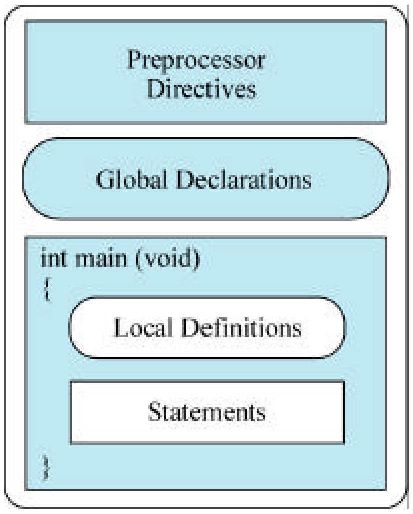
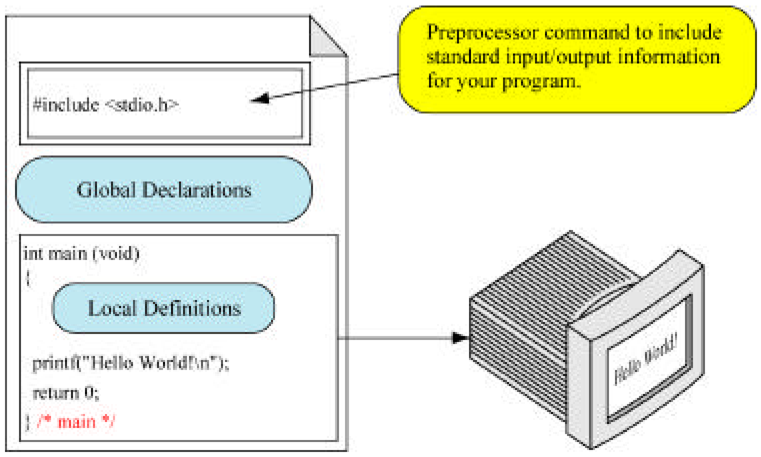
Fundamentals of C

COMP 1002/1402

Structure of a C Program



First Program


Preprocessor command to include standard input/output information for your program.
Global Declarations
Local Definitions
Hello World!

```
#include <stdio.h>

int main (void)
{

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

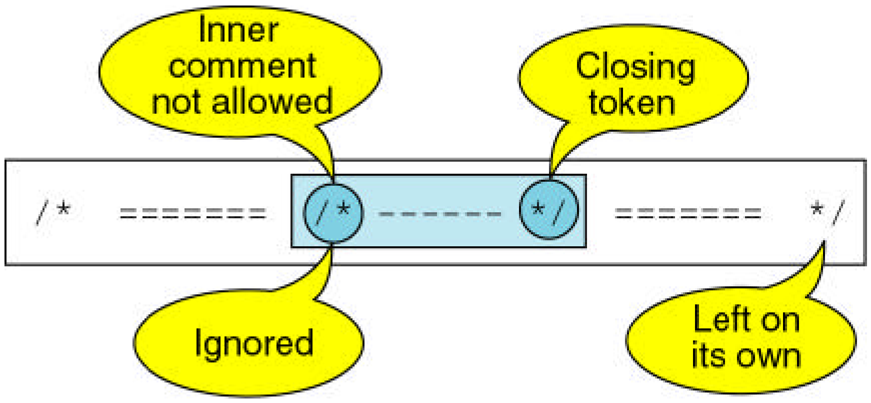
Comments

```
/*    This is a comment. */

/*    This is a comment that
      covers two lines.      */

/*
      It is a very common style to put the opening token
      on a line by itself, followed by the documentation
      and then the closing token on a separate line. Some
      programmers also like to put asterisks at the beginning
      of each line to clearly mark the comment.
*/
```

Comments



Pre-processor Directives

Preprocessor directives start with #

#include copies a file into the source code

```
#include <systemFilename>

#include “undefinedFilename”
```

#include <stdio.h>

- stdio.h is the standard input/output header
- .h files are header files
- Header files contain definitions

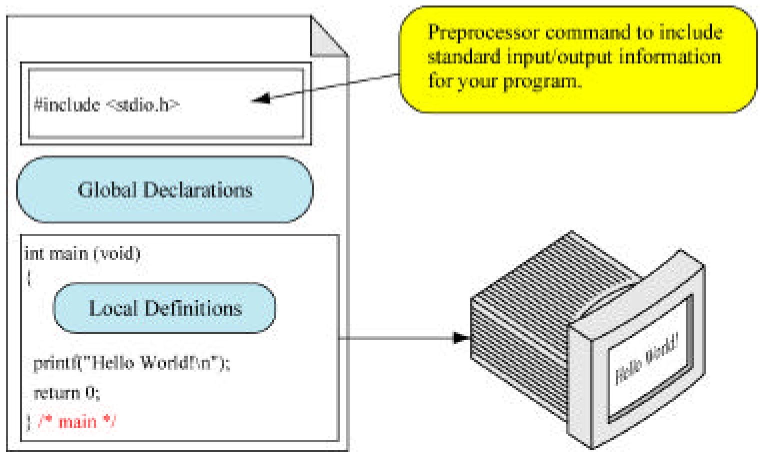
The main() function

Programs must have a main() function:

Two allowed formats:

```
int main( void )
int main(int argc,
         char *argv[])
```

First Program


Preprocessor command to include standard input/output information for your program.
#include <stdio.h>
Global Declarations
int main (void)
{
Local Definitions
printf("Hello World!\n");
return 0;
} /* main */
Hello World!

Variables

A variable is a block of memory that stores data of a particular type and is named with an appropriate identifier.

- An **`int number_of_days`** might begin at A000
- The **`int`** is 4 bytes:

```
A000        10101001
A001        00000000
A002        10101010
A003        11110010
```

Variables

- The name of the variable corresponds to the address of the first byte!
- The machine remembers the address and knows that an **`int`** is four bytes
- Knowing the address is very important in C

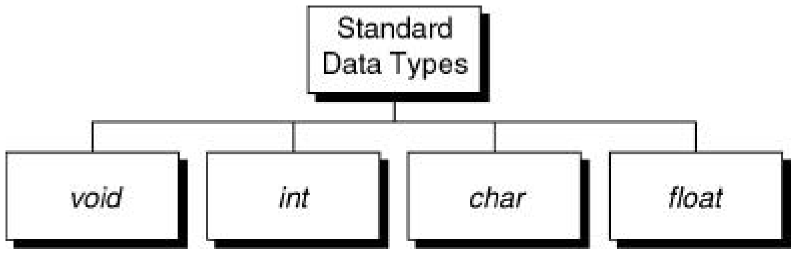
Rules for Naming Variables

- First character: alphabetic or underscore
- Consist only of alphanumeric or underscores
- Only first 31 characters count
- Cannot duplicate a reserved word

Legal/Illegal Names

Legal	Illegal
a	$sum
student_name	2names
TRUE	stdnt number
FALSE	int

Standard Data Types



Primitive Data Types

Data Type	**C-Implementation**
void	**`void`**
character	**`char`** (1 byte)
integer	**`unsigned short int`** (1 byte) **`unsigned int`** (2 or 4 bytes) **`unsigned long int`** (4 or 8 bytes)
	`short int` (1 byte) **`int`** (2 or 4 bytes) **`long int`** (4 or 8 bytes)
floating point	**`float`** (4 bytes) **`double`** (8 bytes) **`long double`** (10 bytes)

Logical Data

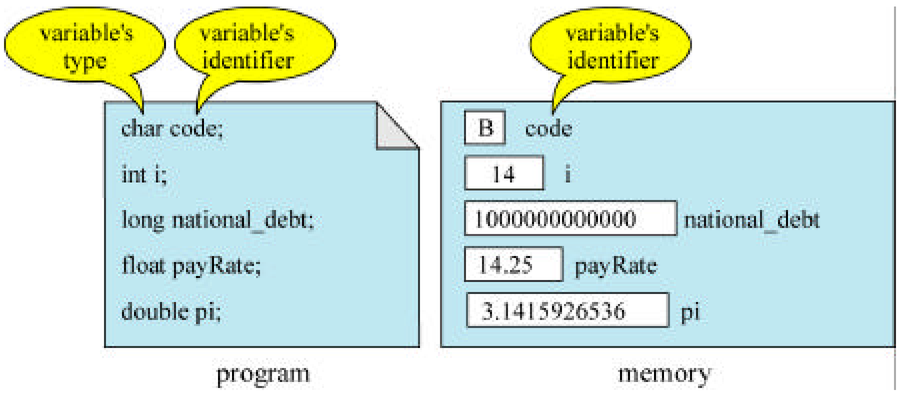
No Boolean data type!!

In C:

any nonzero number is **true**

zero is **false**

Variable Declaration



Variable Initialization

- No variable is initialized until you do so!

```
char code = ‘B’;
int i = 0;
long national_debt = 2931000001L;
unsigned long gnp = 332000101LU;
float aVariable = 3.1415f;
double variable2 = 3.1415926535;
long double variable3 = 3.14159265358979L;
```

Special Characters

ASCII Character	**Symbolic Name**
Null character	`’\0’`
Alert (bell)	`’\a’`
Backspace	`’\b’`
Horizontal tab	`’\t’`
Newline	`’\n’`
Vertical Tab	`’\v’`

Special Characters

ASCII Character	**Symbolic Name**
Form Feed	`’\f’`
Carriage return	`’\r’`
Single quote	`’\’’`
Double Quote	`’\”’`
Backslash	`’\\’`

String Constants

```
""                    /* A null string */
"h"
"Hello World\n"
"HOW ARE YOU"
"Good Morning!"
```

#define

`#define name token(s)`

Replaces the **`name`** with the **`token(s)`**

Example:

```
#define PI 3.1415926535
#define SIZE 1000
```

#define

- No equals sign
- No semicolon
- Multiple lines require \ at end of line

Constants

To define a constant define a variable with:

```
const
```

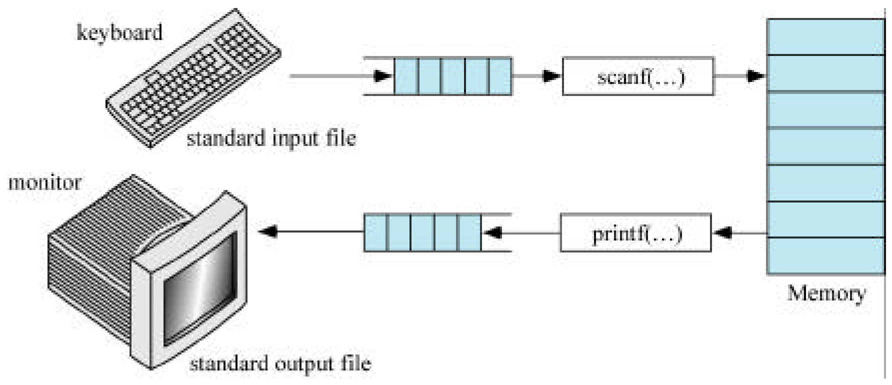
Works:

```
const float pi = 3.1415926;
```

Doesn’t:

```
const float pi;
pi=3.1415926; /* not allowed to change it */
```

Standard Input and Output



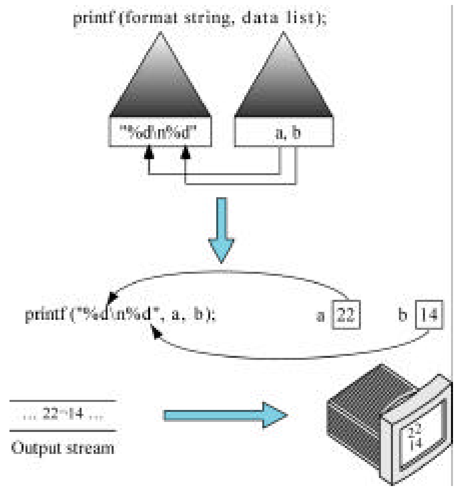
Output

Requires: **`#include <stdio.h>`**

```
printf(format string, data list);
```

Field specifiers inside the format string

`printf` statements



Field Specifiers

`%<flag><minimum width><precision><size>code`

Codes:

Size	Code	Type	Example
none	c	char	%c
h	d	short int	%hd
none	d	int	%d
l or L	d	long int	%Ld
none	f	float	%f
none	f	double	%f
L	f	long double	%Lf

Width

Value	%d	%4d
`12`	`12`	`  12`
`123`	`123`	` 123`
`1234`	`1234`	`1234`
`12345`	`12345`	`12345`

Precision and Flag

Precision:

```
%7.2f   /* float-7 print positions: nnnn.dd */
```

Flag:

```
%-8d    /* left justify flag */
%08d    /* leading zeroes flag */
```

`printf` examples

```
printf(“%06d %c\n%6.3f“,23,’A’,4.23);

000023 A
 4.230

printf(“These are \“\“ double quotes“);

These are ““ double quotes
```

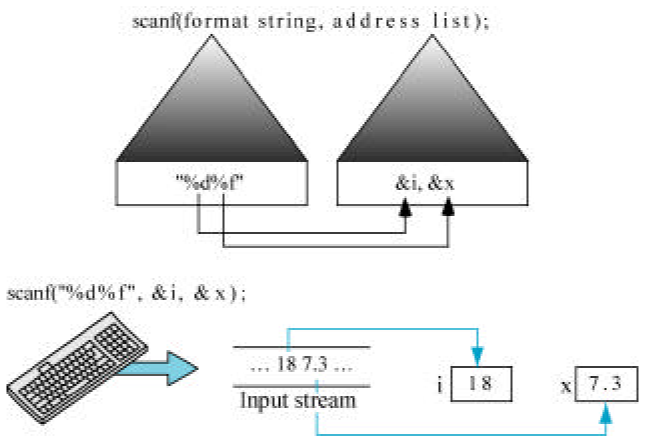
Input

Requires: **`#include <stdio.h>`**

`scanf(format string, address list);`

Field specifiers inside the format string

`scanf` Statements



Field Specifiers

`%<flag><maximum width><size>code`
`(no precision!!)`

Codes:

Size	Code	Type	Example
none	c	char	%c
h	d	short int	%hd
none	d	int	%d
l or L	d	long int	%Ld
none	f	float	%f
none	f	double	%f
L	f	long double	%Lf

Rules

Fields are converted to specific addresses

Addresses of a variable are specified with:

```
&variableName
```

A variety of rules apply to conversion

Rules

- Initial whitespace is ignored (not %c)
- The conversion operation process until:
  - End of file is reached
  - Maximum characters are processed
  - A whitespace character is found after a digit
  - An error is detected

Rules

- A field specifier for each variable
- Other characters must be exactly matched
- Cannot end format string with whitespace

`scanf` examples

```
scanf(“%d%d%d%c”,&a,&b,&c,&d);

scanf(“%d%d%d %c”,&a,&b,&c,&d);

scanf(“%d-%d-%d”,&a,&b,&c);
```

Expressions

An expression is a sequence of operands and operators that reduces to a single value.

2 + 5

Is an expression that evaluates to 7

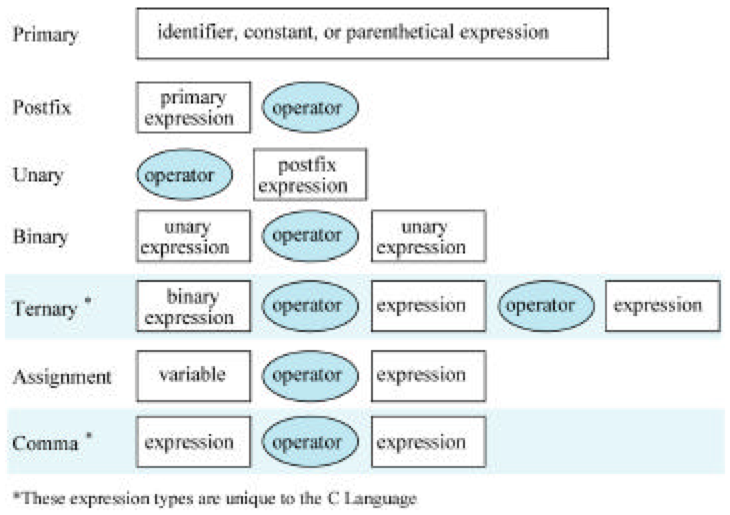
Operators

An **Operator** is a token that requires action.
e.g., + - * / %

An **Operand** receives an operators action
e.g., multiplier * multiplicand

In following diagram of expressions:
rectangles are operands

C Expressions



Primary Expressions

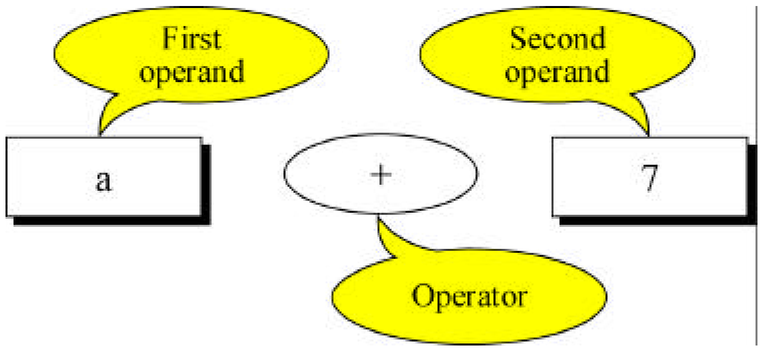
Identifiers, constants (already covered)

Parenthetical expressions:

```
(2 * 3 + 4)  (a = 23 + b * 6)
```

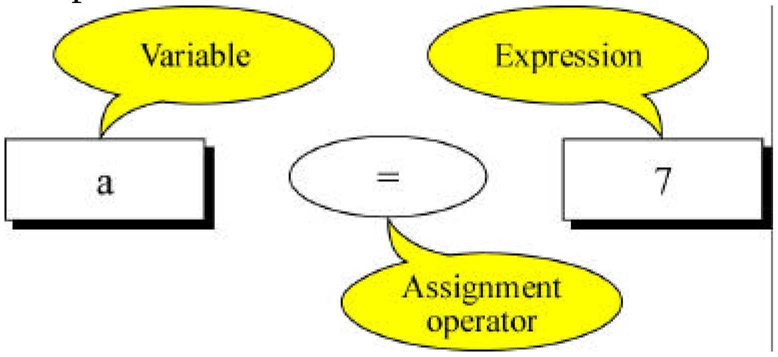
Binary Expressions

Examples: + - * / %



Assignment

Assignment expressions evaluate to the expression on the right of the assignment operator.



Simple Assignment

Contents of Variable x	Contents of Variable y	Expression	Value of Expression	Result of Expression
`10`	`5`	`x=y+2`	`7`	`x = 7`
`10`	`5`	`x=x/y`	`2`	`x=2`
`10`	`5`	`x=y%4`	`1`	`x=1`

Compound Assignment

Contents of Variable x	Contents of Variable y	Expression	Value of Expression	Result of Expression
`10`	`5`	`x *= y`	`50`	`x = 50`
`10`	`5`	`x /= y`	`2`	`x =  2`
`10`	`5`	`x %= y`	`0`	`x =  0`
`10`	`5`	`x += y`	`15`	`x = 15`
`10`	`5`	`x -= y`	`5`	`x =  5`

Postfix Expressions

Contents of `x` Before	Expression	Value of Expression	Contents of `x` After
`10`	`x++`	`10`	`11`
`10`	`x--`	`10`	`9`

Unary Expressions

Contents of `x` Before	Expression	Value of Expression	Contents of `x` After
`10`	`++x`	`11`	`11`
`10`	`--x`	`9`	`9`

Unary Operator **`sizeof`**

`sizeof` is an operator (not a function)

Evaluates to number of bytes for that item

```
sizeof(int)
sizeof(x)
sizeof(3.256)
```

Unary operator + -

`+a` – evaluates to the contents of **`a`**

`-a` – evaluates to the negative contents of **`a`**