy

3、引用作为函数的返回值类型（链式操作）

```

int& getData(void)
{
    static int num = 100;

    //不要返回局部变量的引用
    return num;//返回num 外界就是给num取别名
}

void test08()
{
    //b是num的别名
    int &b = getData();
    cout<<"b = "<<b<<endl;
}

```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator_pro

b = 100

链式操作：

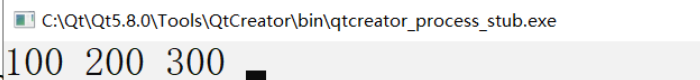
```

struct stu
{
    stu& printStu(stu &ob, int value)
    {
        cout<<value<<" ";
        return ob;
    }
};

void test09()
{
    stu ob1;
    ob1.printStu(ob1, 100).printStu(ob1, 200).printStu(ob1, 300);
}

int main(int argc, char *a

```



4、常引用

给常量取别名

```

1 void test10()
2 {
3     //int &a = 10;//err
4     const int &a = 10;//a就是10的别名
5     //a = 100;//err
6     cout<<a<<endl;
7 }

```

不能通过常引用 修改 内容。

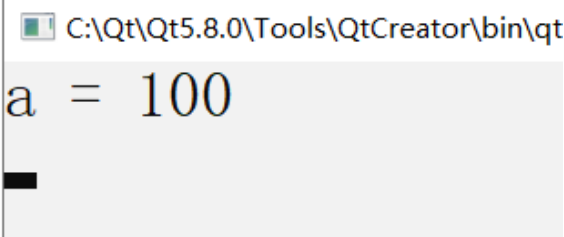
常引用 作为函数的参数：防止函数内部修改外部的值。

```

void printInt(const int &a)
{
    //a = 200;//err
    cout<<"a = "<<a<<endl;
}

void test11()
{
    int num = 100;
    printInt(num);
}

```



5、引用的本质：常量指针

```

1 int a=10;

```

```
2 int &b = a; //b为a的别名 int * const b = &a;
3 b = 100; //a的值为100 *b = 100;
```

知识点9 【内联函数】（了解）

内联函数：在编译阶段 将内联函数中的函数体 替换函数调用处。避免函数调用时的开销。

内联函数 必须在定义的时候 使用关键字inline修饰，不能在声明的时候使用inline

```
1 //函数声明的时候 不要使用inline
2 int my_add(int x, int y);
3 void test01()
4 {
5     cout<<my_add(100,200)<<endl;
6 }
7 //内联函数 在定义的时候使用inline
8 inline int my_add(int x, int y)
9 {
10     return x+y;
11 }
```

1、宏函数和内联函数的区别（重要背）

宏函数和内联函数 都会在适当的位置 进行展开 避免函数调用开销。

宏函数的参数没有类型，不能保证参数的完整性。

内联函数的参数有类型 能保证参数的完整性。

宏函数在预处理阶段展开

内联函数在编译阶段展开

宏函数没有作用域的限制，不能作为命名空间、结构体、类的成员

内联函数有作用域的限制，能作为命名空间、结构体、类的成员

2、内联函数的注意事项

在内联函数定义的时候加inline修饰

类中的成员函数 默认都是内联函数（不加inline 也是内联函数）

有时候 就算加上inline也不一定是内联函数（内联函数条件）

不能存在任何形式的循环语句

不能存在过多的条件判断语句

函数体不能过于庞大

不能对函数取地址

有时候不加inline修饰 也有可能是内联函数。

内不内联 由编译器决定。

知识点10 【函数重载】（重要）

函数重载 是c++的多态的特性（静态多态）。

函数重载：用同一个函数名 代表不同的函数功能。

1、函数重载的条件：（背）

同一作用域，函数的参数**类型**、**个数**、**顺序**不同 都可以重载。（返回值类型**不能**作为重载的条件）

```
1 void printFun(int a)
2 {
3     cout<<"int"<<endl;
4 }
5 void printFun(int a, char b)
6 {
7     cout<<"int char"<<endl;
8 }
9 void printFun(char a, int b)
10 {
11     cout<<"char int"<<endl;
12 }
13 void printFun(char a)
14 {
15     cout<<"char"<<endl;
16 }
17
18 void test02()
19 {
20     printFun(10);
21     printFun(10, 'a');
22     printFun('a', 10);
23     printFun('a');
24 }
```

```
int
int char
char int
char
```

c++中 不能直接将函数名作为函数的入口地址 (为啥呢?)

函数名和参数 共同决定函数的入口地址

2、函数重载的底层实现原理:

```
void func(){}↓  
void func(int x){}↓  
void func(int x,char y){}←
```

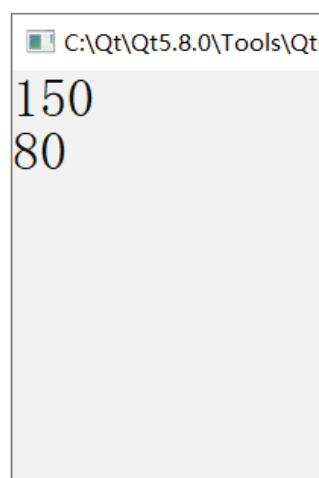
以上三个函数在 linux 下生成的编译之后的函数名为:←

```
_Z4funcv //v 代表 void,无参数↓  
_Z4funci //i 代表参数为 int 类型↓  
_Z4funcic //i 代表第一个参数为 int 类型, 第二个参数为 char 类型←
```

知识点11 【函数的缺省参数】 (重要)

在函数声明处 给函数参数一个默认的值, 如果函数调用处, 用户没用传实参, 编译器就可以使用这个默认的值。

```
//建议 在函数声明处 给出缺省值  
int my_sub(int x=100, int y=20);  
void test03()  
{  
    cout<<my_sub(200, 50)<<endl;  
    cout<<my_sub()<<endl;  
}  
int my_sub(int x, int y)  
{  
    return x-y;  
}
```



```
C:\Qt\Qt5.8.0\Tools\Qt  
150  
80
```

注意点:

如果函数的某个参数设置为默认参数, 那么这个参数的右边的所有参数 都必须是默认参数。

案例: 一下默认参数正确的是

```
1 int func(int a, int b, int c=10); //正确  
2 int func(int a, int b=20, int c); //错误 c必须默认参数  
3 int func(int a=10, int b, int c); //错误 b c必须默认参数  
4 int func(int a, int b=10, int c=20); //正确  
5 int func(int a=10, int b, int c=20); //错误 b必须默认参数  
6 int func(int a=10, int b=20, int c=20); //正确
```

知识点12 【占位参数】 (了解) 重载++运算符

```

void myFun(int x)
{
    cout<<"A:int x="<<x<<endl;
}
//没有形参名的形参 叫占位参数
void myFun(int x, int)
{
    cout<<"B:int x="<<x<<endl;
}
void test04()
{
    myFun(10);
    myFun(10, 20);
}

```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator_proc

```

A:int x=10
B:int x=10

```

占位参数 也可以是缺省参数（默认参数）

```

//没有形参名的形参 叫占位参数
void myFun(int x, int=0)
{
    cout<<"B:int x="<<x<<endl;
}
void test04()
{
    myFun(10);
    myFun(10, 20);
}

```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator_proc

```

B:int x=10
B:int x=10

```

默认参数和函数重载同时出现 一定要注意二义性

```
57 void myFun(int x)
58 {
59     cout<<"A:int x="<<x<<endl;
60 }
61
62 void myFun(int x, int y=10)
63 {
64     cout<<"B:int x="<<x<<" y="<<y<<endl;
65 }
66 void test04()
67 {
68     myFun(10);
69     myFun(10, 20);
70 }
```

都可识别 产生二义性