



Devi Ahilya Vishwavidyalaya

# Programming With C



# What is a Computer Program?

A **program** is a **set of step-by-step instructions** to the computer telling it to carry out the **tasks** that you want it to do to produce the **results** you want.



# What is Programming?

- Programming consists of two distinct steps:
- **algorithmic design** (the problem solving stage, analogous to the work of an architect designing a building)
- **coding** (the construction phase)



# Levels of Programming Languages

- Machine language
- Assembly Language
- High Level Languages



# Machine Language

- Actual binary code that gives basic instructions to the computer.
- These are usually simple commands like adding two numbers or moving data from one memory location to another.
- Different for each computer processor



# Assembly Language

- A way for humans to program computers directly without memorizing strings of binary numbers.
- There is a one-to-one correspondence with machine code.
  - For example ADD and MOV are mnemonics for addition and move operations that can be specified in single machine language instructions.

Different for each computer processor



# High-level language

- Permits humans to write complex programs without going step-by step.
- High-level languages include Pascal, FORTRAN, Java, Visual Basic, and many more.
- One command in a high-level language may translate to tens of machine language instructions.



# Translation

Computers can only run **machine language** programs directly.

**Assembly language** programs are assembled, or translated into machine language.

Likewise, programs written in high-level languages, like Java, must also be translated into machine language before they can be run. To do this translation **compile** a program.





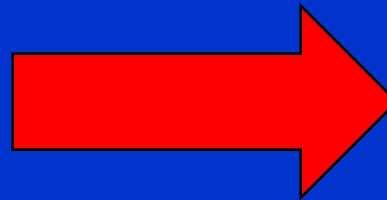
# Translation

```
#include <stdio.h>

int main()
{
    printf("Hello World");

    return 0;
}
```

**Source code**



```
10100110 01110110
00100110 00000000
11111010 11111010
01001110 10100110
11100110 10010110
11001110 00101110
10100110 01001110
11111010 01100110
01001110 10000110
```

*etc...*

**Executable code**

- **Compilers and linkers** translate a high level program into executable machine code



# Structured Programming

- **STRUCTURED PROGRAMMING**  $\equiv$  A technique for organizing and coding computer programs in which a **hierarchy of modules** is used, each having a **single entry** and a **single exit** point, and in which **control** is **passed downward** through the structure **without UNconditional** branches to higher levels of the structure. **Three** types of control flow are used: (1) **sequential**, (2) **selection**, and (3) **iteration**.



# Programming language C

- C is a general purpose programming language.
- C is a middle level language.
- C is a structured language.



# Programming language C

Why C is called “a middle level language”?

C contains the features of high level language portability — it is easy to adapt software written for one type of computer to another type. the functionality low level language.

- operators such as &, |,>,< etc. simulate to low level instruction codes.
- Direct manipulation of bits, bytes and addresses.



# Writing C Programs

- A programmer uses a **text editor** to create or modify files containing C code.
- Code is also known as **source code**.
- A file containing source code is called a **source file**.
- After a C source file has been created, the programmer must **invoke the C compiler** before the program can be **executed (run)**.



# Invoking the tcc Compiler

At the prompt, type

```
tcc pgm.c
```

where *pgm.c* is the C program source file.

There is a better way use of IDE instead of command.



# The Result : `pgm.obj`, `pgm.exe`

- If there are no errors in `pgm.c`, this command produces an **executable file**, which is one that can be executed (run).
- The `tcc` compiler puts `exe` extension of the executable file. Also the `obj` file contains the machine level code.
- To execute the program, at the prompt, type  
`pgm.exe`
- Although we call this process “compiling a program,” what actually happens is more complicated.



# 3 Stages of Compilation

## Stage 1: Preprocessing

- Performed by a program called the **preprocessor**
- Modifies the source code (in RAM) according to **preprocessor directives (preprocessor commands)** embedded in the source code
- Strips comments and white space from the code
- The source code as stored on disk is not modified.





# 3 Stages of Compilation (con't)

## Stage 2: **Compilation**

- Performed by a program called the **compiler**
- Translates the preprocessor-modified source code into **object code (machine code)**
- Checks for **syntax errors** and **warnings**
- Saves the object code to a disk file, if instructed to do so (we will not do this).
  - If any compiler errors are received, no object code file will be generated.
  - An object code file will be generated if only warnings, not errors, are received.



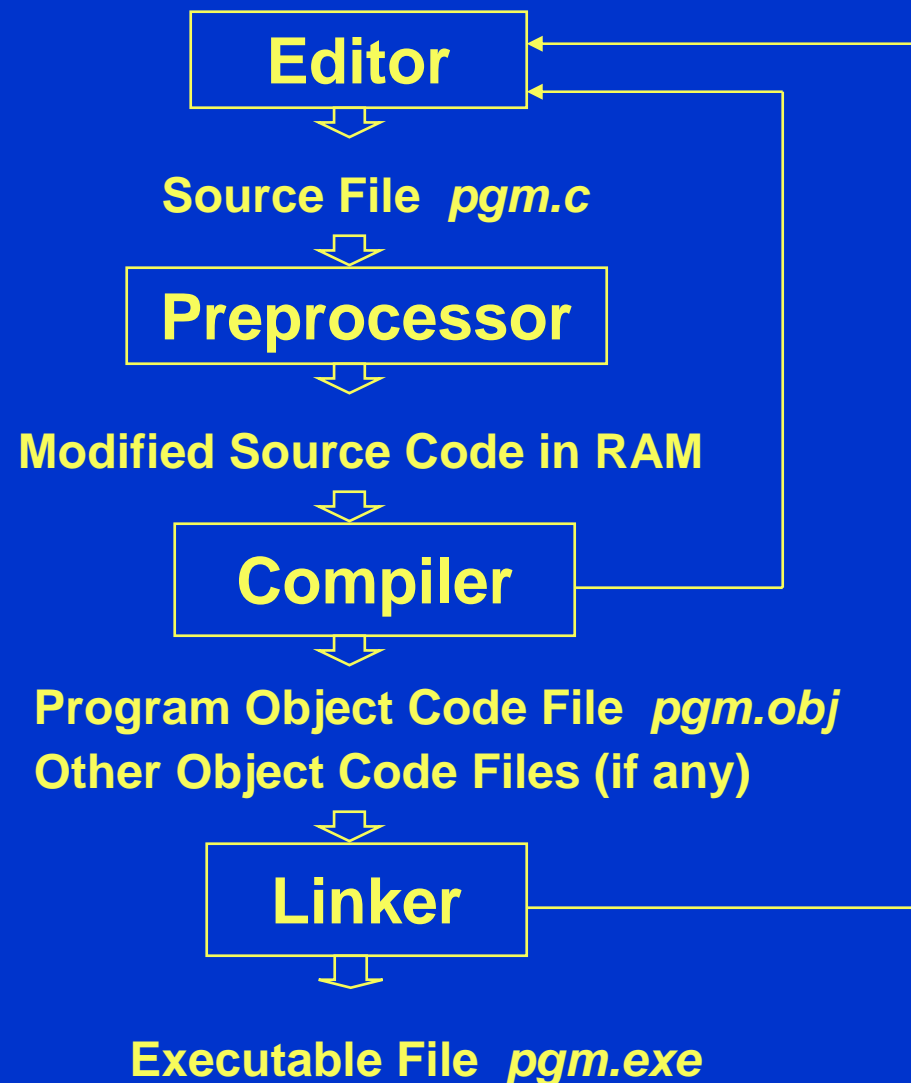
# 3 Stages of Compilation (con't)

## Stage 3: Linking

- Combines the program object code with other object code to produce the executable file.
- The other object code can come from the **Run-Time Library**, other libraries, or object files that you have created.
- Saves the executable code to a disk file. On the Linux system, that file is called **a.out**.
  - If any linker errors are received, no executable file will be generated.



# Program Development





# A Simple C Program

```
/* Filename:    hello.c
   Author:      Brian Kernighan & Dennis Ritchie
   Date written: ??/?/1978
   Description: This program prints the greeting
                "Hello, World!"

*/

#include <stdio.h>

int main ( void )
{
    printf ( "Hello, World!\n" ) ;
    return 0 ;
}
```



# Anatomy of a C Program

*program header comment*

*preprocessor directives (if any)*

```
int main ( )  
{  
    statement(s)  
    return 0 ;  
}
```



# Program Header Comment

- A **comment** is descriptive text used to help a reader of the program understand its content.
- All comments must begin with the characters `/*` and end with the characters `*/`
- These are called **comment delimiters**
- The program header comment always comes first.



# Preprocessor Directives

- Lines that begin with a # in column 1 are called **preprocessor directives (commands)**.
- Example: the **#include <stdio.h>** directive causes the preprocessor to include a copy of the standard input/output header file **stdio.h** at this point in the code.
- This header file was included because it contains information about the **printf ( )** function that is used in this program.



# stdio.h

- When we write our programs, there are libraries of functions to help us so that we do not have to write the same code over and over again.
- Some of the functions are very complex and long. Not having to write them ourselves make it easier and faster to write programs.
- Using the functions will also make it easier to learn to program!





# int main ( void )

- Every program must have a **function** called **main**. This is where program execution begins.
- main() is placed in the source code file as the first function for readability.
- The **reserved word** “int” indicates that main() **returns** an integer value.
- The parentheses following the reserved word “main” indicate that it is a function.
- The reserved word “void” means nothing is there.



# The Function Body

- A left brace (curly bracket) -- { -- begins the **body** of every function. A corresponding right brace -- } -- ends the function body.
- The style is to place these braces on separate lines in column 1 and to indent the entire function body 3 to 5 spaces.



# `printf (“Hello, World!\n”) ;`

- This line is a **C statement**.
- It is a **call** to the function **printf ( )** with a single **argument (parameter)**, namely the **string** “Hello, World!\n”.
- Even though a string may contain many characters, the string itself should be thought of as a single quantity.
- Notice that this line ends with a semicolon. All statements in C end with a semicolon.



# return 0 ;

- Because function main() returns an integer value, there must be a statement that indicates what this value is.
- The statement

return 0 ;

indicates that main() returns a value of zero to the operating system.

- A value of 0 indicates that the program successfully terminated execution.
- Do not worry about this concept now. Just remember to use the statement.



# Another C Program

```
/******
```

```
** File: proj1.c
```

```
** Author: _____
```

```
** Date: 9/15/01
```

```
** E-mail: _____
```

```
**
```

```
** This program prompts the user for two integer values then displays  
** their product.
```

```
**
```

```
*****/
```



# Another C Program (con't)

```
#include <stdio.h>

int main( void )
{
    int value1, value2, product ;
    printf("Enter two integer values: ") ;
    scanf("%d%d", &value1, &value2) ;
    product = value1 * value2 ;
    printf("Product = %d\n", product) ;
    return 0 ;
}
```



# Tokens

- The smallest element in the C language is the token.
- It may be a single character or a sequence of characters to form a single item.



# Tokens are:

- Tokens can be:
  - Numeric constants
  - Character constants
  - String constants
  - Keywords
  - Names (identifiers)
  - Punctuation
  - Operators





# Numeric Constants

- Numeric constants are an uninterrupted sequence of digits (and may contain a period). They never contain a comma.
- Examples:
  - 123
  - 98.6
  - 1000000



# Character Constants

- One character from a defined character set.
- Surrounded on the single quotation mark.
- Examples:
  - 'A'
  - 'a'
  - '\$'
  - '4'



# String Constants

- A sequence characters surrounded by double quotation marks.
- Considered a single item.
- Examples:
  - “DAVV”
  - “I like ice cream.”
  - “123”
  - “DHOOM-2”
  - “car”



# Keywords

- Sometimes called reserved words.
- Are defined as a part of the C language.
- Can not be used for anything else!
- Examples:
  - int
  - while
  - for



# Names

- Sometimes called identifiers.
- Can be of anything length, but on the first 31 are significant (too long is as bad as too short).
- Are case sensitive:
  - abc is different from ABC
- Must begin with a letter and the rest can be letters, digits, and underscores.
- There can be one exception to beginning letter that variable name can start with underscore( `_` ) but it is used by C library.



# Punctuation

- Semicolons, colons, commas, apostrophes, quotation marks, braces, brackets, and parentheses.
- ; : , ‘ “ [ ] { } ( )



# Operators

- There are operators for:
  - assignments
  - mathematical operations
  - relational operations
  - Boolean operations
  - bitwise operations
  - shifting values
  - calling functions
  - subscripting
  - obtaining the size of an object
  - obtaining the address of an object
  - referencing an object through its address



# What Are Variables in C?

- **Variables** in C have the same meaning as variables in algebra. That is, they represent some unknown, or variable, value.

$$x = a + b$$

$$z + 2 = 3(y - 5)$$

- Remember that variables in **algebra** are represented by a single alphabetic character.





# Naming Variables

- Variables in C may be given representations containing multiple characters. But there are rules for these representations.
- Variable names (identifiers) in C
  - May only consist of letters, digits, and underscores
  - May be as long as you like, but only the first 31 characters are significant
  - May not begin with a digit
  - May not be a C **reserved word (keyword)**



# Reserved Words (Keywords) in C

• auto	break	int	long
• case	char	register	return
• const	continue	short	signed
• default	do	sizeof	static
• double	else	struct	switch
• enum	extern	typedef	union
• float	for	unsigned	void
• goto	if	volatile	while



# Naming Conventions

- C programmers generally agree on the following **conventions** for naming variables.
  - Begin variable names with lowercase letters
  - Use **meaningful** identifiers
  - Separate “words” within identifiers with underscores or mixed upper and lower case.
  - Examples: surfaceArea    surface\_Area  
   surface\_area
  - Be consistent!



# Naming Conventions (con't)

- Use all uppercase for **symbolic constants** (used in **#define** preprocessor directives).
- Note: symbolic constants are not variables, but make the program easier to read.
- Examples:

```
#define PI 3.14159
```

```
#define AGE 52
```



# Case Sensitivity

- **C is case sensitive**
  - It matters whether an **identifier**, such as a variable name, is uppercase or lowercase.
  - Example:

area

Area

AREA

ArEa

are all seen as different variables by the compiler.



# Which Are Legal Identifiers?

AREA	area_under_the_curve
3D	num45
Last-Chance	#values
x_yt3	pi
num\$	%done
lucky***	



# Declaring Variables

- Before using a variable, you must give the compiler some information about the variable; i.e., you must **declare** it.
- The **declaration statement** includes the **data type** of the variable.
- They must be declared just after the start of block (i.e. start of a function) and before any other executable statement.
- Examples of variable declarations:

```
int meatballs ;  
float area ;
```



# Declaring Variables (con't)

- When we declare a variable
  - Space is set aside in memory to hold a value of the specified data type
  - That space is associated with the variable name
  - That space is associated with a unique **address**
- Visualization of the declaration

int meatballs ;







# Notes About Variables

- You must not use a variable until you somehow give it a value.
- You can not assume that the variable will have a value before you give it one.
  - Some compilers do, others do not! This is the source of many errors that are difficult to find.



# Simple Data Types

Type	Typical Size in Bits	Minimal Range
char	8	–128 to 127
unsigned char	8	0 to 255
signed char	8	–128 to 127
int	16	–32,768 to 32,767
unsigned int	16	0 to 65,535
signed int	16	Same as int
short int	16	–32,768 to 32,767
unsigned short int	16	0 to 65,535
signed short int	16	Same as short int
long int	32	–2,147,483,648 to 2,147,483,647
signed long int	32	Same as long int
unsigned long int	32	0 to 4,294,967,295
float	32	1E–37 to 1E+37 with six digits of precision
double	64	1E–37 to 1E+37 with ten digits of precision
long double	80	1E–37 to 1E+37 with ten digits of precision



# Using Variables: Initialization

- Variables may be given initial values, or **initialized**, when declared. Examples:

`int length = 7 ;`      

length
7

`float diameter = 5.9 ;`      

diameter
5.9

`char initial = 'A' ;`      

initial
'A'



# Using Variables: Assignment

- Variables may have values assigned to them through the use of an **assignment statement**.
- Such a statement uses the **assignment operator =**
- This operator does not denote equality. It assigns the value of the right-hand side of the statement (the **expression**) to the variable on the left-hand side.
- Examples:

diameter = 5.9 ;

area = length \* width ;

Note that only single variables (LValue) may appear on the left-hand side of the assignment operator.



# Functions

- It is necessary for us to use some functions to write our first programs.
- Functions are parts of programs that perform a certain task and we have to give them some information so the function can do the task.
- We will show you how to use the functions as we go through the course and later on will show you how to create your own.



# Getting Input from User

- Every process requires some input from the user. Variables hold the input values.
- We have a function called `scanf( )` that will allow us to do that.
- The function `scanf` needs two pieces of information to display things.
  - The data type of input values
  - Address where to store the values
- `scanf( "%f", &diameter );`



# Displaying Variables

- Variables hold values that we occasionally want to show the person using the program.
- We have a function called `printf( )` that will allow us to do that.
- The function `printf` needs two pieces of information to display things.
  - How to display it
  - What to display
- `printf( "%f\n", &diameter );`



```
printf( “%f\n”, diameter );
```

- The name of the function is “printf”.
- Inside the parentheses are:
  - print specification, where we are going to display:
    - a floating point value (“%f”)
    - We want to have the next thing started on a new line (“\n”).
  - We want to display the contents of the variable diameter.
- printf( ) has many other capabilities.





# Backslash Codes

Code	Meaning
<b>\b</b>	<b>Backspace</b>
<b>\f</b>	<b>Form feed</b>
<b>\n</b>	<b>New line</b>
<b>\r</b>	<b>Carriage return</b>
<b>\t</b>	<b>Horizontal tab</b>
<b>\"</b>	<b>Double quote</b>
<b>\'</b>	<b>Single quote</b>
<b>\\</b>	<b>Backslash</b>
<b>\v</b>	<b>Vertical tab</b>
<b>\a</b>	<b>Alert</b>
<b>\?</b>	<b>Question mark</b>
<b>\N</b>	<b>Octal constant (where N is an octal constant)</b>
<b>\xN</b>	<b>Hexadecimal constant (where N is a hexadecimal constant)</b>



# Format Specifiers for printf and scanf

Data Type	Printf specifier	Scanf specifier
long double	%Lf	%Lf
double	%f	%lf
float	%f	%f
unsigned long int	%lu	%lu
long int	%ld	%ld
unsigned int	%u	%u
int	%d	%d
short	%hd	%hd
char	%c	%c



# Both printf and scanf Returns a Value

- We can call printf as  
i=810;  
n=printf("%d",i);
- We also can call a scanf  
m=scanf("%d%f",&i,&f)

What will be the value of n & m if every thing goes fine.



# Example: Declarations and Assignments

```
#include <stdio.h>
```

```
int main( void )  
{
```

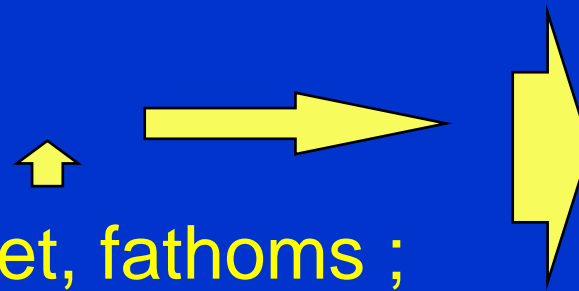
```
    int inches, feet, fathoms ;
```

```
    fathoms = 7 ;
```

```
    feet = 6 * fathoms ;
```

```
    inches = 12 * feet ;
```

```
    -  
    -  
    -
```



inches  
garbage

feet  
garbage

fathoms  
garbage



fathoms  
7



feet  
42



inches  
504



## Example: Declarations and Assignments (cont'd)

```
—  
—  
—  
printf (“Its depth at sea: \n”) ;  
printf (“    %d fathoms \n”, fathoms) ;  
printf (“    %d feet \n”, feet) ;  
printf (“    %d inches \n”, inches) ;  
  
    return 0 ;  
}
```



# Enhancing Our Example

- What if the depth were really 5.75 fathoms? Our program, as it is, couldn't handle it.
- Unlike integers, floating point numbers can contain decimal portions. So, let's use floating point, rather than integer.
- Let's also ask the user to enter the number of fathoms, by using the `scanf( )` function.



# Enhanced Program

```
#include <stdio.h>
int main ( void )
{
    float inches, feet, fathoms ;

    printf ("Enter the depth in fathoms : ") ;
    scanf ("%f", &fathoms) ;
    feet = 6 * fathoms ;
    inches = 12 * feet ;
    printf ("Its depth at sea: \n") ;
    printf ("    %f fathoms \n", fathoms) ;
    printf ("    %f feet \n", feet) ;
    printf ("    %f inches \n", inches) ;
    return 0 ;
}
```



# `scanf (“%f”, &fathoms) ;`

- The `scanf( )` function also needs two items:
  - The input specification “%f”. (Never put a “\n” into the input specification.)
  - The address of where to store the information. (We can input more than one item at a time if we wish, as long as we specify it correctly.)
- Notice the “&” in front of the variable name. It says to use the address of the variable to hold the information that the user enters.





# Final “Clean” Program

```
#include <stdio.h>
```

```
#define FEET_PER_FATHOM 6
```

```
#define INCHES_PER_FOOT 12
```

```
int main( void )
```

```
{
```

```
    float inches ;    /* number of inches deep */
```

```
    float feet ;      /* number of feet deep */
```

```
    float fathoms ;   /* number of fathoms deep */
```

```
    /* Get the depth in fathoms from the user */
```

```
    printf ( “Enter the depth in fathoms : ” ) ;
```

```
    scanf ( “%f”, &fathoms ) ;
```



# Final “Clean” Program (con’t)

```
/* Convert the depth to inches */
```

```
feet    = FEET_PER_FATHOM * fathoms ;  
inches = INCHES_PER_FOOT * feet ;
```

```
/* Display the results */
```

```
printf (“Its depth at sea: \n”) ;  
printf (“    %f fathoms \n”, fathoms) ;  
printf (“    %f feet \n”, feet);  
printf (“    %f inches \n”, inches);  
  
return 0 ;  
}
```



# Good Programming Practices

- Place each variable declaration on its own line with a descriptive comment.
- Place a comment before each logical “chunk” of code describing what it does.
- Do not place a comment on the same line as code (with the exception of variable declarations).
- Use spaces around all arithmetic and assignment operators.
- Use blank lines to enhance readability.



# Good Programming Practices (con't)

- Place a blank line between the last variable declaration and the first executable statement of the program.
- Indent the body of the program 3 to 5 spaces -- be consistent!
- Comments should explain why you are doing something, not what you are doing it.

```
a = a + 1 /* add one to a */    /* WRONG */  
        /* count new student */ /* RIGHT*/
```



# Another Sample Program

```
#include <stdio.h>
```

```
#define PI 3.14159
```

```
int main ( void )
```

```
{
```

```
    float radius = 3.0;
```

```
    float area;
```

```
    area = PI * radius * radius;
```

```
    printf( "The area is %f.\n", area );
```

```
    return 0 ;
```

```
}
```



# Arithmetic Operators in C

<u>Name</u>	<u>Operator</u>	<u>Example</u>
Addition	+	num1 + num2
Subtraction	-	initial - spent
Multiplication	*	fathoms * 6
Division	/	sum / count
Modulus	%	m % n



# Division

- If both operands of a division expression are integers, you will get an integer answer. The fractional portion is thrown away.

- Examples :  
$$17 / 5 = 3$$
$$4 / 3 = 1$$
$$35 / 9 = 3$$



# Division (con't)

- Division where at least one operand is a floating point number will produce a floating point answer.
- Examples :  
$$17.0 / 5 = 3.4$$
$$4 / 3.2 = 1.25$$
$$35.2 / 9.1 = 3.86813$$
- What happens? The integer operand is temporarily converted to a floating point, then the division is performed.





# Division By Zero

- Division by zero is mathematically undefined.
- If you allow division by zero in a program, it will cause a **fatal error**. Your program will terminate execution and give an error message.
- **Non-fatal errors** do not cause program termination, just produce incorrect results.



# Modulus

- The expression  $m \% n$  yields the integer remainder after  $m$  is divided by  $n$ .
- Modulus is an integer operation -- both operands MUST be integers.
- Examples :  
 $17 \% 5 = 2$   
 $6 \% 3 = 0$   
 $9 \% 2 = 1$   
 $5 \% 8 = 5$



# Uses for Modulus

- Used to determine if an integer value is even or odd

$$5 \% 2 = 1 \text{ odd} \quad 4 \% 2 = 0 \text{ even}$$

If you take the modulus by 2 of an integer, a result of 1 means the number is odd and a result of 0 means the number is even.



# Arithmetic Operators

## Rules of Operator Precedence

<u>Operator(s)</u>	<u>Precedence &amp; Associativity</u>
( )	Evaluated first. If <b>nested</b> , innermost first. If on same level, evaluated <b>left to right</b> .
* / %	Evaluated second. If there are several, evaluated <b>left to right</b> .
+ -	Evaluated third. If there are several, evaluated <b>left to right</b> .
=	Evaluated last, <b>right to left</b> .



# Using Parentheses

- Use parentheses to change the order in which an expression is evaluated.

$a + b * c$       Would multiply  $b * c$  first,  
then add  $a$  to the result.

If you really want the sum of  $a$  and  $b$  to be multiplied by  $c$ , use parentheses to force the evaluation to be done in the order you want.

$(a + b) * c$

- Also use parentheses to clarify a complex expression.



# Practice With Evaluating Expressions

Given integer variables  $a$ ,  $b$ ,  $c$ ,  $d$ , and  $e$ , where  $a = 1$ ,  $b = 2$ ,  $c = 3$ ,  $d = 4$ , evaluate the following expressions:

$$a + b - c + d$$

$$a * b / c$$

$$1 + a * b \% c$$

$$a + d \% b - c$$

$$e = b = d + c / b - a$$



# Relational Operators

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	is equal to
!=	is not equal to

**Relational expressions** evaluate to the integer values 1 (true) or 0 (false).

All of these operators are called **binary operators** because they take two expressions as **operands**.



# Practice with Relational Expressions

```
int a = 1, b = 2, c = 3 ;
```

<u>Expression</u>	<u>Value</u>	<u>Expression</u>	<u>Value</u>
$a < c$		$a + b \geq c$	
$b \leq c$		$a + b == c$	
$c \leq a$		$a != b$	
$a > b$		$a + b != c$	
$b \geq c$			





# Arithmetic Expressions: True or False

- **Arithmetic expressions** evaluate to numeric values.
- An arithmetic expression that has a value of zero is false.
- An arithmetic expression that has a value other than zero is true.



# Practice with Arithmetic Expressions

```
int    a = 1, b = 2, c = 3 ;
```

```
float  x = 3.33, y = 6.66 ;
```

<u>Expression</u>	<u>Numeric Value</u>	<u>True/False</u>
-------------------	----------------------	-------------------

$a + b$

$b - 2 * a$

$c - b - a$

$c - a$

$y - x$

$y - 2 * x$



# Increment and Decrement Operators

- The **increment operator** ++
- The **decrement operator** --
- Precedence: lower than (), but higher than \* / and %
- Associativity: right to left
- Increment and decrement operators can only be applied to variables, not to constants or expressions



# Increment Operator

- If we want to add one to a variable, we can say:

`count = count + 1 ;`

- Programs often contain statements that increment variables, so to save on typing, C provides these shortcuts:

`count++ ;    OR    ++count ;`

Both do the same thing. They change the value of count by adding one to it.



# Postincrement Operator

- The position of the ++ determines when the value is incremented. If the ++ is after the variable, then the incrementing is done last (a **postincrement**).

```
int amount, count ;
```

```
count = 3 ;
```

```
amount = 2 * count++ ;
```

- amount gets the value of  $2 * 3$ , which is 6, and then 1 gets added to count.
- So, after executing the last line, amount is 6 and count is 4.



# Preincrement Operator

- If the ++ is before the variable, then the incrementing is done first (a **preincrement**).

```
int amount, count ;  
  
count = 3 ;  
amount = 2 * ++count ;
```

- 1 gets added to count first, then amount gets the value of  $2 * 4$ , which is 8.
- So, after executing the last line, amount is 8 and count is 4.



# Code Example Using ++

```
#include <stdio.h>
int main ( )
{
    int i = 1 ;

    /* count from 1 to 10 */
    while ( i < 11 )
    {
        printf ("%d ", i) ;
        i++ ;                /* same as ++i */
    }
    return 0 ;
}
```



# Decrement Operator

- If we want to subtract one from a variable, we can say:

`count = count - 1 ;`

- Programs often contain statements that decrement variables, so to save on typing, C provides these shortcuts:

`count-- ;`    OR    `--count ;`

Both do the same thing. They change the value of count by subtracting one from it.





# Postdecrement Operator

- The position of the -- determines when the value is decremented. If the -- is after the variable, then the decrementing is done last (a **postdecrement**).

```
int amount, count ;
```

```
count = 3 ;
```

```
amount = 2 * count-- ;
```

- amount gets the value of  $2 * 3$ , which is 6, and then 1 gets subtracted from count.
- So, after executing the last line, amount is 6 and count is 2.



# Predecrement Operator

- If the -- is before the variable, then the decrementing is done first (a **predecrement**).

```
int amount, count ;
```

```
count = 3 ;
```

```
amount = 2 * --count ;
```

- 1 gets subtracted from count first, then amount gets the value of  $2 * 2$ , which is 4.
- So, after executing the last line, amount is 4 and count is 2.



# A Hand Trace Example

```
int answer, value = 4 ;
```

Code

Value  
4

Answer  
garbage

```
value = value + 1 ;
```

```
value++ ;
```

```
++value ;
```

```
answer = 2 * value++ ;
```

```
answer = ++value / 2 ;
```

```
value-- ;
```

```
--value ;
```

```
answer = --value * 2 ;
```



# Lvalue Required

`answer++ = value-- / 3 ;`

- In C any value that is having an address is called an Lvalue.



# Practice

Given

```
int a = 1, b = 2, c = 3 ;
```

What is the value of this expression?

```
++a * b - c--
```

What are the new values of a, b, and c?



# More Practice

Given

```
int a = 1, b = 2, c = 3, d = 4 ;
```

What is the value of this expression?

```
++b / c + a * d++
```

What are the new values of a, b, c, and d?



# Assignment Operators

=      +=      -=      \*=      /=      %=

Statement

Equivalent Statement

`a = a + 2 ;`

`a += 2 ;`

`a = a - 3 ;`

`a -= 3 ;`

`a = a * 2 ;`

`a *= 2 ;`

`a = a / 4 ;`

`a /= 4 ;`

`a = a % 2 ;`

`a %= 2 ;`

`b = b + ( c + 2 ) ;`

`b += c + 2 ;`

`d = d * ( e - 5 ) ;`

`d *= e - 5 ;`



# Practice with Assignment Operators

```
int i = 1, j = 2, k = 3, m = 4 ;
```

Expression

Value

$i += j + k$

$j *= k = m + 5$

$k -= m /= j * 2$





# Code Example Using /= and ++ Counting the Digits in an Integer

```
#include <stdio.h>

int main ( )
{
    int num, temp, digits = 0 ;
    temp = num = 4327 ;
    while ( temp > 0 )
    {
        printf ("%d\n", temp) ;
        temp /= 10 ;
        digits++ ;
    }
    printf ("There are %d digits in %d.\n", digits, num) ;
    return 0 ;
}
```



# Operator Precedence and Associativity

## Precedence

( )

++ -- ! + (unary) - (unary) (type)

\* / %

+ (addition) - (subtraction)

< <= > >=

== !=

&&

||

= += -= \*= /= %=

, (comma)

## Associativity

left to right/inside-out

right to left

left to right

left to right

left to right

left to right

left to right

left to right

right to left

right to left



# Review: Structured Programming

- All programs can be written in terms of only three control structures
  - The **sequence** structure
    - Unless otherwise directed, the statements are executed in the order in which they are written.
  - The **selection** structure
    - Used to choose among alternative courses of action.
  - The **repetition** structure
    - Allows an action to be repeated while some condition remains true.



# Selection: the **if** statement

```
if ( condition )  
{  
    statement(s)    /* body of the if statement */  
}
```

The braces are not required if the body contains only a single statement. However, they are a good idea and are required by the 104 C Coding Standards.



# Examples

```
if ( age >= 18 )
```

```
{
```

```
    printf("Vote!\n") ;
```

```
}
```

```
if ( value == 0 )
```

```
{
```

```
    printf ("The value you entered was zero.\n") ;
```

```
    printf ("Please try again.\n") ;
```

```
}
```



# Good Programming Practice

- Always place braces around the body of an if statement.
- Advantages:
  - Easier to read
  - Will not forget to add the braces if you go back and add a second statement to the body
  - Less likely to make a semantic error
- Indent the body of the if statement 3 to 5 spaces -- be consistent!



# Selection: the **if-else** statement

```
if ( condition )  
{  
    statement(s)  /* the if clause */  
}  
else  
{  
    statement(s)  /* the else clause */  
}
```



# Example

```
if ( age >= 18 )  
{  
    printf("Vote!\n") ;  
}  
else  
{  
    printf("Maybe next time!\n") ;  
}
```





# Example

```
if ( value == 0 )  
{  
    printf ("The value you entered was zero.\n") ;  
    printf("Please try again.\n") ;  
}  
else  
{  
    printf ("Value = %d.\n", value) ;  
}
```



# Good Programming Practice

- Always place braces around the bodies of the if and else clauses of an if-else statement.
- Advantages:
  - Easier to read
  - Will not forget to add the braces if you go back and add a second statement to the clause
  - Less likely to make a semantic error
- Indent the bodies of the if and else clauses 3 to 5 spaces -- be consistent!



# The Conditional Operator

`expr1 ? expr2 : expr3`

If `expr1` is true then `expr2` is executed, else `expr3` is evaluated, i.e.:

`x = ((y < z) ? y : z);`

OR

`(y < z) ? printf("%d is smaller",y): printf("%d is smaller",y);`



# Nesting of if-else Statements

```
if ( condition1 )
{
    statement(s)
}
else if ( condition2 )
{
    statement(s)
}
... /* more else clauses may be here */
else
{
    statement(s) /* the default case */
}
```



# Example

```
if ( value == 0 )
{
    printf ("The value you entered was zero.\n") ;
}
else if ( value < 0 )
{
    printf ("%d is negative.\n", value) ;
}
else
{
    printf ("%d is positive.\n", value) ;
}
```



# Gotcha! = versus ==

```
int a = 2 ;

if ( a = 1 ) /* semantic (logic) error! */
{
    printf ("a is one\n") ;
}
else if ( a == 2 )
{
    printf ("a is two\n") ;
}
else
{
    printf ("a is %d\n", a) ;
}
```



# Gotcha (con't)

- The statement `if (a = 1)` is syntactically correct, so no error message will be produced. (Some compilers will produce a warning.) However, a semantic (logic) error will occur.
- An assignment expression has a value -- the value being assigned. In this case the value being assigned is 1, which is true.
- If the value being assigned was 0, then the expression would evaluate to 0, which is false.



# Logical Operators

- So far we have seen only **simple conditions**.  
if ( count > 10 ) . . .
- Sometimes we need to test multiple conditions in order to make a decision.
- **Logical operators** are used for combining simple conditions to make **complex conditions**.

&&    is AND        if ( x > 5 && y < 6 )

||        is OR        if ( z == 0 || x > 10 )

!        is NOT        if ( ! ( bob > 42 ) )





# Example Use of &&

```
if ( age < 1  &&  gender == 'm')  
{  
    printf ("Infant boy\n") ;  
}
```



# Truth Table for &&

<u>Expression<sub>1</sub></u>	<u>Expression<sub>2</sub></u>	<u>Expression<sub>1</sub> &amp;&amp; Expression<sub>2</sub></u>
0	0	0
0	nonzero	0
nonzero	0	0
nonzero	nonzero	1

$\text{Exp}_1 \ \&\& \ \text{Exp}_2 \ \&\& \ \dots \ \&\& \ \text{Exp}_n$  will evaluate to 1 (true) only if **ALL subconditions** are true.



# Example Use of ||

```
if (grade == 'D' || grade == 'F')  
{  
    printf ("See with your Juniors !\n") ;  
}
```



# Truth Table for ||

<u>Expression<sub>1</sub></u>	<u>Expression<sub>2</sub></u>	<u>Expression<sub>1</sub>    Expression<sub>2</sub></u>
0	0	0
0	nonzero	1
nonzero	0	1
nonzero	nonzero	1

Exp<sub>1</sub> && Exp<sub>2</sub> && ... && Exp<sub>n</sub> will evaluate to 1 (true) if only ONE subcondition is true.



# Example Use of !

```
if ( ! (x == 2) ) /* same as (x != 2) */  
{  
    printf("x is not equal to 2.\n") ;  
}
```



# Truth Table for !

Expression

! Expression

0

1

nonzero

0



# Gotcha! && or ||

```
int a = 0 ;
int b=1;
if ( (a++ == 1) && (b++==1 ) ) /* semantic (logic) error! */
{
    printf ("First Gotcha\n") ;
}
else if ( (a++ == 0) || (b++==1 ) )
{
    printf ("Second Gotcha\n") ;
}
else
{
    printf ("a is %d\n", a) ;
}
```



# Gotcha (con't)

- While evaluating a condition if first subpart of a Complex condition having && operator is false than the remaining subpart will not be evaluated.
- Similarly While evaluating a condition if first subpart of a Complex condition having || operator is true than the remaining subpart will not be evaluated.





# Some Practice Expressions

```
int a = 1, b = 0, c = 7;
```

<u>Expression</u>	<u>Numeric Value</u>	<u>True/False</u>
a		
b		
c		
a + b		
a && b		
a    b		
!c		
!!c		
a && !b		
a < b && b < c		
a > b && b < c		
a >= b    b > c		



# More Practice

Given

```
int a = 5, b = 7, c = 17 ;
```

evaluate each expression as True or False.

1.  $c / b == 2$
2.  $c \% b \leq a \% b$
3.  $b + c / a != c - a$
4.  $(b < c) \&\& (c == 7)$
5.  $(c + 1 - b == 0) || (b = 5)$



# Review: Repetition Structure

- A **repetition structure** allows the programmer to specify that an action is to be repeated while some condition remains true.
- There are three repetition structures in C, the **while** loop, the **for** loop, and the **do-while** loop.



# The while Repetition Structure

```
while ( condition )  
{  
    statement(s)  
}
```

The braces are not required if the loop body contains only a single statement. However, they are a good idea and are required by the 104 C Coding Standards.



# Parts of a While Loop

- Every while loop will always contain three main elements:
  - Priming: initialize your variables.
  - Testing: test against some known condition.
  - Updating: update the variable that is tested.



# Simple While Loop

```
#include <stdio.h>
#define MAX 10
main ()
{
    int index = 1;
    while (index <= MAX) {
        printf ("Index:  %d\n", index)
        index = index + 1;
    }
}
```

1. Priming

2. Test Condition

3. Update

## OUTPUT:

Index: 1  
Index: 2  
Index: 3  
Index: 4  
Index: 5  
Index: 6  
Index: 7  
Index: 8  
Index: 9  
Index: 10



# Example

```
while ( children > 0 )  
{  
    children = children - 1 ;  
    cookies = cookies * 2 ;  
}
```



# Good Programming Practice

- Always place braces around the body of a while loop.
- Advantages:
  - Easier to read
  - Will not forget to add the braces if you go back and add a second statement to the loop body
  - Less likely to make a semantic error
- Indent the body of a while loop 3 to 5 spaces -- be consistent!





# Another while Loop Example

- Problem: Write a program that calculates the average exam grade for a class of 10 students.
- What are the program inputs?
  - the exam grades
- What are the program outputs?
  - the average exam grade



# The Pseudocode

<total> = 0

<grade\_counter> = 1

While (<grade\_counter> <= 10)

    Display "Enter a grade: "

    Read <grade>

    <total> = <total> + <grade>

    <grade\_counter> = <grade\_counter> + 1

End\_while

<average> = <total> / 10

Display "Class average is: ", <average>



# The C Code

```
#include <stdio.h>

int main ( )
{
    int counter, grade, total, average ;
    total = 0 ;
    counter = 1 ;
    while ( counter <= 10 )
    {
        printf ("Enter a grade : ") ;
        scanf ("%d", &grade) ;
        total = total + grade ;
        counter = counter + 1 ;
    }
    average = total / 10 ;
    printf ("Class average is: %d\n", average) ;
    return 0 ;
}
```



# Versatile?

- How versatile is this program?
- It only works with class sizes of 10.
- We would like it to work with any class size.
- A better way :
  - Ask the user how many students are in the class. Use that number in the condition of the while loop and when computing the average.



# New Pseudocode

<total> = 0

<grade\_counter> = 1

**Display** “Enter the number of students: “

**Read** <num\_students>

**While** (<grade\_counter> <= <num\_students>)

    Display “Enter a grade: ”

    Read <grade>

    <total> = <total> + <grade>

    <grade\_counter> = <grade\_counter> + 1

**End\_while**

<average> = <total> / <num\_students>

**Display** “Class average is: “, <average>



# New C Code

```
#include <stdio.h>

int main ( )
{
    int numStudents, counter, grade, total, average ;
    total = 0 ;
    counter = 1 ;

    printf ("Enter the number of students: ") ;
    scanf ("%d", &numStudents) ;
    while ( counter <= numStudents) {
        printf ("Enter a grade : ") ;
        scanf ("%d", &grade) ;

        total = total + grade ;
        counter = counter + 1 ;
    }

    average = total / numStudents ;
    printf ("Class average is: %d\n", average) ;
    return 0 ;
}
```



# Why Bother to Make It Easier?

- Why do we write programs?
  - So the user can perform some task
- The more versatile the program, the more difficult it is to write. BUT it is more useable.
- The more complex the task, the more difficult it is to write. But that is often what a user needs.
- Always consider the user first.



# Using a Sentinel Value

- We could let the user keep entering grades and when he's done enter some special value that signals us that he's done.
- This special signal value is called a **sentinel value**.
- We have to make sure that the value we choose as the sentinel isn't a legal value. For example, we can't use 0 as the sentinel in our example as it is a legal value for an exam score.





# The Priming Read

- When we use a sentinel value to control a while loop, we have to get the first value from the user before we encounter the loop so that it will be tested and the loop can be entered.
- This is known as a **priming read**.
- We have to give significant thought to the initialization of variables, the sentinel value, and getting into the loop.



# New Pseudocode

**<total> = 0**

**<grade\_counter> = 1**

**Display “Enter a grade: “**

**Read <grade>**

**While ( <grade> != -1 )**

**<total> = <total> + <grade>**

**<grade\_counter> = <grade\_counter> + 1**

**Display “Enter another grade: ”**

**Read <grade>**

**End\_while**

**<average> = <total> / <grade\_counter>**

**Display “Class average is: “, <average>**



# New C Code

```
#include <stdio.h>
int main ( )
{
    int counter, grade, total, average ;

    total = 0 ;
    counter = 1 ;
    printf("Enter a grade: ") ;
    scanf("%d", &grade) ;
    while (grade != -1) {
        total = total + grade ;
        counter = counter + 1 ;
        printf("Enter another grade: ") ;
        scanf("%d", &grade) ;
    }
    average = total / counter ;
    printf ("Class average is: %d\n", average) ;
    return 0 ;
}
```



# Final “Clean” C Code

```
#include <stdio.h>
int main ( )
{
    int counter ;    /* counts number of grades entered */
    int grade ;      /* individual grade */
    int total;       /* total of all grades */
    int average ;    /* average grade */

    /* Initializations */

    total = 0 ;
    counter = 1 ;
```



# Final “Clean” C Code (con’t)

```
/* Get grades from user */  
/* Compute grade total and number of grades */
```

```
printf("Enter a grade: ") ;  
scanf("%d", &grade) ;  
while (grade != -1) {  
    total = total + grade ;  
    counter = counter + 1 ;  
    printf("Enter another grade: ") ;  
    scanf("%d", &grade) ;  
}
```

```
/* Compute and display the average grade */
```

```
average = total / counter ;  
printf ("Class average is: %d\n", average) ;
```

```
return 0 ;
```

```
}
```



# Using a while Loop to Check User Input

```
#include <stdio.h>

int main ( )
{
    int number ;
    printf ("Enter a positive integer : ") ;
    scanf ("%d", &number) ;
    while ( number <= 0 )
    {
        printf ("\nThat's incorrect. Try again.\n") ;
        printf ("Enter a positive integer: ") ;
        scanf ("%d", &number) ;
    }
    printf ("You entered: %d\n", number) ;
    return 0 ;
}
```



# Counter-Controlled Repetition (Definite Repetition)

- If it is known in advance exactly how many times a loop will execute, it is known as a **counter-controlled loop**.

```
int i = 1 ;  
while ( i <= 10 )  
{  
    printf("i = %d\n", i) ;  
    i = i + 1 ;  
}
```



# Counter-Controlled Repetition (con't)

- Is the following loop a counter-controlled loop?

```
while ( x != y )  
{  
    printf("x = %d", x) ;  
    x = x + 2 ;  
}
```





# Event-Controlled Repetition (Indefinite Repetition)

- If it is NOT known in advance exactly how many times a loop will execute, it is known as an **event-controlled loop**.

```
sum = 0 ;  
printf("Enter an integer value: ") ;  
scanf("%d", &value) ;  
while ( value != -1) {  
    sum = sum + value ;  
    printf("Enter another value: ") ;  
    scanf("%d", &value) ;  
}
```



# Event-Controlled Repetition (con't)

- An event-controlled loop will terminate when some **event** occurs.
- The event may be the occurrence of a sentinel value, as in the previous example.
- There are other types of events that may occur, such as reaching the end of a data file.



# The 3 Parts of a Loop

```
#include <stdio.h>
int main ()
{
    int i = 1 ;
    /* count from 1 to 100 */
    while ( i < 101 )
    {
        printf ("%d ", i) ;
        i = i + 1 ;
    }
    return 0 ;
}
```

Initialization of **loop control variable**

test of **loop termination condition**

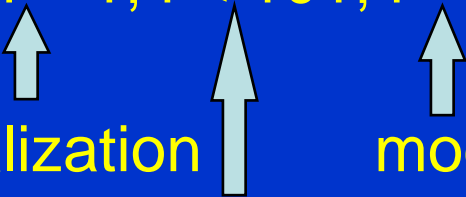
modification of loop control variable



# The for Loop Repetition Structure

- The **for** loop handles details of the counter-controlled loop “automatically”.
- The initialization of the the loop control variable, the termination condition test, and control variable modification are handled in the **for** loop structure.

```
for ( i = 1; i < 101; i = i + 1 )  
{  
  initialization  
  test  
  modification  
}
```





# When Does a for Loop Initialize, Test and Modify?

- Just as with a while loop, a for loop
  - initializes the loop control variable before beginning the first loop iteration,
  - modifies the loop control variable at the very end of each iteration of the loop, and
  - performs the loop termination test before each iteration of the loop.
- The for loop is easier to write and read for counter-controlled loops.



# A for Loop That Counts From 0 to 9

```
for ( i = 0; i < 10; i = i + 1 )  
{  
    printf ("%d\n", i) ;  
}
```



# We Can Count Backwards, Too

```
for ( i = 9; i >= 0; i = i - 1 )  
{  
    printf ("%d\n", i) ;  
}
```



# We Can Count By 2's ... or 7's ... or Whatever

```
for ( i = 0; i < 10; i = i + 2 )  
{  
    printf ("%d\n", i) ;  
}
```





# The **do-while** Repetition Structure

```
do  
{  
    statement(s)  
} while ( condition ) ;
```

- The body of a **do-while** is ALWAYS executed at least once. Is this true of a **while** loop? What about a **for** loop?



# Example

```
do
{
    printf ("Enter a positive number: ") ;
    scanf ("%d", &num) ;
    if ( num <= 0 )
    {
        printf ("\nThat is not positive. Try again\n") ;
    }
} while ( num <= 0 ) ;
```



# An Equivalent while Loop

```
printf ("Enter a positive number: ") ;  
scanf ("%d", &num) ;  
while ( num <= 0 )  
{  
    printf ("\nThat is not positive. Try again\n") ;  
    printf ("Enter a positive number: ") ;  
    scanf ("%d", &num) ;  
}
```

- Notice that using a while loop in this case requires a priming read.



# An Equivalent for Loop

```
printf ("Enter a positive number: ") ;  
scanf ("%d", &num) ;  
  
for ( ; num <= 0; )  
{  
    printf ("\nThat is not positive. Try again\n") ;  
    printf ("Enter a positive number: ") ;  
    scanf ("%d", &num) ;  
}
```

- A for loop is a very awkward choice here because the loop is event-controlled.



# So, Which Type of Loop Should I Use?

- Use a **for** loop for counter-controlled repetition.
- Use a **while** or **do-while** loop for event-controlled repetition.
  - Use a **do-while** loop when the loop must execute at least one time.
  - Use a **while** loop when it is possible that the loop may never execute.



# Infinite Loop

- Infinite Loop: A loop that never ends.
  - Generally, you want to avoid these!
  - There are special cases, however, when you do want to create infinite loops on purpose.
- Common Exam Questions:
  - Given a piece of code, identify the bug in the code.
  - You may need to identify infinite loops.



# Infinite Loop Example #1

```
#include <stdio.h>
```

```
#define MAX 10
```

```
main ()
```

```
{
```

```
    int index =1;
```

```
    while (index <= MAX)
```

```
    {
```

```
        printf ("Index: %d\n", index);
```

```
    }
```

```
}
```

Index: 1

Index: 1

Index: 1

Index: 1

Index: 1

...

[forever]



# Infinite Loop, Example #2

```
#include <stdio.h>
```

```
/*no MAX here*/
```

```
main ()
```

```
{
```

```
    int index = 1;
```

```
    while (index > 0)
```

```
    {
```

```
        printf ("Index: %d\n", index);
```

```
        index = index + 1;
```

```
    }
```

```
}
```

Index: 1

Index: 2

Index: 3

Index: 4

Index: 5

... [forever] ?





# Nested Loops

- Loops may be **nested (embedded)** inside of each other.
- Actually, any control structure (sequence, selection, or repetition) may be nested inside of any other control structure.
- It is common to see nested for loops.



# Nested for Loops

```
for ( i = 1; i < 5; i = i + 1 )
{
    for ( j = 1; j < 3; j = j + 1 )
    {
        if ( j % 2 == 0 )
        {
            printf ("O") ;
        }
        else
        {
            printf ("X") ;
        }
    }
    printf ("\n") ;
}
```



How many times is the "if" statement executed?

What is the output ?



# The **break** Statement

- The **break** statement can be used in **while**, **do-while**, and **for** loops to cause premature exit of the loop.



# Example break in a for Loop

```
#include <stdio.h>
int main ( )
{
    int i ;
    for ( i = 1; i < 10; i = i + 1 )
    {
        if (i == 5)
        {
            break ;
        }
        printf ("%d ", i) ;
    }
    printf ("\nBroke out of loop at i = %d.\n", i) ;
    return 0 ;
}
```

## OUTPUT:

1 2 3 4

**Broke out of loop at i = 5.**



# The **continue** Statement

- The **continue** statement can be used in **while**, **do-while**, and **for** loops.
- It causes the remaining statements in the body of the loop to be skipped for the current iteration of the loop.



# Example continue in a for Loop

```
#include <stdio.h>
int main ( )
{
    int i ;
    for ( i = 1; i < 10; i = i + 1 )
    {
        if (i == 5)
        {
            continue ;
        }
        printf ("%d ", i) ;
    }
    printf ("\nDone.\n") ;
    return 0 ;
}
```

## OUTPUT:

1 2 3 4 6 7 8 9

Done.



# Debugging Tips

- Trace your code by hand (a **hand trace**), keeping track of the value of each variable.
- Insert temporary printf() statements so you can see what your program is doing.
  - Confirm that the correct value(s) has been read in.
  - Check the results of arithmetic computations immediately after they are performed.



# Multiple Selection

- So far, we have only seen **binary selection**.

```
if ( age >= 18 )
```

```
{
```

```
    printf("Vote!\n") ;
```

```
}
```

```
if ( age >= 18 )
```

```
{
```

```
    printf("Vote!\n") ;
```

```
}
```

```
else
```

```
{
```

```
    printf("Maybe next time!\n") ;
```

```
}
```





# Multiple Selection (con't)

- Sometimes it is necessary to **branch** in more than two directions.
- We do this via **multiple selection**.
- The multiple selection mechanism in C is the **switch** statement.



# Multiple Selection with if

```
if (day == 0 ) {  
    printf ("Sunday") ;  
}  
if (day == 1 ) {  
    printf ("Monday") ;  
}  
if (day == 2) {  
    printf ("Tuesday") ;  
}  
if (day == 3) {  
    printf ("Wednesday") ;  
}
```

```
if (day == 4) {  
    printf ("Thursday") ;  
}  
if (day == 5) {  
    printf ("Friday") ;  
}  
if (day == 6) {  
    printf ("Saturday") ;  
}  
if ((day < 0) || (day > 6)) {  
    printf("Error - invalid day.\n") ;  
}
```



# Multiple Selection with if-else

```
if (day == 0 ) {  
    printf ("Sunday") ;  
} else if (day == 1 ) {  
    printf ("Monday") ;  
} else if (day == 2) {  
    printf ("Tuesday") ;  
} else if (day == 3) {  
    printf ("Wednesday") ;  
} else if (day == 4) {  
    printf ("Thursday") ;  
} else if (day == 5) {  
    printf ("Friday") ;  
} else if (day = 6) {  
    printf ("Saturday") ;  
} else {  
    printf ("Error - invalid day.\n") ;  
}
```

This if-else structure is more efficient than the corresponding if structure. Why?



# The **switch** Multiple-Selection Structure

```
switch ( integer expression )  
{  
    case constant1 :  
        statement(s)  
        break ;  
    case constant2 :  
        statement(s)  
        break ;  
    . . .  
    default:  
        statement(s)  
        break ;  
}
```



# switch Statement Details

- The last statement of each case in the switch should almost always be a break.
- The break causes program control to jump to the closing brace of the switch structure.
- Switch statement can only test for equality condition (==).
- A switch statement will compile without a default case, but always consider using one.



# Good Programming Practices

- Include a default case to catch invalid data.
- Inform the user of the type of error that has occurred (e.g., “Error - invalid day.”).
- If appropriate, display the invalid value.



# switch Example

```
switch ( day )
{
    case 0: printf ("Sunday\n") ;
            break ;
    case 1: printf ("Monday\n") ;
            break ;
    case 2: printf ("Tuesday\n") ;
            break ;
    case 3: printf ("Wednesday\n") ;
            break ;
    case 4: printf ("Thursday\n") ;
            break ;
    case 5: printf ("Friday\n") ;
            break ;
    case 6: printf ("Saturday\n") ;
            break ;
    default: printf ("Error -- invalid day.\n") ;
            break ;
}
```

Is this structure more efficient than the equivalent nested if-else structure?



# Why Use a switch Statement?

- A nested if-else structure is just as efficient as a switch statement.
- However, a switch statement may be easier to read.
- Also, it is easier to add new cases to a switch statement than to a nested if-else structure.





# The **char** Data Type

- The **char** data type holds a single character.

```
char ch;
```

- Example assignments:

```
char grade, symbol;
```

```
grade = 'B';
```

```
symbol = '$';
```

- The char is held as a one-byte integer in memory. The ASCII code is what is actually stored, so we can use them as characters or integers, depending on our need.



# The char Data Type (con't)

- Use

```
scanf ("%c", &ch) ;
```

to read a single character into the variable ch.  
(Note that the variable does not have to be called "ch".)

- Use

```
printf ("%c", ch) ;
```

to display the value of a character variable.



# char Example

```
#include <stdio.h>
int main ( )
{
    char ch ;

    printf ("Enter a character: ") ;
    scanf ("%c", &ch) ;
    printf ("The value of %c is %d.\n", ch, ch) ;
    return 0 ;
}
```

If the user entered an A, the output would be:

The value of A is 65.



# The getchar ( ) Function

- The getchar( ) function is found in the **stdio** library.
- The getchar( ) function reads one character from **stdin** (the **standard input buffer**) and returns that character's ASCII value.
- The value can be stored in either a character variable or an integer variable.



# getchar ( ) Example

```
#include <stdio.h>
int main ( )
{
    char ch ;    /* int ch would also work! */

    printf ("Enter a character: ") ;
    ch = getchar( ) ;
    printf ("The value of %c is %d.\n", ch, ch) ;
    return 0 ;
}
```

If the user entered an A, the output would be:

The value of A is 65.



# Problems with Reading Characters

- When getting characters, whether using `scanf( )` or `getchar( )`, realize that you are reading only one character.
- What will the user actually type? The character he/she wants to enter, followed by pressing ENTER.
- So, the user is actually entering two characters, his/her response and the **newline character**.
- Unless you handle this, the newline character will remain in the stdin stream causing problems the next time you want to read a character. Another call to `scanf( )` or `getchar( )` will remove it.



# Improved getchar( ) Example

```
#include <stdio.h>
int main ( )
{
    char ch, newline ;

    printf ("Enter a character: ") ;
    ch = getchar( ) ;
    newline = getchar( ) ; /* could also use scanf("%c", &newline) ; */
    printf ("The value of %c is %d.\n", ch, ch) ;
    return 0 ;
}
```

If the user entered an A, the output would be:

The value of A is 65.



# Additional Concerns with Garbage in stdin

- When we were reading integers using `scanf( )`, we didn't seem to have problems with the newline character, even though the user was typing ENTER after the integer.
- That is because `scanf( )` was looking for the next integer and ignored the newline (**whitespace**).
- If we use `scanf ("%d", &num);` to get an integer, the newline is still stuck in the input stream.
- If the next item we want to get is a character, whether we use `scanf( )` or `getchar( )`, we will get the newline.
- We have to take this into account and remove it.





# EOF Predefined Constant

- `getchar( )` is usually used to get characters from a file until the end of the file is reached.
- The value used to indicate the end of file varies from system to system. It is **system dependent**.
- But, regardless of the system you are using, there is a `#define` in the `stdio` library for a symbolic integer constant called **EOF**.
- **EOF** holds the value of the end-of-file marker for the system that you are using.



# getchar( ) Example Using EOF

```
#include <stdio.h>
int main ()
{
    int grade, aCount, bCount, cCount, dCount, fCount ;
    aCount = bCount = cCount = dCount = fCount = 0 ;
    while ( (grade = getchar( ) ) != EOF ) {
        switch ( grade ) {
            case 'A': aCount++; break ;
            case 'B': bCount++; break ;
            case 'C' : cCount++; break ;
            case 'D': dCount++; break ;
            case 'F': fCount++; break ;
            default : break ;
        }
    }
    return 0 ;
}
```



# Incremental Programming Review

- Write your code in incomplete but working pieces.
- For example, for your projects,
  - Don't write the whole program at once.
  - Just write enough to display the user prompt on the screen.
  - Get that part working first (compile and run).
  - Next, write the part that gets the value from the user, and then just print it out.



# Increment Programming Review (con't)

- Get that working (compile and run).
- Next, change the code so that you use the value in a calculation and print out the answer.
- Get that working (compile and run).
- Continue this process until you have the final version.
- Get the final version working.
- Bottom line: Always have a working version of your program!



# Example of Incremental Programming

## Problem:

- Write an **interactive** program that allows the user to calculate the interest accrued on a savings account. The interest is compounded annually.
- The user must supply the principal amount, the interest rate, and the number of years over which to compute the interest.



# Rough Algorithm

Print explanation of the program

Get <principal> from user

Get <interest rate> from user

Get <number of years> from user

<amount> = <principal>

While (<number of years> > 0 )

    amount = amount + (amount X <interest rate>)

    <number of years> = <number of year> + 1

End\_while

<interest accrued> = <amount> - <principal>

Display report



# Report Design

Interest rate : 7.0000 %

Period : 20 years

Principal at start of period : 1000.00

Interest accrued : 2869.68

Total amount at end of period : 3869.68



# Version #1

```
/* Filename:    interest.c
 * Author:      Hemant Mehta
 * Date written: 11/14//06
 * Description: This program computes the interest accrued in an account
 *              that compounds interest annually. */
#include <stdio.h>
int main ( )
{
    /* Print Instructions */
    printf ("This program computes the interest accrued in an account that\n");
    printf ("compounds interest annually.  You will need to enter the amount\n");
    printf ("of the principal, the interest rate and the number of years.\n\n");

    return 0;
}
```





# Output #1

This program computes the interest accrued in an account that compounds interest annually. You will need to enter the amount of the principal, the interest rate and the number of years.



# Version #2

```
/* Filename:    interest.c
 * Author:
 * Date written: 11/14//99
 * Description: This program computes the interest accrued in an account
 *              that compounds interest annually. */
#include <stdio.h>
int main ( )
{
    float principal, rate ;
    int   years ;

    /* Print Instructions */
    printf ("This program computes the interest accrued in an account that\n") ;
    printf ("compounds interest annually. You will need to enter the amount\n") ;
    printf ("of the principal, the interest rate and the number of years.\n\n") ;

    /* Get input from user */
    printf ("Enter the principal amount : ") ;
    scanf ("%f", &principal) ;
    printf ("Enter the interest rate as a decimal (for 7%% enter .07) : ") ;
    scanf ("%f", &rate) ;
    printf ("Enter the number of years : ") ;
    scanf ("%d", &years) ;
    printf ("\nprincipal = %f, rate = %f, years = %d\n", principal, rate, years) ;
    return 0 ;
}
```



# Output #2

This program computes the interest accrued in an account that compounds interest annually. You will need to enter the amount of the principal, the interest rate and the number of years.

Enter the principal amount : **1000.00**

Enter the interest rate as a decimal (for 7% enter .07) : **.07**

Enter the number of years : **20**

principal = 1000.000000, rate = 0.070000, years = 20



# Version #3

```
/* Filename:    interest.c
 * Author:      _____
 * Date written: 11/14/99
 * Description: This program computes the interest accrued in an account
 *              that compounds interest annually.                */
#include <stdio.h>
int main ( )
{
    float principal, rate, amount, interest ;
    int    years, i ;

    /* Print Instructions */
    printf ("This program computes the interest accrued in an account that\n");
    printf ("compounds interest annually.  You will need to enter the amount\n");
    printf ("of  the principal, the interest rate and the number of years.\n\n");

    /* Get input from user */
    printf ("Enter the principal amount : ");
    scanf ("%f", &principal);
    printf ("Enter the interest rate as a decimal (for 7%% enter .07) : " );
    scanf ("%f", &rate);
    printf ("Enter the number of years : ");
    scanf ("%d", &years);
```



## Version #3 (con't)

```
/* Save the original principal amount by varying another variable, amount */
amount = principal;

/* Calculate total amount in the account after the specified number of years */
for ( i = 0 ; i < 1 ; i++ )
{
    amount += amount * rate ;
}

/* Calculate accrued interest */
interest = amount - principal ;

printf ("\nprincipal = %f, rate = %f, years = %d\n", principal, rate, years ) ;
printf ("amount = %f, interest = %f\n");

return 0 ;
}
```



# Output #3

This program computes the interest accrued in an account that compounds interest annually. You will need to enter the amount of the principal, the interest rate and the number of years.

Enter the principal amount : **1000.00**

Enter the interest rate as a decimal (for 7% enter .07) : **.07**

Enter the number of years : **20**

principal = 1000.000000, rate = 0.070000, years = 20

amount = 1070.000000, interest = 70.000000



# Version #4

```
/* Filename:    interest.c
 * Author:      _____
 * Date written: 11/14/99
 * Description: This program computes the interest accrued in an account
 *              that compounds interest annually.          */
#include <stdio.h>
int main ( )
{
    float principal, rate, amount, interest ;
    int   years, i ;

    /* Print Instructions */
    printf ("This program computes the interest accrued in an account that\n");
    printf ("compounds interest annually.  You will need to enter the amount\n");
    printf ("of  the principal, the interest rate and the number of years.\n\n");

    /* Get input from user */
    printf ("Enter the principal amount : ");
    scanf ("%f", &principal);
    printf ("Enter the interest rate as a decimal (for 7%% enter .07) : " );
    scanf ("%f", &rate);
    printf ("Enter the number of years : ");
    scanf ("%d", &years);
```



## Version #4 (con't)

```
/* Save the original principal amount by varying another variable, amount */
amount = principal;

/* Calculate total amount in the account after the specified number of years */
for ( i = 0 ; i < 2 ; i++ )
{
    amount += amount * rate ;
}

/* Calculate accrued interest */
interest = amount - principal ;

printf ("\nprincipal = %f, rate = %f, years = %d\n", principal, rate, years ) ;
printf ("amount = %f, interest = %f\n");

return 0 ;
}
```





# Output #4

This program computes the interest accrued in an account that compounds interest annually. You will need to enter the amount of the principal, the interest rate and the number of years.

Enter the principal amount : **1000.00**

Enter the interest rate as a decimal (for 7% enter .07) : **.07**

Enter the number of years : **20**

principal = 1000.000000, rate = 0.070000, years = 20

amount = 1144.900000, interest = 144.900000



# Version #5

```
/* Filename:    interest.c
 * Author:      _____
 * Date written: 11/14/99
 * Description: This program computes the interest accrued in an account
 *              that compounds interest annually.          */
#include <stdio.h>
int main ( )
{
    float principal, rate, amount, interest ;
    int    years, i ;

    /* Print Instructions */
    printf ("This program computes the interest accrued in an account that\n");
    printf ("compounds interest annually.  You will need to enter the amount\n");
    printf ("of the principal, the interest rate and the number of years.\n\n");

    /* Get input from user */
    printf ("Enter the principal amount : ");
    scanf ("%f", &principal);
    printf ("Enter the interest rate as a decimal (for 7%% enter .07) : ");
    scanf ("%f", &rate);
    printf ("Enter the number of years : ");
    scanf ("%d", &years);
```



# Version #5 (con't)

```
/* Save the original principal amount by varying another variable, amount */
amount = principal;

/* Calculate total amount in the account after the specified number of years */
for ( i = 0 ; i < years ; i++ )
{
    amount += amount * rate ;
}

/* Calculate accrued interest */
interest = amount - principal ;

printf ("\nprincipal = %f, rate = %f, years = %d\n", principal, rate, years ) ;
printf ("amount = %f, interest = %f\n");

return 0 ;
}
```



# Output #5

This program computes the interest accrued in an account that compounds interest annually. You will need to enter the amount of the principal, the interest rate and the number of years.

Enter the principal amount : **1000.00**

Enter the interest rate as a decimal (for 7% enter .07) : **.07**

Enter the number of years : **20**

principal = 1000.000000, rate = 0.070000, years = 20

amount = 3869.680000, interest = 2869.680000



# Final Version

```
/* Filename:    interest.c
 * Author:      _____
 * Date written: 11/14/99
 * Description: This program computes the interest accrued in an account
 *              that compounds interest annually.          */
#include <stdio.h>
int main ( )
{
    float principal, rate, amount, interest ;
    int   years, i ;

    /* Print Instructions */
    printf ("This program computes the interest accrued in an account that\n");
    printf ("compounds interest annually.  You will need to enter the amount\n");
    printf ("of  the principal, the interest rate and the number of years.\n\n");

    /* Get input from user */
    printf ("Enter the principal amount : ");
    scanf ("%f", &principal);
    printf ("Enter the interest rate as a decimal (for 7%% enter .07) : " );
    scanf ("%f", &rate);
    printf ("Enter the number of years : ");
    scanf ("%d", &years);
```



# Final Version (con't)

```
/* Save the original principal amount by varying another variable, amount */
amount = principal;

/* Calculate total amount in the account after the specified number of years */
for ( i = 0 ; i < years ; i++ )
{
    amount += amount * rate ;
}

/* Calculate accrued interest */
interest = amount - principal ;

/* Print report */
printf ("Interest rate : %.4f %%\n", 100 * rate ) ;
printf ("    Period : %d years\n\n", years ) ;
printf ("    Principal at start of period : %9.2f", principal );
printf ("    Interest accrued : %9.2f", interest );
printf ("Total amount at end of period : %9.2f", amount);

return 0 ;
}
```



# Final Output

This program computes the interest accrued in an account that compounds interest annually. You will need to enter the amount of the principal, the interest rate and the number of years.

Enter the principal amount : **1000.00**

Enter the interest rate as a decimal (for 7% enter .07) : **.07**

Enter the number of years : **20**

Interest rate : 7.0000 %

Period : 20 years

Principal at start of period : 1000.00

Interest accrued : 2869.68

Total amount at end of period : 3869.68



# Top-Down Design

- If we look at a problem as a whole, it may seem impossible to solve because it is so complex.  
Examples:
  - writing a tax computation program
  - writing a word processor
- Complex problems can be solved using **top-down design**, also known as **stepwise refinement**, where
  - We break the problem into parts
  - Then break the parts into parts
  - Soon, each of the parts will be easy to do





# Advantages of Top-Down Design

- Breaking the problem into parts helps us to clarify what needs to be done.
- At each step of refinement, the new parts become less complicated and, therefore, easier to figure out.
- Parts of the solution may turn out to be reusable.
- Breaking the problem into parts allows more than one person to work on the solution.



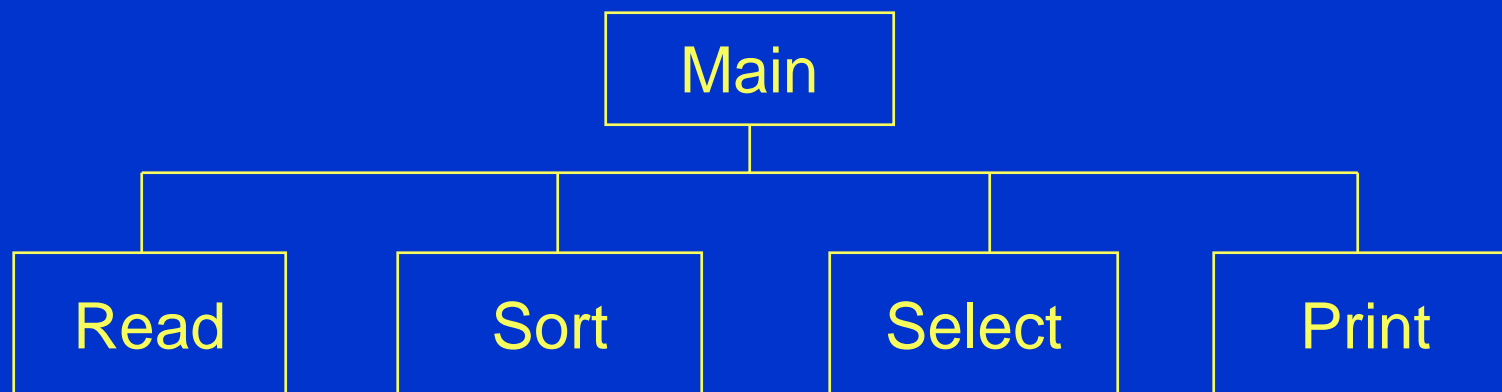
# An Example of Top-Down Design

- Problem:
  - We own a home improvement company.
  - We do painting, roofing, and basement waterproofing.
  - A section of town has recently flooded (zip code 21222).
  - We want to send out pamphlets to our customers in that area.



# The Top Level

- Get the customer list from a file.
- Sort the list according to zip code.
- Make a new file of only the customers with the zip code 21222 from the sorted customer list.
- Print an envelope for each of these customers.





# Another Level?

- Should any of these steps be broken down further? Possibly.
- How do I know? Ask yourself whether or not you could easily write the algorithm for the step. If not, break it down again.
- When you are comfortable with the breakdown, write the pseudocode for each of the steps (**modules**) in the **hierarchy**.
- Typically, each module will be coded as a separate function.



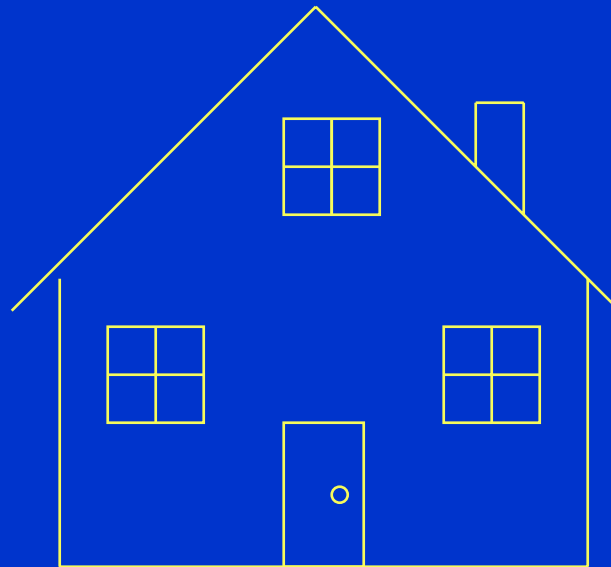
# Structured Programs

- We will use top-down design for all remaining programming projects.
- This is the standard way of writing programs.
- Programs produced using this method and using only the three kinds of control structures, sequential, selection and repetition, are called **structured programs**.
- Structured programs are easier to test, modify, and are also easier for other programmers to understand.



# Another Example

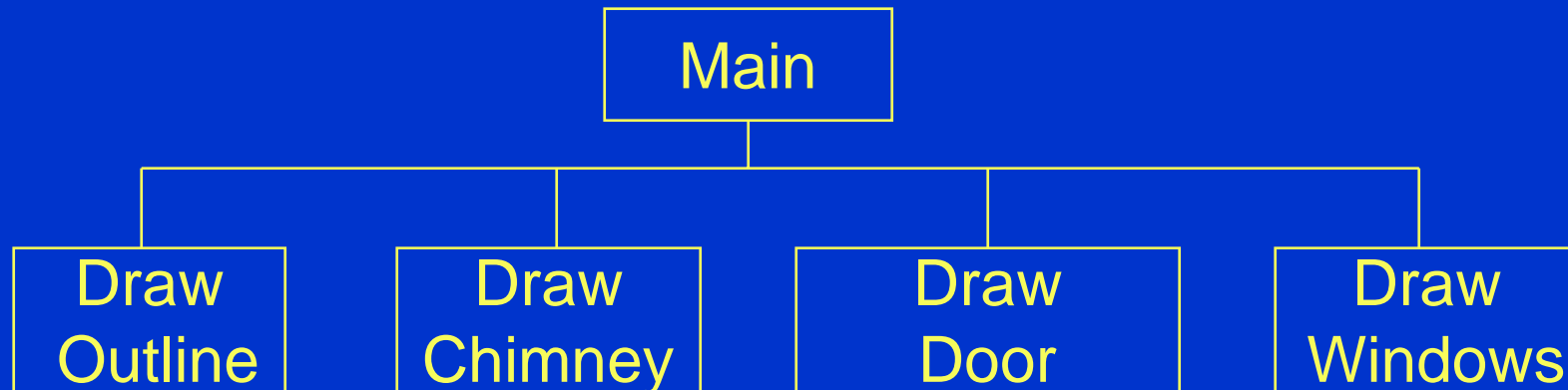
- Problem: Write a program that draws this picture of a house.





# The Top Level

- Draw the outline of the house
- Draw the chimney
- Draw the door
- Draw the windows





# Pseudocode for Main

Call Draw Outline

Call Draw Chimney

Call Draw Door

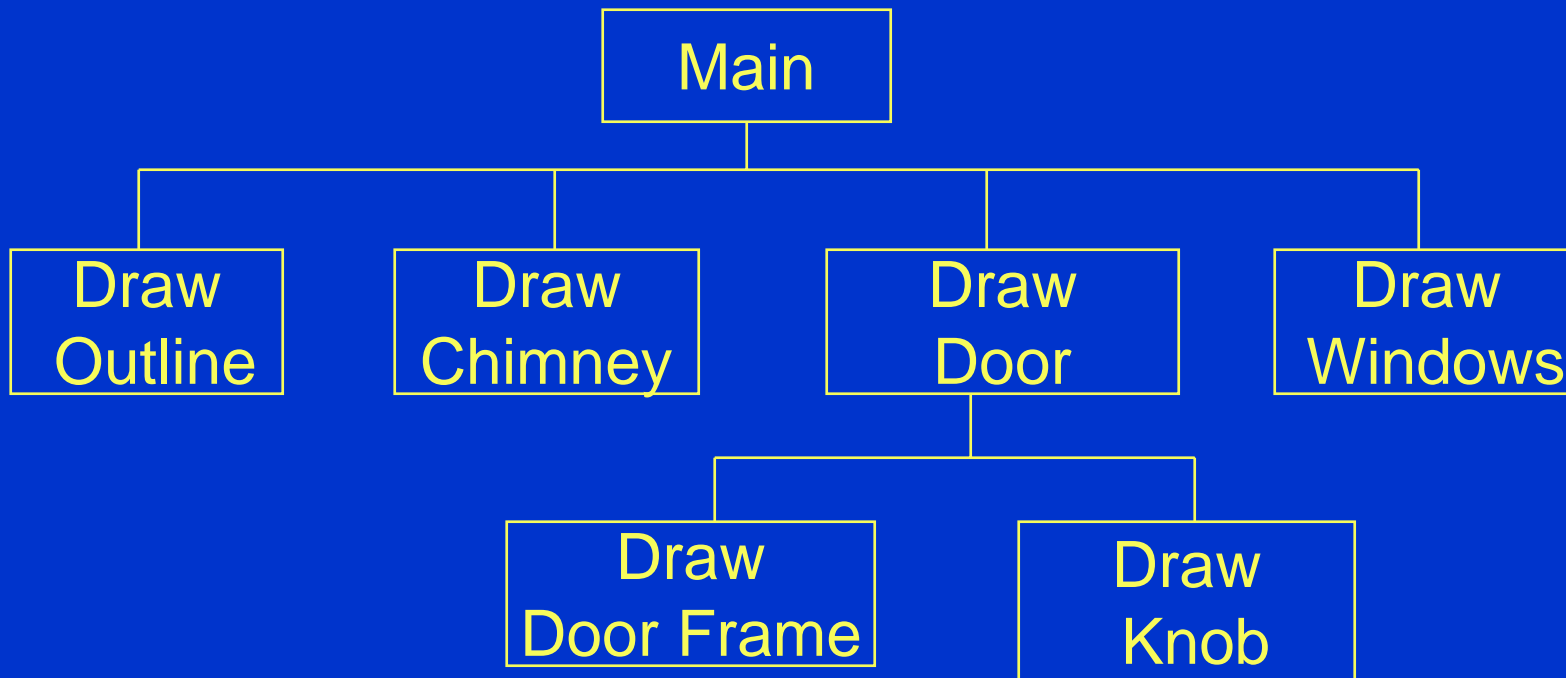
Call Draw Windows





# Observation

- The door has both a frame and knob. We could break this into two steps.





# Pseudocode for Draw Door

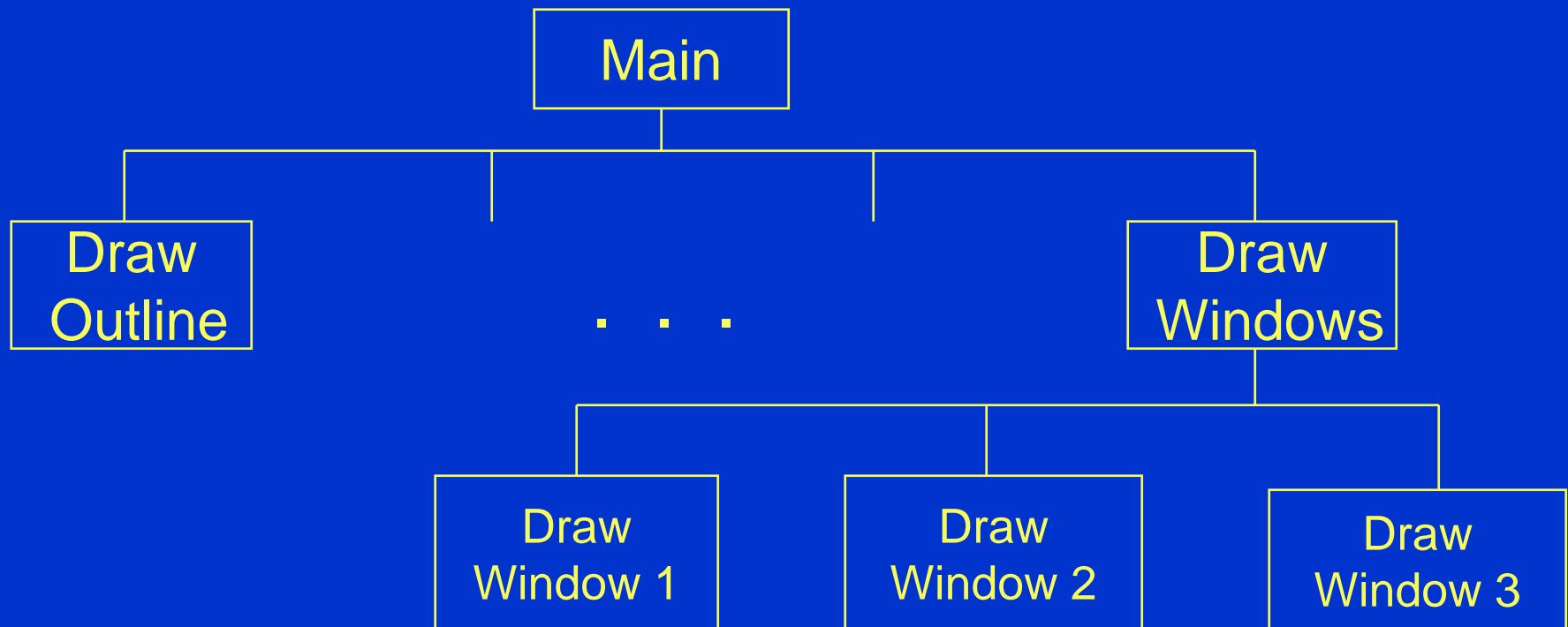
Call Draw Door Frame

Call Draw Knob



# Another Observation

- There are three windows to be drawn.



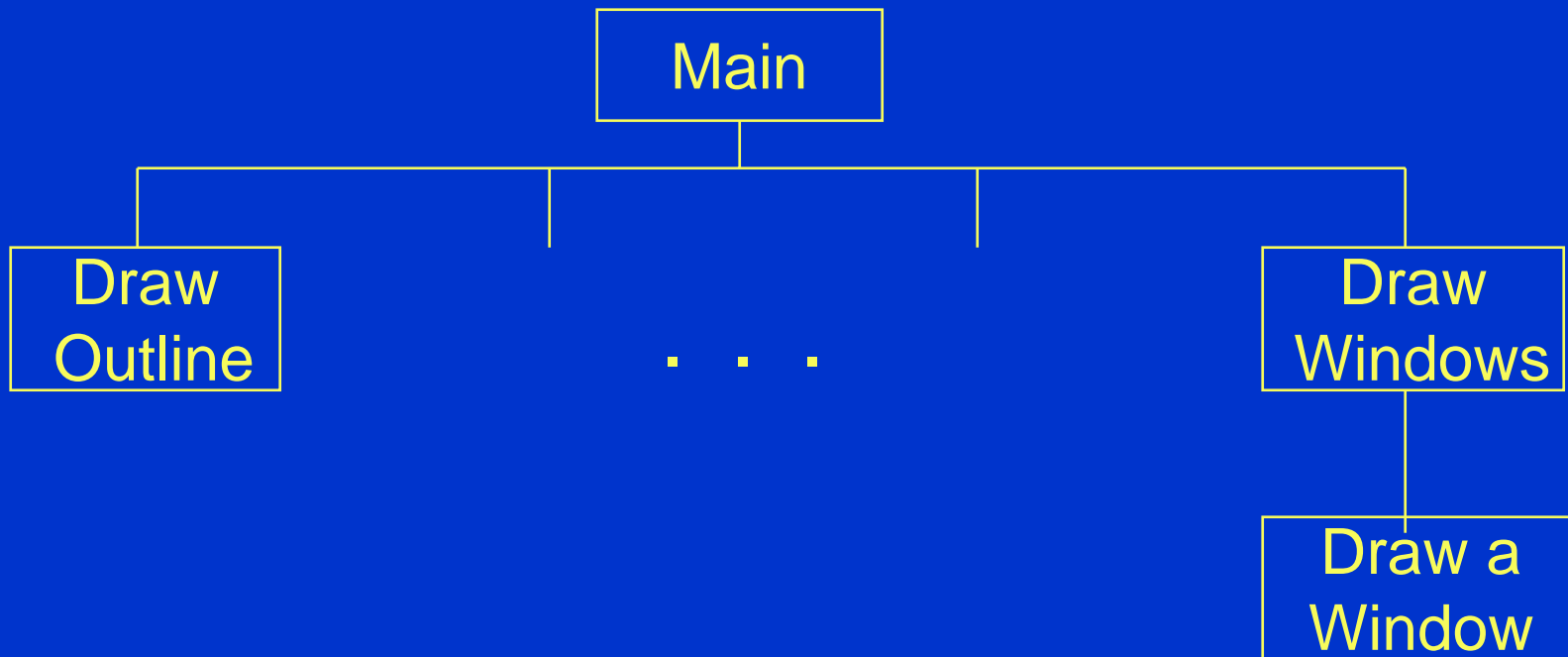


# One Last Observation

- But don't the windows look the same? They just have different locations.
- So, we can reuse the code that draws a window.
  - Simply copy the code three times and edit it to place the window in the correct location, or
  - Use the code three times, “sending it” the correct location each time (we will see how to do this later).
- This is an example of **code reuse**.



# Reusing the Window Code





# Pseudocode for Draw Windows

Call Draw a Window, sending in Location 1

Call Draw a Window, sending in Location 2

Call Draw a Window, sending in Location 3



# Review of Structured Programming

- Structured programming is a problem solving strategy and a programming methodology that includes the following guidelines:
  - The program uses only the sequence, selection, and repetition control structures.
  - The flow of control in the program should be as simple as possible.
  - The construction of a program embodies top-down design.



# Review of Top-Down Design

- Involves repeatedly **decomposing** a problem into smaller problems
- Eventually leads to a collection of small problems or tasks each of which can be easily coded
- The **function** construct in C is used to write code for these small, simple problems.





# Functions

- A C program is made up of one or more functions, one of which is `main( )`.
- Execution always begins with `main( )`, no matter where it is placed in the program. By convention, `main( )` is located before all other functions.
- When program control encounters a function name, the function is **called (invoked)**.
  - Program control passes to the function.
  - The function is executed.
  - Control is passed back to the calling function.



# Sample Function Call

```
#include <stdio.h>
```

```
int main ( )  printf is the name of a predefined  
{            function in the stdio library
```

```
    printf ("Hello World!\n") ;  
    return 0 ;  
}
```

this statement is  
is known as a  
**function call**

this is a string we are **passing**  
as an **argument (parameter)** to  
the printf function



# Functions (con't)

- We have used few predefined functions such as:
  - printf
  - scanf
  - getchar
- Programmers can write their own functions.
- Typically, each module in a program's design hierarchy chart is implemented as a function.
- C function names follow the same naming rules as C variables.



# Sample Programmer-Defined Function

```
#include <stdio.h>
```

```
void printMessage ( void ) ;
```

```
int main ( )
```

```
{
```

```
    printMessage ( ) ;
```

```
    return 0 ;
```

```
}
```

```
void printMessage ( void )
```

```
{
```

```
    printf ( "A message for you:\n\n" ) ;
```

```
    printf ( "Have a nice day!\n" ) ;
```

```
}
```



# Examining printMessage

```
#include <stdio.h>
```

```
void printMessage ( void ) ;
```




function **prototype**

```
int main ( )
```

```
{
```

```
    printMessage ( ) ;
```



function **call**

```
    return 0 ;
```

```
}
```

```
void printMessage ( void )
```



function **header**

```
{
```

```
    printf ("A message for you:\n\n") ;
```

```
    printf ("Have a nice day!\n") ;
```

```
}
```



function **body**



function **definition**



# The Function Prototype

- Informs the compiler that there will be a function defined later that:

returns this type



has this name



takes these arguments



`void printMessage (void) ;`

- Needed because the function call is made before the definition -- the compiler uses it to see if the call is made properly



# The Function Call

- Passes program control to the function
- Must match the prototype in name, number of arguments, and types of arguments

```
void printMessage (void) ;  
int main ( ) same name no arguments  
{  
    printMessage ( ) ;  
    return 0 ;  
}
```

Diagram illustrating the function call: Arrows point from the text *same name* to the function name `printMessage` in both the prototype and the call. Arrows point from the text *no arguments* to the empty parentheses `()` in both the prototype and the call.



# The Function Definition

- Control is passed to the function by the function call. The statements within the function body will then be executed.

```
void printMessage ( void )  
{  
    printf ("A message for you:\n\n") ;  
    printf ("Have a nice day!\n") ;  
}
```

- After the statements in the function have completed, control is passed back to the **calling function**, in this case `main( )` . Note that the calling function does not have to be `main( )` .





# General Function Definition Syntax

```
type functionName ( parameter1, . . . , parametern )  
{  
    variable declaration(s)  
    statement(s)  
}
```

If there are no parameters, either

functionName( ) OR functionName(void)

is acceptable.

- There may be no variable declarations.
- If the **function type (return type)** is void, a return statement is not required, but the following are permitted:

return ; OR return( ) ;



# Using Input Parameters

```
void printMessage (int counter) ;  
int main ( )  
{  
    int num;  
    printf ("Enter an integer: ") ;  
    scanf ("%d", &num) ;  
    printMessage (num) ;  
    return 0 ;  
}
```

one argument of type int matches the one **formal parameter** of type int

```
void printMessage (int counter)  
{  
    int i ;  
    for ( i = 0; i < counter; i++ )  
    {  
        printf ("Have a nice day!\n") ;  
    }  
}
```



# Final “Clean” C Code

```
#include <stdio.h>
```

```
void printMessage (int counter) ;
```

```
int main ( )
```

```
{
```

```
    int num ;    /* number of times to print message */
```

```
    printf (“Enter an integer: “) ;
```

```
    scanf (“%d”, &num) ;
```

```
    printMessage (num) ;
```

```
    return 0 ;
```

```
}
```



# Final “Clean” C Code (con’t)

```
/******  
** printMessage - prints a message a specified number of times  
** Inputs: counter - the number of times the message will be  
**           printed  
** Outputs: None  
/******/  
void printMessage ( int counter )  
{  
    int i ; /* loop counter */  
  
    for ( i = 0; i < counter; i++ )  
    {  
        printf (“Have a nice day!\n”) ;  
    }  
}
```



# Good Programming Practice

- Notice the **function header comment** before the definition of function `printMessage`.
- Your header comments should be neatly formatted and contain the following information:
  - function name
  - function description (what it does)
  - a list of any input parameters and their meanings
  - a list of any output parameters and their meanings
  - a description of any special conditions



# Functions Can Return Values

```
/******  
** averageTwo - calculates and returns the average of two numbers  
** Inputs: num1 - an integer value  
**          num2 - an integer value  
** Outputs: the floating point average of num1 and num2  
*****/  
float averageTwo (int num1, int num2)  
{  
    float average ; /* average of the two numbers */  
  
    average = (num1 + num2) / 2.0 ;  
    return average ;  
}
```



# Using averageTwo

```
#include <stdio.h>
float averageTwo (int num1, int num2) ;
int main ( )
{
    float ave ;
    int value1 = 5, value2 = 8 ;
    ave = averageTwo (value1, value2) ;
    printf ("The average of %d and %d is %f\n", value1, value2, ave) ;
    return 0 ;
}

float averageTwo (int num1, int num2)
{
    float average ;

    average = (num1 + num2) / 2.0 ;
    return average ;
}
```



# Parameter Passing

- **Actual parameters** are the parameters that appear in the function call.

average = averageTwo (**value1**, **value2**) ;

- **Formal parameters** are the parameters that appear in the function header.

float averageTwo (int **num1**, int **num2**)

- Actual and formal parameters are matched by position. Each formal parameter receives the value of its corresponding actual parameter.





# Parameter Passing (con't)

- Corresponding actual and formal parameters do not have to have the same name, but they may.
- Corresponding actual and formal parameters must be of the same data type, with some exceptions.



# Local Variables

- Functions only “see” (have access to) their own **local variables**. This includes `main( )` .
- Formal parameters are declarations of local variables. The values passed are assigned to those variables.
- Other local variables can be declared within the function body.



# Parameter Passing and Local Variables

```
#include <stdio.h>
float averageTwo (int num1, int num2) ;
int main ( )
{
    float ave ;
    int value1 = 5, value2 = 8 ;

    ave = averageTwo (value1, value2) ;
    printf ("The average of ") ;
    printf ("%d and %d is %f\n",
            value1, value2, ave) ;
    return 0 ;
}
```

value1

value2

ave

5

8

int

int

float

```
float averageTwo (int num1, int num2)
{
    float average ;

    average = (num1 + num2) / 2.0 ;
    return average ;
}
```

num1

num2

average

int

int

float

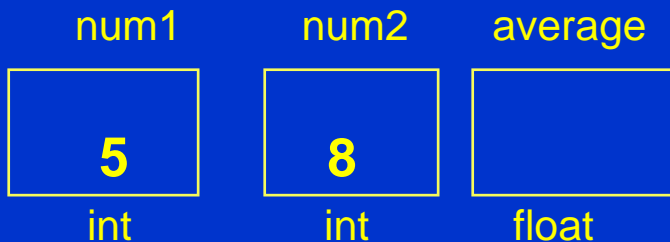


# Same Name, Still Different Memory Locations

```
#include <stdio.h>
float averageTwo (int num1, int num2) ;
int main ( )
{
    float average ;
    int num1 = 5, num2 = 8 ;

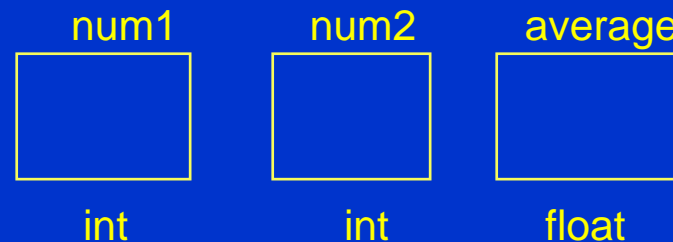
    average = averageTwo (num1,
                           num2) ;

    printf ("The average of ") ;
    printf ("%d and %d is %f\n",
            num1, num2, average) ;
    return 0 ;
}
```



```
float averageTwo (int num1, int num2)
{
    float average ;

    average = (num1 + num2) / 2.0 ;
    return average ;
}
```





# Changes to Local Variables Do **NOT** Change Other Variables with the Same Name

```
#include <stdio.h>
```

```
void addOne (int number) ;
```

```
int main ( )
```

```
{
```

```
    int num1 = 5 ;
```

```
    addOne (num1) ;
```

```
    printf ("In main: ") ;
```

```
    printf ("num1 = %d\n", num1) ;
```

```
    return 0 ;
```

```
}
```

num1

5

int

```
void addOne (int num1)
```

```
{
```

```
    num1++ ;
```

```
    printf ("In addOne: ") ;
```

```
    printf ("num1 = %d\n", num1) ;
```

```
}
```

num1

int

**OUTPUT**

In addOne: num1 = 6

In main: num1 = 5



# Call by Value and Call by Reference

By **Default** the function calling is by **Value** i.e. there is no relation between actual and formal parameter. Change in formal parameter doesn't affect actual parameter.

In case if we require the value updated by a function, there is a provision for single updated value by using return type of the function. But in case we require more than one value then we must call the function by reference.

In case of call by **reference** we use pointers.



# Header Files

- Header files contain function prototypes for all of the functions found in the specified library.
- They also contain definitions of constants and data types used in that library.



# Commonly Used Header Files

## Header File

## Contains Function Prototypes for:

<stdio.h>

standard input/output library functions and information used by them

<math.h>

math library functions

<stdlib.h>

conversion of numbers to text, text to numbers, memory allocation, random numbers, and other utility functions

<time.h>

manipulating the time and date

<ctype.h>

functions that test characters for certain properties and that can convert case

<string.h>

functions that manipulate character strings





# Using Header Files

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main ( )
{
    float side1, side2, hypotenuse ;
    printf("Enter the lengths of the right triangle sides: ") ;
    scanf("%f%f", &side1, &side2) ;
    if ( (side1 <= 0) || (side2 <= 0) {
        exit (1) ;
    }
    hypotenuse = sqrt ( (side1 * side1) + (side2 * side2) ) ;
    printf("The hypotenuse = %f\n", hypotenuse) ;
    return 0 ;
}
```



# User Defined Header Files

Save different functions in a file (extension .h). This file can be used as header file. e.g. myheader.h

In order to include the this user defined header file we can

- Either put the file in the ***include*** folder.
- Or keep it in the current folder and include it with  
`#include "myheader.h"`



# The stack and the heap

- Local variables, function arguments, return value are stored on a stack
- Each function call generates a new "stack frame"
- After function returns, stack frame disappears
  - along with all local variables and function arguments for that invocation



# The stack and the heap

```
void contrived_example(int i, float f)
{
    int j = 10;
    double d = 3.14;
    int arr[10];
    /* do some stuff, then return */
    return (j + i);
}
```



# The stack and the heap

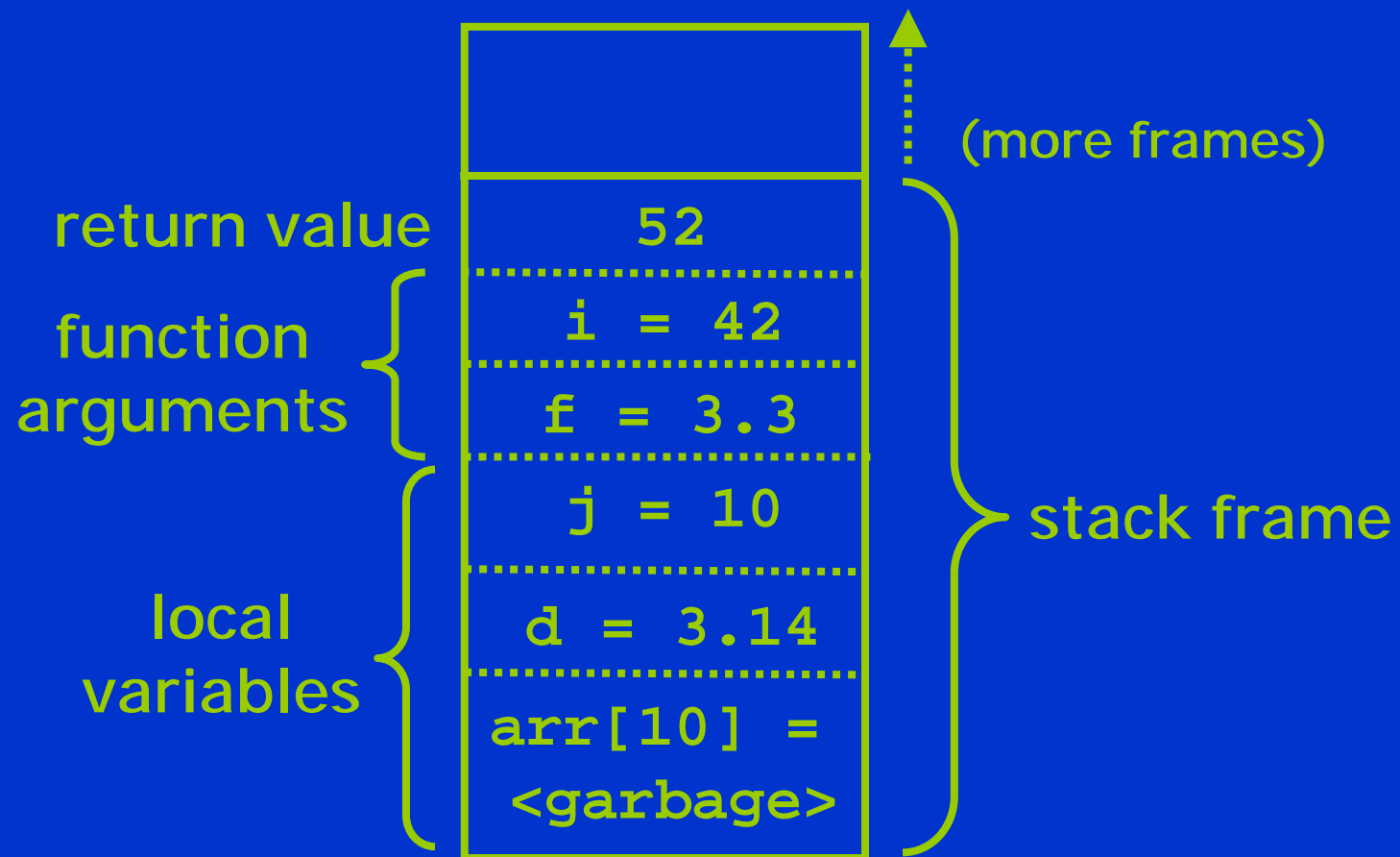
```
/* somewhere in code */
```

```
int k = contrived_example(42, 3.3);
```

- What does this look like on the stack?



# The stack and the heap





# The stack and the heap

- Another example:

```
int factorial(int i)
{
    if (i == 0) {
        return 1;
    } else {
        return i * factorial (i - 1);
    }
}
```



# The stack and the heap

- Pop quiz: what goes on the stack for `factorial(3)`?
- For each stack frame, have...
  - no local variables
  - one argument (`i`)
  - one return value
- Each recursive call generates a new stack frame
  - which disappears after the call is complete

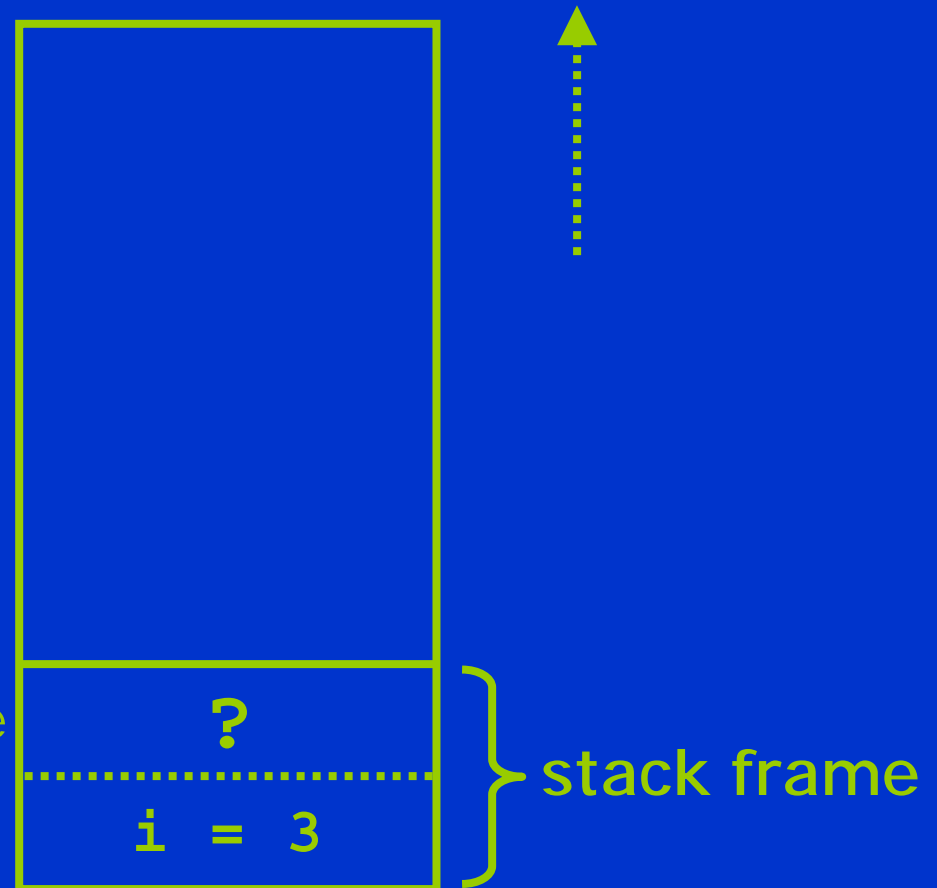




# The stack and the heap

`factorial(3)`

return value





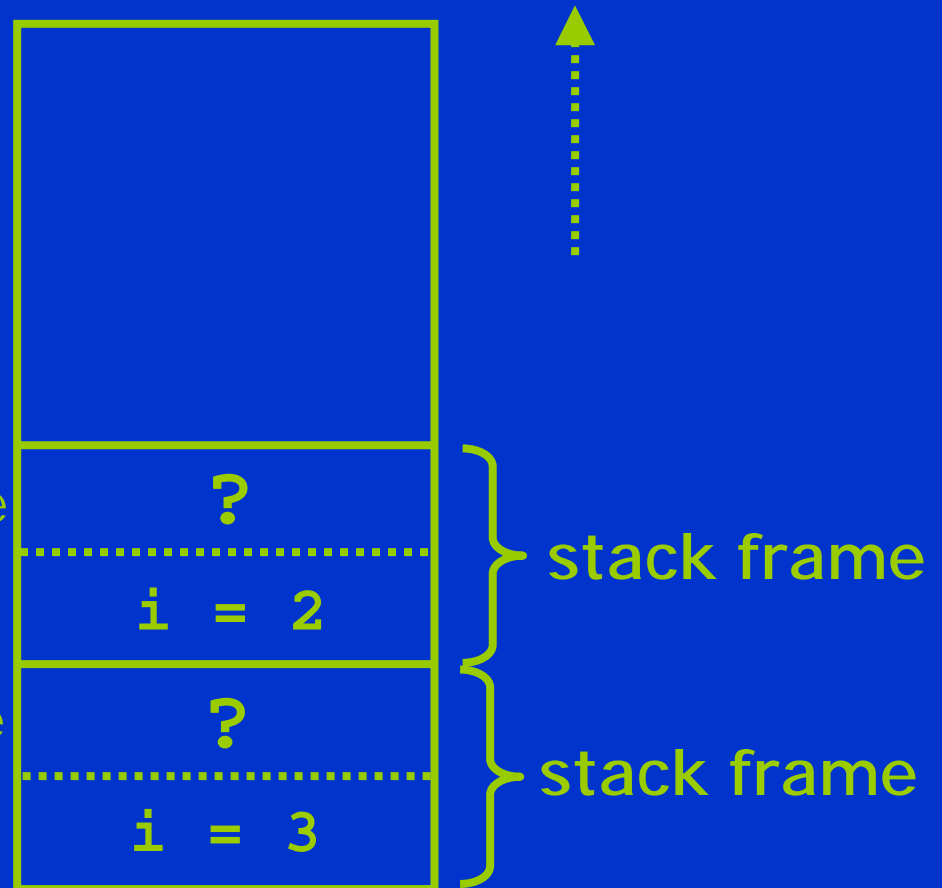
# The stack and the heap

`factorial(2)`

`factorial(3)`

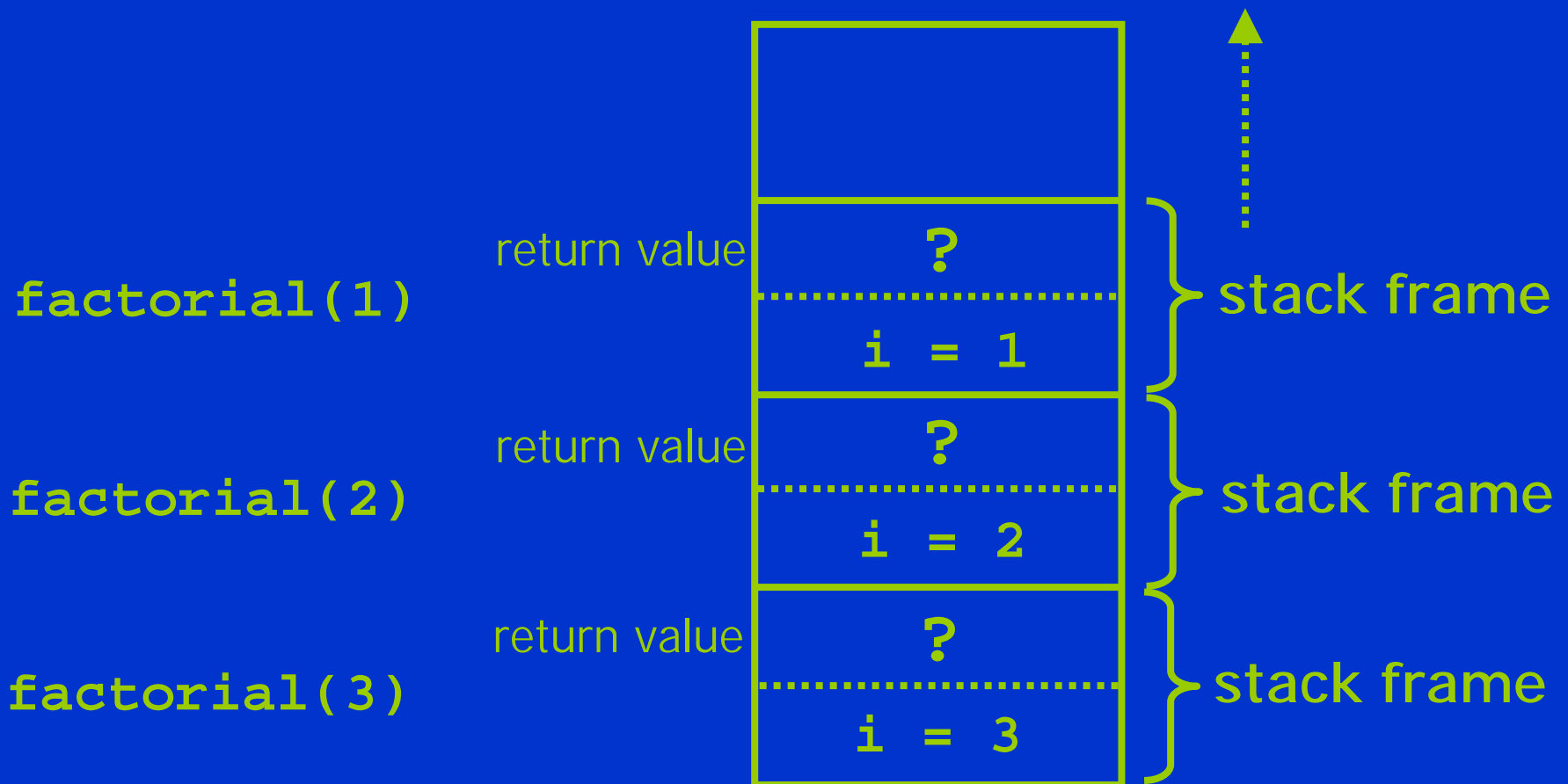
return value

return value





# The stack and the heap





# The stack and the heap

`factorial(0)`

return value

?

`i = 0`

} stack frame

`factorial(1)`

return value

?

`i = 1`

} stack frame

`factorial(2)`

return value

?

`i = 2`

} stack frame

`factorial(3)`

return value

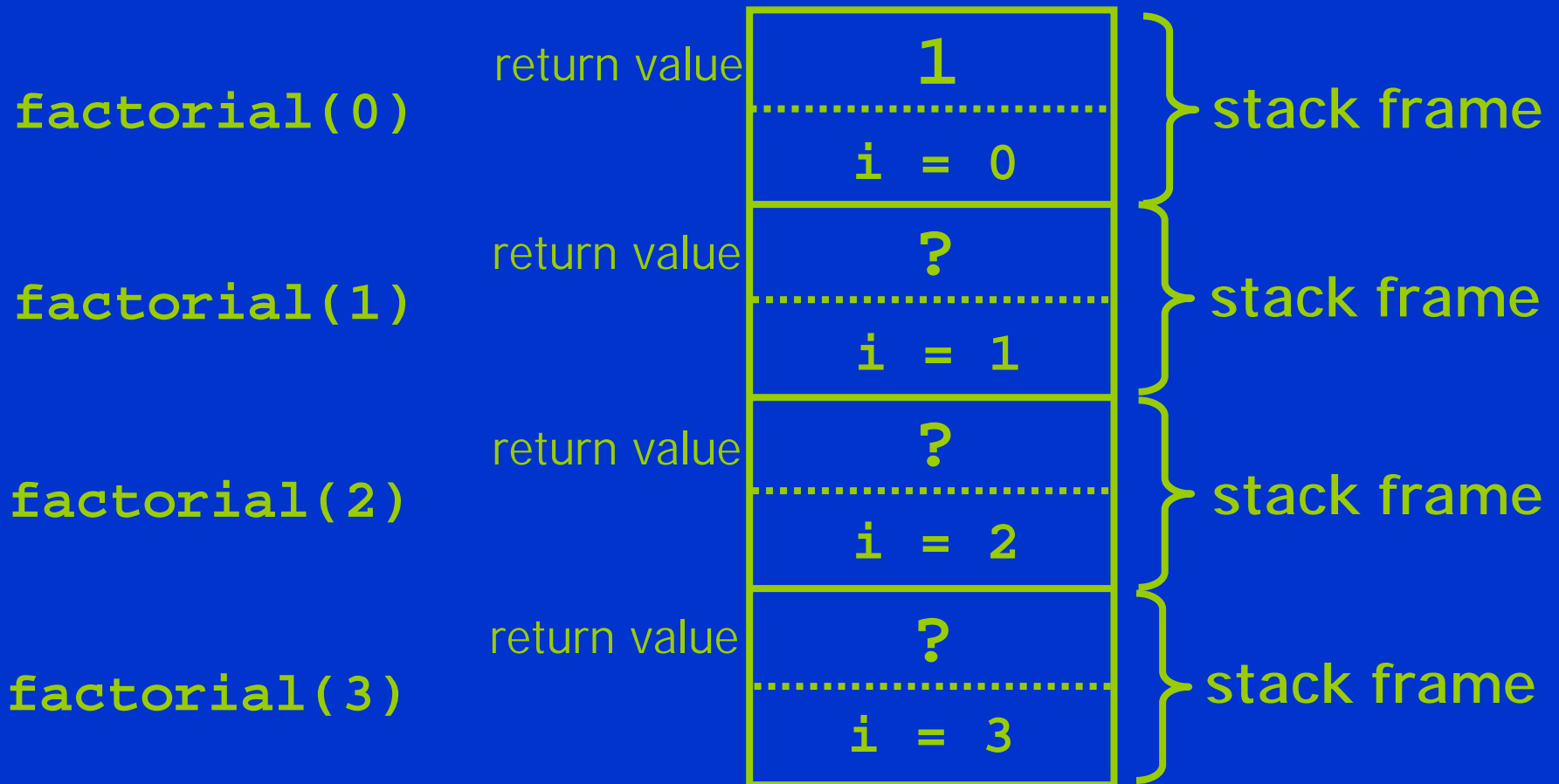
?

`i = 3`

} stack frame



# The stack and the heap





# The stack and the heap

`factorial(1)`

return value

1

`i = 1`

} stack frame

`factorial(2)`

return value

?

`i = 2`

} stack frame

`factorial(3)`

return value

?

`i = 3`

} stack frame



# The stack and the heap

`factorial(2)`

return value

2

`i = 2`

} stack frame

`factorial(3)`

return value

?

`i = 3`

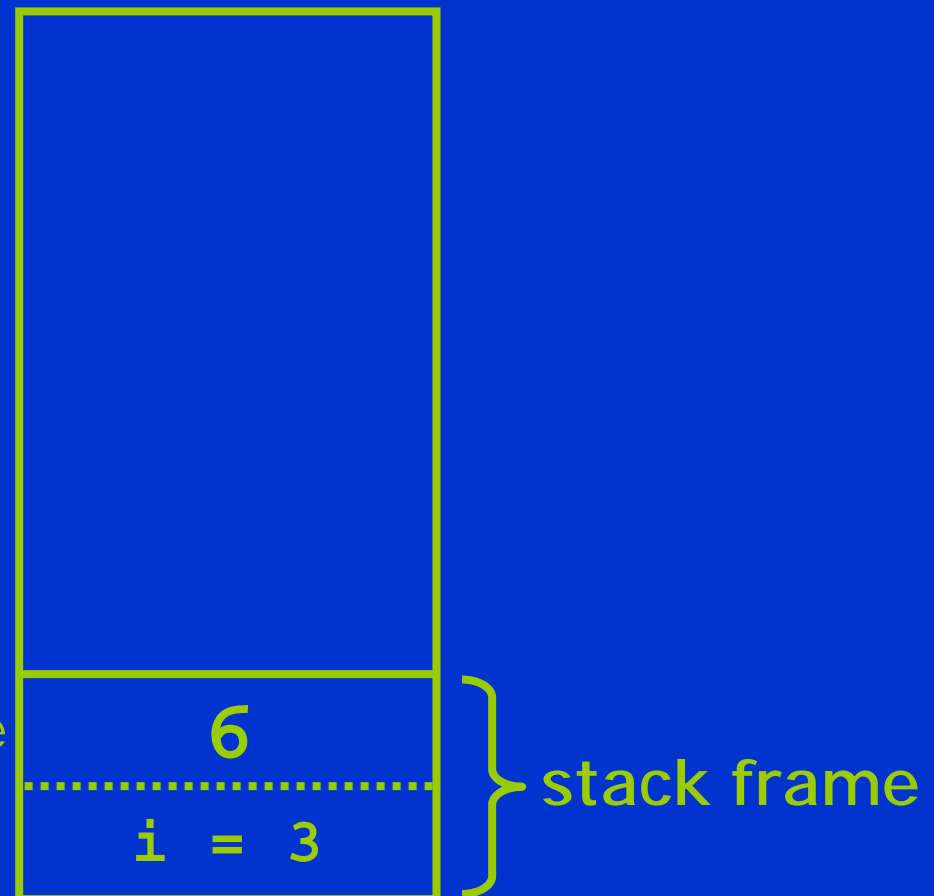
} stack frame



# The stack and the heap

`factorial(3)`

return value



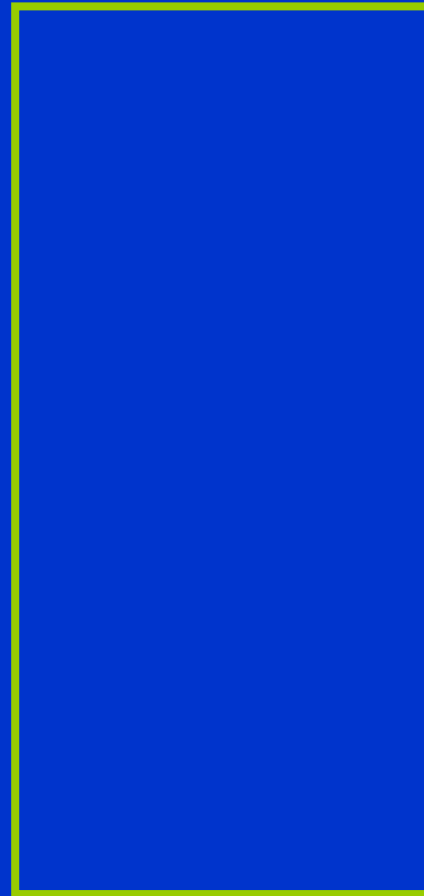




# The stack and the heap

`factorial(3)`

result: 6





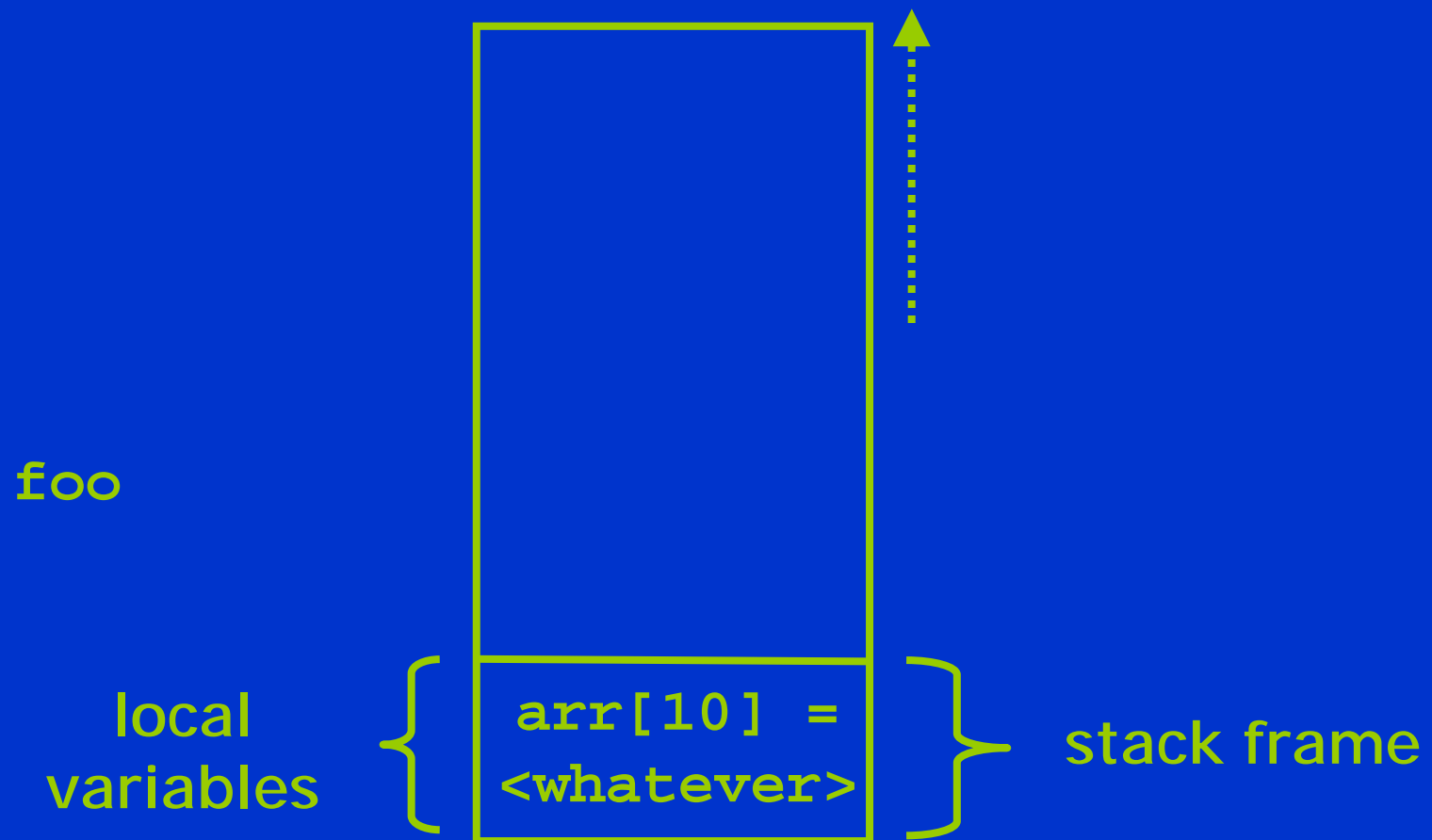
# The stack and the heap

```
void foo() {  
    int arr[10]; /* local (on stack) */  
    /* do something with arr */  
} /* arr is deallocated */
```

- Local variables sometimes called "automatic" variables; deallocation is automatic



# The stack and the heap





# The stack and the heap

- The "heap" is the general pool of computer memory
- Memory is allocated on the heap using `malloc()` or `calloc()`
- Heap memory must be explicitly freed using `free()`
- Failure to do so → memory leak!



# The stack and the heap

```
void foo2() {  
    int *arr;  
  
    /* allocate memory on the heap: */  
    arr = (int *)calloc(10, sizeof(int));  
  
    /* do something with arr */  
}  
/* arr is NOT deallocated */
```

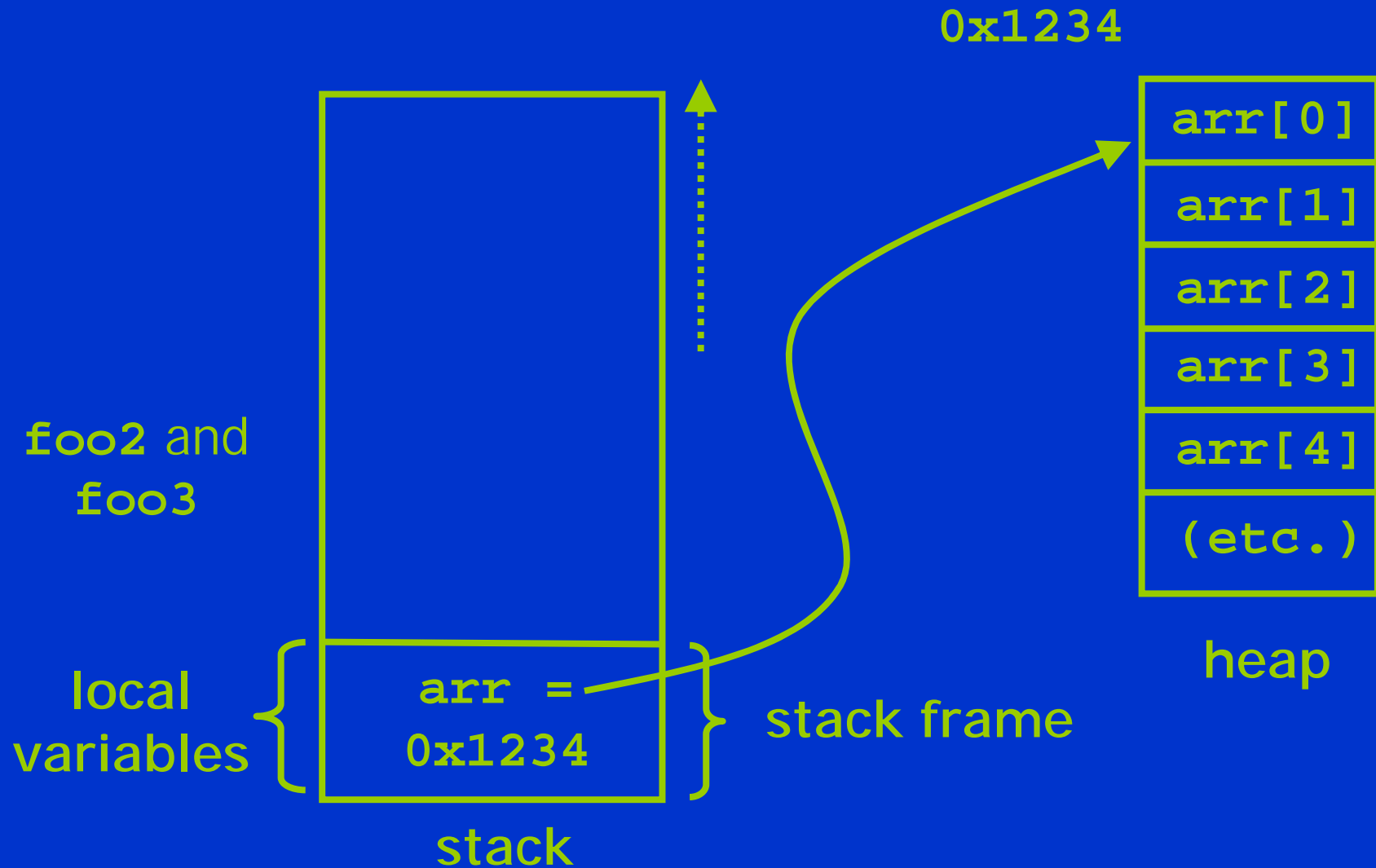


# The stack and the heap

```
void foo3() {  
    int *arr;  
  
    /* allocate memory on the heap: */  
    arr = (int *)calloc(10, sizeof(int));  
  
    /* do something with arr */  
  
    free(arr);  
  
}
```



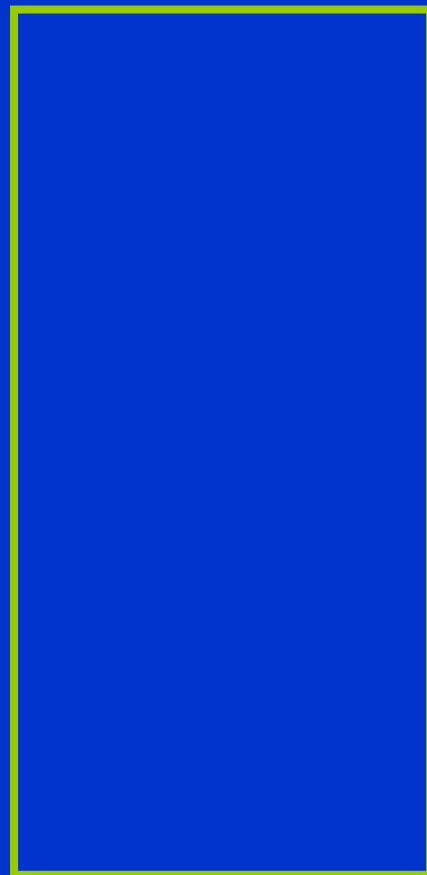
# The stack and the heap





# The stack and the heap

(after `foo2`  
exits,  
without  
freeing  
memory)

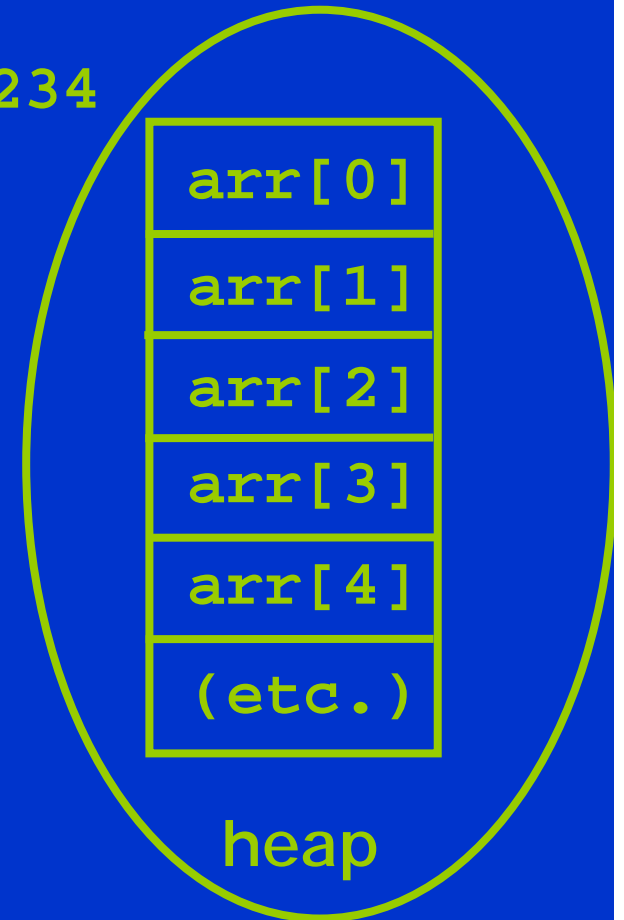


stack



memory  
leak

0x1234

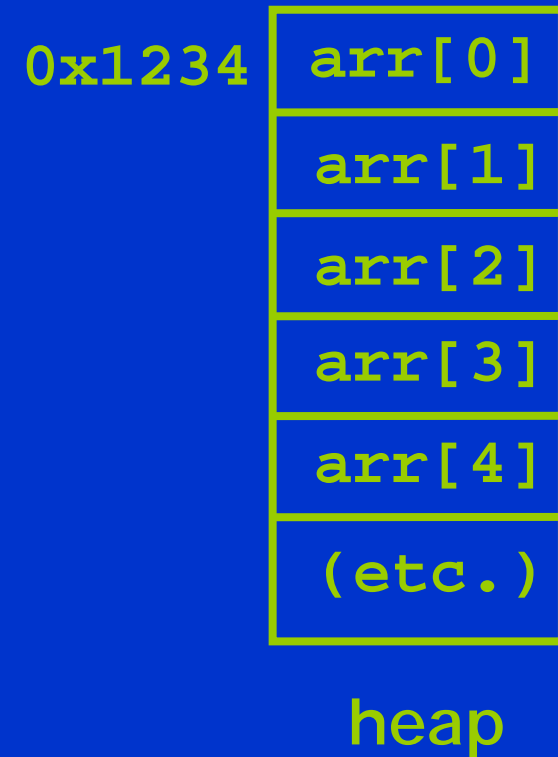
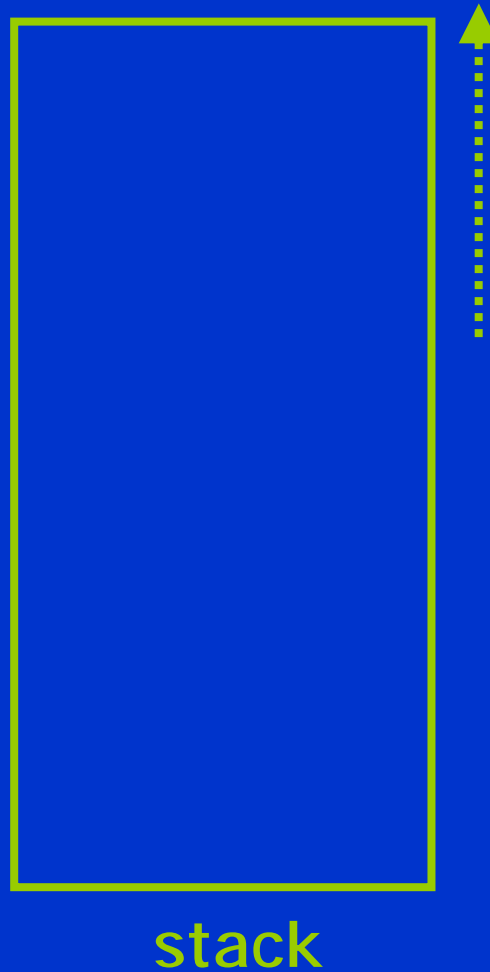






# The stack and the heap

(after `foo3`  
exits, with  
freeing  
memory)





# Thank You

