Function Pointers

Instructor: Jee Ho Ryoo

Announcements

- Assignment 3
    - Extend deadline (Friday)
    - Corner case
    - Dec 7: 900-1030 SE325
    - Sample input
- No lab next week
- Friday all day office hour
- Final Exam
    - Dec 12: 1030AM
    - SW01-1205
    - Coverage (after midterm, will email)
    - More MCQ
    - 2 Coding Questions
    - Cheat sheet

What are they?

Just like having pointers to data types (int*, char*, etc) we can have pointers to functions
A function pointer is a variable that stores the address of a function
It provides us with more flexibility when using functions

Function Pointer Declaration

Consider this function called *SomeFunc* that returns void and takes in no values, this is how we declare a pointer for it:

```
void SomeFunc() {
    printf("Good morning!");
}

int main(){
    void (*SomeFuncPtr)() = &SomeFunc;
    return 0;
}
```

Function Pointer Declaration (cont.)

Let's update the function to see how the pointer declaration changes

Notice when we have function arguments, the empty brackets now are filled with the correct type

```
int SomeFunc(char c)
{
    return c;
}

int main(){
    int (*SomeFuncPtr)(char) = &SomeFunc;
    return 0;
}
```

Function Pointer Declaration (cont.)

We can point the function pointer on a different line like so

```
int SomeFunc(char c) {
    return c;
}

int main() {
    int (*SomeFuncPtr)(char);
    SomeFuncPtr = &SomeFunc;
    return 0;
}
```

Function Pointer Declaration (cont.)

You can also omit the address-of (&) operator when pointing it at a function

```
int SomeFunc(char c){
    return c;
}

int main() {
    int (*SomeFuncPtr)(char)
    SomeFuncPtr   = SomeFunc;
    return 0;
}
```

Invoking a function pointer

Continuing off our example, this is how we invoke the function pointer

…

```
int main() {
    int (*SomeFuncPtr)(char) = &SomeFunc;
    int x = (*SomeFuncPtr)('q’);
    printf("%d", x);
    return 0;
}
```

Invoking a function pointer (cont.)

We can also remove the asterisk * and brackets and call the function pointer like this

…

```
int main(){
    int (*SomeFuncPtr)(char) = &SomeFunc;
    int x = SomeFuncPtr('q’);
    printf("%d", x);
    return 0;
}
```

Using an array of function pointers

Just like regular pointers, we can use an array of function pointers, lets update the class activity we just did
Notice where the array square brackets are []

*int (*MathPtr[])(int, int) = {&MultiplyValues, &AddValues};*

Using an array of function pointers (cont.)

Let's invoke our function pointers now using our array index

```
…
int (*MathPtr[])(int, int) = {&MultiplyValues, &AddValues};

int main(){
    printf("%d\n", (*MathPtr[0])(5, 6));
    printf("%d", (*MathPtr[1])(7, 8));
    return 0;
}
```

Using an array of function pointers (cont.)

Let's update this example to work with an index entered through scanf

```
…
int (*MathPtr[])(int, int) = {&MultiplyValues, &AddValues};

int main(){
    int index, x = 5, y = 6;
    scanf("%d", &index);
    printf("%d\n", (*MathPtr[index])(x, y));
    return 0;
}
```

Function pointers with functions

Just like regular data type pointers, function pointers can be used as an argument for a function

Let's continue with from our previous example

```
…
void Calculate(int (*MathPtr)(int, int), int x, int y){
    int z = (*MathPtr)(x, y);
    printf("%d", z);
}
```

Function pointers with functions (cont.)

Let's use this function now with again continuing off our previous examples

…

```
int main() {
    int index, x = 5, y = 6;
    scanf("%d", &index);

    Calculate((*MathPtr[index]), x, y);
    return 0;
}
```

Function pointers with functions (cont.)

Function pointers can also be returned from functions
This function returns our pointer type and has one integer called *index* as an argument
We can return one of the pointers from our array of function pointers

```
…

int (*GetMathPtr(int index))(int, int) {
   return (*MathPtr[index]);
}
```

Function pointers with functions (cont.)

Let's put our new function as an argument to our *Calculate* function

…

```
int main(){
    int index, x = 5, y = 6;
    scanf("%d", &index);
    Calculate((*GetMathPtr(index)), x, y);
    return 0;
}
```

Function pointers with functions (cont.)

```
int MultiplyValues(int x, int y){   return x*y;   }
int AddValues(int x, int y) {    return x+y;   }
int (*MathPtr[])(int, int) = {&MultiplyValues, &AddValues};
void Calculate(int (*MathPtr)(int, int), int x, int y)
{
    int z = (*MathPtr)(x, y);
    printf("%d", z);
}
int (*GetMathPtr(int index))(int, int)
{
    return (*MathPtr[index]);
}
int main()
{
    int index, x = 5, y = 6;
    scanf("%d", &index);
    Calculate(*(GetMathPtr(index)), x, y);
    return 0;
}
```

Using typedef with function pointers

Do make things more readable we can give a type a new name using typedef

*typedef int (*MathPtr)(int, int);*

Once we have given it a new name it can be used without its full definition each time

Using typedef with function pointers (cont.)

For example after using typedef on our function pointer, we can declare our array like this

```
typedef int (*MathPtr)(int, int);
MathPtr mathPtr[] = {&MultiplyValues, &AddValues};
```

This is a lot more manageable compared to the old way

```
int (*MathPtr[])(int, int) = {&MultiplyValues, &AddValues};
```

Using typedef with function pointers (cont.)

We can also update the syntax to our *GetMathPtr* function which is great!

```
MathPtr GetMathPtr(int index){
    return mathPtr[index];   //also removed the brackets and asterisk
}
```

Compared to the old way

```
int (*GetMathPtr(int index))(int, int){
    return (*MathPtr[index]);
}
```

Using typedef with function pointers (cont.)

```
typedef int (*MathPtr)(int, int);
int MultiplyValues(int x, int y){ return x*y; }
int AddValues(int x, int y){ return x+y; }
MathPtr mathPtr[] = {MultiplyValues, AddValues};
void Calculate(MathPtr mathPtr, int x, int y)
{
       int z = mathPtr(x, y);
       printf("%d", z);
}
MathPtr GetMathPtr(int index)
{
       return mathPtr[index];
}
int main()
{
       int index, x = 5, y = 6;
       scanf("%d", &index);
       Calculate(GetMathPtr(index), x, y);
       return 0;
}
```

Preprocessors

Instructor: Jee Ho Ryoo

What is the C preprocessor (CPP) ?

Before the code compiles there is pre-processing that takes place

All CPP commands begin with a hash symbol (#)

The four main types of CPP directives are

Macros

File inclusion

Conditional Compilation

#undef

Why use macros?

Since macros are preprocessed not compiled, it can help with compile time

Although most modern compilers won't see a difference
Can help with code organization and code flexibility

Macros

We define macros using the *#define* directive followed by a name and some code

```
#define SIZE 10

int main(){
    printf("%d", SIZE);
    return 0;
}
```

Macros (cont.)

Macros can also have arguments and parameter like functions aka *function-like macros*

```
#define ADD_NUM(num) num+1

int main(){
    printf("%d", ADD_NUM(2));
    return 0;
}
```

Macros (cont.)

Order of operations is important when working with macros, consider this example

```
#define MULTIPLY(num) num*num

int main(){
    int x = 2;
    printf("%d", MULTIPLY(x+1));
    return 0;
}
```

Macros (cont.)

Order of operations is important when working with macros, consider this example

```
#define MULTIPLY(num) num*num

int main(){
    int x = 2;
    printf("%d", MULTIPLY(x+1));
    return 0;
}
```

The result here is 5 not 9 because, (x+1*x+1) = (2x+1) = (4+1) = 5

Macros (cont.)

When there is a multi lined macro you use the backslash \ symbol to continue to the next line

```
#define NUMBERS 9, \
                8, \
                7

int main(){
    int arr[] = {NUMBERS};
    printf("%d", arr[1]);
    return 0;
}
```

The Stringize (#) Operator

aka the *number-sign operator* it converts a macro parameter into a string constant (no quotes)

```
#define FUNC(str1, str2)\
          printf(#str1 " are" #str2);

int main() {
   FUNC(Pirates, Cool);
}
```

The Token Pasting (##) Operator

The token pasting operator combines two tokens

```
#define FUNC(str1, str2, num)\
          printf(#str1 " are " #str2 " %s", str##num);

int main() {
    char str3[] = "yar";
    FUNC(Pirates, Cool, 3);
}
```

The Token Pasting (##) Operator (cont.)

What are tokens?

- Tokens are the basic building blocks in C

Six types

- Keywords (eg: int, while)
- Identifiers (eg: main, x)
- Constants (eg: 10, 7)
- Strings (eg: "hello")
- Special Symbols (eg: (), {})
- Operators(eg: +, /, -, *)

File inclusion

Tells the compiler to include a file

We have used the *#include* directive so far to include certain header files

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

You can also include user defined files as well

```
#include <extra.c>
```

Conditional Compilation

Will compile a specific part of a program based on conditions
Common condition compilation directives are:

#ifdef: checks if a macro is defined

#ifndef: checks if a macro is <u>not</u> defined

#if: checks a macro condition

#elif: checks a macro condition if previous #ifdef, #ifndef, or #if is false

#else: will execute if previous #ifdef, #ifndef, #if, or #elif is false

Conditional Compilation Example

```
#define TESTING_PHASE 0
int main()
{
        #ifdef TESTING_PHASE
            printf("in testing phase\n");
        #if TESTING_PHASE == 0
            printf("first testing phase\n");
        #elif TESTING_PHASE == 1
            printf("second testing phase\n");
        #else
            printf("third testing phase\n");
        #endif
        #endif
            return 0;
}
```

#undef directive

#undef undefines an existing macro

```
#define TESTING_PHASE 0

int main() {
    #ifdef TESTING_PHASE
        printf("in testing phase\n");
    #endif
    #undef TESTING_PHASE
    #ifndef TESTING_PHASE
        printf("done testing phase");
    #endif
    return 0;
}
```

Makefile, .o, .h, and .c

Instructor: Jee Ho Ryoo

Breaking your program into files

main.c

stack.c

stack.h

Breaking your program into files

main.c

The main function, to actually do the “job”

stack.c

The code for a stack of integers.

stack.h

The “declarations” of a stack of integers.

Why break them up?

- main just needs a stack
  - It does not want (or need) to care how it is built or used!
- Smaller files are easier to read
- Faster to compile
  - More on this later
- Breaks the program into logical CHUNKS

stack.h

```
typedef struct S_stack {
  int number;
  struct S_stack *next;
} stack;

void push(int number, stack **stk_ptr);
int pop(stack **stk_ptr);
```

No actual code!
Just “this is the structure” and
These are the functions. … “never mind how they work”

stack.c

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

#include "stack.h"
```

Why include stack.h?
Note the “” instead of <>

<> means “include from the system libraries”

For predefined .h files

“” means “include from THIS directory”

For your OWN .h files

stack.c

```
void push(int number, stack **stk_ptr) {
  stack *stk, *tmp;
  stk = *stk_ptr;
  tmp = malloc(sizeof(stack));
  tmp->number = number;
  tmp->next = stk;
  stk = tmp;
  *stk_ptr = stk;
}
```

stack.c

```
int pop(stack **stk_ptr) {
  int number;
  stack *stk, *tmp;
  stk = *stk_ptr;
  tmp = stk;
  number = tmp->number;
  stk = stk->next;
          free(tmp);
  *stk_ptr = stk;
  return number;
}
```

main.c

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

#include "stack.h"
```

Why include stack.h this time?

main.c

```
int main() {
  stack *stk = NULL;
  push(7, &stk);
  push(2, &stk);
  push(9, &stk);
  push(12,&stk);
  printf("%d\n",pop(&stk));
  printf("%d\n",pop(&stk));
  printf("%d\n",pop(&stk));
  printf("%d\n",pop(&stk));
  printf("%d\n",pop(&stk));
  return 0;
}
```

Compiling multiple files (Opt 1)

gcc –Wall main.c stack.c

Compiles BOTH files... and makes a.out

Advantages:

Easy to remember

Disadvantages:

If you have a LOT of .c files, then it becomes tedious AND slow!

Compiling multiple files (Opt 2)

gcc –Wall –c main.c

turns main.c into main.o

gcc –Wall –c stack.c

turns stack.c into stack.o

gcc –Wall –o stacktest stack.o main.o

takes stack.o and main.o and makes “stacktest” out of them

Called “LINKING”

Whats a .o?

An “Object File”

Contains the compiled contents of the corresponding .c program

For example:

stack.o contains the computer-language version of stack.c

Can’t turn a .h into a .o (no code in .h)

Compiling multiple files (Opt 2)

Advantages:

Faster (Only recompile parts then re-link)

Disadvantages:

Loads of typing!

Makefiles

Automate the process

You tell the Makefile:

- What you want to make
- How it goes about making it

And it figures out

- What needs to be (re) compiled and linked
- What order to do it in

You just type “make”

Makefiles

Can be HUGELY complex

Just use the one I give you, and only modify the top parts

Makefiles could be a class on their own…

Makefile

```
CC       = gcc
CFLAGS   = -Wall
LDFLAGS  =
OBJFILES = stack.o main.o
TARGET   = stacktest

all: $(TARGET)

$(TARGET): $(OBJFILES)
            $(CC) $(CFLAGS) -o $(TARGET) $(OBJFILES)
$(LDFLAGS)

clean:
            rm -f $(OBJFILES) $(TARGET) *~
```

Makefile

```
CC       = gcc
CFLAGS   = -Wall
LDFLAGS  =
OBJFILES = stack.o main.o
TARGET   = stacktest

all: $(TARGET)

$(TARGET): $(OBJFILES)
            $(CC) $(CFLAGS) -o $(TARGET) $(OBJFILES)
$(LDFLAGS)

clean:
            rm -f $(OBJFILES) $(TARGET) *~
```

Which compiler to use

Makefile

```
CC       = gcc
CFLAGS   = -Wall
LDFLAGS  =
OBJFILES = stack.o main.o
TARGET   = stacktest

all: $(TARGET)

$(TARGET): $(OBJFILES)
	    $(CC) $(CFLAGS) -o $(TARGET) $(OBJFILES)
$(LDFLAGS)

clean:
	    rm -f $(OBJFILES) $(TARGET) *~
```

Which flags to use
-ggdb -Wall etc...

Makefile

```
CC       = gcc
CFLAGS   = -Wall
LDFLAGS  =
OBJFILES = stack.o main.o
TARGET   = stacktest

all: $(TARGET)

$(TARGET): $(OBJFILES)
            $(CC) $(CFLAGS) -o $(TARGET) $(OBJFILES)
$(LDFLAGS)

clean:
            rm -f $(OBJFILES) $(TARGET) *~
```

Which libraries to use
-lm   -lefence   etc…

Makefile

```
CC       = gcc
CFLAGS   = -Wall
LDFLAGS  =
OBJFILES = stack.o main.o
TARGET   = stacktest

all: $(TARGET)

$(TARGET): $(OBJFILES)
            $(CC) $(CFLAGS) -o $(TARGET) $(OBJFILES)
$(LDFLAGS)

clean:
            rm -f $(OBJFILES) $(TARGET) *~
```

Which object files are part of the final program

Makefile

```
CC       = gcc
CFLAGS   = -Wall
LDFLAGS  =
OBJFILES = stack.o main.o
TARGET   = stacktest

all: $(TARGET)

$(TARGET): $(OBJFILES)
            $(CC) $(CFLAGS) -o $(TARGET) $(OBJFILES)
$(LDFLAGS)

clean:
            rm -f $(OBJFILES) $(TARGET) *~
```

What to name the final prog

Makefile

```
CC        = gcc
CFLAGS    = -Wall
LDFLAGS   =
OBJFILES = stack.o main.o
TARGET    = stacktest

all: $(TARGET)

$(TARGET): $(OBJFILES)
            $(CC) $(CFLAGS) -o $(TARGET) $(OBJFILES)
$(LDFLAGS)

clean:
            rm -f $(OBJFILES) $(TARGET) *~
```

TAB
not several spaces
Sorry…

To use our Makefile:

Just type “make”

- It will figure out which .c files need to be recompiled and turned into .o files
  - If the .c file is newer than the .o file or the .o file does not exist
- Figures out if the program needs to be re-linked
  - If any of the .o files changed or
  - If the program does not exist

To use our Makefile:

Or type "make clean"

- Deletes:
  - all the .o files
  - all the ~ files (from emacs)
  - the program itself
- Leaves:
  - .c files
  - .h files
  - Makefile

To use our Makefile:

make clean
make

What happens?