C++ As a Better C

C++ Object Oriented Programming

Pei-yih Ting
NTOU CS

Contents

- ✧ Comments
- ✧ User-defined type names
- ✧ Function prototypes in C++
- ✧ Function signatures
- ✧ Better I/O library
- ✧ Default function arguments
- ✧ Macros
- ✧ Inline functions
- ✧ #define vs. constant variables
- ✧ new and delete operators
- ✧ Reference variables
- ✧ Stricter typing systems

Comments

- Comments in C++ vs. C

```
/* You can do this
   across multiple lines */
//  Or you can do this on a single line
```

ends at end of line

- Advantages of //
  - What's the problem?

```
if (b>a)
   return b; /* could be also b>=
else
   return a; /* note that we return a in case of a tie */
```

missing */

  - Solution with //

```
if (b>a)
   return b; // could be also b>=
else
   return a; // note that we return a in case of a tie
```

- Rules:
  - Use // syntax for single-line comments
  - Use /*…*/ syntax for multi-line comments

User-defined Type Names

- struct, enum, union tags are type names
  - struct:
    ```
    struct Stack {
      …
    };
    ```
    - C: struct Stack operatorStack;
    - C++: Stack operatorStack;
  - union:
    ```
    union Value {
      int iValue;
      double dValue;
    };
    ```
    - C: union Value field;
    - C++: Value field;
  - enum:
    ```
    enum Color {RED, GREEN, BLUE};
    ```
    - C: enum Color bgColor;
    - C++: Color bgColor;

```
typedef struct tag
{
  …
} Stack;
Stack operatorStack;
```

Function Prototypes in C++

- Function prototypes are REQUIRED
  - Otherwise you must define the function before you use it, i.e. in Pascal-style
  - In K&R C (before ANSI C), a function *foo* used without suitable prototype has default prototype, arguments are passed with default promotion rules (i.e. 4bytes / 8bytes rule)

```
int foo();
```

- void as an argument in C prototypes
  - What do the following 2 prototypes differ in traditional C? (K&R C)

```
int foo(void);
int foo();
```

A function foo that takes an indeterminate number of arguments

  - In C++, the above two are equivalent. The second one is preferred.
- The notorious **variable argument list**, represented by ellipses (…)
  - int printf(const char *format, …);

C++ still keep it for compatibility

Function Signatures

- C: a function is identified completely by its name

  C++: a function is identified by its signature (name, #params, types of params and const modifier)

- Ex. in C,

```
void draw(int) {}
void draw(double) {}
```

  error: 'void __cdecl draw(int)' already has a body

  in C++, both the above two are OK, compiler encodes the function name with type safe linkage rules (called *name mangling*)

- Note: Function return type is not a part of its signature

  Access privilege is not a part of its signature

- C++ calls a C function:

```
extern "C" int func(int *, float);
```

- C calls a C++ function:

```
extern "C" {
   int fun(int *, float){….};
}
```

  or

```
extern "C" int fun(int *, float){
…
}
```

in C++ file

Better Input/Output

- Type-aware I/O processing, mixed data types

  int x = 5; double y = 6.0; char *s = "Hello";

  - C output: printf("x=%d y=%f s=%s", x, y, s);
  - C++ output: cout << x << y << s;
  - C input: scanf("%d%f%s", &x, &y, s);
  - C++ input: cin >> x >> y >> s; (no ampersand & trap)
- Header file: iostream
- Insertion operator: <<, inserts data into the output stream
- Extraction operator, >>, extracts data from the input stream
- Errors:
  - cout >> age;
  - cin << age;
- Mix C stdio with C++ iostream: ios::sync_with_stdio(); ← not preferred
- cerr
- clog

Default Function Arguments

- Function arguments can be given default (optional) values.

```
void printName(char *first, char *last, bool inverted=true);
void main() {
   char firstName[50]="Joe", lastName[50]="Smith";
   printName(firstName, lastName);
   printName(firstName, lastName, false);
}
…
void printName(char *first, char *last, bool inverted) {
   if (!inverted)
      cout << first << ' '<< last << '\n';
   else
      cout << last << ' '<< first << '\n';
}
```

specified only in the prototype, OK to differ in different scopes

Good for avoiding seldom-used parameters

- Rules:
  - Can have any number of default arguments
  - Default arguments must come after non-default arguments, and not the other way around

Macros

- Preprocessor macro introduces subtle bugs if not careful

```
#define square(x) (x*x)
void main() {
   int x=5, y;
   y = square(x);
   cout << y;
}
```

Output: 25

- The problem with macros
  - The preprocessor knows nothing about C syntax or semantics
  - Cannot debug into a macro function (a macro is invisible to the compiler/ debugger)
- The same macro fails on the following

```
int x=5, y=6;
cout << square(x+y);
```

Output: 41

- Corrections

```
#define square(x) ((x)*(x))
```

Macros (cont'd)

- Not every macro problem can be solved by parenthesizing

```
#define inverse(x) (1/(x))
double x=5;
cout << "x=" << inverse(x) << endl;
int y=5;
cout << "y=" << inverse(y) << endl;
```

Output:
x=.2
y=0

- Corrections:

```
#define inverse(x) (1.0/(x))
```

- Arguments of a macro could be evaluated more than once

```
#define square(x) ((x)*(x))
…
int x=5;
cout << "square of 5 is " << square(x++) << ", x=" << x;
```

Output:
square of 5 is 30, x=7

- There are various problems accompanying macros. They all require prudent inspections.

Inline Functions

- C++ has inline functions, which provide the same functionality as macros without the above drawbacks

```
inline int square(int x); // function prototype, not a macro
void main() {
   int x=5, y=6;
   cout << square(x+y);
}
inline int square(int x) { return x * x; }
```

Output: 121

```
inline double inverse(double x);
void main() {
   int x=5;
   cout << inverse(x);
}
inline double inverse(double x) { return 1 / x;}
```

Output: .2

- The compiler can only inline **simple** functions (compiler-dependent) and will IGNORE all other inline requests.

Declare Variables On-the-fly

- C: Local variables must be declared at the beginning of a block.
  C++: Local variables can be declared anywhere inside a block, the scope extends to the end of the block.
- Ex.

```
void main() {
    int array[5] = {0, 1, 2, 3, 4};
    cout << array[0] << endl;
    …
    int sum = 0;                                          } the scope
    for (int i=0; i<5; i++)        } the scope               of sum
        sum += array[i];             of i
    cout << sum;
}
```

- Why should you do this? better readability
  encourages single-usage variables
  - Most commonly used for temporary loop variables

Declare Variables On-the-fly (cont'd)

- Cannot branch over 'a variable definition with initialization'

error

```
void main()
{
    int x;
    x = 1;
    goto test;
    int y=5;
test:
    x = 2;
    y = 10;
}
```

```
void main()
{
    int x;
    x = 1;
    goto test;
    int y;
test:
    x = 2;
    y = 5;
}
```

error

```
void main()
{
    int x=1;
    switch (x) {
    case 1:
        int y=5;
        break;
    case 2:
        y=10;
        …
    }
}
```

```
void main()
{
    int x=1;
    switch (x) {
    case 1:
        int y;
        break;
    case 2:
        y=10;
        …
    }
}
```

Compilation OK, but better not do this, use suitable block structure instead

#define vs. const

- **Defines should be replaced by constant variables in C++**

```
#define kMaxSize 1000            // do not do this
const int kMaxSize = 1000;       // much better
int array[kMaxSize];
```

- **A constant variable is a real variable, therefore, has a type that compiler can check upon, and is visible to the debugger.**
- **Constant arguments promise more: a const argument tells the client that the argument will not be changed and the compiler guarantees that it won’t**

```
static bool isStartWithH(const char *inputString) {
    char firstLetter = inputString[0];
    firstLetter = toupper(firstLetter);
    return firstLetter == 'H';
}
```

Usually used with pointer or reference parameters

```
int size;
cin >> size;
const int kMax = size;
int array[kMax];
```

Compiler guarantees that the following won’t happen

```
static bool isStartWithH(const char *inputString) {
    inputString[0] = toupper(inputString[0]);
    return inputString[0] == 'H';
}
```

More on Constant Variables

- ‘const’ modifies the type specifier differently according to its position

```
void main()
{
    char string1[kMaxSize] = "Hello world";
    char string2[kMaxSize] = "Good bye";
    string1[0] = 'T';                    // legal

    const char *ptrString1 = string1;
    ptrString1[0] = 'T';                 // illegal
    ptrString1++;                        // legal

    char *const ptrString2 = string1;
    ptrString2[0] = 'T';                 // legal
    ptrString2++;                        // illegal
    ptrString2 = string2;                // illegal
    char *const ptrString3;              // illegal

    const char *const ptrString4 = string1;
}
```

char is a constant, char* is not

char* is a constant, char is not

both char and char* are constants

‘static’ modifier in C

- Different semantics with 3 types of usages:

  I. global scope variable: static int g_data;

  II. global scope function: static int func(int x, float y) {…}

  III. local scope variable:
  ```
  int func() {
     static int localData;
     …
  }
  ```

  Identifier scope is restricted to a file

  The life cycle of this variable extends over multiple calls of this function

- File scope variables and functions: type I and type II above
  - their scopes are restricted to the file unit in which they are declared
  - used in C to encapsulate a module, i.e. make that identifier local to a file

file1.c

```
static int x1;
int x2;
static int func1(int x) { … }
int func2(int x) { … }
```

file2.c

```
int func() {
   extern int x1; int func1(int);
   func1(x1);  // both undefined
   func2(x2);  // OK
}
```

- In C++, these semantics remain the same. Besides, constant variables are implicitly static.

New Ways to Handle Memory

- C++ has better ways to allocate/deallocate memory

C	**malloc**	**free**
C++	**new, new[]**	**delete, delete[]**

- Ex.

```
int *x, *y;
int *array;
x = new int;
y = new int(40);
array = new int[100];
delete x;
delete y;
delete[] array;
```

initialization: single-value variables (not for arrays) and objects

new and delete are built-in operators
no #include file necessary

- Why does C++ switch to these new usages?
  - Simplicity: C: array = (int *) malloc(sizeof(int)*100);
    C++: array = new int[100];
  - Auto initialization and clean-up
  - Consistency with C++ object allocation

new / delete Usages

- Errors due to unmatched allocation/deallocation
  - int *x1=new int; … delete[] x1;
  - int *x2=new int[100]; … delete x2;
  - int *x3=new int; … free(x3);
  - int *x4=(int *) malloc(sizeof(int)); … delete x4;
- Special safety checks

```
int *ptr=0;
…
if (!ptr) free(ptr);  // freeing null is fatal in C/C++
delete ptr;           // OK to delete null
```

  - better erase the pointer after deletion (good coding practice)

```
delete ptr; ptr= 0;
```

- Multi-dimensional array: (actually 1-dim data)

```
int (*xp)[3] = new int[20][3];  … delete[] xp;
  or equivalently
typedef int IARY[3]; IARY *xp=new IARY[20]; … delete[] xp;
```

Handling Memory Allocation Errors

- malloc():

```
int *ptr=(int *) malloc(sizeof(int)*200);
if (ptr==0) printf("Memory allocation error!!");
```

- new:

```
int *ptr=new int;
if (ptr==0) printf("Memory allocation error!!");
```

- You can also specify a function to be called in case of memory failure. **Corrective actions** such as freeing memory space can be taken automatically and the **new** operation can be retried.
- Ex.

```
static int newFailed(size_t size) {
    if (gSparePtr!=0) {
        delete [] gSparePtr; // free some spare space
        gSparePtr = 0;
        cout << "[newFailed " << size << "]";
        return 1; // request the new operator to retry
    }
    return 0; // stop retrying
}
```

Handling Memory Allocation Errors

- Installing and resetting the new handler VC6.0

```
#include <new.h>
int *gSparePtr = 0;
static int newFailed(size_t size);
void main() {
    int *ptr[20], i, *spoiled;
    _PNH old_handler = _set_new_handler(newFailed);
    spoiled = new int[150000000];
    gSparePtr = new int[20000000];

    for (i=0; i<20; i++) {
        cout << i << " ";
        ptr[i] = new int[5000000];
        cout << ptr[i] << endl;
    }

    _set_new_handler(old_handler);
}
```

restore the original new handler
can also call _set_new_handler(0) to remove

0 28CB0020
1 29FD0020
2 2B2F0020
3 2C610020
4 2D930020
5 2EC50020
6 2FF70020
7 31290020
8 325B0020
9 338D0020
10 [newFailed 20000000]
34BF0020
11 35F10020
12 37230020
13 38550020
14 00000000
15 00000000
16 00000000
17 00000000
18 00000000
19 00000000

Handling Memory Allocation Errors

- ANSI C++ version of set_new_handler

```
#include <new>
using namespace std;
…
static void newHandler();
…
void main() {
    new_handler old_handler=set_new_handler(newHandler);
    …
    set_new_handler(old_handler);
}
…
static void newHandler() {
…
}
```

In VC6.0 this does not work, because set_new_handler() is implemented as a stub function only.

References

- C simulates "call by reference" through pointers

```
void func(int *ptrData) {
    *ptrData = 10;
}
```

```
void main() {
    int data;
    …
    func(&data);
    …
}
```

- C++ has true references

```
void func(int &param) {
    param = 10;
}
```

no pointer dereference required

```
void main() {
    int data;
    …
    func(data);
    …
}
```

- Some C++ programmers might do the following for saving time and memory

```
void Foo(const CBigData &data) {
…
}
```

References (cont'd)

- There are no promotions or type conversions with references

```
void func(double &data) {
    data = 10;
}
```

```
void main() {
    int data;
    …
    func(data);
    …
}
```

error C2664: 'func' : cannot convert parameter 1 from 'int' to 'double &'

- A reference variable cannot be bound to a temporary object

```
int getValue() {
    int tmp;
    return tmp;
}
int func(int &value);
void main() {
    func(getValue());
}
```

error C2664: 'func' : cannot convert parameter 1 from 'int' to 'int &'

Stricter Typing System

- In C, you can do

Giving up the advantages of strict type checking

```
int *intPtr;
void *genericPtr;
genericPtr = intPtr;          // convert typed pointer to generic pointer
intPtr = genericPtr;          // generic to typed
```

- In C++,

```
int *intPtr;
void *genericPtr;
genericPtr = intPtr;          // convert typed pointer to generic pointer
intPtr = genericPtr;          // ERROR: cannot convert from 'void *' to 'int *‘
intPtr = (int *) genericPtr;  // explicit type cast
```

- In C++, char literal is not treated as int as in C

```
void func(int i);
void func(char c);
…
```

overloaded functions

func('A') will invoke the second function

Miscellaneous

- Scope resolution operator

```
static int x = 10;
void main() {
    int x = 5;
    cout << x << endl;
    cout << ::x << endl;
}
```

```
Output:
5
10
```

- bool
  - A new type of boolean variable
  - The value can be true or false

Explicit Type Conversion

- C style type casting operator (type coercion)

  ```
  int b = 200;
  unsigned long a = (unsigned long int) b;
  ```

  Basically commands the compiler to "forget about type checking" – introduction a hole in the C/C++ type checking system.

- C++ style explicit casts: (including Run-time type information, RTTI)
  - **static_cast**: for well-behaved and reasonably well-behaved casts, ex. int to float, float to int, forcing a conversion from a void*
  - **const_cast**: to cast away const or volatile, i.e. make a const variable non-const
  - **reinterpret_cast**: cast one type to whatever types you like, most dangerous
  - **dynamic_cast**: for type-safe downcasting

Explicit Type Conversion

```
int i; float f;
…
void *vp = &i;
float *fp = static_cast<float *>(vp);
i = static_cast<int>(f);
```

```
const int i = 0;
j  = const_cast<int*>(&i); // Preferred
*j = 10;
cout << "i=" << i << " *j=" << *j << endl;  // weird, compiler make it 0
                                            // directly in the code
```

Output:
i=0 *j=10

```
struct X { int a[100]; } x;
…
int *xp = reinterpret_cast<int *>(&x);
```

Usage of *typedef*

- typedef is used to define a convenient name for any type in C/C++; in many cases, it clarifies the definition
  - typedef int INT32; // defines the alias name INT32 for int
    INT32 var; // is equivalent to int var;

    Merging these two statements!!
  - typedef struct tagBook {
        char author[50];
        char title[50];
    } Book;   // defines the alias name Book for struct tagBook
    Book book; // is equivalent to struct tagBook book;
  - typedef int IntArray[100]; // defines the alias name IntArray
    IntArray data; // is equivalent to int data[100];
  - typedef double (*FP)(int, double *); // defines the alias name FP
    FP fptr; // is equivalent to double (*fptr)(int, double *);