

Structured Programming

Control Structures

- ◆ In this course, we teach *structured programming*, a disciplined approach to programming that results in programs that are easy to read and understand and are less likely to cause errors.
- ◆ We follow accepted program style guidelines (such as using meaningful names for identifiers) to write code that is adequately documented with comments and is clean and readable.

- ◆ Obscure tricks and programming shortcuts are strongly discouraged.
- ◆ *Program maintenance* involves modifying a program to remove previously-undetected bugs and to keep it up-to-date.
- ◆ It is not uncommon to maintain a program for five years or more, often after the programmers who originally coded it have left the company or have moved onto other positions.

- ◆ Structured programming uses *control structures* to control the flow of statement execution in a program.

- ◆ The control structures of a programming language let us combine individual statements into a single program entity with one entry point and one exit point.

- ◆ We can then write a program as a sequence of control structures rather than a sequence of individual statements.

◆ There are three categories of control structures:

1. **Sequence**

2. **Selection (Decision)**

3. **Iteration (Repetition)**

- ◆ In C++, sequence is illustrated by *compound statements*.
- ◆ A compound statement is a group of statements enclosed by braces.
- ◆ Control flows from statement 1 to statement 2 and so on.

Structure Charts

- ◆ One of the most fundamental ideas in problem solving is dividing a problem into subproblems and solving each subproblem independently.
- ◆ In attempting to solve a subproblem at one level, we often introduce new subproblems at a lower level.
- ◆ This is similar to refining an algorithm.
- ◆ We can use a diagram called a *structure chart* to show the relationship between a problem and its subproblems.

- ◆ As we trace down this diagram, we go from an abstract original problem to a more detailed subproblem.

- ◆ Structure charts are intended to show the relationship between subproblems.

- ◆ The algorithm (not the structure chart) shows the order in which we must carry out each step to solve the problem.

- ◆ A structure chart proceeds from the original problem at the top level down to its detailed subproblems at the bottom level. This is called a *top-down* approach.

Functions

- ◆ A C++ *function* is a grouping of program statements into a single program unit.
- ◆ Just like the **cmath** and **iomanip** functions that we have already called in programs, each C++ function that we write can be activated through the execution of a function call.
- ◆ Just like other identifiers in C++, a function must be declared before it can be referenced in a program body.

- ◆ The definition of any function is similar to the definition of the **main** function -- a *prototype* followed by a *function body*.
- ◆ The function body always starts with { and ends with } -- i.e., a compound statement.
- ◆ Each function body may contain declarations for its own variables.
- ◆ These identifiers are considered *local* to the function; in other words, they can be referenced only within the function.
- ◆ Function declarations must appear before any calls to the functions.

- ◆ One way to declare a function is to give its full definition.

- ◆ The other way is just to use a *prototype*.

- ◆ A prototype tells us three pieces of information:
 1. The name of the function (its *identifier*);
 2. The number of inputs, and the name and data type of each input (the *arguments*);
 3. The data type of its output (the *return value*).

Function Declaration / Prototype Syntax:

```
return-type  
function-name ( arg1-type arg1-name,  
                arg2-type arg2-name, etc. )
```

Examples:

```
int main( ) ;
```

```
void draw_circle( ) ;
```

```
double pow( double base, double exponent ) ;
```

```
void print_3_ints( int i1, int i2, int i3 ) ;
```

Function Definition Syntax:

```
function-prototype
{
    local declarations
    executable statements
    return statement
}
```

Interpretation . . .

The function **function-name** is defined, returning a value of type **return-type**, and accepting zero or more arguments of type **arg1-type**, **arg2-type**, etc. Any arguments are called **arg1-name**, **arg2-name**, etc.

Any identifiers that are declared in the optional **local declarations** are defined only during the execution of the function and can be referenced only within the function.

The **executable statements** of the function body describe the data manipulation to be performed by the function.

The **return statement** at the end of the function body transfers control back to the calling function (like `main`) and, if the return-type from the prototype is not **void**, return a value back to the caller.

Placement of Function Declarations in a Program

- ▶ C++ provides more than one way to include function declarations in a program.
- ▶ Since function declarations must appear before any calls to the functions, we can place the definitions of the functions between the preprocessor directives and the definition of the main function.
- ▶ We can place only function declarations (prototypes) before the definition of main, and then place the function definitions after the end of main.

- ▶ The order in which the functions are declared and/or defined is not important.

- ▶ The order in which each function is executed is determined by the order of the function call statements in the body of the main function.

- ▶ Each function should begin with a comment that describes its purpose. If the function subprograms were more complex, we would include comments on each major algorithm step just as we do in function main.

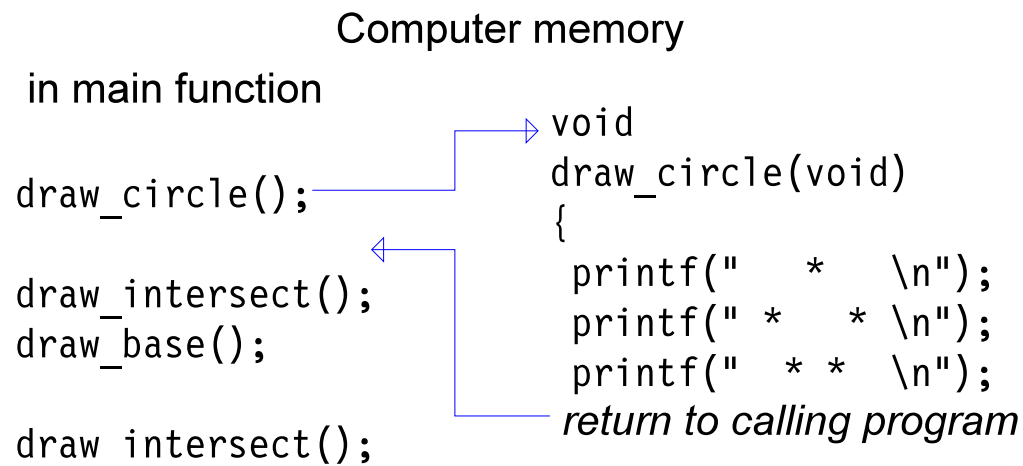
- ▶ The compiler must know about the function subprogram declarations before it translates the main function where the subprogram functions are called.

- ▶ As it translates each function subprogram, where the compiler reaches the end of the program body, it inserts a machine-language statement that causes a *transfer of control* back from the function to the calling statement.

- ▶ In the main function body, the compiler translates a function call statement as a transfer of control to the function.

Example:

Flow of Control Between Function main and Function Subprogram



Procedural Abstraction

- ◆ One important advantage of subprograms is that their use allows us to remove code from the main function.
- ◆ The code that we remove represents the detailed procedure for solving a subproblem.
- ◆ Since these details are provided in the function subprograms are not in the main function, we can write the main function as soon as we have specified the initial algorithm.

- ◆ We can delay writing the function for an algorithm step until we have finished refining that step.
- ◆ This approach to program design is called *procedural abstraction*.
- ◆ Procedural abstraction lets us defer implementation details and write our program in logically independent sections. (This is the same way that we develop our solution algorithm).
- ◆ Another advantage of using functions is that they may be executed more than once in a program. Also, once you have written and tested a function subprogram, you may use it in other programs or other functions.