COMP322 - Introduction to C++

Winter 2011

Lecture 3 - Pointers and References

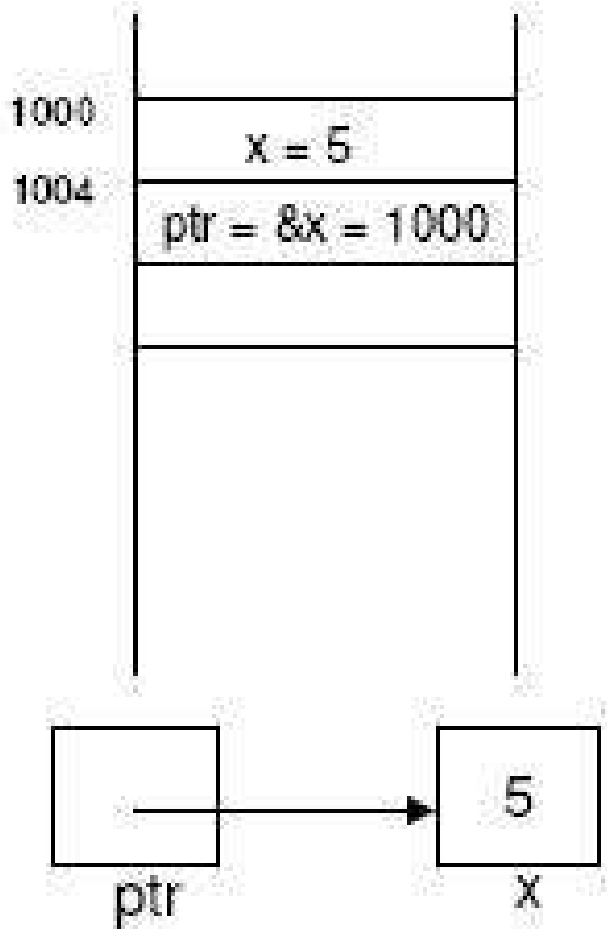
Milena Scaccia

School of Computer Science
McGill University

January 18, 2011

Pointers - Introduction

Definition: A pointer is a type of variable in C/C++ that holds the address of another variable.



Motivation

Why use pointers?

- Powerful programming construct
- Pointers allow sections of code to share information easily (instead of copying information back and forth)
- Enable complex linked data structures such as linked lists and binary trees
- Can be used to dynamically allocate memory

"With increased power comes increased responsibility"

Pointers

We can declare pointers as follows:

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

There are two main operators:

- Unary '&': "Address-of" operator

```
int x = 1;
int *p = &x; // Set 'p' equal to the address of 'x'
```

- Unary '*': Dereference operator

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

The standard library defines `NULL` as an illegal pointer value. It generally has the same bit pattern as zero. It is used to denote that a pointer does not point to any valid memory address.

Danger 1: Wild/Bad Pointer

- Bad: To declare a pointer but not give it a "pointee", i.e. not initialize it
- This is known as a wild pointer, or a bad pointer
- A bad pointer is uninitialized: its initial value is a bad value (random address), and it is the responsibility of the programmer to overwrite this bad value with the correct code
- Do not confuse a bad pointer with a `NULL` pointer!

Bad Pointer Continued

```
void WildPointer(){
  int *p; // allocate the pointer, but not initialize it
  *p = 29; // this dereference is a serious runtime error

  // Who knows what you may overwrite?

  // How to fix this?

  /* Solution:
  int x = 29;
  int *p = &x; */
}
```

- A dereference operation on a bad pointer can cause some serious problems
- A bad pointer dereference may corrupt a random area of memory, which may have the consequence of altering the operation of the program so that it goes wrong at some indefinite time later.

Comparison with Java

In higher-level languages such as Java,

- Pointers are used behind the scenes to implement complex types such as arrays and objects.
- The run-time system sets each pointer to `NULL` when it is allocated, and checks it each time it is dereferenced - this way, the program will halt immediately and not behave unpredictably if there is a problem.
- Java: Fewer bugs but slower
- C/C++: Faster but potentially more bugs

Three Basic Rules for Pointers

1. Declare a pointer and initialize it

```
int x = 1;
int *p;
p = &x;
```

2. Dereference a pointer to access/modify its pointee

```
*p = 29; // Now x has the value 29
```

3. Assignment between pointers makes them point to the same thing, aka "sharing"

```
int *q;
q = p; // q and p both point to x
```

Pointers

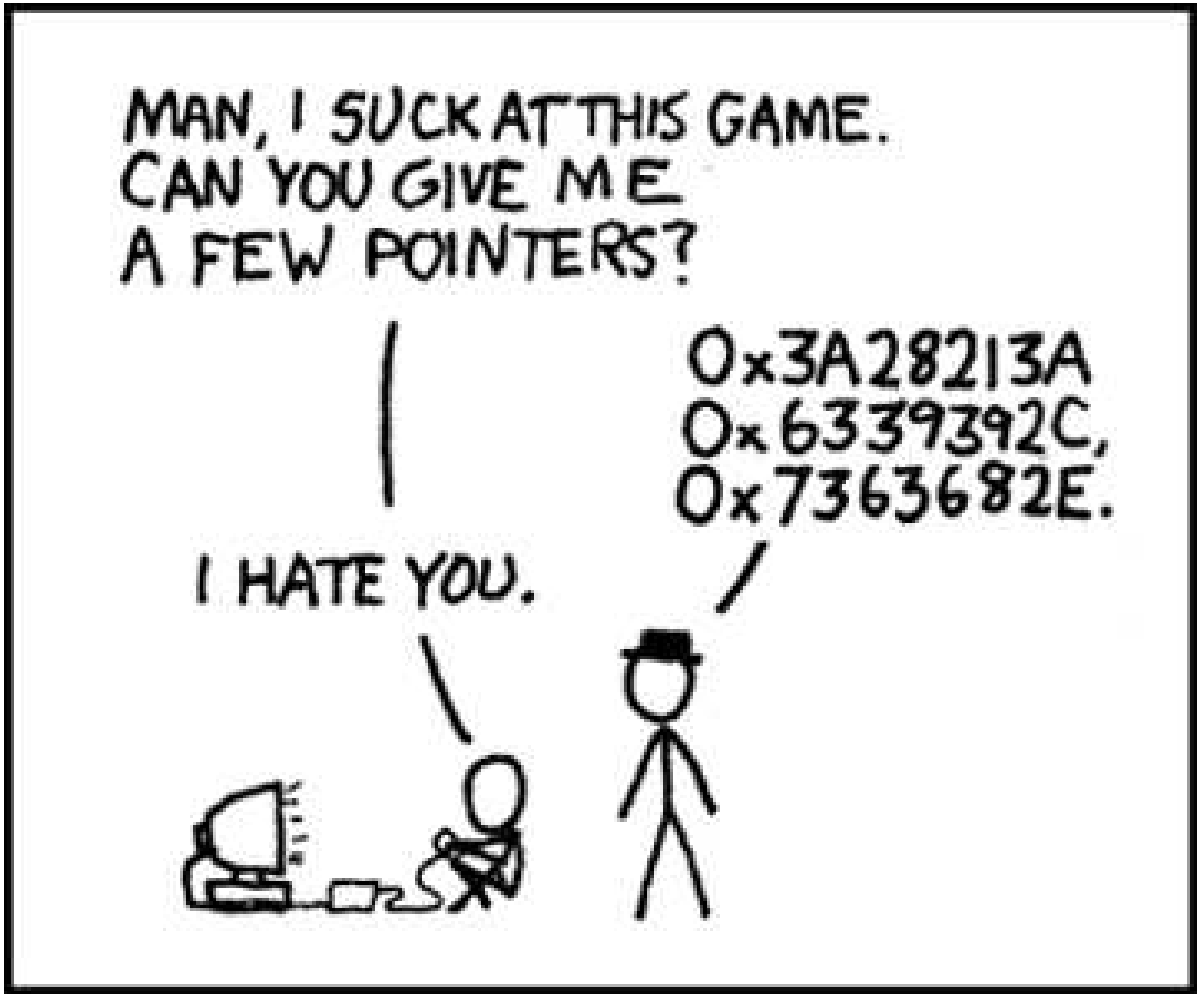


Figure: xkcd: Pointers

Pointer Examples

```
int x = 2;
int y = 4;

int *p1 = x; // Is this legal? NO
int *p2;

// Are these similar? NO
*p2 = x; // value of x assigned to value that p2 points to
p2 = &x; // address of x assigned to p2

int *p3 = &x;
*p3 += 10;
cout << x <<endl;; // What will this print? 12

int *p4 = &x;
int *p5 = &y;

// Are these similar?  NO
p4 = p5; // p4 and p5 now point to the same place
*p4 = *p5; //overwriting p4's data value with that of p5's

// Are these similar?  NO
*p2 += 1;  //This increments the value
*p2++; //This dereferences, and then modifies the address of p2
```

Operator Precedence

- *p2++ first dereferences p2, and then modifies p2's target address (same as *(p2++))
- (*p2)++ increments the value pointed to, while the address in p2 remains unchanged (same as ++*p2)
- *++p2 first modifies p2's target address and then dereferences p2

Pointer arithmetic

We have the ability to modify a pointer's target address with arithmetic operations. Adding or subtracting from a pointer moves it by a multiple of the size of the datatype it points to.

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.
- We cannot add two pointers. Not meaningful.
- 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 precendence of '*' and '.'. To simplify this, C++ defines the -> operator:

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

Pointers to structs continued

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

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

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

char c; // a character
char *pc // a pointer to a character
char **ppc; // a pointer to a pointer to a character
```

Pointers for dynamic memory allocation

Do not need to know in advance how much space is needed.
Memory can be allocated on-the-fly.

```
int *create_vector(int size) {
    int *vector = new int[size]; // Allocate 'size' integers
    // perform some initialization
    return vector;
}

void delete_vector(int *vector) {
  delete [] vector; // brackets tell delete this is an array
}
```

To be covered in detail in Lecture 4

Danger 2 - Dangling Pointer

- When the object a pointer pointed to is deleted, without modifying the value of the pointer, so that the pointer still points to the memory location of the deallocated memory
- As the system may reallocate the previously freed memory to another process, if the original program then dereferences the (now) dangling pointer, unpredictable behavior may result, as the memory may now contain completely different data.
- Can cause segmentation faults or silent corruption of unrelated data

Notions unique to C++

The C++ standard library also supports:

- Smart Pointers: the `auto_ptr` class from the C++ standard library simulates simple pointers but also performs other tasks e.g. automatic initialization, automatic cleanup, takes care of garbage collection

- References: quite different from a pointer, but another form of reference

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! We also cannot create an array of references.

References as function parameters

An official feature to the C++ programming language. The real utility of references occurs as an alternative method for function parameter passing. Looks nicer than our previous swap using pointers.

```
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; // No *'s required, the compiler takes care of it
  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, notice no &'s required
}
```

References as return values

Useful for returning a large object without copying it. E.g.

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

struct Symbol 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;
}
```