NWEN 241
Pointers

Winston Seah

School of Engineering and Computer Science
Victoria University of Wellington

Background

- All information accessible to a running computer program must be stored somewhere in the computer's memory.
- C provides the ability to access specific memory locations, using “pointers”.
- Memory locations are identified by their ***address***.
- How long are the addresses?

Intel Core i7 has 64-bit addresses:

```
int *ip; // defines a variable of type integer pointer

printf(“%lu”, sizeof(ip));
```

What is the output of the simple printf statement?

Pointer basics

- Address of the location containing the data
  - all pointers are typed based on the type of entity that they point to;
  - to declare a pointer, use * preceding the variable name as in:

int *x;

- To set a pointer to a variable's address use **&** before the variable as in:

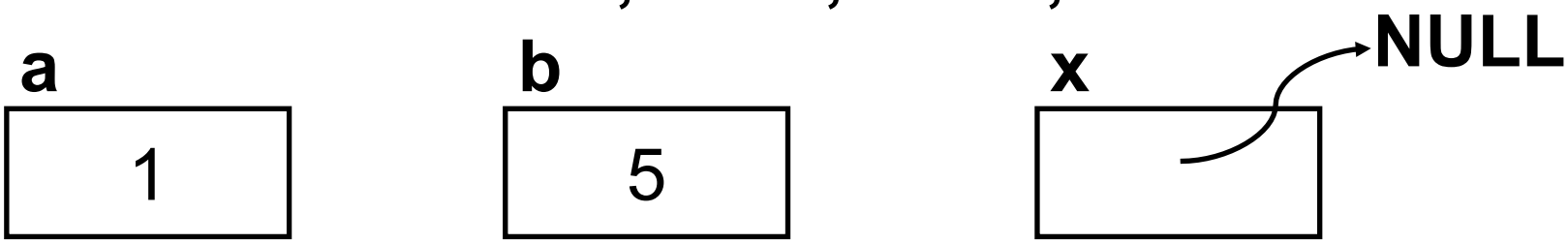
x = &a;

  - **&** means "return the memory address of"
  - **x** will now point to **a**, i.e., **x** stores **a**'s address

Pointer basics

Declaration:

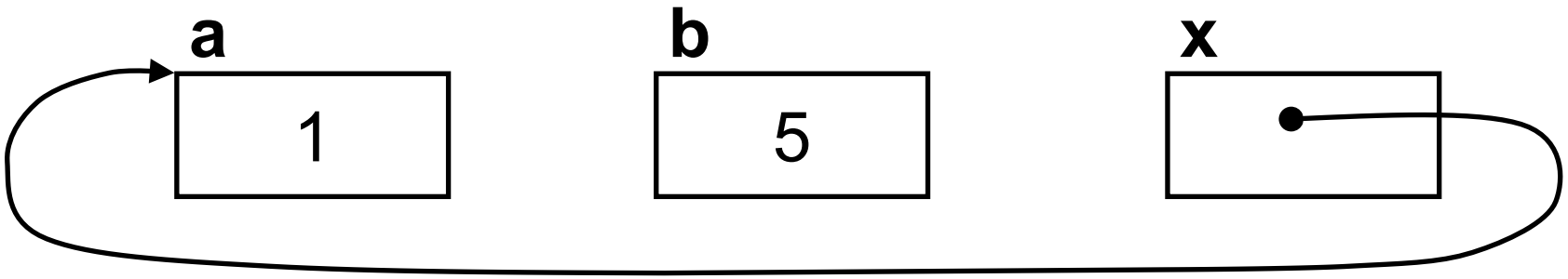
int a = 1, b = 5; int *x;



NULL – pointer literal/constant to non-existent addr.

Assignment:

x = &a;



Pointer basics

- If you access **x**, you merely get the address
- To get the value in the variable/location that **x** points to, use * as in ***x**

```
*x = *x + 1; // adds 1 to variable a whose
             // address is contained in x
```

- * is known as the ***indirection*** (or ***dereferencing***) operator as it requires a second access, that is, this is a form of *indirect addressing.* E.g.

```
b = *x;
```

Pointer basics

- Recall:

```
int a = 1, b = 5; int *x;

x = &a;          // What is the value of x ?

*x = *x + 1;     // a =    ;  b =    ;

b = *x;
```

- What is the value of **b** ?

Usage of pointers

i. Provide an alternative means of accessing information stored in arrays, especially when working with strings; there exists an intimate link between arrays and pointers in C.
ii. To handle variable parameters passed to functions.
iii. To create dynamic data structures, that are built up from blocks of memory allocated from the heap at run time. This is only visible through the use of pointers.

Pointers & Arrays

Recall:

- Arrays in C are pointed to, i.e. the variable that you declare for the array is actually a pointer to the first array element
- You can interact with the array elements either through pointers or by using [ ]

```
int z[], *ip; ip = &z[0];
```

`z[0]`, `*ip` or `*z` can all be used to access the first element of the array `z[]`

Pointers & Arrays

- What about accessing **`z[1]`** using pointers ?

- Note that **`ip=ip+1`** (or **`ip++`**) moves the pointer 4 bytes, instead of 1 to point to the next array element; amount added depends on size of array element
  - ? for an array of **`double`**s
  - ? for an array of **`char`**s
  - ? for an array of **`int`**s

Pointers & Arrays

Iterating through elements of an array:

```
int j;
for(j = 0; j < n; j++)
     a[j]++;
```

vs

```
int *pj;
for(pj = a; pj < a + n; pj++)
       (*pj)++;
```

- **`pj`** is a pointer to an **`int`**
- Start with **`pj`** pointing at **`a`**, i.e., **`pj`** points to **`a[0]`**
- The loop iterates while **`pj < a + n`**
  - **`pj`** is a pointer, so it is an address
  - **`a`** is a pointer to the beginning of an array of **`n`** elements; so **`a+n`** is the size of the array
  - **`pj++`** increments the pointer to point at the next element in the array
  - The instruction **`(*pj)++`** says “take what **`pj`** points to and increment it”

Pointers Arithmetic

Iterating through elements of an array:

```
int j;
for(j = 0; j < n; j++)
     a[j]++;
```

VS

```
int *pj;
for(pj = a; pj < a + n; pj++)
       (*pj)++;
```

- NOTE:
  (*pj)++; // increments what **pj** points to
  *(pj++); // increments the pointer to point at the
  // next array element
- What do each of these do?

  *pj++;

  ++*pj;

Pointers Arithmetic

- Subtraction on pointers

```
int a[10] = {…};
int *ip;
for(ip = &a[9]; ip >= a; ip--)
      …
```

- Example:
  Pass to a function the address of the 3rd element of an array **`&a[2]`** and use pointer subtraction to get to **`a[0]`** and **`a[1]`**.

```
int addem(int *ip)
{
   int temp;
   temp = *ip + *(ip - 1) + *(ip - 2);
   return temp;
}
```

```
int a[3] = {…};
printf("%d", addem(&a[2]));
```

Strings ❤ Pointers

```
int strlen (char *s)
{
    int n;
    for(n = 0; *s != '\0'; s++)
        n++;
    return n;
}
```

```
int strcmp (char *s, char *t)
 {
    int i;
    for(i=0;s[i] = = t[i];i++)
        if(s[i] = = '\0')
            return 0;
    return s[i] – t[i];
 }
```

```
void strcpy (char *s, char *t)
{
    int i = 0;
    while((s[i] = t[i]) != '\0')
        i++;
}
```

```
int strcmp (char *s, char *t)
 {
    for ( ; *s = = *t; s++, t++)
        if (*s = = '\0') return 0;
    return *s - *t;
 }
```

```
void strcpy (char *s, char *t)
{
    while((*s = *t) != '\0')
    {
        s++;   t++;
    }
}
```

Notice in the second strcmp and second and third strcpy, the use of pointers to iterate through the strings

The conciseness of the last strcmp and strcpy make them hard to understand

```
void strcpy (char *s, char *t)
{
    while((*s++ = *t++) != '\0');
}
```

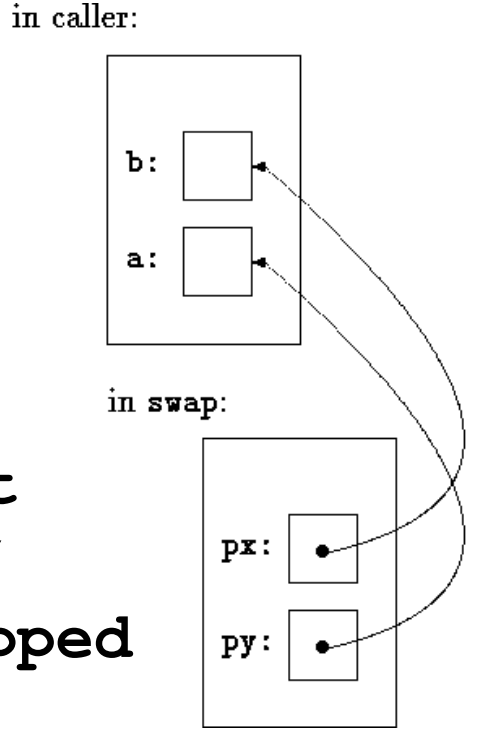
Reference / variable parameters

To make changes to a variable that exist after a function ends, we pass the address of (a pointer to) the variable to the function (a reference parameter)

Then we use indirection operator inside the function to change the value the parameter points to:

```
int a, b; swap (&a, &b);

void swap(int *px, int *py)
{ int temp;
  temp = *px;
  *px = *py;    // values stored at
  *py = temp;   // addresses of 'a'
}               // and 'b' are swapped
```



Returning pointers from functions

A function can also return a pointer value:

```
float *findMax(float A[], int N) {
  int I;
  float *theMax = &(A[0]);

  for (I = 1; I < N; I++)
    if (A[I] > *theMax) theMax = &(A[I]);

  return theMax;
}

void main() {
  float A[5] = {0.0, 3.0, 1.5, 2.0, 4.1};
  float *maxA;

  maxA = findMax(A,5);
  *maxA = *maxA + 1.0;
  printf("%.1f %.1f\n",*maxA,A[4]);
}
```

Returning pointers from functions

Caution!!!

- In functions, do not do **`return p`**, where **`p`** is a pointer to a **local** variable.

Recall:

- local variables are deallocated when the function ends
  - so whatever **`p`** is pointing to will no longer be available
  - but if you return the pointer, then you still are pointing at that memory location even though you no longer know what is there

Question:

- Why is it allowed in the previous example?

Pointer to pointers

A pointer can also be made to point to a pointer variable (but the pointer must be of a type that allows it to point to a pointer)

Example:

```
int V = 101;
int *P = &V;  /* P points to int V */
int **Q = &P; /* Q points to int pointer P */

printf("%d %d %d\n", V, *P, **Q);
                  /* prints 101 3 times */
```

Pointer Types

Pointers are generally of the same size (enough bytes to represent all possible memory addresses), but it is inappropriate to assign an address of one type of variable to a different type of pointer

Example:

```
int V = 101;
float *P = &V; /* Generally results in a
  Warning */
```

Warning rather than error because C will allow you to do this (it is appropriate in certain situations)

Casting Pointers

When assigning a memory address of a variable of one type to a pointer that points to another type, it is best to use the cast operator to indicate the cast is intentional (this will remove the warning).

Example:

```
int V = 101;
float *P = (float *) &V;
/* Casts int address to float * */
```

Removes warning, but is still unsafe to do this !!!

General (void) pointer

A **`void *`** is considered to be a general pointer

No cast is needed to assign an address to a **`void *`** or from a **`void *`** to another pointer type

Example:

```
int V = 101;
void *G = &V; /* No warning */
float *P = G; /* No warning, still unsafe */
```

Certain library functions return **`void *`** results

A useful quote to remember

"*Pointers have been lumped with the **goto** statement as a marvelous way to create impossible-to-understand programs. This is certainly true when they are used carelessly, and it is easy to create pointers that point somewhere unexpected. With discipline, however, pointers can also be used to achieve clarity and simplicity*." – Kernighan and Ritchie, 1988.