Comp 151

Reference Variables

Creating a Reference Variable

- A reference is an alternative name (alias) for an object. Reference variables are usually used in parameter passing to functions. Before looking at how to use them as parameters we first examine their stand-alone properties.

```
int j = 1;
int& r = j;          // now r = j = 1
int x = r;           // now x = 1
r = 2;               // now r = j = 2
```

- The notation "`int& r = j;`" means that r is a reference variable that is another name for `j`.
- A reference allows indirect manipulation of an object, somewhat like a pointer, without requiring complicated pointer syntax.

- A reference must always be bound to an object. It must therefore be initialized when it is created:

```
int j = 1;
int& r1 = j;          // ok
int& r2;              // error!
```

- A reference is always implicitly const in C++ (unlike many other languages, including Java).
  - Unlike a pointer, in C++ a reference can never be “redirected”. It always refers to the same object it was first initialized to (much like a const pointer).
  - On the other hand, the object it refers to need not be const; you can change the object’s value…

Initialization & Assignment

- Distinguish between *initializing* a reference versus making an *assignment* to it:

```
int j = 10;
int& r = j;

++r;                    // both r and j become 11
j += 8;                 // both r and j become 19
r = 7;                  // both r and j become 7

cout << j << "," << &j << endl;
cout << r << "," << &r << endl;
```

- Remember: A reference is implicitly const, and cannot be "redirected". It always refers to the object it was first initialized to.
- Assignment only changes the "value" of its referenced object (not the address).
- Even the address of a reference is that of the referenced object! I.e., `&r == &j`. In C++, the reference `r` is <u>not</u> itself an object that lives in some memory location (unlike a pointer).

The Different Uses Of &

- Do not confuse the use of & in a reference assignment, e.g.,

```
float& i = j;
```

  versus the use of & as the address of operator, e.g.,

```
float j;
float* pi = &j;
```

- The following is wrong. Why?

```
float j;
float &i = &j;
```

- The following is correct. What does it mean?

```
float j;
float* pi = &j;
float*& ref = pi;
```

```
#include<iostream>
using namespace std;
void main()
{
        int j=1;                    // j is an int
        int* pi = &j;               // pi is an int* initialized to address of j
        int*& ref = pi;             // ref is ref variable of type int*

        cout << "j = " << j;
        cout << "*pi = "<< *pi << " *ref =" << *ref << endl;

        int k =2;
        pi = &k;
        cout << "j = " << j;
        cout << "*pi = " << *pi << " *ref =" << *ref << endl;
}
```

Call-By-Reference and Reference Arguments

- Reference arguments are just a special case of references:

```
int f(int& i) { ++i; return i; }
int main() { int j = 7; cout << f(j) <<endl; cout << j <<endl; }
```

- Variable `i` is a local variable in the function `f`.
- Its type is "`int` reference" and it is created when `f` is called.
- In the call `f(j)`, `i` is created similarly to the construction:

```
int& i = j;
```

- So within the function `f`, `i` will be a (implicitly const) reference that serves as an alias of the variable `j`, and that cannot be changed while the variable `i` exists.
- But every time the function `f` is called, a new variable `i` is created and it can be a reference to a different object.

Call-By-Reference versus Call-By-Value

- In C++, an argument to a function may be passed by 2 methods:
  - **call-by-reference** (**CBR**)
  - **call-by-value** (**CBV**)

- In the call `f(j)`, `i` is created similarly to the construction:

CBR	CBV
int& i = j;	int i = j;

Note that in CBV, `i` is NOT an alias of `j`. Thus, any change to `i` will not result in any change to `j`.

Why do Call-By-Reference?

1. **For side effects:** When the function caller wants the function to be able to change the value of passed arguments.

2. **For efficiency:** If you pass a function argument by value, the function gets a local copy of the argument. For large objects, copying is expensive; on the other hand, passing an object by reference does not require copying, only passing a memory address.

```
1. class Large_Obj
2. {
3. public:
4.         int height;
5.         // … plus lots more data members requiring many bytes
6. };
7. void print_height(const Large_Obj& LO) { cout <<LO.height(); }
8. int main() { Large_Obj dinosaur(50); print_height(dinosaur); }
```

Pointer vs. Reference

- A reference can be thought of as a special kind of pointer, but there are 3 big differences to remember!
  - A pointer can point to *nothing* (NULL), but a reference is always bound to an object.
  - A pointer can point to *different* objects at different times (through assignments). A reference is always bound to the *same* object. Assignments to a reference do NOT change which object it refers to, but rather, the value of the referenced object.
  - No explicit dereferencing operators are needed with references. In contrast, the name of a pointer refers to the pointer object, so the * or -> dereferencing operators have to be used to access the object. The name of a reference always refers to the object, so there are no special operators needed for dereferencing.

Example: Pointer vs. Reference

```
#include<iostream.h>
void func1(int* pi) { (*pi)++; }
void func2(int& ri) { ri++; }

void main()
{
    int i=1;  cout << "i = " << i << endl;

    // call using address of i
    func1(&i);  cout << "i = " << i << endl;

    // call using i
    func2(i);  cout << "i = " << i << endl;
}
```

Another example: Pointer vs. Reference

```
IQ w("Maxwell", 180);
IQ x("Newton", 200);
IQ* a = NULL;                    // Ok: 'a' bound to nothing

IQ& y = x;                       // Ok: 'y' is an alias of 'x'
IQ& z;                           // Error: uninitialized ref var!

a = new IQ("Einstein", 250);     // Ok: 'a' points to "Einstein"
a = new IQ("Galileo", 190);      // Ok: 'a' now points to "Galileo"
y = w;                           // 'y' is STILL an alias of 'x' NOT 'w';
                                 // the value of 'w' is copied to 'y' ('x')

a->smarter(10); (*a).print();
y.smarter(20); y.print();
```