> 由 KiOii (*_EM_Cpper*)整理。 (KiOii (*_EM_Cpper*) makes this note.)

Functions

Contents

> A function is a block of code with a name.

6.1 Function Basics

- *define:* **return-type**, a **name**, a list of zero or more **parameters**, and a **body**.
- *execute:* **call operator(())**

Writing a Function

```
/*
* factorial
*/
int fac(int val)
{
    int ret = 1;
    while(val > 1)
        ret *= val--;
    return ret;
}
```

Calling a Function

```
int main()
{
    int j = fact(5);
    cout << "5! is " << j << endl;
    return 0;
}
```

- *doing:* **initializes the function's parameters、transfers control to that function**[1]

Parameters and Arguments

- *define:* **arguments are the initailizers for a function's parameters.**
- *wraning:**which argument initializes which parameter**
  - we have no guarantees about the order in which arguments in whatever order it prefers.[2]

Function Parameter List

- *wraning:* can be **empty but cannot be omitted.**
  - *code-look:*

```
void f1(){/*...*/}
void f2(void){/*兼容C...*}
```

Function Return Type

- *wraning:* return type cannot be an **array** type or a **function** type,but can be a **pointer to an array or a function.**

6.1.1 Loacl Objects

name

- *wraning:* have **scope**、have **lifetimes**
  - The scope of a name is the part of the program's text in which that name is visible.
  - The lifetime of an objects is the time **during the proogram's execution that the object exists.**

local variables

- *define:* **Parameters** and **variables defined inside a function body**
- *wraning:* The lifetime of a local variable depends on how it is defined.

Automatic Objects

- *such that:* Parameters

Local static Objects

- *wraning:* **built-int-type is default initialized to zero**, be initialized just once

6.1.2 Function Declarations

```
void print(vector<int>::const_iterator beg,
           vector<int>::const_iterator end);
// or
void print(vector<int>::const_iterator,
           vector<int>::const_iterator);
```

Function Declarations Go in Header Files

6.1.3 Separate Compilation

> we can store the various parts of the program in sepatate files.

6.2 Argument Passing

> Parameter initialization works the same way as variable initialization.

- *wraning:* when the argument value is copied,the parameter and argument are independent objects.

6.2.1 Passing Arguments by Value

```
void func(int var)
{
    //...
}
int main()
{
    int var2 = 0;
    func(var2);    // paramenter object:var    argument object:var2
    return 0;
}
```

6.2.2 Passing Arguments by Reference

- *parameter:* **const reference or reference**

```
void f(const int &rci){}
void f(int &rci){}
```

6.2.3 const parameters and Arguments

- *wraning:* like the initialization.

```
void f(const int ci);
void f(const int *pci);
void f(int *const cpi);
```

6.2.4 Array Parameter

- *pass an array:* argument is automatically converted to a pointer to the element in the array.

```
void f(int *array){}
int main()
{
    int arr[10]{0};
    f(arr);         // converted to "&arr[0]"
    return 0x0;
}
```

```
void f(const int *beg,const int *end){}
int main()
{
    int arr[10]{};
    f(&arr[0],&arr[10]);
    return 0x0;
}
```

```
void f(const int *array,size_t size){}
int main()
{
    int arr[10]{};
    f(arr,10);
    return 0x0;
}
```

Array Reference Parameters

```
void print(int (&arr)[10])
{
    for(auto elem : arr)
        cout << elem << endl;
}
```

Passing a Multidimensional Array

```
void f(int (*p)[10][10]){}
int main()
{
    int arrr[5][10][10]{};
    f(arr);
    return 0x0;
}
```

6.2.6 Function with Varying Parameters

- *two ways:* **initializer_list、variadic template**
  - *initializer_list:* use for all the arguments have the same type
    - *all the element is const:* Unlike vector, the elements in an initializer_list are always **const values**;

Ellipsis Paramenters

- `void f(parm_list,...);`
- `void f(...);`

6.3 Teturn Type and the return Statement

- *two forms:*
  - **return;**
  - **return expression;**

6.3.1 Function with No Return Value

```
void swap(int &v1,int &v2)
{
    if(v1 == v2)
        return;
    // ...
}
```

6.3.2 Functions That Return a Value

How Values Are Returned

- *define:* The return value is used to initialize a temporary at the call site.

Never Return a Reference or Pointer to a Local Object

```
const string &manip()
{
    return "Empty"; // error
}
```

Reference Returns Are Lvalues

- *return type is reference:* return value is lvalue, other return type return Rvalue.

List Initializing the Return Value

```
vector<string> f()
{
    return {"asfa","asfas"};// initialize the temporary
}
```

Return from main

- EXIT_SUCCESS
- EXIT_FAILURE

6.3.3 Returning a Pointer to an Array

- *use-wraning:*
    - *some-ways:*
        - *use type alias:* `using arrT = int[10];`
        - use trailing return
        - use decltype

Declaring a Function That Returns a Pointer to an Array

- *define:* ***Type*** (* **function** (***paramter_list***))[**dimension**]
- *code-look:* `int (*func(int i ))[10];`

Using a Trailing Return Type

- *standard:* **C++11**
- *wraning:* To signal that the return follows the parameter list,we use auto where the return type ordinarily appears: `auto func(int i)->int(*)[10];`

Using decltype

```
int odd[] = {1,3};
decltype(odd) *arrptr(int i)
{
    return odd;
}
```

6.4 Overloaded Functions

```
void print(const char *p);
void print(const int *beg,const int *end);
void print(const int ia[],size_t size);

int j[2] = {0,1};
print("Hello World");           // calls print(const char*);
print(j,end(j)-begin(j));       // calls print(const int*,size_t);
print(begin(j),end(j));         // calls print(const int*,const int*);
```

- *wraning:* The main function may not be overload.

Defining Overloaded Functions

```
Record lookup(const Account&);
Record lookup(const Phone&);
Record lookup(const Name&);
```

- *difference:* the **number** or the **type(s)** of their parameters.
- *error:*

```
Record lookup(const Account&);
bool lookup(const Account&);
// error:only the return type is different

typedef Phone Telno;
Record lookup(const Phone&);
Record lookup(const Telno&);
// error: nothing different,just alias
```

Overloading and const Parameters

- *top-level pointer:* **indistinguishable**
- *low-level pointer:* **distinguishable**

const_cast and Overload

```
const string &shorterString(const string &s1,const string &s2)
{
    return s1.size() <= s2.size() ? s1 : s2;
}
string &shorterString(string &s1,string &s2)
{
    auto &r = shorterString(const_cast<const string&>(s1),
                            const_cast<const string&>(s2));
    return const_cast<string&>(r);
}
```

Calling an Overloaded Function

- *function matching:* **overload resolution**
- *three result:*
    - best match
    - no match
    - ambiguous call[3]

6.4.1 Overloading and Scope

- *wraning:* the name hides uses of that name declared in an outer scope.[4]
- *wraning:* names do not **overload** across scopes,they **should be in the same scope.**

6.5 Features for Specialized Used

> default arguments、inline、constexpr function

6.5.1 Default Arguments

- *wraning:* if a parameter has a default argument,all parameters that follow it must also have default arguments.[5]

Calling Functions with Default Arguments

```
using sz = string::size_type;
string screen(sz ht = 24,sz wid = 80,char background = ' ');
//...
string window;
window = screen();          // screen(24,80,' ');
window = screen(66);        // screen(66,80,' ');
window = screen(66,256)     // screen(66,256,' ');
window = screen(66,256,'#') // screen(66,256,'#');
// The default arguments are used for the trailing arguments of a call.
```

6.5.2 Inline and constexpr Functions

inline Functions Avoid Function Call Overhead

```
inline const string&
shortString(const string &s1,const string &s2)
{
    return s1.size() <= s2.size() > s1 : s2;
}
```

- *wraning:* less than **75-lines** function、must **not be recursive function**[6]

constexpr Functions

- *wraning:* **return type** and **the type of each parameter** must be a **literal type**, only **one return statement.**

```
constexpr int new_sz(){return 42;}
constexpr int foo = new_sz(); // compiler will call new_sz(),use the return to replace
```

- *wraning:* **constexpr** functions are **implicitly inline.**
- *wraning:* statements in function body that **cannot have actions at run time.**
    - *statements-such as:* **null statement、type aliases、using declarations**
- *wraning:* A **constexpr** function is not required to return a constant expression.[7]

Put inline and constexpr Functions in Header Files

> Compiler needs the definition,not just the declaration,in order to expand the code.
>
> As a result,**inline** and **constexpr** function normally are defined in headers.

6.5.3 Aids for Debugging

- *debugging code:* **excuted** only while the program is beging **developed**.when the application is **completed** and ready to ship,the debugging code is **tuened off**.[8]
- *two preprocessor facilities:* ***assert*** and ***NDEBUG***

The assert Preprocessor Macro

- *define:* a preprocessor macro.
- *use:* `assert(expr);`
  - *header file:* <**cassert**>
  - *wraning:* preprocessor is not compiler,so we use **assert** but not **std::assert**
  - *expr is false:* writes a **message** and **terminates** the program.
  - *expr is true:* do **nothing**.
  - *wraning:* like the inline function

```
assert(word.size() > threshold);
```

The NDEBUG Preprocessor Variable

- *wraning:* the behavior of assert depends on the status of NDEBUG.
  - *if **NDEBUG** is defined:* assert does **nothing**
- *static local variable and defined by compiler:*
  - `__func__ // :holds the function's name`
  - `__FILE__ // :string literal containing the current line number`
  - `__LINE__ // :integer literal containing the current line number`
  - `__TIME__ // :string literal containing the time the file was compiled`
  - `__DATA__ // :string literal containing the data the file was compiled`

6.6 Function Matching

```
void f();
void f(int);
void f(int,int);
void f(double,double = 3.14);
f(5.6);    // calls void f(double,double)
f(42,2,56);// error
```

Determining the Candidate and Viable Functions

1. **candidate function:** the same name
2. **viable function:** can be called with the arguments int the given call.
3. **finding the best match,if any**

6.6.1 Argument Type Conversion

Convetsions are ranked as follows

1. An exact match
    - argument and parameter types are idetical.
    - array、pointer can converted
    - top-level const is added to or discarded from the arguments
2. Match through a const conversion
3. Match through a promotion
4. Match through a arithmetic or pointer conversion
5. Match through a class-type conversion

6.7 Pointers to Functions

```
bool lengthCompare(const string&,const string&);
bool (*pf)(const string&,const string&);
//...
pf = lengthCompare;  // the same to &lengthCompare
bool b1 = pf("hello","world"); // the same to (*pf)("hello","world")
```

- *decltype(function):* get the **function type** but not pointer to function.

---

1. 初始化参数，主调函数执行中断，被调函数开始执行。↩

2. 尽管实参和形参一一对应，但是我们不能保证实参的求值顺序，这是依赖于编译器的。↩

3. 二义性调用↩

4. 内层作用域中的名字将隐藏外层作用域中的同名↩

5. 一旦某个形式参数有了默认的实参值，则它后面的所有形式参数都需要有默认的实参值↩

6. 递归函数(recursive function)↩

7. constexpr函数不一定返回常量表达式↩

8. 这些代码只在开发程序时使用。当应用程序编写完成准备发布时，要先屏蔽掉调试代码。↩