CSC215 Lecture

Variables, Types and Expressions

Outline

- Variables
- Datatypes
  - Basic data types
  - Derived data types
  - User-defined data types
- Expressions
  - Operators: arithmetic, relational, logical, assignment, inc-/dec- rement, bitwise
  - Evaluation
- Formatted input/output

Variables

- Named values
  - Naming rules:
    - Made up of letters, digits and the underscore character ‘_’
    - Must not begin with a digit
    - Must not be a special keyword

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

- Variable declaration:
  - Must declare variables before use
  - Variable declaration: `int n; float phi;`
    - int - integer data type
    - float - floating-point data type
  - Many other types
- Variable initialization:
  - Uninitialized variable assumes a default value
  - Variables initialized via assignment operator: `n = 3;`
  - Can also be initialized at declaration: `float phi = 1.6180339887;`
  - Can declare/initialize multiple variables at once: `int a, b, c = 0, d = 4;`

Basic Data Types

- ❑ Data type determines the variable's domain and applicable operations
- ❑ Four types: `char` `int` `float` `double`
- ❑ Modifiers: `signed` `unsigned` `short` `long`
- ❑ Combinations:

	Type	Bits	Range
Char	[signed] char	8	-128 .. 127
	unsigned char	8	0 .. 259
int	[signed] int	16 (at least)	-2^{15} .. $2^{15}-1$
	unsigned int	16 (at least)	0 .. $2^{16}-1$
	[signed] short [int]	16	-2^{15} .. $2^{15}-1$
	unsigned short [int]	16	0 .. $2^{16}-1$
	[signed] long [int]	32 (at least)	-2^{31} .. $2^{31}-1$
	unsigned long [int]	32 (at least)	0 .. $2^{32}-1$
float	float	32	1.2E-38 .. 3.4E+38 (6 dig-prec)
double	double	64	2.3E-308 .. 1.7E+308 (15 dig-prec)
	long double	80 (at least)	3.4E-4932 .. 1.1E+4932 (19 dig-prec)

- ❑ What about boolean? strings?

Boolean?

- ❑ No special boolean type
- ❑ Evaluating boolean and logical expressions:
  - results in integer 1 if the logic is true
  - results in 0 if the logic is false
- ❑ Interpretation of integer as boolean:
  - 0 is perceived as false
  - any non-zero value is perceived as true

Strings ?

- Strings stored as character array
- Null-terminated (last character in array is ’\0’: null character)

```
char course[7] = {'C', 'S', 'C', '2', '1', '5', '\0'};
char course[] = {'C', 'S', 'C', '2', '1', '5', '\0'};
```

- Not written explicitly in string literals

```
char course[7] = "CSC215";
char course[] = "CSC215";
```

- Special characters specified using \ (escape character):
  - **`\\`** – backslash
  - **`\'`** – apostrophe
  - **`\"`** – quotation mark
  - **`\b`**, **`\t`**, **`\r`**, **`\n`** – backspace, tab, carriage return, linefeed
  - **`\o`**`oo`, **`\x`**`hh` – octal and hexadecimal ASCII character codes, e.g. \x41 – `'A'`, \060 – `'0'`

Initialization of Variables

- Local variables:
    - declared inside a function
    - are not initialized by default
- Global variables:
    - declared outside of functions
        - On top of the program
    - are initialized by default:

Type	**Default value**
int	0
char	'\0'
float	0
double	0
pointer	null
Derived types	apply recursively

Constants

- ❑ The previous examples can be rewritten as:

```
int main(void) /* entry point */ {
  const char msg [ ] = "Hello World!";
  /* write message to console */
  puts(msg);
}
```

- ❑ **const** keyword: qualifies variable as constant
- ❑ **char**: data type representing a single character; written in quotes: ’a’, ’3’, ’n’
- ❑ const char msg[]: a constant array of characters

Expressions

- Expression:
  - a sequence of characters and symbols that can be evaluated to a single data item.
  - consists of: literals, variables, subexpressions, interconnected by one or more *operators*
- Operator:
  - Can be unary, binary, and ternary
  - Categories:

Arithmetic:	`+x, -x, x+y , x-y , x*y , x/y , x%y`
Relational	`x==y, x!=y, x<y, x<=y, x>y, x>=y`
Logical	`x&&y, x||y, !x`
Bitwise	`x&y, x|y, x^y, x<<y, x>>y, ~x`
Assignment	`x=y, x+=y, x-=y, x*=y, x/=y, x%=y` `x<<=y, x>>=y, x&=y, x|=y, x^=y`
inc-/dec- rement	`++x, x++, --x, x--`
Conditional	`x?y:z`
More:	`*x, &x, (type)x, sizeof(x), sizeof(<type>)`

Arithmetic Operators

- ❑ 2 Unary operators: `+` `-`
- ❑ 5 Binary operators: `+` `-` `*` `/` `%`
  - If both operands are of type int, the result is of type int
- ❑ Example:

```c
int main() {
  int a = 9, b = 4, c;
  c = a+b;
  printf("a+b = %d \n",c);
  c = a-b;
  printf("a-b = %d \n",c);
  c = a*b;
  printf("a*b = %d \n",c);
  c=a/b;
  printf("a/b = %d \n",c);
  c=a%b;
  printf("Remainder when a divided by b = %d \n",c);
  return 0;
}
```

Relational Operators

- ❑ 6 Binary operators: == != > >= < <=
- ❑ Checks the relationship between two operands:
  - if the relation is true, it yieldss 1
  - if the relation is false, it yields value 0
- ❑ Example:

```
int main(){
  int a = 5, b = 5, c = 10;
  printf("%d == %d = %d \n", a, b, a == b); /* true */
  printf("%d == %d = %d \n", a, c, a == c); /* false */
  printf("%d > %d = %d \n", a, b, a > b); /*false */
  printf("%d > %d = %d \n", a, c, a > c); /*false */
  printf("%d < %d = %d \n", a, b, a < b); /*false */
  printf("%d < %d = %d \n", a, c, a < c); /*true */
  printf("%d != %d = %d \n", a, b, a != b); /*false */
  printf("%d != %d = %d \n", a, c, a != c); /*true */
  printf("%d >= %d = %d \n", a, b, a >= b); /*true */
  printf("%d >= %d = %d \n", a, c, a >= c); /*false */
  printf("%d <= %d = %d \n", a, b, a <= b); /*true */
  printf("%d <= %d = %d \n", a, c, a <= c); /*true */
  return 0;
}
```

Logical Operators

- 1 Unary operator: `!` and 2 binary operators: `&&` `||`
- Example:

```
int main(){
  int a = 5, b = 5, c = 10, result;
  result = (a = b) && (c > b);
  printf("(a = b) && (c > b) equals to %d \n", result);
  result = (a = b) && (c < b);
  printf("(a = b) && (c < b) equals to %d \n", result);
  result = (a = b) || (c < b);
  printf("(a = b) || (c < b) equals to %d \n", result);
  result = (a != b) || (c < b);
  printf("(a != b) || (c < b) equals to %d \n", result);
  result = !(a != b);
  printf("!(a == b) equals to %d \n", result);
  result = !(a == b);
  printf("!(a == b) equals to %d \n", result);
  return 0;
}
```

Bitwise Operators

- 1 Unary operator ~ and 5 binary operators & | ^ << >>
- Examples:

```
int main(){
  int a = 12;
  int b = 25;
  printf("complement=%d\n",~35);
  printf("complement=%d\n",~-12);
  printf("Output = %d", a&b);
  printf("Output = %d", a|b);
  printf("Output = %d", a^b);

  int num=212;
  printf("Right shift by 3: %d\n", num>>3);
  printf("Left shift by 5: %d\n", num<<5);
  return 0;
}
```

```
 35 00000000 00100011  ~
-36 11111111 11011100
```

```
-12 11111111 11110100  ~
 11 00000000 00001011
```

```
12 00000000 00001100
25 00000000 00011001
   ----------------- &
 8 00000000 00001000
```

```
12 00000000 00001100
25 00000000 00011001
   ----------------- |
29 00000000 00011101
```

```
12 00000000 00001100
25 00000000 00011001
   ----------------- ^
21 00000000 00010101
```

```
212 00000000 11010100
 26 00000000 00011010 »
```

```
 212 00000000 11010100
6784 00011010 10000000 «
```

Assignment Operators

- 11 Binary operators: = += -= *= /= %= &= |= ^= <<= >>=
- Example:

```
int main(){
  int a = 5, c;
  c = a;
  printf("c = %d \n", c);
  c += a; /* c = c+a */
  printf("c = %d \n", c);
  c -= a; /* c = c-a */
  printf("c = %d \n", c);
  c *= a; /* c = c*a */
  printf("c = %d \n", c);
  c /= a; /* c = c/a */
  printf("c = %d \n", c);
  c %= a; /* c = c%a */
  printf("c = %d \n", c);
  return 0;
}
```

Increment/Decrement operators

- 2 Binary operators: ++ --
- Example:

```
int main(){
  int a = 10, b = 100;
  float c = 10.5, d = 100.5;
  printf("++a = %d \n", ++a); /* 11 */
  printf("b++ = %d \n", b++); /* 100 */
  printf("c-- = %f \n", c--); /* 10,500000 */
  printf("--d = %f \n", --d); /* 99.500000 */
  return 0;
}
```

Ternary Conditional Operator

- Syntax: `<conditionalExpression> ? <expression1> : <expression2>`
- The conditional operator works as follows:
  - <conditionalExpression> is evaluated first to non-zero (1) or false (0).
  - if <conditionalExpression> is true, <expression1> is evaluated
  - if <conditionalExpression> is false, <expression2> is evaluated.
- Example:

```
int main(){
  char February;
  int days;
  printf("If this year is leap year, enter 1. If not enter any integer: ");
  scanf("%c",&February);
  /* If test condition (February == 'l') is true, days equal to 29. */
  /* If test condition (February =='l') is false, days equal to 28. */
  days = (February == '1') ? 29 : 28;
  printf("Number of days in February = %d",days);
  return 0;
}
```

More Operators

- ❑ sizeof: unary operator returns data (constant, variable, array, structure...)
- ❑ Example:

```
int main(){
  int a, e[10];
  float b;
  double c;
  char d;
  printf("Size of int=%lu bytes\n",sizeof(a));
  printf("Size of float=%lu bytes\n",sizeof(b));
  printf("Size of double=%lu bytes\n",sizeof(c));
  printf("Size of char=%lu byte\n",sizeof(d));
  printf("Size of integer type array having 10 elements = %lu bytes\n", sizeof(e));
  return 0;
}
```

Evaluating Expressions

- **Expression:** A sequence of characters and symbols that can be evaluated to a single data item.
- **Expression evaluation:**
  - Order of operations:
    Use parenthesis to override order of evaluation
  - Example: Assume `x = 2.0 and y = 6.0.`
    Evaluate the statement:
    `float z = x+3*x/(y-4);`
    1. Evaluate expression in parentheses
       → `float z = x+3*x/2.0;`
    2. Evaluate multiplies and divides, from left-to-right
       → `float z = x+6.0/2.0;` → `float z = x+3.0;`
    3. Evaluate addition float:
       → `float z = 5.0;`
    4. Perform initialization with assignment Now, `z = 5.0.`
  - How do I insert parentheses to get z = 4.0?

Operator	Associativity
<function>(), [ ], ->, .	left to right
!, ~, ++, --, +, -, *, (<type>), sizeof	right to left
*, /, %	left to right
+, - (unary)	left to right
<<, >>	left to right
<, <=, >, >=	left to right
==, !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
? :	left to right
= += -= *= /= %= &= ^= |= <<= >>=	right to left
,	lrft to right

Formatted Input and Output

- Function `printf`

  `printf(control_string, arg1, arg2, …);`

  - control_string is the control string or conversion specification consists of % followed by a specifier

    `%[flags][width][.precision][length]specifier`
  - Specifiers (place holders):
    - %d - int (same as %i)
    - %ld - long int (same as %li)
    - %f - decimal floating point
    - %lf - double or long double
    - %e - scientific notation (similar to %E)
    - %c - char
    - %s - string
    - %o - signed octal
    - %x - hexadecimal (similar to %X)
    - %p - pointer
    - %%- %
  - Optional width, length precision and flags

Flags	: -	+	#	0
Width	: *	number		
Length	: h	l	L	
Precision	: .*	.number		

Formatted Input and Output

- Numeric:

```
%[[<FLAG>][<LENGTH>][.<PRECISION>]]<SPECIFIER>
```

-	Left align
+	Prefix sign to the number
#	Prefix 0 to octal, 0x/0X to hexadecimal Force decimal point with e E f G g
0	Pad with leading zeros
□	Replace positive sign with space

`<Number>`	Decimal digits
`*`	Passing it as an arg
Default:	6

`<Number>`	Minimum length
`*`	Passing it as an arg
Default:	All

`%d`	int (same as %i)
`%ld`	long int (same as %li)
`%f`	decimal floating point
`%lf`	double or long double
`%e`	scientific notation (similar to %E)
`%g`	shorter of f and e
`%c`	char
`%o`	signed octal
`%x`	hexadecimal (similar to %X)

- String:

```
%[[<FLAG>][<LENGTH>][.][<WIDTH>]]<SPECIFIER>
```

-	Left align

`%s`	string

`<Number>`	Minimum length
`*`	Passing it as an arg
Default:	All

`<Number>`	Max `number of` characters to print
`*`	Passing it as an arg
Default:	0 with ., all if . is omitted

Formatted Input and Output

- Function scanf

  `scanf(control_string, arg1, arg2, …);`

  - Control_string governs the conversion, formatting, and printing of the arguments
  - Each of the arguments must be a pointer to the variable in which the result is stored
  - So: `scanf("%d", &var);` is a correct one, while `scanf("%d", var);` is not correct
  - Place holders:
    - %d - int (same as %i)
    - %ld - long int (same as %li)
    - %f - float
    - %lf - double
    - %c - char
    - %s - string
    - %x - hexadecimal

Macros

- Preprocessor macros begin with # character
  - `#define msg "Hello World"`
    defines msg as “Hello World” throughout source file
- #define can take arguments and be treated like a function
  #define add3(x,y,z) ((x)+(y)+(z))
  - parentheses ensure order of operations
  - compiler performs inline replacement; not suitable for recursion
- #if, #ifdef, #ifndef, #else, #elif , #endif conditional preprocessor macros
  - can control which lines are compiled
  - evaluated before code itself is compiled, so conditions must be preprocessor defines or literals
  - the gcc option -Dname=value sets a preprocessor define that can be used
  - Used in header files to ensure declarations happen only once
- Conditional preprocessor macros:
  - #pragma preprocessor directive
  - #error, #warning trigger a custom compiler error/warning
  - #undef msg remove the definition of msg at compile time