CSC180: Lecture 34

Wael Aboulsaadat

wael@cs.toronto.edu

http://portal.utoronto.ca/

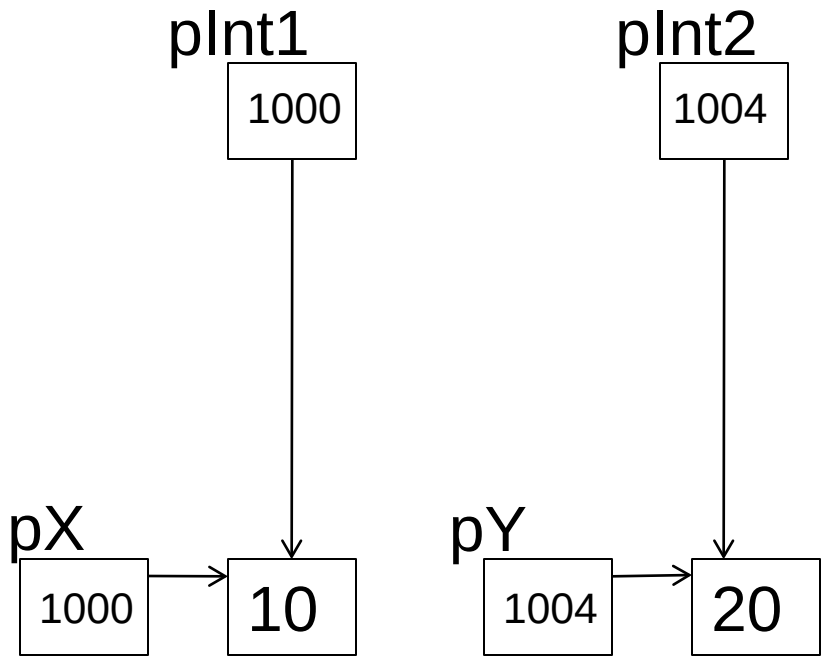
Acknowledgement: These slides are partially based on the slides supplied with Prof. Savitch book: Problem Solving with C

Advanced Pointers

1. Pointers & Pass-by-value

- Example: swapping 2 pointers

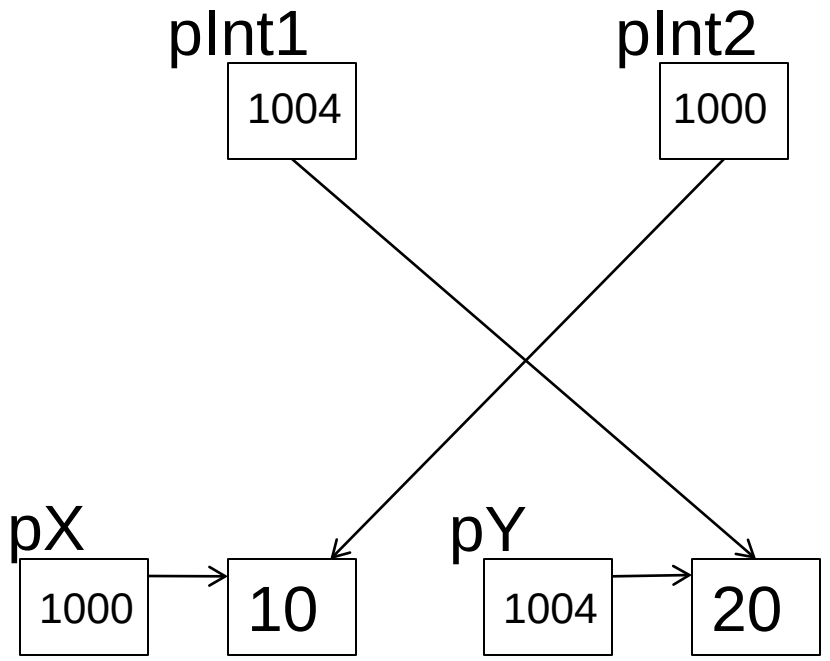
```
void swap_pointers_1( int *pInt1, int *pInt2 )
{
    int  *temp;
    temp   = pInt1;
    pInt1  = pInt2;
    pInt2  = temp;
}
int main()
{
    int      *pX,*pY;
    pX = (int*) malloc( sizeof( int ) );
    pY = (int*) malloc( sizeof( int ) );
    *pX = 10;
    *pY = 20;
    swap_pointers_1( pX, pY );
}
```



1. Pointers & Pass-by-value

- Example: swapping 2 pointers

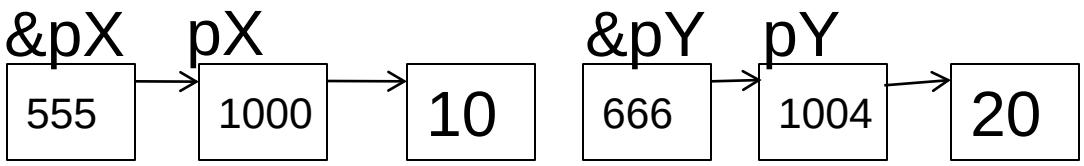
```
void swap_pointers_1( int *pInt1, int *pInt2 )
{
    int  *temp;
    temp   = pInt1;
    pInt1  = pInt2;
    pInt2  = temp;
}
int main()
{
    int      *pX,*pY;
    pX = (int*) malloc( sizeof( int ) );
    pY = (int*) malloc( sizeof( int ) );
    *pX = 10;
    *pY = 20;
    swap_pointers_1( pX, pY );
}
```



1. Pointers & Pass-by-value

- Example: swapping 2 pointers

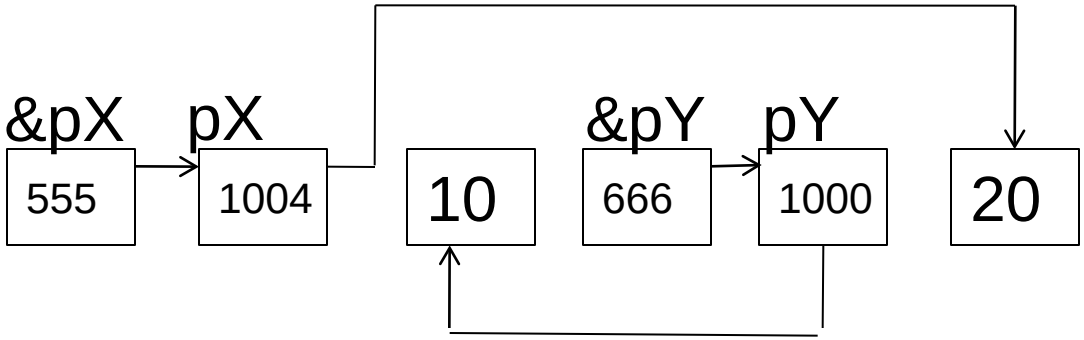
```
void swap_pointers_2( int  **pInt1, int  **pInt2 )
{
      int  *temp;
      temp   = *pInt1;
      *pInt1  = *pInt2;
      *pInt2  = temp;
}
int main()
{
      int      *pX,*pY;
      pX = (int*) malloc( sizeof( int ) );
      pY = (int*) malloc( sizeof( int ) );
      *pX = 10;
      *pY = 20;
      swap_pointers_2( &pX, &pY );
}
```



1. Pointers & Pass-by-value

- Example: swapping 2 pointers

```
void swap_pointers_2( int  **pInt1, int  **pInt2 )
{
      int  *temp;
      temp   = *pInt1;
      *pInt1  = *pInt2;
      *pInt2  = temp;
}
int main()
{
      int      *pX,*pY;
      pX = (int*) malloc( sizeof( int ) );
      pY = (int*) malloc( sizeof( int ) );
      *pX = 10;
      *pY = 20;
      swap_pointers_2( &pX, &pY );
}
```



2. Pointers to functions

- Declaration:

```
returnType (*varName)(parameterTypes);
```

- Examples:

```
int (*f)(int, float);

int* (*g)(int, float);

int* (*g[])(int, float);
```

pointer to a function that takes an integer argument and a float argument and returns an integer

pointer to a function that takes an integer argument and a float argument and returns a *pointer* to an integer

An *array* of pointers to functions – Each function takes an integer argument and a float argument and returns a pointer to an integer

Pointers to functions: WHY?

- They allow for a certain amount of **polymorphism**:
  - “poly” (many) + “morph” (shape)
  - A polymorphic language can handle a range of different data types (“shapes”?) with a single statement
- This is common in OO languages like C++, Java:

```
Animal myPet;
…
myPet.makeSound();
```

This method call will result in different sounds, depending on whether `myPet` holds a `Cow` object, an `Elephant` object, etc.

Example: searching a singly-linked list

```
typedef struct IntNode {
  int value;        struct IntNode *next;
} INTNODE;
```

OK, but it only works for nodes containing integer data. If you want a list of strings, you'll need to define a new type and new function.

```
INTNODE *search_list(INTNODE *node, int const key) {
  while (!node) {
       if (node->value == key)
            break;
       node = node->next;
  }
  return node;
}
```

A more abstract notion of “node”

```
typedef struct Node {
   void *value;     struct Node *next;
} NODE;
```

`void*` is compatible with any pointer type.
So, this member can hold (a pointer to) any value!

```
void construct_node(NODE *node, void *value, NODE *next) {
   node->value = value;
    node->next = next;
}

NODE *new_node(void *value, NODE *next) {
   NODE *node = (NODE *)malloc(sizeof(NODE));
   construct_node(node, value, next);
   return node;
}
```

A more abstract notion of “search list”

- What is it that makes the old `search_list` only work for integers?
  - The `key` parameter is of type `int`
  - The `==` operator is used to compare `int` values –
    but `==` will not work for many types (e.g. structs, strings)

- A solution: pass in an additional argument –
  a comparison function!
  - Programmer must supply a comparison function that’s appropriate for the data type being stored in the nodes
  - This function argument is called a **callback function**:
    - Caller passes in a pointer to a function
    - Callee then “calls back” to the caller-supplied function

Abstract “search list” with callback function

```
NODE *search_list(NODE *node, void const *key,
      int (*compare)(void const *, void const *)) {

  while (node) {
      if (!compare(node->value, key)) break;
      node = node->next;
  }
  return node;

}
```

Assumption: `compare` returns zero if its parameter values are equal; nonzero otherwise

Using callback functions

- If our nodes hold strings, we have a compare function already defined: `strcmp` or `strncmpy`

```
#include <string.h>
…
match = search_list(root, "key", &strcmp);
```

`&` is optional here –
compiler will implicitly take the address

Note: you may get a warning, since `strcmp` is not strictly of the right type:
its parameters are of type `char *` rather than `void *`

Using callback functions

- If our nodes hold other kinds of data, we may need to “roll our own” compare function

```
int compare_ints(void const *a, void const *b) {
   const int ia = *(int *)a, ib = *(int *)b;
   return ia != ib;
}
…
match = search_list(root, key, &compare_ints);
```

3. Jump tables

- Jump table:

```
double add(double, double);
double sub(double, double);
double mul(double, double);
double div(double, double);

double (*oper_func[])(double, double) = {
   add, sub, mul, div
};

…..

result = oper_func[0](op1, op2);
```

Array of pointers to functions. Each function takes two doubles and returns a double