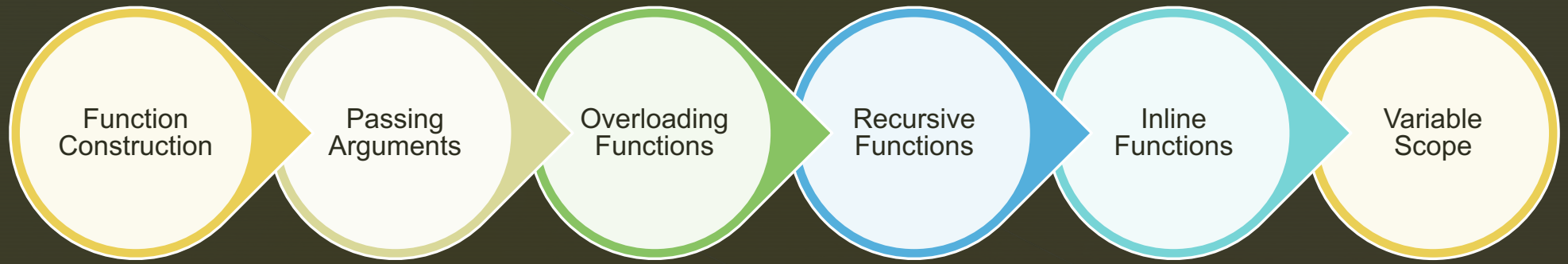
DR. (MRS) T-S.M.A. ADJAIDOO

03 Functions

Today's Lesson



Learning Outcomes

1	• Learners should have a general understanding of functions and how to use them
2	• Learners should understand the various ways of passing arguments to functions
3	• Learners should appreciate the scope of variables with respect to functions

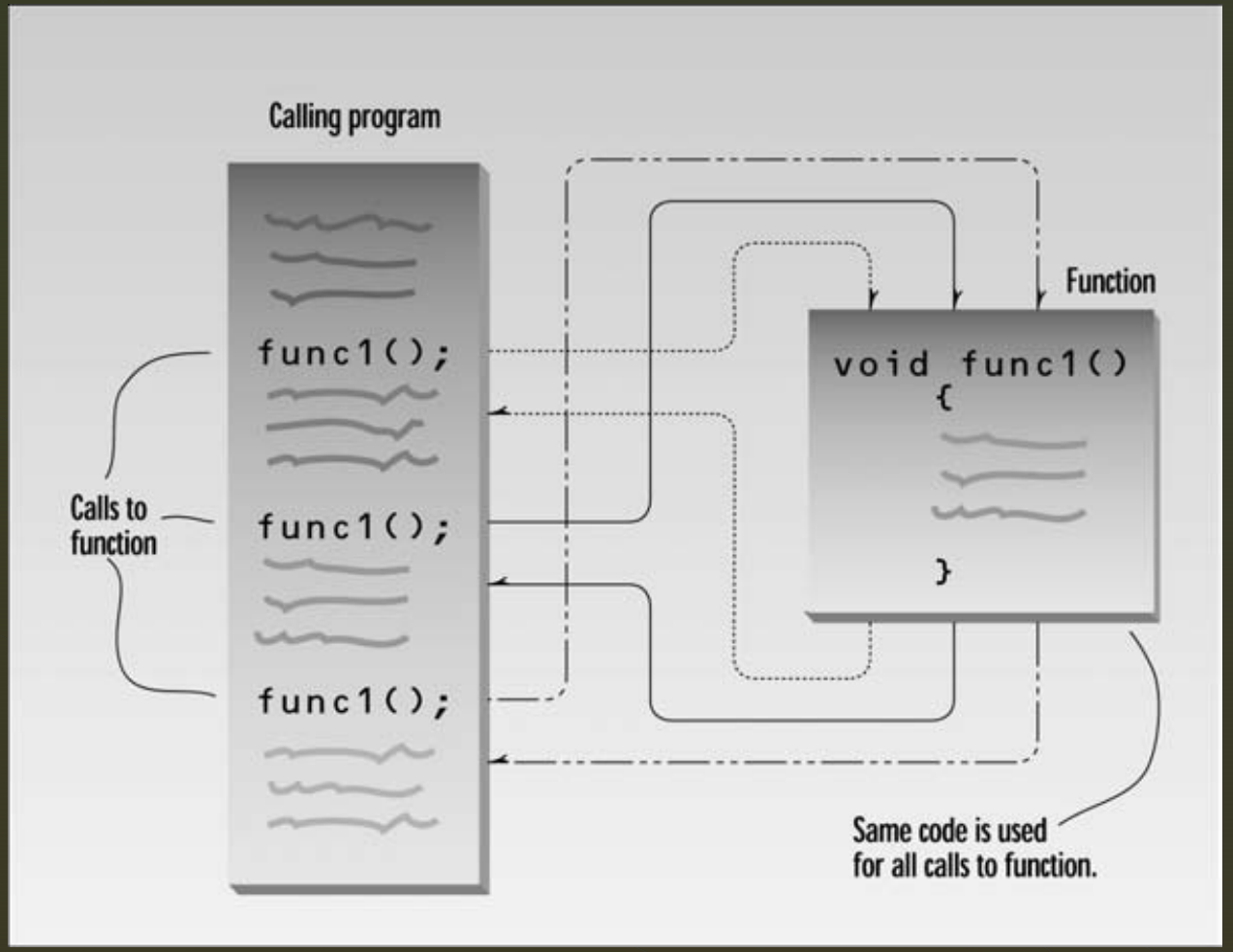
Functions

Functions

- A function groups a number of program statements into a unit and gives it a name.
- This unit can then be invoked from other parts of the program.
- Any sequence of instructions that appears in a program more than once is a candidate for being made into a function.
- The function’s code is stored in only one place in memory, even though the function is executed many times in the course of the program.

Functions

How a function works in a program

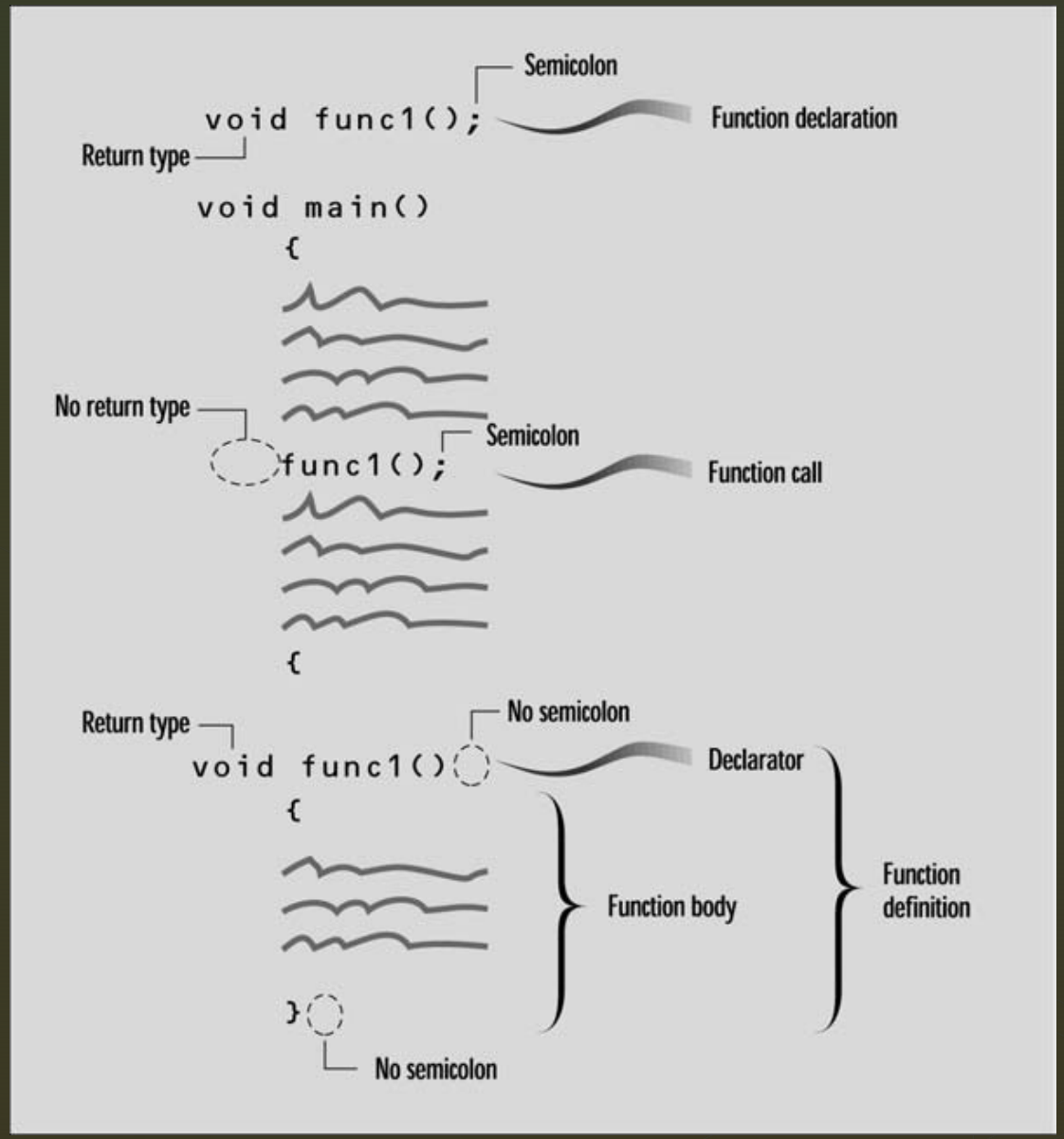


Functions

```
// demonstrates simple function
#include <iostream>
using namespace std;

void starline(); //function declaration
// (prototype)

int main()
{
    starline(); //call to function

    cout << "Data type Range" << endl;

    starline(); //call to function

    cout << "char -128 to 127" << endl
    << "short -32,768 to 32,767" << endl
    << "int System dependent" << endl
    << "long -2,147,483,648 to 2,147,483,647" << endl;

    starline(); //call to function

    return 0;
}
```

```
// starline()
// function definition
void starline() //function declarator
{
    for(int j=0; j<45; j++) //function body
    cout << "*";
    cout << endl;
}
```

Functions


Semicolon
void func1();
Function declaration
Return type
void main()
{
No return type
Semicolon
func1();
Function call
{
Return type
No semicolon
void func1()
Declarator
{
Function body
Function definition
}
No semicolon

Functions

- When the function is called, control is transferred to the first statement in the function body.
- The other statements in the function body are then executed, and when the closing brace is encountered, control returns to the calling program.

Passing Arguments to Functions

- An argument is a piece of data (an int value, for example) passed from a program to the function.
- Arguments allow a function to operate with different values, or even to do different things, depending on the requirements of the program calling it.

Passing Arguments to Functions: Passing Constants

```
// demonstrates function arguments
#include <iostream>
using namespace std;

void repchar(char, int); //function declaration

int main()
{
    repchar('-', 43); //call to function
    cout << "Data type Range" << endl;

    repchar('=', 23); //call to function
    cout << "char -128 to 127" << endl
    << "short -32,768 to 32,767" << endl
    << "int System dependent" << endl
    << "double -2,147,483,648 to 2,147,483,647" << endl;

    repchar('-', 43); //call to function
    return 0;
}
```

```
// repchar()
// function definition
void repchar(char ch, int n) //function declarator
{
for(int j=0; j<n; j++) //function body
cout << ch;
cout << endl;
}
```

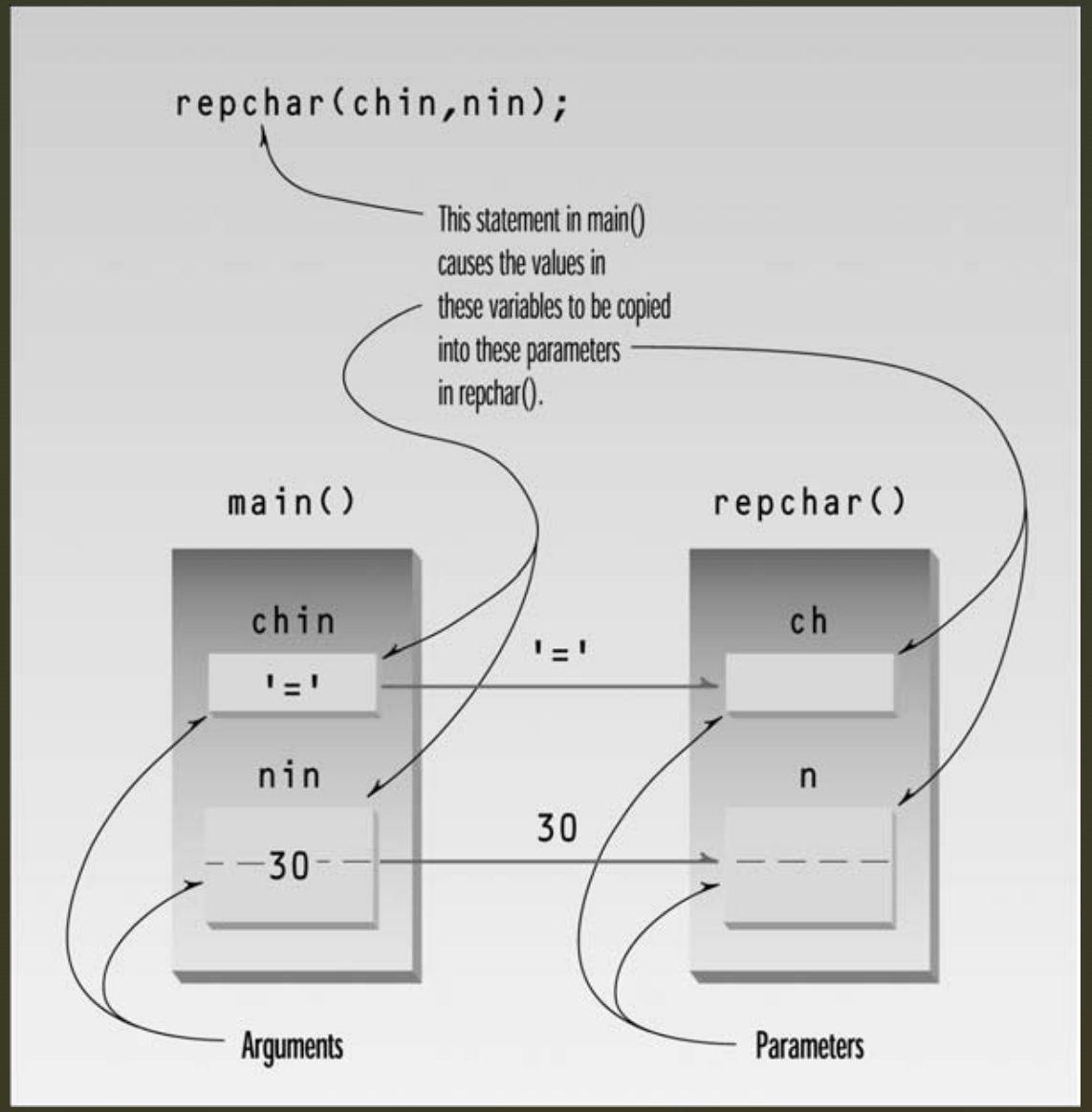
Passing Arguments to Functions: Passing Variables

```
// demonstrates variable arguments
#include <iostream>
using namespace std;

void repchar(char, int); //function declaration

int main()
{
    char chin;
    int nin;
    cout << "Enter a character: ";
    cin >> chin;
    cout << "Enter number of times to repeat it: ";
    cin >> nin;
    repchar(chin, nin);
    return 0;
}
```

```
// repchar()
// function definition
void repchar(char ch, int n) //function declarator
{
    for(int j=0; j<n; j++) //function body
    cout << ch;
    cout << endl;
}
```

Passing Arguments to Functions: Passing by Value

Passing arguments where the function creates copies of the arguments passed to it, is called *passing by value.*



Reference Arguments

- A reference provides an alias—a different name—for a variable.
- One of the most important uses for references is in passing arguments to functions.
- When arguments are passed by value, the called function creates a new variable of the same type as the argument and copies the argument's value into it.
- The function cannot access the original variable in the calling program, only the copy it created
- Passing arguments by value is useful when the function does not need to modify the original variable in the calling program.
- In fact, it offers insurance that the function cannot harm the original variable.

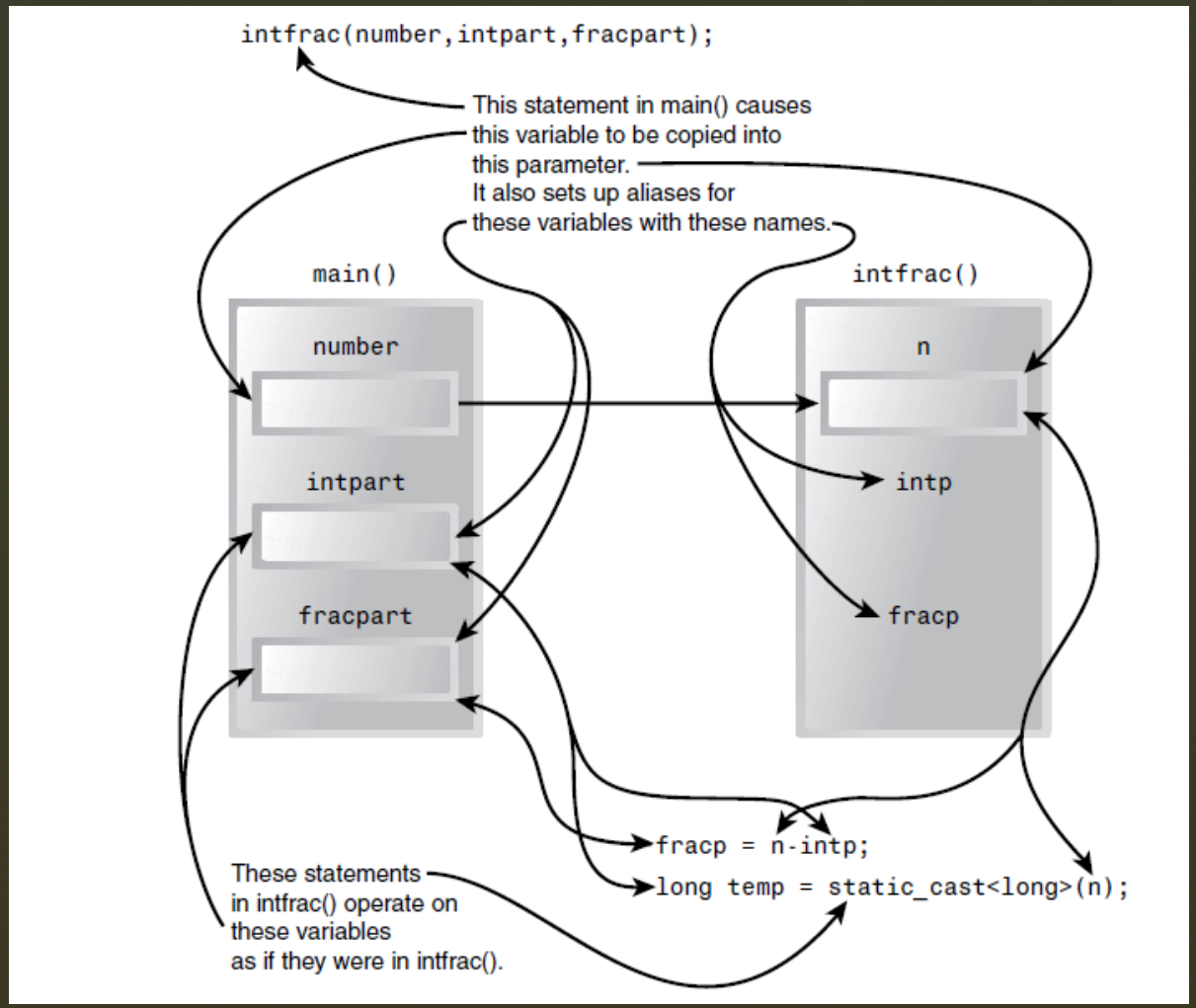
Passing By Reference

- Passing arguments by reference uses a different mechanism.
- Instead of a value being passed to the function, a *reference to the original variable, in the calling program, is passed*
- An important advantage of passing by reference is that the function can access the actual variables in the calling program.
- Among other benefits, this provides a mechanism for passing more than one value from the function back to the calling program.

Passing By Reference

```
// demonstrates passing by reference
#include <iostream>
using namespace std;

int main()
{
    void intfrac(float, float&, float&); //declaration
    float number, intpart, fracpart; //float variables
    do {
        cout << "\nEnter a real number: "; //number from user
        cin >> number;
        intfrac(number, intpart, fracpart); //find int and frac
        cout << "Integer part is " << intpart //print them
        << ", fraction part is " << fracpart << endl;
    }
    while( number != 0.0 ); //exit loop on 0.0
    return 0;
}

// intfrac()
// finds integer and fractional parts of real number
void intfrac(float n, float& intp, float& fracp)
{
    long temp = static_cast<long>(n); //convert to long,
    intp = static_cast<float>(temp); //back to float
    fracp = n - intp; //subtract integer part
}
```

Passing By Reference



Passing By Reference

- Reference arguments are indicated by the ampersand (&) following the data type:

  float& intp

- The & indicates that intp is an alias—another name—for whatever variable is passed as an argument

Passing By Reference

```
// orders two arguments passed by reference
#include <iostream>
using namespace std;

int main()
{
    void order(int&, int&); //prototype
    int n1=99, n2=11; //this pair not ordered
    int n3=22, n4=88; //this pair ordered
    order(n1, n2); //order each pair of numbers
    order(n3, n4);
    cout << "n1=" << n1 << endl; //print out all numbers
    cout << "n2=" << n2 << endl;
    cout << "n3=" << n3 << endl;
    cout << "n4=" << n4 << endl;
    return 0;
}

void order(int& numb1, int& numb2) //orders two numbers
{
    if(numb1 > numb2) //if 1st larger than 2nd,
    {
        int temp = numb1; //swap them
        numb1 = numb2;
        numb2 = temp;
    }
}
```

Overloaded Functions

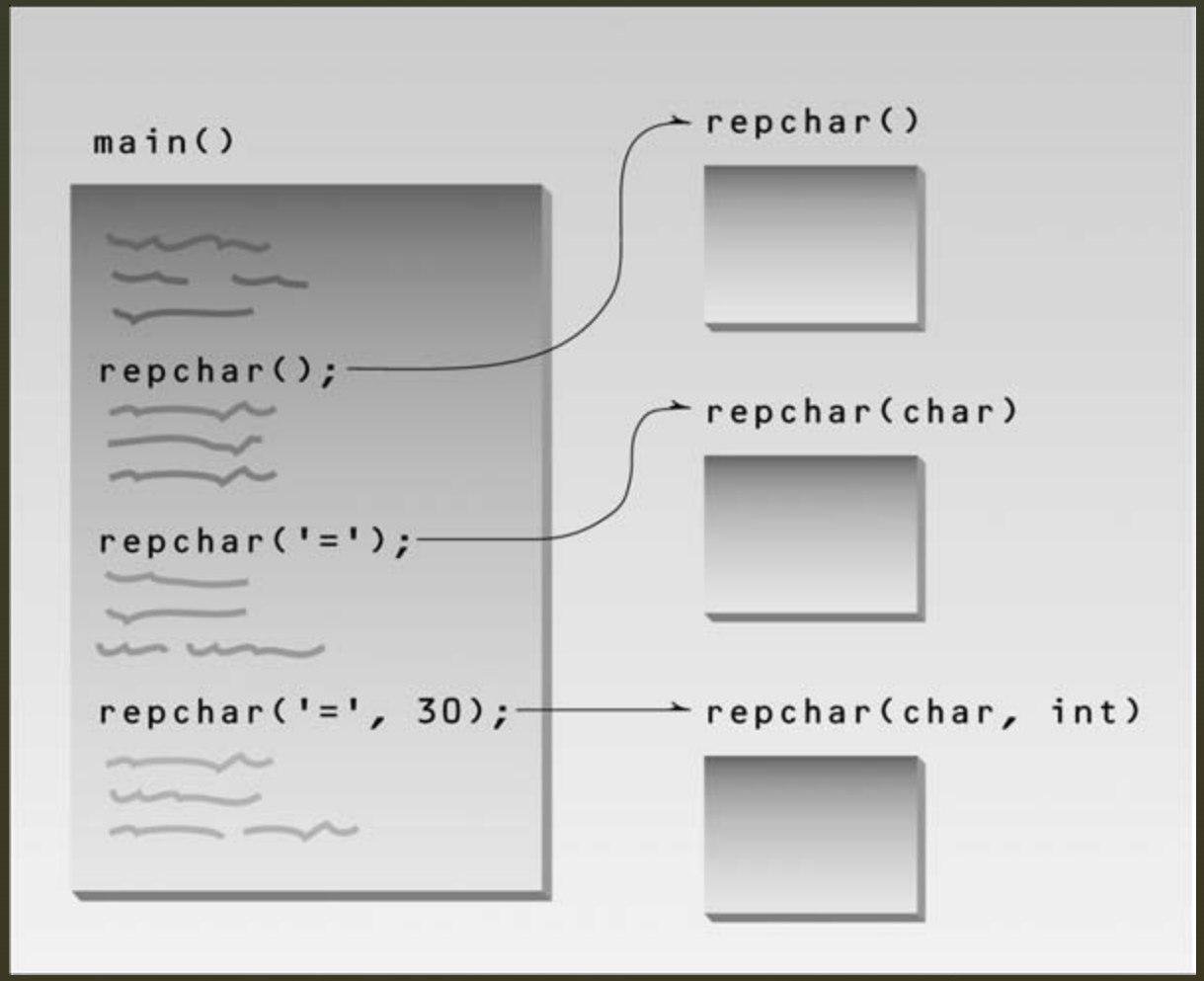
- An overloaded function appears to perform different activities depending on the kind of data sent to it.

Overloaded Functions

```
// demonstrates function overloading
#include <iostream>
using namespace std;

void repchar(); //declarations
void repchar(char);
void repchar(char, int);

int main()
{
    repchar();
    repchar('=');
    repchar('+', 30);
    return 0;
}
```

```
// repchar()
// displays 45 asterisks
void repchar()
{
    for(int j=0; j<45; j++) // always loops 45 times
        cout << '*'; // always prints asterisk
    cout << endl;
}

// repchar()
// displays 45 copies of specified character
void repchar(char ch)
{
    for(int j=0; j<45; j++) // always loops 45 times
        cout << ch; // prints specified character
    cout << endl;
}

// repchar()
// displays specified number of copies of specified character
void repchar(char ch, int n)
{
    for(int j=0; j<n; j++) // loops n times
        cout << ch; // prints specified character
    cout << endl;
}
```

Overloaded Functions



Recursion

The existence of functions makes possible a programming technique called *recursion.*

Recursion involves a function calling itself.

```
//calculates factorials using recursion
#include <iostream>
using namespace std;

unsigned long factfunc(unsigned long); //declaration
int main()
{
int n; //number entered by user
    unsigned long fact; //factorial
    cout << "Enter an integer: ";
    cin >> n;
    fact = factfunc(n);
    cout << "Factorial of " << n << " is " << fact << endl;
    return 0;
}

unsigned long factfunc(unsigned long n){
    if ((n==0)||(n==1))
        return 1;
    else
        return n * factfunc(n-1);
}
```

Inline Functions

- This kind of function is written like a normal function in the source file but compiles into inline code instead of into a function.
- The source file remains well organized and easy to read, since the function is shown as a separate entity.
- However, when the program is compiled, the function body is actually inserted into the program wherever a function call occurs.
- Functions that are very short, say one or two statements, are candidates to be inlined.

Inline Functions

It’s easy to make a function inline: All you need is the keyword inline in the function definition

```
// demonstrates inline functions
#include <iostream>
using namespace std;

// lbstokg()
// converts pounds to kilograms
inline float lbstokg(float pounds)
{
    return 0.453592 * pounds;
}

int main()
{
    float lbs;
    cout << "\nEnter your weight in pounds: ";
    cin >> lbs;
    cout << "Your weight in kilograms is " << lbstokg(lbs)
    << endl;
    return 0;
}
```

Scope and Storage Class

- The scope of a variable determines which parts of the program can access it, and its storage class determines how long it stays in existence.
- Two different kinds of scope are important here: local and file
  - Variables with local scope are visible only within a block.
  - Variables with file scope are visible throughout a file.
- A block is basically the code between an opening brace and a closing brace.

Scope and Storage Class

- There are two storage classes: automatic and static.
  - Variables with storage class automatic exist during the lifetime of the function in which they’re defined.
  - Variables with storage class static exist for the lifetime of the program.

Local Variables

- Variables may be defined inside main() or inside other functions; the effect is the same, since main() is a function.
- Variables defined within a function body are called *local variables* because they have local scope.
- However, they are also sometimes called automatic variables, because they have the automatic storage class.

Local Variables

- A local variable is not created until the function in which it is defined is called.
- Variables defined within a loop body only exist while the loop is executing.
- In the program fragment just given, the variables somevar and othervar don't exist until the somefunc() function is called.
- That is, there is no place in memory where their values can be stored; they are undefined.

```
void somefunc()
    {
        int somevar; //variables defined within
        float othervar; //the function body
        // other statements
    }
```

Local Variables

- When control is transferred to somefunc(), the variables are created and memory space is set aside for them.
- Later, when somefunc() returns and control is passed back to the calling program, the variables are destroyed and their values are lost.
- The name *automatic is used because the variables are automatically created when a function is* called and automatically destroyed when it returns.

```
void somefunc()
    {
        int somevar; //variables defined within
        float othervar; //the function body
        // other statements
    }
```

Local Variables

- The time period between the creation and destruction of a variable is called its lifetime (or sometimes its duration).
- The lifetime of a local variable coincides with the time when the function in which it is defined is executing.
- The idea behind limiting the lifetime of variables is to save memory space.
- If a function is not executing, the variables it uses during execution are presumably not needed.
- Removing them frees up memory that can then be used by other functions.

Scope

- A variable's *scope, also called visibility, describes the locations within a program from which it* can be accessed.
- It can be referred to in statements in some parts of the program; but in others, attempts to access it lead to an *unknown variable error message.*
- *The scope of a variable is that* part of the program where the variable is visible.
- Variables defined within a function are only visible, meaning they can only be accessed, from within the function in which they are defined.

Global Variables

- While local variables are defined within functions, global variables are defined outside of any function.
- A global variable is visible to all the functions in a file.
- A global variable is used when it must be accessible to more than one function in a program.

Static Local Variables

- A static local variable has the visibility of an automatic local variable.
- Its lifetime is the same as that of a global variable, except that it doesn't come into existence until the first call to the function containing it. Thereafter it remains in existence for the life of the program.
- Static local variables are used when it's necessary for a function to remember a value when it is not being executed; that is, between calls to the function.

Static Local Variables

```
// demonstrates static variables
#include <iostream>
using namespace std;

float getavg(float); //declaration

int main()
{
    float data=1, avg;
    while( data != 0 )
        {
        cout << "Enter a number: ";
        cin >> data;
        avg = getavg(data);
        cout << "New average is " << avg << endl;
        }
    return 0;
}

// getavg()
// finds average of old plus new data
float getavg(float newdata)
{
    static float total = 0; //static variables are initialized
    static int count = 0; // only once per program
    count++; //increment count
    total += newdata; //add new data to total
    return total / count; //return the new average
}
```

Static Local Variables

- Here's some sample interaction:

  Enter a number: 10

  New average is 10 ← total is 10, count is 1

  Enter a number: 20

  New average is 15 ← total is 30, count is 2

  Enter a number: 30

  New average is 20 ← total is 60, count is 3

- The static variables total and count in getavg() retain their values after getavg() returns, so they're available the next time it's called.

Exercises

- Raising a number n to a power p is the same as multiplying n by itself p times. Write a function called power() that takes a double value for n and an int value for p, and returns the result as a double value. Use a default argument of 2 for p, so that if this argument is omitted, the number n will be squared. Write a main() function that gets values from the user to test this function.
- Write a function called zeroSmaller() that is passed two int arguments by reference and then sets the smaller of the two numbers to 0. Write a main() program to exercise this function.

Any Questions?

The End

Contact: tsadjaidoo@knust.edu.gh

Office: Caesar Building, Room 413