fnpointers.c (Page 1 of 2)

```
 1: /*--------------------------------------------------------------------*/
 2: /* fnpointers.c                                                       */
 3: /* Author: Bob Dondero                                                */
 4: /*--------------------------------------------------------------------*/
 5:
 6: #include <stdio.h>
 7:
 8: /*--------------------------------------------------------------------*/
 9:
10: /* Return i squared. */
11:
12: static int sqr(int i)
13: {
14:    return i * i;
15: }
16:
17: /*--------------------------------------------------------------------*/
18:
19: /* Demonstrate function pointers.  Return 0. */
20:
21: int main(void)
22: {
23:    int i;
24:
25:    int (*pf)(int);  /* pf is a function pointer.
26:                        pf is a variable which, when dereferenced,
27:                        yields a function that takes an int and
28:                        returns an int. */
29:
30:    /* An ordinary function call: */
31:
32:    i = sqr(5);
33:    printf("%d\n", i);
34:
35:    /* A function call through a function pointer: */
36:
37:    pf = sqr;
38:    i = (*pf)(5);
39:    printf("%d\n", i);
40:
41:    /* Unusual ways of using function pointers: */
42:
43:    pf = &sqr;
44:    i = (*pf)(5);
45:    printf("%d\n", i);
46:
47:    pf = *sqr;
48:    i = (*pf)(5);
49:    printf("%d\n", i);
50:
51:    pf = sqr;
52:    i = pf(5);
53:    printf("%d\n", i);
54:
55:    pf = &sqr;
56:    i = pf(5);
57:    printf("%d\n", i);
58:
59:    pf = *sqr;
60:    i = pf(5);
61:    printf("%d\n", i);
62:
63:    return 0;
```

fnpointers.c (Page 2 of 2)

```
64: }
```

Precept 13
Week 8, Mon / Tue

sort8.c (Page 1 of 2)

```
/*--------------------------------------------------------------------*/
/* sort8.c                                                            */
/* Author: Bob Dondero                                                */
/*--------------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

/*--------------------------------------------------------------------*/

/* Grow the memory chunk pointed to by pdChunk so it can store
   uArrayLength elements of type double.  Return the address of the new
   memory chunk.  */

static double *grow(double *pdChunk, size_t uArrayLength)
{
   size_t uNewSize;
   double *pdNewChunk;

   assert(pdChunk != NULL);

   uNewSize = uArrayLength * sizeof(double);
   pdNewChunk = (double*)realloc(pdChunk, uNewSize);
   if (pdNewChunk == NULL)
   {
      fprintf(stderr, "Cannot allocate memory\n");
      free(pdChunk);
      exit(EXIT_FAILURE);
   }

   return pdNewChunk;
}

/*--------------------------------------------------------------------*/

/* Return -1, 0, or 1 depending upon whether double *pvItem1 is less
   than, equal to, or greater than double *pvItem2, respectively. */

static int compareDouble(const void *pvItem1, const void *pvItem2)
{
   assert(pvItem1 != NULL);
   assert(pvItem2 != NULL);

   if (*(double*)pvItem1 < *(double*)pvItem2) return -1;
   if (*(double*)pvItem1 > *(double*)pvItem2) return 1;
   return 0;
}

/*--------------------------------------------------------------------*/

/* Read numbers from stdin, and write them in ascending order to
   stdout.  Return 0 if successful, or EXIT_FAILURE if stdin contains
   a non-number. */

int main(void)
{
   const size_t INITIAL_ARRAY_LENGTH = 2;
   const size_t ARRAY_GROWTH_FACTOR = 2;

   double *pdNumbers;
   size_t uCount;
   size_t uArrayLength;
```

sort8.c (Page 2 of 2)

```
   double dNumber;
   size_t u;
   int iScanfRet;

   /* Create a small array. */
   uArrayLength = INITIAL_ARRAY_LENGTH;
   pdNumbers = (double*)calloc(uArrayLength, sizeof(double));
   if (pdNumbers == NULL)
   {
      fprintf(stderr, "Cannot allocate memory\n");
      exit(EXIT_FAILURE);
   }

   /* Read the numbers into the array, expanding the size of the
      array as necessary. */
   for (uCount = 0; ; uCount++)
   {
      iScanfRet = scanf("%lf", &dNumber);
      if (iScanfRet == 0)
      {
         fprintf(stderr, "Non-number in stdin\n");
         free(pdNumbers);
         exit(EXIT_FAILURE);
      }
      if (iScanfRet == EOF)
         break;
      if (uCount == uArrayLength)
      {
         uArrayLength *= ARRAY_GROWTH_FACTOR;
         pdNumbers = grow(pdNumbers, uArrayLength);

      }
      pdNumbers[uCount] = dNumber;
   }

   /* Sort the array. */
   qsort(pdNumbers, uCount, sizeof(double), compareDouble);

   /* Write the numbers from the array. */
   for (u = 0; u < uCount; u++)
      printf("%g\n", pdNumbers[u]);

   /* Free the array. */
   free(pdNumbers);

   return 0;
}
```

sort9.c (Page 1 of 2)

```
/*--------------------------------------------------------------------------*/
/* sort9.c                                                                  */
/* Author: Bob Dondero                                                      */
/*--------------------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

/*--------------------------------------------------------------------------*/

/* Grow the memory chunk pointed to by pdChunk so it can store
   uArrayLength elements of type double.  Return the address of the new
   memory chunk.  */

static double *grow(double *pdChunk, size_t uArrayLength)
{
   size_t uNewSize;
   double *pdNewChunk;

   assert(pdChunk != NULL);

   uNewSize = uArrayLength * sizeof(double);
   pdNewChunk = (double*)realloc(pdChunk, uNewSize);
   if (pdNewChunk == NULL)
   {
      fprintf(stderr, "Cannot allocate memory\n");
      free(pdChunk);
      exit(EXIT_FAILURE);
   }

   return pdNewChunk;
}

/*--------------------------------------------------------------------------*/

/* Return -1, 0, or 1 depending upon whether double *pdItem1 is less
   than, equal to, or greater than double *pdItem2, respectively. */

static int compareDouble(const double *pdItem1, const double *pdItem2)
{
   assert(pdItem1 != NULL);
   assert(pdItem2 != NULL);

   if (*pdItem1 < *pdItem2) return -1;
   if (*pdItem1 > *pdItem2) return 1;
   return 0;
}

/*--------------------------------------------------------------------------*/

/* Read numbers from stdin, and write them in ascending order to
   stdout.  Return 0 if successful, or EXIT_FAILURE if stdin contains
   a non-number. */

int main(void)
{
   const size_t INITIAL_ARRAY_LENGTH = 2;
   const size_t ARRAY_GROWTH_FACTOR = 2;

   double *pdNumbers;
   size_t uCount;
   size_t uArrayLength;
```

sort9.c (Page 2 of 2)

```
   double dNumber;
   size_t u;
   int iScanfRet;

   /* Create a small array. */
   uArrayLength = INITIAL_ARRAY_LENGTH;
   pdNumbers = (double*)calloc(uArrayLength, sizeof(double));
   if (pdNumbers == NULL)
   {
      fprintf(stderr, "Cannot allocate memory\n");
      exit(EXIT_FAILURE);
   }

   /* Read the numbers into the array, expanding the size of the
      array as necessary. */
   for (uCount = 0; ; uCount++)
   {
      iScanfRet = scanf("%lf", &dNumber);
      if (iScanfRet == 0)
      {
         fprintf(stderr, "Non-number in stdin\n");
         free(pdNumbers);
         exit(EXIT_FAILURE);
      }
      if (iScanfRet == EOF)
         break;
      if (uCount == uArrayLength)
      {
         uArrayLength *= ARRAY_GROWTH_FACTOR;
         pdNumbers = grow(pdNumbers, uArrayLength);
      }
      pdNumbers[uCount] = dNumber;
   }

   /* Sort the array. */
   qsort(pdNumbers, uCount, sizeof(double),
      (int(*)(const void*, const void*))compareDouble);

   /* Write the numbers from the array. */
   for (u = 0; u < uCount; u++)
      printf("%g\n", pdNumbers[u]);

   /* Free the array. */
   free(pdNumbers);

   return 0;
}
```

```
/*--------------------------------------------------------------------*/
/* stack.h (Version 6)                                                */
/* Author: Bob Dondero                                                */
/*--------------------------------------------------------------------*/

#ifndef STACK_INCLUDED
#define STACK_INCLUDED

/* A Stack_T is a last-in-first-out collection of items. */

typedef struct Stack *Stack_T;

/*--------------------------------------------------------------------*/

/* Return a new Stack_T object, or NULL if insufficient memory is
   available. */

Stack_T Stack_new(void);

/*--------------------------------------------------------------------*/

/* Free oStack. */

void Stack_free(Stack_T oStack);

/*--------------------------------------------------------------------*/

/* Push pvItem onto oStack.  Return 1 (TRUE) if successful, or 0
   (FALSE) if insufficient memory is available. */

int Stack_push(Stack_T oStack, const void *pvItem);

/*--------------------------------------------------------------------*/

/* Pop and return the top item of oStack. */

void *Stack_pop(Stack_T oStack);

/*--------------------------------------------------------------------*/

/* Return 1 (TRUE) if oStack is empty, or 0 (FALSE) otherwise. */

int Stack_isEmpty(Stack_T oStack);

/*--------------------------------------------------------------------*/

/* Apply function *pfApply to each element of oStack, passing
   pvExtra as an extra argument.  That is, for each element pvItem
   of oStack, call (*pfApply)(pvItem, pvExtra). */

void Stack_map(Stack_T oStack,
               void (*pfApply)(void *pvItem, void *pvExtra),
               const void *pvExtra);

#endif
```

stack6/stack.c (Page 1 of 2)

```
  1: /*--------------------------------------------------------------------*/
  2: /* stack.c (Version 6)                                                */
  3: /* Author: Bob Dondero                                                */
  4: /*--------------------------------------------------------------------*/
  5:
  6: #include <assert.h>
  7: #include <stdlib.h>
  8: #include "stack.h"
  9:
 10: /*--------------------------------------------------------------------*/
 11:
 12: /* Each item is stored in a StackNode.  StackNodes are linked to
 13:    form a list.  */
 14:
 15: struct StackNode
 16: {
 17:    /* The item. */
 18:    const void *pvItem;
 19:
 20:    /* The address of the next StackNode. */
 21:    struct StackNode *psNextNode;
 22: };
 23:
 24: /*--------------------------------------------------------------------*/
 25:
 26: /* A Stack is a "dummy" node that points to the first StackNode. */
 27:
 28: struct Stack
 29: {
 30:    /* The address of the first StackNode. */
 31:    struct StackNode *psFirstNode;
 32: };
 33:
 34: /*--------------------------------------------------------------------*/
 35:
 36: Stack_T Stack_new(void)
 37: {
 38:    Stack_T oStack;
 39:
 40:    oStack = (Stack_T)malloc(sizeof(struct Stack));
 41:    if (oStack == NULL)
 42:       return NULL;
 43:
 44:    oStack->psFirstNode = NULL;
 45:    return oStack;
 46: }
 47:
 48: /*--------------------------------------------------------------------*/
 49:
 50: void Stack_free(Stack_T oStack)
 51: {
 52:    struct StackNode *psCurrentNode;
 53:    struct StackNode *psNextNode;
 54:
 55:    assert(oStack != NULL);
 56:
 57:    for (psCurrentNode = oStack->psFirstNode;
 58:         psCurrentNode != NULL;
 59:         psCurrentNode = psNextNode)
 60:    {
 61:       psNextNode = psCurrentNode->psNextNode;
 62:       free(psCurrentNode);
 63:    }
 64:
 65:    free(oStack);
```

stack6/stack.c (Page 2 of 2)

```
 66: }
 67:
 68: /*--------------------------------------------------------------------*/
 69:
 70: int Stack_push(Stack_T oStack, const void *pvItem)
 71: {
 72:    struct StackNode *psNewNode;
 73:
 74:    assert(oStack != NULL);
 75:
 76:    psNewNode = (struct StackNode*)malloc(sizeof(struct StackNode));
 77:    if (psNewNode == NULL)
 78:       return 0;
 79:
 80:    psNewNode->pvItem = pvItem;
 81:    psNewNode->psNextNode = oStack->psFirstNode;
 82:    oStack->psFirstNode = psNewNode;
 83:    return 1;
 84: }
 85:
 86: /*--------------------------------------------------------------------*/
 87:
 88: void *Stack_pop(Stack_T oStack)
 89: {
 90:    const void *pvItem;
 91:    struct StackNode *psNextNode;
 92:
 93:    assert(oStack != NULL);
 94:    assert(oStack->psFirstNode != NULL);
 95:
 96:    pvItem = oStack->psFirstNode->pvItem;
 97:    psNextNode = oStack->psFirstNode->psNextNode;
 98:    free(oStack->psFirstNode);
 99:    oStack->psFirstNode = psNextNode;
100:    return (void*)pvItem;
101: }
102:
103: /*--------------------------------------------------------------------*/
104:
105: int Stack_isEmpty(Stack_T oStack)
106: {
107:    assert(oStack != NULL);
108:
109:    return oStack->psFirstNode == NULL;
110: }
111:
112: /*--------------------------------------------------------------------*/
113:
114: void Stack_map(Stack_T oStack,
115:                void (*pfApply)(void *pvItem, void *pvExtra),
116:                const void *pvExtra)
117: {
118:    struct StackNode *psCurrentNode;
119:
120:    assert(oStack != NULL);
121:    assert(pfApply != NULL);
122:
123:    for (psCurrentNode = oStack->psFirstNode;
124:         psCurrentNode != NULL;
125:         psCurrentNode = psCurrentNode->psNextNode)
126:       (*pfApply)((void*)psCurrentNode->pvItem, (void*)pvExtra);
127: }
```

stack6/teststack.c (Page 1 of 2)

```
  1: /*--------------------------------------------------------------------*/
  2: /* teststack.c (Version 6)                                            */
  3: /* Author: Bob Dondero                                                */
  4: /*--------------------------------------------------------------------*/
  5:
  6: #include <stdio.h>
  7: #include <stdlib.h>
  8: #include <assert.h>
  9: #include "stack.h"
 10:
 11: /*--------------------------------------------------------------------*/
 12:
 13: /* Write an error message to stderr indicating that not enough memory
 14:    is available.  Then exit with status EXIT_FAILURE. */
 15:
 16: static void handleMemoryError(void)
 17: {
 18:    fprintf(stderr, "Insufficient memory\n");
 19:    exit(EXIT_FAILURE);
 20: }
 21:
 22:
 23: /*--------------------------------------------------------------------*/
 24:
 25: /* Print string pvItem using format string pvExtra. */
 26:
 27: static void printString(void *pvItem, void *pvExtra)
 28: {
 29:    assert(pvItem != NULL);
 30:    assert(pvExtra != NULL);
 31:    printf((char*)pvExtra, (char*)pvItem);
 32: }
 33:
 34: /*--------------------------------------------------------------------*/
 35:
 36: /* Accumulate double *pvItem into *pvExtra. */
 37:
 38: static void sumDouble(void *pvItem, void *pvExtra)
 39: {
 40:    double *pdItem;
 41:    double *pdSum;
 42:    assert(pvItem != NULL);
 43:    assert(pvExtra != NULL);
 44:    pdItem = (double*)pvItem;
 45:    pdSum = (double*)pvExtra;
 46:    *pdSum += *pdItem;
 47: }
 48:
 49: /*--------------------------------------------------------------------*/
 50:
 51: /* Free pvItem.  pvExtra is unused. */
 52:
 53: static void freeDouble(void *pvItem, void *pvExtra)
 54: {
 55:    assert(pvItem != NULL);
 56:    free(pvItem);
 57: }
 58:
 59: /*--------------------------------------------------------------------*/
 60:
 61: /* Test the Stack ADT.  Return 0, or EXIT_FAILURE if not enough memory
 62:    is available. */
 63:
```

stack6/teststack.c (Page 2 of 2)

```
 64: int main(void)
 65: {
 66:    Stack_T oStack1;
 67:    Stack_T oStack2;
 68:    int iSuccessful;
 69:    double *pd;
 70:    double dSum = 0.0;
 71:
 72:    /* Create and use a Stack of strings. */
 73:
 74:    oStack1 = Stack_new();
 75:    if (oStack1 == NULL) handleMemoryError();
 76:
 77:    iSuccessful = Stack_push(oStack1, "Ruth");
 78:    if (! iSuccessful) handleMemoryError();
 79:
 80:    iSuccessful = Stack_push(oStack1, "Gehrig");
 81:    if (! iSuccessful) handleMemoryError();
 82:
 83:    iSuccessful = Stack_push(oStack1, "Mantle");
 84:    if (! iSuccessful) handleMemoryError();
 85:
 86:    Stack_map(oStack1, printString, "%s\n");
 87:
 88:    Stack_free(oStack1);
 89:    oStack1 = NULL; /* Unnecessary but safe. */
 90:
 91:    /* Create and use a Stack of doubles. */
 92:
 93:    oStack2 = Stack_new();
 94:    if (oStack2 == NULL) handleMemoryError();
 95:
 96:    pd = (double*)malloc(sizeof(double));
 97:    if (pd == NULL) handleMemoryError();
 98:    *pd = 1.1;
 99:    iSuccessful = Stack_push(oStack2, pd);
100:    if (! iSuccessful) handleMemoryError();
101:
102:    pd = (double*)malloc(sizeof(double));
103:    if (pd == NULL) handleMemoryError();
104:    *pd = 2.2;
105:    iSuccessful = Stack_push(oStack2, pd);
106:    if (! iSuccessful) handleMemoryError();
107:
108:    pd = (double*)malloc(sizeof(double));
109:    if (pd == NULL) handleMemoryError();
110:    *pd = 3.3;
111:    iSuccessful = Stack_push(oStack2, pd);
112:    if (! iSuccessful) handleMemoryError();
113:
114:    pd = NULL; /* Unnecessary. */
115:
116:    Stack_map(oStack2, sumDouble, &dSum);
117:
118:    printf("The sum is %g.\n", dSum);
119:
120:    Stack_map(oStack2, freeDouble, NULL);
121:
122:    Stack_free(oStack2);
123:    oStack2 = NULL; /* Unnecessary but safe. */
124:
125:    return 0;
126: }
```

Precept Activity Instructions:

- Study and compare the `stack6/teststack.c` and `stack6/stack.c` code.
- Discuss what happens when `teststack.c` calls:
  `Stack_map(oStack2, sumDouble, &dSum);`
- Discuss what happens when `teststack.c` calls:
  `Stack_map(oStack2, freeDouble, NULL);`
- In both cases, what are the three arguments that are passed to `Stack_map`?
- Why is the third argument of the `Stack_map()` call `NULL` when the callback function is `freeDouble`?
- Do we need to assert `pvExtra` is not `NULL` in the `Stack_map` function? Why or why not?

Precept Activity Answers:

- Study and compare the `stack6/teststack.c` and `stack6/stack.c` code.
- Discuss what happens when `teststack.c` calls:
  `Stack_map(oStack2, sumDouble, &dSum);`
  - `Stack_map() travels to each node in oStack2 and calls the sumDouble() function. It sends sumDouble()the current node's pvItem and the pointer it received to the client's sum variable.`
- Discuss what happens when `teststack.c` calls:
  `Stack_map(oStack2, freeDouble, NULL);`
  - `Stack_map() travels. to each node in oStack2 and calls the freeDouble() function. It sends freeDouble() the current node's pvItem and the NULL pointer it received from the client.`
- In both cases, what are the three arguments that are passed to `Stack_map`?
  - The three arguments are the pointer to the stack, the function pointer to the client's callback function and a pointer to one extra thing that might or might not be needed by the callback function.
- Why is the third argument of the `Stack_map()` call `NULL` when the callback function is `freeDouble`?
  - freeDouble() does not need any extra information.
- Do we need to assert `pvExtra` is not `NULL` in the `Stack_map` function? Why or why not?
  - We do NOT need or want to assert that pvExtra is not NULL. If the callback function does not need pvExtra, then pvExtra SHOULD be NULL.