C/C++ Functions

Mark Redekopp

A QUICK LOOK

Functions

- Also called *procedures* or *methods*
- Collection of code that performs a task
- Do a task given some inputs (but we don't care how it does it…a.k.a. "black box" methodology)
  - Black-box = We know ***what*** it does, but not ***how*** it does it
- Attributes
  - Has a name to identify it (e.g. ‘avg’)
  - Takes in 0 or more inputs (a.k.a. parameters or arguments)
  - Performs computation
    - Start and end of code belonging to the function are indicated by curly braces { … }
    - Sequence of statements
  - Returns at most a single value (i.e. 1 value or 0 values = 'void')
  - Return value is substituted for the function call

Execution of a Function

- Statements in a function are executed sequentially by default
- Defined once, called over and over
- Functions can 'call' other functions
  - Goes and executes that collection of code then returns to continue the current function
- Compute max of two integers

  Each 'call' causes the program to pause the current function, go to the called function and execute its code with the given arguments then return to where the calling function left off,
- Return value is substituted in place of the function call

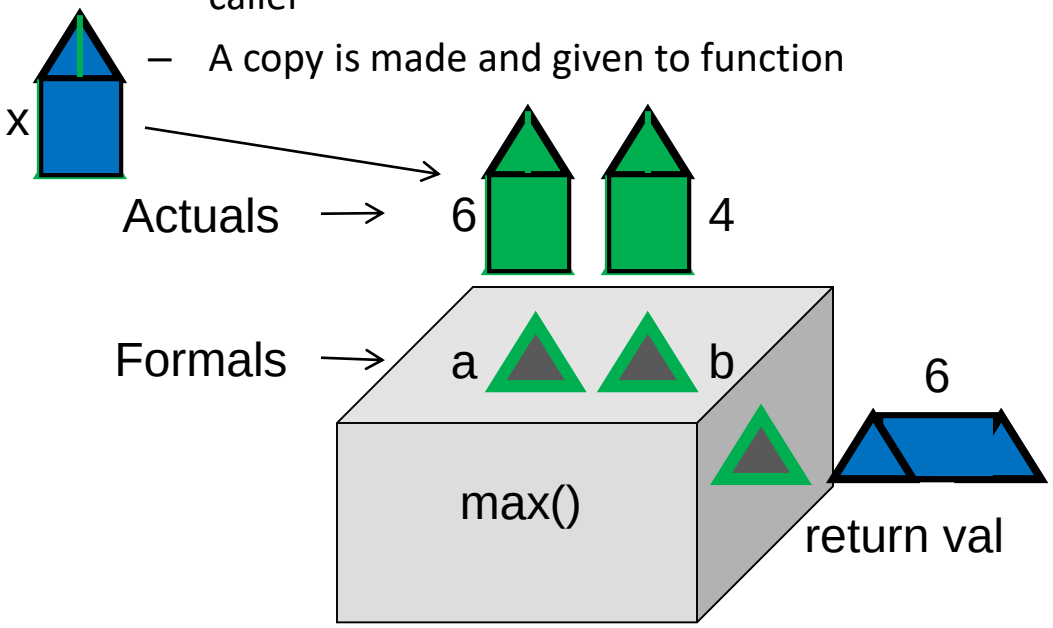
```
#include <iostream>
using namespace std;

int max(int a, int b)
{
  if(a > b)
    return a;
  else
    return b;
}


int main(int argc, char *argv[])
{
   int x=6, z;
   z = max(x,4);
   cout << "Max is " << z << endl;
   z = max(125, 199);
   cout << "Max is " << z << endl;
   return 0;
}
```

6

199

Parameter Passing

- Formal parameters, a and b
  - Type of data they expect
  - Names that will be used internal to the function to refer to the values
- Actual parameters
  - Actual values input to the function code by the caller
  - A copy is made and given to function



Each type is a "different" shape (int = triangle, double = square, char = circle). Only a value of that type can "fit" as a parameter..

```
#include <iostream>
using namespace std;

int max(int a, int b)
{
  if(a > b)
    return a;
  else
    return b;
}

int main(int argc, char *argv[])
{
   int x=6, z;
   z = max(x,4);
   cout << "Max is " << z << endl;
   z = max(125, 199);
   cout << "Max is " << z << endl;
   return 0;
}
```

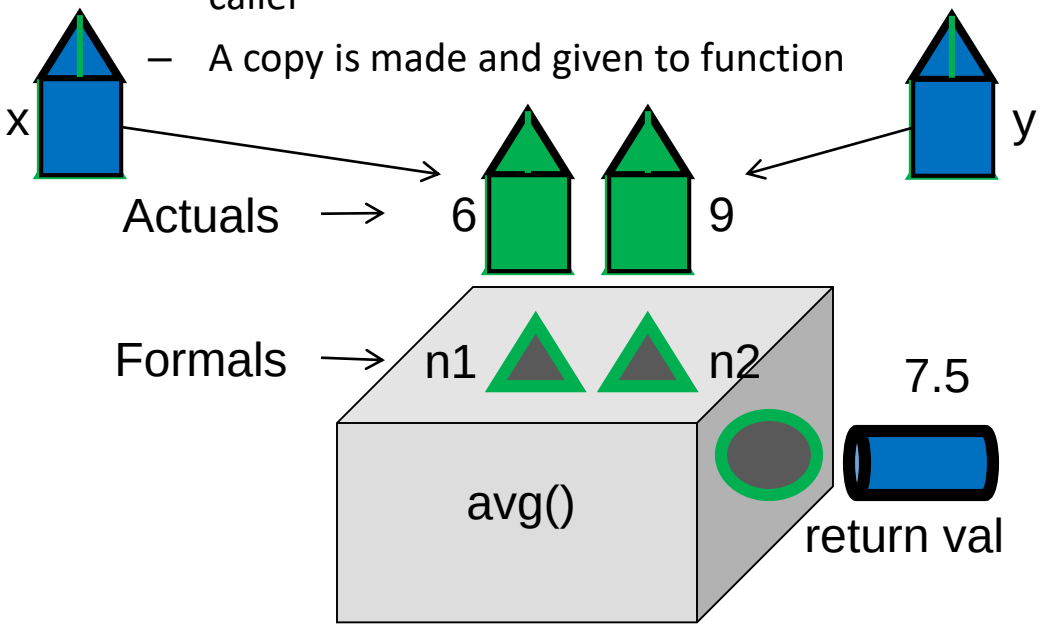
Formals

Actuals

Actuals

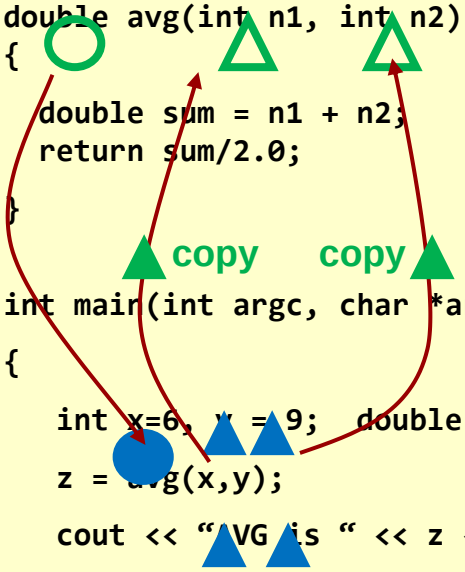
Parameter Passing

- Formal parameters, n1 and n2
  - Type of data they expect
  - Names that will be used internal to the function to refer to the values
- Actual parameters
  - Actual values input to the function code by the caller
  - A copy is made and given to function



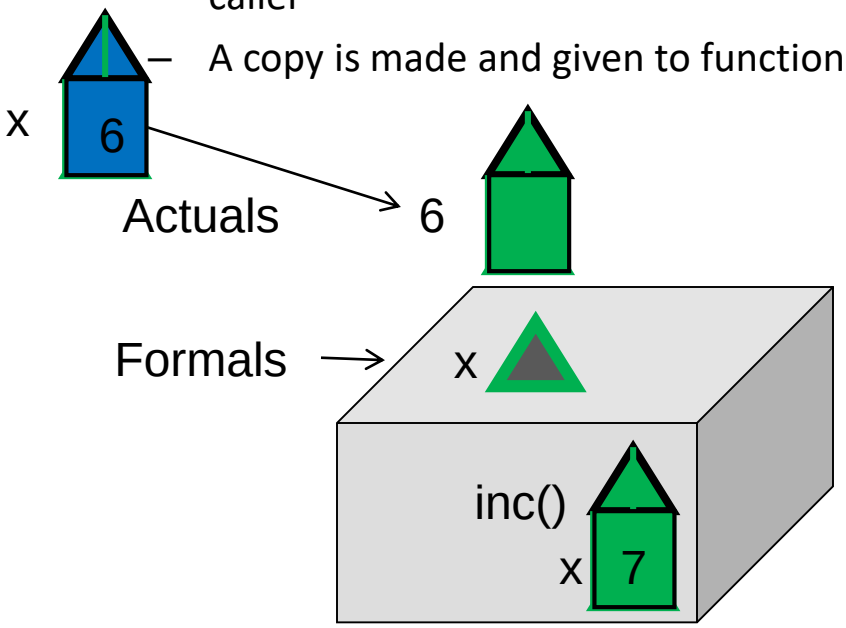
Each type is a "different" shape (int = triangle, double = square, char = circle). Only a value of that type can "fit" as a parameter..

```
#include <iostream>
using namespace std;

double avg(int n1, int n2)
{
  double sum = n1 + n2;
  return sum/2.0;
}

int main(int argc, char *argv[])
{
   int x=6, y = 9;  double z;
   z = avg(x,y);
   cout << "AVG is " << z << endl;
   z = avg(x, 2);
   cout << "AVG is " << z << endl;
   return 0;
}
```



Parameter Passing

- Formal parameters, n1 and n2
  - Type of data they expect
  - Names that will be used internal to the function to refer to the values
- Actual parameters
  - Actual values input to the function code by the caller
  - A copy is made and given to function



Each type is a "different" shape (int = triangle, double = square, char = circle). Only a value of that type can "fit" as a parameter..

```
#include <iostream>
using namespace std;

void inc(int x)
{
  x = x+1;
}

int main(int argc, char *argv[])
{
   int x=6;
   inc(x);
   cout << "X is " << x << endl;
   return 0;
}
```

Program Decomposition

- C is a procedural language
  - Main unit of code organization, problem decomposition, and abstraction is the “function” or “procedure”
  - Function or procedure is a unit of code that
    - Can be called from other locations in the program
    - Can be passed variable inputs (a.k.a. arguments or parameters)
    - Can return a value to the code that called it
- C++ is considered an “object-oriented” language (really just adds objected-oriented constructs to C)
  - Main unit of organization, problem decomposition, and abstraction is an object (collection of code & associated data)

Exercise

- To decompose a program into functions, try listing the ***verbs*** or ***tasks*** that are performed to solve the problem
  - Model a game of poker as a series of tasks/procedures…
    - Shuffle(), deal(), bet(), flop(), …
  - A database representing a social network
    - addUser(), addFriend(), updateStatus(), etc.

Anatomy of a function

- Return type (any valid C type)
  - void, int, double, char, etc.
  - void means return nothing
- Function name
  - Any valid identifier
- Input arguments inside ()
  - Act like a locally declared variable
- Code
  - In {…}
- Non-void functions must have 1 or more return statements
  - First 'return' executed immediately quits function

```
void printMenu()
{
  cout << "Welcome to ABC 2.0:" << endl;
  cout << "====================" << endl;
  cout << "  Enter an option:" << endl;
  cout << "    1.) Start" << endl;
  cout << "    2.) Continue" << endl;
  cout << "    3.) End\n" << endl;
}

bool only_2_3_factors(int num)
{
   while(num % 2 == 0){
     ...
   }
   ...
   if(num==1)
     return 1;


   return 0;

}

double triangle_area(double b, double h)
{
   double area;
   area = 0.5 * b * h;
   return area;
}
```

Function Prototypes

- The compiler (g++/clang++) needs to "know" about a function before it can handle a call to that function
- The compiler will scan a file from top to bottom
- If it encounters a call to a function before the actual function code it will complain…[Compile error]
- …Unless a prototype ("declaration") for the function is defined earlier
- A prototype only needs to include data types for the parameters but not their names (ends with a ';')
  - Prototype is used to check that you are calling it with the correct syntax (i.e. parameter data types & return type) (like a menu @ a restaurant)

X

```
int main()
{
  double area1,area2,area3;
  area3 = triangle_area(5.0,3.5);
}


double triangle_area(double b, double h)
{
   return 0.5 * b * h;
}
```

Compiler encounters a call to triangle_area() before it has seen its definition (Error!)

```
double triangle_area(double, double);

int main()
{
  double area1,area2,area3;
  area3 = triangle_area(5.0,3.5);
}


double triangle_area(double b, double h)
{
   return 0.5 * b * h;
}
```

Compiler sees a prototype and can check the syntax of any following call and expects the definition later.

The Need For Prototypes

- How would you order the functions in the program on the left if you did NOT want to use prototypes?
- You can't!

```
int main()
{
  cout << f1(5) << endl;
}

int f1(int x)
{
   return f2(x*x);
}

int f2(int y)
{
  if(x > 10000) return;
  else f1(y);
}
```

```
int f1(int);
int f2(int);

int main()
{
  cout << f1(5) << endl;
}

int f1(int x)
{
   return f2(x*x);
}

int f2(int y)
{
  if(x > 10000) return;
  else f1(y);
}
```

Overloading: A Function's Signature

- What makes up a signature (uniqueness) of a function
  - name
  - **number** and **type** of arguments
- No two functions are allowed to have the same signature; the following 6 functions are unique and allowable…
  - `int f1(int),  int f1(double),  int f1(int, double)`
  - `void f1(char), double f1(),  int f1(int, char)`
- Return type does not make a function unique
  - `int f1()` and `double f1()` are not unique and thus not allowable
- Two functions with the same name are said to be "**overloaded**"
  - `int max(int, int);  double max(double, double);`

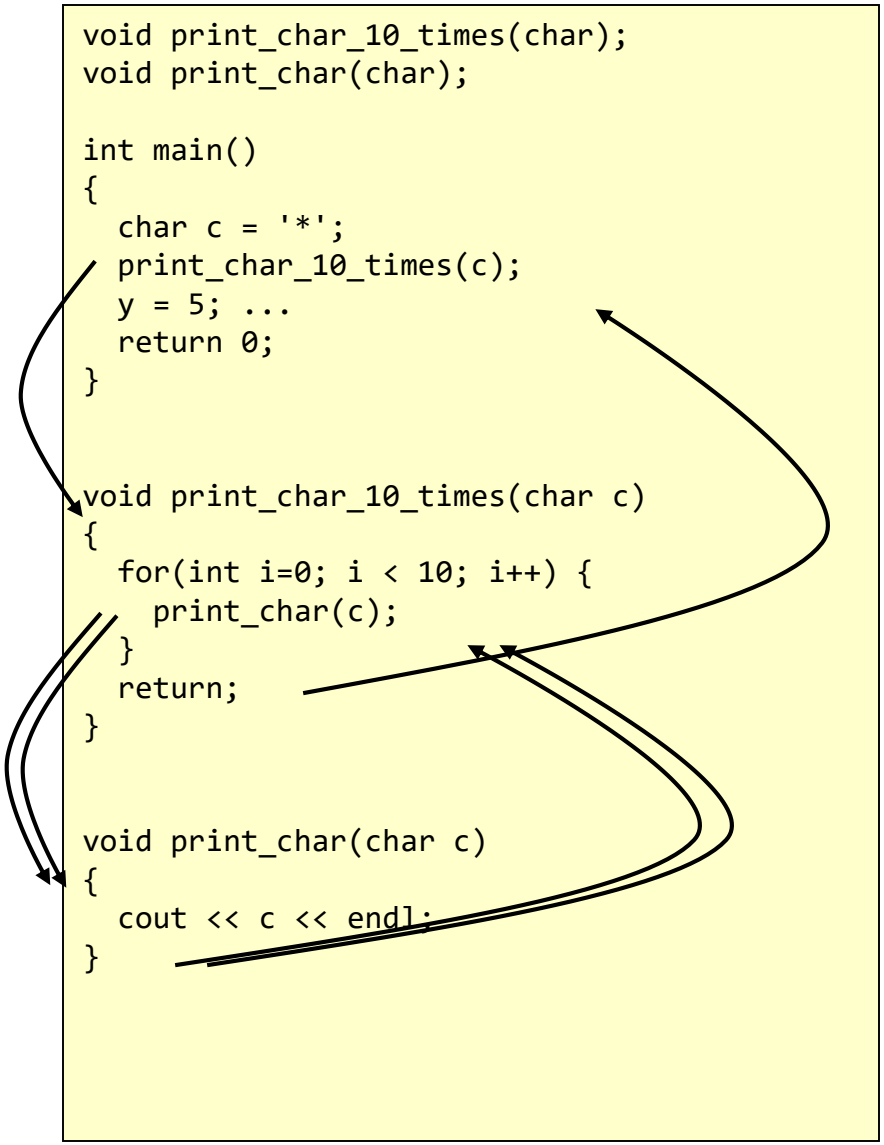
Practice

- Remove Factors
  - Websheets Exercise: cpp/functions/remove_factor
- Draw an ASCII square on the screen
  - Websheets Exercise: cpp/functions/draw_square
- Practice overloading a function
  - Websheets Exercise: cpp/functions/overload

FUNCTION CALL SEQUENCING

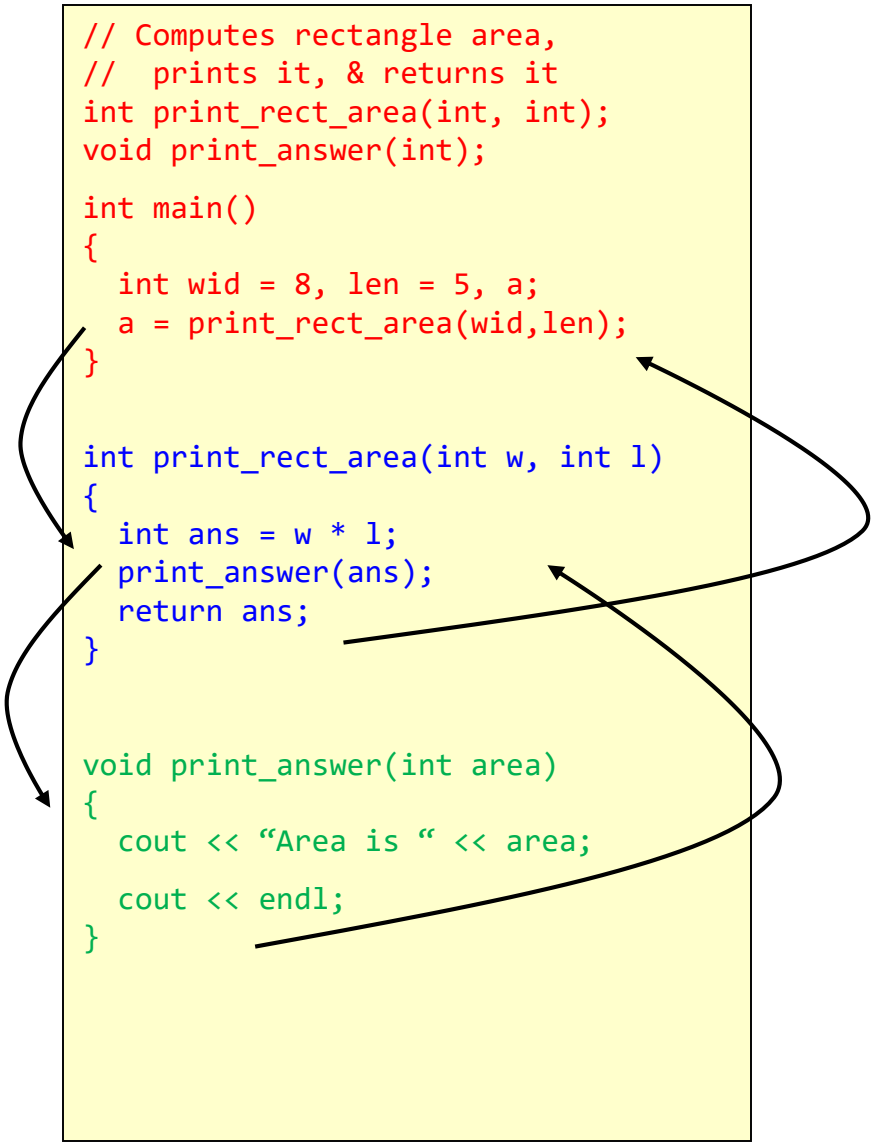
Function Call Sequencing

- Functions can call other functions and so on…
- When a function is called the calling function is suspended (frozen) along with all its data and control jumps to the start of the called function
- When the called function returns execution resumes in the calling function
- Each function has its own set of variables and "scope"
  - Scope refers to the visibility/accessibility of a variable from the current place of execution



More Function Call Sequencing

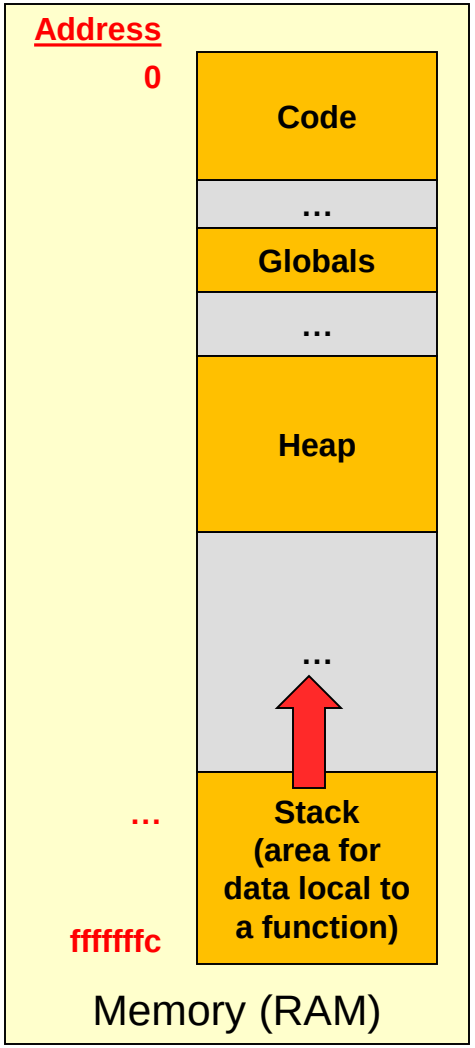
- As one function calls another, they execute in a last-in, first-out fashion (i.e. the last one called is the first one to finish & return)
  - Just like in the cafeteria the last plate put on the top of the stack is the first one to be pulled off (always access the top item)
- How does the computer actually track where to return to when a function completes



```
// Computes rectangle area,
//  prints it, & returns it
int print_rect_area(int, int);
void print_answer(int);

int main()
{
  int wid = 8, len = 5, a;
  a = print_rect_area(wid,len);
}


int print_rect_area(int w, int l)
{
  int ans = w * l;
  print_answer(ans);
  return ans;
}


void print_answer(int area)
{
  cout << "Area is " << area;

  cout << endl;
}
```

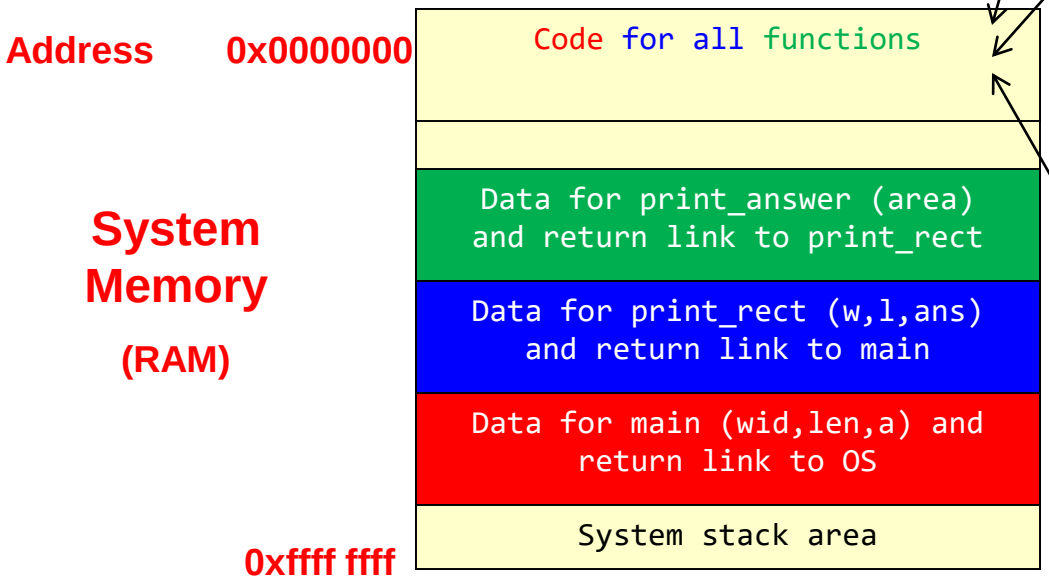
Memory Organization

- 32-bit address range (0x0 – 0xffffffff)
- Code usually sits at lower addresses
- Global variables/data somewhere after code
- Heap: Area of memory that can be allocated and de-allocated during program execution (i.e. dynamically at run-time) based on the needs of the program
- System stack (memory for each function instance that is alive)
  - Local variables
  - Return link (where to return)
  - etc.



More Function Call Sequencing

- Computer maintains a "stack" of function data and info in memory (i.e. RAM)
  - Each time a function is called, the computer allocates memory for that function on the top of the stack and a link for where to return
  - When a function returns that memory is de-allocated and control is returned to the function now on top

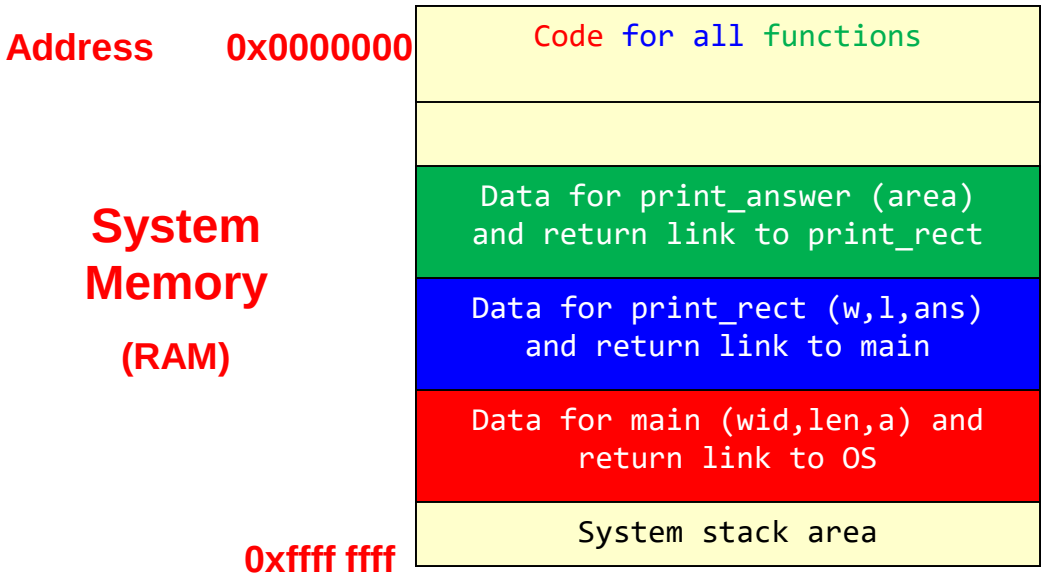



```
// Computes rectangle area,
//  prints it, & returns it
int print_rect_area(int, int);
void print_answer(int);

int main()
{
  int wid = 8, len = 5, a;
  a = print_rect_area(wid,len);
}


int print_rect_area(int w, int l)
{
  int ans = w * l;
  print_answer(ans);
  return ans;
}


void print_answer(int area)
{
  cout << "Area is " << area;

  cout << endl;
}
```

LOCAL VARIABLES & SCOPE

Local Variables

- Any variable declared inside a function is called a “local” variable
- It lives in the stack area for that function
- It dies when the function returns




```
// Computes rectangle area,
//  prints it, & returns it
int print_rect_area(int, int);
void print_answer(int);

int main()
{
  int wid = 8, len = 5, a;
  a = print_rect_area(wid,len);
}


int print_rect_area(int w, int l)
{
  int ans = w * l;
  print_answer(ans);
  return ans;
}


void print_answer(int area)
{
  cout << “Area is “ << area;

  cout << endl;
}
```

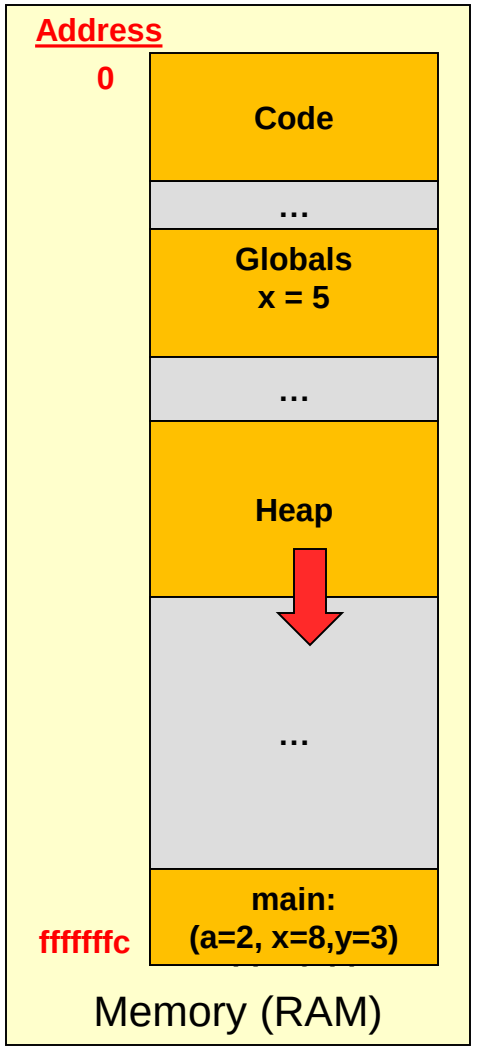
Scope

- Global variables live as long as the program is running
- Variables declared in a block { … } are ‘local’ to that block
  - { … } of a function
  - { … } of a loop, if statement, etc.
  - Die/deallocated when the program reaches the end of the block…don’t try to access them intentionally or unintentionally after they are ‘out of scope’/deallocated
  - Actual parameters act as local variables and die when the function ends
- When variables share the same name the closest declaration will be used by default

Scope Example

- Globals live as long as the program is running
- Variables declared in a block { … } live as long as the block has not completed
  - { … } of a function
  - { … } of a loop, if statement, etc.
- When variables share the same name the closest declaration will be used by default

```
#include <iostream>
using namespace std;

int x = 5;

int main()
{
  int a, x = 8, y = 3;
  cout << "x = " << x << endl;
  for(int i=0; i < 10; i++){
    int j = 1;
    j = 2*i + 1;
    a += j;
  }
  a = doit(y);
  cout << "a=" << a ;
  cout << "y=" << y << endl;
  cout << "glob. x" << ::x << endl;
}


int doit(int x)
{
   x--;
   return x;
}
```



PASS BY VALUE

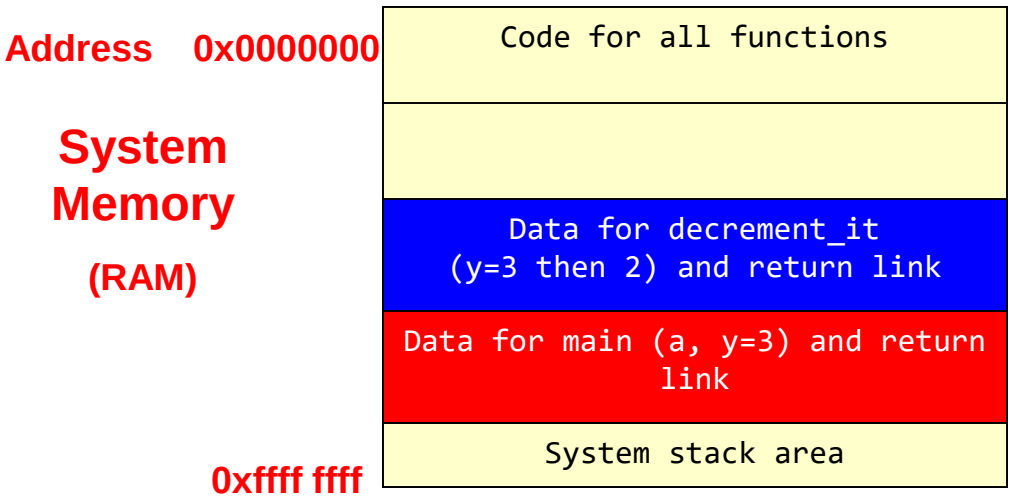
Pass-by-Value

- Passing an argument to a function makes a copy of the argument
- It is like e-mailing an attached document
  - You still have the original on your PC
  - The recipient has a copy which he can modify but it will not be reflected in your version
- Communication is essentially one-way
  - Caller communicates arguments to callee, but callee cannot communicate back because he is working on copies…
  - The only communication back to the caller is via a return value.

Pass by Value

- Notice that actual arguments are different memory locations/variables than the formal arguments
- When arguments are passed a **copy** of the actual argument value (e.g. 3) is placed in the formal parameter (x)
- The value of y cannot be changed by any other function (remember it is local)

```
void decrement_it(int);

int main()
{
  int a, y = 3;
  decrement_it(y);
  cout << "y = " << y << endl;
  return 0;
}

void decrement_it(int y)
{
   y--;
}
```



Nested Call Practice

- Find characters in a string then use that function to find how many vowels are in a string
  - Websheets Exercise: cpp/functions/vowels

Another Exercise

- Guessing game
  - Number guessing game [0-19]…indicate higher or lower until they guess correctly or stop after 5 unsuccessful guesses
  - Use a function to perform one "turn" of the game and return whether the user guessed the number correctly
    - bool guessAndCheck(int secretNum);

```
#include <iostream>
#include <cstdlib>
#include <ctime>

int main()
{
  srand(time(0));
  int secretNum = rand() % 20;
  // Now create a game that
  //  lets the user try to guess
  //  the random number in up to
  //  5 guesses

}
```