Structured Program Development in C

CS 2060 Week 2

Prof. Jonathan Ventura

Goto Considered Harmful

```c
#include <stdio.h>

int main()
{
  int i = 0;

LOOP:
  printf("%d\n",i);
  if ( i < 10 )
  {
    i = i+1;
    goto LOOP;
  }
}
```

- Early programming languages relied extensively on "goto," otherwise known as "unconditional jump."
- "goto" exists in C – you can jump directly to a label defined anywhere inside a function.

Goto Considered Harmful

```c
#include <stdio.h>

int main()
{
  int i = 0;

LOOP:
  printf("%d\n",i);
  if ( i < 10 )
  {
    i = i+1;
    goto LOOP;
  }
}
```

- Why might "goto" be a bad idea?

Goto Considered Harmful

```c
#include <stdio.h>

int main()
{
  int i;

  goto LOOP;
  i = 0;

LOOP:
  printf("%d\n",i);
  if ( i < 10 )
  {
    i = i+1;
    goto LOOP;
  }
}
```

- What does this program do?

Goto Considered Harmful

```c
#include <stdio.h>

int main()
{
  int i;

  goto LOOP;
  i = 0;

LOOP:
  printf("%d\n",i);
  if ( i < 10 )
  {
    i = i+1;
    goto LOOP;
  }
}
```

- What does this program do?
- Nobody knows!

  (i is not explicitly initialized.)

Goto Considered Harmful

```c
#include <stdio.h>

int main()
{
  int i;

  goto LOOP;
  i = 0;

LOOP:
  printf("%d\n",i);
  if ( i < 10 )
  {
    i = i+1;
    goto LOOP;
  }
}
```

- goto makes the flow of the program hard to understand.
- The program can jump anywhere at any time – we might violate reasonable assumptions about the program state.
- We don't know what values the program's variables are supposed to have when we reach the LOOP label.

Goto Considered Harmful

Go To Considered Harmful

- In 1968, Edgar Dijkstra wrote an influential note entitled "Go To Statement Considered Harmful."
  - This is the origin of the "considered harmful" meme you may see programmers use...

Goto Considered Harmful

Go To Considered Harmful

- In 1968, Edgar Dijkstra wrote an influential note entitled "Go To Statement Considered Harmful."
  - This is the origin of the "considered harmful" meme you may see programmers use...
- He wrote that "...the quality of programmers is a decreasing function of the density of **go to** statements they produce."

Goto Considered Harmful

Go To Considered Harmful

- In 1968, Edgar Dijkstra wrote an influential note entitled "Go To Statement Considered Harmful."
  - This is the origin of the "considered harmful" meme you may see programmers use...
- He wrote that "...the quality of programmers is a decreasing function of the density of **go to** statements they produce."
- Earlier Böhm and Jacopini (1966) had proved that programs did not need `goto`.
  - More precisely, a language can be Turing complete without having a `goto` statement.

Goto Considered Harmful

Go To Considered Harmful

- In 1968, Edgar Dijkstra wrote an influential note entitled "Go To Statement Considered Harmful."
  - This is the origin of the "considered harmful" meme you may see programmers use...
- He wrote that "...the quality of programmers is a decreasing function of the density of **go to** statements they produce."
- Earlier Böhm and Jacopini (1966) had proved that programs did not need `goto`.
  - More precisely, a language can be Turing complete without having a `goto` statement.
- This led to an era of "goto elimination" and the rise of "structured programming," which the C language exemplifies.

Structured Programming in C

Structured programming in C consists of three types of **control structures**:

- Sequence structure

  Execute statements one after the other, in the order they are written.

Structured Programming in C

Structured programming in C consists of three types of **control structures**:

- Sequence structure

  Execute statements one after the other, in the order they are written.

- Selection structure
  - Select one of two (or more) paths based on a condition.

Structured Programming in C

Structured programming in C consists of three types of **control structures**:

- Sequence structure

  Execute statements one after the other, in the order they are written.

- Selection structure
  - Select one of two (or more) paths based on a condition.
  - `if`, `if...else`, `switch`

Structured Programming in C

Structured programming in C consists of three types of **control structures**:

- Sequence structure

  Execute statements one after the other, in the order they are written.

- Selection structure
  - Select one of two (or more) paths based on a condition.
  - `if`, `if...else`, `switch`
- Iteration structure
  - Loop until a stop condition is reached.

Structured Programming in C

Structured programming in C consists of three types of **control structures**:

- Sequence structure

  Execute statements one after the other, in the order they are written.

- Selection structure
  - Select one of two (or more) paths based on a condition.
  - `if`, `if...else`, `switch`
- Iteration structure
  - Loop until a stop condition is reached.
  - `for`, `while`, `do...while`

Selection with if...else

```c
#include <stdio.h>

int main()
{
  int a,b,c;
  scanf("%d %d %d",&a,&b,&c);

  if ( a > b )
  {
    if ( a > c )
    {
      printf("%d is the biggest\n",a);
    }
  }
  else
  {
    printf("%d is not the biggest\n",a);
  }
}
```

- if...else is used for selection.

Selection with if...else

```c
#include <stdio.h>

int main()
{
  int a,b,c;
  scanf("%d %d %d",&a,&b,&c);

  if ( a > b )
    if ( a > c )
      printf("%d is the biggest\n",a);
  else
    printf("%d is not the biggest\n",a);
}
```

- What will this program do?

Dangling else

```c
#include <stdio.h>

int main()
{
  int a,b,c;
  scanf("%d %d %d",&a,&b,&c);

  if ( a > b )
    if ( a > c )
      printf("%d is the biggest\n",a);
  else
    printf("%d is not the biggest\n",a);
}
```

- It is not clear from the syntax whether the `else` belongs to the first `if` or the second one.
- This is called a **dangling else**.
- In practice, C will associate the `else` with the nearest `if` (the second one).

C Conditional Operator

```c
#include <stdio.h>

int main()
{
  int a,b;
  scanf("%d %d",&a,&b);

  int max;
  int min;

  if ( a > b ) max = a; else max = b;
  if ( a < b ) min = a; else min = b;

  printf( "max is %d\n", max );
  printf( "min is %d\n", min );
}
```

- The conditional operator allows us to rewrite `if/else` structures in an abbreviated way.

C Conditional Operator

```c
#include <stdio.h>

int main()
{
  int a,b;
  scanf("%d %d",&a,&b);

  int max = ( a > b ) ? a : b;
  int min = ( a < b ) ? a : b;

  printf( "max is %d\n", max );
  printf( "min is %d\n", min );
}
```

- This code is identical to the previous, just shorter.

C Conditional Operator

- The conditional operator has three parts:

```
( condition ) ?  true case :  false case
```

- If `condition` is true, the true case expression will be returned.
- If `condition` is false, the false case expression will be returned.

C Conditional Operator

```c
#include <stdio.h>

int main()
{
  int a,b,c;
  scanf("%d %d %d",&a,&b,&c);

  int a_is_max = ( a > b ) ? ( a > c ) ? 1 : 0 : 0;

  ( a_is_max ) ? printf( "a is the max\n" ) : printf( "a is not the max\n" );
}
```

- Conditionals can be nested as well.

C Conditional Operator

```c
#include <stdio.h>

int main()
{
  int a,b,c;
  scanf("%d %d %d",&a,&b,&c);

  int a_is_max = ( a > b ) ? ( a > c ) ? 1 : 0 : 0;

  ( a_is_max ) ? printf( "a is the max\n" ) : printf( "a is not the max\n" );
}
```

- Conditionals can be used without making an assignment.

C Conditional Operator

```c
#include <stdio.h>

int main()
{
  int a,b,c;
  scanf("%d %d %d",&a,&b,&c);

  int a_is_max = ( a > b && a > c );

  ( a_is_max ) ? printf( "a is the max\n" ) : printf( "a is not the max\n" );
}
```

- In this case, it would be clearer to avoid the nested conditional.

The `while` Iteration Statement

```c
#include <stdio.h>

int main()
{
  int i = 0;
  while ( i < 10 )
  {
    printf("%d\n",i);
  }
}
```

- The `while` statement loops a code block while the condition is true.
- The condition is tested at the **start** of each iteration.

The `while` Iteration Statement

```c
#include <stdio.h>

int main()
{
  int i;
  scanf( "%d", &i );

  while ( i >= 10 ) i = i - 10;
  while ( i < 0 ) i = i + 10;

  printf( "%d\n", i );
}
```

- What does this program do?
- How could we replace the `while` statements with a single operation?

The `while` Iteration Statement

```c
#include <stdio.h>

int main()
{
  int i;
  scanf( "%d", &i );

  i = i % 10;

  printf( "%d\n", i );
}
```

- Use the mod (%) operator.

Counter-controlled iteration

```c
#include <stdio.h>

int main()
{
  int i = 1;
  int j = 2;
  while ( i < 10 )
  {
    j = j * 2;
    i = i + 1;
  }
  printf("%d %d\n",i,j);
}
```

- A `while` loop can be used for counter-controlled iteration.
- When the number of iterations is known, it is called **definite iteration**.

Counter-controlled iteration

```c
#include <stdio.h>

int main()
{
  int i = 1;
  int j = 2;
  while ( i < 10 )
  {
    j = j * 2;
    i = i + 1;
  }
  printf("%d %d\n",i,j);
}
```

- How many iterations (loops) will this code do?
- What will this program output?

Counter-controlled iteration

```c
#include <stdio.h>

int main()
{
  int i = 1;
  int j = 2;
  while ( i < 10 )
  {
    j = j * 2;
    i = i + 1;
  }
  printf("%d %d\n",i,j);
}
```

- Nine iterations (stops once `i == 10`)
- Outputs 1024
  - After each iteration, $j = 2^i$
  - $2^{10} = 1024$

Counter-controlled iteration

```c
#include <stdio.h>

int main()
{
  int i = 1;
  int j = 2;
  while ( i < 10 )
  {
    j = j * 2;
    i = i + 1;
  }
  printf("%d %d\n",i,j);
}
```

- Nine iterations (stops once `i == 10`)
- Outputs 1024
  - After each iteration, $j = 2^i$
  - $2^{10} = 1024$

Floating point numbers

- Values with a fractional part can be represented using a floating-point data type: `float`
  - `float` typically indicates a 32-bit (single precision) floating point data type
  - `double` is used for 64-bit (double precision) floating point data type
- The format specifier `%f` is used to read and print `floats` in `scanf` and `printf`.

Type conversion using the cast operator

```c
#include <stdio.h>

int main()
{
  int num = 12;
  int den = 15;

  float frac = num/den;
  printf("%f\n",frac);
}
```

- What will this program output?

Type conversion using the cast operator

```c
#include <stdio.h>

int main()
{
  int num = 12;
  int den = 15;

  float frac = num/den;
  printf("%f\n",frac);
}
```

- What will this program output?
- 0

Type conversion using the cast operator

```c
#include <stdio.h>

int main()
{
  int num = 12;
  int den = 15;

  float frac = num/den;
  printf("%f\n",frac);
}
```

- What will this program output?
- 0
- C will perform integer division on integers unless you explicitly convert (one of) the variables to floating point.

Type conversion using the cast operator

```c
#include <stdio.h>

int main()
{
  int num = 12;
  int den = 15;

  float frac = (float)num/den;
  printf("%f\n",frac);
}
```

- C provides a unary **cast operator** for explicit type conversion.
- To cast a variable to float we use `(float)`.
- This code will now output `0.8` as expected.

Type conversion using the cast operator

```c
#include <stdio.h>

int main()
{
  int num = 12;
  int den = 15;

  float frac = (float)num/den;
  printf("%f\n",frac);
}
```

- Here we are only explicitly converting `num` to `float`.
- C will only perform arithmetic on variables of the same type.

Type conversion using the cast operator

```c
#include <stdio.h>

int main()
{
  int num = 12;
  int den = 15;

  float frac = (float)num/den;
  printf("%f\n",frac);
}
```

- The compiler performs **implicit conversion** to make operands have the same type.
- Here, `den` will be implicitly converted to `float`.

Type conversion using the cast operator

```c
#include <stdio.h>

int main()
{
  int num = 12;
  int den = 15;

  float frac = (float)num/den;
  printf("%f\n",frac);
}
```

- Note that the cast operator does not change the type of the variable.
- It creates a temporary copy of the variable's value with the requested type.
- That copy is used in the arithmetic expression.

Formatting floating point numbers

```c
#include <stdio.h>

int main()
{
  int num = 13;
  int den = 15;

  float frac = (float)num/den;
  printf("%f\n",frac);    // 0.866667
  printf("%.2f\n",frac); // 0.87
  printf("%.5f\n",frac); // 0.86667
}
```

- We can specify the number of digits after the decimal point in `printf`.
- `"%.5f"` will print five digits after the decimal point.

Formatting floating point numbers

```c
#include <stdio.h>

int main()
{
  int num = 12;
  int den = 15;

  float frac = (float)num/den;
  printf("%f\n",frac);    // 0.800000
  printf("%.2f\n",frac); // 0.80
  printf("%.5f\n",frac); // 0.80000
}
```

- Another example with zeros.

- C has some special operators for abbreviating assignment expressions.
  - For example,
    `a+= 5`
    will add five to `a` and assign the result to `a`.
  - It is a shorthand way of writing
    `a = a + 5`
- This is another way in which C encourages shorter but less readable code...

Arithmetic assignment operators in C

Assignment operator	Sample expression	Explanation
+=	`c += 7`	`c = c + 7`
-=	`d -= 4`	`d = d - 4`
*=	`e *= 5`	`e = e * 5`
/=	`f /= 3`	`f = f / 3`
%=	`g %= 9`	`g = g % 9`

Table: Arithmetic assignment operators in C.

- C also provides special increment and decrement operators.
  - For example,
    a++ or ++a
    will add one to a.
- The difference is in what the expression evaluates to.
  - a++ will evaluate to a **before** the increment.
  - ++a will evaluate to a **after** the increment.

- These are called **prefix** or **postfix** operators.
  - ++a is a **preincrement** because the increment takes place **before** the value is returned.
  - a++ is a **postincrement** because the increment takes place **after** the value is returned.
- We can also write --a (**predecrement**) or a-- (**postdecrement**).

Increment example

```c
#include <stdio.h>

int main()
{
  int i = 1;
  int j = 2;
  while ( i < 10 )
  {
    i = i + 1;
    j = j * 2;
  }
  printf("%d %d\n", i, j ); // 10 1024
}
```

- Here is our example of computing $j = 2^i$ again.

Increment example

```c
#include <stdio.h>

int main()
{
  int i = 1;
  int j = 2;
  while ( i < 10 )
  {
    i++;
    j *= 2;
  }
  printf("%d %d\n", i, j ); // 10 1024
}
```

- We can use assignment and increment operators here for shorter code.

Type conversion using the cast operator

```c
#include <stdio.h>

int main()
{
  int i = 1;
  int j = 2;
  while ( i++ < 10 ) j *= 2;
  printf("%d %d\n", i, j );
}
```

- What will this output?

Type conversion using the cast operator

```c
#include <stdio.h>

int main()
{
  int i = 1;
  int j = 2;
  while ( ++i < 10 ) j *= 2;
  printf("%d %d\n", i, j );
}
```

- What will this output?

Type conversion using the cast operator

```c
#include <stdio.h>

int main()
{
  int i = 0;
  int j = 2;
  while ( ++i < 10 ) j *= 2;
  printf("%d %d\n", i, j );
}
```

- What will this output?

Part 1: Streaming Statistics

- In this assignment you will develop a program which
  - receives a stream of numbers, and
  - outputs a set of statistics about the data stream.

Part 1: Streaming Statistics

- In this assignment you will develop a program which
  - receives a stream of numbers, and
  - outputs a set of statistics about the data stream.
- "Internet of Things" (IoT) example: Internet-connected temperature sensor.
  - The sensor reads the room temperature every minute and posts the measurement to the cloud.
  - Your server app reads the measurements and computes running statistics to display to the user.

Part 1: Streaming Statistics

- In this assignment you will develop a program which
  - receives a stream of numbers, and
  - outputs a set of statistics about the data stream.
- "Internet of Things" (IoT) example: Internet-connected temperature sensor.
  - The sensor reads the room temperature every minute and posts the measurement to the cloud.
  - Your server app reads the measurements and computes running statistics to display to the user.
- **Note:** We do not want to store all the measurements ever recorded! It is too much data.
  - Instead, we want to optimally compute running statistics on the data stream.

Part 1: Streaming Statistics

- The input to the program will be positive integers.
  - When a negative one is received, the program should exit.
  - The program should first output a header line with a name for each column, separated by tabs.
    Separate the column names using a tab (`\t`)
- Each time a measurement is received, your program should compute and output (tab-separated):
  - The number of measurements seen so far
  - A running average of all measurements seen so far
  - The highest value seen
  - The second highest value seen
- **Rules:**
  - Work in integers for all data (don't use floating point).
  - Do not use arrays, sorting or swapping.

Part 1: Streaming Statistics

- **Example input:**

```
33 34 32 26 10 35 38 40 42 -1
```

- **Example output:**

```
Count Average Highest Second highest
1 33 33 0
2 33 34 33
3 33 34 33
4 31 34 33
5 27 34 33
6 28 35 34
7 29 38 35
8 31 40 38
9 32 42 40
```

Part 2: Prime tester

- Write a program that takes a positive integer as input and tests whether the number is prime or not.
  - Remember: a number is prime if it is only divisible by itself and one.
  - How do we test if `a` is divisible by `b`?

Part 2: Prime tester

- Write a program that takes a positive integer as input and tests whether the number is prime or not.
  - Remember: a number is prime if it is only divisible by itself and one.
  - How do we test if `a` is divisible by `b`?
  - `a` is divisible by `b` if `( a % b == 0 )`.

Part 2: Prime tester

- Write a program that takes a positive integer as input and tests whether the number is prime or not.
  - Remember: a number is prime if it is only divisible by itself and one.
  - How do we test if `a` is divisible by `b`?
  - `a` is divisible by `b` if `( a % b == 0 )`.
- If the number is not a prime, the program should output the smallest prime factor of the the number.

Part 2: Prime tester

- Write a program that takes a positive integer as input and tests whether the number is prime or not.
  - Remember: a number is prime if it is only divisible by itself and one.
  - How do we test if `a` is divisible by `b`?
  - `a` is divisible by `b` if `( a % b == 0 )`.
- If the number is not a prime, the program should output the smallest prime factor of the the number.
- If the number is prime, the program should output that the number is prime.

Part 2: Prime tester

- Write a program that takes a positive integer as input and tests whether the number is prime or not.
  - Remember: a number is prime if it is only divisible by itself and one.
  - How do we test if `a` is divisible by `b`?
  - `a` is divisible by `b` if `( a % b == 0 )`.
- If the number is not a prime, the program should output the smallest prime factor of the the number.
- If the number is prime, the program should output that the number is prime.
- **Rules:**
  - Use a `while` loop, not `for` or `do...while`.