Lectures and Assignments

Compilation Pipeline

Lecture Notes

Lecture 1: Compilation Pipeline (PDF - 1.0MB)

Lab Exercises

The primary goal of this lab period is to get your C compiler up and running.

We have 2 “Hello, World!” examples for you to practice compiling to make sure that everything is working.

Lab 1 files (ZIP) (This ZIP file contains: 3 .c files and 1 .h file.)

Compile `hello1` with:

```
gcc hello.c -o hello1
```

Compile `hello2` with:

```
gcc main.c hello.c -o hello2
```

Assignment 1

Setup

Assignment 1 files (ZIP) (This ZIP file contains: 3 .c files and 2 .h files.)

The zip contains 3 C files:

1. fibeverse.c
2. fibonacci.c
3. reverse.c

And 2 header files (.h):

1. fibonacci.h
2. reverse.h

You can compile them with this command (though it won't work at first; see Problem 1):

```
gcc -Wall -std=c99 _fibeverse.c reverse.c fibonacci.c_ -o **fibeverse**
```

You can run the resulting program with two arguments: a number, then a string (in quotes):

```
./fibeverse 6 'what a trip that was!'  8  was! that trip a what
```

The first line it prints is the 6th fibonacci number. The second line is the string you provided, with the words reversed.

Problem 1

Unfortunately, the code doesn't compile as-is! Fix the compile errors and warnings. `gcc` should produce no output with the above command when you are done.

Problem 2

I can't decide whether I want a program that computes Fibonacci numbers or a program that reverses strings! Let's modify fibeverse so that it can be compiled into either.

Use the preprocessor macros we taught in class to make it so that I can choose which program it is at compile time.

When I compile it with this command, it should compute the Fibonacci number but not reverse the second argument:

```
gcc -Wall -std=c99 **-DFIBONACCI** fibeverse.c reverse.c fibonacci.c -o
**fibonacci**
```

Then I can run it like this:

```
./fibonacci 8
```

When I use this command, it should reverse the string I provide as the first argument, and not do any fibonacci calculation:

```
gcc -Wall -std=c99 **-DREVERSE** fibeverse.c reverse.c fibonacci.c -o **reverse**
```

Then I can run it like this:

```
./reverse 'a brave new world'
```

It should work as it originally did when I provide both compiler flags:

```
gcc -Wall -std=c99 **-DFIBONACCI -DREVERSE** fibeverse.c reverse.c fibonacci.c -o
**fibeverse**
```

Solutions

Assignment 1 solution (PDF)

6.S096 | January IAP 2013 | Undergraduate

Introduction To C And C++

Lectures and Assignments

Core C: Control Structures, Variables, Scope, and Uninitialized Memory

Lecture Notes

Lecture 2: Core C (PDF)

Lab Exercises

The primary goal of this lab period is to make sure you know all of the basics of writing code in C and to avoid common pitfalls unique to C and C++.

Exercise 1

Starting with an empty .c file, write a C program to calculate the first 100 triangular numbers (0, 1, 3, 6, 10, 15, 21, …). You may check back at previous example code and slides if you get stuck, but take this an opportunity to try to recall, from memory, the standard boilerplate code you'll be writing often when starting a new C program, like the proper `#include` statements to be able to print to the console.

Exercise 2

Find and fix the variable shadowing bug in the following C program, which is intended to calculate the factorial function:

```
#include <stdio.h>

int factorial (int n) {
        int i = 1;
        while (n > 1) {
                i = i * n;
                int n = n - 1;
        }
        return i;
}

int main (int argc, char *argv[]) {
        int fac4 = factorial(4);
        int fac5 = factorial(5);
        printf("4! = %d, 5! = %d\n", fac4, fac5);
        return 0;
}
```

Assignment 2

Problem 1

Rewrite the program below, replacing the `for` loop with a combination of `goto` and `if` statements. The output of the program should stay the same. That is, the program should print out the arguments passed to it on the command line.

```
#include <stdio.h>

int main(int argc, char ** argv){
    for (int i = 1; i < argc; i++) {
        printf("%s\n", argv[i]);
    }
    return 0;
}
```

If you ran the program like this: `./prog one two three`, it would print

```
one
two
three
```

Problem 2

In lecture, we covered a variety of control structures for looping, such as `for`, `while`, `do/while`, `goto`, and several for testing conditions, such as `if` and `switch`.

Your task is to find seven different ways to print the odd numbers between 0 and 10 on a single line. You should give them names like `amaze1`, `amaze2`, etc. Here's a bonus amaze (that you can

use as one of your seven if you like):

```
void amazeWOW() {
        int i;
        printf("amazeWOW:\t");
        for (i = 0; i <= 10; i++) {
                if (i % 2 == 1) {
                        printf("%d ", i);
                }
        }
        printf("\n");
}
```

You may need to get a little creative!

Put all of your functions in the same C file and call them in order from the `main()` function. We should be able to compile your program with this command and see no errors:

```
gcc -Wall -std=c99 -o amaze amaze.c
```

Solutions

Solutions are not available for this assignment.

Lectures and Assignments

C Memory Management

Topics: Computer memory layout (heap, stack, call stack), pointers and addresses, arrays, strings, and manual memory allocation/deallocation.

Lecture Notes

Lecture 3: C Memory Management (PDF)

Lab Exercises

The primary goal of this lab period is to introduce pointers, addresses, arrays, and manual memory management.

Exercise 1

```
#include <stdio.h>

void square(int num) {
        num = num * num;
}

int main() {
        int x = 4;
        square(x);
        printf("%d\n", x);
        return 0;
}
```

Take a look at the above code. In lecture, I pointed out that function variables are passed by value, not reference. So this program will currently print out 4. Compile the code to confirm this.

Use pointers and addresses to modify the code so x is passed by reference instead and is squared. This will involve changes to the square function that *does not* involve changing void to

int and giving square a return statement. Make sure your code compiles with `-Wall` flag without warnings.

Exercise 2

While coding up this exercise, listening to Hakuna Matata, I was so worry-free I forgot how to use C!

Fix the following code so that it creates a string `str` and copies "`hakuna matata!`" into it.

```
#include <stdio.h>

void main() {
        char str[];
        ???(, "hakuna matata!"); // this line should copy "hakuna matata!"
                                 // into our char array
        printf("%s\n", str);
        // Anything else?
}
```

After confirming your fix works, change the code to use heap memory instead of the stack. Remember, everything you `malloc` you must also `free`!

Assignment 3

Problem 1

sort (C)

In sort.c, I've implemented a basic implementation of insertion sort (not too efficient, but a very simple sorting algorithm). Look at and understand the code (read comments), and put the proper argument data type for the sort function's first argument. Compile and run the code to make sure it works (it sorts the numbers).

Now, replace all array index access (places where I access the array with `[]`, such as in `A[i]`) in the entire program by using pointer addition instead. Also, where I create the array (line 34, `int array[n]`;), replace that with an array creation using malloc (of same size).

Hint: Since we're using malloc, we must also make another change!

Make sure your program compiles and runs correctly (numbers are sorted).

Problem 2

resize (C)

The purpose of resize.c is to create an initial array of a user-specified size, then dynamically resize the array to a new user-specified size. I've given a shell of the code, including code to get user-specified sizes as `int`s.

However, the code is missing a few things. You must manage the memory for the array! Look at the comments in the code that describe what should be done and fill in blanks. Make sure the program compiles and runs as expected.

Solutions

Assignment 3 solution (PDF)

Lectures and Assignments

Data Structures, Debugging

Topics: Using structs, unions, typedef, and enums, and how to debug with Valgrind and GDB.

Lecture Notes

Lecture 4: Data Structures, Debugging (PDF)

Lab Exercises

The primary goal of this lab period is to introduce debugging tools, and use of unions/structs.

Exercise 1

Download and install Valgrind on your system, if it's not already. To test if you have Valgrind, run `valgrind --version`. It should print the version of Valgrind that is installed.

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

void fn()
{
    int* x = malloc(10 * sizeof(int));
    printf("%d",*x);
    x[10] = 0;
}

int main()
{
    fn();
    return 0;
}
```

There are 3 sources of memory errors in this code. Run valgrind to determine what they are (although I suspect you can probably tell from the code anyways).

Exercise 2

Use a union to print the individual bytes of an `int`. (Hint: Recall the size of `int`s and other data types.)

Exercise 3

Determine how much memory is required for each of the `struct`s below. How much of that memory is padding between the members?

```
struct X
{
    short s;
    int i;
    char c;
};

struct Y
{
    int i;
    char c;
    short s;
};

struct Z
{
    int   i;
    short s;
    char  c;
};
```

Assignment 4

Today's assignment combines the material from the past few lectures. Your job is to fill in the skeleton code we provide. I have commented the code with what each section should do.

Assignment 4 files (ZIP) (This ZIP file conatins: 2 .c files and 1 .h file.)

You can learn more about binary search trees and find pseudo-code on the binary search tree page on Wikipedia.

Your job is to implement a binary search tree, a data structure of connected nodes with a tree shape. Each node has a node identifier (a number), data (payload), and 2 children (left and right). The children are other nodes referenced with a pointer, with the constraint that the left node's ID is less than the parent node's ID, and the right node's ID is larger than the parent node ID. No two nodes will have the same identifier. A node can have less than two children; in that case, one or more of its child pointers can be `NULL`.

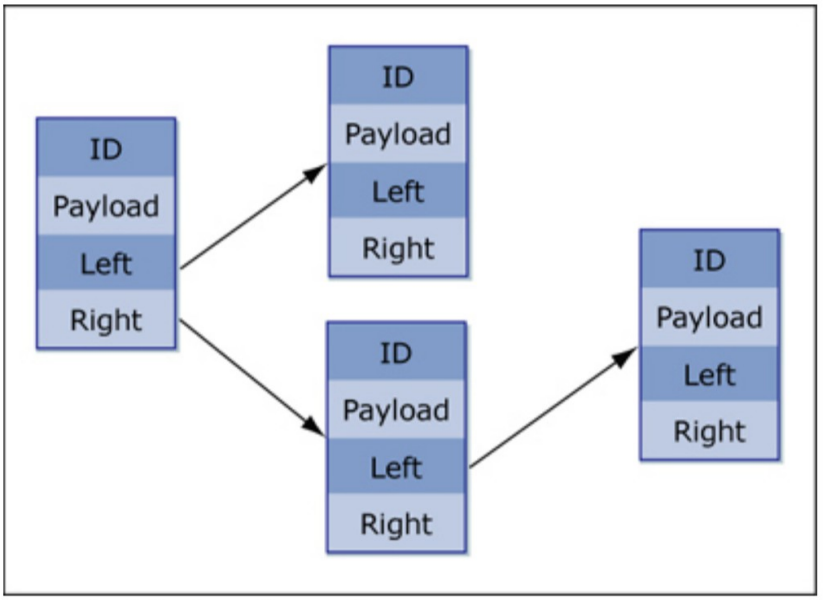

Image by MIT OpenCourseWare.

`user.c` contains the `main()` function. We will replace this function with one for grading. You should use your `main()` function to test that your functions to insert into and search the binary tree work correctly.

This is a C/C++ course, not an algorithms course, but if you want a challenge, try implementing node deletion as well!

Your job is to complete the data structure and function declarations in `bintree.h`, then complete the implementation of your functions in `bintree.c`. If you want to define additional functions to simplify your program, that's fine. You cannot change the return types or argument types of the included functions, though. Even though we don't require the deletion function, make sure to free all memory you allocate!

Make sure your program compiles without warning, runs, and *definitely* use `valgrind` to ensure you have no memory leaks.

```
$ gcc -Wall -std=c99 user.c bintree.c -o bintree
$ ./bintree
<your test output>
```

Solutions

Solutions are not available for this assignment.

Lectures and Assignments

C++ Introduction, Classes, and Templates

Lecture Notes

Lecture 5: C++ Introduction, Classes, and Templates (PDF)

Lab Exercises

There were no lab exercises to accompany this lecture.

Assignment 5

Problem 1

Now that we've transitioned from learning C to learning C++, we should be able to transition some C-style code that uses `struct`, `typedef`, and ordinary functions into C++ code that uses a single class to do the same job.

Download the C code below, and create a new file, `p1_grades.cpp`.

p1_grades (C)

Your job is to create a C++ class named `Grade` that has the same functionality as the old `struct Grade` and associated functions in the original C file. When you are finished, your C++ file should only have one function definition outside of the class: `main()`. You should use the following definition of main:

```
int main() {
        Grade g;
        int percent;

        printf("Enter two grades separated by a space. Use a percentage for
the first and letter for the second: ");
        scanf("%d", &percent;);
        scanf("\n");

        g.setByPercent(percent);
        g.print();

        g.setByLetter(getchar());
        g.print();

        return 0;
}
```

The names and interface of your Grade class should match the way the Grade instance is being used in the main function above. (That is, it should have member functions named `setByPercent`, etc. that are compatible with the use of those functions in `main`.)

Note the general way the grade program works: The user runs the program and is asked to enter two pieces of input. The first is an integer between 1 to 100 representing a percentage grade. The second input, separated on the command line by a space, is a letter grade (A, B, C, D, F). The output is two lines; each line shows the original and converted forms of the grade. So, for example, entering the input "`100 F`" would generate the lines:

```
Grade: 100: A
Grade: 45: F
```

The two input grades aren't related (a 100 isn't an F!). Instead, the inputs are used to show both directions of the conversion. Letter grades are converted to "nearby" percentages that fall into the right range. Don't worry about changing the grading logic. You can use the existing scale / system, including reusing `GRADE_MAP`.

Make sure your program compiles without warning, runs, and *definitely* use `valgrind` to ensure you have no memory leaks.

```
$ g++ -Wall p1_grades.cpp -o p1_grades
$ ./p1_grades
< provide a [percentageGrade] [letterGrade] input pair (like "97 D") >
<your test output>
```

Problem 2

In this problem, you will be converting a class that is specialized for integers into a templated class that can handle many types, including integers and structs. You will create a templated class named `List` that correctly initializes, manages, and de-allocates an array of a specified length. This is a nice class because the normal C arrays we've seen do not keep track of their length at runtime, but the class we're building will do that for us!

p2_templates (CPP)

When you're finished writing your templated `List` class, you should change your `main()` function to this code below (this is the same code that's in the starter file, p2_templates.cpp):

```
int main() {
    List<int> integers(10);
    for (int i = 0; i < integers.length; i++) {
        integers.set(i, i * 100);
        printf("%d ", integers.get(i));
    }
    printf("\n"); // this loop should print: 0 100 200 300 400 500 600 700 800 9

    List<Point *> points(5);
    for (int i = 0; i < points.length; i++) {
        Point *p = new Point;
        p->x = i * 10;
        p->y = i * 100;
        points.set(i, p);
        printf("(%d, %d) ", points.get(i)->x, points.get(i)->y);
        delete p;
    }
    printf("\n"); // this loop should print: (0, 0) (10, 100) (20, 200) (30, 300
}
```

This main function makes use of a typedef struct called Point. Here's its definition:

```
typedef struct Point_ {
    int x;
    int y;
} Point;
```

When run, your `main()` function should use the templated `List` class you've written yourself to produce this output:

```
0 100 200 300 400 500 600 700 800 900
(0, 0) (10, 100) (20, 200) (30, 300) (40, 400)
```

Naturally, this output should be generated by accessing the members of your templated `List` class, not by a hard-coded print statement.

To get you started, we've written a non-templated `IntList` class that just handles lists of integers:

```
class IntList {
        int * list;

public:
        int length;

        IntList(int len) {
                list = new int[len];
                length = len;
        }

        ~IntList() {
                delete[] list;
        }

        int get(int index) {
                return list[index];
        }

        void set(int index, int val) {
                list[index] = val;
        }
};
```

You should use this class as a model for your own, templated `List` class, but you won't need IntList at all in the final code that you turn in, because you will have replaced it with your templated List class.

Make sure your program compiles without warning, runs, and *definitely* use `valgrind` to ensure you have no memory leaks.

```
$ g++ -Wall p2_templates.cpp -o p2_templates
$ ./p2_templates
<your test output>
```

Solutions

Solutions are not available for this assignment.

Lectures and Assignments

C++ Inheritance

Lecture Notes

Lecture 6: C++ Inheritance (PDF)

Lab Exercises

Take a look at this example code:

```
#include <stdio.h>

class Shape {
public:
        virtual ~Shape();
        virtual void draw() = 0;
};

class Circle : public Shape {
public:
        virtual ~Circle();
        virtual void draw();
};

Shape::~Shape() {
        printf("shape destructor\n");
}

// void Shape::draw() {
//      printf("Shape::draw\n");
// }

Circle::~Circle() {
        printf("circle destructor\n");
}
```

```
void Circle::draw() {
        printf("Circle::draw\n");
}

int main() {
        Shape *shape = new Circle;
        shape->draw();
        delete shape;

        return 0;
}
```

Put it in a file named `lab6.cpp` and then compile it like this:

```
$ g++ -Wall lab6.cpp -o lab6
$ ./lab6
Circle::draw
circle destructor
shape destructor
```

Verify your understanding of how the `virtual` keyword and method overriding work by performing a few experiments:

1. Remove the `virtual` keyword from each location individually, recompiling and running each time to see how the output changes. Can you predict what will and will not work?
2. Try making `Shape::draw` non-pure by removing `= 0` from its declaration.
3. Try changing `shape` (in `main()`) from a pointer to a stack-allocated variable.

Assignment 6

rps (CPP)

In the file rps.cpp, implement a class called `Tool`. It should have an `int` field called strength and a `char` field called `type`. You may make them either private or protected. The `Tool` class should also contain the function `void setStrength(int)`, which sets the strength for the `Tool`.

Create 3 more classes called `Rock`, `Paper`, and `Scissors`, which inherit from `Tool`. Each of these classes will need a constructor which will take in an `int` that is used to initialize the `strength` field. The constructor should also initialize the `type` field using `'r'` for `Rock`, `'p'` for `Paper`, and `'s'` for `Scissors`.

These classes will also need a public function `bool fight(Tool)` that compares their strengths in the following way:

- Rock's strength is doubled (temporarily) when fighting scissors, but halved (temporarily) when fighting paper.

- In the same way, paper has the advantage against rock, and scissors against paper.
- The `strength` field shouldn't change in the function, which returns `true` if the original class wins in strength and `false` otherwise.

You may also include any extra auxiliary functions and/or fields in any of these classes. Run the program without changing the main function, and verify that the results are correct.

```
$ g++ -Wall rps.cpp -o rps
$ ./rps
<your test output>
```

Solutions

Solutions are not available for this assignment.