Secure Coding

4 FUNCTIONS

4.1 Introduction

- Definitions of Functions

Function syntax:

```
type-of-return function-name(argument-list)
{
    //body
}
```

The argument list is a way for functions to communicate with each other by passing information.

4.1 Introduction

The argument list can contain 0, 1, or more arguments, separated by commas, depending on the function.

Example:

```
int  cube ( int  n )  //heading
{
   return  n * n * n ; //body
}
```

Remark:

A function can not define itself by nesting.

4.1 Introduction

- **Function return values**

A C++ function prototype must specify the return value type of the function.

To specify that no value is returned, use the void keyword.

4.1 Introduction

Example:Here are some complete function prototypes:

```
int f1(void);   // Returns an int, takes no arguments
int f2();       // Like f1() in C++ but not in Standard C!
float f3(float, int, char, double); // Returns a float
void f4(void);  // Takes no arguments, returns nothing
```

4.1 Introduction

To return a value from a function, you use the **return** statement.

Return exits the function back to the point right after the function call.

4.1 Introduction

You can have more than one return statement in a function definition:

Example:

```
char cfunc(int i)
{
  if(i == 0)
      return 'a';
  if(i == 1)
      return 'g';
  if(i == 5)
      return 'z';
  return 'c';
}
```

4.1 Introduction

- Function Calls

One function calls another by using the name of the called function next to ( ) enclosing an argument list.

Remark:

A function can call itself, too.

4.1 Introduction

1 When a function is called, temporary memory is set up.

2 Then the flow of control passes to the first statement in the function’s body. The called function’s body statements are executed until one of these occurs:

return statement (with or without a return value),

or,

closing brace of function body.

3 Then control goes back to where the function was called.

4.1 Introduction

Example:
```
#include <iostream.h>
double    power (double x, int n);
void main()
{
     double a=3;
     int b=5;
     cout <<  power(a,b) << endl;
}
double    power (double x, int n)
{
     double val = 1.0;
     while (n- -)
          val *= x;
     return(val);
}
```

Output:

```
3 to the power 5 is 243
```

4.1 Introduction

Example： n !

$$n! = \begin{cases} 1 & (n = 0) \\ n(n-1)! & (n > 0) \end{cases}$$

4.1 Introduction

```
#include <iostream.h>
long fac(int n)
{
  long f;
  if (n<0)
      cout<<"n<0,data error!"<<endl;
  else
     if (n==0)
              f=1;
     else
              f=fac(n-1)*n;
  return(f);
}
```

4.1 Introduction

```
void main()
{
    long fac(int n);
    int n;
    long y;

    cout<<"Enter a positive integer:";
    cin>>n;
    y=fac(n);
    cout<<n<<"!="<<y<<endl;
}
```

Output:

```
Enter a positive integer:8

8!=40320
```

4.1 Introduction

- Prototypes

In old (pre-Standard) C, you could call a function with any number or type of arguments and the compiler wouldn’t complain.

4.1 Introduction

Standard C and C++ use a feature called **function prototyping**.

With function prototyping, you must use a description of the types of arguments when declaring and defining a function.

4.1 Introduction

In a function prototype, the argument list contains the types of arguments that must be passed to the function and identifiers for the arguments.

The order and type of the arguments must match in the declaration, definition, and function call.

4.1 Introduction

Example:

```
char grade(int exam1, int exam2, float exam_weight)
{
   //grade's body
}
```

4.1 Introduction

In C++, prototypes are required and every function must be declared prior to being used. A return type must be specified.

Example:

```
int print(void)   //C style
{
   //print body
}
int print()   //C++style
{
   //print body
}
```

4.1 Introduction

- **The main Function**

Every program should contain a function called main.

Example:

```
int main()
{
   //main's body
   return 0;
}
```

4.1 Introduction

```
int main(int argc, char* argv[])
{
   //main's body
   return 0;
}
```

or

```
void main()
{
  //main's body
}
```

4.1 Introduction

- References

A reference, signaled by the ampersand &, provides an alternative name for storage.

Example:

```
int x;
int& ref = x;
```

Remark:

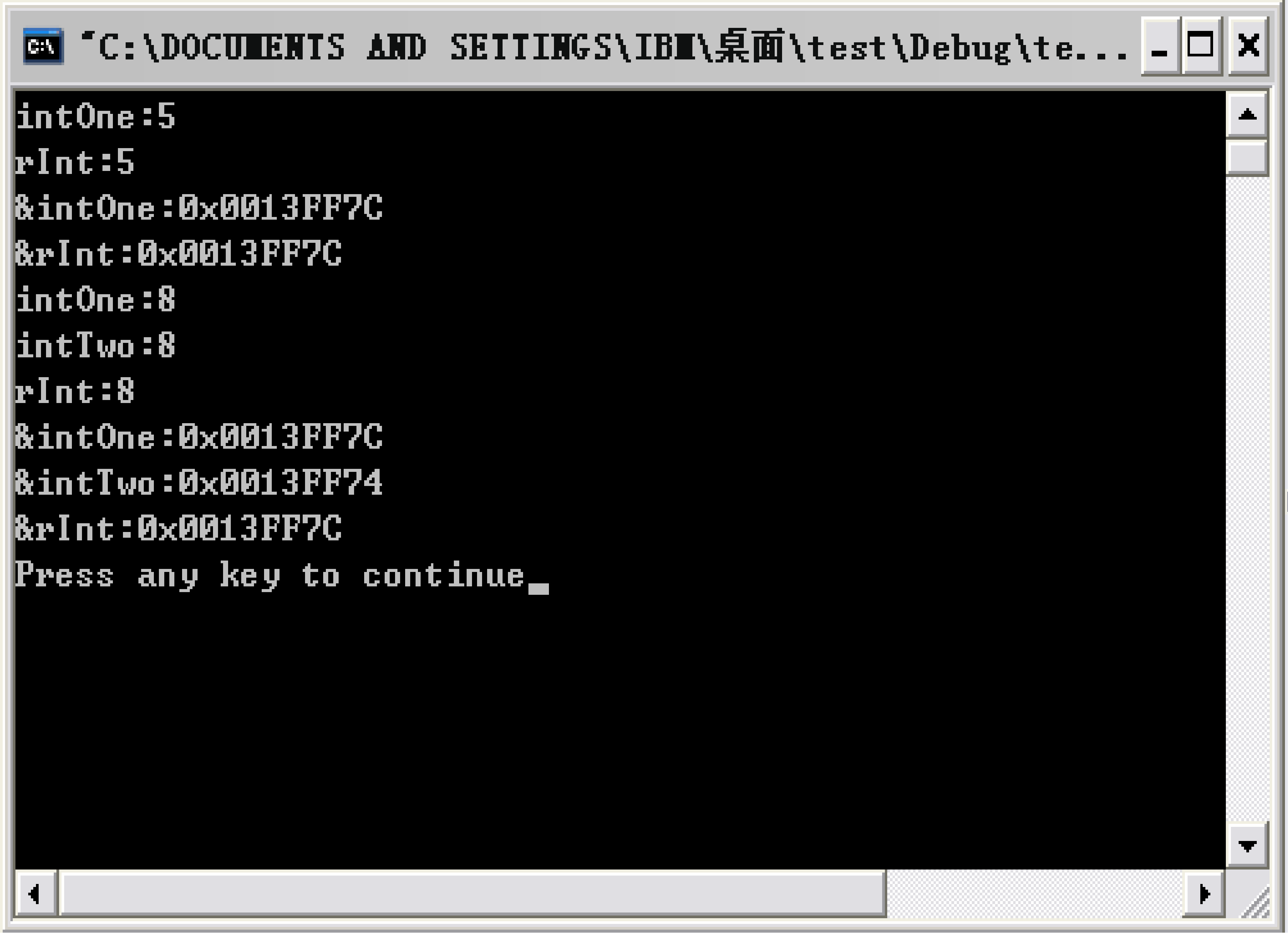
Both x and ref are stored in the same int storage.

```
#include <iostream.h>
void main()
{
  int intOne;
  int& rInt=intOne;

  intOne=5;
  cout << "intOne:"<<intOne<<endl;
  cout <<"rInt:"<<rInt<<endl;
  cout << "&intOne:"<<&intOne<<endl;
  cout <<"&rInt:"<<&rInt<<endl;

  int intTwo=8;
  rInt=intTwo;
  cout << "intOne:"<<intOne<<endl;
  cout << "intTwo:"<<intTwo<<endl;
  cout <<"rInt:"<<rInt<<endl;

  cout << "&intOne:"<<&intOne<<endl;
  cout << "&intTwo:"<<&intTwo<<endl;
  cout <<"&rInt:"<<&rInt<<endl;
}
```



```
intOne:5
rInt:5
&intOne:0x0013FF7C
&rInt:0x0013FF7C
intOne:8
intTwo:8
rInt:8
&intOne:0x0013FF7C
&intTwo:0x0013FF74
&rInt:0x0013FF7C
Press any key to continue
```

4.1 Introduction

- Call by Reference

If we designate a parameter as a reference parameter using the ampersand &, we obtain call by reference in which the reference parameter refers to the actual argument passed to the function and not to a copy of the argument.

4.1 Introduction

Example:

```
#include <iostream.h>
void swap(int a, int b)
{
    int t;
    t=a;
    a=b;
    b=t;
}
```

```
int main()
{
    int i=7, j=10;
    swap(i, j);
    cout<<"i="<<i<<endl;
    cout<<"j="<<j<<endl;
    return 0;
}
```

4.1 Introduction

Example:

```
#include <iostream.h>
void swap(int& a, int& b)
{
    int t;
    t=a;
    a=b;
    b=t;
}
```

```
int main()
{
    int i=7, j=10;
    swap(i, j);
    cout<<"i="<<i<<endl;
    cout<<"j="<<j<<endl;
    return 0;
}
```

4.1 Introduction

If you pass only a copy of 7 to a function, it is called "pass-by-value" and the function will not be able to change the contents of age. It is still 7 when you return.

But, if you pass 4000, the address of i to a function, it is called "pass-by-reference" and the function will be able to change the contents of i. It will be 10 when you return.

4.1 Introduction

- **Return by Reference**

Return by value:

Example:

```
int val1()
{
    //……
    return i;
}
```

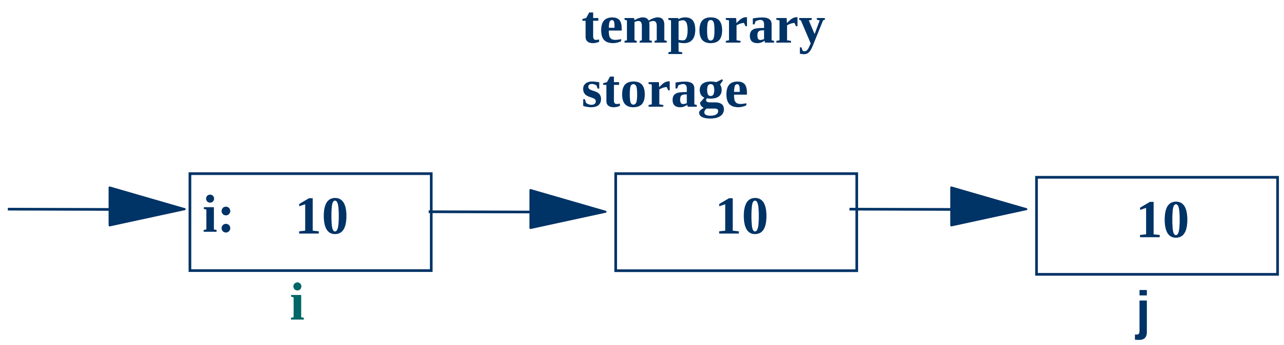
When function is invoked, the value i is copied into temporary storage, which the invoking function can then access.

4.1 Introduction

If the function val1 is invoked as

    j = val1();

the value i is copied into temporary storage and then copied into j.



4.1 Introduction

Return by Reference :

An alternative to return by value is return by reference, in which the value returned is not copied into temporary storage.

Example:

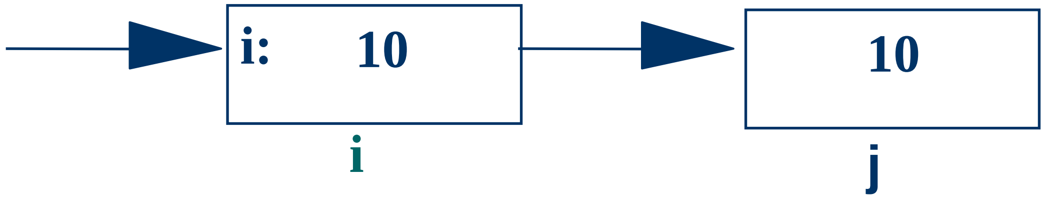
```
int& val2()
{
    //……
    return i;
}
```

4.1 Introduction

If the function val2 is invoked as

```
    j = val2();
```

the value in i is copied into j.



Example

```
#include <iostream.h>
int a=0;
int& fun()
{
    return a;
}
void main()
{
    fun( )++;
    return;
}
```

Right or wrong?

4.2 Inline Functions

The keyword inline can be used in a function declaration to request that a function be expanded "inline".

That is, that each occurrence of a call of the function be replaced with the code that implements the function.

Remark:

The compiler, for various reasons, may not be able to honor the request.

4.2 Inline Functions

Example:

```
#include <iostream.h>
inline void swap(int& a, int& b)
{
    int t;
    t=a;
    a=b;
    b=t;
}
```

```
int main()
{
    int i=7, j=10;
    swap(i, j);
    cout<<"i="<<i<<endl;
    cout<<"j="<<j<<endl;
    return 0;
}
```

4.2 Inline Functions

Assuming that the compiler honors the request to expand swap inline, no function call occurs at the line

```
swap(i, j);
```

Because swap is an inline function, the compiler replaces the line

```
swap(i, j);
```

with the code that implements swap.

4.3 Default Arguments

A **default argument** is a value given in the declaration that the compiler automatically inserts if you don’t provide a value in the function call.

4.3 Default Arguments

Example:

```
int add(int x=5,int y=6)
{
    return  x+y;
}
```

```
void main(void)
{
  int i;
  i=add(10,20); //10+20
  i=add(10);       //10+6
  i=add();          //5+6
}
```

4.3 Default Arguments

Rules :

First, only trailing arguments may be defaulted. That is, you can't have a default argument followed by a non-default argument.

Second, once you start using default arguments in a particular function call, all the subsequent arguments in that function's argument list must be defaulted.

4.3 Default Arguments

Example:

```
int add(int x,int y=5,int z=6);  //right
int add(int x=1,int y=5,int z);  //wrong
int add(int x=1,int y,int z=6);  //wrong
```

4.3 Default Arguments

Example:

```
int add(int x,int y=5,int z=6);
   add(10,12,1);   //right
   add();          //wrong
   add(12);        //right
   add(1,,12);     //wrong
```

4.3 Default Arguments

Default arguments are only placed in the declaration of a function (typically placed in a header file).

The compiler must see the default value before it can use it.

Example:

```
#include <iostream.h>

#include <iomanip.h>

int get_volume(int length,

            int width = 2, int height = 3)

{

    cout<<setw(5)<<length

         <<setw(5)<<width

         <<setw(5)<<height<<' ';

  return length * width * height;

}
```

```
int main()

{

  int x = 10, y = 12, z = 15;

  cout << "Some box data is " ;

  cout << get_volume(x, y, z) << endl;

  cout << "Some box data is " ;

  cout << get_volume(x, y) << endl;

  cout << "Some box data is " ;

  cout << get_volume(x) << endl;

  cout << "Some box data is ";

  cout << get_volume(x, 7) << endl;

  cout << "Some box data is ";

  cout << get_volume(5, 5, 5) << endl;

  return 0;

}
```

4.3 Default Arguments

Output:

```
Some box data is   10   12   15  1800
Some box data is   10   12    3   360
Some box data is   10    2    3    60
Some box data is   10    7    3   210
Some box data is    5    5    5   125
```

4.4 Overloading Functions

C++ permits identically named functions within the same scope if they can be distinguished by the number, data types and order of its arguments.

If there are multiple definitions of a function, is said to be overloaded.

4.4 Overloading Functions

Example :

```
int    add(int x, int y);   // type of parameters are different
float add(float x, float y);
```

Example :

```
int add(int x, int y);       // numbers of parameters are different
int add(int x, int y, int z);
```

4.4 Overloading Functions

Example:

```
int add(int x, int y);
int add(int a, int b); //wrong

int   add(int x, int y);
void add(int x, int y); //wrong
```

Example:

```
#include <iostream>
using namespace std;
void print(int a);
void print(double a);
int main()
{
    int x=8;
    double y=8;
    print(x);
    print(y);
    return 0;
}
```

```
void print (int a)
{
    cout<<a<<endl;
}
void print (double a)
{
    cout<<a<<endl;
}
```

4.4 Overloading Functions

Remarks:

A) Overloaded functions are used to give a common name to similar behavior on different data types.

B) Be careful if we use both default arguments and overloading functions.

4.4 Overloading Functions

Example:

```
int abs(int x)
{ …… }
int abs(int x, int y=10)
{ …… }

abs(-100);   //Which function should be invoked?
```

4.5 Template Functions

Template functions are used for simplify the overloading functions.

Syntax:

```
template <typename  T>
FunctionDefine
```

4.5 Template Functions

Example:

```
#include<iostream.h>
template<typename T>
T abs(T x)
{   return x<0?-x:x;   }

void main()
{
    int n=-5;
    double d=-5.5;
    cout<<abs(n)<<endl;
    cout<<abs(d)<<endl;
}
```

Output:

```
5

5.5
```

4.5 Template Functions

The C++ compiler confirms the type of argument by the parameter of abs().

For example, in the segment

```
abs(n)
```

the parameter n is type of int, so in the template function abs(), type of T is int.

Then the C++ compiler makes a function as follows:

```
int abs(int x)
{
   return x<0?-x:x;
}
```

4.5 Template Functions

Remark:

Template compilation mode:

- **A. Inclusion Mode**
- **B. Separation Mode**

Exercise

Given the definition of function, the return value of fun() is (A)

```
int* fun( int a )
{
    int *t, n;
    n=a;
    t=&n;
    return t;
}
```

A an unusable address of memory location

B an usable address of memory location

C value of n

D value of a

Summarize

- Inline Functions
- Default Arguments
- Overloading Functions