Introduction to C Programming

Windows Operating System

Our Purpose

- This is *not* a C Programming Course
- You must already know how to program
- You will be familiarized with the basics of the C language
- Will cover simple Windows Programming
- User Interface programming will *not* be covered

C Functions

- All C programming must be part of a C function.
- Example Declaration:

```
void MyFunc(int a,int b)
{
    int c;

    c = a + b;
}
```

Formal Syntax

- A function is declared as follows
- The <Type> is return value type and function characteristics

```
<Type>   <Function Name> ( <Argument List> )
{
      <Local Variable Declarations>

      <Executable Code>
}
```

Types

- The most common types in C are the following:

int	16-bit integer
long	32-bit integer
short	16-bit integer
char	8-bit integer or character
float	32-bit floating point
double	64-bit floating point

Type Declarations

- Type Declarations declare simple variables as well as pointers and arrays
- *int a;* -- defines *a* to be a 16-bit integer.
- *long b,c,d;* -- defines *b*, *c*, and *d* to be 32-bit integers.
- *char *xyz;* -- *xyz* is a pointer to a *char*.
- *int Totals[15]* -- *Totals* is an array of 15 *ints*.

Function Headers

- The type *void* is used to indicate no return value, or no argument list.
- Example: *void Func1(void)*
- Each argument must have a declared type preceding its name
- Example: *int F2(int a, int b, char c)*

Function Bodies

- A function body consists of two parts:
  - Declaration of Local Variables
  - Executable code
- Example:

```
int F2(int a, int b)
{
        int c;

        c = a*a;
        c += b;
        return c;
}
```

Global Variables

- Arguments and Local Variables are accessible only inside the function where they are declared.
- Variable declarations that are placed outside of any function are accessible to all functions, and retain their values for the life of the program.

Globals: An Example

```
int a;  // a global variable

void f1(int b,int c)
{
        int k; // local k

        k = b*b;
        a = k + c;
}
```

```
// a different b and c
void f2(int b,int c)
{
        int k; // a different k

        k = b + 2;
        // the same a as before
        a = k * c;
}
```

Multiple Files

- Complex programs consist of many files.
- To make global variables in one file accessible to another file, use the *extern* keyword

```
// in file x.c
int x;
```

```
// in file y.c
extern int x;
```

Both files use the same copy of *x*.

Header Files

- If you have many extern statements that appear in many different files,
  - Place all externs in one file with a .h extension, such as *externs.h*
  - Place the statement *#include "externs.h"* at the beginning of each file
- This is also a good place for structure definitions and function templates.

Function Templates

- Every function should have a function template.
- The template must physically appear before the first use of the function.
- The template is identical to the function header, but ends in a semicolon, and has no body.

Template Example

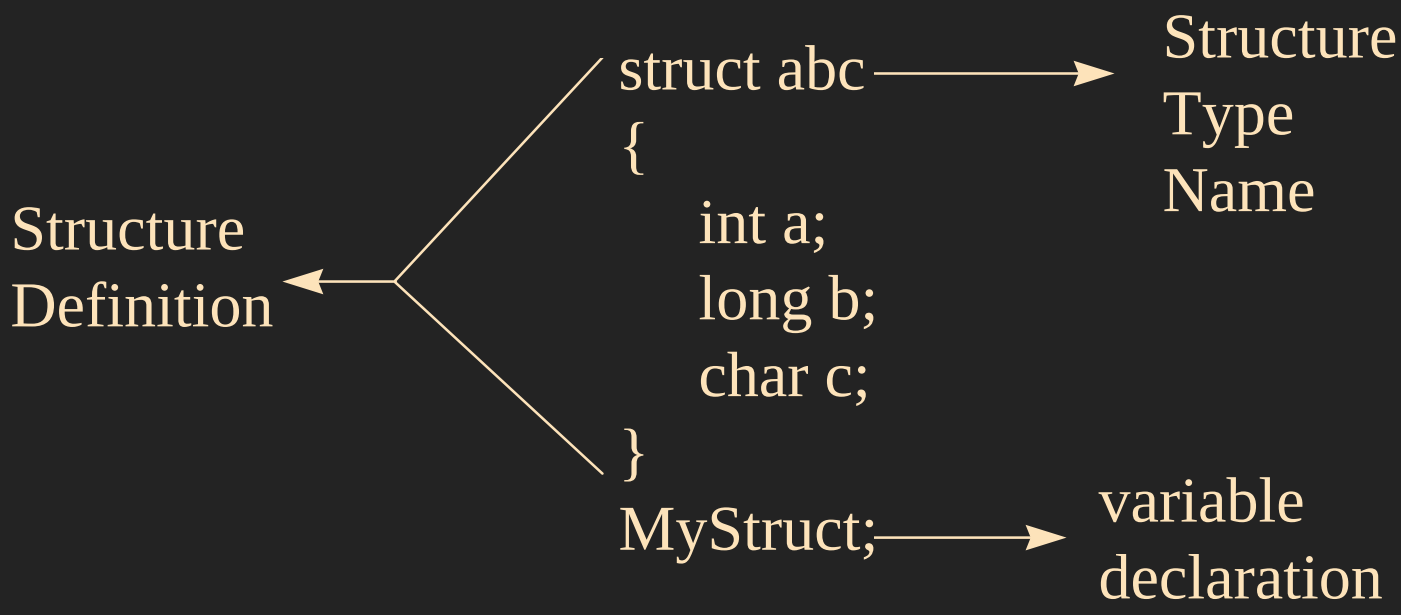
```
float factorial(float x);
float Binomial(float a,float b)
{
        int n,d;

      n = factorial(a);
      d = factorial(b)*factorial(a-b);
      return n/d;
}
float factorial(float x)
{ ...
```

Templates: Notes

- If a function is defined before it is used, the template is not required, but should be used anyway.
- Templates should be placed in a common header file with a *.h* extension.
- Templates make functions defined in one file accessible to functions in other files.

Structures

- Structures are collections of variables of different types, as in the following example.

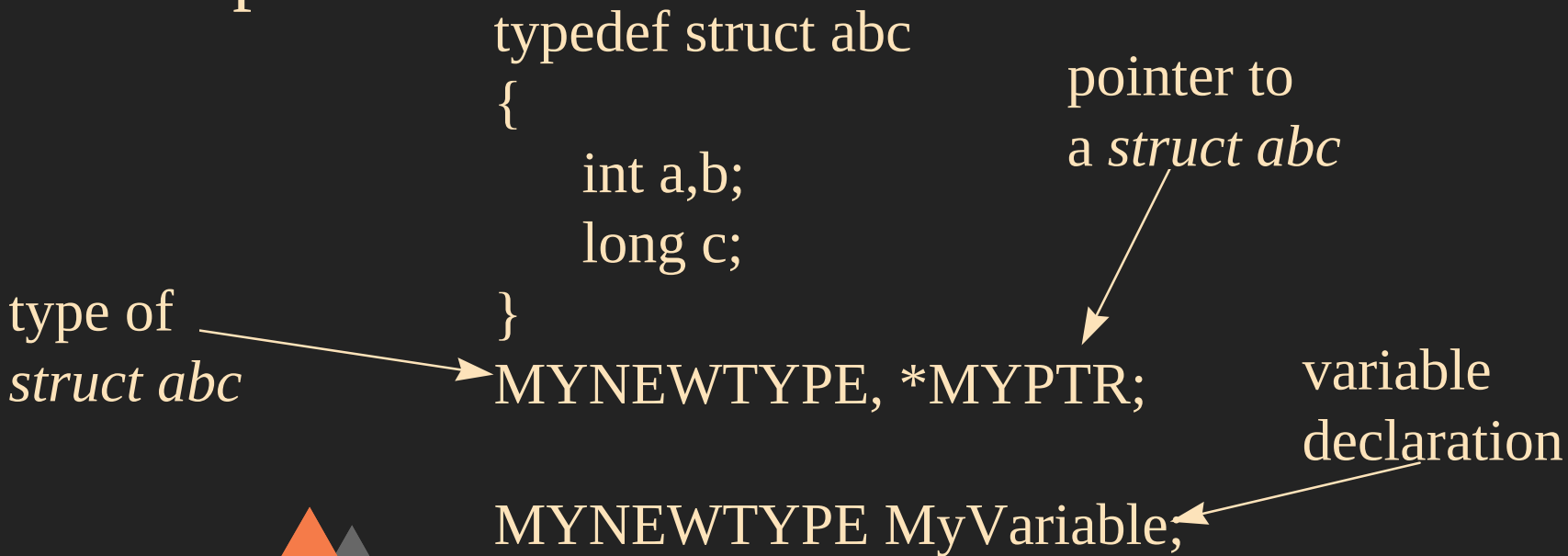


Structures: Notes

- The details of the structure need only be specified once. Subsequent definitions of the same type of structure are done as follows.
- struct abc m,n,q;
- The variable declaration on the previous slide is optional.

New Types

- The most common use of structures is to declare new types as in the following example.



Type Definitions: Notes

- All commonly used structures should be declared as new types.
- By convention, new type names are capitalized.
- New type names can be used anywhere that int, long, … are used.
- Place type definitions in a common *.h* file.

Accessing Variables

- Arrays are accessed as in other languages, but the first index is always zero.
- Example *A[3,4] = B[0];*
- Structure elements are accessed using a period.
- Example: *MyStruct.a = YourStruct.c;*

Accessing Variables II

- Pointers are dereferenced by preceding them with an asterisk.
- Example *MyPtr = *YourPtr;
- Pointers to structures are dereferenced using the -> operator.
- Example MyPtr->a = YourPtr->b;

Arrays and Pointers

- A pointer can point to either a simple variable or an array.
- Accessing an array pointer is identical to accessing an array.
- Example MyArray[3] = MyPtr[5];
- *MyPtr and MyPtr[0] are identical.

Passing Data to Functions

- All arguments are passed by value.
- Arrays are passed by passing the address of the array to the function. Access is identical to accessing the array directly.
- Structures are copied and passed by value.
- All floats are converted to doubles, and converted back inside the function.

Passing by Reference

- Declare the function argument as a *pointer* to the desired type.
- When passing a variable, precede it by the & operator, which extracts the address of the variable.
- Reference the variable through the pointer.
- Use this to avoid copying massive structures to the argument stack.

Assignments

- The equals sign is the assignment operator.
  - *a = b + c;*
- All common arithmetic operators, *except exponentiation*, can be used.
- % is used for remainders, & for bit-wise AND, | for bit-wise OR, ~ for bit-wise NOT, and ^ for bit-wise Exclusive-OR.

Other Operators

- The expression *A* << *k* can be used to shift *A* to the left by *k* bits.
- The operator >> is used for right shift.
- Any Binary Operator can Be combined with the equals sign.

```
a += 5;  // increment a by 5.
b *= 4; // multiply b by 4.
```

Even More Operators

- To increment or decrement a variable by 1, use the ++ and -- operators.
- A++, increments A by 1
- B--, decrements B by 1
- All assignment expressions have a value, so A = B = C = D = 1, sets A, B, C, and D to 1.

Comparisons

- The Comparison Operators are as follows

==	Equals
!=	Not Equals
<	Less Than
>	Greater Than
<=	Less than or equal
>=	Greater than or equal

Comparison Results

- All Comparison Operators Produce a Numeric value: False produces zero, while True produces One.
- Complex Tests can be created using AND, OR and NOT operators.
  - && logical AND
  - || logical OR
  - ! logical NOT

If Statements

- The format of the *if* statement is as follows.

```
if  (<Numeric Expression>)
{
    <True-Body>
}
else
{
     <False-Body>
}
```

If Evaluation

- If the numeric expression is zero, it is considered to be False, otherwise it is considered to be True.
- If the expression is True, the True-Body is executed, otherwise the False-Body is executed.
- The False-Body may be omitted, along with the *else* keyword and the enclosing braces.

While Statements

- The format of the *while* statement is as follows.

```
while (<Numeric Expression>)
{
    <While Body>
}
```

While Execution

- If the Numeric Expression is zero, it is considered to be False, otherwise it is considered to be True.
- The Loop-Body is executed until the Numeric Expression becomes False.
- The loop body will be skipped entirely if the expression is initially false.

For Loops

- In C, the *for* statement is used for most loops. The syntax is as follows.

```
for (<Start-Body> ; <Condition> ; <Continue-Body>)
{
    <For-Body>
}
```

For Execution

- The C for statement is a special case of the while.
- The Start-Body is executed before the loop begins.
- The Condition is tested before executing the Loop-Body.
- The Continue-Body is executed after the Loop-Body.

More *For* Execution

- The loop-body continues to execute until the condition becomes false.
- If the condition is initially false, the Loop-Body will be skipped entirely.
- The Start-Body, and Continue-Body may consist of several statements separated by commas.

For Details

- Any part, Start-Body, Continue-Body, or Condition may be omitted. The semi-colons are required.

For Example 1

- Processing an Array

```
for (i = 0 ; i < ArraySize ; i++)
{
    A[i] += 10;
}
```

For Example 2

- Processing a Singly-Linked List with Previous-Element Pointer

```
for (Curr=Start,Prev=NULL ;
        Curr != NULL && Curr->Type != Red ;
          Prev=Curr,Curr=Curr->Next)
{
      Curr->Size += 3;
}
```

Break and Continue

- Early termination of a loop is accomplished using the *break* and *continue* statements.
- *Break* terminates the current loop immediately. The current-loop is the most deeply nested loop containing the *break* statement.
- *Continue* is similar to *break*, but goes on to the next iteration of the loop.

Case Statements

- The Case statement is actually called Switch, and has the following format.

```
switch (<numeric expression>)
{
    case <value-1>:
    {
    }
    break;
    case <value-2>:
    { ...
}
```

Case Details

- The Numeric-Expression must be something that evaluates to an integer.
- <value-1>, <value-2>, … must be integer *constants*.
- Don’t forget the *break* statements, or you will be sorry.

Case Variations

- If you want to do the same thing for two different values, say 5, and 17, you can place case labels one after the other as follows.

```
case 5:
case 17:
{
    <case-body>
}
break;
```

Case Variations II

- The equivalent of the *else* keyword is the Case *default* label, which is used as follows.

```
default:
{
    <Default-Body>
}
break;
```

Windows Programming

- Windows User Interface Programming is extremely complicated in C. We will avoid this complexity by using Visual Basic.
- For Speed and Versatility, it is convenient to program some things in C, even when using Visiual Basic.

Dynamic Link Libraries

- Contain no user-interface code.
- Contain only useful functions.
- Compiled independently of other programs.
- Can be used by other programs, including those written in Visual Basic.

Creating DLLs

- Write your C programs as you normally would.
- Create the *.c* files and *.h* files you need.
- For each function that will be used by some other program, change the header:

  OLD: int f1(int a, int b)

  NEW: int FAR PASCAL _export int f1(int a, int b)

Creating DLLs II

- Add the following function.

```
int FAR PASCAL LibMain(HANDLE hInstance,
                       WORD wDataSeg,
                       WORD wHeapSize,
                       LPSTR lpszCmdLine)
{
    return 1 ;
}
```

Creating DLLs III

- Add the following function.

```
int FAR PASCAL _export WEP(int exittype)
{
    return 1;
}
```

Creating DLLs IV

- Add the following line to the beginning of each *.c* file, or to the beginning of your common *.h* file.

```
#include <windows.h>
```

Creating DLLs V

- If you haven’t done so already, tell your compiler that you want to create a .dll file. (This is usually done when first starting the project.)
- Compile, and correct syntax errors, until you have a clean compile.
- See documentation on Visual Basic for testing procedures.

Introduction to C Programming

Windows Operating System

Our Purpose

- This is *not* a C Programming Course
- You must already know how to program
- You will be familiarized with the basics of the C language
- Will cover simple Windows Programming
- User Interface programming will *not* be covered

C Functions

- All C programming must be part of a C function.
- Example Declaration:

```
void MyFunc(int a,int b)
{
    int c;

    c = a + b;
}
```

Formal Syntax

- A function is declared as follows
- The <Type> is return value type and function characteristics

```
<Type>   <Function Name> ( <Argument List> )
{
      <Local Variable Declarations>

      <Executable Code>
}
```

Types

- The most common types in C are the following:

int	16-bit integer
long	32-bit integer
short	16-bit integer
char	8-bit integer or character
float	32-bit floating point
double	64-bit floating point

Type Declarations

- Type Declarations declare simple variables as well as pointers and arrays
- *int a;* -- defines *a* to be a 16-bit integer.
- *long b,c,d;* -- defines *b*, *c*, and *d* to be 32-bit integers.
- *char *xyz;* -- *xyz* is a pointer to a *char*.
- *int Totals[15]* -- *Totals* is an array of 15 *ints*.

Function Headers

- The type *void* is used to indicate no return value, or no argument list.
- Example: *void Func1(void)*
- Each argument must have a declared type preceding its name
- Example: *int F2(int a, int b, char c)*

Function Bodies

- A function body consists of two parts:
  - Declaration of Local Variables
  - Executable code
- Example:

```
int F2(int a, int b)
{
        int c;

        c = a*a;
        c += b;
        return c;
}
```

Global Variables

- Arguments and Local Variables are accessible only inside the function where they are declared.
- Variable declarations that are placed outside of any function are accessible to all functions, and retain their values for the life of the program.

Globals: An Example

```
int a;  // a global variable

void f1(int b,int c)
{
        int k; // local k

        k = b*b;
        a = k + c;
}
```

```
// a different b and c
void f2(int b,int c)
{
        int k; // a different k

        k = b + 2;
        // the same a as before
        a = k * c;
}
```

Multiple Files

- Complex programs consist of many files.
- To make global variables in one file accessible to another file, use the *extern* keyword

```
// in file x.c
int x;
```

```
// in file y.c
extern int x;
```

Both files use the same copy of *x*.

Header Files

- If you have many extern statements that appear in many different files,
  - Place all externs in one file with a .h extension, such as *externs.h*
  - Place the statement *#include "externs.h"* at the beginning of each file
- This is also a good place for structure definitions and function templates.

Function Templates

- Every function should have a function template.
- The template must physically appear before the first use of the function.
- The template is identical to the function header, but ends in a semicolon, and has no body.

Template Example

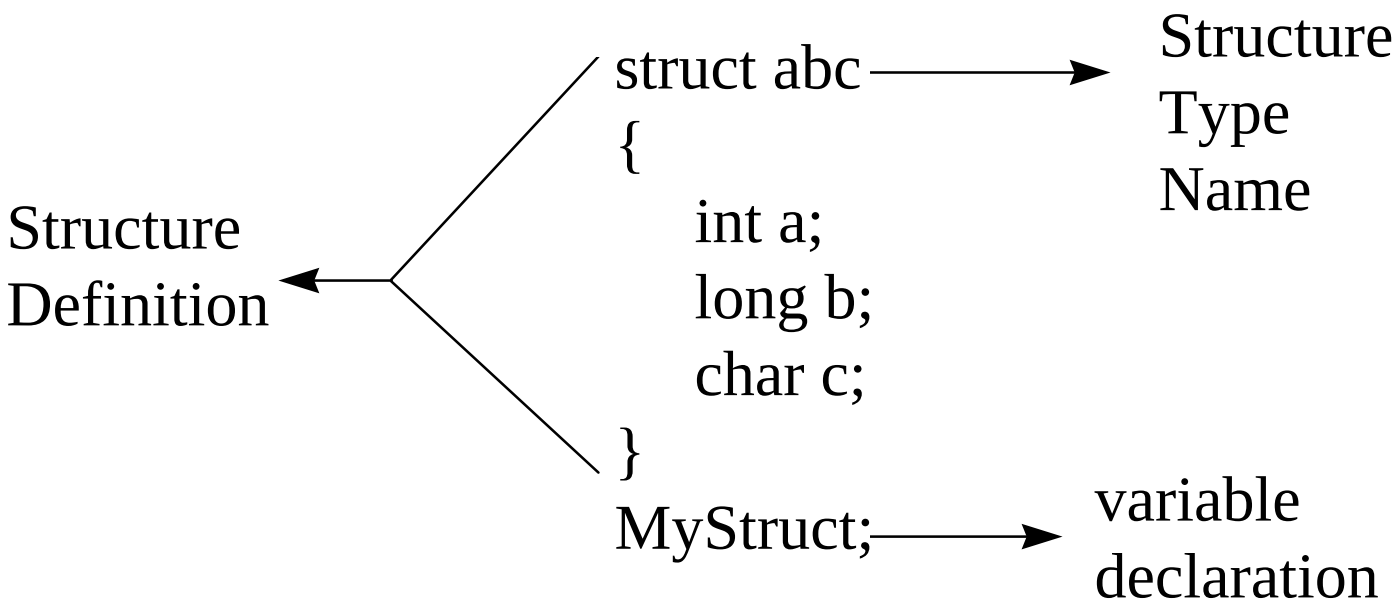
```
float factorial(float x);
float Binomial(float a,float b)
{
        int n,d;

      n = factorial(a);
      d = factorial(b)*factorial(a-b);
      return n/d;
}
float factorial(float x)
{ ...
```

Templates: Notes

- If a function is defined before it is used, the template is not required, but should be used anyway.
- Templates should be placed in a common header file with a *.h* extension.
- Templates make functions defined in one file accessible to functions in other files.

Structures

- Structures are collections of variables of different types, as in the following example.

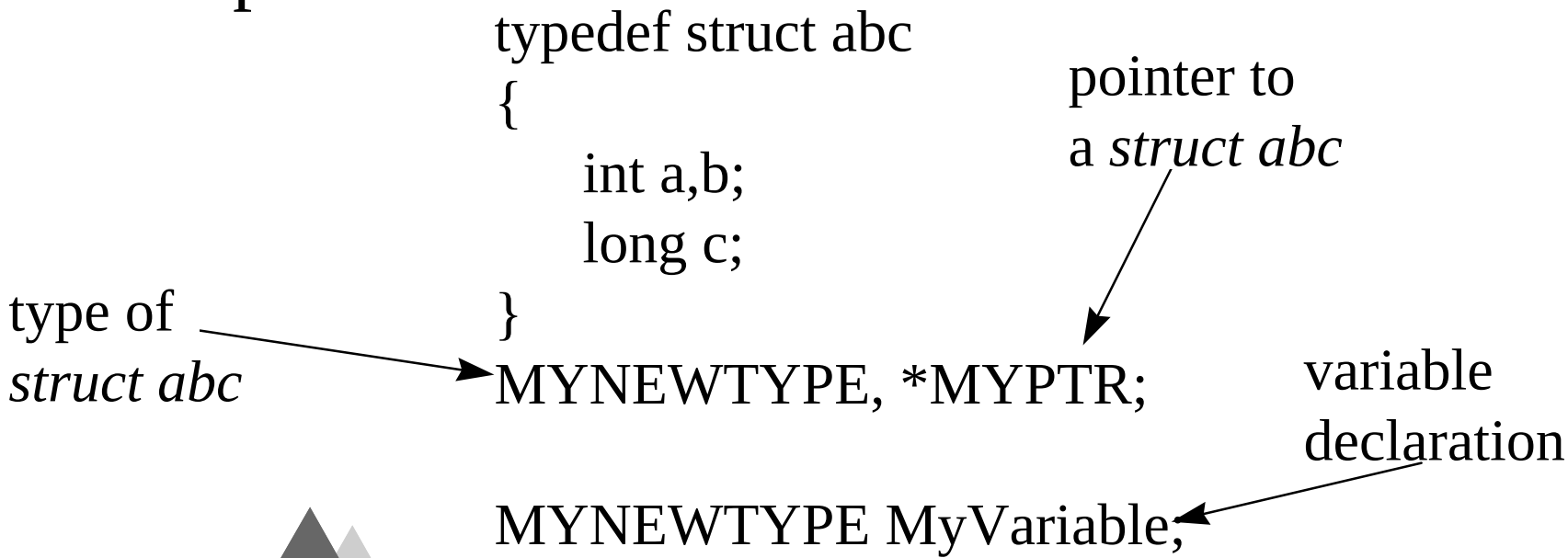


Structures: Notes

- The details of the structure need only be specified once. Subsequent definitions of the same type of structure are done as follows.
- struct abc m,n,q;
- The variable declaration on the previous slide is optional.

New Types

- The most common use of structures is to declare new types as in the following example.



Type Definitions: Notes

- All commonly used structures should be declared as new types.
- By convention, new type names are capitalized.
- New type names can be used anywhere that int, long, … are used.
- Place type definitions in a common *.h* file.

Accessing Variables

- Arrays are accessed as in other languages, but the first index is always zero.
- Example *A[3,4] = B[0];*
- Structure elements are accessed using a period.
- Example: *MyStruct.a = YourStruct.c;*

Accessing Variables II

- Pointers are dereferenced by preceding them with an asterisk.
- Example *MyPtr = *YourPtr;
- Pointers to structures are dereferenced using the -> operator.
- Example MyPtr->a = YourPtr->b;

Arrays and Pointers

- A pointer can point to either a simple variable or an array.
- Accessing an array pointer is identical to accessing an array.
- Example MyArray[3] = MyPtr[5];
- *MyPtr and MyPtr[0] are identical.

Passing Data to Functions

- All arguments are passed by value.
- Arrays are passed by passing the address of the array to the function. Access is identical to accessing the array directly.
- Structures are copied and passed by value.
- All floats are converted to doubles, and converted back inside the function.

Passing by Reference

- Declare the function argument as a *pointer* to the desired type.
- When passing a variable, precede it by the & operator, which extracts the address of the variable.
- Reference the variable through the pointer.
- Use this to avoid copying massive structures to the argument stack.

Assignments

- The equals sign is the assignment operator.
  - *a = b + c;*
- All common arithmetic operators, *except exponentiation*, can be used.
- % is used for remainders, & for bit-wise AND, | for bit-wise OR, ~ for bit-wise NOT, and ^ for bit-wise Exclusive-OR.

Other Operators

- The expression *A* << *k* can be used to shift *A* to the left by *k* bits.
- The operator >> is used for right shift.
- Any Binary Operator can Be combined with the equals sign.

```
a += 5;  // increment a by 5.
b *= 4; // multiply b by 4.
```

Even More Operators

- To increment or decrement a variable by 1, use the ++ and -- operators.
- A++, increments A by 1
- B--, decrements B by 1
- All assignment expressions have a value, so A = B = C = D = 1, sets A, B, C, and D to 1.

Comparisons

- The Comparison Operators are as follows

==	Equals
!=	Not Equals
<	Less Than
>	Greater Than
<=	Less than or equal
>=	Greater than or equal

Comparison Results

- All Comparison Operators Produce a Numeric value: False produces zero, while True produces One.
- Complex Tests can be created using AND, OR and NOT operators.
  - && logical AND
  - || logical OR
  - ! logical NOT

If Statements

- The format of the *if* statement is as follows.

```
if  (<Numeric Expression>)
{
    <True-Body>
}
else
{
     <False-Body>
}
```

If Evaluation

- If the numeric expression is zero, it is considered to be False, otherwise it is considered to be True.
- If the expression is True, the True-Body is executed, otherwise the False-Body is executed.
- The False-Body may be omitted, along with the *else* keyword and the enclosing braces.

While Statements

- The format of the *while* statement is as follows.

```
while (<Numeric Expression>)
{
    <While Body>
}
```

While Execution

- If the Numeric Expression is zero, it is considered to be False, otherwise it is considered to be True.
- The Loop-Body is executed until the Numeric Expression becomes False.
- The loop body will be skipped entirely if the expression is initially false.

For Loops

- In C, the *for* statement is used for most loops. The syntax is as follows.

```
for (<Start-Body> ; <Condition> ; <Continue-Body>)
{
    <For-Body>
}
```

For Execution

- The C for statement is a special case of the while.
- The Start-Body is executed before the loop begins.
- The Condition is tested before executing the Loop-Body.
- The Continue-Body is executed after the Loop-Body.

More *For* Execution

- The loop-body continues to execute until the condition becomes false.
- If the condition is initially false, the Loop-Body will be skipped entirely.
- The Start-Body, and Continue-Body may consist of several statements separated by commas.

For Details

- Any part, Start-Body, Continue-Body, or Condition may be omitted. The semi-colons are required.

For Example 1

- Processing an Array

```
for (i = 0 ; i < ArraySize ; i++)
{
    A[i] += 10;
}
```

For Example 2

- Processing a Singly-Linked List with Previous-Element Pointer

```
for (Curr=Start,Prev=NULL ;
        Curr != NULL && Curr->Type != Red ;
          Prev=Curr,Curr=Curr->Next)
{
      Curr->Size += 3;
}
```

Break and Continue

- Early termination of a loop is accomplished using the *break* and *continue* statements.
- *Break* terminates the current loop immediately. The current-loop is the most deeply nested loop containing the *break* statement.
- *Continue* is similar to *break*, but goes on to the next iteration of the loop.

Case Statements

- The Case statement is actually called Switch, and has the following format.

```
switch (<numeric expression>)
{
    case <value-1>:
    {
    }
    break;
    case <value-2>:
    { ...
}
```

Case Details

- The Numeric-Expression must be something that evaluates to an integer.
- <value-1>, <value-2>, … must be integer *constants*.
- Don’t forget the *break* statements, or you will be sorry.

Case Variations

- If you want to do the same thing for two different values, say 5, and 17, you can place case labels one after the other as follows.

```
case 5:
case 17:
{
    <case-body>
}
break;
```

Case Variations II

- The equivalent of the *else* keyword is the Case *default* label, which is used as follows.

```
default:
{
    <Default-Body>
}
break;
```

Windows Programming

- Windows User Interface Programming is extremely complicated in C. We will avoid this complexity by using Visual Basic.
- For Speed and Versatility, it is convenient to program some things in C, even when using Visiual Basic.

Dynamic Link Libraries

- Contain no user-interface code.
- Contain only useful functions.
- Compiled independently of other programs.
- Can be used by other programs, including those written in Visual Basic.

Creating DLLs

- Write your C programs as you normally would.
- Create the *.c* files and *.h* files you need.
- For each function that will be used by some other program, change the header:

OLD: int f1(int a, int b)

NEW: int FAR PASCAL _export int f1(int a, int b)

Creating DLLs II

- Add the following function.

```
int FAR PASCAL LibMain(HANDLE hInstance,
                       WORD wDataSeg,
                       WORD wHeapSize,
                       LPSTR lpszCmdLine)
{
    return 1 ;
}
```

Creating DLLs III

- Add the following function.

```
int FAR PASCAL _export WEP(int exittype)
{
    return 1;
}
```

Creating DLLs IV

- Add the following line to the beginning of each *.c* file, or to the beginning of your common *.h* file.

```
#include <windows.h>
```

Creating DLLs V

- If you haven’t done so already, tell your compiler that you want to create a .dll file. (This is usually done when first starting the project.)
- Compile, and correct syntax errors, until you have a clean compile.
- See documentation on Visual Basic for testing procedures.