CISC 105
Loops and Functions
June 20, 2005

Announcements

- Midterm next Monday
  - Cumulative, including today's lecture
- Project 1 handed out next Monday
  - Due July 11
- Email addresses for CPM during break
- No class July 4

Requesting Help

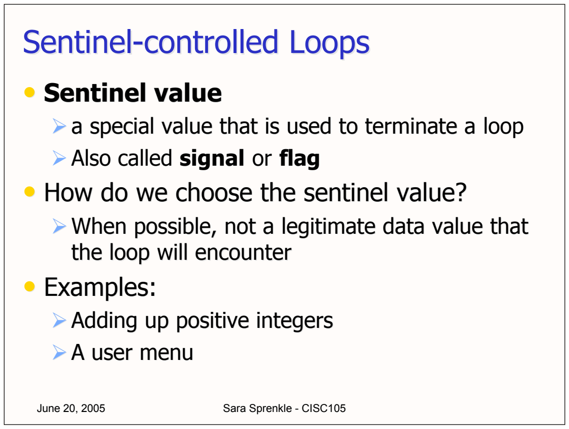
- Before you ask…
  - Try to understand the compiler error message
  - Add debugging statements
  - Trace through your program, looking for common problems, as discussed in class
- If you need help, email (Gang or me)
  - The program (either copy-paste or as an attachment, e.g. how you attach your tar file)
  - The error you're getting (either the compiler error or the erroneous output)

Review

- Constants
- Math Library
- User Input
- Conditionals
  - if, if-else, if-else-if, switch
- Loops
  - while, do-while, for

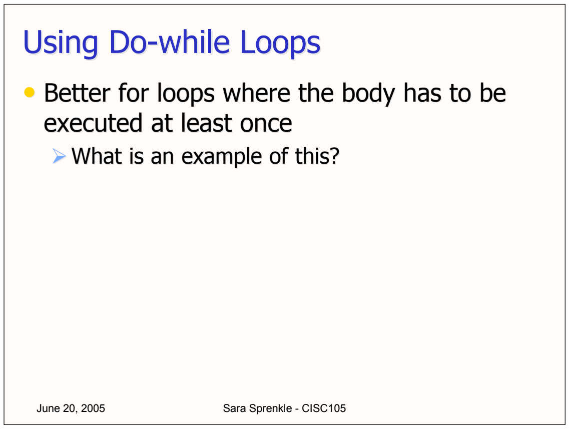
Quiz 2

Review Quiz



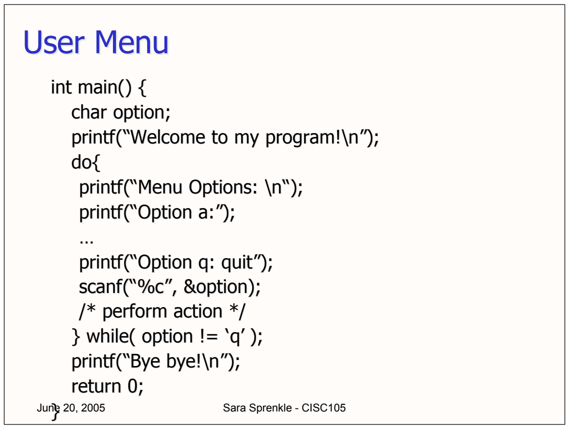
Sentinel-controlled Loops

- **Sentinel value**
  - a special value that is used to terminate a loop
  - Also called **signal** or **flag**
- How do we choose the sentinel value?
  - When possible, not a legitimate data value that the loop will encounter
- Examples:
  - Adding up positive integers
  - A user menu



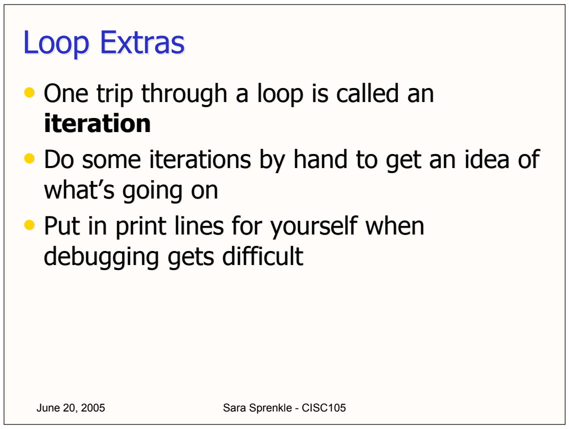
Using Do-while Loops

- Better for loops where the body has to be executed at least once
  - What is an example of this?



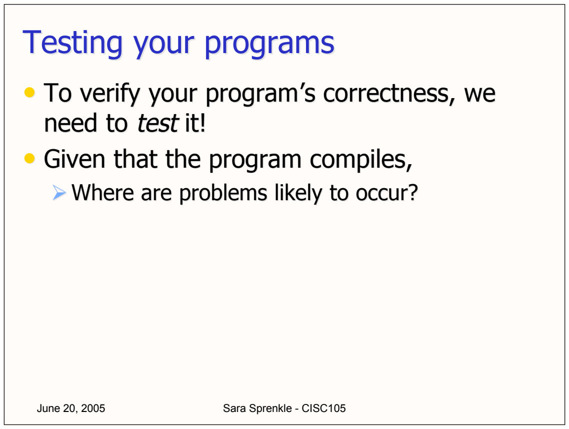
User Menu

```
int main() {
   char option;
   printf("Welcome to my program!\n");
   do{
    printf("Menu Options: \n");
    printf("Option a:");
    …
    printf("Option q: quit");
    scanf("%c", &option);
    /* perform action */
   } while( option != 'q' );
   printf("Bye bye!\n");
   return 0;
}
```



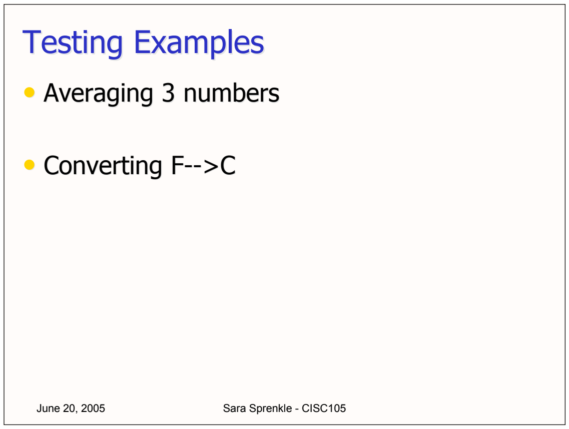
Loop Extras

- One trip through a loop is called an **iteration**
- Do some iterations by hand to get an idea of what's going on
- Put in print lines for yourself when debugging gets difficult



Testing your programs

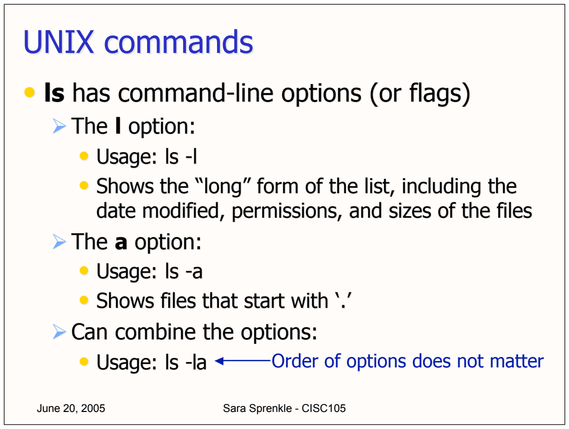
- To verify your program's correctness, we need to *test* it!
- Given that the program compiles,
  - Where are problems likely to occur?



Testing Examples

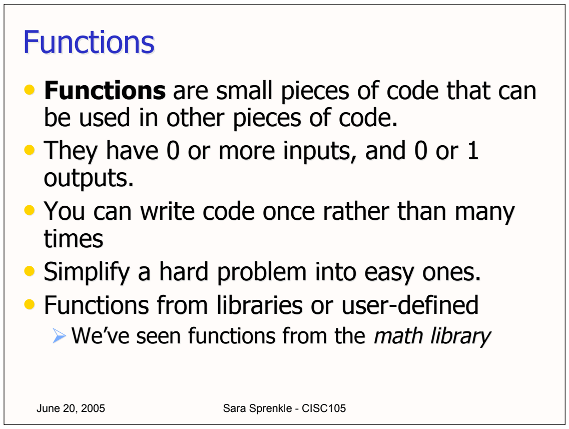
- Averaging 3 numbers

- Converting F-->C



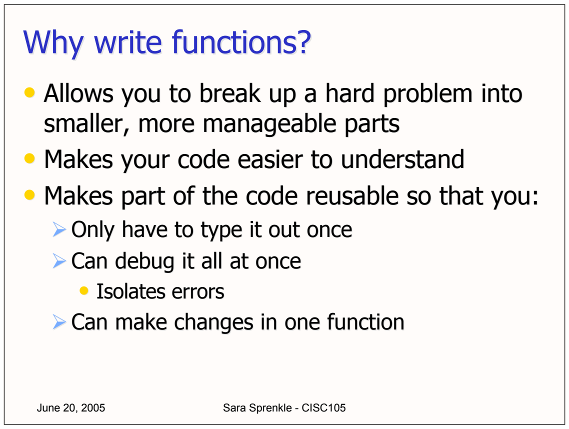
UNIX commands

- **ls** has command-line options (or flags)
  - The **l** option:
    - Usage: ls -l
    - Shows the "long" form of the list, including the date modified, permissions, and sizes of the files
  - The **a** option:
    - Usage: ls -a
    - Shows files that start with '.'
  - Can combine the options:
    - Usage: ls -la ← Order of options does not matter



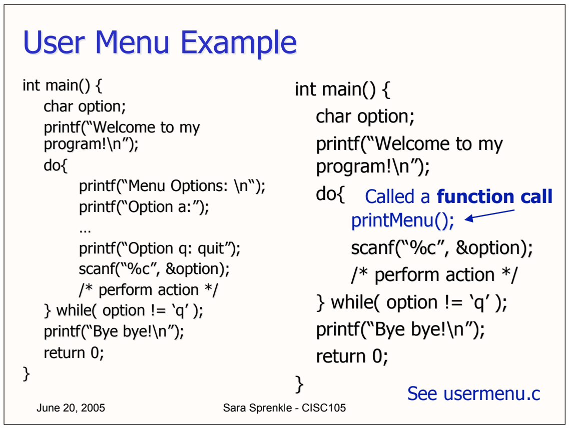
Functions

- **Functions** are small pieces of code that can be used in other pieces of code.
- They have 0 or more inputs, and 0 or 1 outputs.
- You can write code once rather than many times
- Simplify a hard problem into easy ones.
- Functions from libraries or user-defined
  - We've seen functions from the *math library*



Why write functions?

- Allows you to break up a hard problem into smaller, more manageable parts
- Makes your code easier to understand
- Makes part of the code reusable so that you:
  - Only have to type it out once
  - Can debug it all at once
    - Isolates errors
  - Can make changes in one function



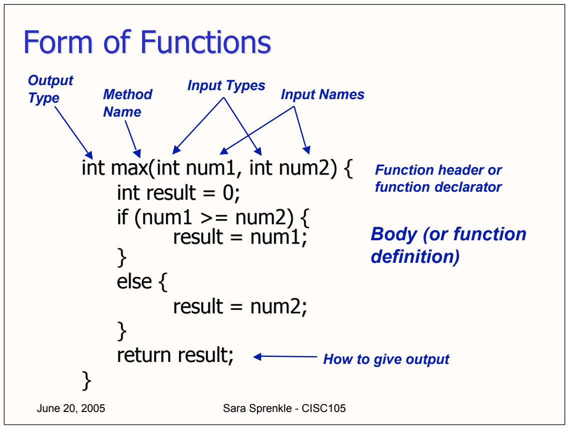
User Menu Example

```
int main() {
   char option;
   printf("Welcome to my
   program!\n");
   do{
        printf("Menu Options: \n");
        printf("Option a:");
        …
        printf("Option q: quit");
        scanf("%c", &option);
        /* perform action */
   } while( option != 'q' );
   printf("Bye bye!\n");
   return 0;
}
```

```
int main() {
   char option;
   printf("Welcome to my
   program!\n");
   do{
        printMenu();
        scanf("%c", &option);
        /* perform action */
   } while( option != 'q' );
   printf("Bye bye!\n");
   return 0;
}
```

Called a **function call**

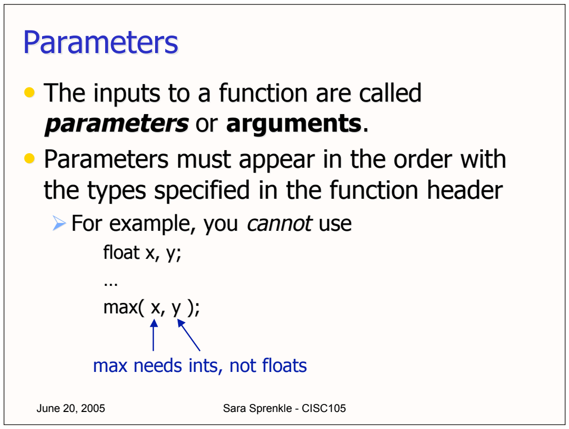
See usermenu.c



Form of Functions

```
int max(int num1, int num2) {
     int result = 0;
     if (num1 >= num2) {
          result = num1;
     }
     else {
          result = num2;
     }
     return result;
}
```

Output Type
Method Name
Input Types
Input Names
Function header or function declarator
Body (or function definition)
How to give output

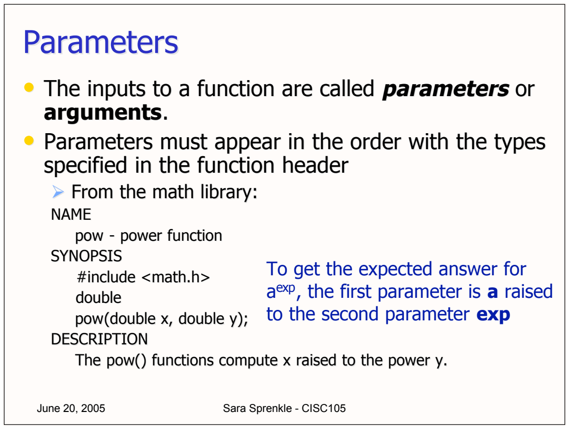


Parameters

- The inputs to a function are called ***parameters*** or **arguments**.
- Parameters must appear in the order with the types specified in the function header
  - For example, you *cannot* use

```
float x, y;
…
max( x, y );
```

max needs ints, not floats

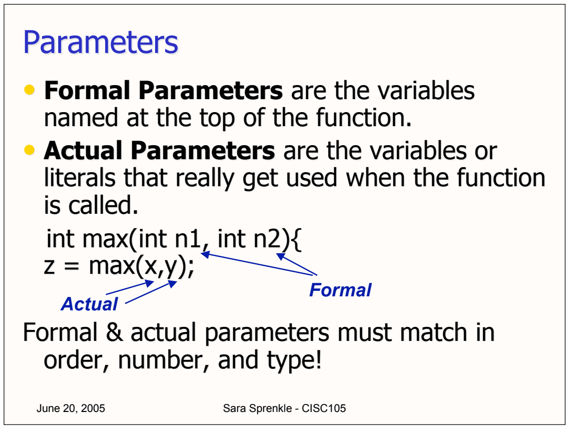


Parameters

- The inputs to a function are called ***parameters*** or **arguments**.
- Parameters must appear in the order with the types specified in the function header
  - From the math library:

```
NAME
    pow - power function
SYNOPSIS
     #include <math.h>
    double
    pow(double x, double y);
DESCRIPTION
    The pow() functions compute x raised to the power y.
```

To get the expected answer for a^{exp}, the first parameter is **a** raised to the second parameter **exp**



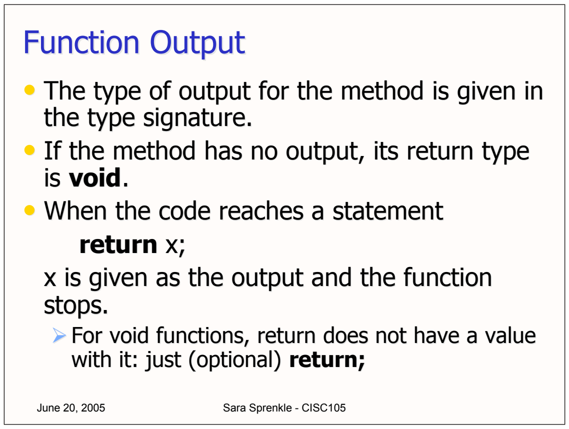
Parameters

- **Formal Parameters** are the variables named at the top of the function.
- **Actual Parameters** are the variables or literals that really get used when the function is called.

int max(int n1, int n2){
z = max(x,y);

Actual
Formal

Formal & actual parameters must match in order, number, and type!

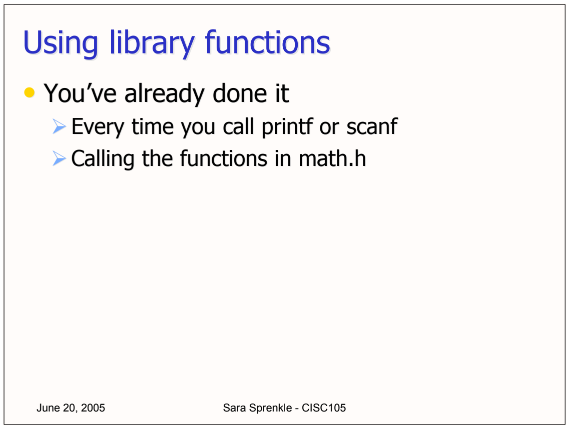


Function Output

- The type of output for the method is given in the type signature.
- If the method has no output, its return type is **void**.
- When the code reaches a statement

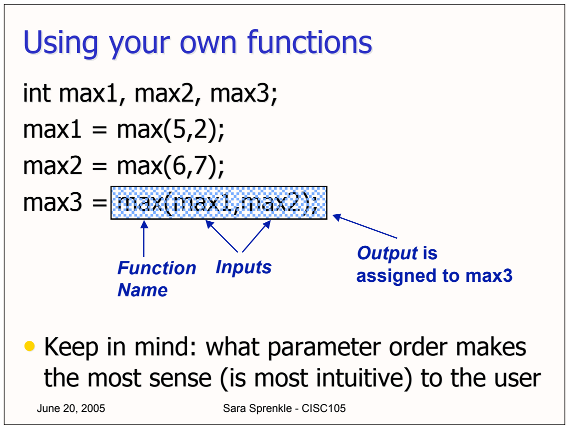
  return x;

  x is given as the output and the function stops.
  - For void functions, return does not have a value with it: just (optional) **return;**



Using library functions

- You've already done it
  - Every time you call printf or scanf
  - Calling the functions in math.h



Using your own functions

int max1, max2, max3;

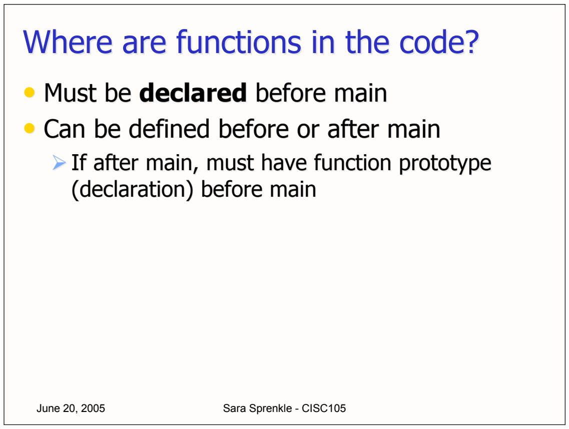
max1 = max(5,2);

max2 = max(6,7);

max3 = max(max1,max2);

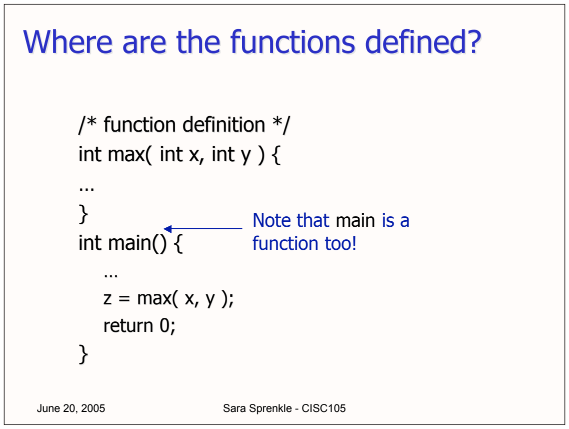
Function Name
Inputs
Output is assigned to max3

- Keep in mind: what parameter order makes the most sense (is most intuitive) to the user



Where are functions in the code?

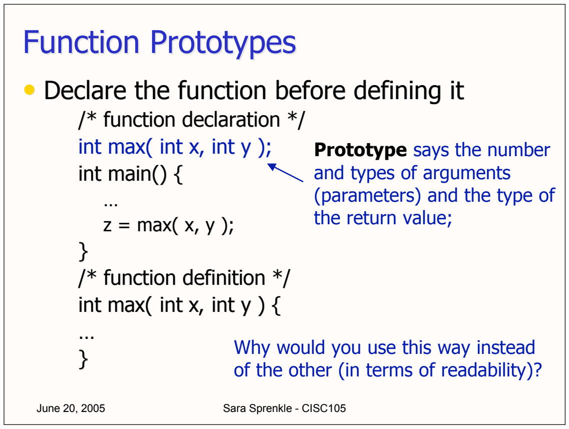
- Must be **declared** before main
- Can be defined before or after main
  - If after main, must have function prototype (declaration) before main



Where are the functions defined?

```
/* function definition */
int max( int x, int y ) {
…
}
int main() {
    …
    z = max( x, y );
    return 0;
}
```

Note that main is a function too!



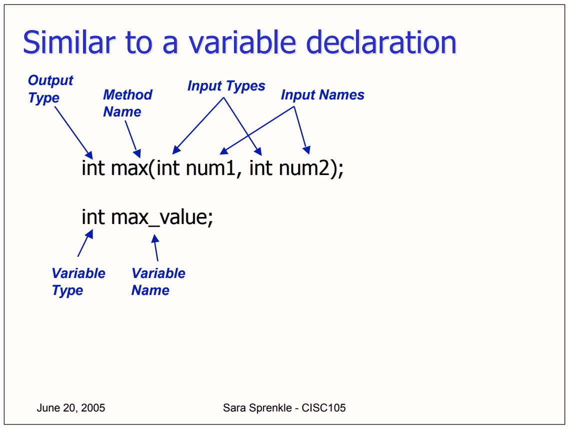
Function Prototypes

- Declare the function before defining it

```
/* function declaration */
int max( int x, int y );
int main() {
    …
    z = max( x, y );
}
/* function definition */
int max( int x, int y ) {
…
}
```

Prototype says the number and types of arguments (parameters) and the type of the return value;

Why would you use this way instead of the other (in terms of readability)?



Similar to a variable declaration

Output Type

Method Name

Input Types

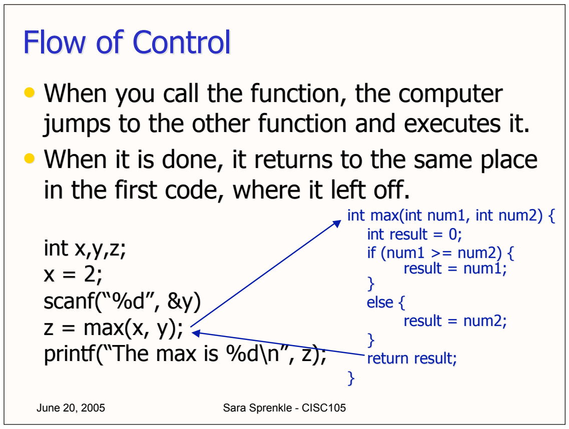
Input Names

```
int max(int num1, int num2);

int max_value;
```

Variable Type

Variable Name

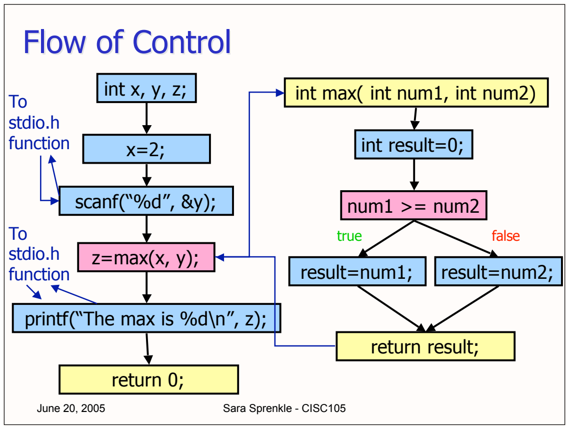


Flow of Control

- When you call the function, the computer jumps to the other function and executes it.
- When it is done, it returns to the same place in the first code, where it left off.

```
int x,y,z;
x = 2;
scanf("%d", &y)
z = max(x, y);
printf("The max is %d\n", z);
```

```
int max(int num1, int num2) {
   int result = 0;
   if (num1 >= num2) {
         result = num1;
   }
   else {
         result = num2;
   }
   return result;
}
```



Flow of Control

int x, y, z;

x=2;

scanf("%d", &y);

z=max(x, y);

printf("The max is %d\n", z);

return 0;

To stdio.h function

To stdio.h function

int max(int num1, int num2)

int result=0;

num1 >= num2

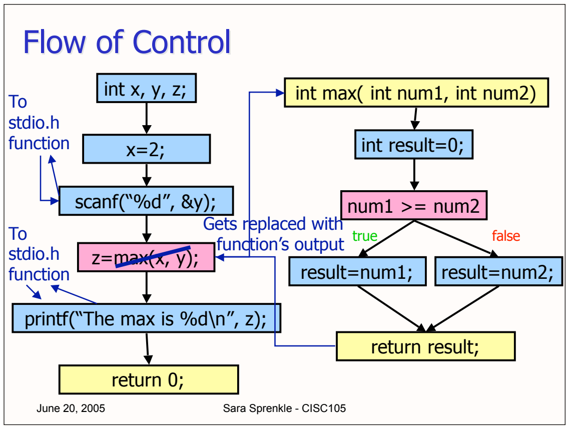
true

false

result=num1;

result=num2;

return result;



Flow of Control

int x, y, z;

x=2;

scanf("%d", &y);

z=~~max(x, y)~~;

printf("The max is %d\n", z);

return 0;

To stdio.h function

To stdio.h function

Gets replaced with function's output

int max(int num1, int num2)

int result=0;

num1 >= num2

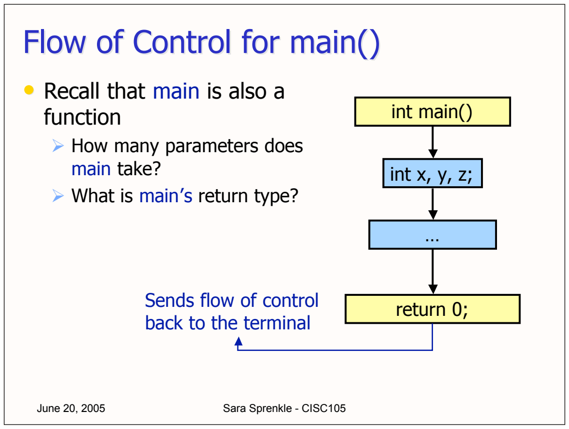
true

false

result=num1;

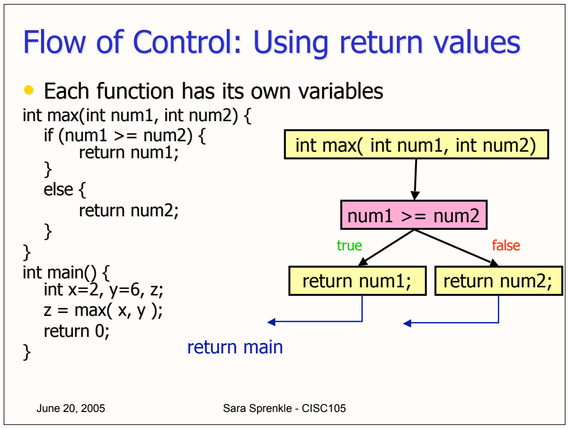
result=num2;

return result;



Flow of Control for main()

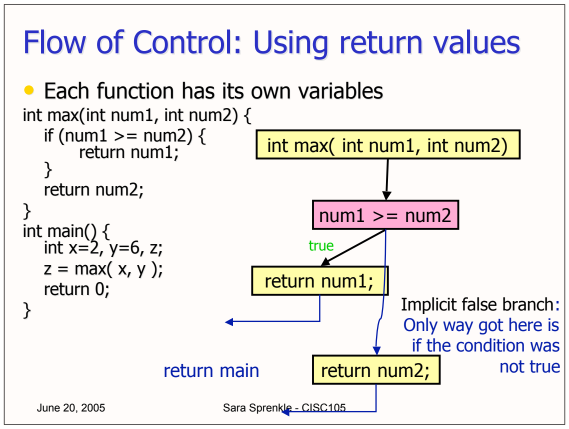
- Recall that main is also a function
  - How many parameters does main take?
  - What is main's return type?



Flow of Control: Using return values

- Each function has its own variables

```
int max(int num1, int num2) {
    if (num1 >= num2) {
        return num1;
    }
    else {
        return num2;
    }
}
int main() {
    int x=2, y=6, z;
    z = max( x, y );
    return 0;
}
```

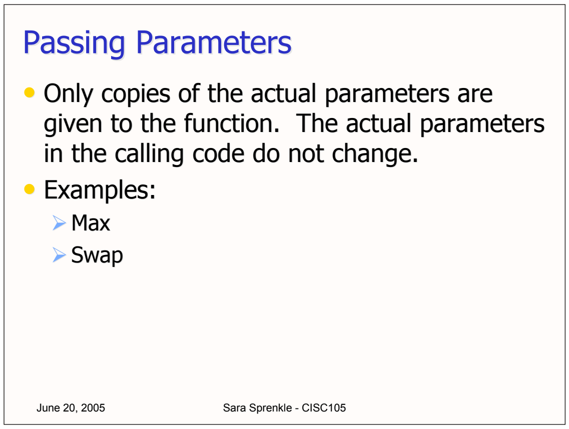


Flow of Control: Using return values

- Each function has its own variables

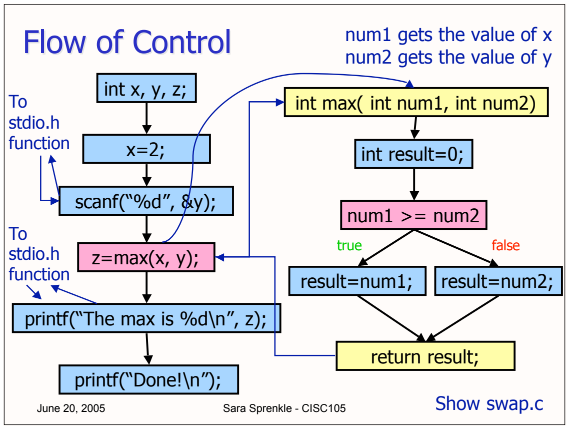
```
int max(int num1, int num2) {
    if (num1 >= num2) {
        return num1;
    }
    return num2;
}
int main() {
    int x=2, y=6, z;
    z = max( x, y );
    return 0;
}
```

Implicit false branch: Only way got here is if the condition was not true



Passing Parameters

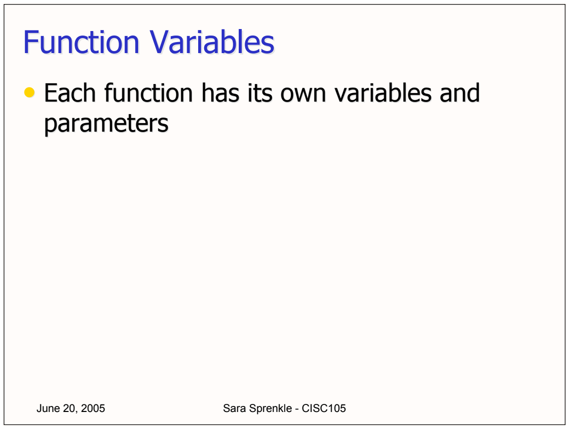
- Only copies of the actual parameters are given to the function. The actual parameters in the calling code do not change.
- Examples:
  - Max
  - Swap



Flow of Control

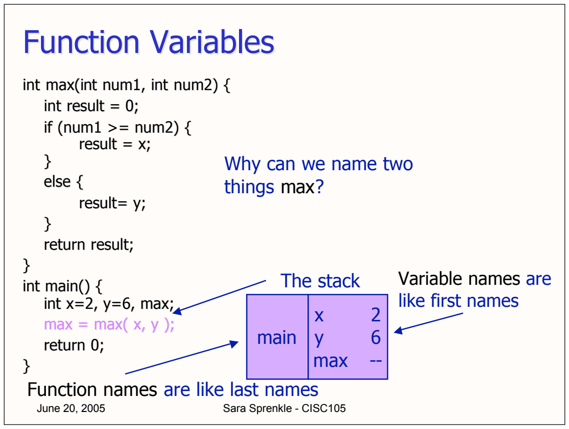
num1 gets the value of x
num2 gets the value of y

Show swap.c



Function Variables

- Each function has its own variables and parameters



Function Variables

```
int max(int num1, int num2) {
    int result = 0;
    if (num1 >= num2) {
          result = x;
    }
    else {
          result= y;
    }
    return result;
}
int main() {
    int x=2, y=6, max;
    max = max( x, y );
    return 0;
}
```

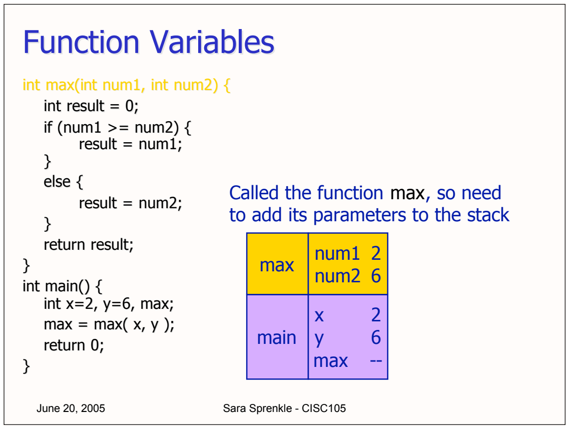
Why can we name two things max?

The stack

Variable names are like first names

main	x	2
	y	6
	max	--

Function names are like last names

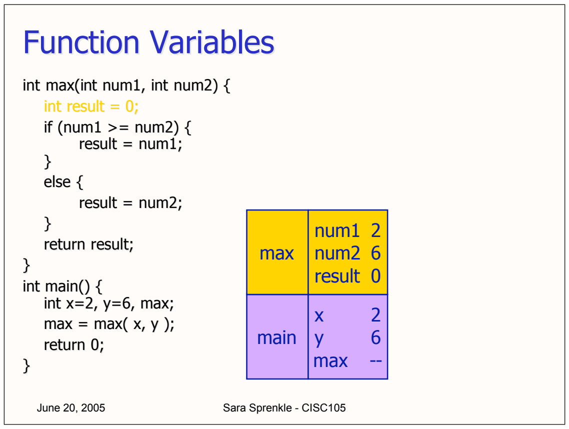


Function Variables

```
int max(int num1, int num2) {
    int result = 0;
    if (num1 >= num2) {
          result = num1;
    }
    else {
          result = num2;
    }
    return result;
}
int main() {
    int x=2, y=6, max;
    max = max( x, y );
    return 0;
}
```

Called the function max, so need to add its parameters to the stack

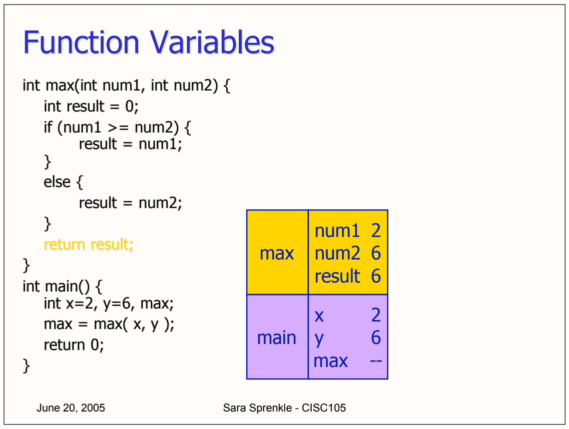
max	num1	2
	num2	6
main	x	2
	y	6
	max	--



Function Variables

```
int max(int num1, int num2) {
    int result = 0;
    if (num1 >= num2) {
          result = num1;
    }
    else {
          result = num2;
    }
    return result;
}
int main() {
    int x=2, y=6, max;
    max = max( x, y );
    return 0;
}
```

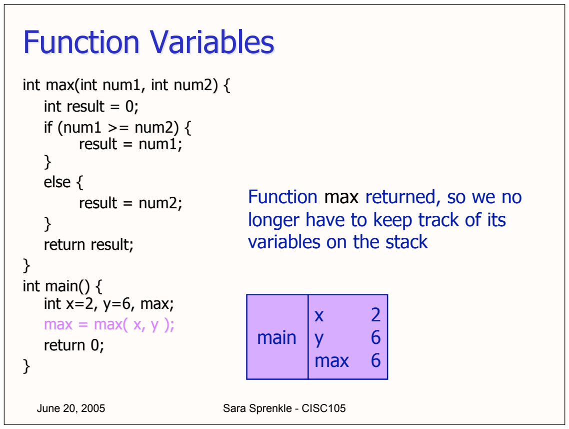
max	num1	2
	num2	6
	result	0
main	x	2
	y	6
	max	--



Function Variables

```
int max(int num1, int num2) {
    int result = 0;
    if (num1 >= num2) {
          result = num1;
    }
    else {
          result = num2;
    }
    return result;
}
int main() {
    int x=2, y=6, max;
    max = max( x, y );
    return 0;
}
```

max	num1	2
	num2	6
	result	6
main	x	2
	y	6
	max	--

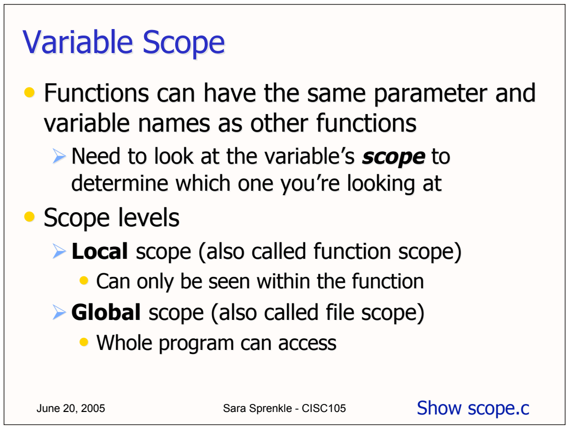


Function Variables

```
int max(int num1, int num2) {
    int result = 0;
    if (num1 >= num2) {
          result = num1;
    }
    else {
          result = num2;
    }
    return result;
}
int main() {
    int x=2, y=6, max;
    max = max( x, y );
    return 0;
}
```

Function max returned, so we no longer have to keep track of its variables on the stack

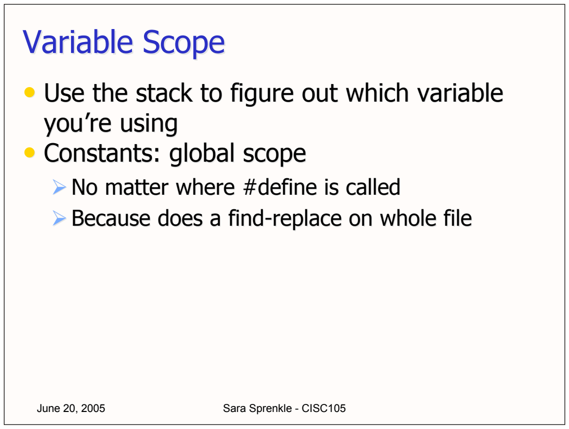
main	x	2
	y	6
	max	6



Variable Scope

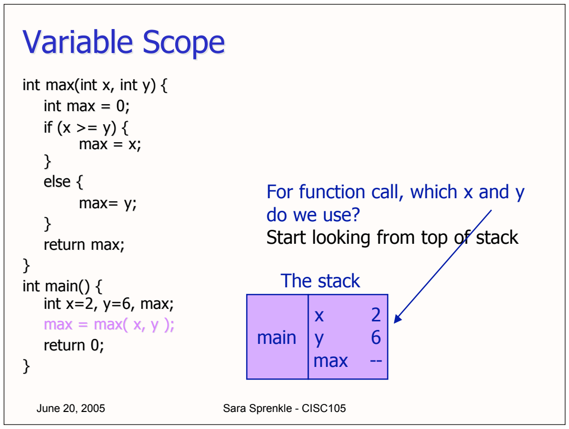
- Functions can have the same parameter and variable names as other functions
  - Need to look at the variable's ***scope*** to determine which one you're looking at
- Scope levels
  - **Local** scope (also called function scope)
    - Can only be seen within the function
  - **Global** scope (also called file scope)
    - Whole program can access

Show scope.c



Variable Scope

- Use the stack to figure out which variable you're using
- Constants: global scope
  - No matter where #define is called
  - Because does a find-replace on whole file



Variable Scope

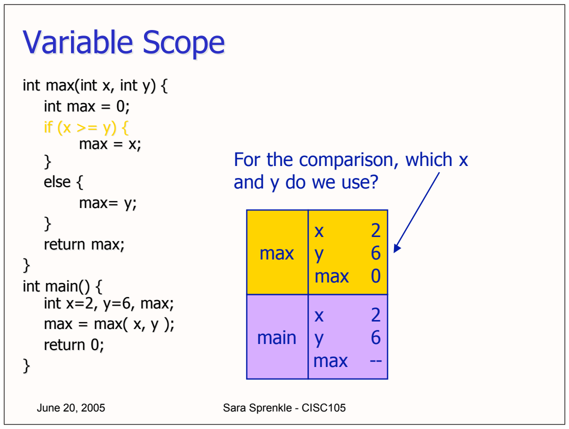
```
int max(int x, int y) {
    int max = 0;
    if (x >= y) {
          max = x;
    }
    else {
          max= y;
    }
    return max;
}
int main() {
    int x=2, y=6, max;
    max = max( x, y );
    return 0;
}
```

For function call, which x and y do we use?

Start looking from top of stack

The stack

main	x y max	2 6 --

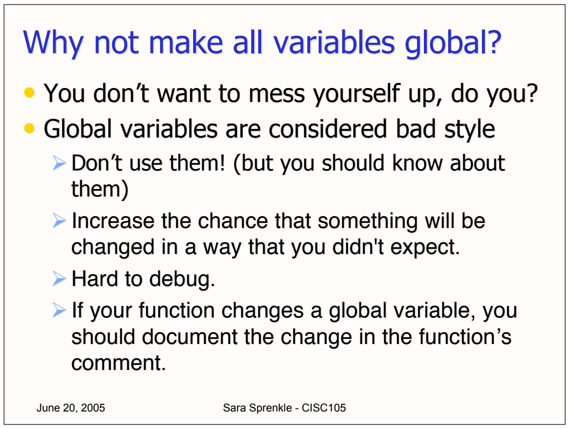


Variable Scope

```
int max(int x, int y) {
    int max = 0;
    if (x >= y) {
          max = x;
    }
    else {
          max= y;
    }
    return max;
}
int main() {
    int x=2, y=6, max;
    max = max( x, y );
    return 0;
}
```

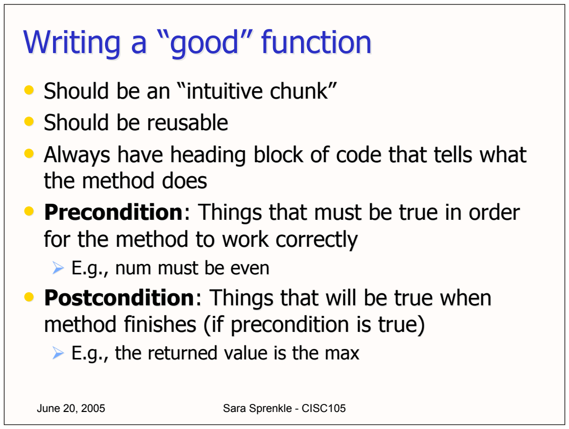
For the comparison, which x and y do we use?

max	x y max	2 6 0
main	x y max	2 6 --



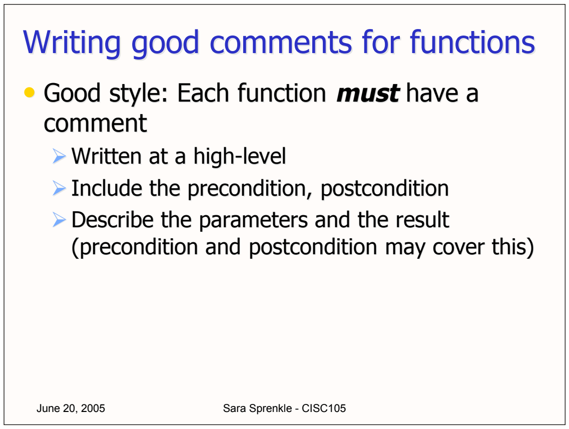
Why not make all variables global?

- You don't want to mess yourself up, do you?
- Global variables are considered bad style
  - Don't use them! (but you should know about them)
  - Increase the chance that something will be changed in a way that you didn't expect.
  - Hard to debug.
  - If your function changes a global variable, you should document the change in the function's comment.



Writing a "good" function

- Should be an "intuitive chunk"
- Should be reusable
- Always have heading block of code that tells what the method does
- **Precondition**: Things that must be true in order for the method to work correctly
  - E.g., num must be even
- **Postcondition**: Things that will be true when method finishes (if precondition is true)
  - E.g., the returned value is the max



Writing good comments for functions

- Good style: Each function ***must*** have a comment
  - Written at a high-level
  - Include the precondition, postcondition
  - Describe the parameters and the result (precondition and postcondition may cover this)

Using functions

- Temperature converter
- Average calculator
- Calculate a number raised to a power

Recursive Functions

- Functions can call themselves
  - Divide and conquer:
    - common way to solve problems
  - Break the problem down into a smaller problem that you can solve
- Consider factorial:
  - n! = n * (n-1)!
  - (n-1)! = (n-1) * (n-2)!
  - (n-2)! = (n-2) * (n-3)!
  - …
  - 1! = 1

Break down to a base case that you know the answer to

Practice: Recursive Functions

- Consider power:
  - $a^{exp} = a^{exp-1} * a$
  - $a^{exp-1} = a^{exp-2} * a$
  - …
  - What's the base case?
  - What's the recursive call?
- Write power
  - iteratively (using a loop)
  - recursively

Goals of Good Programs: Performance

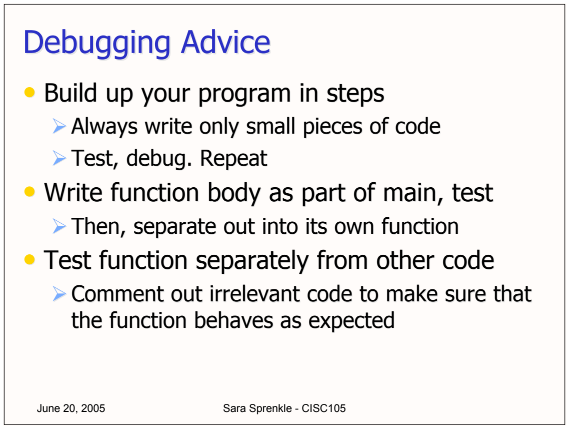
- Program only takes as much space, time as needed
  - No extra variables
  - Don't use double if int will do
  - No extra comparisons

Goals of Good Programs: Readability

- Descriptive variable, function names
- Good, descriptive, high-level comments
- Indentation
  - Use Emacs!
- Vertical spacing
  - Add space between "groups" of related code
- Functions
  - Break up long code into smaller, more readable components
- Line length (Use esc-q in Emacs)

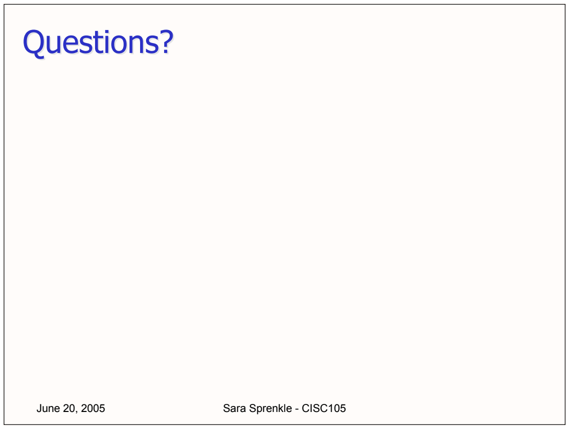
Goals of Good Programs: Extensibility

- Should be able to easily extend your program's use
  - Constants
  - User-input
  - Functions
- Modularity
  - Functions that can be reused in other code
- See the C coding standards on the course web page for more info about these goals



Debugging Advice

- Build up your program in steps
  - Always write only small pieces of code
  - Test, debug. Repeat
- Write function body as part of main, test
  - Then, separate out into its own function
- Test function separately from other code
  - Comment out irrelevant code to make sure that the function behaves as expected



Questions?