

Pointers and Indirection

By the end of this lecture, you will be able to describe the memory model of a C program.

You will also be able to use pointers in a C program to control what happens in memory.

Lecture 04 Summary

- Process Memory
- Pointers
 - Declaring
 - Dereferencing
 - Pointer Arithmetic
- Dynamic Memory Allocation
- Passing Parameters by Reference

How do you break a problem down in order to solve it using a computer program?

Process Memory

- One (good) approach:
 - Find *entities* which exhibit *state*
 - Analyze how the state of each entity changes
 - Create *variables* (or data structures) to hold the state of the entities
 - Create *code* that describes how to change the state of the entities

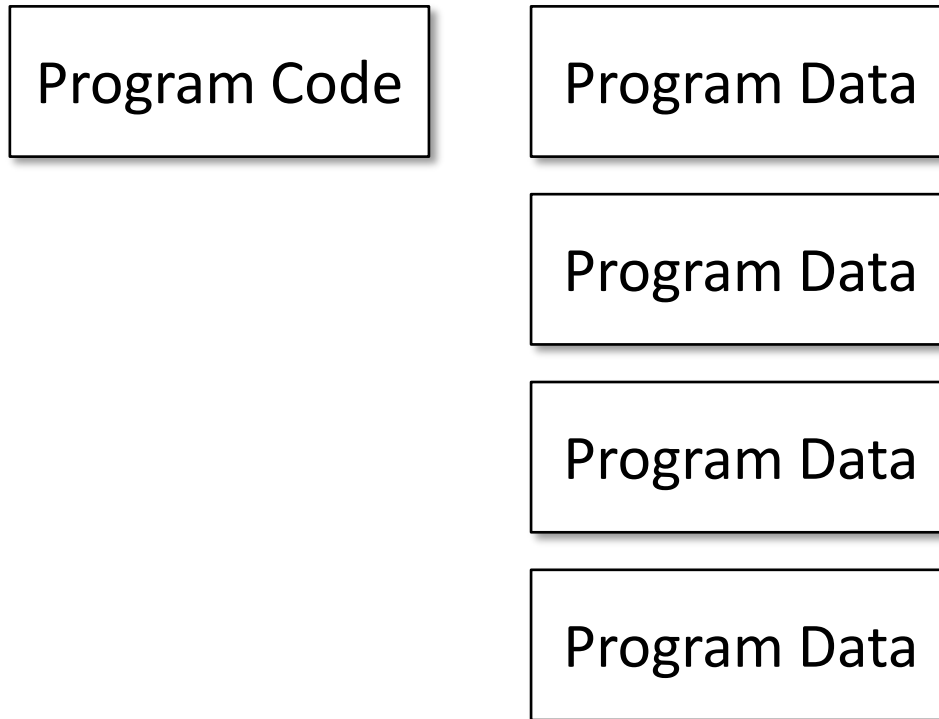
Process Memory

- Program Data
 - the variables which hold the entities' states
- Program Code
 - the instructions which say what to do with the program data

Program Data & Program Code

- How many times can a program be run?
- How many copies of the program can be running at once?
- How many copies of the program data are needed?
- How many copies of the program code are needed?

Program Data & Program Code



Terminology

- *Program* = program code
- *Process* = execution of a program
- Each process has:
 - a program to execute
 - all of the program data for that execution

Program Data

- Contains many segments
- Different for each operating system
 - Linux/Mac OS
 - Windows
- Some segments appear in most OSs

Program Data



Global Variables

- All variables global to the program are stored here.
- Once created, they are never destroyed

How do we create a global variable in C?

Stack

- All variables local to functions are stored here.
- Last-in first-out (LIFO)
- Once the function returns, these variables are destroyed

How do we create a variable on the stack?

Creating a variable on the stack is called *static memory allocation* and all such variables are called *automatic variables*.

Heap

- Reserved for *dynamic memory management*
- The programmer must explicitly create and destroy these variables

How do we create a variable on the heap?

Where is the stack?

Where is the heap?

What happens if they meet?

What might cause them to meet?

Pointers

Pointer Gotchas

- Two parts to think about
 - the *value* of a variable
 - the *address* of a variable
- Each variable has both (even pointers themselves)!
- The value of a pointer is another variable's address

Example

```
main()  
{  
    int x = 100;  
  
    printf("The value of x is %d\n", x);  
    printf("The address of x is %u\n", &x);  
}
```

Declaring a Pointer

```
main()  
{  
    char *charPointer;  
    short *shortPointer;  
    int *intPointer;  
    long *longPointer;  
    long long *longLongPointer;  
    float *floatPointer;  
    double *doublePointer;  
    unsigned char *uCharPointer;  
    unsigned short *uShortPointer;  
    unsigned int *uIntPointer;  
    unsigned long *uLongPointer;  
    unsigned long long *uLongLongPointer;  
}
```

What type is charPointer?

```
char *charPointer;
```


What can its contents be?

```
char *charPointer;
```

Where would it be allocated?

```
char *charPointer;
```

Example

```
main()  
{  
    int x = 100;  
    int *y = &x;  
  
    printf("The value of x is %d\n", x);  
    printf("The address of x is %u\n", y);  
}
```

- How do we make use of *y*?

Dereferencing Pointers

```
main()
{
    int x = 100;
    int *y = &x;

    printf(" x = %d\n", x);
    printf("*y = %d\n", *y);

    x = x + 1;

    printf(" x = %d\n", x);
    printf("*y = %d\n", *y);

    *y = *y + 5;

    printf(" x = %d\n", x);
    printf("*y = %d\n", *y);
}
```

- x is an integer
- y is a pointer to an integer
- x is initialized to 100
- y is initialized to the address of x
- the place in memory called “x” can be accessed in two ways
 - by using the variable name “x”
 - by dereferencing the variable “y”
- *y can be used to read/write from where y points to

What is the output?

```
main()  
{  
    int *x;  
  
    printf("%d\n", *x);  
}
```

What is the output?

```
main()  
{  
    int *x = 0;  
  
    printf("%d\n", *x);  
}
```

What is the output?

```
void function1()  
{  
    int x = 100;  
    int *y = &x;  
  
    printf("*y = %u\n", y);  
}  
  
void function2()  
{  
    function1();  
}  
  
main()  
{  
    function1();  
    function1();  
    function2();  
}
```

Pointer Arithmetic

```
main()
{
    int x = 100;
    int *y = &x;

    printf("x = %d\n", x);
    x++;
    printf("x = %d\n", x);
    x++;
    printf("x = %d\n", x);
    x++;
    printf("x = %d\n", x);

    printf("y = %u\n", y);
    y++;
    printf("y = %u\n", y);
    y++;
    printf("y = %u\n", y);
    y++;
    printf("y = %u\n", y);
}
```

Output:

```
x = 100
x = 101
x = 102
x = 103
y = 3219634196
y = 3219634200
y = 3219634204
y = 3219634208
```

- Why does `y` go up by 4?

Pointer Arithmetic

- Adding n to a pointer makes it point n spots “to the right”
- Subtracting n makes it point n “to the left”
- Using `++` makes it point one “to the right”
- Using `--` makes it point one “to the left”

Recall the equation from yesterday's exercise for calculating the address of an array element given its index.

Given a pointer to the first element, use pointer arithmetic to obtain a pointer to the i^{th} element.

Exercise

```
main()  
{  
    int array[100];  
    int *start = &array[0];  
    int *element;  
    int i;  
  
    for (i = 0; i < 100; i++)  
    {  
        element = ?  
    }  
}
```

Consider this code

```
main()  
{  
    int array[100];  
    int *element;  
    int i;  
  
    for (i = 0; i < 100; i++)  
    {  
        element = array + i;  
        ...  
    }  
}
```

Pointers and Arrays

- In C, arrays are very closely related to pointers
- In fact, these two statements do the exact same thing:

```
array[5] = 20;  
*(array + 5) = 20;
```

- Both could be written more explicitly as:

```
int *elem = &array[0] + 5;  
*elem = 20;
```

What type of variable is `s`?

```
char *s;
```

Possible answers

- Pointer to a character
- An array of characters
- A string
 - Most advanced C programmers would think of this answer first (or even call it a C string)

Consider the following code

```
char *createName(char *first, char *middle, char *last)
{
    char name[100];
    name[0] = '\0';
    strcat(name, first);
    strcat(name, " ");
    strcat(name, middle);
    strcat(name, " ");
    strcat(name, last);

    return name;
}

main()
{
    char *name = createName("Alfredo", "H.", "Pasqualie");
}
```


Dynamic Memory Allocation

- To add a variable to the heap, we need to manually *allocate* the space.
- To remove a variable from the heap, we need to manually *free* up that space.

malloc and free (Example)

```
main()  
{  
    int *intPointer;  
  
    intPointer = (int *)malloc( sizeof(int) );  
    *intPointer = 20;  
  
    free(intPointer);  
}
```

malloc and free

```
void *malloc(int nbytes); /* must cast result */
```

```
void free(void *ptr);
```

Contrast

```
char *createName(char *first,
                 char *middle, char *last)
{
    char name[100];
    name[0] = '\0';
    strcat(name, first);
    strcat(name, " ");
    strcat(name, middle);
    strcat(name, " ");
    strcat(name, last);

    return name;
}
```

```
char *createName(char *first,
                 char *middle, char *last)
{
    int size = strlen(first)
               + strlen(middle)
               + strlen(last) + 3;

    char *name =
        (char *) malloc(size);
    *name = '\0';

    strcat(name, first);
    strcat(name, " ");
    strcat(name, middle);
    strcat(name, " ");
    strcat(name, last);

    return name;
}
```

Pass by Reference

- Using pointers, we have the ability to access pretty much any memory location.
- Most parameters are passed by value in C
- How would we pass by reference?

Example

```
void swap(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}

main()
{
    int x = 100;
    int y = 200;

    printf("x = %d\n", x);
    printf("y = %d\n", y);

    swap(&x, &y);

    printf("x = %d\n", x);
    printf("y = %d\n", y);
}
```

Double Indirection

- What does this mean?

```
char **stringList;
```

Lecture 04 Summary

- Process Memory
- Pointers
 - Declaring
 - Dereferencing
 - Pointer Arithmetic
- Dynamic Memory Allocation
- Passing Parameters by Reference

Next Class

- Abstract Data Types