3
Functions

1

Functions

1. In this lecture, we discuss Functions in C.

3
Functions

1

Functions

1. In this lecture, we discuss Functions in C.

Why Learning Functions?

- With Sequential, Branching and Looping, you will be able to build programs for simple applications. However, for more complex applications, your programs may be long and certain code may be repeated in the program.
- Functions aim to group specific tasks, so that code will not be repeated. It also helps to improve your program readability and efficiency.
- In this lecture, we discuss the concepts on functions.

2

Why Learning Functions?

1. With Sequential, Branching and Looping, you will be able to build programs for simple applications. However, for more complex applications, your programs may be long and certain code may be repeated in the program.
2. Functions aim to group specific tasks, so that code will not be repeated. It also helps to improve your program readability and efficiency.
3. In this lecture, we discuss the concepts on functions.

Functions

– **Function Definition**
– Function Prototypes
– Function Flow
– Parameter Passing: Call by Value
– Storage Scope of Variables
– Functional Decomposition

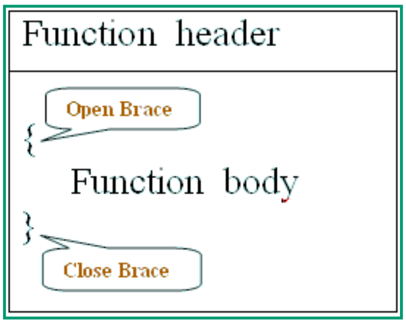
3

Functions

1. Here, we start by discussing function definition.

Function Definition

- A **function** is a self-contained unit of code to carry out a specific task, e.g. printf(), sqrt().
- A **function** consists of
  - a header
  - an opening curly brace
  - a function body
  - a closing curly brace



Example:

```
float findMax(float x, float y)  // header
{
    // function body
    float maxnum;

    if (x >= y)
       maxnum = x;
    else
       maxnum = y;

   return maxnum;
}
```

4

Function Definition

1. A function is a self-contained unit of code to carry out a specific task, e.g. **printf()**, **sqrt()**.
2. Each function definition consists of a function header and a function body.
3. The function body contains the code, which specifies the actions of the function, and the local data used by the function.
4. An example is illustrated in the **findMax()** function, which has the function header and function body.

Function Header

Return_type Function_name (Parameter_list)

- **Function_name**
  – specifies the name given to the function. Try to give a meaningful name to the function.
- **Parameter_list**
  – specifies a list of parameters which contains the data that are passed in by the calling function.
- **Return_type**
  – specifies the **type** of the data to be returned to the calling function.

5

Function Header

1. The function header has the following format: **Return_type Function_name(Parameter_list)**
2. **Function_name** specifies the name given to the function.
3. **Parameter_list** specifies a list of parameters which contains the data that are passed in by the calling function.
4. **Return_type** specifies the type of the data to be returned to the calling function.

Function Header: Parameter List

- Parameters define the **data** **passed into** the function.
- A function can have **no** parameter, **one** parameter or **many** parameters.

type parameterName[, type parameterName]

Example: float **findMax**(float x, float y)

- Each parameter has:
  - **parameter name**
  - **data type** (such as int, char, etc.) of the parameter
- The function assumes that these parameter inputs will be supplied to the function when they are being called.

6

Function Header: Parameter List

1. Parameters define the data passed into the function.
2. The **Parameter_list** can be **void** or a list of declarations for variables called parameters: **type parameterName[, type parameterName]**
4. A function can have no parameter, one parameter or many parameters.
5. Each parameter has a parameter name and data type of the parameter (such as **int**, **char**, etc.).
6. The function assumes that these parameter inputs will be supplied to the function when they are being called.

Function Header: Return_type

- **Return Type** is the data type **returned from** the function, it can be int, float, char, void, or nothing.
- The syntax for the **return** statement is **return (expression);**
- **void** – the function will not return any value.

```
void hello_n_times(int n)
{
     int count;
     for (count = 0; count < n; count++)
          printf("hello\n");
     /* no return statement */
}
```

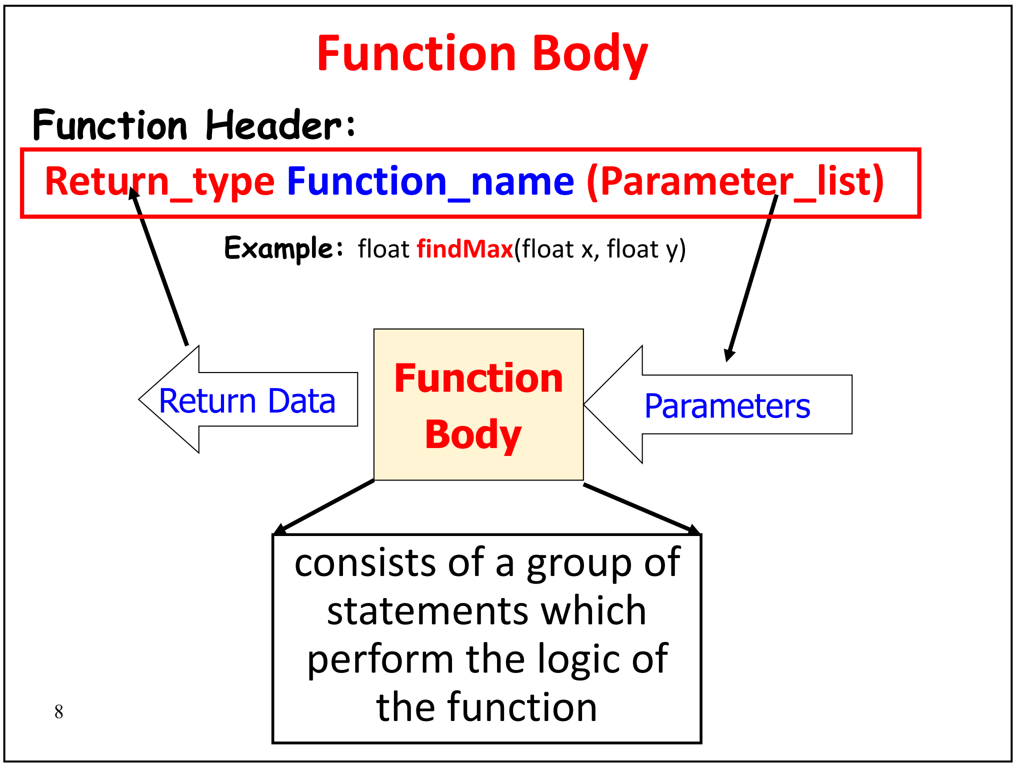
- **nothing** – if defined with **no type**, the **default type** is **int**.

```
successor(int num)   /* i.e. int successor(int num) */
{
     return num + 1; /* has a return statement */
}
```

7

Function Header: Return_type

1. **Return_type** is the data type of the value returned by the function, it can be **int**, **float**, **char**, **void**, etc.
2. The **return** statement is used for functions that return a value.
3. The syntax for the **return** statement is **return (expression);**
4. In the function **hello_n_times()**, when the return type is **void**, the function will not return any value. It prints a string "**Hello**" to the screen the number of times specified by the parameter **n**, which is defined to be of type **int**.
5. If nothing is specified for **Return_type** of a function header, i.e. when a function is defined with no type, for example, in the **successor**() function, then the default type **int** is used for that function. It means the function will return an integer value. Another example is the **main()** function.



<u>Function Body</u>

1. The function body consists of a group of statements.
2. The statements are executed when the function is called.
3. The variables declared inside the function body are called **local** variables and are only known within the function.
4. The main purpose of function body is to perform the logic of the function.

Multiple Return Statements

- The **return** statement terminates the execution of a function and passes the control to the calling function.
- The return statement may appear in **any place** or in **more than one place** inside the function body.

```
int factorial(int n)
{
    int temp = 1;    // local variable

    if (n < 0) {
        printf("error: must be +ve\n");
        return 0;
    }
    else if (n == 0)
        return 1;
    else
        for ( ; n > 0; n-- )
            temp *= n;
    return temp;
}
```

return a value

```
void hello_n_times(int n)
{
    int count;
    if (n < 0)
        return;      <-- no return value
    else
        for (count = 0; count < n; count++)
            printf("Hello!\n");
}
```

9

Multiple Return Statements

1. The **return** statement terminates the execution of the function and passes the control to the calling function.
2. The **return** statement may appear in any place or in more than one place inside the function body.
3. In the program, the function **factorial()** has **return** statements in various locations in the function body. If **n** is less than 0, then an error message is displayed, and the control is returned to the calling function. If **n** equals to 0, then the function returns a value 1. If **n** is greater than 0, then the factorial of **n** is evaluated using a **for** loop, and the result is returned.
4. A type **void** function may also have a **return** statement to terminate the function. This is illustrated in the function **hello_n_times**. However, it must not have a **return** expression. If the function does not have a **return** statement, then the control will be passed back to the calling function when the closing brace of the function body is encountered.
5. Sometimes, it is not necessary to use the value returned by a function. This is illustrated in the use of the functions such as **printf()** and **scanf()**. The **printf()** statement returns a value of type **int** that counts the number of characters printed, whereas the **scanf()** statement returns the number of items that are successfully read. However, if we do not require this information, we do not need to use the return value returned by these functions.

Function: Examples

Compute Grade:

```
char findGrade(float marks) {
    char grade;   // variable

    /* function body */
    if (marks >= 50)
        grade = 'P';
    else
        grade = 'F';
    return grade;
}
```

10

Function: findGrade()

1. The function **findGrade()** expects a parameter of type **float** and returns a value of type **char**.
2. The parameter **marks** is only accessible within the function **findGrade()**.
3. There is one variable **grade** defined in the function **findGrade()**. The variable is a local variable and can only be accessed within the function.
4. The variable is created when the function is called, and destroyed when the function ends.

Function: Examples

Compute Grade:

```
char findGrade(float marks) {
    char grade;   // variable

    /* function body */
    if (marks >= 50)
        grade = 'P';
    else
        grade = 'F';
    return grade;
}
```

Compute Circle Area:

```
float areaOfCircle(float radius) {
   const float pi = 3.14;
   float area;

   /* function body */
   area = pi*radius*radius;
   return area;
}
```

11

Function: areaOfCircle()

1. The function **areaOfCircle()** expects one parameter of type **float**. It returns a value of type **float**.
2. The parameter **radius** is only accessible within the function **areaOfCircle()**.
3. A local variable **area** is also declared in the function. This variable is only accessible within the function **areaOfCircle()**.
4. It is also created when the function is called, and will be destroyed when the function exits.

Functions

– Function Definition
– **Function Prototypes**
– Function Flow
– Parameter Passing: Call by Value
– Storage Scope of Variables
– Functional Decomposition

12

Functions

1. Here, we discuss function prototypes.

Function Prototypes

- We need to declare a function before using it in other functions.
- **Function prototype** is used to declare a function. It provides the information about
  1. the **return type** of the function
  2. the **name** of the function
  3. the **number** and **types** of the arguments
- The declaration may be the same as the function header but terminated by a **semicolon**.

Example:
float **findMax**(float x, float y) **;**

- Two ways to declare parameters in the parameter list:
  (1) With parameter name:
  void hello_n_times(int n);
  (2) Without parameter names:
  double distance(double, double);

13

Function Prototypes

1. We need to declare a function before using it in the **main()** function or other functions.
2. A function declaration is called a **function prototype**. It provides the information about the type of the function, the name of the function, and the number and types of the arguments.
3. The declaration is the same as the function header but terminated by a semicolon. For example, **void hello_n_times(int n);**
4. The function prototype can also be declared without giving the parameter names. For example, **double distance(double, double);**
5. Function prototypes enable the compiler to ensure that functions are being called properly. The compiler will check whether the number of arguments and the type of the arguments of the function call match with the parameters used in the function definition. Warning messages will be given if the number of arguments is different.

Function Prototypes: Where to declare it?

- The declaration has to be done **before** the function is called:
  - (1) **before** the main() header
  - (2) **inside** the main() body or
  - (3) **inside** any function which uses it

14

Function Prototypes: Where to declare it?

1. A function must be declared before it is actually called.
2. It can be declared either before the **main()** header, inside the **main()** body or inside any function which uses it.

Function Prototypes: Before the main()

- The declaration has to be done **before** the function is called:
  - (1) before the main() header
  - (2) inside the main() body or
  - (3) inside any function which uses it

Before the main():

```
#include <stdio.h>
int factorial(int n);   // function prototype

int main( )
{   int x;
    x = factorial(5);   // use factorial() here
}

int factorial(int n)    /* function definition*/
{
    ....
}
```

15

Function Prototype: Before the main()

1. If the function prototype is placed before the **main()** function and at the beginning of the program, it makes the function available to all the functions in the program.
2. In this example program, the function **factorial()** is declared outside the **main()**. Therefore, it can be used by all the functions in the program.

Function Prototype: Inside the main()

- The declaration has to be done **before** the function is called:
  (1) before the main() header
  (2) inside the main() body or
  (3) inside any function which uses it

Inside the main():

```
#include <stdio.h>

int main()
{   int x;
    int factorial(int);          // function prototype
      x = factorial(5);          // use factorial() here
      ....
}

int factorial(int n)             // function definition
{
....
}
```

16

Function Prototype: Inside the main()

1. In the second example program, the function **fact()** is declared inside the **main()** function.
2. This makes the function callable only within the **main()** function.

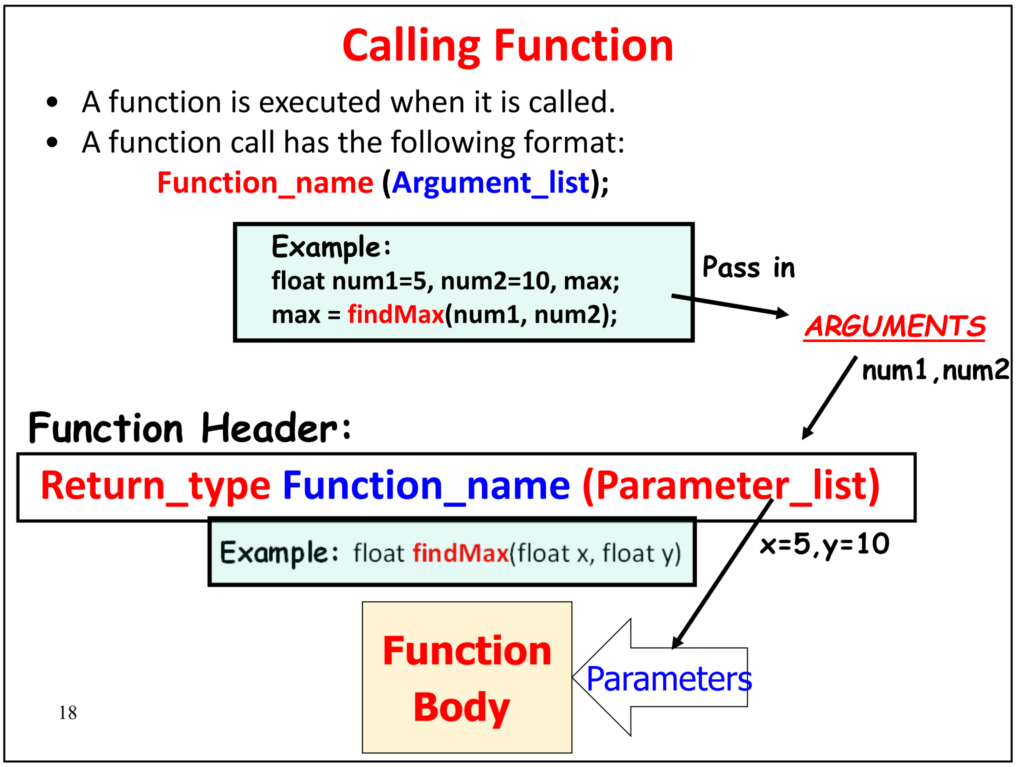
Functions

– Function Definition
– Function Prototypes
– **Function Flow**
– Parameter Passing: Call by Value
– Storage Scope of Variables
– Functional Decomposition

17

Functions

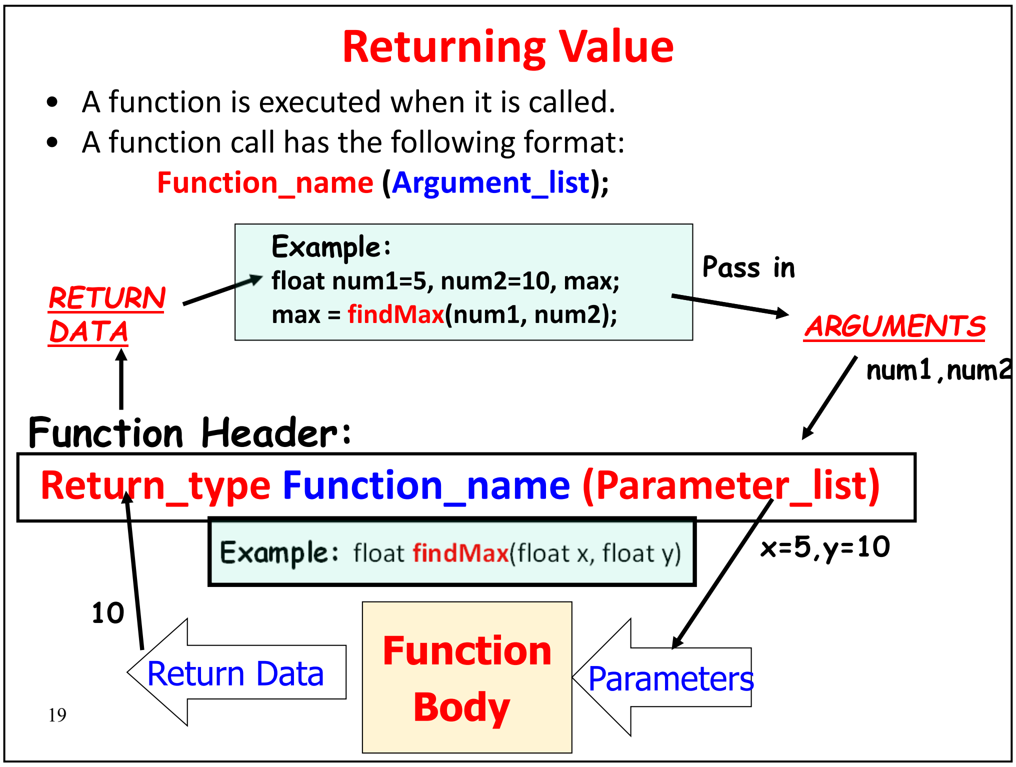
1. Here, we discuss function flow.



<u>**Calling Function**</u>

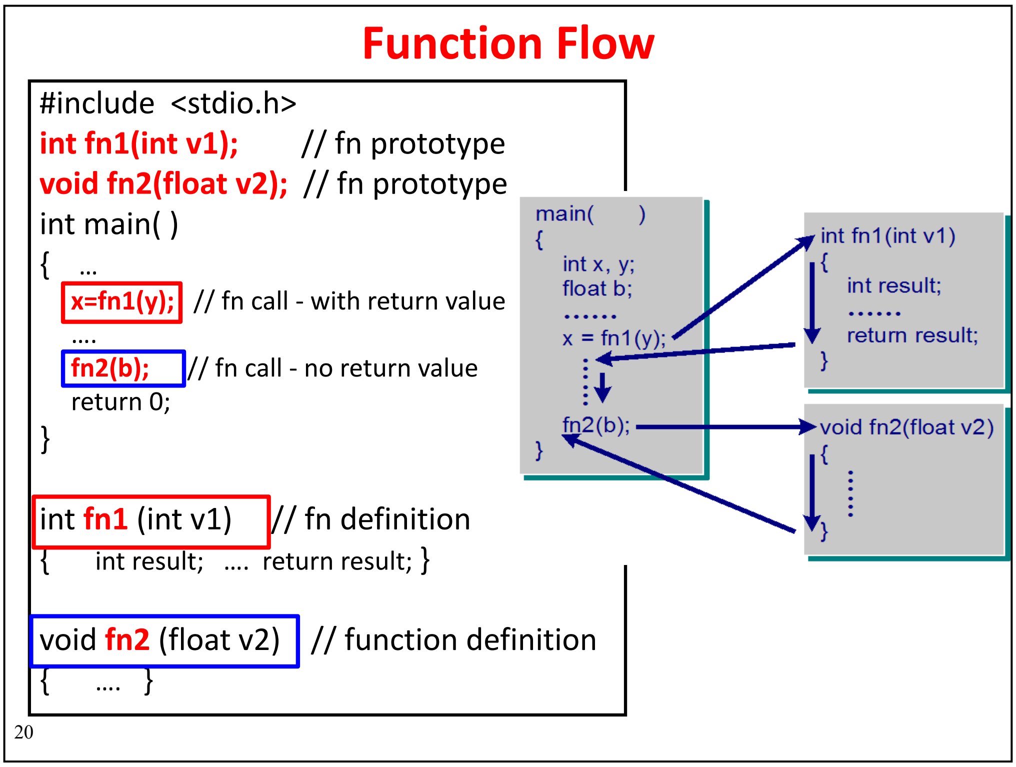
1. A function is executed when it is called.
2. A function call has the following format: **function_name(argument_list);**
3. A function can be called by using the function name followed by a list of **arguments**. For example, **num1** and **num2** in the **findMax()** function.
4. Function arguments can be constants, variables or expressions.
5. For the function **findMax()**, as **num1**=5, **num2**=10, the values 5 and 10 will be passed to the parameters **x** and **y** of the function respectively.

```
float findMax(float x, float y) {
  // function body
  float maxnum;
  if (x >= y)
     maxnum = x;
  else
     maxnum = y;
  return maxnum;
}
```



Returning Value

1. The function **findMax()** computes the maximum value of **x** and **y** (i.e. 5 and 10 respectively), and returns the value of 10 to the calling function.
2. The returned value is then assigned to the variable **max**.



Function Flow

1. In the program, the **main()** function will start the execution.
2. When the function **fn1()** is called, the program transfers the control to the **fn1()** function which then starts execution. As **fn1()** will return a value to the calling function, the statements in the function body of **fn1()** will be executed until a **return** statement is encountered.
3. Control is then transferred to the **main()** function. The value of the variable **result** will be assigned to the variable **x** in **main()**. The next statement after the function call then starts execution.
4. When the second function **fn2()** is called. The control is then transferred to **fn2()**. The function will execute until the end of the function body. Control will then be transferred to the **main()** function.

Function Flow: Example

Compute Grade:

```
#include <stdio.h>
char findGrade(float marks);
int main( )
{
    char answer;
    answer = findGrade(68.5);
    printf("Grade is %c", answer);
    return 0;
}
char findGrade(float marks) {
    char grade;   // variable
    if (marks >= 50)
      grade = 'P';
    else
      grade = 'F';
    return grade;
}
```

Output
Grade is P

21

Function Flow: findGrade()

1. In the program, the **main()** function calls the function **findGrade()**. When the statement: **answer = findGrade(68.5);** is executed, it calls the function **findGrade()**.
2. Control is then transferred to the function **findGrade()**. Information is passed between the calling function and the called function through the argument. In this case, the function receives one argument with the value of **68.5**. It is assigned to the corresponding parameter in the function definition to compute the grade.
3. When the execution of statements in the function body encounters the **return** statement, the control is then transferred to the **main()** function, and the statement just after the function call in **main()** will continue to execute.
4. The name for parameter needs not be the same as function argument. However, the number of arguments and the data type of the arguments must be the same as parameters defined in function definition. In the program, the argument **68.5** must correspond to the parameter **marks** in the function call.
5. Note that the function prototype is declared as: **float findGrade(float marks);** which is placed at the beginning of the program before the **main()** function.

Function Flow: Example

Compute Circle Area:

```
#include <stdio.h>
float areaOfCircle(float);
int main( )
{
    float answer;
    answer = areaOfCircle(2.5);
    printf("Area is %.1f", answer);
    return 0;
}
float areaOfCircle(float radius)  {
   const float pi = 3.14;
   float area;

   /* function body */
   area = pi*radius*radius;
   return area;
}
```

Output
Area is 19.6

22

Function Flow: areaOfCircle()

1. In the program, the **main()** function calls the function **areaOfCircle()**. When the statement: **answer = areaOfCircle(2.5);** is executed, it calls the function **areaOfCircle()**.
2. Control is then transferred to the function **areaOfCircle()**. Information is passed between the calling function and the called function through the argument. In this case, the function receives one argument with the value of **2.5**. It is assigned to the corresponding parameter in the function definition to compute the area of the circle.
3. When the execution of statements in the function body encounters the **return** statement, the control is then transferred to the **main()** function, and the statement just after the function call in **main()** will continue to execute.
4. Note that the function prototype is declared as: **float areaOfCircle(float);** which is placed at the beginning of the program before the **main()** function.

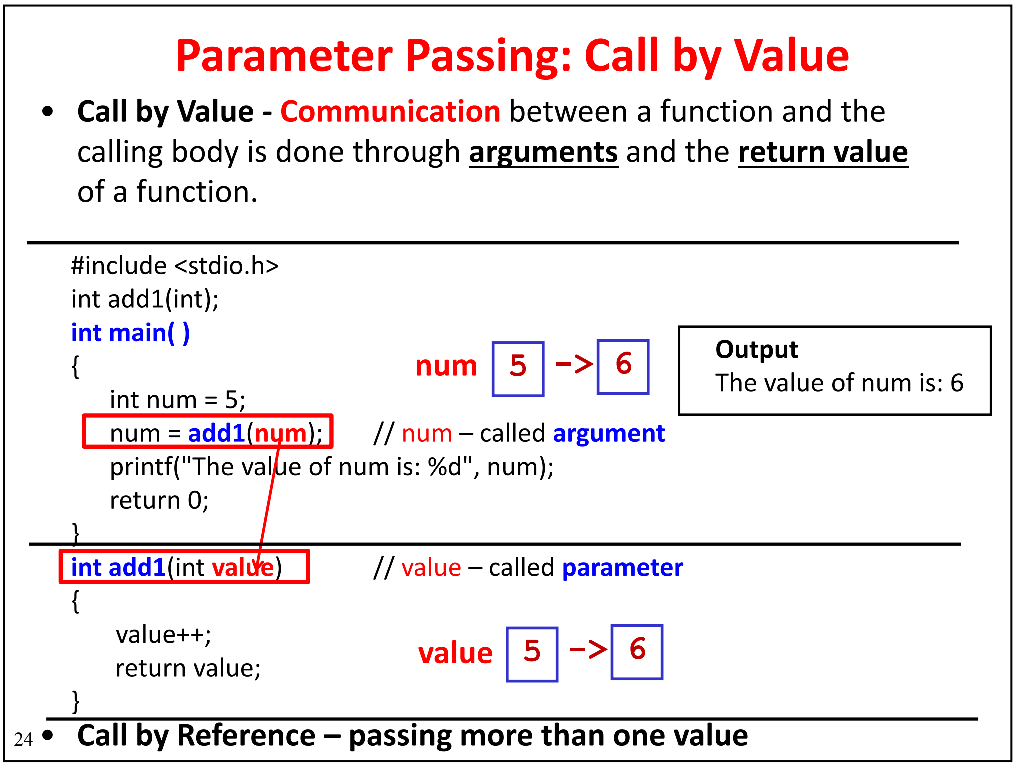
Functions

– Function Definition
– Function Prototypes
– Function Flow
– **Parameter Passing: Call by Value**
– Storage Scope of Variables
– Functional Decomposition

23

Functions

1. Here, we discuss parameter passing using call by value.



Parameter Passing: Call by Value

1. Communications between a called function and the calling function is through **arguments**. The called function then performs the task based on the received argument values. The called function can also return a value to the calling function.
2. Parameter passing between functions may be performed in two ways: **call by value** and **call by reference**. In call by value, the parameters must be declared in the function definition as regular variables. The arguments in function calls can be constants, variables or expressions.
3. When the function is called, the parameters hold a **copy** of the arguments locally. Therefore, any changes to the parameters in a function are done on the copy of the arguments.
4. This is illustrated in the function **add1()**. In the **main()** function, the variable **num** is assigned with 5. **num** is used as the argument when calling the function **add1()**.
5. The value in **num** is passed to the parameter value in the function **add1()**. The variable **value** is a local variable in the function. Then the variable **value** is incremented by 1. Finally, the value in the variable **value** is returned to the calling **main()** function which then assigns the returned value to the variable **num**.
6. In any programs, there are two ways for a called function to return values to the

calling function. The first way is to use the **return** statement as shown in the function **add1()**. However, this can only be used when only **a single value** needs to be returned from a function.

7. If **two or more values** need to be passed back from a called function, we need to use another approach called **call by reference** using pointers. We will discuss call by reference in the chapter on Pointers.

Parameter Passing: Example

```
#include <stdio.h>
#include <math.h>
double distance (double, double);  // function prototype

int main()
{
    double dist;
    double x=2.0, y=4.5, a=3.0, b=5.5;
    dist = distance(2.0, 4.5);    /* 2.0, 4.5 - arguments */
    printf("The dist is %f\n", dist);
    dist = distance(x*y, a*b);    /* x*y, a*b - arguments */
    printf("The dist is %f\n", dist);
    return 0;
}
double distance(double x, double y)   /* x,y-parameters */
  {
    return sqrt(x * x + y * y);
  }
```

Output

The dist is 4.924429

The dist is 18.794946

25

Parameter Passing: Example

1. In the program, it calls the function **distance()**. When the statement: **dist = distance(2.0, 4.5);** is executed, it calls the function **distance()** in **main()**. The arguments **2.0** and **4.5** in the **main()** function correspond respectively to the parameters **x** and **y** in the function call.
2. Control is then transferred to the function **distance()**. Information is passed between the calling function and the called function through arguments. In this case, the function receives two arguments with values of 2.0 and 4.5. They are assigned to the corresponding parameters in the function definition.
3. In addition, we can also use expression as an argument in the function as shown in the following statement: **dist = distance(x*y, a*b);** The arguments **x*y** and **a*b** in the **main()** function correspond respectively to the parameters **x** and **y** in the function call.
4. When the execution of statements in the function body encounters the **return** statement, the control is then transferred to the **main()** function, and the statement just after the function call in **main()** will continue to execute.
5. Note that the names for parameters need not be the same as function arguments. However, the number of arguments and the data type of the arguments must be the same as parameters defined in function definition.

Function Calling Another Function

```c
#include <stdio.h>
 int max3(int, int, int);          /* function prototypes */
 int max2(int, int);
 int main()
{
     int x, y, z;
     printf("input three integers => ");
     scanf("%d %d %d", &x, &y, &z);
     printf("Maximum of the 3 is %d\n", max3(x, y, z));
     return 0;
}
int max3(int i, int j, int k)
{
     printf("Find the max in %d, %d and \%d\n", i, j, k);
     return max2(max2(i, j), max2(j, k));
}
int max2(int h, int k)
{
     printf("Find the max of %d and %d\n", h, k);
     return h > k ? h : k;
}
```

Output

input three integers => *7 4 9*
Find the max in 7, 4 and 9
Find the max of 7 and 4
Find the max of 4 and 9
Find the max of 7 and 9
Maximum of the 3 is 9

26

Function Calling Another Function

1. A function may be called by **main()** or another function through call by value. In the program, the function **max2()** specifies two parameters, **h** and **k**, of type **int**, and receives two function arguments from the calling function. The values of the arguments are then stored in the memory locations of the two parameters, **h** and **k**. The function then compares their values, and returns the larger value to the calling function.
2. The function **max3()** specifies three parameters, **i**, **j** and **k**, and receives the function arguments from the calling function, and compares their values to determine the largest value.
3. The function **max3()** calls the function **max2()** to compare two values at a time and returns the maximum value: **return max2(max2(i,j), max2(j,k));**
4. Here, the function **max2()** is specified in the function **max2()** itself. The returned value from the called function **max2()** will be used again as arguments in the same function **max2()**. The maximum value is then returned to the calling function.

Functions

– Function Definition
– Function Prototypes
– Function Flow
– Parameter Passing: Call by Value
– **Storage Scope of Variables**
– Functional Decomposition

27

Functions

1. Here, we discuss the storage scope of variables.

Scope of Variables in a Function

- Scope of a variable
  - the section of code that can use the variable. In other words, the variable is visible in that section.
- Variables declared in a function is ONLY visible within that function. We call it **block scope**.
- Example below: variables **radius**, **pi** and **area** are **NOT** visible **outside** this function.

```
float areaOfCircle(float radius) {       // parameter – block scope
   const float pi = 3.14;                // const variable – block scope
   float area;                           // local variable – block scope
   area = pi*radius*radius;
   return area;
}
```

28

Scope of Variables in a Function

1. The scope of a variable determines the sections of the code that can use the variable. In other words, the variable is visible in that section of code.
2. Variables declared in a function is ONLY visible within that function. We call it block scope.
3. In the function **areaOfCircle()**, the variables **radius**, **pi** and **area** are not visible outside the function **areaOfCircle()**.

Local and Global Variables

- **Local variables**:
  - They are variables defined **inside** a function.
- **Global variables**:
  - They are variables defined **outside** the functions.

- Should global variables be used in your programs?
  - **Advantages** of using global variables:
    - simplest way of communication between functions
    - efficiency
  - **Disadvantages** of using global variables:
    - less readable program
    - more difficult to debug and modify

➢ **Strongly discouraged to use global variables** – instead you should use **parameter passing between functions** to achieve the same effect. So that **errors** will be **localized** within each function for easy debugging.

29

Local and Global Variables

1. **Local variables** are variables defined inside a function. They have block scope. They can be accessed only within the function. They cannot be accessed by other functions. Local variables are created when the function is invoked, and destroyed after the complete execution of the function.
2. **Global variables** are variables defined outside the functions. They have file (or program) scope. Thus, global variables are visible to all the functions that are defined following its declaration.
3. The advantages of global variables in programs are that global variables are the simplest way of communication between functions and they are efficient. The disadvantages of programs using global variables are that they are less readable and more difficult to debug and modify as any functions in the program can change the value of the global variables.
4. Therefore, it is a **good programming practice** to use local variables, and use parameter passing between functions for communication between functions. In this way, the value of each variable in the function is protected.
5. It is strongly discouraged to use global variables. Instead you should use parameter passing between functions to achieve the same effect. So that errors can be localized within each function for easy debugging.

Local and Global Variables: Example

```c
#include  <stdio.h>
int g_var = 5;                // global variable –  has file scope
int fn1(int, int);
float fn2(float);

int main( ) {
    char reply;               // local - these two variables are only
    int num;                  // known inside main() function - block scope
    …
}
int fn1(int x, int y)         //  local x,y - formal parameters  are only
(                             //  known inside this function – block scope
    float fnum;               //  local - these two variables are known
    int temp;                 //  in this function only – block scope
    g_var += 10;
    …
}
float fn2(float n)  {
    float temp;               //  local and block scope
    …
}
```

30

Local and Global Variables: Example

1. In the example program, the global variable **g_var** is declared outside the **main()** function. Global variables will have the file scope.
2. The variables **fnum** and **temp** are local variables which will have block scope and are only visible inside the function fn1.

Static Variables

- Static variables can be defined inside or outside a function using the **static** keyword.
  - The duration of a static variable is fixed.
  - Static variables are created at the **start** of the program and are destroyed only at the **end** of program execution. That is, they exist **throughout program execution** once they are created.
- If a **static** variable is defined and initialized, it is then **initialized once** when the storage is allocated. If a static variable is defined, but not initialized, it will be initialized to zero by the compiler.
- Static variables are very useful when we need to write functions that retain values between functions.
- We may use global variables to achieve the same purpose. However, **static variables** are preferable as they are **local variables** to the functions, and the shortcomings of global variables can be avoided.

31

Static Variables

1. A **static** variable may be defined inside or outside a function's body. The duration of a static variable is fixed.
2. Static variables are created at the start of the program and are destroyed only at the end of program execution.
3. We can define static variables **inside** a function's body by changing an automatic variable using the keyword **static**.
4. If a static variable is defined and initialized, it is then initialized once when the storage is allocated. If a static variable is defined, but not initialized, it will be initialized to zero by the compiler.
5. The initialization is done when the storage is allocated. If the static variable is defined inside a function's body, then the variable is only visible by the block containing the variable.
6. Static variables are very useful when we need to write functions that retain values between functions.
7. We may use global variables to achieve the same purpose. However, static variables are preferable as they are local variables to the functions, and the shortcomings of global variables can be avoided.

Static Variables

```
#include <stdio.h>
void function();
int main()
{
   int i;
   for (i=0; i<3; i++)      // calling the fn three times
       function();
   return 0;
}
void function()
{
   static int static_var = 0;    /* static variable */
   int local_var = 0;            /* local variable */
   ++static_var;
   ++local_var;
   printf("Static variable: %d\n", static_var);
   printf("Local variable: %d\n", local_var);
}
```

Output

```
Static variable: 1
Local variable: 1
Static variable: 2
Local variable: 1
Static variable: 3
Local variable: 1
```

Note:

Local variable – the variable disappears after each function execution.

Static variable (like global) – the variable stays until the end of program execution.

32

Static Variables

1. In the example program, the static variable **static_var** is declared and initialized only once when storage is allocated at the start of the program. The value of the static variable is retained for different calls to **function()**. The value stored in the static variable will remain until the end of program execution. This is different from the **local_var** variable as it is created and initialized every time when **function()** is called. However, since the static variable **static_var** is declared inside **function()**, it is only visible inside **function()**.

Functions

– Function Definition
– Function Prototypes
– Function Flow
– Parameter Passing: Call by Value
– Storage Scope of Variables
– **Functional Decomposition**

33

Functions

1. Here, we discuss functional decomposition.

Functional Decomposition

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

int main()
{
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
} /* end. line 2000 */
```

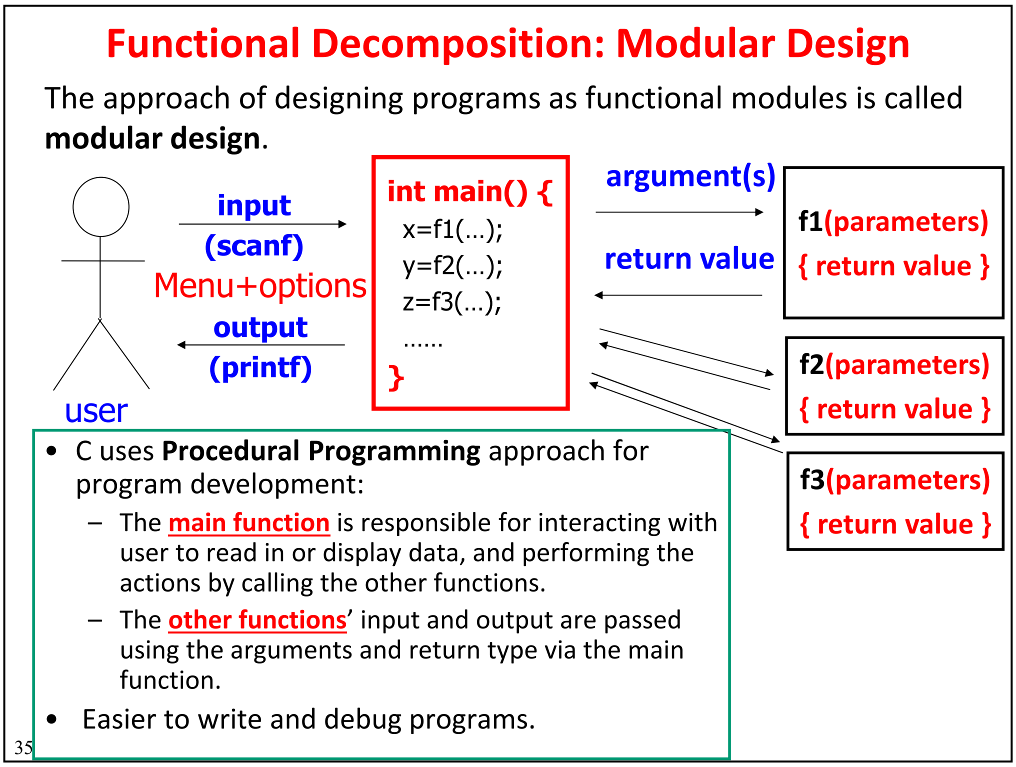
```
#include <stdio.h>
#define ...

int main()
{
.....
} /* line 20 */

float f1(float h)
{
.....
} /* line 55*/
......
void f18()
{
......
} /* line 1560 */
```

- The **main()** function contains about 2000 lines of code which is difficult to read and debug.
- Functional decomposition refers to the top-down stepwise refinement technique that uses the divide-and-conquer strategy to produce smaller functions that are easier to understand. Smaller functions promote software reusability.

34

Functional Decomposition

1. In the original program, the **main()** function contains about 2000 lines of code which is difficult to read and debug.
2. Functional decomposition basically means the top-down stepwise refinement technique that uses the divide-and-conquer strategy. It starts with the high level description of the program and decomposes the program into successively smaller components until we arrive at a set of suitably sized functions (or algorithms). We design the code for the individual functions using stepwise refinement. At each level of refinement, we are only concerned with what the lower level functions will do.
3. Functional decomposition produces smaller functions that are easier to understand. Smaller functions promote software reusability. In general, functions should be small, so that they can be developed and tested separately. They should also be independent of each other.
4. In the example program, it is decomposed into a number of smaller functions. The **main()** function will start program execution and call other functions to perform different required operations.



Functional Decomposition: Modular Design

1. Using the functional decomposition and top-down stepwise refinement technique, a problem is broken up into a number of smaller subproblems or functions. We then develop the algorithms for the functions. These functions can then be implemented using a programming language such as C. These functions are also called **modules**. This approach of designing programs as functional modules is called **modular design**. The functions or modules should be small and self-contained, so that they can be developed and tested separately. They should also be independent of each other.
2. There are a number of advantages for modular design. Modular programs are easier to write and debug, since they can be developed and tested separately. Another advantage is that modular programs can be developed by different programmers as each programmer can work on a single module of the program independently. Moreover, a library of modules can be developed which can then be reused in other programs that require the same implementation. This can reduce program development time and enhances program reliability. Therefore, modular design can simplify program development significantly.
3. When writing C programs, we use procedural programming technique, which is different from object-oriented programming paradigm used in Python, Java and C++.
4. In procedural programming, a typical structure of a program consists of the

main function and other functions for solving a problem. Generally, if the functions are still quite complex, then they can be divided further into smaller functions. And each function should not be longer than a page.

Thank You!

36

Thank You

1. Thanks for watching the lecture video.

4
Pointers

1

Pointers

1. In this lecture, we discuss Pointers in C.

Why Learning Pointers?

- Pointer is a very powerful tool for the design of C programs. A pointer is a variable that holds the value of the **address** or **memory location** of another data object.
- In C, pointers can be used in many ways. These include the passing of variable's address to functions to support call by reference, and the use of pointers for the processing of arrays and strings.
- In this lecture, we discuss the concepts of pointers including address operator, pointer variables and call by reference.

2

Why Learning Pointers?

1. Pointer is a very powerful tool for the design of C programs. A pointer is a variable that holds the value of the address or memory location of another data object.
2. In C, pointers can be used in many ways. These include the passing of variable's address to functions to support call by reference, and the use of pointers for the processing of arrays and strings.
3. In this lecture, we discuss the concepts of pointers including address operator, pointer variables and call by reference.

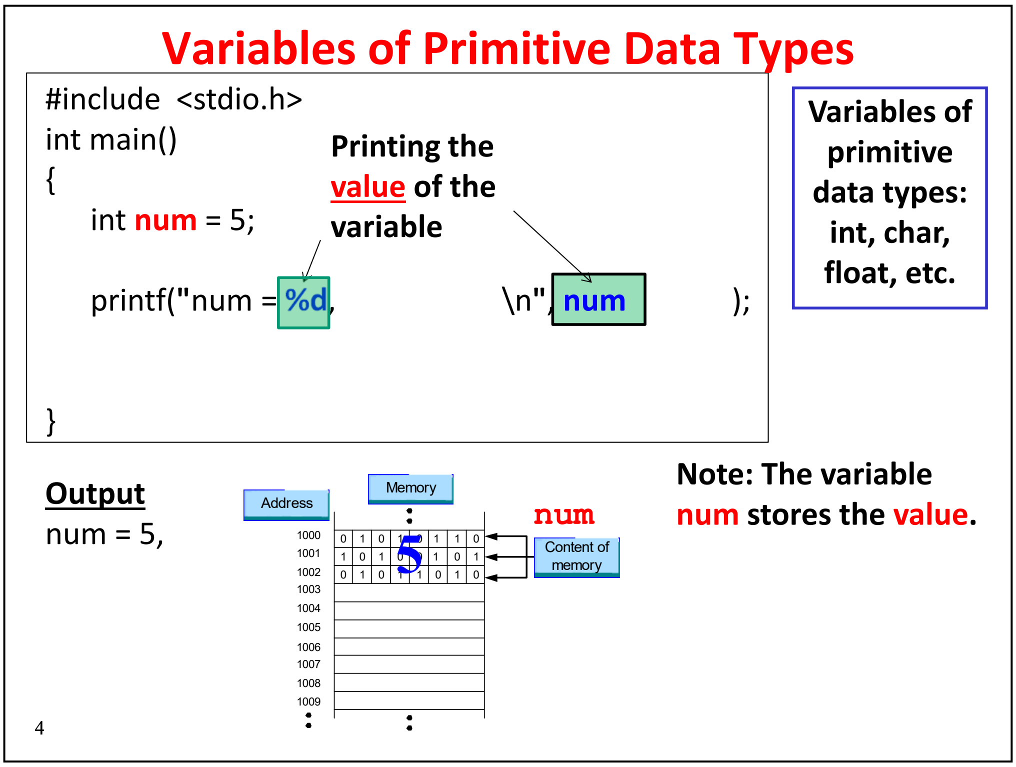
Pointers

– **Primitive Data Types, Variables and Address Operator**
– Pointer Variables
– Call by Reference

3

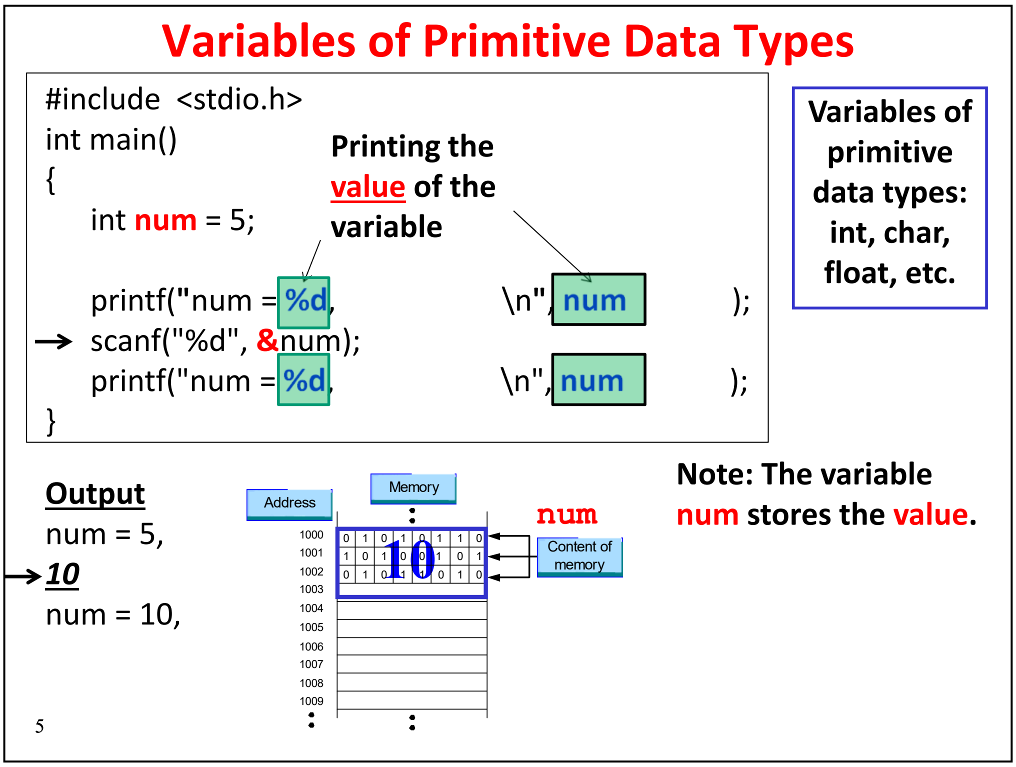
Pointers

1. We start by discussing primitive data types, variables and address operator.



Variables of Primitive Data Types

1. A computer's memory is used to store data objects such as variables and arrays in C. Each memory location has an address that can hold one byte of information. They are organized sequentially and the addresses range from 0 to the maximum size of the memory.
2. When a variable is declared with a certain data type, the corresponding memory location will be allocated for the variable to hold the data of that type.
3. Note that variables of primitive data types such as **int**, **char**, **float**, **double**, etc. are used to store the actual data.
4. For example, in the program, the variable **num** is declared as an **int**. When the variable **num** is initialized with the value 5, the memory location of the variable is used to store the actual value of 5. When the variable **num** is printed with the **printf()** statement, the value 5 will then be printed on the screen.



Variables of Primitive Data Types

1. When the variable **num** is updated (for example, using **scanf()**) to the value of 10, the memory location of the variable is also updated to store the value of 10.
2. When the variable **num** is printed with the **printf()** statement, the value 10 will then be printed on the screen.

Address Operator (&)

```
#include <stdio.h>
int main()
{
    int num = 5;

    printf("num = %d, &num = %p\n", num, &num);
→   scanf("%d", &num);
    printf("num = %d, &num = %p\n", num, &num);
}
```

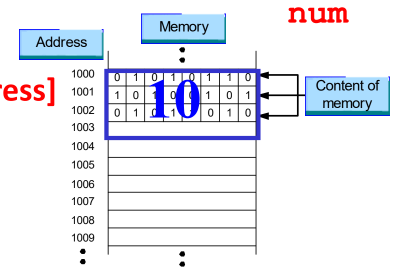
Printing the memory address of the variable

Output

num = 5, &num = 1000 [address]

→ *10*

num = 10, &num = 1000



6

Address Operator

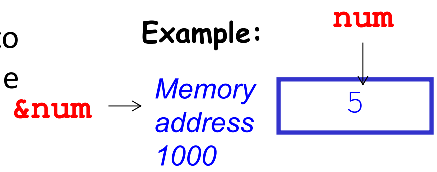
1. The address of a variable can be obtained by the *address operator* (**&**). In the program, we can print the address of the variable **num** (i.e. **&num**). To do this, we need to use **%p** in the conversion specifier in the control string of the **printf()** statement: **printf("num = %d, &num = %p\n", num, &num);**
2. In the **printf()** statement, it prints two values on the screen. The first one is the value 5 that is the initialized value stored at the memory location of the variable **num**. The other is the memory address at which the value 5 is stored. The address of this memory location is 1000. However, as the memory location is assigned by the computer, it may be different every time the same program is run. We use **&num** to find the value of the address.
3. After executing the **scanf()** statement **scanf("%d", &num);** which reads in a value of 10 from the standard input, the value is then stored into the address location of the variable **num**.
4. When we perform the **printf()** statement again, the value stored in **num** has been updated to 10 due to user input. However, the memory address of the variable **num** remains the same throughout the execution of the program.

Primitive Variables: Key Ideas

```
int num=5;
```

(1) num

- It is a variable of data type int and 4 bytes of memory are allocated.
- Its memory location is used to store the integer value of the variable.



(2) &num

- It refers to the memory address of the variable which is used to store the int value of the variable.

Note: You may also print the address of the variable using the printf() statement.

7

Primitive Variables: Key Ideas

1. There are two important ideas related to primitive variables in the above example:
   a) First, the primitive variable (**num**) stores a variable value of data type **int**.
   b) Second, after applying the address operator to the variable **num** (i.e., **&num**), it refers to the memory address of the variable. The memory location is used to store the variable value.

Pointers

– Primitive Data Types, Variables and Address Operator
– **Pointer Variables**
– Call by Reference

8

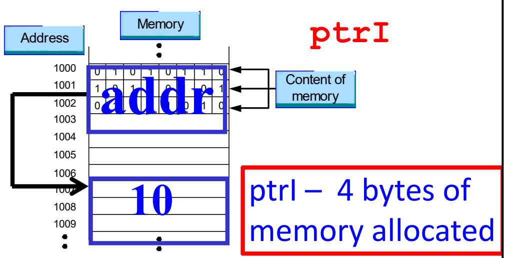
Pointers

1. Here, we discuss pointer variables.

Pointer Variables: Declaration

- **Pointer variable** – different from the primitive variable **num** (variable of primitive data type such as int, float, char) declared earlier, it stores the **address** of memory location of a data object.
- A **pointer variable** is declared by, for example:

```
      int *ptrl;
or    int * ptrl;
or    int* ptrl;
```



- **ptrl** is a pointer variable. It does **not** store the **value** of the variable. It stores the **address** of the memory which is used for storing an Int value.

9

Pointer Variables: Declaration

1. Different from the variables of primitive data types which store the values directly, we may also have variables which store the addresses of memory locations of some data objects. These variables are called **pointer variables**.
2. A **pointer variable** is declared by: **data_type *ptr_name;** where **data_type** can be any C data type such as **char**, **int**, **float** or **double**. **ptr_name** is the name of the pointer variable.
3. The **data_type** is used to indicate the type of data that the variable is pointed to. An asterisk (*****) is used to indicate that the variable is a pointer variable.
4. For example, the statement **int *ptrl;** declares a pointer variable **ptrl** that points to the address of a memory location that is used to store an **int**.
5. Note that the value of a pointer variable is an **address**, which is different from other variables of primitive data types that store the **data** directly. If we want to retrieve the **actual value**, we will need to use **indirection operator** (*), i.e. ***ptrl**.

Pointer Variables: Analogy

ptrI [1006] → *Memory address 1006* [Integer Value Stored (4 bytes)]

- Analogy:

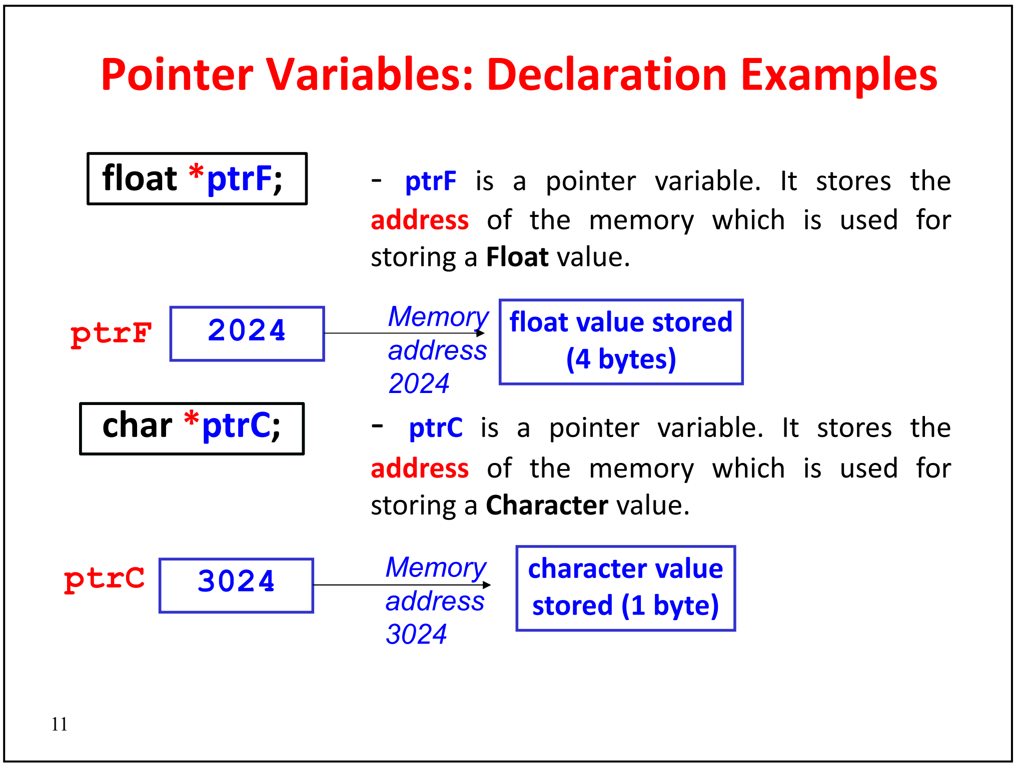
  (1) Address on envelope → your home

  (2) Bank account → your saving/money in the bank

10

<u>Pointer Variables: Analogy</u>

1. Pointer variable is similar to the address written on an envelope which stores the home address, and the actual place can be referred to by the address.
2. It is also similar to a bank account book, which contains the saving information and the bank location that stores the money. The money can be referred to via the bank account.



Pointer Variables: Declaration Examples

1. The statement **float *ptrF;** declares a pointer variable **ptrF** that points to the address of a memory location that is used to store a **float**.
2. Similarly, the statement **char *ptrC;** declares a pointer variable **ptrC** that points to the address of a memory location that is used to store a **char**.
3. When a pointer is declared without initialization, **4 bytes** of memory are allocated to the pointer variable. However, no data or address is stored in the memory.

Pointer Variables: Key Ideas

int * ptrl;

(1) ptrl

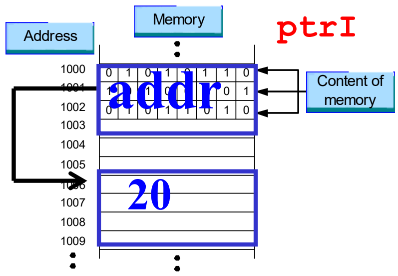
- Pointer variable (4 bytes of memory).
- The value of the variable (i.e. stored in the variable) is an **address**.

(2) *ptrl

- Contains the **content** **(or value)** of the memory location pointed to by the pointer variable ptrl.
- The value is referred to by using the **indirection operator** **(*), i.e.** *ptrl.
- For example: we can assign

  ***ptrl = 20;**

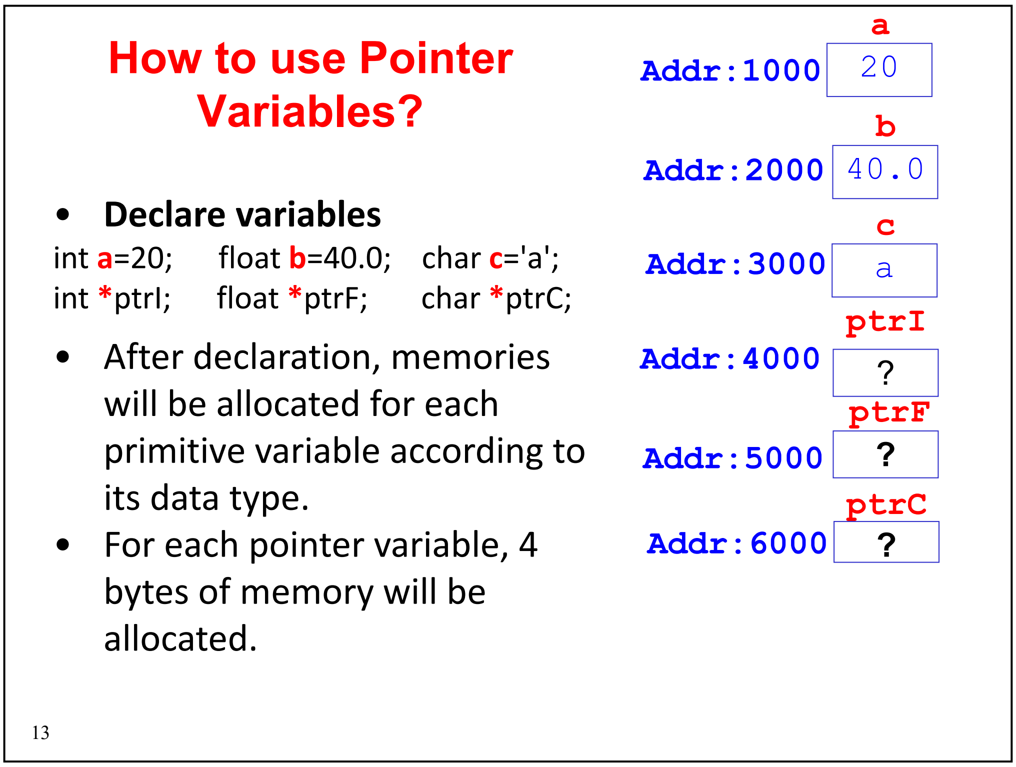
  => the value 20 is stored at the address pointed to by **ptrl**.



12

Pointer Variables: Key Ideas

1. There are two important concepts related to pointers:
   a) The pointer variable (**ptr**) is used to store an address which refers to the location that stores the actual data of the specified data type.
   b) The indirection operator (***ptr**) can be used to retrieve the actual value pointed to by the pointer variable.
2. For example, after declaring the pointer variable, the following assignment statement, ***ptrl = 20;** will update the memory location pointed to by the pointer variable to 20.
3. As such, we can retrieve the actual integer value of 20 referred to by the pointer variable **ptrl** using indirection operator ***ptrl** which will give the value of 20.



<u>How to use Pointer Variables?</u>

1. Declare and initialize the variables and pointer variables as follows:

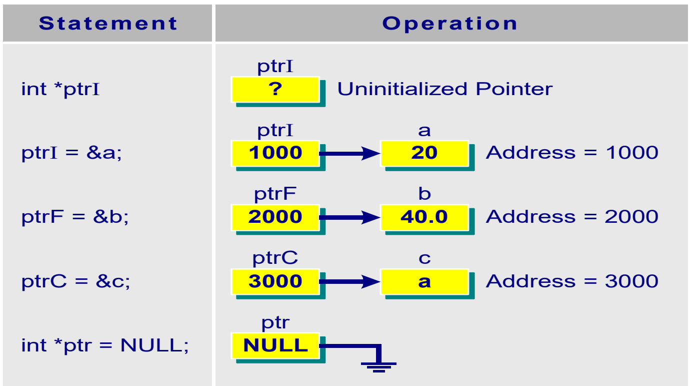
    int a=20; float b=40.0; char c='a';

    int *ptrI; float *ptrF; char *ptrC;

3. Apart from storing an address value in a pointer variable, we can also store the **NULL** value which is defined in <**stdio.h**> in a pointer variable. The pointer is then called a **NULL** pointer. **NULL** is represented in the computer as a series of 0 bits. It refers to the memory location **0**. It is common to initialize a pointer to **NULL** in order to avoid it pointing to a random memory location: **int *ptr = NULL;**

How to use Pointer Variables? (Cont'd.)

int a=20; float b=40.0; char c='a';
int *ptrI; float *ptrF; char *ptrC;
ptrI = &a; => ***ptrI == 20 [same as variable a]**
ptrF = &b; => *ptrF == 40.0 [same as b]
ptrC = &c; => *ptrC == 'a' [same as c]



***ptrI and a – now refer to the same memory content**
Similarly,
***ptrF and b==40.0**
***ptrC and c ==‘a’**

14

How to use Pointer Variables?

1. We can assign variable address to pointer variable as follows:

   ptrI=&a;

   ptrF=&b;

   ptrC=&c;

2. The value of a pointer variable is an address. The pointer variables then point to the memory locations that are used to store the values. The statement **ptrI = &a;** is used to assign the address of the memory location of **a** to the pointer variable **ptrI**. Similarly, we can write the other assignment statements as **ptrF = &b;** and **ptrC = &c;**

3. After a pointer variable is assigned to point to a data object or variable, we can access the value stored in the variable using **indirection operator** (*****). If the pointer variable is defined as **ptr**, we use the expression ***ptr** to dereference the pointer to obtain the value stored at the address pointed at by the pointer **ptr**.

4. After the assignment operations, we will have:
   - **ptrI** stores the memory address of the variable **a**;
   - **ptrF** stores the memory address of the variable **b**;
   - **ptrC** stores the memory address of the variable **c**.

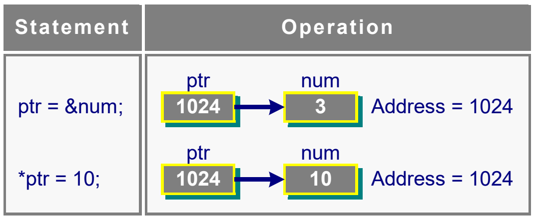
5. It implies that:

- ***ptrI** and **a** will have the same value of 20;
- ***ptrF** and **b** will have the same value of 40.0;
- ***ptrC** and **c** will have the same value of ‘a’.

2. It means that after the assignment **ptrI = &a;** we will be able to retrieve the value of the variable **a** through either (1) the variable **a** directly; or (2) dereferencing the pointer variable ***ptrI**. Therefore, we can write programs more flexibly by using pointer variable.

Pointer Variables – Example 1

```
#include <stdio.h>
int main()
{
    int num = 3;  // integer var
    int * ptr;    // pointer var

    ptr = &num;   // assignment

    // Question: what will be ptr, *ptr, num?
    printf("num = %d, &num = %p\n", num, &num);

    printf("ptr = %p, *ptr = %d\n", ptr, *ptr);

}
```



Output
num = 3, &num = 1024
ptr = 1024, *ptr = 3

Note: num and *ptr have the same value

15

Pointer Variables: Example 1

1. In the program, the primitive type variable **num** stores the value of 3, and the address of the memory location of the variable **num** is 1024.
2. A pointer variable **ptr** is declared to point to a variable of type **int**. The statement **ptr = #** assigns the address of the variable **num** to the pointer variable **ptr**.
3. The first printf statement **printf("num = %d, &num = %p\n", num, &num);** prints the value of **num**, and the address of the variable **num**.
4. Therefore, it prints 3 for the value of **num**, and 1024 for the address of **num**.
5. The second printf statement **printf("ptr = %p, *ptr = %d\n", ptr, *ptr);** prints the value (or address value) stored in the pointer variable **ptr**, and the content of the memory location pointed to by the pointer variable.
6. The value stored in the pointer variable (i.e. **ptr**) is 1024, and the value referred to by the pointer variable (i.e. ***ptr**) is 3. These values are printed on the screen.
7. As can be seen, the values for **ptr** and **&num** are the same (i.e. 1024). And the values for the variable **num** and the dereferencing of the pointer variable ***ptr** are the same (i.e. 3).

Pointer Variables – Example 1 (Cont'd.)

```
#include  <stdio.h>
int main()
{
   int num = 3; // integer var
   int *ptr;      // pointer var

   ptr = &num;

   printf("num = %d, &num = %p\n", num, &num);

   printf("ptr = %p, *ptr = %d\n", ptr, *ptr);

   *ptr = 10;
   // What will be the values: *ptr, num, &num?
   printf("num = %d, &num = %p\n", num, &num);
   return 0;
}
```

Statement	Operation
ptr = #	ptr 1024 → num 3 Address = 1024
*ptr = 10;	ptr 1024 → num 10 Address = 1024

Output
num = 3, &num = 1024
ptr = 1024, *ptr = 3
num = 10, **&num = 1024**
[*ptr = 10]

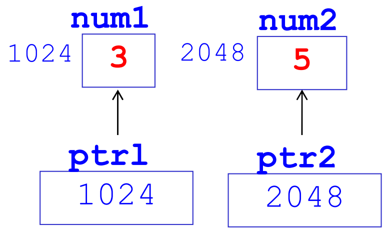
16

Pointer Variables: Example 1

1. The statement ***ptr = 10;** assigns the value 10 to the variable **num**.
2. Since **ptr** stores the address of **num**, the change of value at this memory location has the same effect as changing the value stored at **num**.
3. Therefore, the value stored at **num** is 10. And ***ptr** is also 10. There is no change to the address of the memory location of **num** (i.e. 1024).

Pointer Variables – Example 2

```c
/* Example to show the use of pointers */
 #include  <stdio.h>
int  main()
{
    int num1 = 3, num2 = 5;  // integer variables
    int *ptr1, *ptr2;        // pointer variables

    ptr1 = &num1;  /* put the address of num1 into ptr1 */
    // What are the values for num1, *ptr1?
    printf("num1 = %d, *ptr1 = %d\n", num1, *ptr1);

    ptr2 = &num2;  /* put the address of num2 into ptr2 */
    // What are the values for num2, *ptr2?
    printf("num2 = %d, *ptr2 = %d\n", num2, *ptr2);
```



Output
num1 = 3, *ptr1 = 3
num2 = 5, *ptr2 = 5

17

Pointer Variables: Example 2

1. In the program, the following statements assign the address of **num1** into **ptr1**, and the address of **num2** into **ptr2**:

   ptr1 = &num1;

   ptr2 = &num2;

2. Therefore, we have

   num1 = 3 and ***ptr1** = 3; and

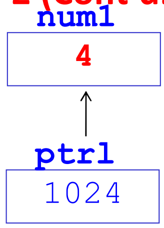
   num2 = 5 and ***ptr2** = 5.

   These values are printed on the screen.

Pointer Variables – Example 2 (Cont'd.)

/* increment by 1 the content of the memory location pointed by ptr1 */

```
(*ptr1)++;
```



// What are the values for num1, *ptr1?

printf("num1 = %d, *ptr1 = %d\n", num1, ***ptr1**);

Output

num1 = 4, *ptr1 = 4

18

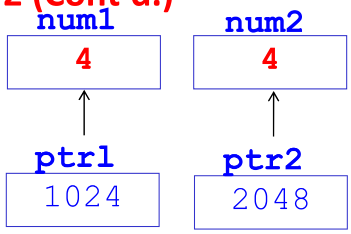
Pointer Variables: Example 2

1. The statement **(*ptr1)++;** increments 1 to the content of the memory location pointed to by **ptr1**.
2. Therefore, we have ***ptr1**= 4, and **num1** =4. The values are then printed on the screen.

Pointer Variables – Example 2 (Cont'd.)

/* copy the content of the location pointed by ptr1 into the location pointed by ptr2*/

```
*ptr2 = *ptr1;
```



// What are the values for num2, *ptr2?

```
printf("num2 = %d,*ptr2 = %d\n",num2, *ptr2);
```

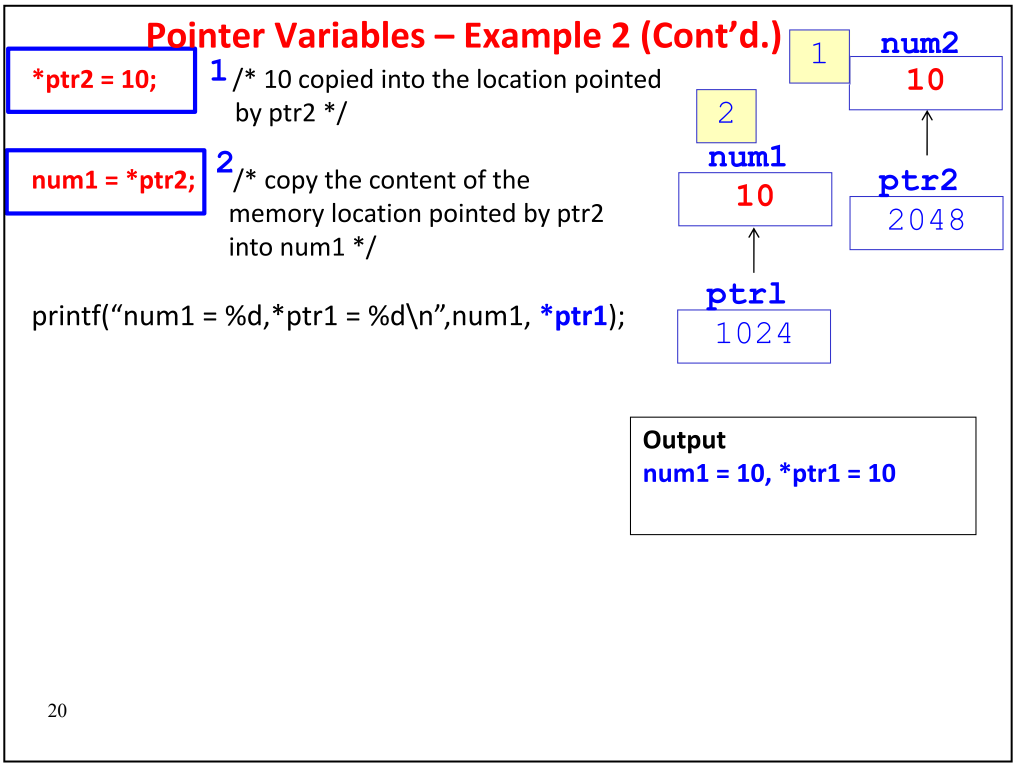
Output

num2 = 4, *ptr2 = 4

19

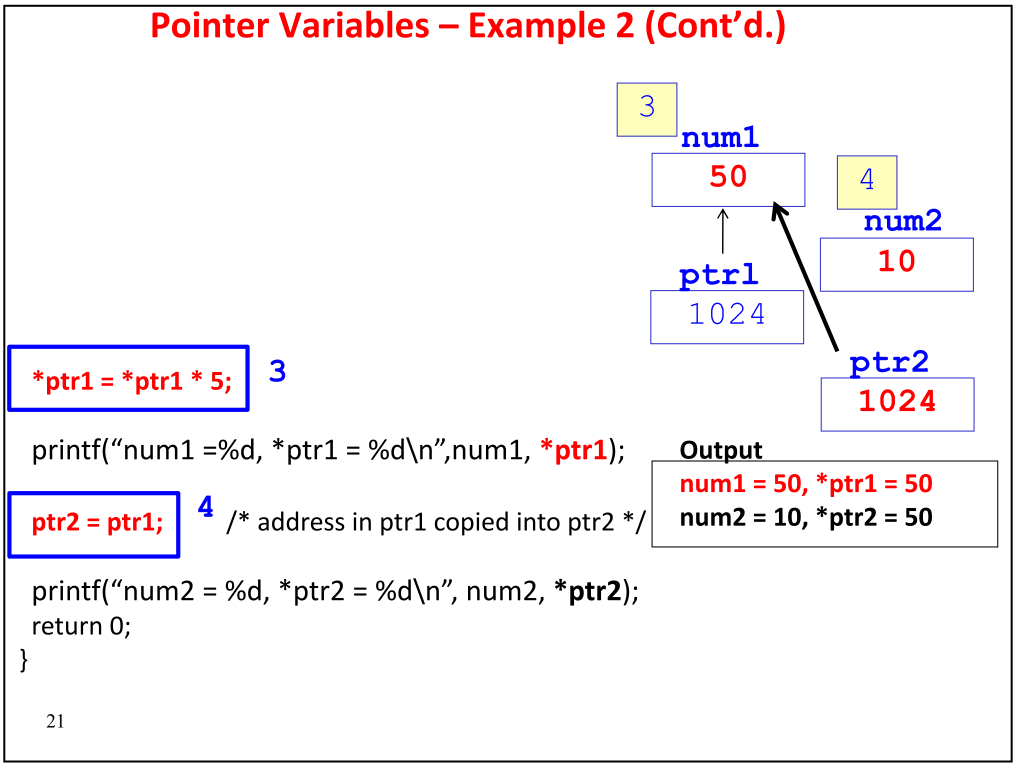
Pointer Variables: Example 2

1. The statement ***ptr2 = *ptr1;** copies the content of the location pointed to by **ptr1** into the location pointed to by **ptr2**.
2. Since ***ptr1** = 4, we have ***ptr2** = 4 and **num2** = 4. The values are then printed on the screen.



Pointer Variables: Example 2

1. The statement ***ptr2** = 10**;** assigns the value 10 to the content of the memory location pointed to by **ptr2**.
2. Therefore, ***ptr2** = 10, and **num2** = 10.
3. The statement **num1 = *ptr2;** copies the content of the memory location pointed to by **ptr2** into **num1**.
4. Since ***ptr2** = 10, **num1** = 10, we have **num1** = 10, and ***ptr1** = 10. These values are then printed on the screen.



<u>**Pointer Variables: Example 2**</u>

1. In the statement ***ptr1 = *ptr1 * 5;** since ***ptr1** = 10, we have ***ptr1***5 = 50.
2. The new value 50 is assigned to the content of the memory location pointed to by **ptr1**.
3. Therefore, we have ***ptr1** = 50, and **num1** = 50. These values are then printed on the screen.
4. The last statement **ptr2 = ptr1;** copies the address in **ptr1** into **ptr2**, so that the pointer variable **ptr2** points to the same memory location as **ptr1**. Therefore, we have ***ptr2** = 50.
5. However, the value of **num2** is not changed, we have **num2** = 10. The values of **num2** = 10 and ***ptr2** = 50 are printed on the screen.
6. The concepts of using pointers in the program are summarized as follows:
   - **ptr1**, **ptr2** - They refer to the values (which are memory addresses) stored in the pointer variables **ptr1** and **ptr2**.
   - **&ptr1**, **&ptr2** - They refer to the memory addresses of the variables **ptr1** and **ptr2**.
   - ***ptr1**, ***ptr2** - They refer to the values (which are primitive data) whose memory locations are stored in the memory locations of the pointer variables **ptr1** and **ptr2**.

Using Pointer Variables (within the Same Function): Key Steps

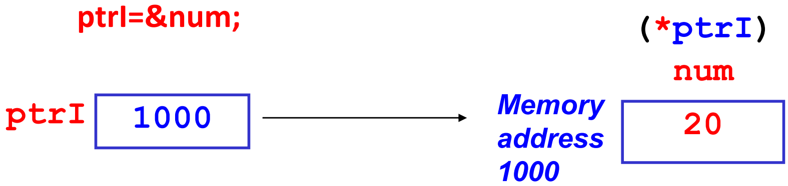
1. Declare variables and pointer variables:

    int num=20;

    int *ptrI;

2. Assign the address of variable to pointer variable:

    ptrI=#



Then you can retrieve the value of the variable num through *ptr as well.

22

<u>Using Pointer Variables: Key Steps</u>

1. There are two key steps on using pointer variables:
    a) First, declare variables and pointer variables, such as

        int num=20;

        int *ptrI;

    b) Second, assign the address of variable to pointer variable, that is

        ptrI=#

Pointers

– Primitive Data Types, Variables and Address Operator

– Pointer Variables

– **Call by Reference**

23

Pointers

1. Here, we discuss the concept of call by reference.

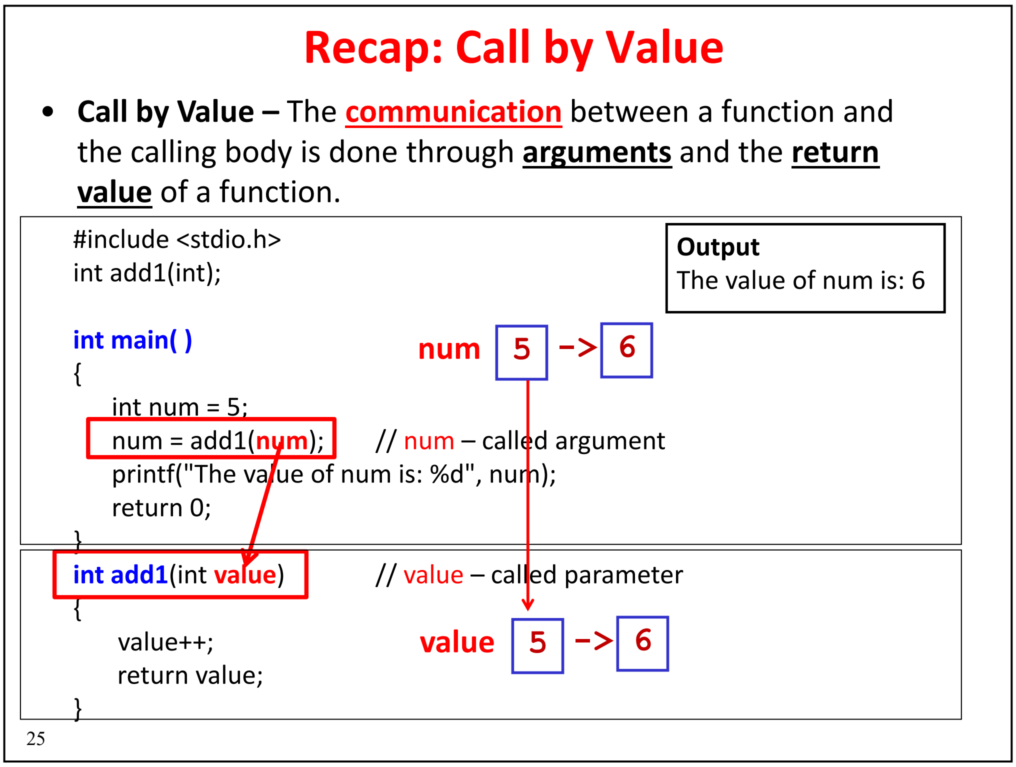
Call by Reference

- Parameter passing between functions has two modes:
  - **call by value** [discussed in the last lecture on Functions]
  - **call by reference** [to be discussed in this lecture]
- **Call by reference**: the parameter in the function holds the address of the argument variable, i.e., the **parameter** is a **pointer variable**. Therefore,
  - In the **function header**'s parameter declaration list, the **parameters** must be prefixed by the **indirection operator ***.
    E.g. void distance(double *x, double *y)
  - In the **function call**, the **arguments** must be **pointers** (or using the address operator as the prefix).
    E.g. distance(&x1, &y1);

24

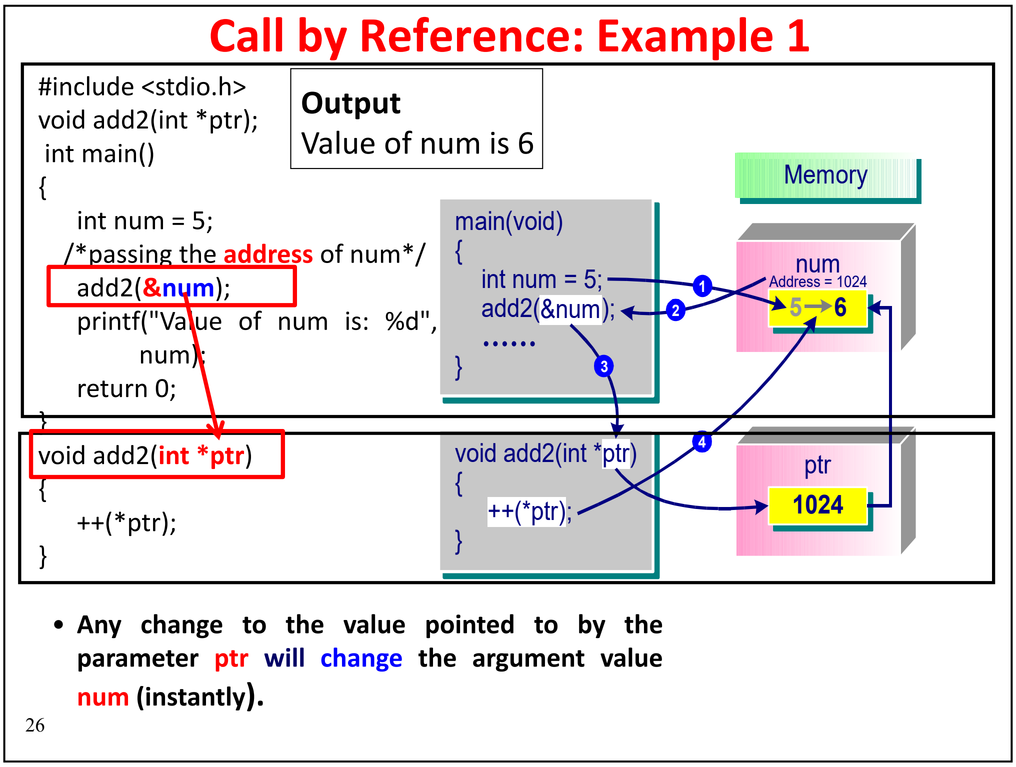
Call by Reference

1. Parameter passing between functions can be done either through call by value or call by reference. Call by value has been discussed in the last chapter on functions.
2. In call by reference, parameters hold the addresses of the arguments, i.e. parameters are **pointers**. Therefore, any changes to the values pointed to by the parameters change the arguments. The arguments must be the addresses of variables that are local to the calling function.
3. In a function call, the **arguments** must be **pointers** (or using address operator as the prefix). For example, **distance(&x1, &y1);**
4. In the function header, the **parameters** in the parameter declaration list must be prefixed by the **indirection operator**. For example, **void distance(double *x, double *y);**



Recap: Call by Value

1. This example illustrated call by value which was discussed in the chapter on Functions.
2. In this example, we pass in the value of the variable **num** as argument from the **main()** function to the parameter **value** of the function **add1()**. The value is then used locally for processing in the function **add1()**.



Call by Reference: Example 1

1. In the program, the variable **num** is initially assigned with a value 5 in **main()**.
2. The address of the variable **num** is then passed as an argument to the function **add2()** (in **Step 2**) and stored in the parameter **ptr** in the function (in **Step 3**).
3. In the function **add2()**, the value of the memory location pointed to by the variable **ptr** (i.e. **num**) is then incremented by 1 (in **Step 4**). It implies that the value stored in the variable **num** becomes 6. When the function ends, the control is then returned to the calling **main()** function. Therefore, when **num** is printed, the value 6 is displayed on the screen.
4. In this example, note that the parameter variable **ptr** in **add2()** is used to store the address of the variable **num** in **main()**. After passing the variable address of **num** into the parameter variable **ptr**, all the operations on **ptr** in function **add2()** will update the content of the variable **num** (which is in **main()**) indirectly.

Call by Reference: Key Steps

1. In the **function definition**, the **parameter** must be prefixed by **indirection operator ***:

   add2( ): void add2(int *ptr) { …}

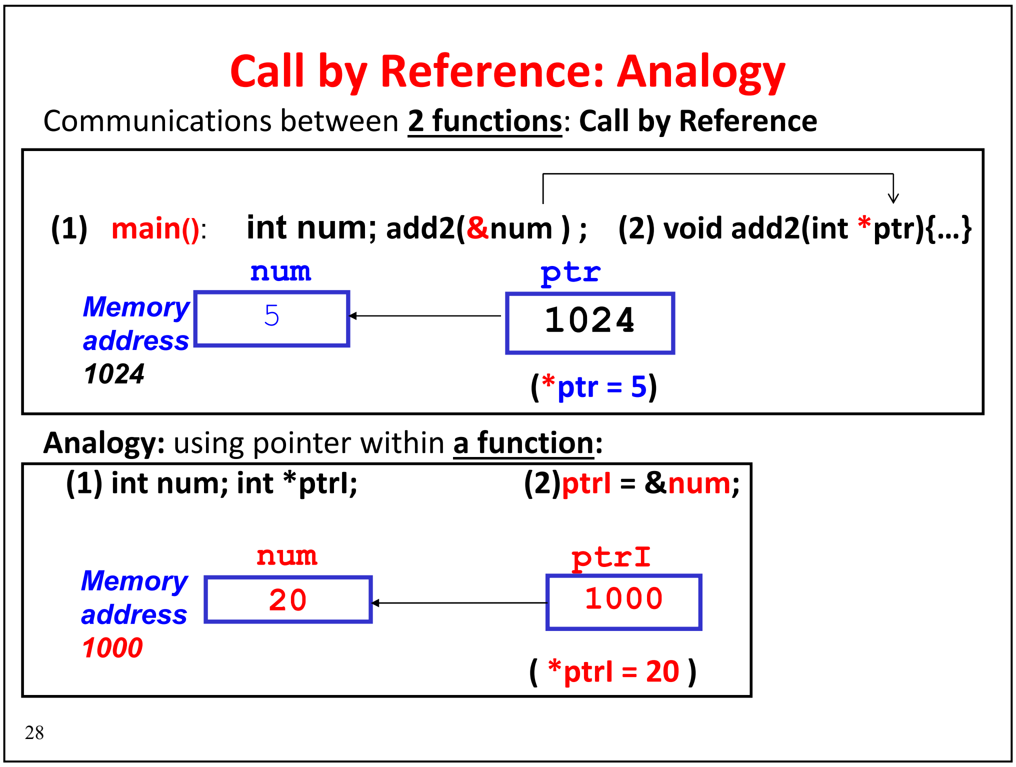
2. In the **calling function**, the **arguments** must be pointers (or using **address** operator as the prefix):

   main( ): int num; add2(&num) ;

27

Call by Reference: Key Steps

1. There are two key steps when using call by reference:
   a) First, in the function, the parameter must be prefixed by the indirection operator: e.g. **void add2(int *ptr);**
   b) Second, in the calling function (e.g. **main()**), the arguments must be pointers (or using address operator as the prefix): e.g. **add2(&num);**



Call by Reference: Analogy

1. Using call by reference via pointer is very similar to that of using pointers within a function.
2. When using pointers within a function:
   a) We first declare the variable and pointer variable: **int num; int *ptrI;**
   b) Then, assign the address of the variable **num** to the pointer variable **ptrI**: **ptrI = #**
3. Therefore, when the variable **num** is updated to 20: that is, **num** is 20**; *ptrI** is also **20;**

Call by Reference – Example 2

```
#include<stdio.h>
void function1 (int a, int *b); void function2 (int c, int *d);
void function3 (int h, int *k);
int main() {
    int x, y;
    x = 5;  y = 5;          address          /* (i) */
    function1(x, &y);                        /* (x) */
    return 0;
}                                 pointer
void function1(int a, int *b)  {             /* (ii) */
    *b = *b + a;                             /* (iii) */
    function2(a, b);                         /* (ix) */
}                                  pointer
void function2(int c, int *d) {              /* (iv) */
    *d = *d * c;                             /* (v) */
    function3(c, d);                         /* (viii) */
}                                 pointer
void function3(int h, int *k) {              /* (vi) */
    *k = *k - h;                             /* (vii) */
}
```

29

Call by Reference: Example 2

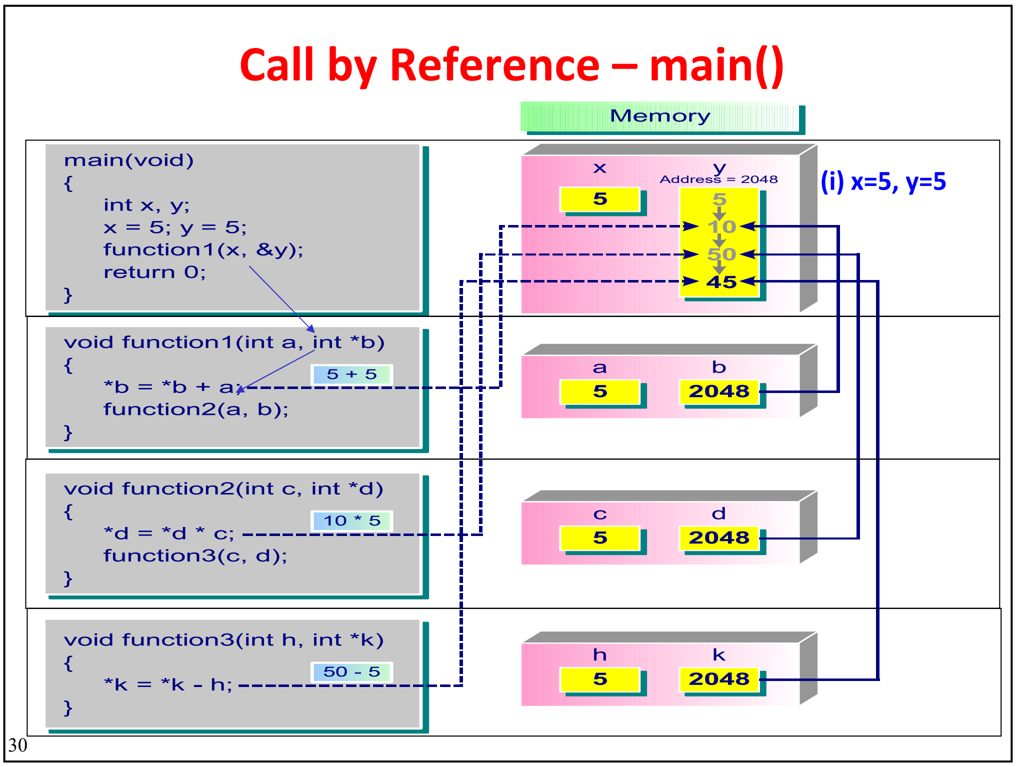
1. In the function definitions, we have defined the following three functions:

    void function1(int a, int *b)

    void function2(int c, int *d)

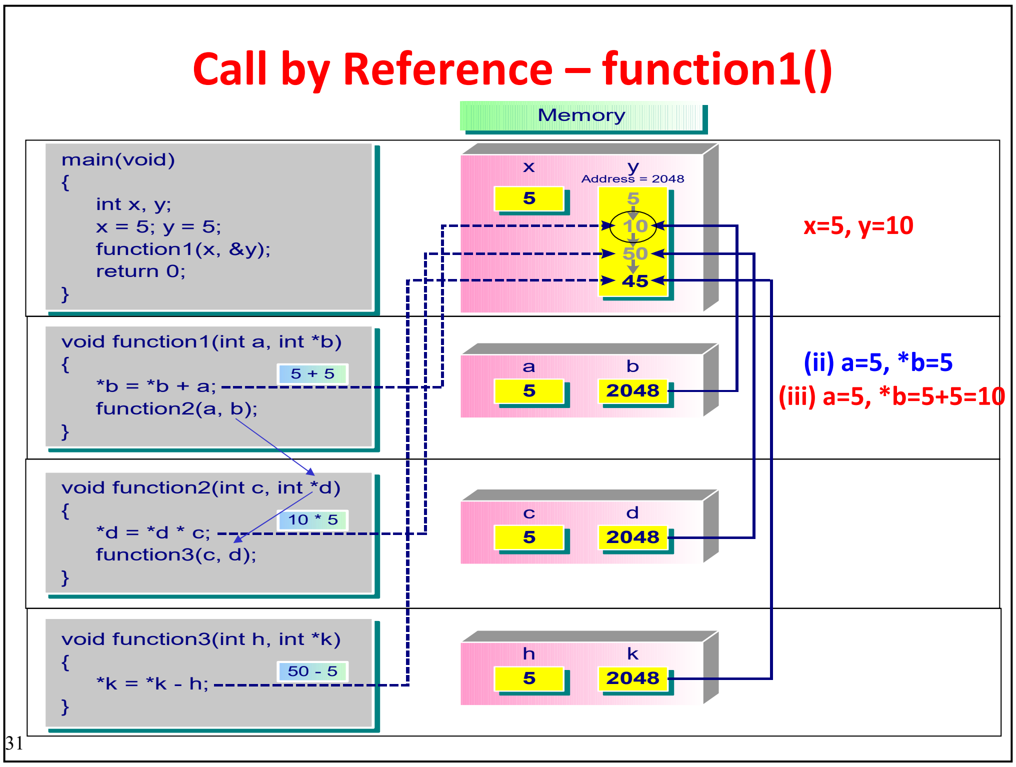
    void function3(int h, int *k)

2. The parameters **a**, **c** and **h** are passed into the functions using call by value, whereas the parameters **b**, **d** and **k** are passed into the functions using call by reference, i.e. addresses are passed to the functions instead of actual values.
3. In fact, the memory contents of these parameters contain the address of the variable **y** in the **main()** function. Any changes to the dereferenced pointers such as ***b**, ***d** and ***k** refer indirectly to the changes to the contents stored in the memory location of the variable **y**.



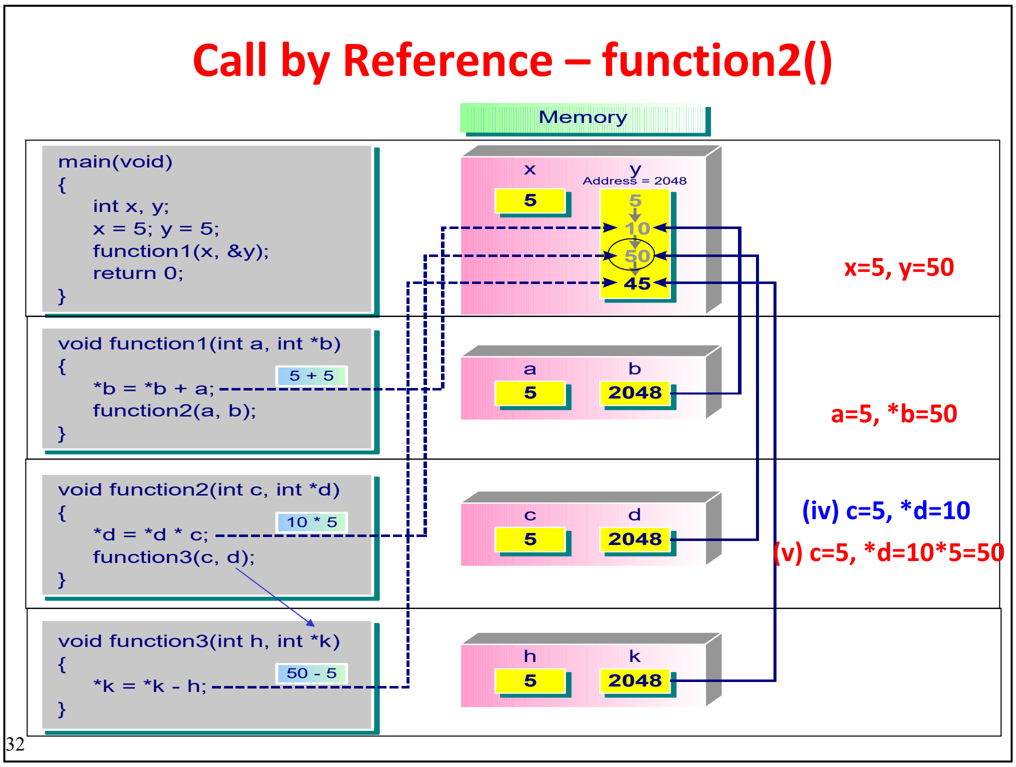
Call by Reference: main()

1. When the program starts execution, in the **main()** function, memory locations are allocated to the variables **x** and **y** accordingly. The two variables are assigned with the value of 5. Therefore, the memory locations of the variables **x** and **y** store the value of 5 directly.
2. The **main()** function then calls **function1()** by passing the value of **x** (i.e. 5) and the address of **y** (i.e. 2048) to the corresponding parameters **a** and **b** respectively.
3. The mode of parameter passing for **a** is call by value, and for **b** is call by reference. As such, the parameter **b** refers to the memory location of **y** in the **main()** function.



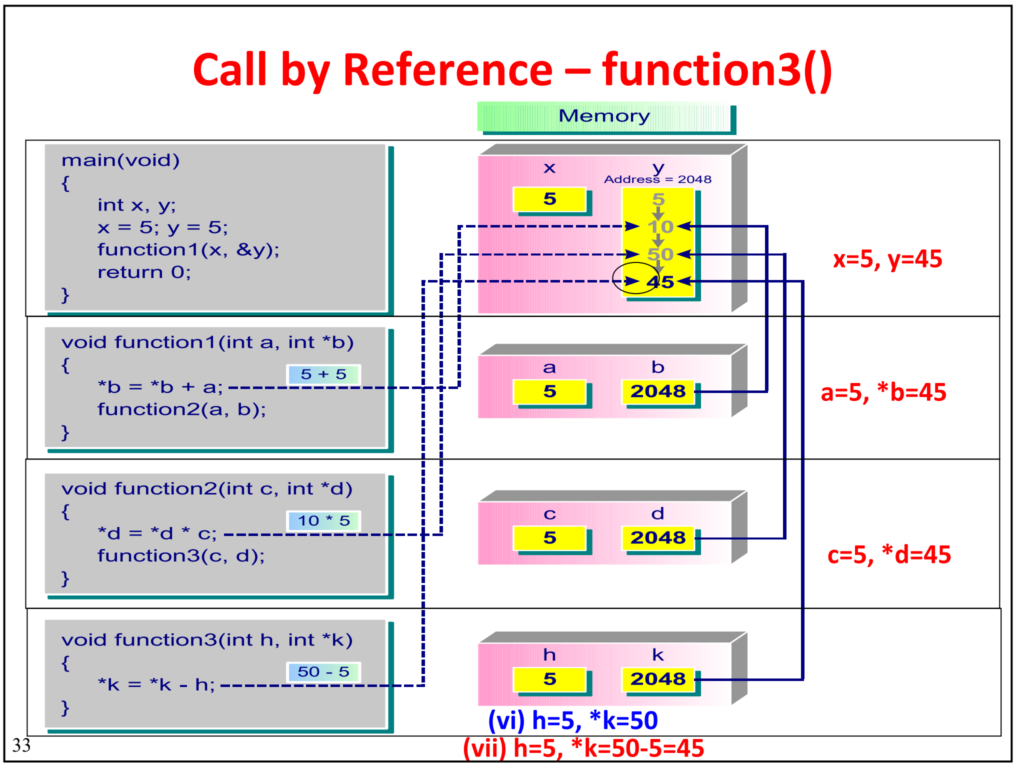
Call by Reference: function1()

1. When **function1()** is executed, the statement ***b = *b + a;** will update the value of ***b** = 5 + 5 = 10; As the pointer variable **b** refers to the location of the variable **y** in the **main()** function, the update in fact is carried out at the memory location of **y**.
2. Therefore, the value of **y** = 10 (in **main()**), and the value of ***b** = 10 (in **function1()**). There is no change in the value of the variable **a** which is 5.
3. After that, **function1()** calls **function2()** by passing in the values of **a** and **b** into the parameters **c** and **d** respectively.



Call by Reference: function2()

1. When **function2()** is executed, the statement ***d = *d * c;** will update the value of ***d** = 10 * 5 = 50; As the pointer variable **d** contains the same address value as **b** in **function1()**, it also refers to the location of the variable **y** in the **main()** function. The update in fact is carried out at the memory location of **y**.
2. Therefore, the value of **y** is also 50 (in **main()**), and the value of ***d** = 50 (in **function2()**). There is no change in the value of the variable **c** which is 5.
3. After that, **function2()** calls **function3()** by passing in the values of **c** and **d**.



Call by Reference: function3()

1. When **function3()** is executed, the statement ***k = *k - h;** will update the value of ***k** = 50 - 5 = 45; As the pointer variable **k** contains the same address value as **d** in **function2()**, it also refers to the location of the variable **y** in the **main()** function. The update in fact is carried out at the memory location of **y**.
2. Therefore, the value of **y** is also 45 (in **main()**), and the value of ***k** = 45 (in **function3()**). There is no change in the value of the variable **h** which is 5.
3. After that, **function3()** finishes the execution and terminates. The control passes to **function2()** for execution and then terminates, which in turn returns to **function1()**, and then terminates and finally returns to the **main()** function.

Call by Reference – Example 2

	x	y	a	*b	c	*d	h	*k	remarks
(i)	5	5	-	-	-	-	-	-	in main
(ii)	5	5	5	5	-	-	-	-	in fn 1
(iii)	5	10	5	10	-	-	-	-	in fn 1
(iv)	5	10	5	10	5	10	-	-	in fn 2
(v)	5	50	5	50	5	50	-	-	in fn 2
(vi)	5	50	5	50	5	50	5	50	in fn 3
(vii)	5	45	5	45	5	45	5	45	in fn 3
(viii)	5	45	5	45	5	45	-	-	return to fn 2
(ix)	5	45	5	45	-	-	-	-	return to fn 1
(x)	5	45	-	-	-	-	-	-	return to main

34

Call by Reference: Example 2

1. The values for each variable and parameter for each function in the program are summarized in the table.
2. Note that the value of the variable **x** in the **main()** function does not change throughout program execution, while the value of variable **y** is changed after each function call.

When to Use Call by Reference

When to use call by reference:

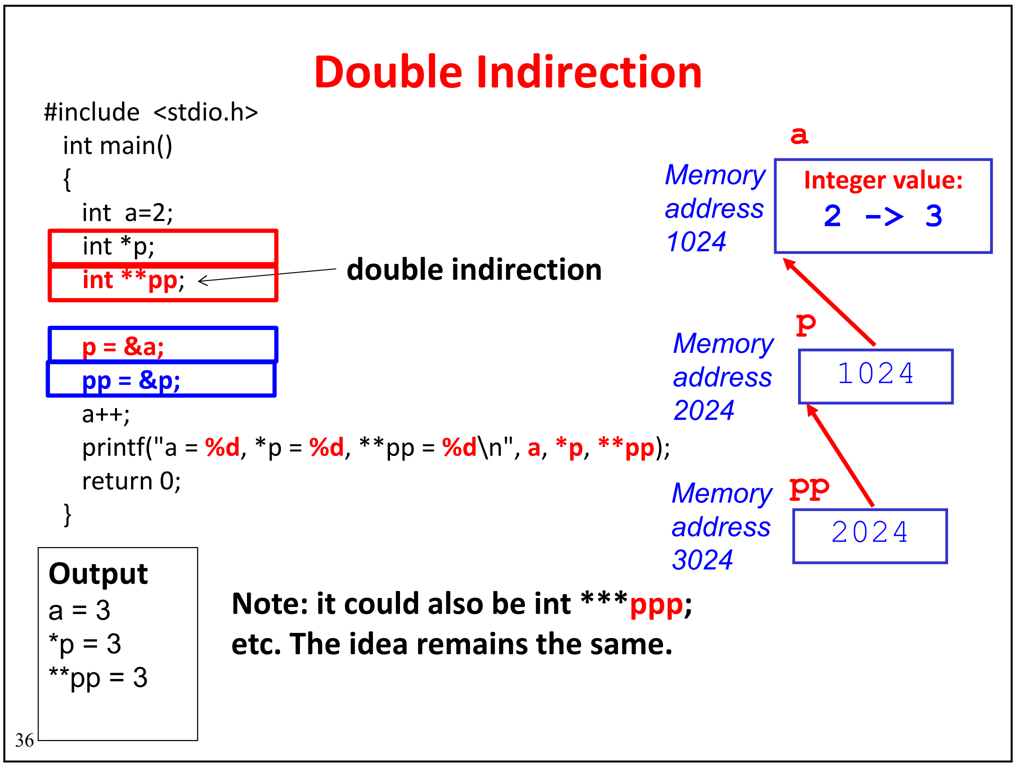
(1) When you need to pass **more than one value** back from a function.

(2) When using call by value will result in a **large piece of information** being copied to the formal parameter, for efficiency reason, for example, passing large arrays or structure records.

35

When to Use Call by Reference

1. Generally, call by reference is used in the following situations:
    a) First, when we need to pass more than one value back from a function.
    b) Second, in the case that when we use call by value, it will result in a large piece of information being copied to the parameter. This could happen when we pass a large array size or structure record.



<u>Double Indirection</u>

1. We have seen examples on using indirection operator. **Double indirection** is also quite commonly used in C programming.
2. In the program, **p** is a pointer variable. A variable can also be declared as **int **pp**; using double indirection. **pp** is also a pointer variable.
3. After the first two assignment statements: **p = &a;** and **pp = &p;** the pointer variable **pp** will be used to store an address of another pointer variable **p** (i.e. it points to another pointer variable), which will in turn store the address of a variable (i.e. **a**) of the corresponding primitive type.
4. Therefore, after the variable declaration and assignment, we will have the output:

   a=2;

   ***p=2;**

   ****p=2;**

Thank You!

37

Thank You

1. Thanks for watching the lecture video.