Course Notes Set 2:

COMP1200-001

Introduction to Computing for Engineers and Scientists

C Programming

Getting Started with C

Computer Science and Software Engineering

Auburn University

Naming variables

Choose a name that represents the information

“declare” all variables names near the beginning of module

No more than 31 characters

- 1st char: a-z, A-Z, _
- Other char: a-z, A-Z, _, 0-9
- Upper and lower case
- NO blanks

C is Case Sensitive

C reserved words called keywords cannot be used as a variable.

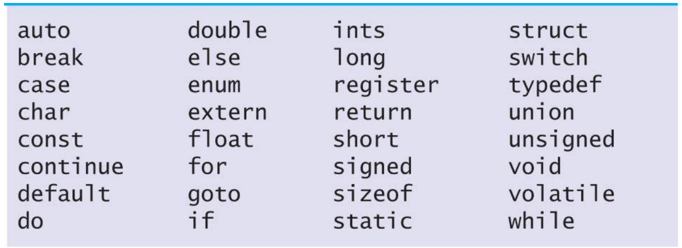
Instructor’s rules for Variable Names

– **BE DESCRIPTIVE**
  - Choose a name that represents the information

– **Camel backing or underscore**
  - `Ex. aveMonthlyRain` OR `ave_monthly_rain`

– **Constant variables are NOT changed**
  - all capital letters, ex. `MAX_ENROLLMENT = 100`

Keywords

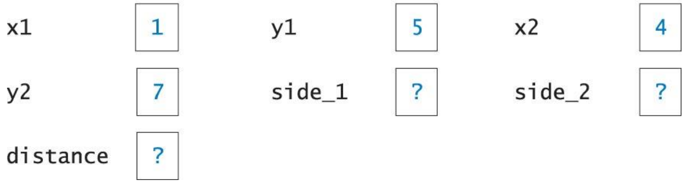


```
auto        double      ints        struct
break       else        long        switch
case        enum        register    typedef
char        extern      return      union
const       float       short       unsigned
continue    for         signed      void
default     goto        sizeof      volatile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

Etter, Engineering Problem Solving with C, Third Edition, © 2005 Pearson Education, Inc. All rights reserved. 0-13-142971-X

Constants and Variables

```
double x1=1, y1=5, x2=4, y2=7,
       side_1, side_2, distance;
```



Etter, Engineering Problem Solving with C, Third Edition, © 2005 Pearson Education, Inc. All rights reserved. 0-13-142971-X

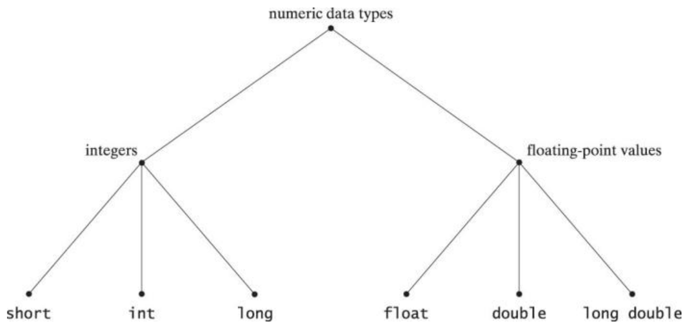
Constant values

Set by using a *constant macro*.

A *constant macro* is created by using a *preprocessor directive.*

```
#define BOILING_POINT_ F 212
#define FREEZING_POINT_F 32
#define PI 3.14149
#define COURSE_PAR 72
```

Numeric Data Types



Etter, Engineering Problem Solving with C, Third Edition, © 2005 Pearson Education, Inc. All rights reserved. 0-13-142971-X

Numeric Data

Type	Storage	Min Value	Max Value
short	16 bits	-32,768	32,767
int	32 bits	-2,147,483,648	2,147,483,647
long	64 bits	< -9 x 1018	> 9 x 1018
float	32 bits	+/- 3.4 x 1038 with 7 significant digits	32 bits = 8 exp + 24 mantissa
double	64 bits	+/- 1.7 x 10308 with 15 significant digits	64 bits = 11 exp + 53 mantissa

Arithmetic: operators

Operator Types

An assignment may also be an arithmetic computation such as:

```
x    = 45 * 20;
area = length * width;
c    = 3.12 * d;
```

Notation		Allowed Types	
*	multiplication	Any	
/	division	Any	
+	addition	Any	
-	subtraction	Any	
%	remainder	integer	8 % 3 ➔ 2

Arithmetic examples

```
int    a = 10,  b = 3,   c = 7, d;
double e = 5.1, f = 2.3, g;

d = a * b;          // 10 * 3 = 30

d = a * b - c;      // 10 * 3 - 7 = 23

d = c / b;          // 7 / 3 = 2 fraction truncated

g = e / a;          // 5.1 / 10 = 0.51
```

Arithmetic examples

```
int    a = 10,  b = 3,   c = 7, d;
double e = 5.1, f = 2.3, g;

d = -1;              // d = negative one

g = e * f;           // 5.1 * 2.3 = 11.73

g = -3.7 * e - f;    // -3.7 * 5.1 – 2.3 = -21.17

d = c % b;           // 7 % 3 = 1
```

C order of precedence

1. **Parenthesis**. Anything between parenthesis is evaluated before anything outside them. Evaluation is from the innermost set of parenthesis outward.

2. **Unary Operators + and -.** Operators that change the sign of an operand (i.e. -2.3, +23, -x). Don't confuse these with addition and subtraction.

3. **Binary Operators *, /, and %.** Multiplication, Division, and Modulus. Evaluation order is from left to right. That is, if any of these operators follow one another, the leftmost one is performed first.

4. **Binary Operators + and -.** Addition and Subtraction. Evaluation order is from left to right.

Precedence examples

```
int a = 3, b = 5, c = 7, d;

d = a - b * c;                // d =
d = (a - b) * c;              // d =
d = a * b / c;                // d =
d = a * (b / c);              // d =
```

Precedence examples

```
int a = 3, b = 5, c = 7, d;

d = a - b * c;                // d = - 32
d = (a - b) * c;              // d = -14
d = a * b / c;                // d = 2
d = a * (b / c);              // d = 0
```

Precedence examples

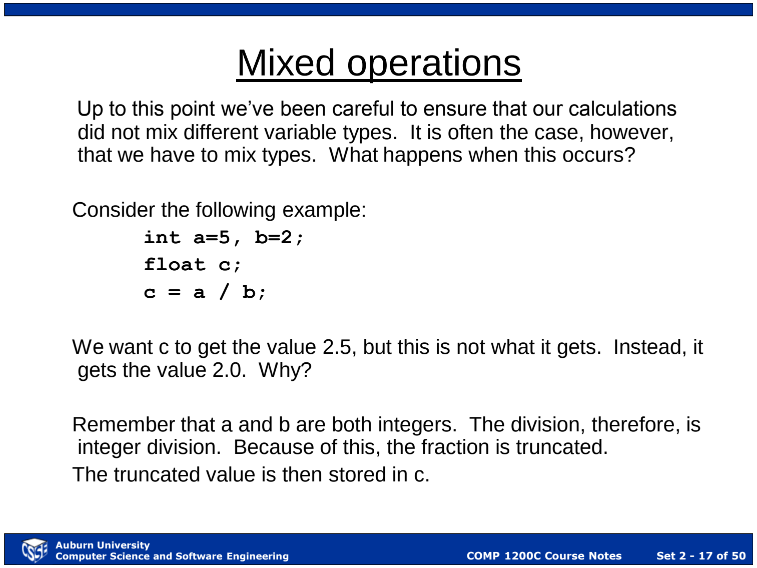
```
int a = 3, b = 5, c = 7, d;

d = a * -2;                   // d =
d = (a + 2 * b) * c;          // d =
d = ((a + 2) * b) * c;        // d =
```

Precedence examples

```
int a = 3, b = 5, c = 7, d;

d = a * -2;                   // d = -6
d = (a + 2 * b) * c;          // d = 91
d = ((a + 2) * b) * c;        // d = 175
```



Mixed operations

Up to this point we've been careful to ensure that our calculations did not mix different variable types. It is often the case, however, that we have to mix types. What happens when this occurs?

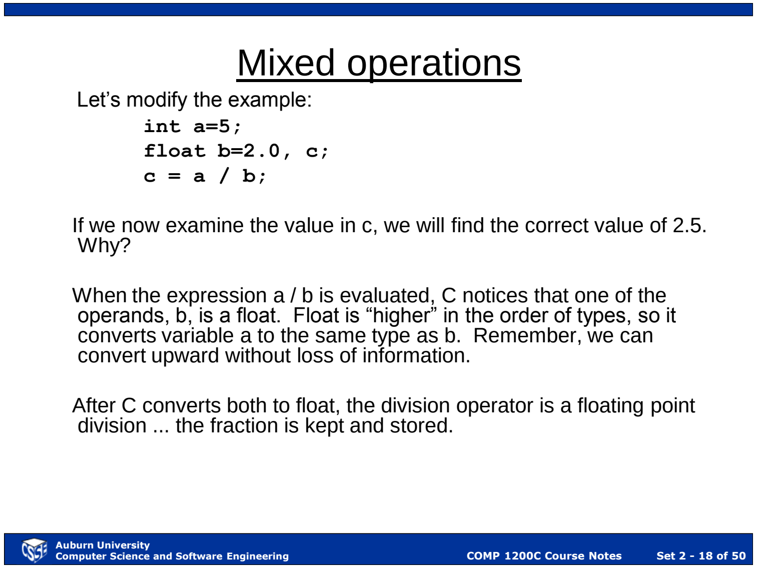
Consider the following example:

```
int a=5, b=2;
float c;
c = a / b;
```

We want c to get the value 2.5, but this is not what it gets. Instead, it gets the value 2.0. Why?

Remember that a and b are both integers. The division, therefore, is integer division. Because of this, the fraction is truncated.

The truncated value is then stored in c.



Mixed operations

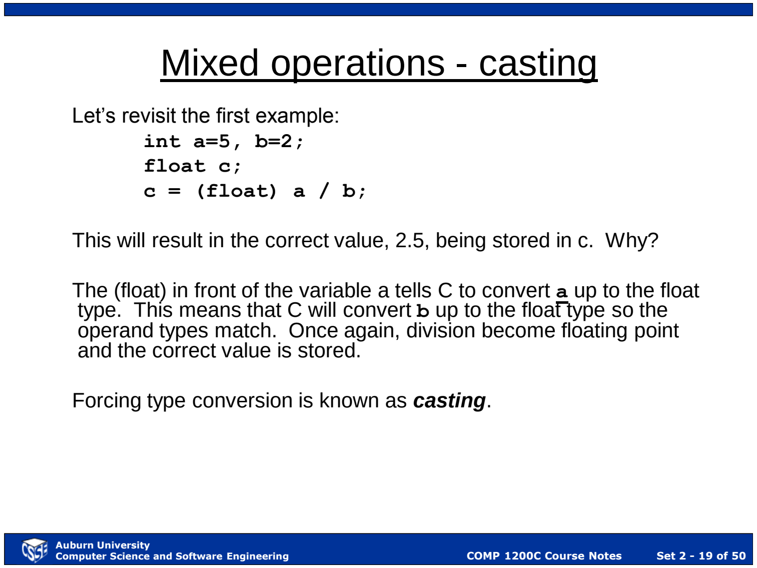
Let's modify the example:

```
int a=5;
float b=2.0, c;
c = a / b;
```

If we now examine the value in c, we will find the correct value of 2.5. Why?

When the expression a / b is evaluated, C notices that one of the operands, b, is a float. Float is "higher" in the order of types, so it converts variable a to the same type as b. Remember, we can convert upward without loss of information.

After C converts both to float, the division operator is a floating point division ... the fraction is kept and stored.



Mixed operations - casting

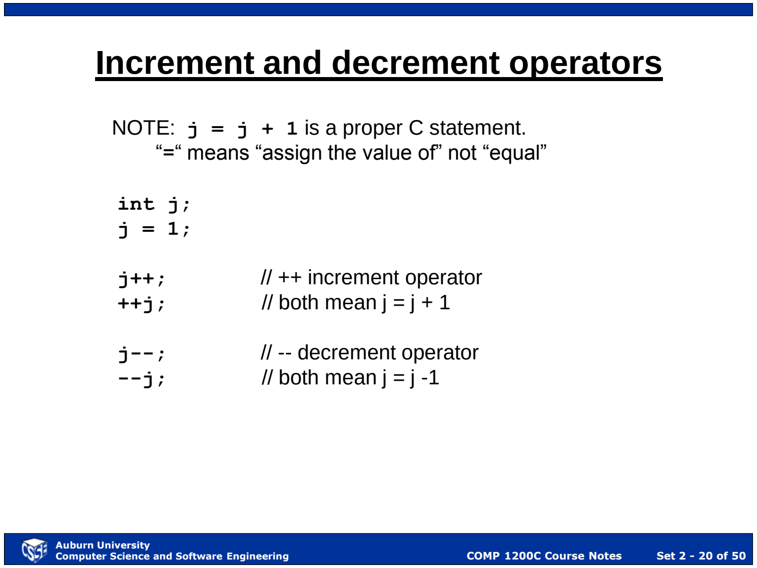
Let's revisit the first example:

```
int a=5, b=2;
float c;
c = (float) a / b;
```

This will result in the correct value, 2.5, being stored in c. Why?

The (float) in front of the variable a tells C to convert `a` up to the float type. This means that C will convert `b` up to the float type so the operand types match. Once again, division become floating point and the correct value is stored.

Forcing type conversion is known as ***casting***.



Increment and decrement operators

NOTE: `j = j + 1` is a proper C statement.
"=" means "assign the value of" not "equal"

```
int j;
j = 1;

j++;            // ++ increment operator
++j;            // both mean j = j + 1

j--;            // -- decrement operator
--j;            // both mean j = j -1
```

But, are they the same?

```
int h, i, j, k;
i = 1;
j = 1;
```

`k = i++;`
first k = i
then i++,
so now k is 1
and i is 2

`h = ++j;`
first j++
then h = j,
so now j is 2
and h is 2

Other shortcuts

`j = j + 7;` Can be written as: `j += 7;`

`k = k * i;` Can be written as: `k *= i;`

Updated Precedence Rules

1. Parenthesis.
2. Unary Operators +, -, ++, --.
3. Binary Operators *, /, and %.
4. Binary Operators + and -.
5. Assignment Operators =, +=, -=, *=, /=, %=.

Updated Precedence Rules

Assignment Operators =, +=, -=, *=, /=, %=.

```
x = x + 3;          d = d / 4.5;
x += 3;             d /= 4.5;
```

`x =+ 3;` no error, x = positive 3

`d =/ 4.5;` abbrev_assign.c:11:
error: parse error before '/' token

Functions

Functions in the stdio [stdio.h] library:

```
printf, fprintf, scanf, fscanf.
```

The math [math.h] library contains functions:

```
sqrt(x)        sin(x)
log(x)         cos(x)
sqrt(x)        tan(x)
pow(x,y)       log10(x)
   ...
```

C Program

Find the area of a circle

C program

1. Problem Statement

Given the radius of a circle, compute its area.

```
a = PI * r * r
```

Output both the radius and the area with two digits after the decimal point.

2. Input/Output Description

Information needed to solve the problem is the radius of the circle, and the value of . Output is the computed area and the radius.

Work Problem By Hand

Given	radius = 20.5, and PI = 3.1415963
Then:	area = PI * radius * radius
	area = 3.1415963 * 20.5 * 20.5
	area = 1320.2558451

C program

3. Algorithm

An algorithm is a step-by-step method for solving a problem. This problem is fairly simple and can be solved in only a few steps:

1. Get the radius
2. Compute the area of the circle
3. Print the radius and area of the circle.

Convert this algorithm into code

First, some comments:

```
/*
********************************************
  Program area.c

   Computes the area of a circle given its
   radius.
********************************************
*/
```

Now, the preprocessor declarations we'll need, first up is the include that gets our printf function:

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

Next we'll want a symbolic constant for to make reading the program easier:

```
#define PI 3.1415963
```

Convert this algorithm into code

Now we'll create a *main function* to hold the variables and executable code for our algorithm:

```
#include <stdio.h>
#define PI 3.1415963
int main()
{
  // Get the radius
  // Compute the area of the circle
  // Print the radius and area of the circle
  return 0;
}
```

Memory storage

In order to compute the result, we need memory storage locations for the radius of the circle, and the computed area. We'll need two variables: radius and area. Since floating point values are used, we'll make the of type *double*:

```
#include <stdio.h>
#define PI 3.1415963

int main()
{
  // Get the radius
  double radius=20.5;     // Radius of the circle
  double area;            // Computed area

  // Compute the area of the circle
  // Print the radius and area of the circle

   return 0;
}
```

Compute the area

Now we add the code which computes the area
(step 2 of the algorithm):

```
#include <stdio.h>
#define PI 3.1415963

int main()
{
  // Get the radius
  double radius=20.5;     // Radius of the circle
  double area;            // Computed area

  // Compute the circle's area
  area = PI * radius * radius;

  // Print the radius and area of the circle
  return 0;
}
```

Now we want to output the information.
We'll need a **`printf`** statement that prints the radius and the area:

printf to print output

```
#include <stdio.h>
#define PI 3.1415963

int main()
{
   // Get the radius
    double radius=20.5;  // Radius of the circle
    double area;         // Computed area

    // Compute the circle’s area
    area = PI * radius * radius;

   // Print the radius and area of the circle
    printf(“Radius=%6.2f\nArea=%8.2f\n”,radius,area);

   return 0;
}
```

Program screen output

The results:

```
Radius= 20.50
Area= 1320.26
```

scanf to input from keyboard

Question: What's the problem with this program?

Answer: Anytime we want to use a different radius, we have to modify the program and recompile it. Not good!

We solve this problem by using one of C's input functions, ***scanf***.

The general format for the scanf statement is:

`scanf(`<*control string*>, <*variable list*>**);**

scanf and printf

```
#include <stdio.h>
#define PI 3.1415963

int main(void)
{
  double radius;
  double area;
  // Get the radius
  printf(“Enter radius: “);  // the prompt
  scanf(“%lf”,&radius);      // reads radius

  // Compute the area of the circle
  area = PI * radius * radius;

  // Print the radius and area of the circle
  printf(“Radius=%6.2lf\nArea=%8.2f\n”, radius, area);

  return 0;
}
```

Program screen output

With the printf added, when the program is run, the following will appear to the user:

```
Enter radius: _
```

The user can then input the radius:

```
Enter radius: 5.07
```

And get the results:

```
Radius = 5.07
Area =  80.75
```

scanf() and printf()

Control Strings in scanf

Unlike the **`printf`** statement, there are only type specifiers in the control string. No plain text or escape sequences are used (they are ignored if present).

It is very important that the type specifier and the type of variable being read from the keyboard match, or errors can occur.

Variable Type	Specifier	
`int`	**`%d`**	
`float`	**`%f`**	
`double`	**`%lf`**	**`<use %f with printf()>`**
`char`	**`%c`**	

scanf Examples

Statement: `scanf(“%d %d”, &myIntOne, &myIntTwo);`

User Input: `20  20`

User Input: `-3 1190027`

Statement: `scanf(“%d %f”, &myInt, &myFloat);`

User Input: `51 4.08`

User Input: `3 2.0`

The argument in **`scanf()`** function uses the address of the variable.

`scanf(“%f %d %lf”, &myFloat, &myInt, &myDouble);`

User Input: `67.8    93        0.00481`

(*notice that “white space” is ignored*)

printf and variables

printf(*format_string, argument_list*);

Format_string – a string literal in “ “
Format_specification – conversion specification

```
    int     %[field width]d              %5d
    float   %[field width][.precision]f  %9.2f
    double  %[field width][.precision]f  %9.0f
```

```
/* Lesson for 3-5 scanf example */
#include <stdio.h>
int main()
{
  float income;
  double expense;
  int month, hour, minute;

  printf("What month is it?\n");
  scanf("%d", &month);
  printf("You have entered month=%5d\n",month);

  printf("Please enter your income and expenses\n");
  scanf("%f %lf",&income,&expense);
  printf("Entered income = %8.2f, expenses = %8.2f\n", income,
                                                        expense);

  printf("Please enter the time, e.g.,12:45\n");
  scanf("%d : %d",&hour,&minute);
  printf("Entered Time = %2d:%2d\n",hour,minute);
  return 0;
}
```

Format specification

```
   printf("In month %d, my income was $%f.\n", month,
1                                                income);
   In month 12, my income was $1200.000000.
   printf("In month %5d, my income was $%9.3f.\n", month,
2                                                income);
   In month    12, my income was $ 1200.000.
   printf("In month %2d, my income was $%5.2f.\n", month,
3                                                income);
   In month 12, my income was $1200.00.
   printf("In month %f, my income was $%d.\n", month,
4                                                income);
   In month 0.000000, my income was $1083359232.
```

printf with variables and constant

```
float applePrice;
float melonPrice;
applePrice = .50;
melonPrice = 2.0;
printf("*****  ON SALE  *****\n");
printf("Fruit type      Price\n");
printf("Apple           $%5.2f\n", applePrice);
printf("Pear            $%5.2f\n", .35);
printf("Melon           $%5.2f\n", melonPrice);
```

```
*****  ON SALE  *****
Fruit type      Price
Apple           $ 0.50
Pear            $ 0.35
Melon           $ 2.00
```

printf with + and - flags

```
applePrice = .50;
melonPrice = 2.0;
printf('*****  ON SALE  *****\n');
printf('Fruit type      Price\n');
printf('Apple           $%-9.2f\n', applePrice);
printf('Pear            $%+9.2f\n', .35);
printf('Melon           $%09.2f\n', melonPrice);
```

```
*****  ON SALE  *****
Fruit type      Price
Apple           $0.50
Pear            $    +0.35
Melon           $000002.00
```

Very small values and precision

%e exponential type format

```
float ratio = .00000000123;
printf("ratio = %5.3e  ratio = %5.3f\n", ratio, ratio);

ratio = 1.230e-09  ratio = 0.000
```

Errors and Debugging

Debugging

Syntax errors

- Violating the “grammar” rules of C
- Diagnosed by the C compiler

Run-time errors

- Semantic errors caused by the violation of rules during running the program

Logic errors

- Most difficult to recognized
- Up to programmer to find

How to reduce number of errors

To write a program, follow the five-step process for problem solving:

1. State the problem clearly.
2. Describe the input and output information.
3. Work the problem by hand for a simple set of data.
4. Develop a solution that is general in nature.
5. Test the solution with a variety of data sets.

Think and plan before you code!

How to reduce number of errors

Develop good habits:

1. Write your programs neatly.
2. Add blank lines at natural locations.
3. Line up your opening and closing braces.
4. Indent nested blocks.
5. Add comments properly.

Organize and structure your code!

How to debug a program

- Don’t get frustrated.
- Check the obvious first.
- Be systematic about making changes.
- Look before and after the statement number in the error message.
- Desk check your code…Read your code. Does it do what you intended it to do?

Don’t assume it’s correct…be objective.
Everyone makes mistakes.