

COMP322 - Introduction to C++

Lecture 03 - Pointers and references

Robert D. Vincent

School of Computer Science

20 January 2010



Pointers

A *pointer* in C++ holds the address of another object.

We can declare pointers as follows:

```
int *p, *q; // p and q are pointers to base type 'int'
```

There are two operators which are useful to know:

- ▶ Unary '*': Dereference operator

```
int y = *p; // Set 'y' equal to the int stored at 'p'
```

- ▶ Unary '&': “Address-of” operator

```
int x;  
int *p = &x; // Set 'p' equal to the address of 'x'
```

The standard library defines NULL as an illegal pointer value. It generally has the same bit pattern as zero.

Pointer arithmetic

Pointer operations are extremely powerful. Almost any object can have its address taken, and be assigned to a pointer.

In addition, we can do math on pointers in restricted ways:

- ▶ $pointer = pointer + integer$: Add an integer to a pointer, the result is a pointer.
- ▶ $pointer = pointer - integer$: Subtract an integer from a pointer, the result is a pointer.
- ▶ $integer = pointer - pointer$: Subtract two pointers to get the integral number of elements between them. Pointers *must* be of the same base type, and should point to the same vector.
- ▶ Pointer comparison using $>$, $<$, $>=$, $<=$, $!=$, $==$

Pointers and arrays

In C++, pointer and array expressions are often equivalent:

```
int vec[10];  
int *p = &vec[0];    // Points to the first element  
int *q = &vec[10];    // Points past the last element  
int *r = p+10;        // Identical to prior initialization  
int x = vec[1];        // Get value of second element  
int y = *(p+1);        // Ditto  
int n = 1;  
p = p + n;            // p == &vec[1]  
q = q - 1;            // q == &vec[9]  
x = *--r;              // r == &vec[9] and x == vec[9]  
y = *p++;              // p == &vec[2] and y == vec[1]
```

Note also that the name of an array is equivalent to a pointer to the first element of the array:

```
int vec[10];  
int *p = vec;          // Legal, p == &vec[0]
```

Pointers as function arguments

We can modify function arguments by passing a pointer rather than the value.

```
void swap1(int a, int b) { // Non-working
    int tmp = a;
    a = b;
    b = tmp;
}
```

```
void swap2(int *a, int *b) { // Swap through pointers
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

```
int main() {
    int x = 1, y = 2;
    swap1(x, y);    // x==1, y==2 after return
    swap2(&x, &y);  // x==2, y==1 after return
}
```

Pointers to structs or unions

It's often useful to pass around pointers to large objects.

```
struct Example {  
    int index;  
    int data[100];  
    char text[128];  
};  
  
struct Example exmpl;  
struct Example *ex_p = &exmpl;  
  
int n = (*ex_p).index;
```

The parentheses in the prior expression are necessary because of the relative precedence of '*' and '.'. To simplify this, C++ defines the -> operator:

```
int n = ex_p->index;
```

Pointers to structs continued

We can use struct pointers to defined linked lists, for example:

```
#include <stdlib.h> // for NULL
struct LinkedList {
    struct LinkedList *link; // Recursive use is legal here
    int value;
};

struct LinkedList *
find_element(struct LinkedList *list, int value) {
    struct LinkedList *ptr;
    for (ptr = list; ptr != NULL; ptr = ptr->link) {
        if (value == ptr->value)
            return ptr;
    }
    return NULL;
}
```

Remember the magic value NULL which is used to indicate an illegal pointer value.

Pointers to pointers

To add to our confusion, it is legal to have arrays of pointers, pointers to pointers, and so on, *ad infinitum*.

```
struct Symbol {
    struct Symbol *link; // Next in linked list
    int value;           // Value
    char name[32];       // Text name
};

struct Symbol *hashtable[100]; // Open hash table

#include <iostream>

int main(int argc, char **argv) {
    // argc is the number of arguments
    // argv is a list of char * (string) arguments
    for (int i = 0; i < argc; i++) {
        std::cout << *argv++ << std::endl; // Print arguments
    }
}
```

Pointer traps and pitfalls

Use of pointers routinely leads to a number of errors:

- ▶ Failure to initialize:

```
int *p;  
  
*p = 1;           // Error - 'p' points to nothing
```

- ▶ Returning a pointer to a local variable:

```
struct Point *makept(int x, int y) {  
    struct Point pt;  
    pt.x=x;  
    pt.y=y;  
    return (&pt);  // Undefined upon return!  
}
```

- ▶ Misunderstanding pointer comparison

```
char *s1 = "apple";  
char *s2 = "apple";  
if (s1 == s2) {  
    // This may nor may not be true!  
}
```

References

C++ supports the creation of *references* to other data objects. These references are essentially an alias for the named object.

```
int x;                // Declare x
int & y = x;          // y is a reference to x

x = 1;
y = 2;
cout << x << endl;
// this will print 2 rather than 1
```

The definition of a reference must be initialized:

```
int x;
int & y;                // Error!
```

A reference cannot be reassigned to a new object, and must have the same type as its associated object. And we cannot define a reference to a reference!

References as function parameters

The real utility of references occurs as an alternative method for function parameter passing.

```
void swap1(int a, int b) { // Non-working
    int tmp = a;
    a = b;
    b = tmp;
}
```

```
void swap3(int &a, int &b) { // Does the right thing
    int tmp = a;
    a = b;
    b = tmp;
}
```

```
int main() {
    int x = 1, y = 2;
    swap1(x, y);    // x==1, y==2 after return
    swap3(x, y);    // x==2, y==1 after return
}
```

Reference parameters and efficiency

In traditional call by value, a complete copy of a data object is made. This may be inefficient when dealing with large objects.

```
struct Example {  
    int value[100];  
    char text[1024];  
};  
  
int get_vn1(Example ex, int n) { // Slow...  
    return ex.value[n];  
}  
  
int get_vn2(const Example &ex, int n) { // Fast!!  
    return ex.value[n];  
}
```

Use of the `const` keyword prevents the function from modifying the contents of the structure.

Reference details

A reference parameter must be bound to a object of the same type, and the object must be an *lvalue*.

```
void swap1(int a, int b) { // Non-working
    int tmp = a;
    a = b;
    b = tmp;
}

void swap3(int &a, int &b) {
    int tmp = a;
    a = b;
    b = tmp;
}

int main() {
    int a=1, b=2, vec[10];
    float c=3;

    swap1(a,b+1);           // Legal but no effect
    swap3(a,b+1);           // Error, 'b+1' is not an lvalue
    swap3(vec[1], vec[2]); // OK
    swap1(a,c);             // Legal but useless
    swap3(a,c);             // Error!
}
```

But what is an lvalue?

In C++, an *lvalue* is an expression that yields a result at which another value can be stored.

Lvalues include variables, struct fields, and array or pointer dereferences.

```
int x, array[10], *intptr = &x;
struct point {
    unsigned int x, y;
} pt;
```

```
// These are all legal as the lhs is an lvalue
x = 1;
pt.x = 2;
array[1] = 3;
*intptr = 4;
*(array+x) = 5;
```

```
x+1 = 1;           // Not an lvalue
pt.x/2 = 2;        // Not an lvalue
array = 3;         // Not an lvalue
func() = 4;        // Not usually an lvalue
```

References as return values

You can also return a reference from a function. This is useful in two situations:

- ▶ **To return a large object without copying it**
- ▶ To return an lvalue

```
struct Symbol {
    double value;
    char name[32];
} symtable[100];

int symnext = 0;

Symbol & alloc_sym(const char name[], double v) {
    Symbol *psym = &symtable[symnext++];
    strcpy(psym->name, name);
    psym->value = v;
    return *psym;
}
```

References as return values

You can also return a reference from a function. This is useful in two situations:

- ▶ To return a large object without copying it
- ▶ **To return an lvalue**

```
struct Point {  
    int x,y;  
};  
  
int & ptx(struct Point &pt) {  
    return pt.x;  
}  
  
int main() {  
    Point p;  
  
    ptx(p) = 10;    // ptx(p) is an lvalue here!  
}
```