Chapter 7 Functions

7.1 Function basics

- Prototypes and function headers
  - A prototype provides the function's interface information (input type, return type, and function name) to the compiler, so the compiler will not be "surprised" when it encounters the function's calling in compile time. The only way to avoid function prototyping is to place the function definition ahead of its calls.
  - The *signature* of a function is, generally speaking, the information provided by the function's prototype, excluding the function name and return type.
  - In a function's prototype, the input argument's names are optional, and do not have to be the same as the name used in the function header.
  - Compilers matches function definitions with their function prototypes by matching function headers and prototypes.
  - Function overloading is permitted in C++ (not C). That is, it is permitted functions that have different number and/or types of input arguments to share the same function name.
- Passing arguments by value: largely omitted
  - *Formal argument* and *actual argument*: the formal argument is used to receive the value pass by the actual argument, and therefore changes made to the formal argument will not affect the actual argument.
  - A Formal argument is a copy of the actual argument. The very act of copying takes place when the values are passed to them, so this mechanism may waste a lot of memory sapce if the size of the actual argument is gigantic. In this case, it is recommended to use passing by pointer or passing by reference. (In MATLAB, *lazy-copy* mechanism is adopted to avoid unnecessary copying.)
- Passing arguments by pointer: largely omitted
  - An array name is the synonym of the array's first element's address. So passing an array's name is actually passing an address. Therefore in the function prototype and headline you should declare the formal argument as pointers.

```
int a[2] = {1,2};
int b[2][3] = {1,2,3,4,5,6};
int c[2][3][4] = {...};
void fun_a1(int a[]);         void fun_a2(int *a);
void fun_b1(int b[][3]);      void fun_b2(int (*b)[3]);
                              void fun_b3(int *b[3]);      // error
void fun_c1(int c[][3][4]);   void fun_c2(int (*c)[3][4]);
                              void fun_c3(int *c[3][4]);   // error
```

- Passing arguments by reference: see Chapter 8

7.2 Pointers and `const`

- Using `const` with pointers prevents inadvertent attempts to modify the object the pointer points to. However, there are some subtle effects that must also be taken into account.

- Assigning a regular variable's address to a pointer-to-`const` is valid, and it forbids you from modifying the variable using the pointer. But it is permitted to modify the variable using the variable itself, of course.

```
int a = 1; const int *pa = &a;
*pa = 2;  // error
a = 2;    // ok
```

- Assigning a `cosnt` variable's address to a pointer-to-`const` is valid. Trivial.

```
const int a = 1; const int *pa = &a;
*pa = 2;  // error
a = 2;    // error
```

- Assigning a `cosnt` variable's address to a regular pointer is INVALID. This feature prevents you to circumvent the `const` protection enjoyed by the variable to make modification.

```
const int a = 1;
int * pa = &a;    // error
```

  And it is also implied by this feature that the type of the formal argument of a function must accommodate to the corresponding actual argument's `const` status.

```
const int a[] = {1,2};
int func1(int a[], int n); // should have been 'const int a[]'
int func2(int *a, int n);  // should have been 'const int *'
...
int p = func1(a,2);        // error
int q = func2(a,2);        // error
```

- Note that a `const` prefixed to a pointer's declaration means it is prohibited to make modification to the pointed-to variable through the pointer.
  - It does not the mean the the value of the pointed-to variable cannot be altered, as noted above;
  - The pointer itself can still be modified, a.k.a. redirecting the pointer.

```
int a = 0, b = 9;
const int *p = &a;
*p = 1;            // error
a = 1;             // ok
p = &b;            // ok
*p = 8;            // error
```

- If you want to protect the pointer itself from modification, you should insert `const` immediately in front of the pointer name, e.g. `int * const p = &a;`. Pointers declared in this way allows changing the pointed-to variable using this pointer, but prohibits redirecting the pointer to another variable.
- A pointer declared in this way protects the pointer itself from modification, and prohibits changing the value of the pointed-to variable through the pointer: `const int * const p = &a;`.

7.3 Recursion: functions calling themselves

- Unlike C, C++ does not allow `main()` to call itself.
- Single recursive call and multiple recursive calls are allowed.
- Recursion depth has limits, dependent on the stack size dedicated to the process.

Chapter 8 More about Functions

8.1 Inline functions

- *Inline function* facility is an addition to C++. In C, the preprocessor instruction `#define` can be used to provide *macro*s, a crude implementation of inline code.
- To understand the enhancement brought by inline functions, it is necessary to understand (though superficially) the mechanism of calling normal functions.
    - Suppose you have a program translated into machine instructions. When the CPU executes the program, its first step is to load the instructions into the memory and each instruction has a particular address.
    - The CPU sets its program counter register (PC) to the address of the starting instruction, and then executes the program by going through the instructions step-by-step, or jump backward or forward if the program contains loops or branching statements.
    - Meanwhile, if the CPU encounters a (normal) function call instruction, it saves the address of the instruction immediately following the function call, and stores current function's arguments to the stack, and then jump to the starting address of the function to execute its instructions, and jump back to the instruction whose address is saved.
    - There is an overhead to jump forth and back, keeping track of where to jump and storing arguments. The overhead translates into elapsed time.
    - So inline functions are introduced to save the overhead. The code of an inline function is "in line" with the other code in the program. That is, the compiler replaces the function call with the corresponding code. With this facility, the CPU can execute the program step-by-step, instead of jumping back and forth.
    - The amount of overhead saved by an inline call is significant only if the actual time spent on the instructions of the function is relatively small compared to the time spent on CPU's additional jumping and saving operations.
- There are two ways to use the feature of inline functions.
    - Prefix the function's prototype OR definition with keyword `inline`.
    - (alternative) omit the prototype and place the entire definition (the header and the body) where the prototype would normally go. Prefix it with `inline`.
- The compiler does not have to honor your request to make a function inline, if it determines that the function calls itself (recursion) or the function is too large (so the overhead saved by inline calling is not significant).
- An obsolete alternative: macros
    - C (and C++) provides an alternative to the `inline` facility - the *macro*s, or *macroinstruction*s. A macro is an instruction that specifies how a certain input sequence should be mapped to an output sequence. For macros used in programming languages, such instruction directs the compiler to substitute certain sequences in the code with the corresponding output sequence, and

therefore the mapping occurs during compile time.

- In C and C++, macros are realized by the preprocessor instruction `#define`.
- An example:

```
#define SQUARE(X) X*X
...
a = SQUARE(5);     // replaced by a = 5*5;
b = SQUARE(1+2);   // replaced by b = 1+2*1+2; NOT (1+2)*(1+2)
c = SQUARE(c++);   // replaced by c++*c++;
                   // equivalent to: c = (c*c, c=c+2);
#define SQUARE(X) (X)*(X)
...
e = SQUARE(1+2);   // replaced by (1+2)*(1+2)
```

8.2 Reference variables

- *Reference* is a new compound type introduced in C++. A reference is an alias for an actual variable: `type &ref = var;`. More precisely, here we discuss *lvalue reference*.
- A reference, in essence, acts like a `const` pointer: you have to initialize it when you create it, and if a reference pledges its allegiance to a variable, it sticks to its pledge.

```
int a;
int &ra = a, * const pa = &a;  // ok: int &ra = a;  int * const pa = &a;
ra = 1;       *pa = 1;         // both equivalent to a = 1;
int &ra2 = *pa;                // also ok
std::cout << ra << " " << *pa << " " << ra2 << std::endl; // 1 1 1
```

- Restrictions:
  - Declaring a pointer to reference type is forbidden, which means declarations such as `int& *p = ...` generates a compile error.
  - Declaring an array-of-reference is forbidden, which means declarations such as `int &a[10] = ...` generates a compile error.
  - Declaring a reference should be accompanied by its initialization, e.g. `int &ra = a'`, unless the declaration is inside a function argument list, or declares a data member in a class definition.
  - C++ does not specify a method to get the address of a reference variable itself.
- Reference-to-array: the golden rule is to remember the fact that the precedence of `[]` is higher than that of `&`.

```
int a[3] {1,2,3};
int (&ra)[3] = a; // just like: int (*pa)[3] = a;
```

or, you can use a `typedef` to simplify:

```
int a[3] {1,2,3};
typedef int Int3[3]; typedef Int3 &rInt3;
Int3 &ra1 = a;       rInt3 ra2 = a;        // ra1 is equivalent to ra2
```

- Passing arguments by reference: like a pointer, it is permitted to modify the value of the actual argument by using the reference. In many implementations, a reference is essentially treated as a pointer, which can be showed by the assembly code.

	pointer	**reference**
function prototype	prefix `*`	prefix `&`
binding an actual variable	prefix `&`	none
function header	prefix `*`	prefix `&`
function body	prefix `*`	none

- Just like pointer-to-`const`, if you want to protect a variable from being modified through its reference, you can prefix the declaration with `const`. And if you want to revoke the protection, you can use `const_cast<>`.

```
1  int a = 0;
2  const int &ra = a;
3  ra = 1;           // error
4  int &rra = const_cast<int &>(ra);
5  rra = 1;          // ok
```

- Temporary variables, reference arguments, and `const`
  - As you can understand, if the prototype of a function is `int cube(int a);`, then these function calls are valid: `cube(x);` (directly assign `x` to `a`) and `cube(x+1);` (evaluate the expression and assign the resulting value to `a`).
  - However, if the prototype is `int cube(int &a);`, then `cube(x);` still works (directly assign `x` to reference `a`), but `cube(x+1)` does not because `x+1` is not of type `int &`. To sum up, a reference variable has to refer to a lvalue of the correct type, but there is an exception (discussed below).

    > *lvalue*: a data object that can be referenced by address. It has two kinds: *modifiable lvalue*, a.k.a. data objects that can appear on the left side of an assignment operator `=`: regular variables, array elements, structure members, references, pointers, dereferenced pointers; and *non-modifiable lvalue*: data objects that satisfy the definition of lvalue but is an array, or has read-only protection because its declaration is prefixed by keyword `const`.
    >
    > Multiple-term expressions and literal constants (except quoted strings, which has addresses) are non-lvalues, namely, *rvalue*s.

  - In contemporary C++, there is an exception to the foregoing rule - if the prototype is `int cube(const int &a);`, then function calls such as `cube(x+1)` is valid, thanks to `const`. That is because if the actual argument does not match a reference argument, the compiler generates a temporary variable, and refer the reference variable to that newly-created temporary variable, provided that
    - the actual argument is of the correct type but is not an lvalue; or
    - the actual argument is of the wrong type, but is of a type that can be implicitly converted to the correct type. This feature is intensively utilized since a derived class object can be implicitly converted to the base class.

```
int cube(const int &a);  // function prototype
int b = 2; char c = 'g'; int &rb = b; int *pb = &b;
// normal, no temporary variable is needed : lvalue and correct type
int u = cube(b);   int v = cube(rb);   int w = cube(*pb);
// 1. the actual argument is of the correct type but is not an lvalue
int x = cube(1);   int y = cube(b+1);
// 2. the actual argument is of the wrong type, but is of a type
//    that can be converted to the correct type
int z = cube(c);
```

- The aforementioned exception works only if the reference is a reference-to-`const`. Why does the C++ standard exclude normal references? The reason is if the intention of the function is to modify the variable a reference refers to, creating a temporary variable and having the reference refer to that temporary variable thwarted the purpose.

  > In some old C++ implementations, the rules are indeed freer. They issue a warning instead of an error.

- Reference as a return type
  - Normal return mechanism: Assume function A calls function B. In normal, when B returns a value, the program copy the returned value to A.
  - In the case above, if B returns a reference, then the program copy the address of the referred-to variable to A. This approach is significantly more efficient if the normal mechanism involves copying a value that is of huge size, say, a structure.

    > Returning a pointer is another approach to improve the efficiency.

  - Normal returned objects are rvalues - they cannot be accessed by addresses. Hence a call to a function that returns a normal object can only appear on the right side of `=`. But references are lvalues, so a call to a function that returns a reference can appear on the left side of `=`. Plus, you can use qualifier `const` to protect the returned object from inadvertent modification.

    ```
    int &func1();       // function prototype. It returns a reference
    const int &func2(); // function prototype. It returns a reference
    ...
    func1() = 2;        // permitted. Modifies the referred-to variable
    func2() = 2;        // error. The returned object cannot be modified
    ```

  - Just like returning a pointer, you should avoid returning the reference to a variable that ceases to exist when the called function terminates.

    ```
    int & modeSwitch(int &mode)  // you should avoid codes like this
    {
      int newMode = ++mode; if(newMode == 5) newMode = 1;
      return newMode; // newMode is about to cease to exist
    }
    ```

    - The simplest way to avoid this problem is to return a reference which was passed in as an argument to the called function.

```
int & modeSwitch(int &mode)  // good
{
  mode++; if(mode == 5) mode = 1;
  return mode;
}
```

- Another way is to use `new` to create new storage. The storage class of the memory space allocated by `new` is dynamic storage instead of automatic storage, so this block of memory will not be freed when the function terminates. You have to manually deallocate the memory with `delete`.

```
int & modeSwitch(int mode)  // also good
{
  int *pNewMode = new int {mode};
  (*pNewMode)++; if(*pNewMode == 5) *pNewMode = 1;
  return *pNewMode;
} // remember to deallocate
...
int main()
{
  int mode = 1;
  int &nextMode = modeSwitch(mode);
  std::cout << nextMode << endl;
  int *p = &nextMode; delete p;// deallocate (must be a pointer)
}
```

- Using references to passing class objects is of common practice. A base class reference can refer to a derived class object without requiring a type cast, because a derived class object can be implicitly converted to the base class.

> NOTE: If the the data object is a class object, it is recommended you prefer using a reference over a pointer. There are two reasons: the semantics of class design often, though not always, requires using a reference; and it looks neater to use reference since there is no prefixes to the reference in function calls and function bodies, whereas pointers have `*`s.

8.3 Default arguments

- *Default argument*s simplifies function calls by implicitly providing default argument values.
- Default values are designated in function prototypes (NOT in function headers), and has to be set from right to left. That is, in order to set a default value for an argument, the programmer has to set default values for arguments on the right.

```
int harpo(int n, int m = 4);  // prototype. "m" can be left out.
...
int a = harpo(1);             // equivalent to harpo(1,4);
int b = harpo(2,3);           // override the default value m = 4.
```

> Q: Why set them in function prototypes instead of function headers?

> A: It makes more sense because a compiler get the information from function prototypes, and it will not be surprised if it meets a function call that omits some arguments.
>
> Because default values are required to be set from right to left, it is recommended to place arguments that rarely change to the right.

8.4 Function overloading

- Overloaded functions have different function signatures, e.g. different number and/or types of arguments. They do not need to have the same return type.
- The function-matching process does discriminate between `const` and non-`const` arguments. That is, the compiler selects the "best match", though it is permitted to assign a non-`const` variable to a `const` variable. For instance,

```
void show(char *s); void show(const char *s); // overloaded
void secondShow(const char *s);    // no overloading
void thirdShow(char *s);           // no overloading
...
const char s1[20] = "How is the weather?";
char s2[20] = "How is business?";
show(s1);           // matches show(const char *)
show(s2);           // matches show(char *)
secondShow(s1);  secondShow(s2);   // both valid
thirdShow(s1);      // error: cannot assign 'const char *' to 'char *'
```

- The "best match" property of function overloading gives you flexibility to customize the the implementation of a function based on the argument type. For example,

```
int quadruple(int &a);      // prototype: no overloading
...
int x = 1; const int y = 2;
int u = quadruple(x), v = quadruple(y); // both call the function
```

```
int quintuple(int a), quintuple(const int a); // overloaded (comma is ok)
...
int x = 1; const int y = 2;
int u = quintuple(x), v = quintuple(y);  // call their respective matches
```

> Best match: if there are multiple valid matches, the compiler choose the best match - the match that requires minimum, if any, type conversion.

More examples:

```
void staff(int & a);       // matches modifiable lvalue
void staff(const int & a); // matches rvalue, const lvalue
void boss(int & a);        // matches modifiable lvalue
void boss(const int & a);  // matches const lvalue
void boss(const int && a); // matches rvalue
```

- *Name decoration*: a compiler uses a preset convention to transform ("decorate") a function name in accordance with the function's prototype, for its own use. For instance, if a function's prototype is `long func(int, float);`, the compiler might record the function with a more unsightly representation: `?func@@YAXH`. Each function, overloaded or not, has its own unique internal representation. Different compilers may use different convention for decorating.

8.5 Function templates

- A *function template* is a generic function description. It defines a function in terms of generic type for which a specific type, such as `int` or `double`, can be substituted. By passing a type as a parameter to a template, the compiler can be directed to generate a function for that particular type.
  - The process is termed *generic programming*, one of the features of C++ (the other is *object-oriented programming*, OOP).
  - The template feature is sometimes referred to as *parameterized type*s, because specific types are represented by parameters.
  - The template feature is of usefulness if you need functions that apply the same algorithm to multiple types of data. It saves you time and troubles by avoiding copying the same code multiple times and made petty changes to them.
  - The function generated by a function template is called a *template-generated function*, or more shortly, a *template function*.
- Usage: an example. Note that `template <typename AnyType>` is NOT a statement itself; rather, it is part of the function template prototype and header. Plus, in the *template parameter list* `<...>`, the keyword `typename` can be replaced by `class`, because the former one does not exist until C++98.

```
// prototype. Keyword "typename" and "class" has identical meaning here.
template <typename AnyType>       // no semicolon here in the middle
void swap(AnyType &a, AnyType &b);
```

```
// definition
template <typename AnyType>       // no semicolon here in the middle
void swap(AnyType &a, AnyType &b)
{
   AnyType temp = a;
   a = b; b = temp;
}
```

```
// call. The compiler instantiates the function template.
int i = 1, j = 2; char x = 'g', y = 'h';
swap(i,j);    // the compiler generates void swap(int &a, int &b);
swap(x,y);    // the compiler generates void swap(char &a, char &b);
```

- Multiple `typename`s are permitted. For example,

```
template <typename T1, typename T2> double calculate(T1 &a, T2 &b);
template <typename T, typename U> T modify(T &a, U &b, char *c);
// "typename" and "class" can be used interchangeably
template <typename T, class U> void display(T &a, U &b, char *c);
```

- Overloaded function templates: as mentioned before, a function template is useful if you need functions that apply the same algorithm to multiple types of data. Sometimes, however, not all types use the same algorithm. *Overloaded template* facility is one approach to handle such possibility (another approach is *explicit specialization*, discussed in 8.6).
    - Usage: overloaded function templates need distinct function signatures, as with overloaded functions.

```
// prototypes of overloaded function templates
template <typename T> void swap(T &a, T &b);
template <typename T> void swap(T *a, T *b, int n);
// definitions of overloaded function templates
template <typename T> void swap(T &a, T &b)
{...}
template <typename T> void swap(T *a, T *b, int n)
{...}
```

- Nontype parameters
    - In addition to type parameters, we can define template functions that take *nontype parameter*s. A nontype parameter represents a value rather than a type.
    - When the compiler instantiate a template function, it replaces the nontype parameter with a value that is provided by the function call or deduced by the compiler.
    - Template arguments used for nontype template parameters must be constant expressions - expressions whose value are determined during compile time instead of runtime.
    - Usage: specify the nontype parameters with specific type names instead of `typename` or `class`.

```
template <unsigned N, unsigned M>
int compare(const char (&a)[N], const char (&b)[M]);
```

      When we call the function: `compare("hi", "mom")`, the compiler instantiates the function template: `int compare(const char (&a)[3], const char (&b)[4])`.

- `inline` function template: the `inline` specifier follows (not leads) the template parameter list, i.e. `template <typename T> inline T min(const T&, const T&);`.

8.6 Specializations of function templates

- 3 sorts of specializations: explicit specialization, explicit/implicit instantiation.
- *Explicit specialization* of a function template is useful if you have a special case in which a particular type of data needs a different algorithm, and you might not be able to use overloaded function templates because the signatures are the same.
- Third-generation specialization rule: after some experimentation with other approaches, the C++98 standard settled on this approach:
    - For a given function name, you can have a non-template function, a template function, and an explicit specialization template function, along with overloaded versions of all these.
    - The prototype and definition for an explicit specialization should be preceded by `template <>` and should mention the specialized type by name.

- A explicit specialization function overrides template-generated functions, and a non-template function overrides both:

  > For matched functions that have the same name:
  >
  > non-template function >
  >
  > explicit specialization function >
  >
  > function-template-generated function

- Usage: an example

```
// TOP priority: non-template function
void swap(job &, job &);     // suppose "job" is a structure type
// 2nd priority: specialization function
template <> void swap<job>(job &, job &);
   // "<job>" can be dropped because the argument list contains the info
// 3rd priority: template-generated function
template <typename T> void swap(T &, T &);
```

  - As mentioned previously, if more than one of these prototypes is present, the compiler chooses the non-template version over explicit specializations and template versions, and it chooses an explicit specialization over a version generated from a template.
  - An explicit specialization function is NOT generated by the cognominal function template; it uses its own function definition, whose algorithm can be different from the template's.

- Terms: instantiation v. specialization
  - Including a function template in the code does not in itself generate a function. It is merely a blueprint of a function.
  - When the compiler uses the function template to generate a function for a particular type, the result is termed an *instantiation* of the template. A function template will NOT be compiled UNTIL instantiation. Two kinds of instantiations:
    - *Implicit instantiation*: the normal instantiation. The compiler generates a implicit instantiation when it encounters a statement which calls the function name with a specific type of data.
    - *Explicit instantiation*: instruct the compiler to create a particular instantiation: by declaring the particular variety you want. The syntax is demonstrated in this example:

```
// choice #1: using a declaration:
template void swap<int>(int, int);
// there is no template parameter list after "template"
```

```
// choice #2: tampering with the function call:
swap<int>(a,b); // a explicit instantiation of function template
swap(c,d);      // a implicit instantiation of function template
```

      Upon seeing the explicit instantiation, the compiler uses the `swap()` function template to generate an instantiation for `int`. You have to ensure the type specified in the explicit instantiation matches the input arguments.

- Explicit specialization is not explicit instantiation. The former tells the compiler "do NOT use the cognominal function template; instead, you this function definition defined for this type", whereas the latter tells the compiler "use the cognominal function template to generate a function for this type."

```
// explicit specialization
template <> void swap<job>(job &, job &);// provide definition later
// explicit instantiation
template void swap<int>(int, int); // use a declaration: choice #1
swap<int>(a,b);                    // use a function call: choice #2
```

- It is an error to try to use both an explicit instantiation (uses the template) and an explicit specialization (does not use the template) for the same type(s) in the same file, or more generally, the same translation unit.
- Implicit instantiation, explicit instantiation, and explicit specialization, are collectively termed *specialization*s. What they have in common is that they represent a function that uses specific types rather than one that is a generic description. The function is based either on a function template (implicit/explicit instantiation), or on a specific function definition provided in addition (explicit specialization).

- Which function version does the compiler pick?
  - The strategy to decide which function to use among cognominal function definitions and/or function templates is called *overload resolution*. There are 3 phases, broadly speaking:
    - **Phase 1: look at the name** Assemble a list of candidate functions. These are functions and/or template functions that have the same name as the called function.
    - **Phase 2: look at the number and types of arguments**: Assemble a list of viable functions from the candidate functions. These are functions with the correct number of arguments and for which there is an implicit conversion of sequence, which includes the case of an exact match of types between actual arguments and formal arguments.
    - **Phase 3: decide**: if there is a best viable function, use that function; if not, generates an compile error.
  - An example: suppose there is a function call - `may('a');`. And the compiler finds that there are not one, but many cognominal functions/function templates:

```
void may(int);                          //#1
float may(float, float = 3);            //#2
void may(char);                         //#3
char * may(const char *);               //#4
char may(const char &);                 //#5
template<class T> void may(const T &);  //#6
template<class T> void may(T *);        //#7
```

    - #1~#7 are the list of candidate functions. Among them, #4 and #7 are not viable because a `char` cannot be converted implicitly to a pointer, be it a `char *` or a `T *`.
    - The remaining are viable. Which is the best? In general, the ranking from best to worst is: (1) exact match with no or trivial conversions, with regular functions outranking templates; (2) conversion by promotion (e.g. from `char` to `short` to `int`, and `float` to `double`); (3) conversion by other nontrivial conversions (e.g. from `int` to `char`, or `long` to `double`); (4)

user-defined conversions, such as those defined in class declarations.

> *Trivial conversion*s: a match with trivial conversions are considered qualified for an exact match (inferior to matches with no conversion).

from an actual argument	to a formal argument
`Type`	`Type &`
`Type &`	`Type`
`Type []`	`Type *`
`Type (argument-list)`	`Type(*)(argument-list)`
`Type`	`const Type` or `volatile Type`
`Type *`	`const Type` or `volatile Type`

- If more than one functions tied (have the same highest ranking), the compiler might not be able to complete the process of overload resolution and generates an error message, probably using words like *ambiguous*.
- *Partial overloading rules*: if there are more than one function has the same ranking, there is still a hope that the compiler uses *partial overloading rules* to resolute the overload problem: it picks the best-specialized one because it underwent fewer conversions in being generated. For example, in the following code

```
// overloaded function templates
template <typename T> show(T arr[], int n);    //#1
template <typename T> show(T *arr[], int n);   //#2
...
// function calls
double a[2] {0,1}; double *b[2] = {a, a+1};
show(a,2);      // use #1, no doubt
show(b,2);      // which to use?
```

  The function call `show(b,2);` can use both #1 and #2, the parameterized type `T` being interpreted as `int *` and `int` respectively. The compiler picks #2, because it makes the specific assumption that the array contains pointers and thus it is more specialized than #1, though they are both exact matches. If you remove #2, the compiler has no choice but to pick #1.

- If a function call has multiple arguments and is matched to multiple prototypes, the compiler must look at matches for all the arguments and pick the best match. For a function to be better than another, it has to provide a better match for at least one argument and at least as good a match for all other arguments.
- Make your own choice: you may dictate your own pick, which can be different from the compiler's choice made by overload resolution.

```
//prototypes
template <class T> T lesser(T a, T b);     //#1
int lesser (int a, int b);                 //#2
...
// call
int nAvg = 5; double c = 7.2, dMax = 7;
// use #2, chosen by the compiler. Because a non-template function
// overrides template functions and explicit specializations
int res1 = lesser(6, nAvg);
// use #1 with int, chosen by you by using '<>'
int res2 = lesser<>(6, nAvg);
// use #1 with double, chosen by the compiler. Because the
// non-template function does not match.
double res3 = lesser(c, dMax);
// use #1 with int, chosen by you by using '<int>'
double res4 = lesser<int>(c, dMax); // convert 'double' to 'int'
```

8.7 The `decltype` keyword

- The keyword `decltype` is added in C++11. Its purpose is to tell a compiler what type a data should be without explicitly specifying the type, in a variable declaration.
- Syntax: `decltype(expression) var;`. Used as a variable declaration statement.

```
short x = 1; char c = 'a'; // thus x+c is of type 'int' (promotion law)
// regular way
int r1 = x + c;
// using 'decltype'
decltype(x + c) r2;        // the compiler knows the type shoud be 'int'
r = x + c;
// or in one line:
decltype(x + c) r3 = x + c;// the compiler knows the type shoud be 'int'
```

- How does the compiler deduce the correct type? It has to go through a complex checklist, and the rule are rather interesting, as shown below.
  - **Stage 1:** if `expression` is an unparenthesized identifier, then `var` is of the same type as the identifier. For instance, if `x` is an `int`, then `decltype(x) w;` makes `w` an `int`. If not, go to stage 2.
  - **Stage 2:** if `expression` is a function call, then `var` has the type of the function's return type. Note that the compiler does not actually execute that function call; it just examines the prototype to learn the return type. If not, go to stage 3.
  - **Stage 3:** if `expression` is an lvalue, the `var` is a reference to that lvalue. For instance, if `x` is an `int`, then `decltype((xx)) w;` makes `w` an `int &`. If not, go to stage 4.

    > It might seem contradicted to stage 1, but there is actually no contradiction. If `x` is an `int`, then `decltype(x) w` makes `w` an `int` instead of an `int &` because it has been captured by stage 1 already.
    >
    > Besides, parentheses do not change the value or lvalueness of an expression. If `xx` is an lvalue, then `(xx)` is, too. `(xx) = 1;` has the same effect as `xx = 1;`.

    - **Stage 4:** if none of the preceding cases apply, the `var` is of the same type as `expression`, as showed by the foregoing example code.
- Note that the validness of using `decltype(x+y)` requires that `x` and `y` are already declared ("in scope") so the compiler knows their types, be it built-in types, compound types, or parameterized types.
    - For example, function template prototype `template<class T1, class T2> decltype(x+y) add (T1 x, T2 y);` is invalid, because `x` and `y` are not yet declared when `decltype` is invoked. To solve this, C++11 added the facility of trailing return type, as discussed in 8.8.

8.8 Trailing return type

- The function prototypes has an alternative syntax besides the traditional one: `returnType funcName(argument-list);`. This new facility is called *trailing return type*: `auto funcName(argument-list) -> returnType;`
- The keyword `auto` here does NOT mean automatic type deduction; it is just a placeholder, so you still have to tell the compiler what the return type is by specifying the type, using a parameterized type, or using `decltype`.
- Examples

```
// prototypes
double greater1(double a, double b);          // traditional syntax
auto greater2(double a, double b) -> double;          // new syntax
// definitions
double greater1(double a, double b)           // traditional syntax
{...}
auto greater2(double a, double b) -> double           // new syntax
{...}
```

- This facility can be applied to function templates, too.

```
template <class T1, class T2> auto add(T1 x, T2 y) -> decltype(x+y);// ok
template <class T1, class T2> decltype(x+y) add(T1 x, T2 y);     // error
// When 'decltype' is used, 'x' and 'y' are not declared yet.
```

□