COP3514 Program Design

Spring 2022

Instructor: Mauricio Pamplona Segundo

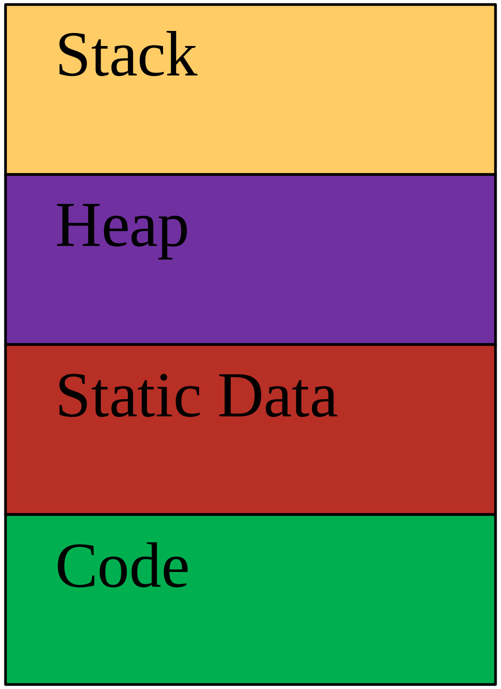
Agenda

- Functions that take a function pointer as a parameter.
- Call functions that take a function pointer as a parameter.
- Use the quicksort function in the Standard C Library.

Chapter 17

Function pointers

A Program’s Memory Layout



Stack

Heap

Static Data

Code

Local variables

Dynamically allocated memory

Global (external) variables

Program instructions

Pointers to Functions

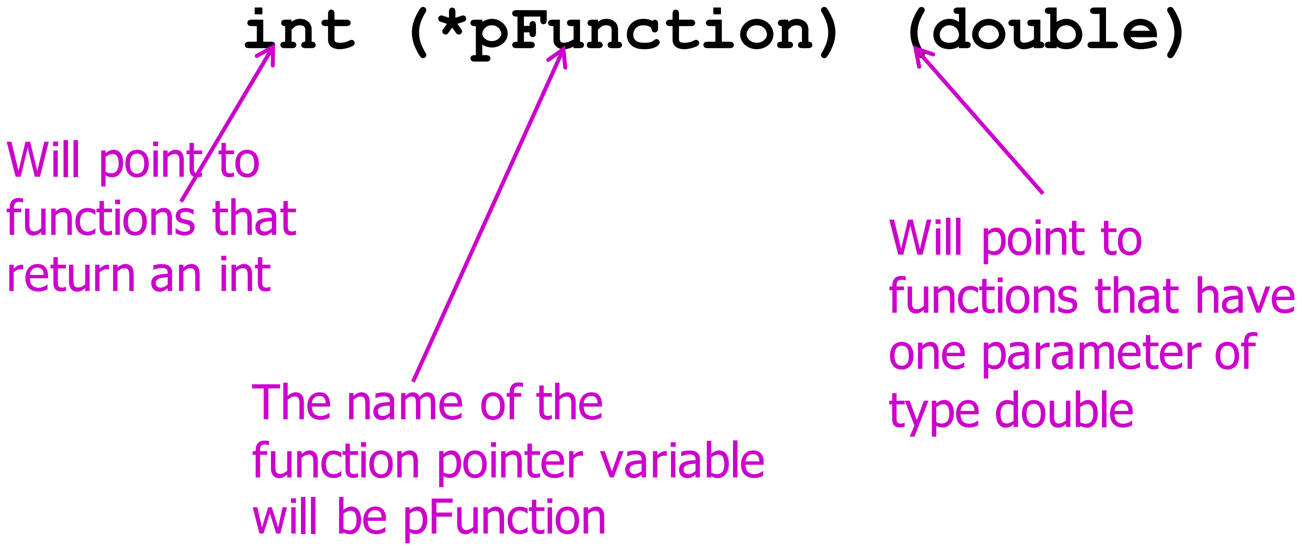
- C doesn’t require that pointers point only to *data;* it’s also possible to have pointers to *functions*.
- Functions occupy memory locations, so every function has an address.
- We can use function pointers in much the same way we use pointers to data.
- Passing a function pointer as an argument is fairly common.

Function Pointers

- When passing a function pointer as an argument, function pointer must be *declared*.
- Declaration tells the compiler
  - Return type
  - Number of parameters
  - Type of each parameter
- Function calls using a function pointer must have the right number of arguments and right types.

Declaring a Function Pointer

- The declaration specifies the return type and the types of the arguments.



```
int (*pFunction) (double)
```

Will point to functions that return an int

The name of the function pointer variable will be pFunction

Will point to functions that have one parameter of type double

Function Pointers as Arguments

- A function named `integrate` that integrates a mathematical function `f` can be made as general as possible by passing `f` as an argument.
- Prototype for `integrate`:

```
double integrate(double (*f)(double),
                 double a, double b);
```

The parentheses around `*f` indicate that `f` is a pointer to a function.

- An alternative prototype:

```
double integrate(double f(double),
                 double a, double b);
```

Function Pointers as Arguments

- A call of `integrate` that integrates the `sin` (sine) function from 0 to π/2:

  ```
  result = integrate(sin, 0.0, PI / 2);
  ```

- When a function name isn't followed by parentheses, the C compiler produces a pointer to the function.

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

#define PI 3.1415926

double integrate(double (*f)(double), double a, double b);

int main()
{
        double result;
        result = integrate(sin, 0.0, PI/2);
        printf("integrating sin function from 0.0 to PI/2,
   result is %.3lg\n", result);

        result = integrate(exp,  0.0, PI/2);
        printf("integrating exp function from 0.0 to PI/2,
   result is %.3lg\n", result);

         result = integrate(sqrt,  0.0, PI/2);
        printf("integrating sqrt function from 0.0 to PI/2,
   result is %.3lg\n", result);

   return 0;

}
```

Function Pointers as Arguments

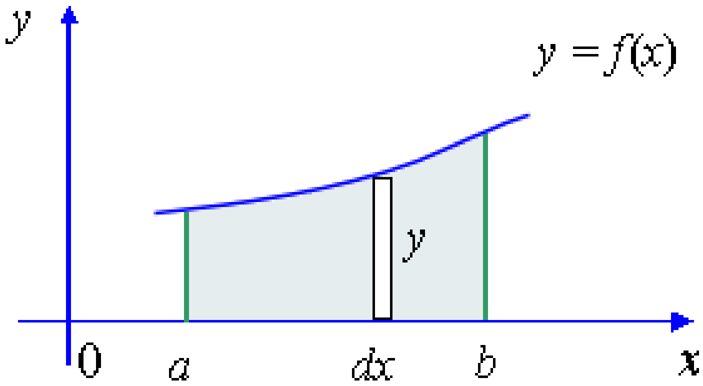
- Within the body of `integrate`, we can call the function that `f` points to:

  ```
  y = (*f)(x);
  ```

- Writing `f(x)` instead of `(*f)(x)` is allowed.

Implement the integrate function

- The integral of a function f from point a to point b is basically the area under the function from a to b.



- To calculate the area, we can sample the function at a small “stepsize” and calculate the area of the thin rectangle and then add up the areas.

```
double integrate(double (*f)(double),
  double a, double b)
{
    double stepsize = 0.01;
    double integral = 0.0;
    double x, area;
    for(x = a + stepsize; x <=b; x +=
  stepsize){
         area = f(x)*stepsize;
         integral += area;
     }
     return integral;
}
```

To compile a c program that include math.h:
gcc –lm program_name.c

Question?

- Which of the following statements will calculate the integral of square root of 0 to 1, using the integrate function described in the previous slide?

```
A) result=integrate(0.0, 1);
B) result=integrate(sqrt(x), 0.0, 1);
C) result=integrate(sqrt,  0.0, 1);
D) result=integrate(sqrt(0.0, 1));
```

qsort function

The `qsort` Function

- Some of the most useful functions in the C library require a function pointer as an argument.
- One of these is `qsort`, which belongs to the `<stdlib.h>` header.
- `qsort` is a general-purpose sorting function that's capable of sorting any **array**.

The `qsort` Function

- `qsort` must be told how to determine which of two array elements is "smaller."
- This is done by passing `qsort` a pointer to a ***comparison function.***
- When given two pointers `p` and `q` to array elements, the comparison function must return an integer that is:
  - *Negative* if `*p` is "less than" `*q`
  - *Zero* if `*p` is "equal to" `*q`
  - *Positive* if `*p` is "greater than" `*q`

The `qsort` Function

- Prototype for `qsort`:

```
void qsort(void *base, size_t nmemb, size_t size,
  int (*compar)(const void *, const void *));
```

- `base` must point to the first element in the array (or the first element in the portion to be sorted).
- `nmemb` is the number of elements to be sorted.
- `size` is the size of each array element, measured in bytes.
- `compar` is a pointer to the comparison function.

Using qsort

- Suppose we have a function that compares two integers, called `int_compare`

- To use `qsort` to sort the integer array `data`:

```
qsort (data, length, sizeof(int),
int_compare);
```

Sorting integers

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

int int_compare(const void* p, const void* q);

int main()
{
        int n, i;
        int *a;

        printf("Enter the length of the array: ");
        scanf("%d", &n);

        a = malloc(n*sizeof(int));
```

```
         for(i = 0; i < n; i++)
        {
                printf("Enter a number: ");
                scanf("%d", &a[i]);
        }

        qsort(a, n, sizeof(int), int_compare);

        printf("In sorted order:\n");

        for(i = 0; i < n; i++)
                printf("%d\t", a[i]);

        printf("\n");

        return 0;
}

int int_compare(const void* p, const void* q){
   //code to be filled in
}
```

Const Pointers as Function Parameters

- The compare function passed to `qsort` must be declared with two `const void*` parameters.

  ```
  int (*compar)(const void *, const void *);
  ```

- qsort will call the function when it needs to compare array entries.
  - qsort will pass the *addresses* of array elements to be compared.
  - Compare function must typecast the arguments as pointers to whatever type is in the array.
  - Must not use the pointers to modify anything.
  - Returns an int with the result of the comparison.

The `qsort` Function

- When `qsort` is called, it sorts the array into ascending order, calling the comparison function whenever it needs to compare array elements.
- A call of `qsort` that sorts the `inventory` array:

```
qsort(inventory, num_parts,
      sizeof(struct part), compare_parts);
```

- `compare_parts` is a function that compares two `part` structures.

The `qsort` Function

- Writing the `compare_parts` function is tricky.
- `qsort` requires that its parameters have type `void *`, but we can’t access the members of a `part` structure through a `void *` pointer.
- To solve the problem, `compare_parts` will assign its parameters, `p` and `q`, to variables of type `struct part *`.

The `qsort` Function

- A version of `compare_parts` that can be used to sort the `inventory` array into ascending order by part number:

```
int compare_parts(const void *p, const void *q)
{
  const struct part *p1 = p;
  const struct part *q1 = q;

  if (p1->number < q1->number)
    return -1;
  else if (p1->number == q1->number)
    return 0;
  else
    return 1;
}
```

The `qsort` Function

- Most C programmers would write the function more concisely:

```
int compare_parts(const void *p, const void *q)
{
  if (((struct part *) p)->number <
      ((struct part *) q)->number)
    return -1;
  else if (((struct part *) p)->number ==
           ((struct part *) q)->number)
    return 0;
  else
    return 1;
}
```

The **`qsort`** Function: Sort an array of strings

- To sort an array of strings using `qsort`, can we pass `strcmp` itself to `qsort`?

```
char *words[MAX_WORDS];
…
qsort(words, num_words,
sizeof(char *), strcmp); /*Wrong*/
```

The `qsort` Function: Sort an array of strings

- We can’t pass `strcmp` itself to `qsort`:
  - `qsort` requires a comparison function with two `const void *` parameters.

```
int (*compar)(const void *, const void *);
```

  - Prototype for the `strcmp` function:

```
int strcmp(const char *s1, const char *s2);
```

  - `strcmp` assumes `s1` and `s2` are strings (`char *` pointers).
  - `strcmp` compares the strings `s1` and `s2`, returning a value less than, equal to, or greater than 0.

Sort an array of strings

- Need to cast parameters of comparison function `(const void *)` to type `char**` – pointers to strings
- Then use `*` (indirection) operater to access the strings.
- Then use `strcmp` to compare strings in the comparison function for `qsort`:

```
int compare_strings(const void *p,
const void *q)
{
    return strcmp(*(char **)p, *(char
**)q);
}
```

Exercise #1

- Download `sum.c` on Canvas>Week 14 and complete the following function.
- The call `sum(g, i, j)` should return `g(i)+` …`+g(j).`

```
int sum (int (*f) (int), int start,
  int end);
```

The program reads in `start` and `end` (integers). In main function, add code so that the program displays the sum of factorials, the sum of squares, and the sum of cubes from `start` to `end`.

sum.c

```c
#include <stdio.h>

int sum (int (*f) (int), int start, int end);
int fact(int n);
int square(int n);
int cube(int n);

int main()
{
        int start, end;
        printf("Enter start value: ");
        scanf("%d", &start);
        printf("Enter end value: ");
        scanf("%d", &end);

        //display the sum of factorials, the sum of squares,
        //and the sum of cubes from start to end



        return 0;
}
```

```
int sum (int (*f) (int), int start, int end)
{


}

int fact(int n)
{
 if (n <= 1)
  return 1;
 else
  return n * fact(n - 1);
}

int square(int n)
{
 return n*n;
}

int cube(int n)
{
 return n*n*n;
}
```

Exercise #2

- Download `sort_ints.c` and complete the comparison function of integers for the `qsort` function.

```
int int_compare(const void* p, const
  void* q);
```

- In general, the comparison function for the `qsort` function must return an integer that is:
  - *Negative* if `*p` is "less than" `*q`
  - *Zero* if `*p` is "equal to" `*q`
  - *Positive* if `*p` is "greater than" `*q`

Exercise: Sorting integers

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

int int_compare(const void* p, const void* q);

int main()
{
        int n, i;
        int *a;

        printf("Enter the length of the array: ");
        scanf("%d", &n);

        a = malloc(n*sizeof(int));
```

```
         for(i = 0; i < n; i++)
        {
                printf("Enter a number: ");
                scanf("%d", &a[i]);
        }

        qsort(a, n, sizeof(int), int_compare);

        printf("In sorted order:\n");

        for(i = 0; i < n; i++)
                printf("%d\t", a[i]);

        printf("\n");

        return 0;
}

int int_compare(const void* p, const void* q){
   //code to be filled in

}
```