Intermediate Programming

Day 7

Outline

- Exercise 6
- Function declarations
- Passing arrays to functions
- Recursion
- Review questions

Exercise 6

- Open files for I/O

compound.c

```c
…
int main()
{
	…
	// TODO: Open filename for reading, handle errors
	FILE *in = fopen( filename , "r" );
	if( !in )
	{
		fprintf( stderr , "[ERROR] Could not open file for reading: %s\n" , filename );
		return 1;
	}

	// TODO: Open output.txt file for writing, handle errors
	FILE *out = fopen( "output.txt" , "w" );
	if( !out )
	{
		fprintf( stderr , "[ERROR] Could not open file for writing: output.txt\n" );
		fclose( in );
		return 1;
	}
	…
}
```

Exercise 6

- Repeatedly parse principal and rate and write to output file

compound.c

```
…
int main()
{
	…
	// Open files for I/O

	// TODO: parse p, r from file, proceed with loop if successful
	while ( ( parse=fscanf( in , " %f %f" , &p , &r ) ) == 2)
	{
		…
		// TODO: print the three answers to the output file
		//	using "%0.2f %0.2f %0.2f\n" as the fprintf
		//	format string.  Print ci_annual, ci_monthly
		//	then ci_cont.
		fprintf( out , "%0.2f %0.2f %0.2f\n" , ci_annual , ci_monthly , ci_cont );
	}
	…
}
```

Exercise 6

- Check status and close files if all is good

compound.c

```
…
int main()
{
	…
	// Open files for I/O
	// Repeatedly parse principal and rate and write to output file

	// TODO: return non-0 if error prevented us from completing
	if( parse && parse!=EOF )
	{
		fprintf( stderr , "[ERROR] Failed to read input: %d\n" , parse );
		return 1;
	}

	// TODO: use ferror to check both input and output for errors
	if( ferror( in ) || ferror( out ) )
	{
		fprintf( stderr , "[ERROR] file handle(s) in bad state\n" );
		return 1;
	}

	// TODO: close both input and output using fclose
	fclose( in );
	fclose( out );
	…
}
```

Exercise 6

- Compute the compounded interest

compound.c

```
…
float compound_interest( float p , float r , int n )
{
	…
	if( n>0 )
	{
		// TODO: Compute and return compound interest
		return p * pow( 1+r/n , n );
	}
	else
	{
		// TODO: Compute and return continuously compounded interest
		return p * exp( r );
	}
}
…
```

Outline

- Exercise 6
- Function declarations
- Passing arrays to functions
- Recursion
- Review questions

Function declarations

If we (only) define a function after we use it, the compiler will complain.

```
#include <stdio.h>

int main( void )
{
        int a = 7;
        int b = foo(a);
        printf( "%d\n" , b );
        return 0;
}

int foo( int x )
{
        int y = 5;
        return x+y;
}
```

```
>> gcc foo.c
foo.c: In function ‘main’:
foo.c:7:10: warning: implicit declaration of function ‘foo’ [-Wimplicit-function-declaration]
    7 |  int b = foo( a );
      |          ^~~
>>
```

Function declarations

- We can <u>declare</u> a function before it is called and only <u>define</u> it after.
  - Note semicolon after parameter list
  - Declaration should appear before the first call to the function
  - A function declaration is also known as a *function prototype*
  - Names of parameters (e.g., x) are optional, but can be illuminating
    - It is recommended that you have them (and that they are meaningful) in your declarations so it's easier to understand what the function does and what to pass in.
  - Code tends to be more readable if functions are declared before the main function and defined after.

```c
#include <stdio.h>
int foo( int );
int main( void )
{
	int a = 7;
	int b = foo(a);
	printf( "%d\n" , b );
	return 0;
}

int foo( int x )
{
	int y = 5;
	return x+y;
}
```

Function declarations

Creating an executable is done in three steps

1. Pre-processor:
   Function <u>declarations</u> are brought in using `#include`, etc.
2. Compilation:
   Functions are transformed from source code to object code
3. Linking:
   Bring together relevant object code (e.g. if a function calls another function)

When compiling a function that invokes other functions, the compiler does not need to know what the other function does, only what it takes as input and returns as output.

⇒ The compiler only needs to know the declaration of the invoked functions.

Outline

- Exercise 6
- Function declarations
- Passing arrays to functions
- Recursion
- Review questions

Passing arrays to functions

Recall:

- Argument values in C are passed by value

⇒ The function sees a copy of the value passed in as an argument

⇒ Changes made to the argument within the function will not be seen when the function returns.

```
#include <stdio.h>
void increment( int i ) { i += 1; }
int main( void )
{
    int i = 1;
    printf( "i = %d\n" , i );
    increment( i );
    printf( "i = %d\n" , i );
    return 0;
}
```

```
>> gcc temp.c -std=c99 -pedantic -Wall –Wextra
>> ./a.out
i = 1
i = 1
>>
```

Passing arrays to functions

Recall:

- Argument values in C are passed by value
- The `sizeof` function returns the size of a type/variable

```
#include <stdio.h>
int main(void)
{
        int x = 75;
        printf( "Size of char: %d\n" , sizeof( char ) );
        printf( "Size of int: %d\n" , sizeof( x ) );
        return 0;
}
```

```
>> ./a.out
Size of char: 1
Size of int: 4
>>
```

Passing arrays to functions

Recall:

- Argument values in C are passed by value
- The `sizeof` function returns the size of a type/variable
- You can determine the size of the contents of a static array using `sizeof`

```
#include <stdio.h>
int main(void)
{
        int values[] = { 0 , 130 };
        printf( "Array size: %d\n" , sizeof( values ) );
        return 0;
}
```

```
>> ./a.out
Array size: 8
>>
```

Passing arrays to functions

Recall:

- Argument values in C are passed by value
- The `sizeof` function returns the size of a type/variable
- You can determine the size of the contents of a static array using `sizeof`

Disclaimer:

This is only partially true

Passing arrays to functions

As with other variables, you can pass an array as an argument to a function.

```
#include <stdio.h>

void print( int values[] , int cnt )
{
	for( int i=0 ; i<cnt ; i++ ) printf( " %d" , values[i] );
	printf( "\n" );
}
int main( void )
{
	int v[] = { 9 , 7 , 5 , 6 , 8 , 10 };
	int cnt = (int)sizeof(v)/sizeof(int);
	print( v , cnt );
	return 0;
}
```

```
>> ./a.out
 9 7 5 6 8 10
>>
```

Passing arrays to functions

However:

1. The function <u>can</u> change the contents of the array.

```
#include <stdio.h>

void print( int values[] , int cnt )
{
	for( int i=0 ; i<cnt ; i++ ) printf( " %d" , values[i] );
	printf( "\n" );
}
void setToZero( int values[] , int idx )
{
	values[idx] = 0;
}
int main( void )
{
	int v[] = { 9 , 7 , 5 , 6 , 8 , 10 };
	int cnt = (int)sizeof(v)/sizeof(int);
	print( v , cnt );
	setToZero( v , 1 );
	print( v , cnt );
	return 0;
}
```

```
>> ./a.out
 9 7 5 6 8 10
 9 0 5 6 8 10
>>
```

Passing arrays to functions

However:

1. The function <u>can</u> change the contents of the array.
   You can use the keyword **const** to indicate that your function will not change the contents of the array.

```
#include <stdio.h>

void print( const int values[] , cnt sz )
{
	for( int i=0 ; i<cnt ; i++ ) printf( " %d" , values[i] );
	printf( "\n" );
}
void setToZero( int values[] , int idx )
{
	values[idx] = 0;
}
int main( void )
{
	int v[] = { 9 , 7 , 5 , 6 , 8 , 10 };
	int cnt = (int)sizeof(v)/sizeof(int);
	print( v , cnt );
	setToZero( v , 1 );
	print( v , cnt );
	return 0;
}
```

```
>> ./a.out
 9 7 5 6 8 10
 9 0 5 6 8 10
>>
```

Passing arrays to functions

However:

1. The function <u>can</u> change the contents of the array.
   You can use the keyword **const** to indicate that your function will not change the contents of the array.

```
#include <stdio.h>

void print( const int values[] , cnt sz )
{
	for( int i=0 ; i<cnt ; i++ ) printf( " %d" , values[i] );
	printf( "\n" );
}
void setToZero( const int values[] , int idx )
{
	values[idx] = 0;
}
int main( void )
{
	int v[] = { 9 , 7 , 5 , 6 , 8 , 10 };
	int cnt = (int)sizeof(v)/sizeof(int);
```

```
>> gcc ...
foo.c: In function ‘setToZero’:
foo.c:10:14: error: assignment of read-only location ‘*(values + (sizetype)((long unsigned int)idx * 4))’
   10 |  values[idx] = 0;
      |              ^
>>
```

Passing arrays to functions

However:

1. The function can change the contents of the array.
2. If you pass an array to a function and call the **sizeof** function on it, you will not get the size of the array.

```
#include <stdio.h>

int arraySize( const int values[] )
{
	return sizeof( values );
}
int main( void )
{
	int v[] = { 9 , 7 , 5 , 6 , 8 , 10 };
	printf( "%d %d\n" , sizeof( v ) , arraySize( v ) );
	return 0;
}
```

```
>> ./a.out
24 8
>>
```

Passing arrays to functio

- If function func1 calls func2:
  - ✓ If you declare an array in the body of func1, you can pass it as an argument to func2 (which can then pass it on to another function, etc.)

```
#include <stdio.h>

void swap( int values[] , int idx1 , int idx2 )
{
	int tmp = values[ idx1 ];
	values[ idx1 ] = values[ idx2 ];
	values[ idx2 ] = tmp;
}
void reverse( int values[] , int cnt )
{
	for( int i=0 ; i<cnt/2 ; i++ ) swap( values , i , cnt-1-i );
}
void print( const int values[] , int cnt )
{
	for( int i=0 ; i<cnt ; i++ ) printf( " %d" , values[i] );
	printf( "\n" );
}
int main( void )
{
	int v[] = { 9 , 7 , 5 , 6 , 8 , 10 };
	int cnt = (int)sizeof(v)/sizeof(int);
	print( v , cnt );
	reverse( v , cnt );
	print( v , cnt );
	return 0;
}
```

```
>> ./a.out
 9 7 5 6 8 10
 10 8 6 5 7 9
>>
```

Passing arrays to functio

- If function func1 calls func2:
  - ✓ If you declare an array in the body of func1, you can pass it as an argument to func2
  - ✗ If you declare a (static) array in the body of func2, you cannot return it to func1 (yet).

```
#include <stdio.h>

int *reverse( const int values[] , int cnt )
{
	int rev_values[cnt];
	for( int i=0 ; i<cnt ; i++ ) rev_values[i] = values[cnt-1-i];
	return rev_values;
}
void print( const int values[] , int cnt )
{
	for( int i=0 ; i<cnt ; i++ ) printf( " %d" , values[i] );
	printf( "\n" );
}
int main( void )
{
	int v[] = { 9 , 7 , 5 , 6 , 8 , 10 };
	int cnt = (int)sizeof(v)/sizeof(int);
	print( v , cnt );
	int *rev_v = reverse( v , cnt );
	print( rev_v , cnt );
	return 0;
}
```

```
>> ./a.out
 9 7 5 6 8 10
Segmentation fault (core dumped)
>>
```

Outline

- Exercise 6
- Function declarations
- Passing arrays to functions
- Recursion
- Review questions

Recursion

We’ve already seen that a function can call other functions within it’s body.

```
#include <stdio.h>

void printReverse( const int values[] , int cnt )
{
	for( int i=0 ; i<cnt ; i++ ) printf( " %d" , values[cnt-1-i] );
	printf( "\n" );
}
int main( void )
{
	int v[] = { 9 , 7 , 5 , 6 , 8 , 10 };
	int cnt = (int)sizeof(v)/sizeof(int);
	printReverse( v , cnt );
	return 0;
}
```

```
>> ./a.out
 10 8 6 5 7 9
>>
```

Recursion

We've already seen that a function can call other functions within it's body.

A function can also call itself.

```
#include <stdio.h>

void printReverse( const int values[] , int cnt )
{
	if( !cnt ) printf( "\n" );
	else
	{
		printf( " %d" , values[cnt-1] );
		printReverse( values , cnt-1 );
	}
}
int main( void )
{
	int v[] = { 9 , 7 , 5 , 6 , 8 , 10 };
	int cnt = (int)sizeof(v)/sizeof(int);
	printReverse( v , cnt );
	return 0;
}
```

```
>> ./a.out
 9 7 5 6 8 10
>>
```

Recursion

Like a **while** loop, a recursive function call can go on indefinitely.

```
#include <stdio.h>

void printReverse( const int values[] , int cnt )
{
	if( !cnt ) printf( "\n" );
	else
	{
		printf( " %d" , values[cnt-1] );
		printReverse( values , cnt-1 );
	}
}
int main( void )
{
	int v[] = { 9 , 7 , 5 , 6 , 8 , 10 };
	int cnt = (int)sizeof(v)/sizeof(int);
	printReverse( v , cnt );
	return 0;
}
```

```
>> ./a.out
 9 7 5 6 8 10
>>
```

Recursion

Like a **while** loop, a recursive function call can go on indefinitely.

For it to terminate, you need:

1. <u>A base case</u>
   The parameters are "so simple" that the function can solve the problem directly, without recursion

```
#include <stdio.h>

void printReverse( const int values[] , int cnt )
{
	if( !cnt ) printf( "\n" );
	else
	{
		printf( " %d" , values[cnt-1] );
		printReverse( values , cnt-1 );
	}
}
int main( void )
{
	int v[] = { 9 , 7 , 5 , 6 , 8 , 10 };
	int cnt = (int)sizeof(v)/sizeof(int);
	printReverse( v , cnt );
	return 0;
}
```

```
>> ./a.out
 9 7 5 6 8 10
>>
```

Recursion

Like a **while** loop, a recursive function call can go on indefinitely.

For it to terminate, you need:

1. <u>A base case</u>
   The parameters are "so simple" that the function can solve the problem directly, without recursion
2. <u>Progress towards the base case</u>
   When invoked recursively, the parameters become "simpler"

```
#include <stdio.h>

void printReverse( const int values[] , int cnt )
{
	if( !cnt ) printf( "\n" );
	else
	{
		printf( " %d" , values[cnt-1] );
		printReverse( values , cnt-1 );
	}
}
int main( void )
{
	int v[] = { 9 , 7 , 5 , 6 , 8 , 10 };
	int cnt = (int)sizeof(v)/sizeof(int);
	printReverse( v , cnt );
	return 0;
}
```

```
>> ./a.out
 9 7 5 6 8 10
>>
```

Recursion

When you invoke a function, the execution "jumps" from where you are in the code to where the function is in the code.

In order to be able to track where to go back to after the function is invoked, you need to store (among many other things) where you were when you called the function.

A *stack frame* stores this information for a given function call.

The collection of stack frames generated by all the function calls so far is stored on the *call stack*.

```
#include <stdio.h>

void printReverse( const int values[] , int cnt )
{
	if( !cnt ) printf( "\n" );
	else
	{
		printf( " %d" , values[cnt-1] );
		printReverse( values , cnt-1 );
	}
}
int main( void )
{
	int v[] = { 9 , 7 , 5 , 6 , 8 , 10 };
	int cnt = (int)sizeof(v)/sizeof(int);
	printReverse( v , cnt );
	return 0;
}
```

Recursion

The downside of recursion is that every time you recurse, you need to add another stack frame to the call stack.

⇒ Recursive function calls come with a compute/memory overhead.

```
#include <stdio.h>

void printReverse( const int values[] , int cnt )
{
	if( !cnt ) printf( "\n" );
	else
	{
		printf( " %d" , values[cnt-1] );
		printReverse( values , cnt-1 );
	}
}
int main( void )
{
	int v[] = { 9 , 7 , 5 , 6 , 8 , 10 };
	int cnt = (int)sizeof(v)/sizeof(int);
	printReverse( v , cnt );
	return 0;
}
```

Recursion

The downside of recursion is that every time you recurse, you need to add another stack frame to the call stack.

⇒ Recursive function calls come with a compute/memory overhead.

```
#include <stdio.h>

void printReverse( const int values[] , int cnt )
{
	if( !cnt ) printf( "\n" );
	else
	{
		printf( " %d" , values[cnt-1] );
		printReverse( values , cnt-1 );
	}
}
int main( void )
{
	int v[] = { 9 , 7 , 5 , 6 , 8 , 10 };
	int cnt = (int)sizeof(v)/sizeof(int);
	printReverse( v , cnt );
	return 0;
}
```

cnt=6

```
>> ./a.out
```

main

call stack

Recursion

The downside of recursion is that every time you recurse, you need to add another stack frame to the call stack.

⇒ Recursive function calls come with a compute/memory overhead.

printReverse (cnt=6)
main

call stack

```
#include <stdio.h>

void printReverse( const int values[] , int cnt )
{
        if( !cnt ) printf( "\n" );
        else
        {
                printf( " %d" , values[cnt-1] );
                printReverse( values , cnt-1 );
        }
}
int main( void )
{
        int v[] = { 9 , 7 , 5 , 6 , 8 , 10 };
        int cnt = (int)sizeof(v)/sizeof(int);
        printReverse( v , cnt );
        return 0;
}
```

cnt=5

```
>> ./a.out
 10
```

Recursion

The downside of recursion is that every time you recurse, you need to add another stack frame to the call stack.

⇒ Recursive function calls come with a compute/memory overhead.

printReverse (cnt=5)
printReverse (cnt=6)
main

call stack

```
#include <stdio.h>

void printReverse( const int values[] , int cnt )
{
        if( !cnt ) printf( "\n" );
        else
        {
                printf( " %d" , values[cnt-1] );
                printReverse( values , cnt-1 );
        }
}
int main( void )
{
        int v[] = { 9 , 7 , 5 , 6 , 8 , 10 };
        int cnt = (int)sizeof(v)/sizeof(int);
        printReverse( v , cnt );
        return 0;
}
```

cnt=4

```
>> ./a.out
 10 8
```

Recursion

The downside of recursion is that every time you recurse, you need to add another stack frame to the call stack.

⇒ Recursive function calls come with a compute/memory overhead.

call stack
printReverse (cnt=4)
printReverse (cnt=5)
printReverse (cnt=6)
main

```
#include <stdio.h>

void printReverse( const int values[] , int cnt )
{
        if( !cnt ) printf( "\n" );
        else
        {
                printf( " %d" , values[cnt-1] );
                printReverse( values , cnt-1 );
        }
}
int main( void )
{
        int v[] = { 9 , 7 , 5 , 6 , 8 , 10 };
        int cnt = (int)sizeof(v)/sizeof(int);
        printReverse( v , cnt );
        return 0;
}
```

cnt=3

```
>> ./a.out
 10 8 6
```

Recursion

The downside of recursion is that every time you recurse, you need to add another stack frame to the call stack.

⇒ Recursive function calls come with a compute/memory overhead.

printReverse (cnt=3)
printReverse (cnt=4)
printReverse (cnt=5)
printReverse (cnt=6)
main

call stack

```
#include <stdio.h>

void printReverse( const int values[] , int cnt )
{
	if( !cnt ) printf( "\n" );
	else
	{
		printf( " %d" , values[cnt-1] );
		printReverse( values , cnt-1 );
	}
}
int main( void )
{
	int v[] = { 9 , 7 , 5 , 6 , 8 , 10 };
	int cnt = (int)sizeof(v)/sizeof(int);
	printReverse( v , cnt );
	return 0;
}
```

cnt=2

```
>> ./a.out
 10 8 6 5
```

Recursion

The downside of recursion is that every time you recurse, you need to add another stack frame to the call stack.

⇒ Recursive function calls come with a compute/memory overhead.

```c
#include <stdio.h>

void printReverse( const int values[] , int cnt )
{
	if( !cnt ) printf( "\n" );
	else
	{
		printf( " %d" , values[cnt-1] );
		printReverse( values , cnt-1 );
	}
}
int main( void )
{
	int v[] = { 9 , 7 , 5 , 6 , 8 , 10 };
	int cnt = (int)sizeof(v)/sizeof(int);
	printReverse( v , cnt );
	return 0;
}
```

cnt=1

```
>> ./a.out
 10 8 6 5 7
```

printReverse (cnt=2)
printReverse (cnt=3)
printReverse (cnt=4)
printReverse (cnt=5)
printReverse (cnt=6)
main

call stack

Recursion

The downside of recursion is that every time you recurse, you need to add another stack frame to the call stack.

→ ... calls come with a ... overhead.

printReverse (cnt=1)
printReverse (cnt=2)
printReverse (cnt=3)
printReverse (cnt=4)
printReverse (cnt=5)
printReverse (cnt=6)
main

call stack

```
#include <stdio.h>

void printReverse( const int values[] , int cnt )
{
	if( !cnt ) printf( "\n" );
	else
	{
		printf( " %d" , values[cnt-1] );
		printReverse( values , cnt-1 );
	}
}
int main( void )
{
	int v[] = { 9 , 7 , 5 , 6 , 8 , 10 };
	int cnt = (int)sizeof(v)/sizeof(int);
	printReverse( v , cnt );
	return 0;
}
```

cnt=0

```
>> ./a.out
 10 8 6 5 7 9
```

Recursion

The downside of recursion is that every time you recurse, you need to add another stack frame to the call stack.

calls come with a
overhead.

printReverse (cnt=0)
printReverse (cnt=1)
printReverse (cnt=2)
printReverse (cnt=3)
printReverse (cnt=4)
printReverse (cnt=5)
printReverse (cnt=6)
main

call stack

```
#include <stdio.h>

void printReverse( const int values[] , int cnt )
{
	if( !cnt ) printf( “\n” );
	else
	{
		printf( “ %d” , values[cnt-1] );
		printReverse( values , cnt-1 );
	}
}
int main( void )
{
	int v[] = { 9 , 7 , 5 , 6 , 8 , 10 };
	int cnt = (int)sizeof(v)/sizeof(int);
	printReverse( v , cnt );
	return 0;
}
```

```
>> ./a.out
 10 8 6 5 7 9
```

Recursion

The downside of recursion is that every time you recurse, you need to add another stack frame to the call stack.

calls come with a overhead.

printReverse (cnt=1)
printReverse (cnt=2)
printReverse (cnt=3)
printReverse (cnt=4)
printReverse (cnt=5)
printReverse (cnt=6)
main

call stack

```c
#include <stdio.h>

void printReverse( const int values[] , int cnt )
{
	if( !cnt ) printf( "\n" );
	else
	{
		printf( " %d" , values[cnt-1] );
		printReverse( values , cnt-1 );
	}
}
int main( void )
{
	int v[] = { 9 , 7 , 5 , 6 , 8 , 10 };
	int cnt = (int)sizeof(v)/sizeof(int);
	printReverse( v , cnt );
	return 0;
}
```

```
>> ./a.out
 10 8 6 5 7 9
```

Recursion

The downside of recursion is that every time you recurse, you need to add another stack frame to the call stack.

⇒ Recursive function calls come with a compute/memory overhead.

printReverse (cnt=2)
printReverse (cnt=3)
printReverse (cnt=4)
printReverse (cnt=5)
printReverse (cnt=6)
main

call stack

```
#include <stdio.h>

void printReverse( const int values[] , int cnt )
{
	if( !cnt ) printf( "\n" );
	else
	{
		printf( " %d" , values[cnt-1] );
		printReverse( values , cnt-1 );
	}
}
int main( void )
{
	int v[] = { 9 , 7 , 5 , 6 , 8 , 10 };
	int cnt = (int)sizeof(v)/sizeof(int);
	printReverse( v , cnt );
	return 0;
}
```

```
>> ./a.out
 10 8 6 5 7 9
```

Recursion

The downside of recursion is that every time you recurse, you need to add another stack frame to the call stack.

⇒ Recursive function calls come with a compute/memory overhead.

printReverse (cnt=3)
printReverse (cnt=4)
printReverse (cnt=5)
printReverse (cnt=6)
main

call stack

```
#include <stdio.h>

void printReverse( const int values[] , int cnt )
{
    if( !cnt ) printf( "\n" );
    else
    {
        printf( " %d" , values[cnt-1] );
        printReverse( values , cnt-1 );
    }
}
int main( void )
{
    int v[] = { 9 , 7 , 5 , 6 , 8 , 10 };
    int cnt = (int)sizeof(v)/sizeof(int);
    printReverse( v , cnt );
    return 0;
}
```

```
>> ./a.out
 10 8 6 5 7 9
```

Recursion

The downside of recursion is that every time you recurse, you need to add another stack frame to the call stack.

⇒ Recursive function calls come with a compute/memory overhead.

printReverse (cnt=4)
printReverse (cnt=5)
printReverse (cnt=6)
main

call stack

```
#include <stdio.h>

void printReverse( const int values[] , int cnt )
{
	if( !cnt ) printf( "\n" );
	else
	{
		printf( " %d" , values[cnt-1] );
		printReverse( values , cnt-1 );
	}
}
int main( void )
{
	int v[] = { 9 , 7 , 5 , 6 , 8 , 10 };
	int cnt = (int)sizeof(v)/sizeof(int);
	printReverse( v , cnt );
	return 0;
}
```

```
>> ./a.out
 10 8 6 5 7 9
```

Recursion

The downside of recursion is that every time you recurse, you need to add another stack frame to the call stack.

⇒ Recursive function calls come with a compute/memory overhead.

```c
#include <stdio.h>

void printReverse( const int values[] , int cnt )
{
	if( !cnt ) printf( "\n" );
	else
	{
		printf( " %d" , values[cnt-1] );
		printReverse( values , cnt-1 );
	}
}
int main( void )
{
	int v[] = { 9 , 7 , 5 , 6 , 8 , 10 };
	int cnt = (int)sizeof(v)/sizeof(int);
	printReverse( v , cnt );
	return 0;
}
```

```
>> ./a.out
 10 8 6 5 7 9
```

printReverse (cnt=5)
printReverse (cnt=6)
main

call stack

Recursion

The downside of recursion is that every time you recurse, you need to add another stack frame to the call stack.

⇒ Recursive function calls come with a compute/memory overhead.

```
#include <stdio.h>

void printReverse( const int values[] , int cnt )
{
	if( !cnt ) printf( "\n" );
	else
	{
		printf( " %d" , values[cnt-1] );
		printReverse( values , cnt-1 );
	}
}
int main( void )
{
	int v[] = { 9 , 7 , 5 , 6 , 8 , 10 };
	int cnt = (int)sizeof(v)/sizeof(int);
	printReverse( v , cnt );
	return 0;
}
```

```
>> ./a.out
 10 8 6 5 7 9
```

printReverse (cnt=6)
main

call stack

Recursion

The downside of recursion is that every time you recurse, you need to add another stack frame to the call stack.

⇒ Recursive function calls come with a compute/memory overhead.

```c
#include <stdio.h>

void printReverse( const int values[] , int cnt )
{
	if( !cnt ) printf( "\n" );
	else
	{
		printf( " %d" , values[cnt-1] );
		printReverse( values , cnt-1 );
	}
}
int main( void )
{
	int v[] = { 9 , 7 , 5 , 6 , 8 , 10 };
	int cnt = (int)sizeof(v)/sizeof(int);
	printReverse( v , cnt );
	return 0;
}
```

```
>> ./a.out
 10 8 6 5 7 9
```

main

call stack

Recursion

The downside of recursion is that every time you recurse, you need to add another stack frame to the call stack.

⇒ Recursive function calls come with a compute/memory overhead.

```
#include <stdio.h>

void printReverse( const int values[] , int cnt )
{
	if( !cnt ) printf( "\n" );
	else
	{
		printf( " %d" , values[cnt-1] );
		printReverse( values , cnt-1 );
	}
}
int main( void )
{
	int v[] = { 9 , 7 , 5 , 6 , 8 , 10 };
	int cnt = (int)sizeof(v)/sizeof(int);
	printReverse( v , cnt );
	return 0;
}
```

```
>> ./a.out
 10 8 6 5 7 9
>>
```

call stack

Recursion

The downside of recursion is that every time you recurse, you need to add another stack frame to the call stack.

⇒ Recursive function calls come with a compute/memory overhead.

⇒ If you recurse too deeply, you will run out of memory on the call stack.

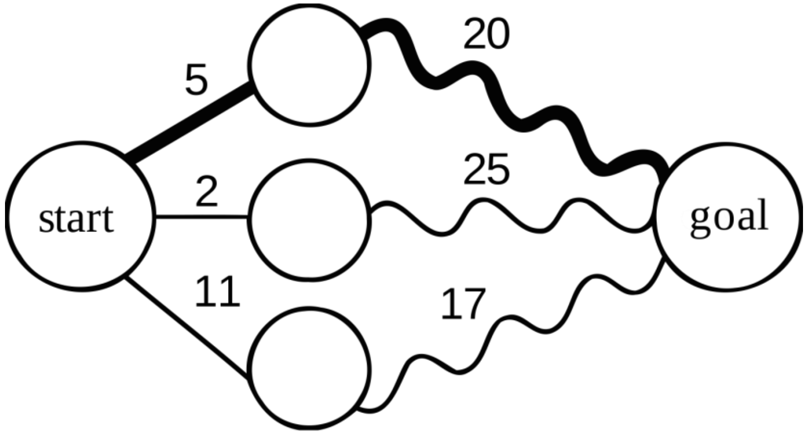
```
#include <stdio.h>

void printReverse( const int values[] , int cnt )
{
	if( !cnt ) printf( "\n" );
	else
	{
		printf( " %d" , values[cnt-1] );
		printReverse( values , cnt-1 );
	}
}
int main( void )
{
	int v[300000] = { 0 };
	int cnt = (int)sizeof(v)/sizeof(int);
	printReverse( v , cnt );
	return 0;
}
```

```
>> ./a.out
.. Segmentation fault (core dumped)
>>
```

Recursion

Recursion can be the natural for problems where:

- The algorithm uses a “divide and conquer” approach
- The data structure is recursive



https://en.wikipedia.org/wiki/Dynamic_programming

Example:

Given a (positively) weighted graph, find a shortest path from the start to the goal:

- Iterate over the neighbors,
  - Compute the shortest path from the neighbor to the goal,
  - Sum the distance to the neighbor and the length of its shortest path
- Return the path with smallest sum

Outline

- Exercise 6
- Function declarations
- Passing arrays to functions
- Recursion
- Review questions

Review questions

1. How do you get the number of elements in an integer array?

`sizeof(arr)/sizeof(int)`, assuming that:

- The array `arr` is statically declared
- `sizeof` is called from within the function where `arr` is declared.

Review questions

2. Is the size of a string array the same as the string length?

No. For example, if we declare:

```
char foo[] = { "abc\0def\0" };
```

Then the array size is 8 while the string length is 3.

Review questions

3. What is the difference between a function declaration and a function definition?

A function declaration does not have a body (and ends with a semi-colon).

Review questions

4. Can you have two functions with the same function name in a program?

No

Review questions

5. How does passing an integer array to a function differs from passing a single integer variable into a function?

The function can modify the contents of the array.

Review questions

6. How can you make an array that is passed into a function not modifiable?

Declare the function argument to be `const`.

Review questions

7. What is the down-side to recursion?

Overhead of using the call stack.

Exercise 7

- Website -> Course Materials -> Exercise 7