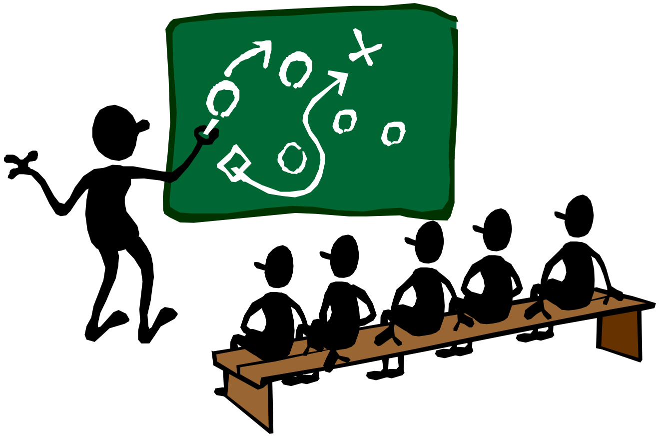
C++ Programming Language
Chapter 4 Parameters & Overloading



Juinn-Dar Huang
Associate Professor
jdhuang@mail.nctu.edu.tw

February 2011

Learning Objectives

- Parameters
  - call-by-value
  - call-by-**reference**
- Function overloading and default arguments
- Testing and debugging program
  - assert macro

Parameters

Two methods of passing arguments as parameters

- Call-by-value
  - only value is passed
  - used by C/C++, which you should be familiar with
- Call-by-reference
  - reference of actual argument is passed
  - i.e., address of actual argument is passed implicitly
  - parameter becomes an alias of actual argument
  - not available in C; should be NEW to most of you

Call-by-Value Parameters

actual argument $\xrightarrow{\textbf{value}}$ **formal parameter**

- Only the value of actual argument is passed
- Passed value is used to initialize formal parameter
- After initialization, they are decoupled
- Modifying formal parameter in callee won't alter actual argument in caller
  - callee has no access to actual argument of caller
- Call-by-value used in all examples of this course so far
- All C functions use call-by-value mechanism

Call-by-Value Example

```
void func(int a) {  a+= 5; cout << a << endl; }

int main() {
    int a = 10;
    func(a);      // guarantee: a won't be altered after function call
    cout << a << endl;
    // do something else
}
```

Output:

15

10

Call-by-Value Pitfall

- Common mistake:
  - declaring parameter again inside function:

```
double fee(int hoursWorked, int minutesWorked) {
  int quarterHours;                  // local variable
  int minutesWorked                  // error!, declare again !
}
```

  - compilation error
    - "Redefinition error…"

- Formal arguments ARE local variables
  - their scope extends to the end of function definition
  - they are initialized by actual arguments

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Issues Raised by Call-by-Value (1/2)

- Call-by-vale mechanism keeps callers safe
  - actual argument is guaranteed unaltered
- However, what if an actual argument is large?

```
struct my_struct {
    int arr[1000000];
};

void func(int x, my_struct y) { … }

int main() {
    int a;
    my_struct b;
    // a and b get initialized
    func(a, b);
    // …
}
```

x: 4 bytes ; y: 4 mega bytes
That is, call-by-value for y = b
will result in a 4MB memory copy
➔ memory & time consuming

Issues Raised by Call-by-Value (2/2)

- Solution in C programming ➔ Using **pointers** !

```
struct my_struct {
    int arr[1000000];
};

void func(int x, my_struct *y) { … }   // sizeof(mystruct *) = 4

int main() {
    int a;
    my_struct b;
    // a and b get initialized
    func(a, &b);
    // …
}
```

x: 4 bytes ; y: 4 bytes
still call-by-value; in this case,
pointer (address) value is explicitly passed
➔ memory & time efficient

actual argument (pointer) $\xrightarrow{\text{value}}$ **formal parameter (pointer)**

Call by Pointer Value

- However, variables in caller CAN be modified **indirectly**
  - better efficiency vs. poorer safety
  - good or bad? ; discuss later…
- Function can also return multiple values in this way

```
void func(int *x, int *y) {
    *x = 5; *y = 6;
}

int main() {
    int a = 1, b = 2;
    func(&a, &b);
    cout << a << endl << b << endl;
    // …
}
```

Dealing with integer pointer instead of plain integer
Any more elegant way?

Output:
5
6

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Call-by-Reference Parameters in C++

- Caller's actual arguments can be modified by callee!
- Formal parameter become alias of actual argument
- Specified by ampersand, &, after type

```
void func(int& x, int& y) {
    x = 5; y = 6;
}

int main() {
    int a = 1, b = 2;
    func(a, b);
    cout << a << endl << b << endl;
    // …
}
```

Dealing with integer (reference) instead of integer pointer

Easier to understand

Output:
5
6

Example Revisited

```
struct my_struct {
    int arr[1000000];
};

void func(int x, my_struct& y) { … }

int main() {
    int a;
    my_struct b;
    // a and b get initialized
    func(a, b);
    // …
}
```

actual argument —reference (address)→ **formal parameter**

x: 4 bytes
call-by-reference → y IS b
address value (4-byte long)
is implicitly passed

as efficient as pointer version
but more elegant

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Behind the Scene: Call-by-Reference

- What is really passed in?
- A reference back to caller's actual argument!
  - ➔ refers to memory location of actual argument
  - ➔ refers to address of actual argument
  - ➔ call-by-reference is an implicit call-by-pointer-value (compiler does tedious work for you)

Juinn-Dar Huang idhuang@mail.nctu.edu.tw

More about References

- A reference is an alternative name for an object
  - in my word: alias
- Generally, its main use
  - argument passing for functions // we have already seen
  - return values for functions // later chapters
- It can still be used inside a function though

```
void func() {
    int i = 2;
    int& r1 = i; // r1 is now an alias of i
    int x = r1;             // x = 2
    r1= 3;                  // i = 3
    int& r2;                // error, reference MUST be initialized
    int *pp = &r1;          // pp points to i
    *pp = 4;                // i = 4;
}
```

Constant Reference Parameters

- Reference arguments are inherently dangerous
  - caller's variables can be changed
  - this may surprise callers and introduce bugs
  - **issue**: better efficiency vs. poorer safety
- Want both safety and efficiency?
  ➔ use call-by-constant-reference !

  void sendConstRef(const int& par1, const int& par2);
  - changes to constant references are **NOT** allowed
  - ensures passed arguments **read-only** in callee
- **Suggestion**:
  - function should avoid modifying arguments through references
  - really want to modify caller's variables? ➔ use pointers instead!

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Parameters & Overloading

Example Revisited Again (1/2)

```
struct my_struct {
    int arr[1000000];
};

void func(int x, const my_struct& y)
{ … }  // won't modify y

int main() {
    int a;
    my_struct b;
    // a and b get initialized
    func(a, b); // b is unaltered
    // …
}
```

勝

New C++ Style

```
struct my_struct {
    int arr[1000000];
};

void func(int x, const my_struct *y)
{ … } // won't modify *y

int main() {
    int a;
    my_struct b;
    // a and b get initialized
    func(a, &b); // b is unaltered
    // …
}
```

Old C Style

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Example Revisited Again (2/2)

```
struct my_struct {
    int arr[1000000];
};

void func(int x, my_struct& y)
{ … }  // can modify y

int main() {
    int a;
    my_struct b;
    // a and b get initialized
    func(a, b); // b can be unaltered
    // …
}
```

New C++ Style

```
struct my_struct {
    int arr[1000000];
};

void func(int x, my_struct *y)
{ … } // can modify *y

int main() {
    int a;
    my_struct b;
    // a and b get initialized
    func(a, &b); // b can be unaltered
    // …
}
```

勝

Old C Style

Parameters & Overloading

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Mixed Parameter Lists

- A function can use both argument passing mechanisms
  - parameter lists can include pass-by-value and pass-by-reference parameters
- Order of arguments in list is critical:

  void mixedCall(int& par1, int par2, double& par3);
  - function call:

    mixedCall(arg1, arg2, arg3);
    - arg1 must be an integer variable, is passed by reference
    - arg2 must be integer type (variable or constant) , is passed by value
    - arg3 must be a double variable, is passed by reference

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Motivation for Function Overloading

- Math function abs in C++ library
  - int abs(int) // returns absolute value of an int
  - long labs(long) // returns absolute value of a long int
  - double fabs(double) // returns absolute value of a double

- Above functions all perform abs operations; they just deal with parameter of different types
- It is annoying to find different names for them

- Is it possible to assign those functions an identical name?
  - you cannot in C

Function Overloading

- 2 or more functions with a same name but different parameter lists
  - they have their own function definitions
- Every function MUST have a unique function signature
  - in C: function name only
  - in C++: function name + **parameter list**

Language objective:

- Allows several functions performing conceptually the same task with different parameters having same name
  - they still have their own unique signature

Overloading Example: Average

- Function computes average of 2 numbers:

```
double average(double n1, double n2)
{
    return ((n1 + n2) / 2.0);
}
```

- Now compute average of 3 numbers:

```
double average(double n1, double n2, double n3)
{
    return ((n1 + n2 + n3) / 3.0);
}
```

- Same name but two different functions
- Function name average is **overloaded**

Overloaded Average()

- Which function gets called?
- Depends on function call itself:
  – avg = average(5.2, 6.7); ➔ two-parameter average()
  – avg = average(6.5, 8.5, 4.2); ➔ three-parameter average()
- Compiler resolves invocation based on signature of function call
  – matches call with appropriate function
- In C++, the following functions are all in library
  – int abs(int);
  – long abs(long);
  – float abs(float);
  – double abs(double);
  – long double abs(long double);

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Overloading Pitfall

- Overloaded functions should perform a **conceptually same** task
  - all abs() functions should always perform getting absolute value
  - DON'T overload an abs() performing other task else

Overloading Resolution

Rules for resolving overloaded functions

- 1st: Exact match
  - Looks for **exact** signature where no argument conversion required
- 2nd: Compatible match (simplified version)
  - looks for **compatible** signature where automatic type conversion is possible:
    - 1st with promotion (e.g.,char ➔ int, float ➔ double, …)
    - 2nd with standard conversion (e.g., int ➔ double, double ➔ int, …)
  - **NOTE**: rules described for compatible match here is general **BUT NOT PRECISE and COMPLETE** enough
  - resolution relies on an extremely complicated set of rules!

Overloading Resolution Example (1/2)

- Example

```
void func(double);
void func(int);
void f() {
    char ch = 1;
    short s = 1;
    func(ch);        // OK, call func(int)
    func(s);         // OK, call func(int)
    func(1);         // OK, call func(int)
    func(1U);        // compilation error, ambiguity
    func(1L);        // OK, call func(int)
    func(1UL);       // compilation error, ambiguity
    func(1.0F);      // OK, call func(double)
    func(1.0);       // OK, call func(double)
    func(1.0L);      // compilation error, ambiguity
}
```

Do you really understand why?
Be very careful if you don't't!

Resolution for Multiple Arguments

- Given following functions

  1: void f(int n, double m);
  2: void f(double n, int m);
  3: void f(int n, int m);

  – following calls:
  f(98, 99); ➔ calls #3; exact match
  f(5.3, 4); ➔ calls #2; exact match
  f(4.3, 5.2); ➔ calls #?; ambiguity

- Avoid such confusing overloading
- Think twice while overloading for different numeric types

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Overloading and Return Type

- Functions can NOT be overloaded just by differentiating return type
  - int func1(int); int func1(double); // OK
  - int func2(int); void func2(int); // compilation error

- C++ standard wants to keep resolution for function call context-independent

```
float sqrt(float);
double sqrt(double);

void func(double da, float fa) {
    float f = sqrt(da);        // call sqrt(double); no ambiguity
    double d = sqrt(da);       // call sqrt(double)
     f = sqrt(fa);             // call sqrt(float)
    d = sqrt(fa);              // call sqrt(float); no ambiguity
}
```


Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Default Arguments

- Allows functions omitting some arguments
- Default arguments are specified when function is declared (or defined if that occurs first)
  - bottom line: declaration-before-use

```
void print(int value, int base = 10); // default base is 10
```

  - possible calls:
    - print(31);   // print 31 as a decimal
    - print(31, 10); // same output as above
    - print(31,16);   // print 31 as a hexadecimal

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Default Arguments Example (1/2)

Display 4.8 Default Arguments

Default arguments

```

#include <iostream>
using namespace std;

void showVolume(int length, int width = 1, int height = 1);
//Returns the volume of a box.
//If no height is given, the height is assumed to be 1.
//If neither height nor width is given, both are assumed to be 1.

int main( )
{
    showVolume(4, 6, 2);
    showVolume(4, 6);
    showVolume(4);

    return 0;
}

void showVolume(int length, int width, int height)
```

A default argument should not be given a second time.

Default Arguments Example (2/2)

```
{
    cout << "Volume of a box with \n"
         << "Length = " << length << ", Width = " << width << endl
         << "and Height = " << height
         << " is " << length*width*height << endl;
}
```

SAMPLE DIALOGUE

Volume of a box with
Length = 4, Width = 6
and Height = 2 is 48
Volume of a box with
Length = 4, Width = 6
and Height = 1 is 24
Volume of a box with
Length = 4, Width = 1
and Height = 1 is 4

Juinn-Dar Huang   jdhuang@mail.nctu.edu.tw

More about Default Arguments (1/2)

- Default arguments are provided for **trailing** arguments only

```
void func(int, int = 0, char* = 0);      // ok
void g(int = 0; int = 0, char*);         // error
void h(int = 0, int, char* = 0);         // error
```

- Must omit arguments starting from the right

```
func(3, 2);             // ok, func(3, 2, 0)
func(3, , p_char);      // error
```

- Note: the effect of default argument can also be achieved by function overloading

```
void print(int value, int base);
inline print(int value) { print(value, 10); }  // like default argument
```

More about Default Arguments (2/2)

- Default arguments cannot be repeated or changed in a subsequent declaration

```
void f(int);
void f(int);            // ok, redeclaration
void g(int);
void g(int = 10);       // ok, give default argument
void g(int);            // ok
void g(int = 10);       // error, cannot repeat default argument
void g(int = 20);       // error, cannot change default argument
```

- Be aware of potential ambiguous calls

```
void func(int = 1, int = 2, int = 3);
void func(int, int);
void f () {
    func(3, 6);        // error, ambiguous call, func(3, 6, 3) or func(3, 6) ?
}
```


Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Testing and Debugging Functions

- Many methods
  - lots of cout statements
    - in calls and definitions
    - used to trace execution
  - built-in debugger provided by compiler
    - environment-dependent
  - assert macro
    - early termination as needed

assert Macro

- Assertion: a statement that is either true or false
- Used to document and check correctness
  – e.g., check preconditions and postconditions
  – typical use: confirm validity of asserted conditions
  – syntax
  assert(assert_condition);
    - no return value
    - evaluates assert_condition
    - terminates if false, continues if true
- Predefined in library <cassert>
  – macros used similarly as functions

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

assert Example

- A function translating a number to a hexadecimal digit

```
#include <cassert>
char int_to_hexchar(int num) {
    assert( (num >= 0) && (num <= 15) );  // check precondition here
    if(num < 10)
        return num + ‘0’;
    else
        return num - 10 + ‘A’;
}
```

- Check precondition
  - If precondition is not satisfied ➔ **assert_condition** is false ➔ program execution terminates!

Stops execution so problem can be investigated
Very useful in debugging

assert On/Off

- Preprocessor provides means

  #define NDEBUG
  #include <cassert>

- Add "#define" line before #include line
  - turns assertions OFF
- Remove "#define" line (or comment out)
  - turns assertions ON

Summary (1/2)

- Argument passing mechanism
  - call-by-value
  - call-by-pointer-value
  - call-by-reference
  - call-by-constant-reference

- Function overloading
  - know when to use it
  - signature: function name + parameters
  - regardless of return type
  - resolution rules

Summary (2/2)

- Default arguments
  - allow function call to omit some arguments
  - if not provided ➔ default values assigned
  - only for trailing arguments
  - omit arguments starting from the right in function call

- assert
  - terminates program if assertions fail
  - used to guard invariants
    (e.g., pre- and post- conditions)

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw