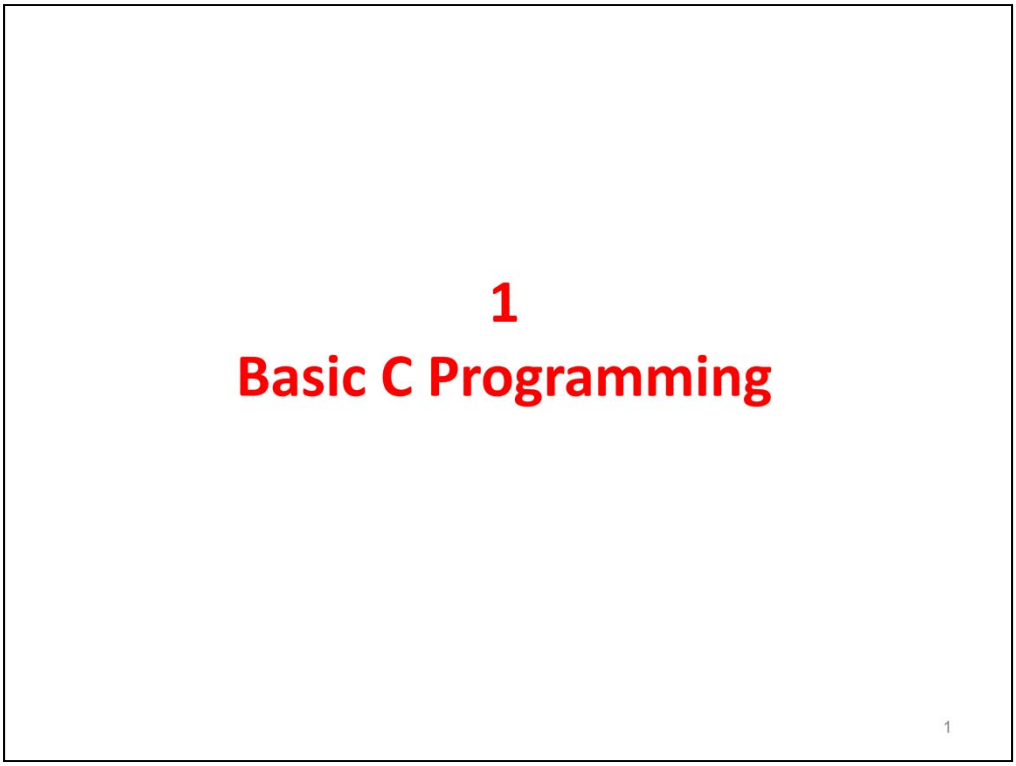
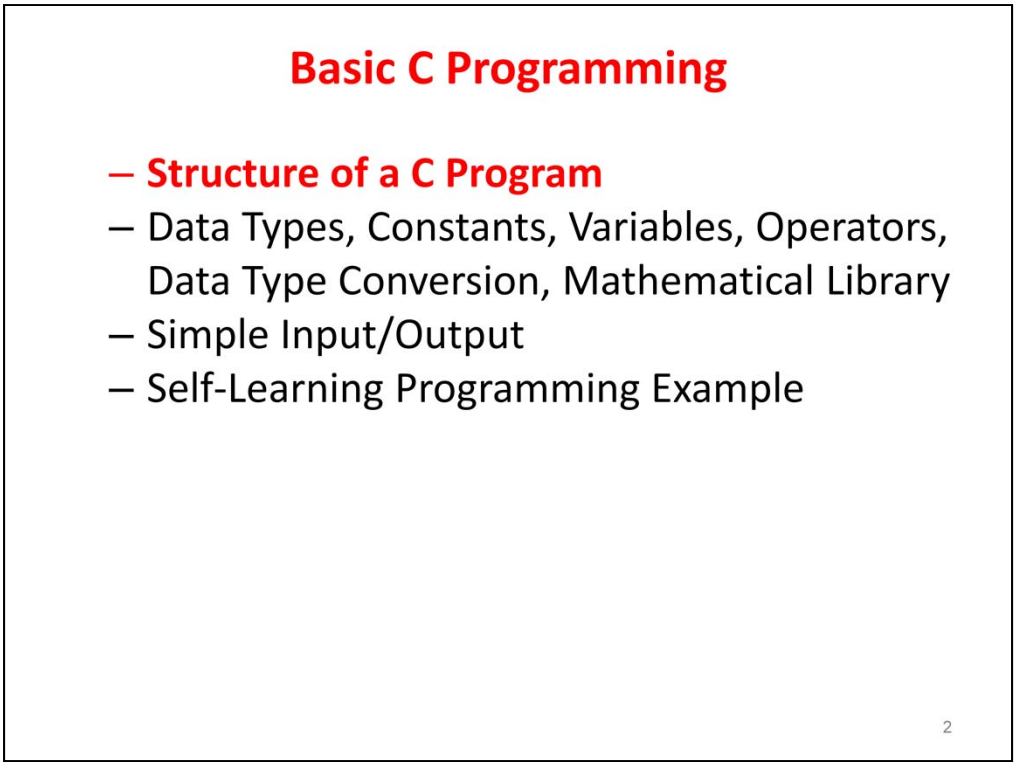


Basic C Programming

1. In this lecture, we discuss basic C programming concepts.



Basic C Programming

1. For basic C programming concepts, we will discuss the basic structure of a C program and the various components for a C program, and a sample C program is used to illustrate these basic components.
2. Then, the program development cycle for creating, compiling and executing a C program will be described.
3. After that, data types, variables, operators and simple input/output in C will be discussed.
4. Here, we start by discussing the basic structure of a C program.

Why Learning C Programming Language?

- **Advantages** on using C
  - Powerful, flexible, efficient, portable
  - Enable the creation of well-structured programs
- Any **disadvantages**?
  - Free style and **not strongly-typed**
  - The use of ***pointers***, which may confuse many students
- **Why** doing data structures in C
  - Efficient
  - Provide ***pointers*** for building data structures which are powerful
  - Bridge to C++ (OO Programming)

Dennis Ritchie

Why Learning C Programming Language?

1. The C programming language has a number of advantages over other conventional programming languages such as BASIC, PASCAL and FORTRAN. It is powerful, flexible, efficient, portable, structured and modular. It enables the creation of well-structured programs.
2. However, C also has a few disadvantages. The free style of expression in C can make it difficult to read and understand. In addition, as C is not a strongly-typed language such as PASCAL and JAVA, it is the programmer's responsibility for ensuring the correctness of the program. Pointer in C is a very useful feature, however, it can also cause programming errors that are difficult to debug and trace if it is not used properly. Nevertheless, these drawbacks can be overcome if good programming style is adopted.
3. In this course, we do data structures in C because C provides pointers for building data structures efficiently. It also bridges well to C++ which is an object-oriented programming language.

C Programming Language

C programming language was created by Dennis Ritchie at AT&T Bell Laboratories in 1972. The language was originally designed as a system software development language to be used with the Unix operating system. Over the past few decades, C has become one of the most popular programming languages for developing a wide range of applications including word processors, graphics applications, database management systems, communication

systems, expert systems and other engineering applications.

Advantages

The C language has a number of advantages over other conventional programming languages such as BASIC, PASCAL and FORTRAN. It is powerful, flexible, efficient, portable, structured and modular.

- **Powerful and Flexible -** C is a powerful and flexible language. C has been used to implement the Unix operating system. Compilers for many other languages such as FORTRAN and BASIC are written in C. Moreover, C also provides the freedom and control in developing programs. It is easy to perform operations on manipulating bits, bytes and addresses using operators and pointers. C also supports programming over hardware and peripherals.
- **Efficient -** C is an efficient language. As C is a relatively small language, efficient executable programs can be generated from C source codes. C programs are compact in storage requirements and can be run very quickly.
- **Portable -** A C program written in one type of computer system can be run on another type of computer system with little or no modification. As C compilers are available for a wide range of systems from personal computers to mainframes, it is easy to port C programs from one hardware platform to another.
- **Structured and Modular -** C has features to enable the creation of well-structured programs that are easy to read. C programs can also be written in functions, which can be compiled and tested separately. Large and complex systems can be written and tested as modules by different people, these modules are then combined into a single program. The functions and modules can also be reused in other applications.

Disadvantages

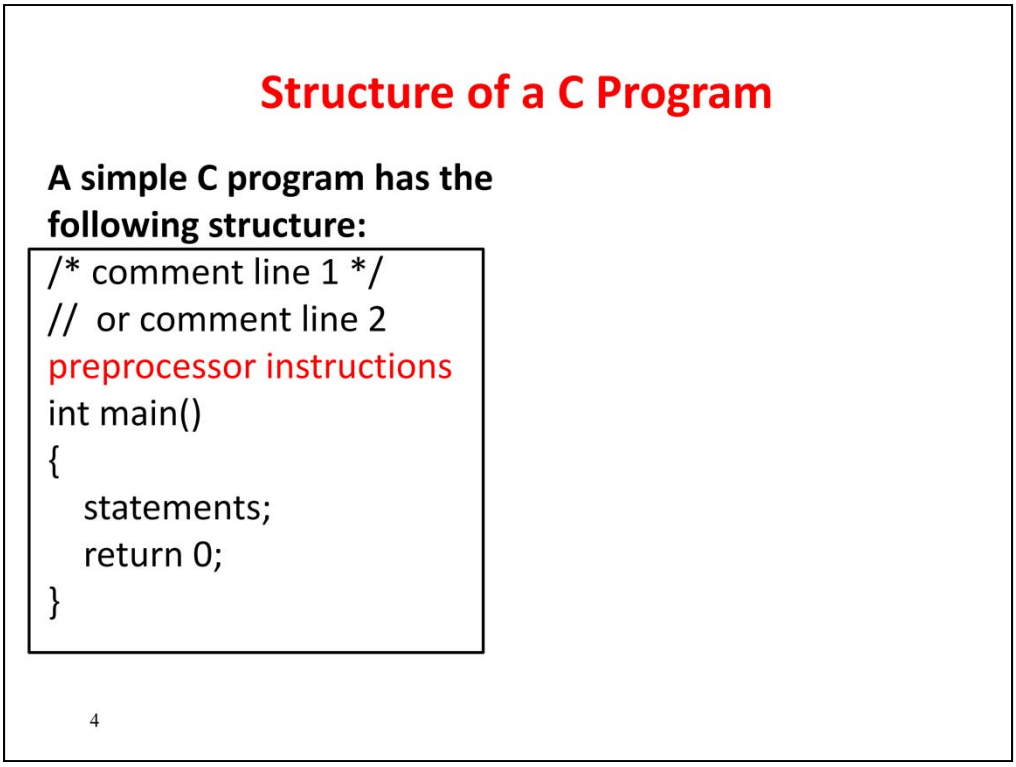
However, C also has a few disadvantages.

•The free style of expression in C can also make it difficult to read and understand.

•In addition, as C is not a strongly-typed language such as PASCAL and JAVA, it is the programmer's responsibility for ensuring the correctness of the program.

•Pointer in C is a very useful feature, however, it can also cause programming errors that are difficult to debug and trace if it is not used properly.

Nevertheless, these drawbacks can be overcome if good programming style is adopted.



Structure of a C Program

1. Here, we show a typical program structure which consists of comments, preprocessor instructions, main() function header, open brace, program statements, return statement and close brace.

Generally, a typical C program has the following components: preprocessor instructions, global declarations, function prototypes, program statements, functions and comments.

Preprocessor Instructions

Preprocessor instructions refer to the instructions to the preprocessor of the compiler. All preprocessor instructions start with the symbol **#**. The C preprocessor supports string replacement, macro expansion, file inclusion and conditional compilation. For example, the **#include** <*filename*> instruction informs the preprocessor to include the file <*filename*> into the text of the source code file before compilation. The files that are to be included usually have the extension **.h** such as *stdio.h*. They are placed at the beginning of the source code file. The *stdio.h* file is required for programs that need to perform input/output operations.

Global Declaration

Global declaration statements declare global variables that are accessible anywhere within

a C program.

Function Prototypes

Function prototypes provide useful information regarding the functions used in the program to the C compiler. They inform the compiler about the name and arguments of the functions contained in the program. They need to appear before the functions are used.

Functions

A C program consists of one or more functions. The **main()** function is required in every C program. A function has a header and a body. The header is **main()**. The body of the **main()** function is enclosed by the braces **{}**. It consists of statements or instructions that the computer can execute. Program execution starts from the beginning of the body. The statement **return 0** can be the last statement of the **main()** function depending on whether the function returns any values. The statement

void main()

does not require a **return** statement in the **main()** body. The statement

int main()

requires the **main()** body to have a **return** statement to return an integer value. The format of the statement

main()

is also accepted by ANSI C, which is equivalent to **int main()**.

Apart from the **main()** function, a program can have many other functions. The function header defines the name of the function, and the inputs expected by the function. The function body contains statements that are written to perform a specific task. The body will be executed when the function is called. The **main()** function or functions can call other functions, which may in turn call other functions.

A function may or may not have arguments, and it may or may not have a return value. The function

int add(int x, int y) { … }

has two arguments and requires a return value, while the function

void add() { … }

has no argument and does not require a return value.

In addition, C provides library functions that perform most of the common tasks such as input operation from the keyboard and output operation to the screen. The **printf()** and **scanf()** statements are the most commonly used library functions for input/output operations. The **printf()** statement displays information on the screen, while the **scanf()** statement reads data from the keyboard and assigns the data to a variable.

Program Statements

A statement is a command to the computer. There are two types of statements: *simple* or *compound* statements. A simple statement is a statement terminated by a semicolon (**;**). There are four types of simple statements: declaration statements, assignment statements, function call statements and control statements. Some examples of C statements are given as follows:

```
int x,y,z; /* declaration statement */
x = 30;    /* assignment statement */
printf("Welcome to Programming with C");    /* function call statement */
for (y=0; y<z; y++) { … }      /* control statement */
```

A compound statement is a sequence of one or more statements enclosed in braces:

```
{
    statement_1;
    statement_2;
    ...
    statement_n;
}
```

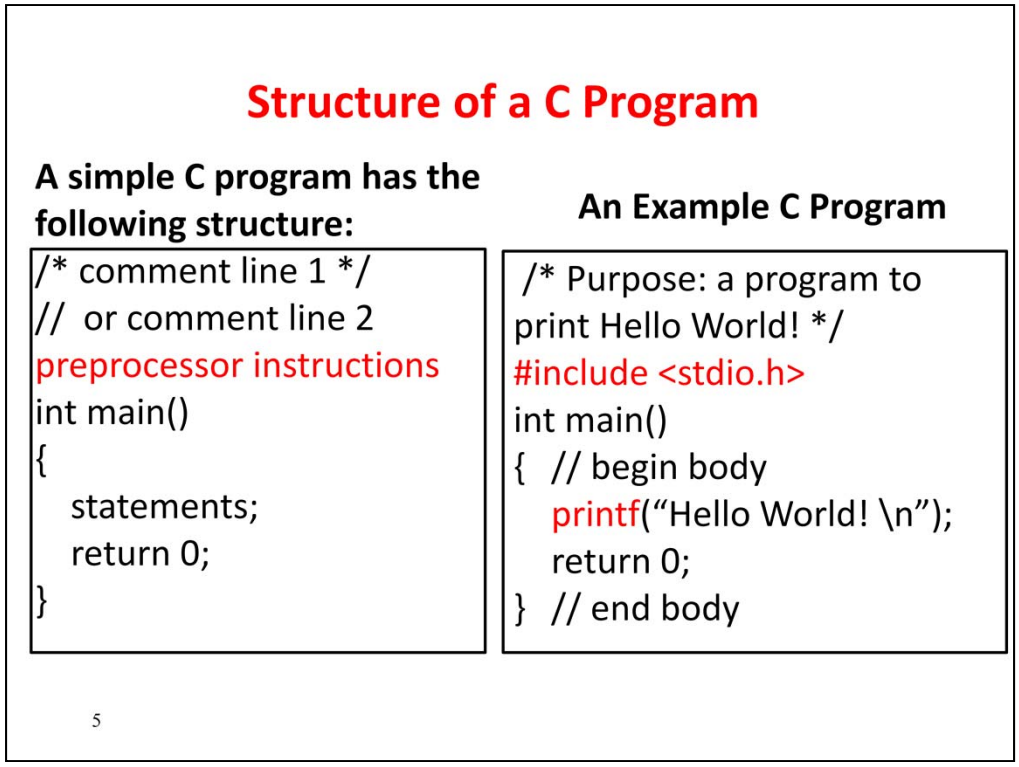
Each of the **statement_i** (where **i** = 1,..,n) can be a simple or a compound statement.

Comments

Comments may be added to a program to explain the purpose of the program, or how a portion of the program works. A comment is a piece of English text. It is enclosed by **/*** and ***/** for a multiple-line comment or **//** for a single line of comment. Comments may appear anywhere in the program. It makes programs more readable. The compiler simply skips the comments when translating the program. There are different ways to write comments:

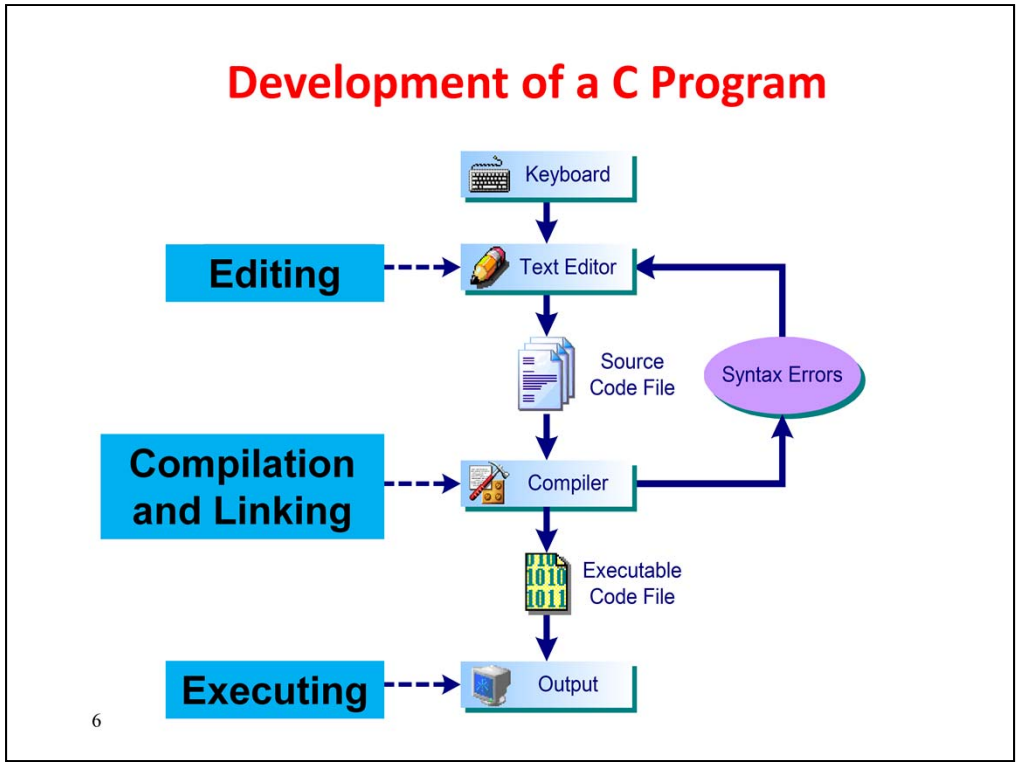
```
/* This is a comment */
int x,y,z; // This is another comment
/* This is the first comment
This is the second comment
This is the third comment */
```

It is important to have comments in the programs. Comments document what the program does, how the variables are defined, and how the logic of the program works. This is extremely helpful when future modification of the program is required. Therefore, it is necessary to develop the habit of using comments to document the programs including the use of comments for programming structures and operations.



An Example C Program

1. An example C program is shown:
   1) The first two lines are the comments which state the purpose of the program (using **/*** and ***/** to enclose the comment).
   2) The **#include** preprocessor directive instructs the C compiler to add the contents of the file stdio.h into the program during compilation. The **stdio.h** file is part of the standard library. It supports input and output operations that the program requires.
   3) In the **main()** function, it requires to return an integer value, so the keyword **int** is used to inform the C compiler about that.
   4) The braces **{ }** are used to enclose the **main()** function body.
   5) The line (**// begin body**) is a comment (using **//** to state the comment).
   6) The **printf()** function is the library output function call to print a character string on the screen.
   7) The statement **return 0** returns the control back to the system.
   8) Finally, the last line (**// end body**) is a comment (using **//** to state the comment).



Development of a C Program

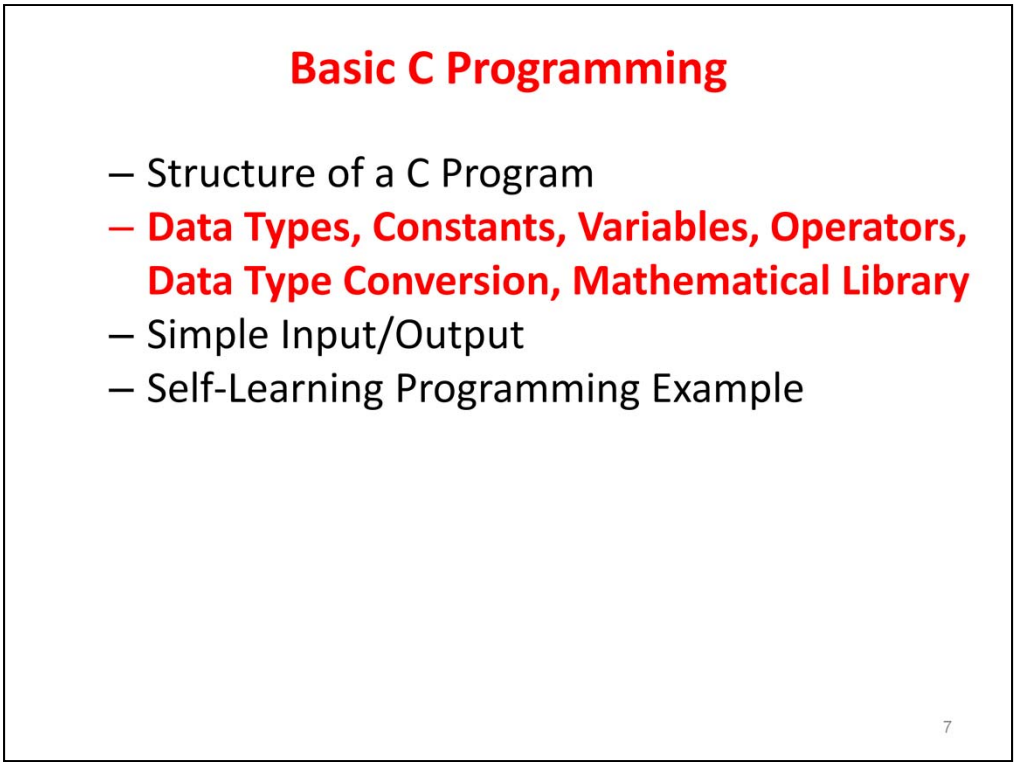
1. To develop a C program, a text editor is first used to create a file to contain the C source code. Most compilers come with editors that can be used to enter and edit source code.
2. Then, the source code needs to be processed by a compiler to generate an object file. If syntax errors occur during compilation, we will need to rectify the errors, and compile the source code again until no further errors are occurred.
3. The linker is then used to link all the object files to create an executable file.
4. Finally, the executable file can be run and tested.

Compilation and Linking

After a C program is created, it will then go through compilation and linking. In compilation and linking, it involves three processes: preprocessor, compiler and linker. The preprocessor processes the preprocessing statements in the source code file. For example, the preprocessing statement **$include <filename>** will inform the preprocessor to include the stated filename into the text of the source code file before compilation. The compiler checks for syntax errors in the source files. All the errors need to be corrected. The linker combines the object code generated from the compiler with other object files (from the included library files) to produce an executable program.

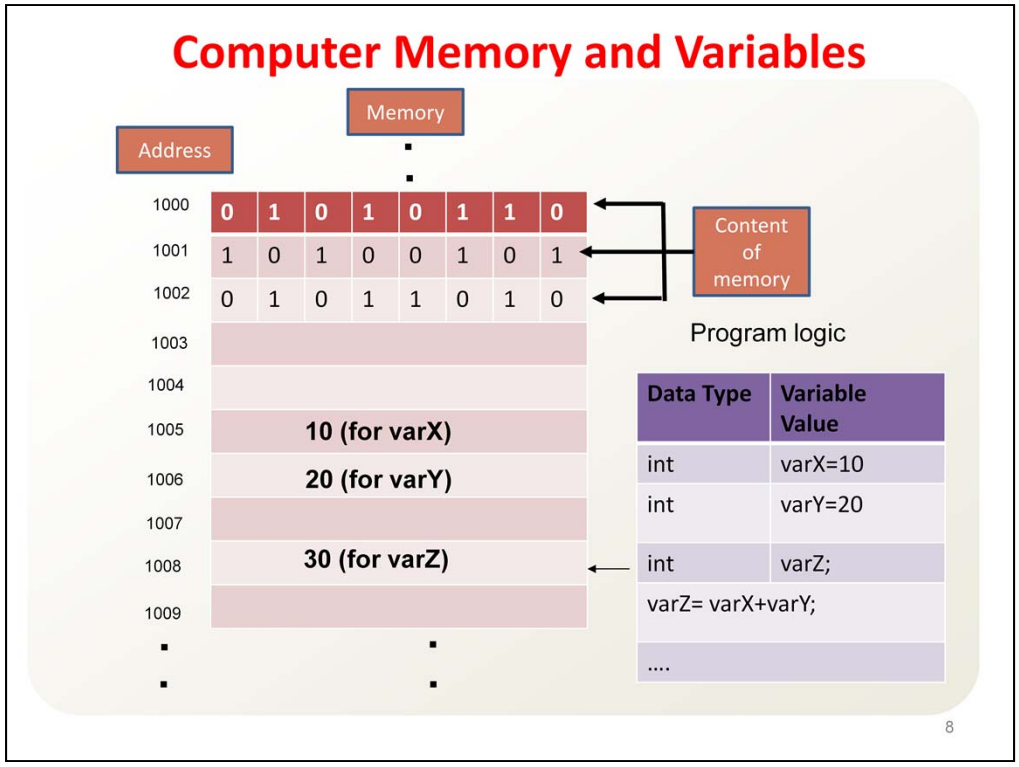
Three processes, namely preprocessor, compiler and linker are involved in compilation and linking.

- **Preprocessor -** The preprocessor processes the preprocessing statements in the source code file. Preprocessing statements can support string replacement, macro expansion, file inclusion and conditional compilation. An example of preprocessing statement is **#include** *<filename>,* which informs the preprocessor to include the file *<filename>* into the text of the source code file before compilation.
- **Compiler -** The compiler checks for syntax errors in the source files. Error messages will be given if errors are found. The locations of the errors are also given in the error messages. All the errors need to be corrected. If no more errors are detected, the compiler then translates the source code into machine language instructions, which are called object code. A file is created to store the object code. The object file has the same name as the source file, but with a different extension (**.obj** or **.o**). The extension is used to indicate that the file contains object code.
- **Linker -** The linker combines the object code with other object files to produce an executable program. The linker allows us to develop programs that can be placed in several files. Functions that are frequently used in other programs are placed in separate files and compiled separately. This can save time to re-compile the functions every time they are used. In addition, only files containing updated source code need to be re-compiled before linking. The linker also provides a convenient way to support the use of standard libraries. Most C programs use standard library functions to perform certain specific tasks such as input/output operations and mathematical functions. During linking, the linker combines the object code with the object code from the standard library provided by the compiler to generate the executable program.



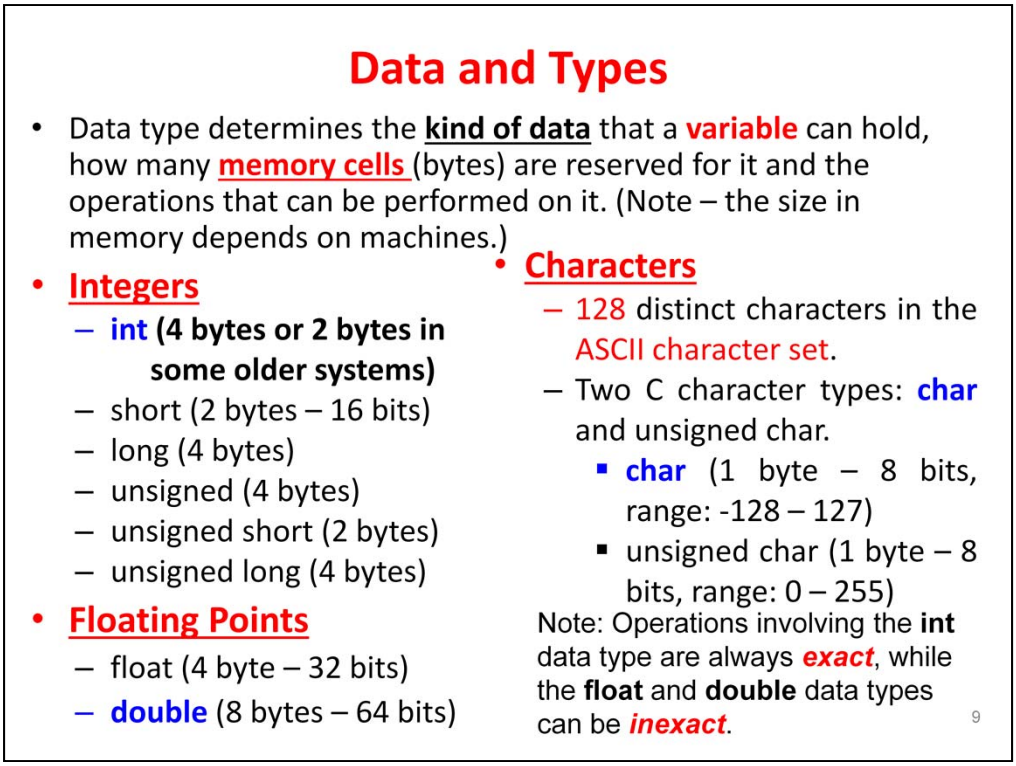
Basic C Programming

1. Here, we discuss data types, constants, variables, operators, data type conversion, and mathematical library.



Computer Memory and Variables

1. A program needs to work with data which are stored in the main memory.
2. A data object in a C program can be a **variable** or a **constant**.
3. Variables are used to store data whose values change during the execution of a program, while constants can be used to store data whose values remain fixed during the execution of a program.
4. Both constants and variables must be of a certain type. The basic data types in C are integer, floating point and character.



Data and Types

1. A data type describes the size (in terms of number of bytes in memory) of an object and how it may be used. Each type has its own computational properties and memory requirements. Therefore, you should select the data type for variables according to your data requirement.
2. There are three basic data types in C for **characters**, **integers** and **floating point numbers**.
3. For characters, we can just use the type **char** which will take up 1 byte of memory.
4. For integers, we can just use the type **int** which will typically take up 4 bytes of memory in current computers.
5. For floating point numbers, apart from the data type **float** which takes up 4 bytes of memory, we can also use the type **double** which will typically take up 8 bytes of memory for storing larger floating point numbers.
6. Note that operations involving the **int** data type are always ***exact***, while the **float** and **double** data types can be ***inexact***.

Data Types

A data type describes the size (in terms of number of bytes in memory) of an object and how it may be used. The three basic data types in C are **char**, **int** and

float. Each type has its own computational properties and memory requirements. Moreover, the same data type may take different sizes on different types of machines. The type **int** generally takes four bytes on most personal computers, but it will take only 2 bytes for some older machines. In addition to the basic data types, C also provides several variations that change the meaning of the basic data types when applied to them. For example, the types **short** and **long** can be applied to integers.

Integer Type

Any data object that is an integer is in this category. The type **int** is a signed integer which may be positive, zero or negative. Operations on integers are exact and faster than floating point operations. In C, there are six ways to store an integer object in a computer for C: short, int, long, unsigned, unsigned short, unsigned long. In principle, we may treat the type **long** as 32 bits, **short** as 16 bits, and **int** as either 16 bits or 32 bits, depending on the machine's natural word size. The range of values allowed for each type depends on the number of bits used. For example, The range of values for 16 bits is (−32768, 32767), whereas the range of values for 32 bits is (−2147483648, 2147483647).

The type **short** is a signed type, which may use less storage than **int**. This can be used to save memory storage if only small numbers are needed. The type **long** is a signed integer that can be used to store integers that require a larger range than type **int**. The type **unsigned** is an integer that can only have a positive value. This can be used in situations where the data objects will never be negative. Therefore, when choosing which data type to be used for an object, we should select the type whose range is just enough to cover all the possible values of the object.

Floating Point Type

Any data object, which is a real number, is in this category. Floating point number representation is similar to scientific notation that is especially useful in expressing very small or very large numbers. Exponential notation is used by most computers to represent floating point numbers. There are three ways to store a floating point object for C in the computer. Therefore, three C floating point types are available: float, double and long double.

32 bits are used to store an object of type **float**. Eight bits are used for the exponent, and 24 bits are used to represent the non-exponential part. It typically has six or seven digits of precision and a range of 10^{-37} to 10^{+38}. The data type **double** is used to store double-precision floating point numbers. It normally uses twice as many bits of memory as **float**, typically 64 bits. ANSI C also allows for a third floating point type called **long double**. It aims to provide for even more precision than **double**.

The three numeric data types **int**, **float** and **double** are stored differently in the computer. The **int** data type requires the least memory space, the **double** data type requires the most memory space, and the **float** data type falls in between. Operations involving the **int** data type are always *exact*, while the **float** and **double** data types can be *inexact*. There is an

infinite number of floating point numbers within each range covered by a type. Only some of these numbers can be represented exactly.

Character Type

Any data object, which is an English letter, an English punctuation mark, a decimal digit, or a symbol such as a space, is in this category. A character is actually stored internally as an integer. The conversion from characters to integer numbers is done commonly using the ASCII code. The ASCII (American Standard Code Information Interchange) code is the most commonly used character set. There are 128 distinct characters in the ASCII character set. Each character has a corresponding numeric ASCII code. For example, the character '**A**' has the ASCII value of 65 (refer to the ASCII character set).

There are two C character types: **char** and **unsigned char**. Both character types use 8 bits (1 byte) to store a character. So there are 256 possible values. The range of values for **char** is (−128, 127), whereas the range of values for **unsigned char** (0, 255).

Constants

- A constant is an object whose value is **unchanged** throughout the life of the program. There are four types of constants: integer constants, floating point constants, character constants and string constants.
- Four types of constant values:
  - **Integer**: e.g. 100, –256; **Floating-point**: e.g. 2.4, –3.0;
  - **Character**: e.g. 'a', '+' ; **String**: e.g. "Hello Students "
- **Defining Constants – by using the preprocessor directive #define**

  Format: **#define** CONSTANTNAME **value**
  E.g. **#define TAX_RATE** 0.12
  /* define a constant TAX_RATE with 0.12 */
- **Defining Constants - by defining a constant variable**

  Format: **const type varName = value;**
  E.g. **const** float **pi = 3.14159**;
  /* declare a float constant variable pi with value 3.14159 */
  printf("pi = %f\n", pi);

10

Constants

1. A constant is an object whose value is unchanged (or cannot be changed) throughout program execution. There are four types of constants: integer constants, floating point constants, character constants and string constants.
2. When defining constants, we can use the preprocessor directive **#define**. The format of **#define** is **#define CONSTANT_NAME value.**
3. Another way to define a constant is to use a constant variable. This is done by using the **const** qualifier as **const type variableName=value;**

Four Types of Constants

A constant is an object whose value is unchanged (or cannot be changed) throughout the program execution. There are four types of constant values:

- Integer: e.g. 100, –256;
- floating point: e.g. 2.4, –3.0;
- character: e.g. 'a', '+'; and
- string constants which are a string of characters: e.g. "have a good day".

Integer Constants

Integer constants may be specified in decimal, octal or hexadecimal notation. A decimal

integer constant is written as a sequence of decimal digits (0-9), e.g. 1234. An octal integer constant is written as a sequence of octal digits (0-7) starting with a zero (**0**), e.g. **077**. A hexadecimal integer constant is a sequence of hexadecimal digits starting with **0x** or **0X**, e.g. **0XFF**. The hexadecimal digits comprise 0-9, and the letters A to F (or a to f), where the letters represent the values 10 to 15 respectively. A decimal, octal or hexadecimal integer constant can immediately be followed by the letter **L** (or letter **l**) to explicitly specify a **long** integer constant, e.g. **1234L** and **0XFFL**.

Floating Point Constants

Floating point constants may be written in two forms. The first is in the form of a sequence of decimal digits including the decimal point, e.g. 3.1234. The second is to use the exponential notation as discussed in the previous section, e.g. **31234e3**. All floating point constants are always stored as **double**. If the floating point numbers are qualified by the suffix **f** or **F**, e.g. **1.234F**, then the values are of type **float**. We may also qualify a floating point number with **l** or **L** to specify the value to be stored is of type **long double**.

Character Constants

Character constants can be given by quoting the numerical value of a character, e.g. 65 for the character **A** in the ASCII character set, or by enclosing it with quotes, e.g. '**A**'. Some useful non-printable control characters are referred to by the *escape sequence* which consists of the backslash symbol (****) followed by a single character. The two symbols together represent the single required character. This is a better alternative, in terms of memorization, than ASCII numbers. For example, '**\n**' represents the newline character instead of using the ASCII number 10. Notice that escape sequences must be enclosed in single quotes.

The escape sequence may also be used to represent octal and hexadecimal ASCII values of a character. The formats for octal and hexadecimal values are '**\0oo**' and '**\xhh**'. For example, '**\007**' and '**\x7**' are octal and hexadecimal values for the alarm bell. The character constant '**A**' (with ASCII encoding of decimal 65, octal 101 or hexadecimal 41) may be represented as '**\0101**' or '**\x41**' for octal and hexadecimal values respectively.

String Constants

A *string* constant is a sequence of characters enclosed in double quotation marks. It is different from a character constant. For example, '**a**' and "**a**" are different as '**a**' is a character while "**a**" is a string. Strings will be discussed in details in the chapter on Character Strings.

Defining Constants

We can use the preprocessor directive **#define**. The format of **#define** is

#define CONSTANT_NAME value

where **CONSTANT_NAME** is the name of the constant. Some examples are given as follows:

#define YES 'Y'

#define GREETINGS "How are you?"

#define ALARM '\a'

It is a C tradition that **CONSTANT_NAME** is in upper case. It makes programs more readable. During compilation, the value of the constant will substitute the name of the constant wherever it appears in the program. By giving a symbolic name to a constant, it improves the readability of the program and makes the program easier to be modified.

Another way to define a constant is to use a constant variable. This is done by using the **const** qualifier as follows:

const type varName=value;

where **type** can be **int**, **float**, **char**, etc. **varName** is the name of the constant variable. **value** is the constant value to be assigned to the constant variable. For example,

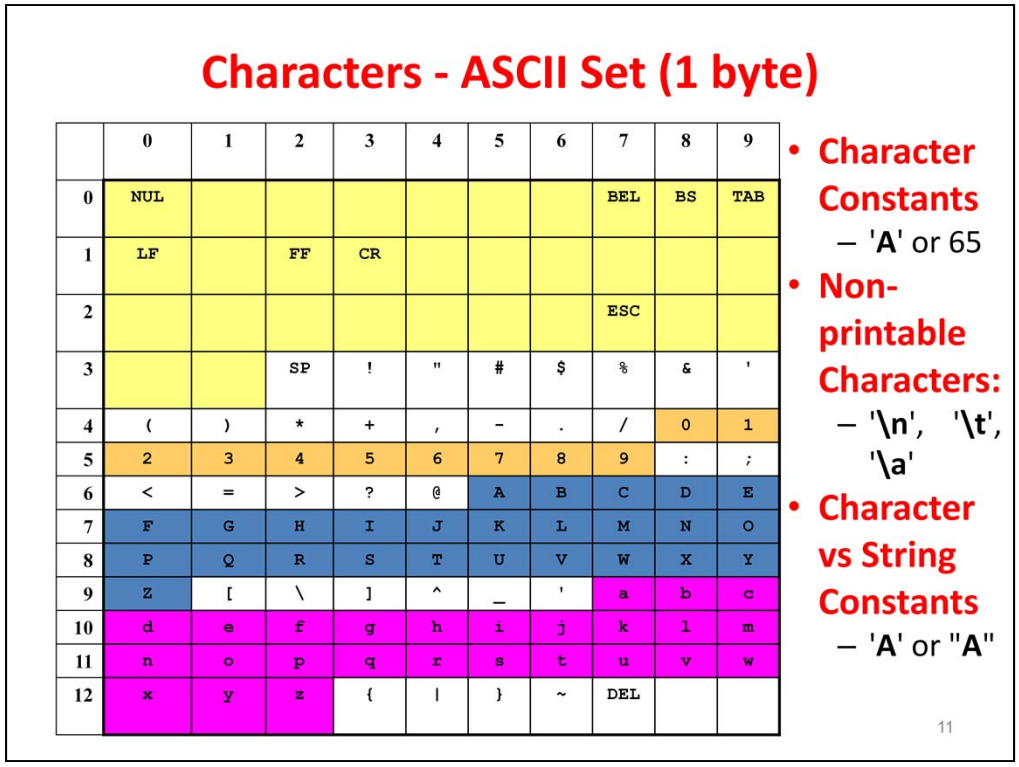
const float pi = 3.14159;

C provides **#define** and **const** to define symbolic names for constants. We recommend using **#define** to name constants of simple data types, and **const** to define constants that depend on another constant. For example, in the declarations

#define TAX_RATE 0.12

const double monthRate = TAX_RATE/12;

#define is used for defining the annual tax rate, while **const** is used to define the monthly tax rate which is one-twelfth of the annual tax rate.



Characters - ASCII Set

1. Character constants can be given by quoting the numerical value of a character, e.g. 65 for the character **A** in the ASCII character set, or by enclosing it with quotes, e.g. '**A**'.
2. Some useful non-printable control characters are referred to by *escape sequence* which consists of the backslash symbol (****) followed by a single character. For example, '**\n**' represents the newline character, instead of using the ASCII number 10.
3. A *string* constant is a sequence of characters enclosed in double quotation marks. It is different from a character constant. For example, '**a**' and "**a**" are different as '**a**' is a character while "**a**" is a string. Strings will be discussed in the chapter on Character Strings.

Character and String Constants

Character constants can be given by quoting the numerical value of a character, e.g. 65 for the character **A** in the ASCII character set, or by enclosing it with quotes, e.g. '**A**'.

A *string* constant is a sequence of characters enclosed in double quotation marks. It is different from a character constant. For example, '**a**' and "**a**" are different as '**a**' is a character while "**a**" is a string. Strings will be discussed in the chapter on Character Strings.

Characters – Escape Sequence

Some useful non-printable control characters are referred to by the *escape sequence* which

consists of the backslash symbol (****) followed by a single character. The two symbols together represent the single required character. This is a better alternative, in terms of memorization, than ASCII numbers. For example, '**\n**' represents the newline character instead of using the ASCII number 10. Some other examples of escape sequence include '**\a**' (alarm bell), '**\t**' (horizontal tab), etc. Notice that escape sequences must be enclosed in single quotes.

The escape sequence may also be used to represent octal and hexadecimal ASCII values of a character. The formats for octal and hexadecimal values are '**\0oo**' and '**\xhh**'. For example, '**\007**' and '**\x7**' are octal and hexadecimal values for the alarm bell. The character constant '**A**' (with ASCII encoding of decimal 65, octal 101 or hexadecimal 41) may be represented as '**\0101**' or '**\x41**' for octal and hexadecimal values respectively.

Variables

- A variable declaration always contains 2 components:
  - its **data_type** (eg. short, int, long, etc.)
  - its **var_name** (e.g. count, numOfSeats, etc.)

  The syntax for variable declaration: **data_type var_name**[, **var_name**];
- Declare your variables at the **beginning** of a function in your program. Examples of variable initializations:

```
int count = 20;
float temperature, result;
```

- The following C **keywords** are **reserved** and **cannot** be used as variable names:

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

12

Variables

1. Variables are symbolic names that are used to store data in memory.
2. The syntax for variable declaration is **data_type var_name[, var_name]**;
3. During program execution, memory storage of suitable size is assigned for each variable.
4. A variable must be declared first before it can be used in your program. Also all variables should be declared at the beginning of a function before writing program statements. Initialization of variables can also be done as part of a declaration.
5. Note that C keywords are reserved and cannot be used as variable names.

Data and Variables

Programs work with data, and data is stored in memory. Variables are symbolic names that are used to store data in memory. The content in the memory location is the current value of the variable. We use the variable name in order to retrieve the value stored at the memory location. Variables are different from constants in that the value of a variable may change during program execution, while the value of a constant remains unchanged throughout the program execution. When a new value is assigned to the variable, the old value is replaced with the new value.

The name of a variable is made up of a sequence of alphanumeric characters and

the underscore '_'. The first character must be a letter or an underscore. There is a system-dependent limit to the maximum number of characters that can be used. The ANSI C standard limit is 31.

Variable name in C is also case sensitive. The variable names **count** and **COUNT** refer to two different variables. However, it is conventional to use lowercase letters to name variables. In addition, meaningful names make programs more readable. For example, '**tax**' will make a program easier to understand than '**t**'. A variable name cannot be any of the *keywords* in C. Keywords have special meanings to C compiler.

Each variable has a type. The basic C data types are integer, floating point and character. Variables are declared by *declaration statements.* A declaration can be done with or without initialization. A declaration statement without initialization has the format

data_type var_name[, var_name];

where data_type can be **int**, **char**, **float**, **double**, etc. **var_name** is the name of the variable. **[..]** may be repeated zero or more times but need to be separated by commas.

During program execution, memory storage of suitable size is assigned for each variable. A variable must be declared first before it is used. The memory location is used to store the current value of the variable. Variable names can then be used to retrieve the value stored in the memory location.

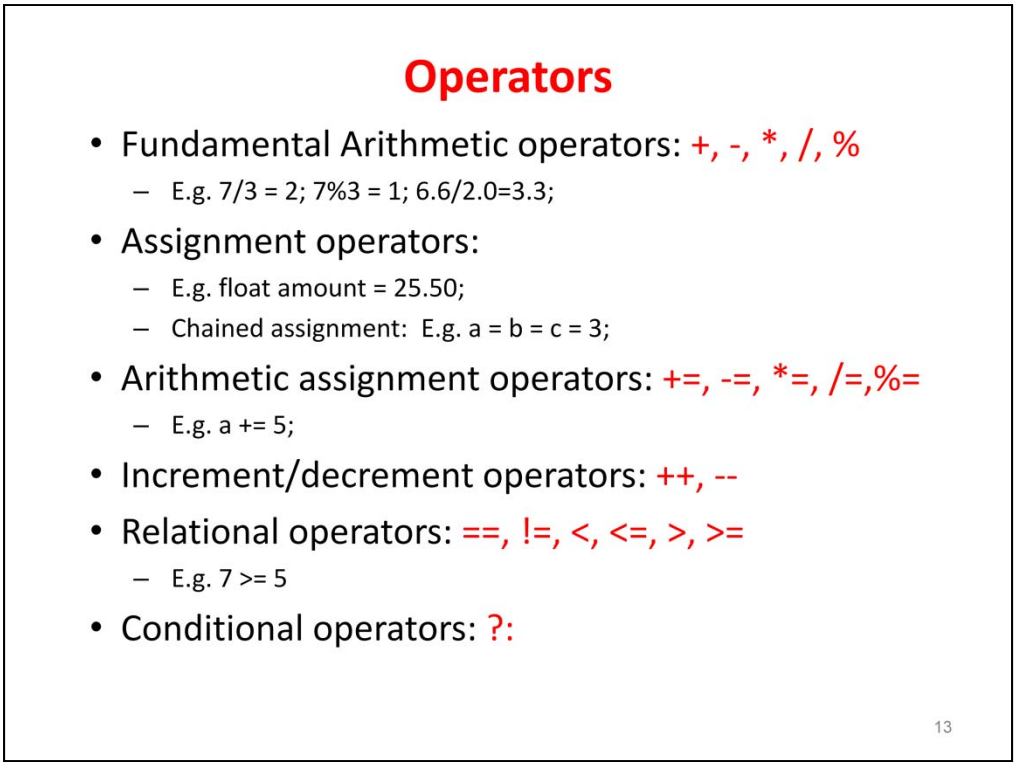
Initialization can also be done as part of a declaration. This means that a variable is given a starting value when it is declared. In the following program,

```
int main()
{
    float tax, salary;
    int numOfChildren = 2, numOfParents = 2;
    char maritalStatus = 'M';
    ....
    return 0;
}
```

The declaration statements for **numOfChildren**, **numOfParents** and **maritalStatus** declare and initialize the variables in one step. It is a very useful feature when developing complex programs.

The declaration statement **int numOfChildren, numOfParents = 2;** only initializes **numOfParents** to 2. **numOfChildren** is not initialized. To improve the readability of the program, it is better to declare initialized and uninitialized variables in separate declaration statements. Also note that a variable is not allowed to be declared twice. The following declaration statements will cause the compiler to issue an error message:

```
float tax, salary;
int tax;
```



Operators

1. Operators in C are mainly classified into fundamental arithmetic operators, assignment operators, arithmetic assignment operators, increment/decrement operators, relational operators and conditional operators.
2. Arithmetic and assignment operators are quite straightforward.
3. Next, we will discuss increment and decrement operators.
4. Relational and conditional operators will be discussed in the next lecture.

An operator is a symbol that causes some operations to be performed on one or more variables (or data values).

Fundamental Arithmetic Operators

There are two types of fundamental arithmetic operators: unary operators and binary operators. Unary operators are positive (**+**), negative (**-**) (e.g. **+31**, **-5**). They can be used to change the sign of a value. Binary operators are addition (**+**), subtraction (**-**), multiplication (*****), division (**/**) and modulus (**%**).

Note that the division of floating point numbers (or one floating point number with an integer) returns a floating point number. However, the division operator (**/**) for integers returns integer results. The modulus operator (**%**) returns the remainder of an integer division. The modulus operator works only on data types **int** and **char**.

The following examples give some of the arithmetic operations:

15.0/4 = 3.75

15/4.0 = 3.75

15/4 = 3

18/4 = 4

15%4 = 3

18%4 = 2

-18%4 = 2

It is also possible to perform arithmetic operations on character variables:

```
char chr1, chr2;
chr1 = 'A';          /* assign 'A' to the variable chr1 */
chr2 = chr1 + 32;  /* add 32 to the variable chr1 */
```

The last statement essentially converts the uppercase letter into the corresponding lowercase character. This is done by adding 32 to the ASCII code for the corresponding uppercase character. The result of the assignment statement is the character constant '**a**' to be assigned to the variable **chr2**. This is a very useful technique in converting an uppercase letter to its lowercase counterpart, or vice versa.

Assignment Operators

An assignment operator (**=**) is used to assign a value to a variable. It is the basic building block of a program. The value of a variable may be changed by the assignment statement in the following format:

```
var_name = expression;
```

where **var_name** is a variable name, and the **expression** (on the right-hand side) provides a value that is assigned to the variable **var_name** on the left-hand side. An expression may be a constant (e.g. **2**, '**h**', **-6.7**), a variable (e.g. **tax**, **salary**), or a combination of operators and operands (e.g. **a+b/c**). Notice that the left-hand side must be a variable name and cannot be a constant or expression with operators. The expression on the right-hand side can be a constant, a variable or an arithmetic expression.

The assignment operator has two operands. The operand on the left-hand side must be a variable or an expression that names the memory location. The expression on the right-hand side can be a variable, a constant or an expression that gives a value to be assigned to the left-hand side. When a variable, e.g. **num**, is declared, the compiler allocates a memory location for storing the value of that variable. Consider the following assignment statement:

```
num = 3;
```

the memory location of the variable **num** is used to store the value 3. The *value* of the variable refers to the value or content of the memory location, i.e. the actual value of the variable. In the following assignment statement:

count = num;

the value of **num** is assigned to the variable **count**.

The statements

3 = num;

num + count = 3;

are *illegal* as the left-hand side is not a memory location. In the first statement, the left-hand side is a constant and in the second statement, the left-hand side is an arithmetic expression, which does not represent a memory location.

Arithmetic Assignment Operators

In addition to the assignment operators, C also has arithmetic assignment operators, which combine an arithmetic operator with the assignment operator. Arithmetic assignment operator statement has the following format:

var_name op= expression;

where the arithmetic assignment operators (**op=**) are **+=**, **-=**, ***=**, **/=** and **%=**. This is equivalent to

var_name = var_name op expression;

The arithmetic assignment operator first performs the arithmetic operation specified by the arithmetic operator, and assigns the resulting value to the variable.

The assignment statements can also involve increment/decrement operators:

var_name1 = var_name2++;

var_name1 = ++var_name2;

var_name1 = var_name2--;

var_name1 = --var_name2;

For example, if we define two variables **m** and **k** as follows:

int m, k=2;

the statement

m = k++;

which is the same as

m = k;

k = k + 1;

will give **m** the value 2 and **k** the value 3 after executing the statement. However, the statement

m = ++k;

which is the same as

k = k + 1;

m = k;

will give **m** the value 3 and **k** the value 3 after executing the statement.

Chained Assignments

Chained assignments are also possible in one statement:

var_name1 = var_name2 = ... = var_nameN = expression;

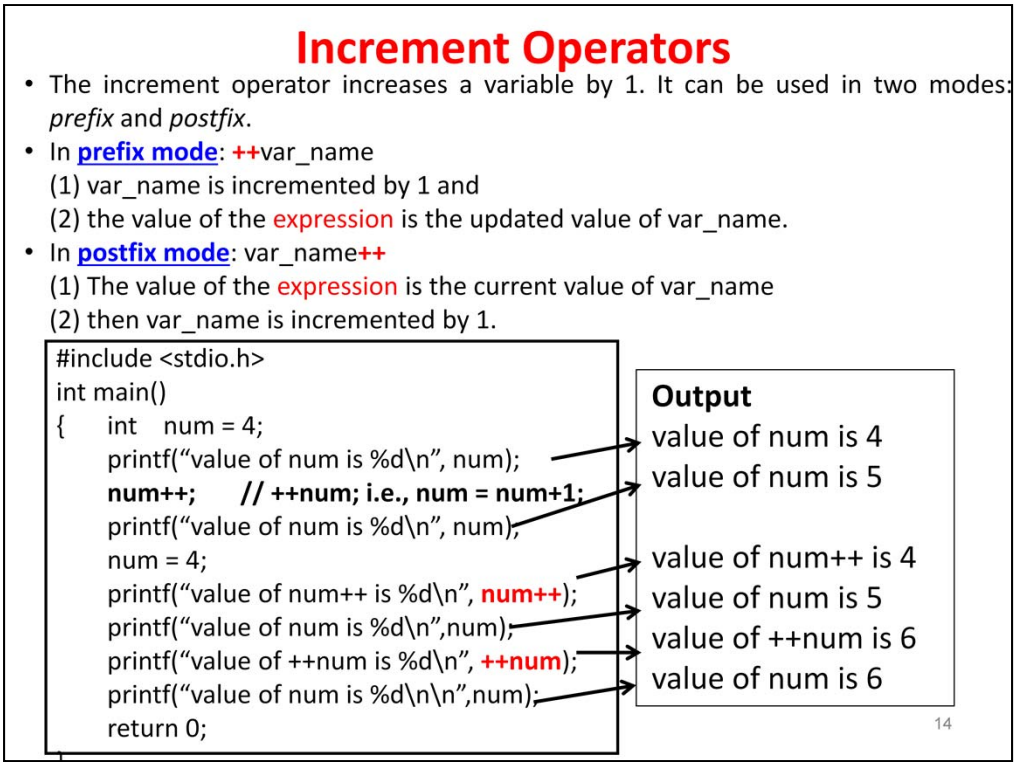
This is equivalent to

var_nameN = expression;

....

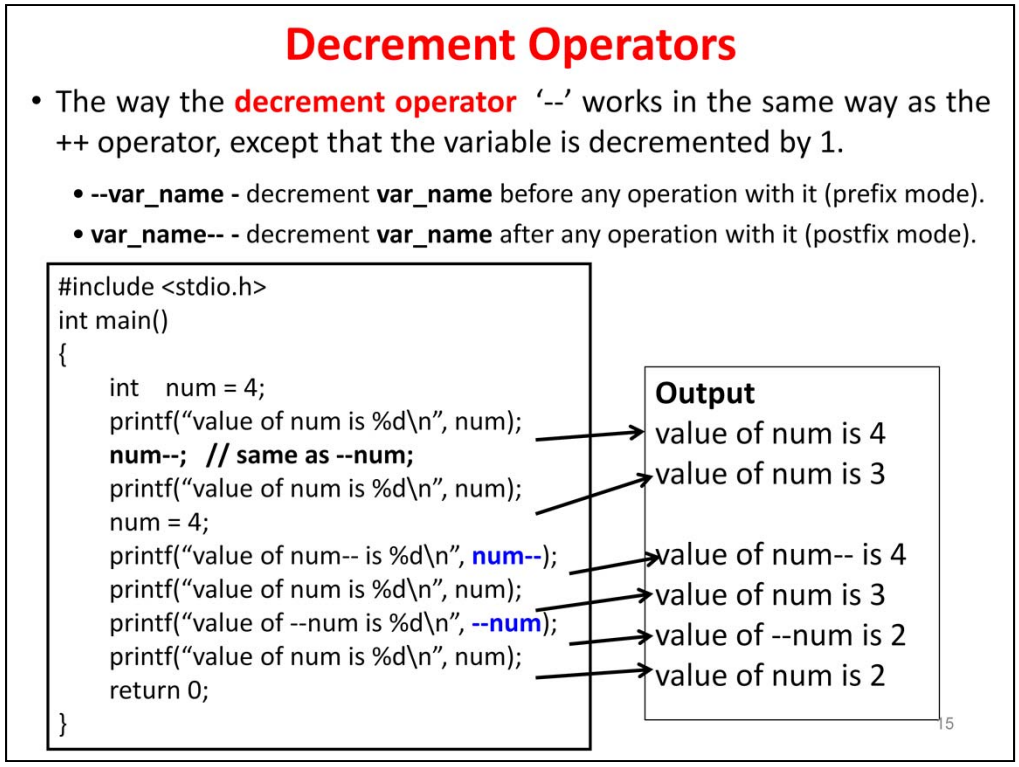
var_name2 = expression;

var_name1 = expression;



Increment/decrement Operators

1. The increment operator (**++**) increases a variable by 1. It can be used in two modes: *prefix* and *postfix*.
2. The format of the prefix mode is **++var_name;** where **var_name** is incremented by 1 and the value of the expression is the updated value of **var_name**. The format of the postfix mode is **var_name++;** where the value of the expression is the current value of **var_name** and then **var_name** is incremented by 1.
3. Notice that one important difference between the prefix and postfix modes is on the time when that operation is performed. In the prefix mode, the variable is incremented before any operation with it, while in the postfix mode, the variable is incremented after any operation with it.In the example program, it shows some examples on the use of increment/decrement operators. The initial value of the variable **num** is 4. The first **printf()** statement prints the value of 4 for **num**. The second **printf()** statement prints the value of 5 after increment. The variable **num** is assigned with the value 4 again. As the third **printf()** statement uses the postfix mode, the value of **num** is incremented by 1 after the printing has taken place. Therefore, the third **printf()** statement prints the value of 4 for **num**. The fourth statement then prints the value of 5. The fifth **printf()** statement uses the prefix mode. The value of **num** is incremented before the printing is taken place. Thus, it prints the value 6. The sixth statement also prints the value 6 for **num**.



Increment/decrement Operators

1. The decrement operator (--) works in a similar way as the **increment** operator, except that the variable is decremented by 1.
   a) **--var_name -** decrement **var_name** before any operation with it in prefix mode.
   b) **var_name-- -** decrement **var_name** after any operation with it in postfix mode.
2. The example code work similarly to that of the **increment** operator.

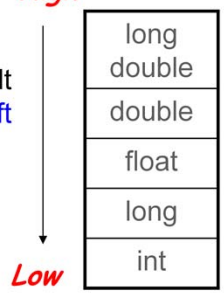
Data Type Conversion

Arithmetic operations require **two numbers** in an expression/assignment are of the **same type**.

If not, three kinds of conversion are available:

1. ***Explicit conversion*** - uses type casting operators, i.e. (int), (float), ..., etc.
   - e.g. (int)2.7 + (int)3.5
2. ***Arithmetic conversion*** - in mix operation, it converts the operands to the type of the **higher ranking** of the two.
   - e.g. double a; a = **2**+ 3.5; // 2 to 2.0 then add
3. ***Assignment conversion*** - converts the type of the result of computing the expression to that of the type of the left hand side if they are different.
   - e.g. int b; b = **2.7 + 3.5**; // 6.2 to 6 then to b

Note: Possible *pit-falls* about type conversion -
Loss of precision: e.g. from **float** to **int**, the fractional part will be lost.



16

Data Type Conversion

1. Data type conversion is the conversion of one data type into another type.
2. Data type conversion is needed when more than one type of data objects appear in an expression/assignment. For example, the statement: **a = 2 + 3.5;** adds two numbers with different data types, i.e. *integer* and *floating point*.
3. However, the addition operation can only be done if these two numbers are of the same data type.
4. Three kinds of conversion can be performed:
   a) Explicit conversion which uses type casting operation.
   b) Arithmetic conversion which converts the operands of an expression to the same data type in an arithmetic operation.
   c) Assignment conversion which converts to the data type of the result during assignment operation.
5. Note that there are possible pit-falls about type conversion, as it may cause the loss of precision.

Type conversion will take place when two operands of an expression are of different data types. Three kinds of conversion can be performed: (1) explicit conversion; (2) arithmetic conversion; and (3) assignment conversion.

In *explicit conversion*, it uses the type *cast* operator, (**type**), e.g. (**int**), (**float**), (**double**), etc.

to force the compiler to convert the value of the operand into the type specified by the operator. For example, the statements

```
int num;
float result;
result = (float)num;
```

explicitly convert the value of **num** into data type **float** and assign the value to the variable **result** of data type **float**.

In *arithmetic conversion*, it performs automatic conversion in any operation that involves operands of two different types. It converts the operands to the type of the higher ranking of the two. This process is also called *promotion*. The ranking of types is based on the amount of memory storage the value needs. The ranking from high to low is given as follows:

long double > double > float > long > int > char

Consider the following statements:

```
float ans1, ans2;
ans1 = 1.23 + 5/4;          /* first statement */
ans2 = 1.23 + 5.0/4;        /* second statement */
```

In the first statement, the expression **5/4** will be evaluated using integer division. It gives the result of 1. This result is then converted to 1.0 and added to the floating point constant 1.23 to give the final result of 2.23. This value is then assigned to the variable **ans1**. In the second statement, the expression **5.0/4** will be evaluated using floating point arithmetic. The value 4 in the expression will be converted to 4.0 before evaluating the expression. The result of 1.25 is then added to 1.23 to give the final result of 2.48. The result is then assigned to the variable **ans2**.

In *assignment conversion*, it converts the type of the result of computing the expression to that of the type of the left-hand side if they are different. If the variable on the left-hand side of the assignment statement has a higher rank or same rank as the expression, then there is no loss of precision. Otherwise, there could be a loss of accuracy. For example, if an expression has a floating point result, which is assigned to an integer variable, the fractional part of the result will be lost. This is because the lower ranking type may not have enough memory storage to store the value of higher ranking type. For example, the statements

```
int i;
float x=2.5, y=5.3;
i = x + y;
```

will give **i** the value 7.

Mathematical Libraries

#include <math.h>

Function	Argument Type	Description	Result Type
ceil(x)	double	Return the smallest double larger than or equal to x that can be represented as an int.	double
floor(x)	double	Return the largest double smaller than or equal to x that can be represented as an int.	double
abs(x)	int	Return the absolute value of x, where x is an int.	int
fabs(x)	double	Return the absolute value of x, where x is a floating point number.	double
sqrt(x)	double	Return the square root of x, where x >= 0.	double
pow(x,y)	double x, double y	Return x to the y power, x^y.	double
cos(x)	double	Return the cosine of x, where x is in radians.	double
sin(x)	double	Return the sine of x, where x is in radians.	double
tan(x)	double	Return the tangent of x, where x is in radians.	double
exp(x)	double	Return the exponential of x with the base e, where e is 2.718282.	double
log(x)	double	Return the natural logarithm of x.	double
log10(x)	double	Return the base 10 logarithm of x.	double

Mathematical Libraries

1. All C compilers provide a set of mathematical functions in the standard library.
2. Some of the common mathematical functions include square root **sqrt()**, power **pow()**, and absolute values **abs()** and **fabs()**.
3. In order to use any of the mathematical functions, we need to place the preprocessor directive **#include <math.h>** at the beginning of the program.

The **sqrt()** function computes the square root of a value of type **double**. It returns the result of type **double**. The statement

y = sqrt(x);

computes the square root of the value stored in the variable **x**, returns and stores the result in the variable **y**.

The functions **abs()** and **fabs()** give the absolute values of an integer and a floating point number respectively. The statements

y = abs(-5) + 3;

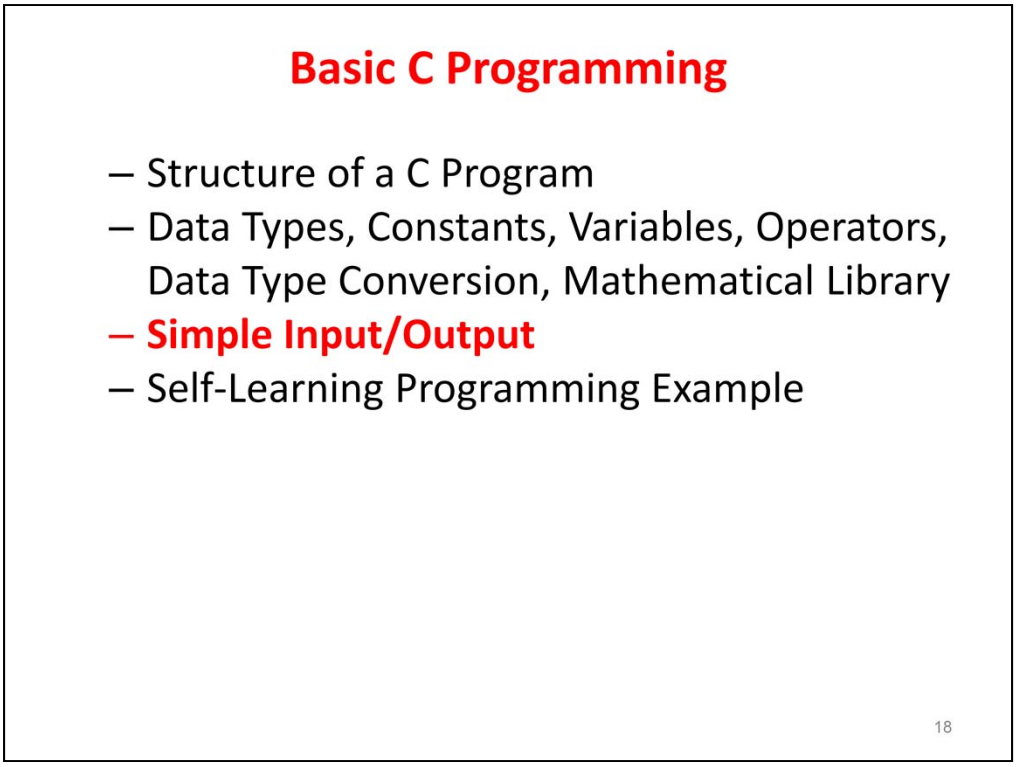
y = fabs(-5.5) + 3.5;

call the functions **abs()** and **fabs()**, and return the values 5 and 5.5 respectively.

The **pow()** function will take in two arguments, **x** and **y**, and returns $\mathbf{x^y}$.

Apart from the above functions, trigonometric functions such as **sin()**, **cos()** and **tan()** that

compute the sine, cosine and tangent of an angle are also provided.



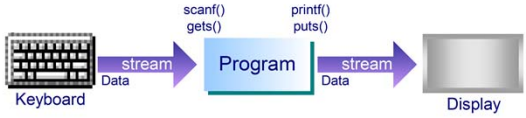
Basic C Programming

1. Here, we discuss simple input/output in C.

Simple Input/Output

- The following two Input/Output functions are most frequently used:

  - **printf()**: output function
  - **scanf()**: input function



- The I/O functions are in the C library **<stdio>**. To use the I/O functions, we need to include the header file:

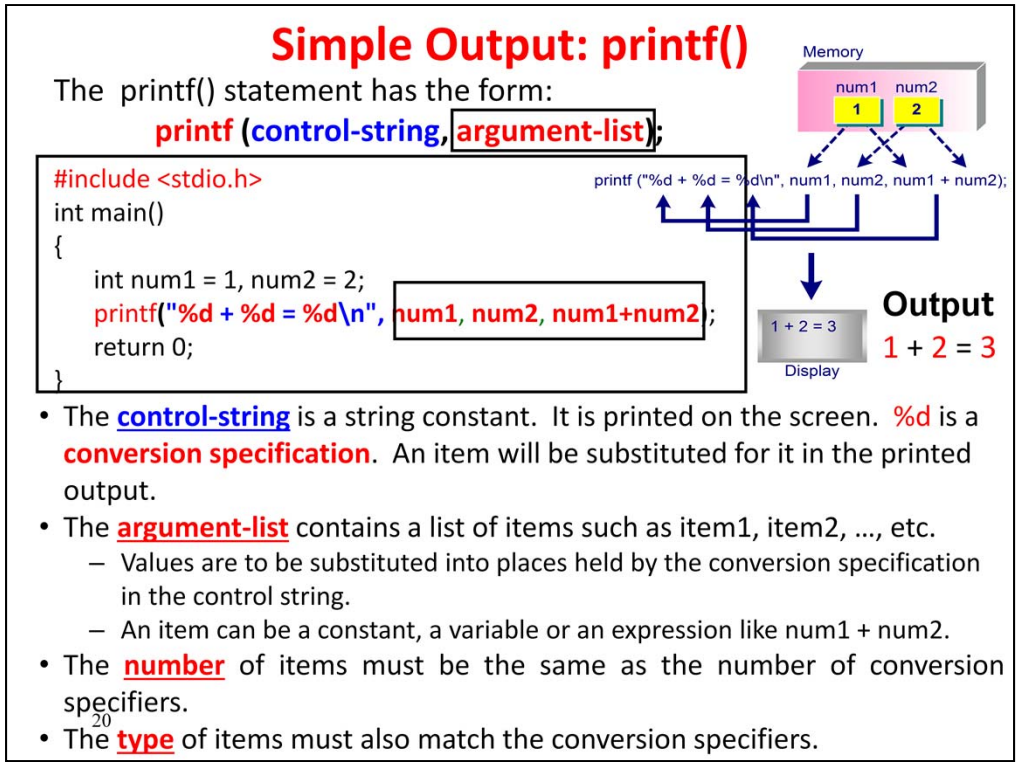
  #include <stdio.h>

  as the **preprocessor instruction** in a program.

19

Simple Input/Output

1. Most programs need to communicate with their environment. Input/output (or I/O) is the way a program communicates with the user. For C, the I/O operations are carried out by the I/O functions in the standard I/O libraries. Input from the keyboard or output to the monitor screen is referred to as standard input/output.
2. There are mainly four I/O functions, which communicate with the user's terminal:
   a) The **printf()** and **scanf()** functions perform formatted input and output respectively.
   b) The **putchar()** and **getchar()** functions perform character input and output respectively.
3. The standard I/O functions are in the library **<stdio>**. To use the I/O functions in **<stdio>**, the preprocessor directive **#include <stdio.h>** is included in order to include the header file in a program.



Simple Output: printf()

1. The **printf()** function allows us to control the format of the output. A **printf()** statement has the following format:

   printf(control-string, argument-list);

2. The **control-string** is a string constant **"%d + %d = %d\n"**. The string is printed on the screen. Two types of information are specified in the **control-string**. It comprises the characters that are to be printed and the **conversion specification** such as **%d**.
3. The **argument-list** contains a list of items such as **num1**, **num2**, **num1+num2** to be printed. They contain values to be substituted into places held by the conversion specifications in the **control-string**. An item can be a constant, a variable or an expression like **num1+num2**.
4. Note that the number of items must be the same as the number of conversion specifications, and the **type** of the item must match with the **conversion specifier**.

The **printf()** function is the most commonly used C library function. It allows us to control the format of the output. The **printf()** function can be used to print data of different data types in C language. A **printf()** statement has the following format:

printf(control-string, argument-list);

Control-string

The **control-string** which is enclosed in double quotation marks is a string constant. The string is printed on the screen. Two types of information are specified in the **control-string**. It comprises the characters that are to be printed and the **conversion specifications** (or **format specifiers**). In the program, **%d** is the conversion specification. It defines the ways the items are displayed on the screen. Conversion specifications can be placed anywhere within the **control-string**. Three conversion specifications with **%d** are specified in the program.

If the **control-string** is longer than one line, the continuation character '****' must be used:

```
printf("If a string is too long to fit on one line, \
is used when it is written on two lines");
```

Argument-list

The **argument-list** contains a list of items such as item1, item2, ..., etc. to be printed. They contain values to be substituted into places held by the conversion specifications in the **control-string**.

An **item** can be a constant, a variable or an expression such as **num1+num2**. This is illustrated in the example program. Note that the number of items must be the same as the number of conversion specifications, and the type of the item must match with the **conversion specifier**.

Control-String: Conversion Specification

- A **conversion specification** is of the form

% [flag] [minimumFieldWidth] [.precision]conversionSpecifier

  - *%* and *conversionSpecifier* are compulsory. The others are optional.

Note:
- We will focus on using the compulsory options % and **conversionSpecifier.**
- If interested, please refer to your textbook for the other options such as *flag*, *minimumFieldWidth* and *precision*.

21

Conversion Specification

1. The format of a conversion specification is **%[flag][minimumFieldWidth][.precision]conversionSpecifier**, where **%** and **conversionSpecifier** are compulsory. The others are optional.
2. The **conversionSpecifier** specifies how the data is to be converted into displayable form.
3. Here, we focus only on using the compulsory options **%** and **conversionSpecifier.**
4. Please refer to your textbook for the other options such as **flag**, **minimumFieldWidth** and **precision**.

The following options provide the additional control on how to print a value:

- A **flag** can be any of the five values: **+**, **-**, **space**, **#** and **0**. Zero or more flags may be present. It controls the display of plus or minus sign of a number, the display of the decimal point in floating point numbers, and left or right justification.
- The **minimumFieldWidth** field gives the minimum field width to be used during printing. For example, "**%10d**" prints an integer with field width of 10. If the width of the item to be printed is less than the specified field width, then it is right-justified and padded with blanks or zeros. If the field width is not enough, a wider field will be used. For example, if the specification is "**%2d**", and the integer value is 123, which is 3 digits long, then the field width of 3 instead of 2 is automatically used.

- The **precision** field follows the **minimumFieldWidth** and specifies the number of digits from the **minimumFieldWidth** after the decimal point. For example, "**%8.4f**" prints a floating point number in a field of 8 characters wide with 4 digits after the decimal point.

Control-String: Conversion Specification

Some common types of *Conversion Specifier*:

d	signed decimal conversion of int
o	unsigned octal conversion of unsigned
x,X	unsigned hexadecimal conversion of unsigned
c	single character conversion
f	signed decimal floating point conversion
s	string conversion

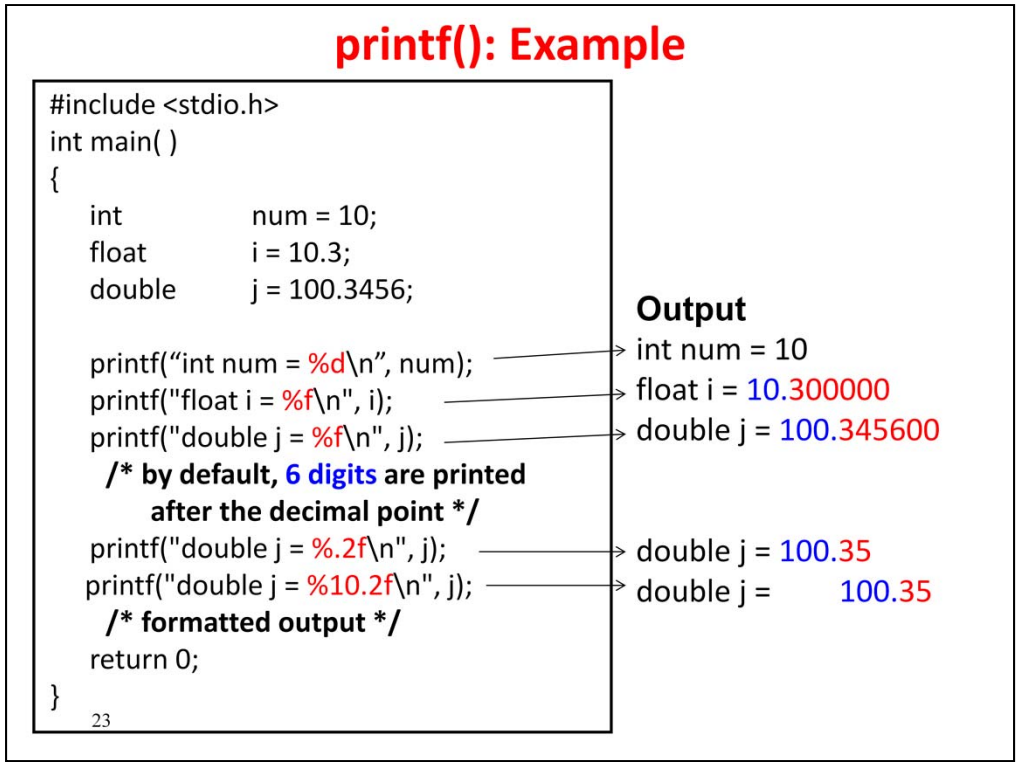
22

Conversion Specifier

1. The most common types of conversion specifier are:
   a) d - decimal integers
   b) c - characters
   c) f - floating point numbers
   d) s - strings

Other types of conversion specifier include **e**, **E**, **p**, **g**, **G** and **%**:

- **e, E, g, G** - floating point number in decimal or scientific format
- **p** - a pointer
- **% -** the % symbol



An Example Program on using printf()

1. In the program, it prints an integer using conversion specification **%d** and two floating point numbers using the conversion specification **%f** with different options.
2. Note that the second and third **printf()** statements print the numbers using **%f**. The default precision of 6 is used, that is, six digits are printed after the decimal point.
3. The fourth **printf()** statement prints the floating point number with precision 2 using conversion specification %.2f, that is, only two digits are printed after the decimal point.
4. Similarly in the fifth **printf()** statement prints the floating point number using conversion specification %10.2f. That is, the field width is limited to 10 with precision 2. Also, the floating point number to be printed is right-justified in the field.

Simple Input: scanf()

- A **scanf()** statement has the form

  scanf (control-string, argument-list);

- ***control-string*** is a string constant containing conversion specifications.
- The **argument-list** contains the **addresses** of a list of items.
  - The **items** in **scanf()** may be any variable matching the type given by the conversion specification. It cannot be a constant. It cannot be an expression like n1 + n2.
  - The **variable name** has to be preceded by an **&**. This is to tell **scanf()** the **address** of the variable so that **scanf()** can read the input value and store it in the variable's memory.
- **scanf()** uses whitespace characters (such as tabs, spaces and newlines) to determine how to separate the input into different fields to be stored.
- **scanf()** stops reading when it has read all the items as indicated by the control string or the **EOF** (end of file) is encountered.

The scanf() Function

1. The **scanf()** function is an input function that can be used to read formatted data. A **scanf()** statement has the following format: **scanf(control-string, argument-list);**, where **control-string** is a string constant containing conversion specifications.
2. The **argument-list** contains the **addresses** of a list of input items. The input items may be any variables matching the type given by the conversion specification. The variable name has to be preceded by an address operator **&**. This is to tell **scanf()** the address of the variable so that **scanf()** can read the input value and store it in the memory that is allocated to the variable. Commas are used to separate each input item in the argument-list.
3. **scanf()** uses whitespace characters (such as tabs, spaces and newlines) to determine how to separate the input into different fields to be stored.
4. **scanf()** stops reading when it has read all the items as indicated by the control string or the **EOF** (end of file) is encountered.

The **scanf()** function is an input function that can be used to read formatted data. The **scanf()** function reads input character strings from the input device (e.g. keyboard), converts the strings character-by-character to values according to the specified format and then stores the values into the variables.

scanf() uses whitespace characters (tabs, spaces and newlines) to determine how to

separate the input into different fields to be stored. It ignores consecutive whitespace characters in between when matching up consecutive conversion specifications to different consecutive fields. Conversion specification that begins with a percent sign (**%**) reads characters from input, converts the characters into values of the specified format, and stores the values to the address memory locations of the specified variables.

Conversion Specification

A conversion specification is of the form:

%[flag][maximumFieldWidth][.precision]conversionSpecifier

where **%** and **conversionSpecifier** are compulsory. The others are optional. It is similar to the conversion specification of the **printf()** statement. Common examples of conversion specifiers are **d**, **f**, **o**, **u**, **x**, **X**, **c** and **s**. Other conversion specifiers include:

- **e, E, g, G** - floating point number in decimal or scientific format.
- **p** - a pointer input.
- **i –** integer number in decimal, octal or hexadecimal integer format.

The options available include:

- The **flag** value of ***** means reading the next input without assigning the value to the corresponding item, i.e. skipping instead of assigning the value to the next variable's memory location.
- The **maximumFieldWidth** in **scanf()** is the maximum number of characters to read rather than the minimum as in the **printf()** statement. The reading of input will also stop when the first whitespace is encountered.

In summary, the **scanf()** function reads the input character one at a time. It skips over whitespace characters until it finds a non-whitespace character. It then starts reading the characters until it encounters a whitespace character. However, **scanf()** will stop reading a particular input field under the following conditions:

- If the **maximumFieldWidth** field is used, **scanf()** stops at the field end or at the first whitespace, whichever comes first.
- When the next character cannot be converted into the expected format, e.g. a letter is read instead of a digit when the expected format is decimal. In this case, the character is placed back to the input stream. This means that the next input starts at the unread, non-digit character.
- A matching non-whitespace character is encountered.

scanf(): Example

- A scanf() statement has the form
  scanf (control-string, argument-list);

```
#include <stdio.h>
int main( )
{
    int   n1, n2;
    float f1;
    double f2;

    printf("Please enter 2 integers:\n");
    scanf("%d %d", &n1, &n2);
    printf("The sum = %d\n", n1+n2);
    printf("Please enter 2 floats:\n");
    scanf("%f %lf", &f1, &f2);
    // Note: use %lf for double data
    printf("The sum = %f\n", f1+f2);
    return 0;
}
```

25

Output

```
Please enter 2 integers:
5 10
The sum = 15
Please enter 2 floats:
5.3 10.5
The sum = 15.800000
```

An Example on using scanf()

1. In the program, the first **scanf()** statement uses a whitespace to separate user input.
2. In the second **scanf()** statement, the **conversion specifier** "**%f**" is used to read a value in **float** data type, while the **conversion specifier** "**%lf**" is used to read a value in **double** data type.

The **scanf()** function can also return the number of items that it has successfully read. If no item is read, **scanf()** returns the value 0. If end-of-file is encountered, it returns **EOF**. We can modify the first **scanf()** statement in the program to return the number of items read as

count = scanf("%d %d", &n1, &n2);

where **count** is a variable of data type **int**.

Character Input/Output

putchar( )

- The syntax of calling putchar is

```
putchar(characterConstantOrVariable);
```

It is equivalent to

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

- The difference is that putchar is **faster** because printf needs to process the control string for formatting. Also, it returns either the integer value of the written character or EOF if an error occurs.

getchar ( )

- The syntax of calling getchar is

```
ch = getchar(); // ch is a character variable.
```

It is equivalent to

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

26

Character Input/Output

1. There are two functions in the <*stdio*> library to manage single character input and output: **putchar()** and **getchar()**.
2. The function **putchar()** takes a single argument, and prints the character. The syntax of calling **putchar()** is

   putchar(characterConstantOrVariable);

   which is equivalent to

   printf("%c", characterConstantOrVariable);
3. The difference is that **putchar()** is faster because **printf()** needs to process the control-string for formatting. Also, it needs to return either the integer value of the written character or **EOF** if an error occurs.
4. The **getchar()** function returns the next character from the input, or **EOF** if end-of-file is reached or if an error occurs. No arguments are required for **getchar()**. The syntax of calling **getchar()** is

   ch = getchar();

   where **ch** is a character variable. It is equivalent to

   scanf("%c", &ch);
5. The **getchar()** function works with the input buffer to get user input from the keyboard. The input buffer is an array of memory locations used to store input data transferred from the user input. A *buffer position indicator* is used to keep track of the position where the data is read from the buffer. The **getchar()** function retrieves the next data item from the position indicated

by the buffer position indicator and moves the buffer position indicator to the next character position. However, the **getchar()** function is only activated when the **<Enter>** key is pressed.

Therefore, when a character is entered, the input buffer receives and stores the input data until the **<Enter>** key is encountered. The **getchar()** function then retrieves the next unread character in the input buffer and advances the buffer position indicator.

For example, in the following program:

```
#include <stdio.h>
int main()
{
    char ch, ch1, ch2;
    putchar('1');
    putchar(ch='a');
    putchar('\n');
    printf("%c%c\n", 49, ch);
    ch1 = getchar();
    ch2 = getchar();
    putchar(ch1);
    putchar(ch2);
    putchar('\n');
    return 0;
}
```

when the user enters the data on the screen:

```
ab<Enter>
```

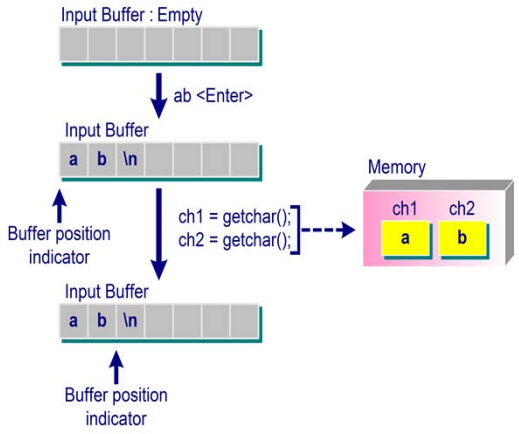
the input data, namely '**a**', '**b**' and '**\n**', will then be stored in the input buffer. The buffer position indicator points at the beginning of the buffer. After reading the two characters, '**a**' and '**b**', with the statements

```
ch1 = getchar();
ch2 = getchar();
```

the buffer position indicator moves two positions, and points to the buffer position that contains the newline '**\n**' character. As illustrated in this example, the two **getchar()** functions execute and read in the characters '**a**' and '**b**' only after the **<Enter>** key is pressed. However, the newline character (**\n**) still remains in the input buffer that needs to be taken care of before processing another input request. As mentioned earlier, one way to deal with the extra newline character is to flush the input buffer by using the **fflush(stdin)** function to empty the input buffer before the next input operation.

Character Input/Output

```
/* example to use getchar() and putchar()  */
 #include <stdio.h>
 int main(void)
{
      char ch, ch1, ch2;
      putchar('1');
      putchar(ch='a');
      putchar('\n');
      printf("%c%c\n", 49, ch);
      ch1 = getchar();
      ch2 = getchar();
      putchar(ch1);
      putchar(ch2);
      putchar('\n');
      return 0;
}
```



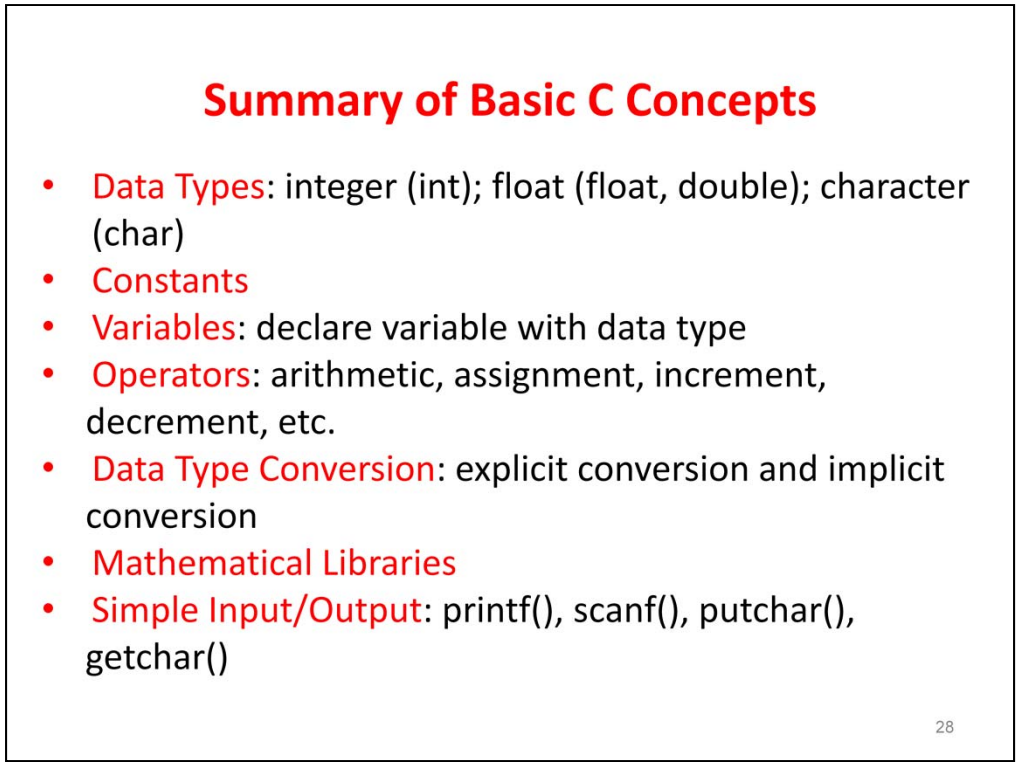
Output

1a
1a
ab *(User Input)*
ab

27

Character Input/Output

1. This slide gives an example on single character input and output: **putchar()** and **getchar()**.



Summary of Basic Concepts

1. The basic C concepts are summarized here:
   a) Data Types: integer (int); float (float, double); character (char)
   b) Constants
   c) Variables: declare variable with data type
   d) Operators: arithmetic, assignment, increment, decrement, etc.
   e) Data Type Conversion: explicit conversion and implicit conversion
   f) Mathematical Libraries
   g) Simple Input/Output: printf(), scanf(), putchar(), getchar()

Self-Learning Programming Example

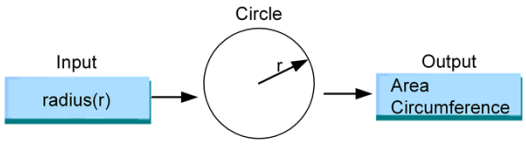
29

Programming Example: Writing a Simple C Program (Sequential Structure)

```
/* Purpose: A sample program to calculate the area and circumference.
   Author: S.C. Hui */

#include <stdio.h>
#define PI 3.14
int main( )
{   // declare variables
    float radius, area, circumference;
    // Read input
      /* Write your code here */
    // Computation
      /* Write your code here */

    // Print output
      /* Write your code here */
    return 0;
}
```



Area = π*r*r
Cirumference= 2*π*r

Output
Enter the radius: 5.0
The area is 78.50
The circumference is 31.40

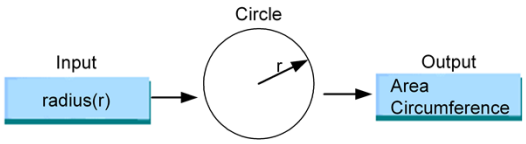
30

Programming Example: Writing a Simple Program

1. In this problem, write a program that computes the area and circumference of a circle. The program reads in the radius of the circle, computes the area and circumference, and displays the area and circumference on the screen.

Programming Example: Writing a Simple C Program (Sequential Structure)

```
/* Purpose: A sample program to calculate the area and circumference.
   Author: S.C. Hui */
#include <stdio.h>
#define PI 3.14
int main( )
{  // declare variables
   float radius, area, circumference;
   // Read input
   printf("Enter the radius: ");
   scanf("%f", &radius);
   // Computation
   area = PI * radius * radius;
   circumference = 2 * PI * radius;
   // Print output
   printf("The area is %.2f\n", area);
   printf("The circumference is %.2f", circumference);
   return 0;
}
```



Area = π*r*r
Cirumference= 2*π*r

Output
Enter the radius: 5.0
The area is 78.50
The circumference is 31.40

31

Program Code

1. The program code is given. The **printf()** and **scanf()** functions are used for I/O operations. The assignment statements are used in programming the logic.

Thank you !!!

32