Reference

C++ Object Oriented Programming
Pei-yih Ting
NTOU CS

Contents

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 dereference operator required

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

no address operator required

It is also the goal of C++ to reduce the usage of pointers.

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

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


References (cont'd)

- There are NO type 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 bind to a temporary object (r-value)

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

int func(const int &value) is OK

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

References as Aliases

- A reference is an **alias to another variable (lvalue).**

```
void main {
   int x = 5;
   int &alias = x;
   cout << "The value of x is " << x << endl;
   cout << "The value of the alias is " << alias << endl;
   alias = 10;
   cout << "The value of x is " << x << endl;
   cout << "The value of the alias is " << alias << endl;
}
```

lvalue is an expression that refers an object, e.g. variables, array cells, or dereferenced pointers, that persists beyond a simple expression

- Like a constant variable, the reference must be initialized in its declaration.

```
int x = 5;
int &alias;
alias = x;
```

Error: 'const' or '&' variable needs initializer

Note: Initialization and assignment are very different.

References are not Pointers

- Cannot be reassigned

```
int &alias = x;
int &alias = y;
```

Error: identifier ‘alias’ re-declared.

- Not related to concept of memory addresses any more

```
int x = 5;
int y = 5;
int &aliasX = x;
int &aliasY = y;
if (aliasX == aliasY)
   cout << "identical.\n";
else
   cout << "different\n";
```

Output: identical

comparing the contents of x and y

```
int x = 5;
int y = 5;
int *ptrX = &x;
int *ptrY = &y;
if (ptrX == ptrY)
   cout << "identical.\n";
else
   cout << "different\n";
```

Output: different

comparing the addresses of x and y

References are not Pointers (cont’d)

- You cannot obtain the address of a reference

```
int x = 5;
int *ptr;
int &alias = x;
ptr = &alias;
```

There are only two variables in this code segment. *ptr* contains the address of *x* (not the address of *alias*, and indeed *alias* itself is not a variable)

- No similar thing as pointer arithmetic

```
int array[] = {3, 2, 1};
int &alias = array[0];
alias++;
cout << alias << '\n' << array[0] << '\n';
```

Output:
4
4

- Can you alias a pointer variable? Yes

```
void main() {
   char *string = "hello";
   Foo(string);
   cout << string;
}
```

```
void Foo(char* &strPtrRef) {
   strPtrRef = "good day";
}
```

Output:
good day

Function Returning a Reference

- Assuming that you want to emulate a Pascal-style 1-based array:

```
int &pArray(int cArray[], int index) {
   return cArray[index-1];
}
void main() {
   int array[] = {1, 2, 3};
   cout << pArray(array, 2) << '\n';
   pArray(array, 1) = 10;
   cout << pArray(array, 1) << '\n';
}
```

Output:
2
10

- Thus, you can use the ‘function call’ as an l-value.

Returning a Reference (cont'd)

- Why is the following code not working?

```
int &pArray(int index) {
   int cArray[] = {1, 2, 3};
   return cArray[index-1];
}
void main() {
   cout << pArray(2) << '\n';
   pArray(1) = 10;
   cout << pArray(1) << '\n';
}
```

Output:
2
1

Reference Saves Computation

- Like the usage of pointers, references used for function arguments can save computation time in copying data.

```
BigDataT x, y;
…
Foo(x, y)
…

void Foo(const BigDataT &inputData, BigDataT &ouputData)
{
  …
  inputData.accessor(); // access the aliased variable by inputData, i.e. x, directly
  …                     // without changing it
  outputData.mutator(); // access y directly and modify its value
  …
}
```


References as Data Members

```
double gCustomerCreditLimit = 1000;
…
class Patron {
public:
   Patron(double &limit);
   void Charge(double amount);
private:
   const double &fCreditLimit;
};
…
Patron::Patron(double &limit): fCreditLimit(limit) {
}
…
Patron patron(gCustomerCreditLimit);
…
```

Initialization-list:
the only way to initialize
a reference member variable
or a const member variable

The Hidden Perils of C++

```
class String {
public:
   String();
   String(const char *inputStr);
   ~String();
   const char *GetString() const;
private:
   char *fString;
};

String::String(char *inputStr) {
   fString = new char[strlen(inputStr)+1];
   strcpy(fString, inputStr);
}

String::~String() {
   delete[] fString;
}
```

```
void main() {
   String string1("Hello");
   String string2 = string1;

   cout << string1.GetString() << endl;
   cout << string2.GetString() << endl;
}
```

This piece of code often makes your program crash. The lack of explicit **copy constructor** creates two pointers for the same piece of memory.

Copy Ctor X(X&), X(const X&)

- Definition of a **copy constructor**

```
String(const String &src) {
    fString = new char[strlen(src.fString)+1];
    strcpy(fString, src.fString);
}
```

- It is necessary that the copy constructor use reference as parameter. Without reference parameter, it would cause recursive invocations with any call by value parameter .
- Implicit usage of a copy constructor
  1. String string2 = string1;
  2. String string2(string1);
  3. Calling a function fun(string1); and returning an object.

```
void fun(String stringParam) {
    …
}
```


Array of References is Illegal

```
void fun(int &array[]) {
   int i;
   for (i=0; i<10; i++)
      array[i] = i;
}
```

```
void main() {
   int i, array[10];
   fun(array);
}
```

```
testRef.cpp(4) : error C2234: '<Unknown>' : arrays of references are illegal
testRef.cpp(8) : error C2440: '=' : cannot convert from 'int' to 'int *'
      Conversion from integral type to pointer type requires reinterpret_cast,
      C-style cast or function-style cast
```

```
void fun1(int **&dptr) {
   dptr = (int **) new int*[10];
}
void fun2(int ***tptr) {
   *tptr = (int **) new int*[10];
}
```

```
void main() {
   int **doublePtr1, **doublePtr2;
   fun1(doublePtr1);
   fun2(&doublePtr2);
}
```