C/C++ Programming Language

CS205 Spring

Feng Zheng

Week 7

南方科技大学

SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Content

- Brief Review
- Reference Variable
- Function Overloading
- Function Template
- Summary

Brief Review

Content of Last Class

- Various function applications
  - Arrays
  - C-style strings
  - Structure
  - String class and array objects
  - Recursion
  - Pointer to functions

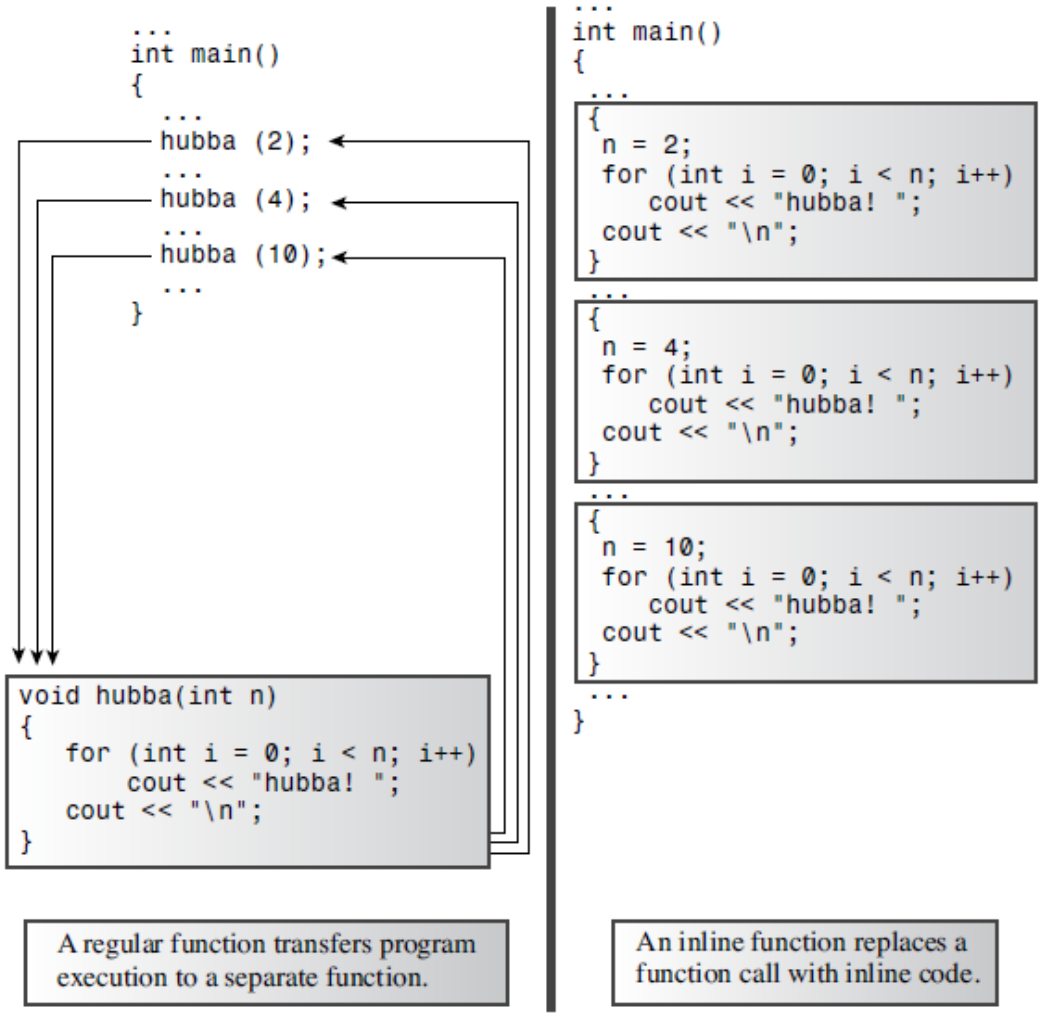
Adventures in Functions

Mechanism of Function Call

- Codes are the data as well
  - The product of the compilation process is an executable program
    - ✓ Consist of a set of machine language instructions
  - The operating system loads these instructions into the memory
    - ✓ Each instruction has a particular memory address
    - ✓ Jump backward or forward to a particular address (loop or branching)
- Normal function: Jump forth and back
  - Store the memory address of the instruction immediately following the function call
  - Copy function arguments to the stack
  - Jump to the memory location that marks the beginning of the function
  - Execute the function code
  - Jump back to the instruction whose address it saved

C++ Inline Functions

- Compiler replaces the function call with the corresponding function code
  - Run a little faster than regular functions
  - Come with a memory penalty
  - Be selective about using inline functions
- Two steps
  - Preface the declaration with the keyword inline.
  - Preface the function definition with the keyword inline
- Inline versus macros
  - Macros don't pass by value



Reference Variables

- A new compound type to the language—the reference variable
  - A reference is a name that acts as an alias
  - An alternative name, for a previously defined variable
- Use a reference as an argument
  - The function works with the original data instead of with a copy
  - A alternative to pointers for processing large structures

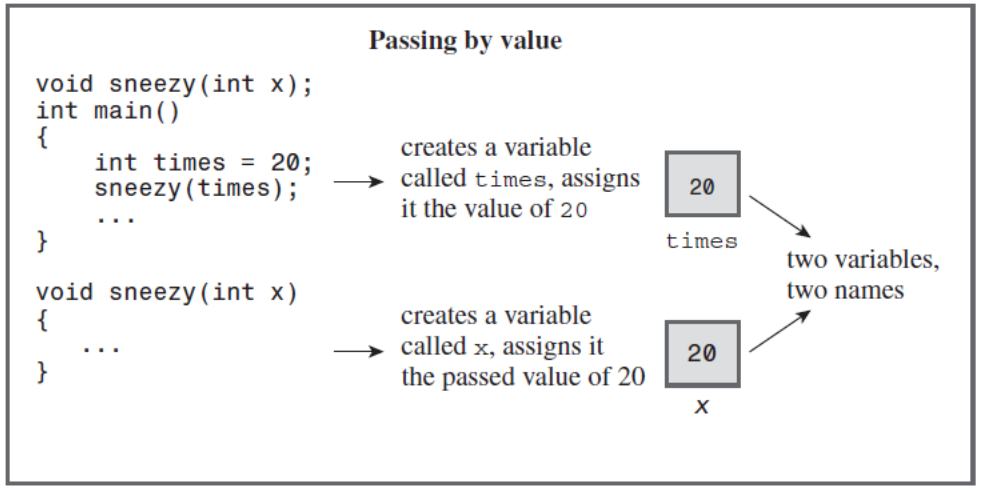
Creating a Reference Variable

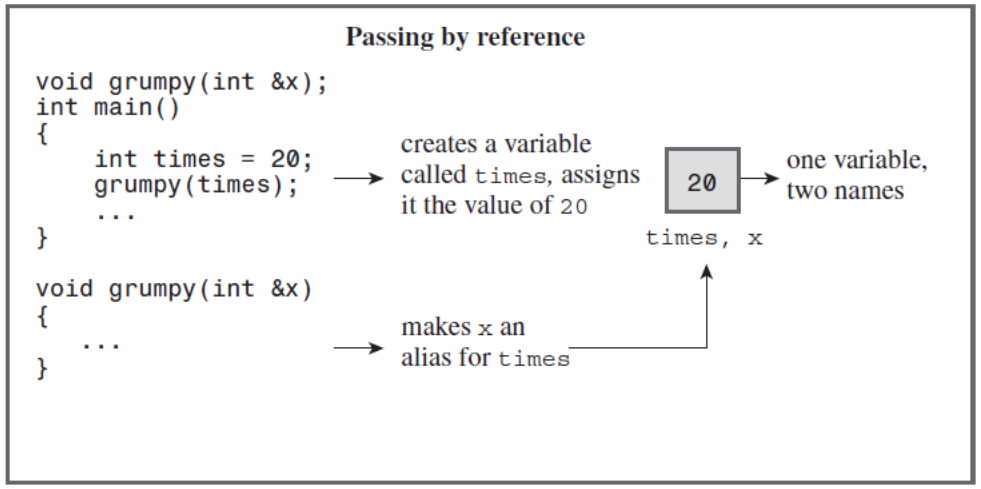
- The & symbol
  - Indicate the address of a variable
  - Declare references
    - ✓ int & means reference-to int
    - ✓ The reference declaration allows you to use two variables interchangeably
    - ✓ Both refer to the same value and the same memory location
  - It is necessary to initialize the reference when you declare it
- Run secref.cpp

References as Function Parameters

- Passing by reference allows a called function to access variables in the calling function

- Run swaps.cpp





Temporary Variables, Reference Arguments, and const

- The compiler generates a temporary variable
  - 1: When the actual argument is the correct type but isn't an lvalue
    - ✓ An lvalue is a data object that can be referenced by address including variable, array element, structure member, reference, dereferenced pointer
    - ✓ Non-lvalues include literal constants and expressions with multiple terms
  - 2: When the actual argument is of the wrong type, but it's of a type that can be converted to the correct type
  - The reason to use: temporary variables cause no harm
- A function with reference arguments is to modify variables
- Use const when you can (non-modifiable lvalue)
  - Protect you against programming errors that inadvertently alter data
  - Process both const and non-const actual arguments when omits const in the prototype
  - Generate and use a temporary variable appropriately

Using References with a Structure

- References were introduced primarily for use with C++'s user-defined types, not for use with the basic built-in types
- Run strc_ref.cpp
- Why return a reference?
  - Normal return value: involve copying the entire structure to a temporary location and then copying that copy
  - But with a reference return value, the returned value is copied directly to the variable in calling function, a more efficient approach
- Being careful about what a return reference refers to
  - Remember when returning a reference is to avoid returning a reference to a memory location that ceases to exist when the function terminates

Using References with a Class Object

- Run strquote.cpp
  - string class defines a char *-to-string conversion
  - A property of const reference formal parameters is that the original data cannot be modified from inside the function
  - A const string & parameter can handle a string object or a quoted string literal, a null-terminated array of char, or a pointer variable that points to a char (char * or const char *)

When to Use Reference Arguments

- Two main reasons for using reference arguments
  - To allow you to alter a data object in the calling function
  - To speed up a program by passing a reference instead of an entire data object (const)
- A function modifies data in the calling function
  - If the data object is a built-in data type, use a pointer (more clear)
  - An array, use your only choice: a pointer
  - A structure, use a reference or a pointer
  - A class object, use a reference

A function uses passed data without modifying it

- If the data object is small, such as a built-in data type or a small structure, pass it by value
- If the data object is an array, use a pointer because that's your only choice. Make the pointer a pointer to **const**
- If the data object is a good-sized structure, use a **const** pointer or a **const** reference to increase program efficiency
- If the data object is a class object, use a **const** reference.

Default Arguments

- How do you establish a default value?
  - You must use the function prototype
- When you use a function with an argument list, you must add defaults from right to left

```
int harpo(int n, int m = 4, int j = 5);          // VALID
int chico(int n, int m = 6, int j);              // INVALID
int groucho(int k = 1, int m = 2, int n = 3);    // VALID
```

- The actual arguments are assigned to the corresponding formal arguments from left to right; you can't skip over arguments
- Run left.cpp

Function Overloading

- Function overloading:
  - Let you use **multiple** functions sharing the **same** name
  - Comparison: in default argument, it can call the **same** function by using varying numbers of arguments
- C++ enables you to define two functions by the same name, provided that the functions have **different** signatures
  - Function signature: function's argument list
  - Signature can differ in the number of arguments or in the type of arguments, or both
- Run leftover.cpp

Function Templates

- Define a function in terms of a generic type
  - A specific type, such as int or double, can be substituted
  - The process is termed generic programming
  - The keywords **template** and **typename** (class)

```
template <typename AnyType>
void Swap(AnyType &a, AnyType &b)
{
    AnyType temp;
    temp = a;
    a = b;
    b = temp;
}
```

- Run funtemp.cpp
  - Apply the same algorithm to a variety of types

More Templates

- Overloaded templates
  - Solved problem: not all types would use the same algorithm in templates
  - Non-template functions take precedence over template functions
  - Need distinct function **signatures** (overloading)
- Run twotemps.cpp
- Template limitations
  - It's easy to write a template function that cannot handle certain types
  - In some cases, require **different** codes but the arguments would be the same

Specializations

- Explicit specializations
  - If the compiler finds a specialized definition that exactly matches a function call, it uses that definition without looking for templates.
- Third-Generation Specialization
  - A function name can have a non template function, a template function, and an explicit specialization template function, along with all overloaded versions
  - The prototype and definition for an explicit specialization should be preceded by template <> and should mention the specialized type by name
  - A specialization overrides the regular template, and a non template function overrides both
- Run twoswap.cpp

Instantiations and Specializations

- Instantiation
  - ➢ Template: merely a plan for generating a function definition
  - ➢ Instantiation: use the template to generate a function definition
    - ✓ Implicit: the compiler deduces the necessity for making the definition
    - ✓ Explicit: using the <> notation to indicate the type and prefixing the declaration with the keyword template

```
...
template <class T>
void Swap (T &, T &);  // template prototype

template <> void Swap<job>(job &, job &);   // explicit specialization for job
int main(void)
{
  template void Swap<char>(char &, char &); // explicit instantiation for char
  short a, b;
  ...
  Swap(a,b);    // implicit template instantiation for short
  job n, m;
  ...
  Swap(n, m);   // use explicit specialization for job
  char g, h;
  ...
  Swap(g, h);  // use explicit template instantiation for char
  ...
}
```

Which Function Version Does the Compiler Pick?

- Multiple functions of the same name
  - Include: function overloading, function templates, and function template overloading……
- Overload resolution
  - **Phase 1**—Assemble a list of candidate functions
  - **Phase 2**—From the candidate functions, assemble a list of feasible functions
    - ✓ Correct number of arguments
    - ✓ An exact match for each type of actual argument to the type of the corresponding formal argument
  - **Phase 3**—Determine whether there is a best viable function

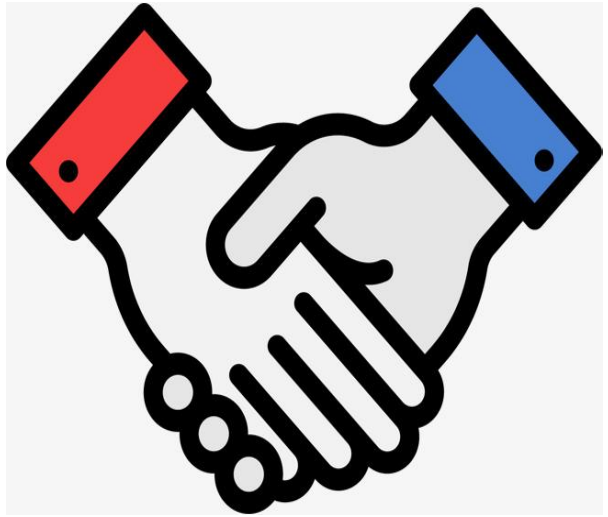
Exact Matches and Best Matches

- C++ allows some "trivial conversions" when making an exact match
  - If there's just one, that function is chosen
  - If more than one are tied, but only one is a non template function, that non template function is chosen
  - If more than one candidate are tied and all are template functions, but one template is more specialized than the rest, that one is chosen
- Error
  - If there are two or more equally good non template functions
  - If there are two or more equally good template functions, none of which is more specialized than the rest
  - If there are no matching calls, that is also an error

From an Actual Argument	To a Formal Argument
`Type`	`Type &`
`Type &`	`Type`
`Type []`	`* Type`
`Type (argument-list)`	`Type (*)(argument-list)`
`Type`	`const Type`
`Type`	`volatile Type`
`Type *`	`const Type *`
`Type *`	`volatile Type *`

Summary

- Inline function
- Reference variables
- Functions of the same name
  - Default arguments
  - Function overloading
  - Function template



Thanks

zhengf@sustech.edu.cn