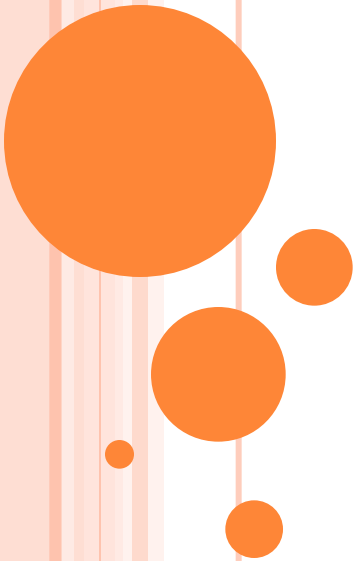
Functions & Parameter Passing



General vs. specialized functions

- Suppose you had a function called drawSquare, which, when called, always produces the following output:

```
* * * *
* * * *
* * * *
* * * *
```

- This is an example of a very specialized function, since it is only capable of drawing a very specific figure

A GENERAL SQUARE FUNCTION

- Contrast the previous example with a similar function that takes an argument specifying the square's size; for example, drawSquare(2) would produce the following output:

```
* *
* *
```

- An even more general function might take an argument specifying the character to draw with; for example, drawSquare(2, '$') would produce:

```
$ $
$ $
```

Specifying function requirements

- We can see that a more general function might require arguments; but how do we know what the requirements are?
- A general method for specifying the requirements for a function call is called function declaration or function *prototyping*

Function prototype

- Declaration statement for function
- Provides information necessary to call function
  - Data type of function's return value (if any)
  - Data type(s) of any required argument(s), listed in the order required: this is called the *parameter list*
- Usually appears above main() or in a header file
- Function prototypes are declaration *statements* – each one ends with a semicolon

Function prototype examples

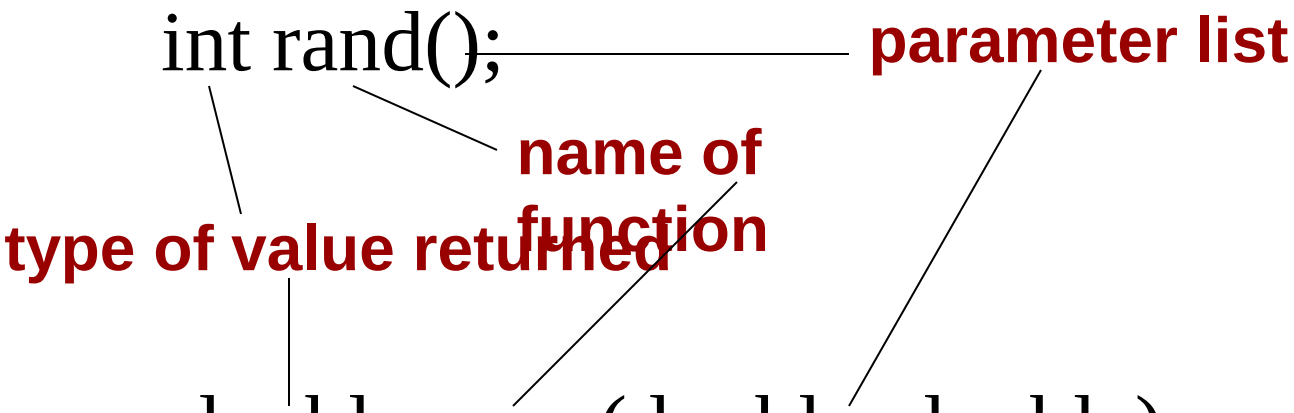
- The function prototypes for some familiar functions are listed below:

```
int rand(); // appears in the stdlib.h header file
double sqrt (double); // appears in math.h
double pow (double, double); // also in math.h
```

Function prototypes

- A function prototype includes the following parts:
  - The function's return type; can be any data type, including void (meaning no return value)
  - The function name (must be a valid C++ identifier – same rules apply for functions and variables)
  - The function's parameter list, which specifies the data type of each required argument, and optionally includes a name for each parameter

Examples



Note: a function's parameter list may, but does not have to, include a name for each parameter; the parameters are not named in the examples above

More examples

- Previously, we considered 3 possible versions of a drawSquare function; the prototypes for each version are shown below:

```
void drawSquare();
    // draws a 4x4 square made of asterisks
void drawSquare(int);
    // draws a square of the specified size, made of asterisks
void drawSquare(int size, char pixel);
    // draws a square of the specified size using the specified
    // picture element
```

- Note that these are examples of overloaded functions – their prototypes differ only in their parameter lists

Function definition

- A function prototype merely declares a function, specifying requirements for a function call
- A function call invokes a function
- A function definition *implements* the function; that is, it contains the code that will be executed when the function is called

Function definition

- A function definition has two parts:
  - The heading, which supplies the same information as the function prototype
  - The body, which implements the function

Function definition

```
void drawSquare()
{
        for (int x=0; x<4; x++)
        {
                for (int y=0; y<4; y++)
                        cout << "* ";
                cout << endl;
        }
}
```

Function heading

Function body

Notes on function definition

- Although the function's heading contains the same information as the prototype, it appears in slightly different form:
  - There is no semicolon at the end of the function heading
  - If there are any parameters, they must be named, even if they were not named in the prototype; if they *were* named in the prototype, the names must match *exactly*

Example

The second drawSquare prototype didn't specify a name for its parameter; the function definition must provide a parameter name

```
void drawSquare(int);           // prototype
void drawSquare(int size)       // definition
{
        for (int x=0; x<size; x++)
        {
                for (int y=0; y<size; y++)
                        cout << "* ";
                cout << endl;
        }
}
```

Example

The third drawSquare function's prototype specified names for both parameters; the function definition must use exactly the same names

```
void drawSquare(int size, char pixel);          // prototype

void drawSquare(int size, char pixel)           // definition
{
        for (int x=0; x<size; x++)
        {
                for(int y=0; y<size; y++)
                        cout << pixel << ' ';   // output character,
                cout << endl;                   // then space
        }
}
```

Value-returning functions

- The function definition examples we have seen so far have all been void functions
- A value-returning function has an additional requirement in its definition; such a function must include a return statement
- The return statement specifies what value a function returns; this is the value that a function call represents when used in an expression

EXAMPLE

The function below returns the cube of its argument:

```
double cubeIt (double value)
{
        double cube;
        cube = value * value * value;
        return cube;
}
```

Another way to write the same function:

```
double cubeIt (double value)
{
        return value * value * value;
}
```

Void functions and return statements

- Most void functions can be written without return statements
- However, sometimes it's handy to be able to use a return statement in a void function
- If desired, the following statement can be used in a void function:

```
return;
```

Example

- Suppose you are writing a function that displays a menu of choices for the user, then performs some action based on the user's choice
- If the user chooses a valid menu item, a function is called to perform the chosen task
- If an invalid item is chosen, or the user chooses to quit, the function returns without performing any further action
- The next slide shows an example of such a function

Example

```
void doSomething()
{
        int choice = getChoice();    // calls function that displays menu &
        switch(choice)               // gets user's preference
        {
                case 1:
                        drawSquare();
                        break;
                case 2:
                        drawCircle();
                        break;
                default:
                        return;
        }
}
```

Example: safe input of numbers

- We have already discussed the problems involved with extraction of numeric data in interactive programs
  - User can, accidentally or deliberately, type bad data
  - When bad data are encountered, the input stream shuts down, and no more data can be read
- We have also seen an effective solution to this problem; instead of reading numbers directly, read characters and convert them to numbers

Safe input example

The code below provides a safe input method for a positive integer:

```
int number = 0;
char c;
cin.get(c);
while(c != '\n')
{
        if(isdigit(c))
        {
                number = number * 10;
                number = number + (int)(c – '0');
        }
        cin.get(c);
}
```

Safe input example

- The code on the previous slide is a good example of a useful routine that we might want to use several times within a program
- Rather than copy and paste this code wherever we need to input a number, we can just encapsulate the code within a function, then call the function whenever we need to read a number

Function definition placement

- Function definitions can be placed either before or after the main function in a program, but not within the block that defines main()
- Function prototypes should always appear before main(), and before any other function definitions
- If function definitions are placed above main(), then function prototypes may be omitted; however, you need to know how to read and write prototypes, so it is good practice to use them

Header files

- Functions are often written to be generally useful
- If a function works well to solve a problem in one program, it would probably solve a similar problem in a different program
- The use of *header files* can make your code more re-useable

Header files contain declarations, such as the following:

- **function prototypes like**
  double sqrt(double);
- **named constants like**
  const int INT_MAX = 32767;
- **classes like**
  string, ostream, istream
- **objects like**
  cin, cout

Header files

- Header files also contain documentation; for example, each function prototype in a header file would be accompanied by a comment explaining what the function does
- A header file is a text file written in C++, but the file extension is .h instead of .cpp

Implementation files

- When header files contain function prototypes, a corresponding *implementation file* is required
- The implementation file contains the definition(s) of the function(s) declared in the header file
- The implementation file has the same name as the header file, but has a .cpp extension
- An implementation file can be compiled, but it is not a program, because it doesn't contain a main() function

Using your own libraries

- To create your own library of functions, start by placing your function prototype(s) in a header file
- Define your function(s) in an implementation file
- Write your program, including main(), in a separate file
- The next three slides illustrate this process

Header file

- The text of the file safenum.h is shown below:

```
#ifndef SAFENUM_H
#define SAFENUM_H
int getNum();
// Reads and returns a number from the standard input
    stream; ignores
// extraneous characters.  If no digits are entered, returns 0
#endif
```

- The statements that start with the # character are preprocessor directives that indicate that the code between the #ifndef directive and the #endif directive should only be included in a program if it hasn't been included already; this prevents a possible linker error

Implementation file

- The implementation file contains the definition(s) of function(s) declared in the header file; it should contain any preprocessor directives needed for the function code, as well as a preprocessor directive indicating inclusion of the header file
- In this example, the syntax would be:

  #include “safenum.h”
- This syntax assumes that the header file and implementation file are in the same folder on your disk; if they are not, the full path to the file would need to be included in the quoted string

Program file

- The program file would contain main(), and would have a name that is different from the header and implementation files
- As well as any other necessary preprocessor directives, the program file would need an include directive for the new header; in this example:

  #include “safenum.h”
- Again, the full path would be necessary if the header is in a different disk folder

Linking your header files

- The process described in the last several slides is the correct one in the most general sense, independent of any particular compiler or programming environment
- In most IDEs (such as dev or Visual C++), however, there is at least one additional step required to properly link your header file, implementation file and program – this involves creating a project file
- This is an environment-specific task; it will vary according to which IDE you're using

Parameter passing methods

- In all the function prototype and definition examples we have seen thus far, parameters are specified as if they were simple variables, and the process of parameter passing is comparable to the process of assigning a value to a variable:
  - Each parameter is assigned the value of its corresponding argument
  - Although the value of a parameter may change during the course of a function, the value of the corresponding argument is not affected by the change to the parameter

Passing by value

- The process described on the previous slide, and illustrated in all examples thus far, is called ***passing by value*** - this is the default method for parameter passing
- When arguments are passed by value:
  - a ***copy*** of each argument ***value*** is passed to its respective parameter
  - the parameter is stored in a separate memory location from the storage location of the argument, if the argument has one
  - Any valid expression can be used as an argument

Example

The program below illustrates what happens when arguments are passed by value. A tracing of the changes in the program's variables is shown on the right.

```
int multiply (int, int);

int main()
{
        int a, b, c;
        a = 2;
        b = 5;
        c = multiply(a,b);
        a = multiply(b,c);
        return 0;
}

int multiply (int x, int y)
{
        x = x * y;
        return x;
}
```

a	b	c	x	y
2	5	10	2	5
50			10	10
			5	
			50	

When the program ends, the variables remaining in memory have the values shown in red

Limitations of pass by value

- Recall that a function can have either one return value or no return value
- If we want a function's action to affect more than one variable in the calling function, we can't achieve this goal using return value alone – remember, our options are one or none
- The next example illustrates this problem

Example – swap function

Suppose we want to write a function that swaps two values: that is, value a is replaced by value b, and value b is replaced by the original value of a. The function below is an attempt to achieve this goal.

```
void swap (int x, int y)
{
        int tmp = x;
        x = y;
        y = tmp;
}
```

Output:
Before swap, a=2 and b=6
After swap, a=2 and b=6

The function appears to work correctly. The next step is to write a program that calls the function so that we can test it:

```
int main()
{
        int a=2, b=6;
        cout << “Before swap, a=” << a << “ and b=”
             << b << endl;
        swap(a,b);
        cout << “After swap, a=” << a << “ and b=”
             << b << endl;
        return 0;
}
```

What went wrong?

- In the swap function, parameters x and y were passed the values of variables a and b via the function call swap(a, b);
- Then the values of x and y were swapped
- When the function returned, x and y were no longer in memory, and a and b retained their original values
- Remember, when you pass by value, the parameter only gets a copy of the corresponding argument; changes to the copy don’t change the original

Building a Better Swap Function: Introducing Reference Parameters

- C++ offers an alternative parameter-passing method called ***pass-by-reference***
- When we pass by reference, the data being passed is the ***address*** of the argument, not the argument itself
- The parameter, rather than being a separate variable, is a reference to the same memory that holds the argument – so any change to the parameter is also a change to the argument

Revised swap function

We indicate the intention to pass by reference by appending an ampersand (&) to the data type of each reference parameter. The improved swap function illustrates this:

```
void swap (int& x, int& y)
{
        int tmp = x;
        x = y;
        y = tmp;
}
```

The reference designation (&) means that x and y are not variables, but are instead references to the memory addresses passed to them

If we had the same main program as before, the function call:

swap(a,b);

indicates that the first parameter, x, is a reference to a, and the second parameter, y, is a reference to b

How pass-by-reference works

- In the example on the previous slide, x and y referenced the same memory that a and b referenced
- Remember that variable declaration does two things:
  - Allocates memory (one or more bytes of RAM, each of which has a numeric address)
  - Provides an identifier to reference the memory (which we use instead of the address)
- Reference parameters are simply additional labels that we temporarily apply to the same memory that was allocated with the original declaration statement
- Note that this means that arguments passed to reference parameters must be variables or named constants; in other words, the argument must have its own address

EXAMPLE

Earlier, we looked at a trace of the program below, on the left. The program on the right involves the same function, this time converted to a void function with an extra reference parameter.

```
int multiply (int, int);
int main()
{
        int a, b, c;
        a = 2;
        b = 5;
        c = multiply(a,b);
        a = multiply(b,c);
        return 0;
}

int multiply (int x, int y)
{
        x = x * y;
        return x;
}
```

```
void multiply (int, int, int&);
int main()
{
        int a, b, c;
        a = 2;
        b = 5;
        multiply(a,b,c);
        multiply(b,c,a);
        return 0;
}

void multiply (int x, int y, int& z)
{
        z = x * y;
}
```

Using I/O stream objects as arguments

- We have seen at least one function that takes an input stream as one of its arguments: getline
- As you know, getline takes two arguments: an input stream object and a string variable
- Based on what you know now, what parameter-passing method does getline use for the string variable?

Using I/O stream objects as arguments

- The prototype for getline looks something like this:

```
void getline(istream&, string&);
```

- The stream variable, like the string variable, is passed by reference
- Stream variables are always passed by reference; you will get a compiler error if you attempt to write a function with a value parameter of a stream type

Example

The following function opens an input file using the file name specified by the program's user. Since the file stream is passed by reference, the file passed to the function will be open in both the current function and the calling function.

```
bool openInputFile (ifstream&);
// attempts to open the input file specified by the user; returns true if open succeeds,
// false if open fails
```

```
bool openInputFile (ifstream& inf)
{
        string fileName;
        cout << "Enter name of input file to open: ";
        cin >> fileName;
        return inf.open(fileName.c_str());
        // c_str function converts string to format required by open()
        // function returns the result of the attempt to open the file (true or false)
}
```

Making getNum more versatile

- Previously, we examined a function that provided a “safe” input mechanism for integer numbers
- The function read a set of characters from the standard input stream, converting digit characters into their int equivalents and accumulating an int result, which the function returned
- The function would be more versatile if it could read input from any input stream, not just cin; a few minor modifications make this possible

Revisions to getNum

Prototype

Old: int getNum();

New: int getNum(istream&);

Function calls

Old: num = getNum();

New: num = getNum(cin); or:

```
ifstream infile;
openInputFile(infile);
// using function from a few slides
// back, and assuming success:

num = getNum(infile);
```

Function definition

```
int getNum(istream& ins)
{
        int num=0;
        char c;

        // skipping parts not changed

        ins.get(c); // old: cin.get(c);
        while (c != '\n')
        {
                …
                ins.get(c);
        }
         // skipping parts not changed
        return num;
}
```

Scope of identifiers

- The part of a program where a specific identifier can be recognized is called the ***scope*** of the identifier
- Scope in a program is similar to fame in the real world
  - If scope is ***local***, then the identifier can only be used within its local area; outside this area, the identifier is unknown. Like me (and probably you), this identifier is not famous.
  - If scope is ***global***, the identifier can be used anywhere in the program, because a global identifier is “famous”

Determining the scope of an identifier

- The scope of an identifier is determined by the location of its declaration statement
- An identifier that is declared within a block (inside a function, for example) is local to that block
- An identifier that is declared outside any block is global to the file in which it is declared

Examples

- The following identifiers are typically declared in global space:
  - Function names
  - Named constants
- Variables are almost always (you can forget about “almost” for the duration of this course) declared locally to functions
- Some identifiers may be more local yet; the control variable of a for loop is often declared at the beginning of the loop; its scope is limited to the loop itself

EXAMPLE

Determine the scope of each identifier in the following program

```
const double PI = 3.14159;
double area (double radius);
double circumference (double r);

int main()
{
        double radius;
        for (int x=0; x<5; x++)
        {
                cout << “Give me #: ”;
                cin >> radius;
                cout << “Area of circle ”
                << x << “ is ” << area(radius)
                << endl << “Circumference is ”
                << circumference(radius) <<
                endl;
        }
        return 0;
}
```

```
double area (double radius)
{
        double a;
        a = pow(radius, 2);
        a = a * PI;
        return a;
}

double circumference(double r)
{
        return 2 * PI * r;
}
```

Detailed Scope Rules

1. Function ***name*** has global scope if declared outside any other function.
2. Function ***parameter*** scope is identical to scope of a ***local*** variable declared in the outermost block of the function body.
3. Global scope extends from ***declaration*** to the ***end of the file***, except as noted on the next slide

Detailed scope rules, continued

4. Local scope extends from ***declaration*** to the ***end of the block*** where identifier is declared. This scope includes any nested blocks, except as noted in rule 5.
5. An identifier's scope ***does not*** include any nested block that contains a locally declared identifier with the same name (local identifiers have name precedence).

How Compiler Determines Scope

- When an expression refers to an identifier, the compiler first checks the local declarations.
- If the identifier isn't local, compiler works outward through each level of nesting until it finds an identifier with same name. There it stops.
- Any identifier with the same name declared at a level further out is never reached.
- If compiler reaches global declarations and still can't find the identifier, an error message results.

Name Precedence (or Name Hiding)

- When a function declares a local identifier with the same name as a global identifier, the local identifier takes precedence within that function
- So far as the code in the body of the function is concerned, the global identifier doesn't exist; the local identifier takes precedence, effectively hiding the global identifier (your friends are talking about you, not the president)