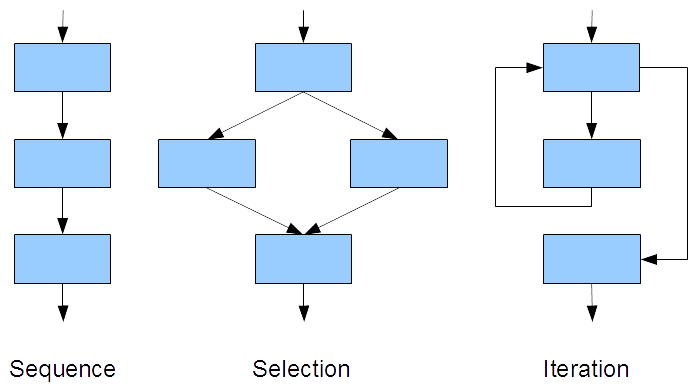
**BTP100: Final Exam Review**

1. State two reasons why structured programming insists on one entry point and one exit point to any construct in a program such as for-next, while, a function, etc… (Week 3 – Computations: Logic)

Answer [Aim for one entry point, one exit point. You do all initialization at one spot – the entry point, and all code cleanup at one spot – the exit point. Breaks, continues are discouraged in loops. Returns in the middle of a function are discouraged]:

A complete programming language includes facilities to implement sequential constructs, in which one statement follows another and the statements are executed in order, and two other constructs, which represent modifications of sequential constructs.  Selection constructs represent different paths through the set of instructions.  Iteration constructs represent repetition of the same set of instructions until a specified condition has been met.  The three classes of constructs required to complete a programming language are illustrated in the figure below.

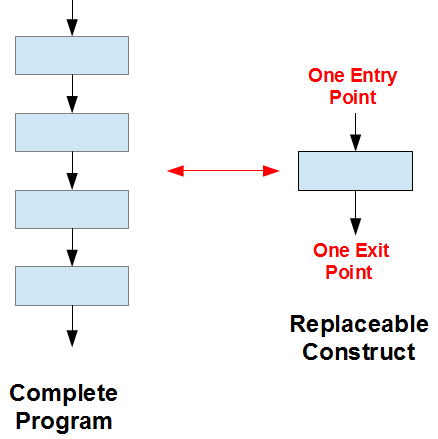


Since programmers who maintain application software are typically not those who develop that software originally and the maintenance programmers may change throughout the lifetime of a software application, it is critical that the software is not only readable but also easy to upgrade and maintain.  The principles of structured programming, which were developed in the 1960s, provide important coding guidelines that respect this objective.

This chapter introduces the selection and iteration constructs supported by the C language and describes how to implement structured programming principles in coding iterations.

Structured Programming

A structured program consists of sets of simple constructs, each of which has one entry point and one exit point.  Any programmer may replace one construct with an upgraded construct without affecting the other constructs in the program or introducing errors ("bugs").



The simplest example of a structured construct is a *sequence*.  A sequence is either a simple statement or a code block.  A *code block* is a set of statements enclosed in a pair of curly braces to be executed sequentially.

For example (more practical examples below)

|  |  |
| --- | --- |
| // single statement (original)  printf("I like pizza\n"); | // code block (upgrade)  {  printf("I like pizza\n");  printf("I want more pizza\n");  } |

Unlike a single statement, a C code block does not require a terminating semi-colon of its own (after the closing brace).

1. What are two ways a function can change and return a variable? (Week 6 – Modularity: Functions, Pointers)

Answer [The function can return a value which is assigned to a variable in the calling code, or a function can take the address of a variable and change this variable within itself]:

A function consists of a header and a body.  The body is the code block that contains the detailed instructions to be performed by the function.  The header immediately precedes the body and includes the following *in order*

1. the type of the function's return value
2. the function's identifier
3. a parentheses-enclosed list of parameters that receive data from the caller

*type identifier*(*type parameter*, ..., *type parameter*)

{

// function instructions

return *x*; // x denotes the value returned by this function

}

*type* specifies the type of the return value or the function's parameter.  *identifier* specifies the name of the function.  *parameter* is a variable that holds data received from the caller function.

For example,

|  |  |
| --- | --- |
| /\* Raise an integer to an integer  \* power.c  \*/  #include <stdio.h>  int power(int base, int exponent)  {  int i, result;  result = 1;  for (i = 0; i < exponent; i++)  result = result \* base;  return result;  }  int main(void)  {  int base, exp, answer;  printf("Enter base : ");  scanf("%d", &base);  printf("Enter exponent : ");  scanf("%d", &exp);  answer = power(base, exp);  printf("%d^%d = %d\n", base, exp, answer);  } | Enter base : 3  Enter exponent : 4  3^4 = 81 |

The first function returns a value of int type.  power identifies the function.  base and exponent are the function's parameters; both are of int type.

A function can receive in its parameters not only data values but also addresses of program variables.

Consider a function named internal\_swap() that swaps the values stored in two memory locations.  We call this function from main() and note that the swap remains completely within the function itself:

|  |  |
| --- | --- |
| /\* Internal swap  \* internal\_swap.c  \*/  #include <stdio.h>  void internal\_swap (int a, int b)  {  int c;  printf("a is %d, b is %d\n", a, b);  c = a;  a = b;  b = c;  printf("a is %d, b is %d\n", a, b);  }  int main(void)  {  int a, b;  printf("a is ");  scanf("%d", &a);  printf("b is ");  scanf("%d", &b);  internal\_swap(a, b);  printf("After internal\_swap:\na is %d\n"  "b is %d\n", a, b);  return 0;  } | a is 5, b is 6            a is 6, b is 5              a is 5    b is 6          After internal\_swap:  a is 5  b is 6 |

Although internal\_swap() does exchange the values in a and b, the pass by value mechanism preserves the original values in main().

Walkthrough Table

The walkthrough table shows how the changes remain completely within internal\_swap()

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| void | | | int | |
| local\_swap(int a, int b) | | | main(void) | |
| int | int | int | int | int |
| a | b | c | a | b |
| 0x0012FF78 | 0x0012FF7C | 0x0012FF6C | 0x0012FF88 | 0x0012FF84 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  | 5 | 6 |
| 5 | 6 | ? | 5 | 6 |
| 5 | 6 | 5 | 5 | 6 |
| 6 | 6 | 5 | 5 | 6 |
| 6 | 5 | 5 | 5 | 6 |
| 6 | 5 | 5 | 5 | 6 |

The hexadecimal values below the variable identifiers are their addresses in memory.  Note that the addresses of a and b in internal\_swap() are different from those in main().

The program copies the argument values (a and b) as initial values into parameters a and b.  The swapping only affects a and b in internal\_swap().

Pass by Address

To change the original values, we pass the addresses of their variables instead of their values.  We use these addresses to access the original values and change them from within the function.

Consider the following program

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| /\* Swapping values using a function  \* swap.c  \*/  #include <stdio.h>  void swap(int \*p, int \*q)  {  int c;  c = \*p;  \*p = \*q;  \*q = c;  }  int main(void)  {  int a, b;  printf("a is ");  scanf("%d", &a);  printf("b is ");  scanf("%d", &b);  swap(&a, &b);  printf("After swap:\na is %d\n"  "b is %d\n", a, b);  return 0;  }  Walkthrough Table  The walkthrough table shows how the changes carry over to main()   |  |  |  |  |  | | --- | --- | --- | --- | --- | | void | | | int | | | swap(int \*p, int \*q) | | | main(void) | | | int \* | int \* | int | int | int | | p | q | c | a | b | | 0x0012FF78 | 0x0012FF7C | 0x0012FF6C | 0x0012FF88 | 0x0012FF84 |  |  |  |  |  |  | | --- | --- | --- | --- | --- | |  |  |  | 5 | 6 | | 0x0012FF88 | 0x0012FF84 | ? | 5 | 6 | | 0x0012FF88 | 0x0012FF84 | 5 | 5 | 6 | | 0x0012FF88 | 0x0012FF84 | 5 | 6 | 6 | | 0x0012FF88 | 0x0012FF84 | 5 | 6 | 5 | | 0x0012FF88 | 0x0012FF84 | 5 | 6 | 5 |   Some programmers prefer symbolic notation instead of address values.  For example, they use the symbol main::a to refer to the local variable a in the function main().  A walkthrough table using symbolic notation looks something like:   |  |  |  |  |  | | --- | --- | --- | --- | --- | | void | | | int | | | swap(int \*p, int \*q) | | | main(void) | | | int \* | int \* | int | int | int | | p | q | c | a | b | | 0x0012FF78 | 0x0012FF7C | 0x0012FF6C | 0x0012FF88 | 0x0012FF84 |  |  |  |  |  |  | | --- | --- | --- | --- | --- | |  |  |  | 5 | 6 | | main::a | main::b | ? | 5 | 6 | | main::a | main::b | 5 | 5 | 6 | | main::a | main::b | 5 | 6 | 6 | | main::a | main::b | 5 | 6 | 5 | | main::a | main::b | 5 | 6 | 5 | | a is 5    b is 6          After swap:  a is 6  b is 5 |

1. When passing parameters to functions, what is pass by value and pass by address? When would you use pass by value and when would you use pass by address? (Week 6 – Modularity: Pointers)

Answer [Use pass by value when you do not want to change the values of the variables in the calling function. Use pass by address when you do want to change the values of the variables in the calling functions]:

Pass By Value

The C language passes data from a caller to a function by value.  That is, it passes a copy of the value and not the value itself.  The value passed is stored as the inital value is the parameters that corresponds to the argument in the function call.

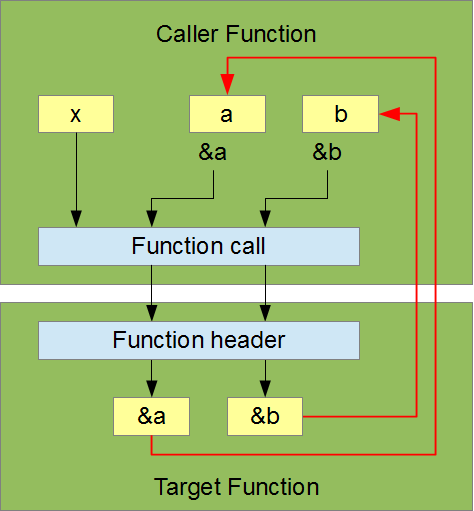
Each parameter is a variable with its own memory location.  We refer to the mechanism of allocating separate memory locations for parameters and using the arguments in the function call to initialize these parameters *pass by value*.  Pass by value facilitates modular design by localizing consequences.  The function being called may change the value of any of its parameters many times, but the values of the corresponding arguments in the caller remain unchanged.  In other words, a function cannot change the value of an argument in the call to the function.  This language feature ensures the variables in the caller are relatively secure.

Pass by Address

To change the original values, we pass the addresses of their variables instead of their values.  We use these addresses to access the original values and change them from within the function.

Multiple Return Values

C function syntax only allows for the return of a single value.  If program design requires a function that returns more than one value, we do so through parameter pointers that hold the addresses of the variables that receive the multiple return values.



The following program converts day of year to month and day of month by calling function day\_to\_dm() to:

|  |  |
| --- | --- |
| /\* Day of Year to Day of Month and Month  \* day\_to\_dm.c  \*/  #include <stdio.h>  // day\_to\_dm return day and month of given day in year  // assumes not leap year  //  void day\_to\_dm(int day, int \*d, int \*m)  {  if (day < 32) {  \*m = 1;  \*d = day;  } else if (day < 60) {  \*m = 2;  \*d = day - 31;  } else if (day < 91) {  \*m = 3;  \*d = day - 59;  } else if (day < 121) {  \*m = 4;  \*d = day - 90;  } else if (day < 152) {  \*m = 5;  \*d = day - 120;  } else if (day < 182) {  \*m = 6;  \*d = day - 151;  } else if (day < 223) {  \*m = 7;  \*d = day - 181;  } else if (day < 254) {  \*m = 8;  \*d = day - 222;  } else if (day < 284) {  \*m = 9;  \*d = day - 253;  } else if (day < 305) {  \*m = 10;  \*d = day - 283;  } else if (day < 335) {  \*m = 11;  \*d = day - 304;  } else if (day < 366) {  \*m = 12;  \*d = day - 334;  }  }  int main(void)  {  int day, d, m;  printf("Day of Year : ");  scanf("%d", &day);  day\_to\_dm(day, &d, &m);  printf("Day/Month is %d/%d\n", d, m);  return 0;  } | Day of Year : 357  Day/Month is 23/12 |

Functions that return values through their parameters can reserve their return values for reporting any error codes produced by the function.

1. With reference to functions, what is meant by
   1. Modular Design (give three design principles)?
   2. Cohesion?
   3. Coupling? (Week 6 – Modularity: Functions)

Answer [see below]:

Procedural programming involves separating source code into self-contained components that can be accessed multiple times from different locations in a complete program.  This approach enables separate coding of each component and assembly of various components into a complete program.  We call this approach to programming solutions *modular design*.

This chapter introduces the principles of modular design, describes the syntax for defining a module in the C language, shows how to pass data from one module to another, suggests a walkthrough table structure for programs composed of several modules and includes an example that validates user input.

Modular Design

Modular design identifies the components of a programming project that can be developed separately.  Each module consists of a set of logical constructs that are related to one another.  A module may refer to other modules.  A trivial example is the program described in the chapter on [Compilers](https://scs.senecac.on.ca/~btp100/pages/content/modul.html):

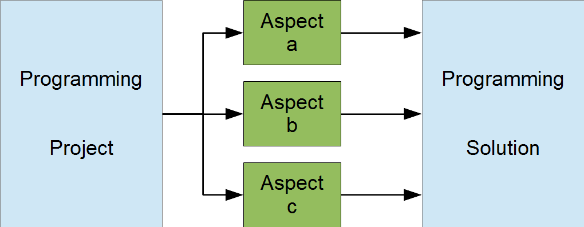
|  |
| --- |
| /\* My first program  hello.c \*/  #include <stdio.h> // information about the printf identifier  int main(void) // program startup  {  printf("This is C"); // output  return 0; // return to operating system  } |

The module named hello.c starts executing at statement int main(void), outputs a string literal and returns control to the operating system.

* main() transfers control to a module named printf()
* printf() executes the detailed instructions for outputting the string literal
* printf() returns control to main()

Design Principles

We can sub-divide a programming project in different ways.  We select our modules so that each one focuses on a narrower aspect of the project.  Our objective is to define a set of modules that simplifies the complexity of the original problem.



Some general guidelines for defining a module include:

1. the module is easy to upgrade
2. the module contains a readable amount of code
3. the module may be used as part of the solution to some other problem

For a structured design, we stipulate that

1. each module has one entry point and one exit point
2. each module is highly cohesive
3. each module exhibits low coupling

Cohesion

*Cohesion* describes the focus: a highly cohesive module performs a single task and only that task.

In creating a cohesive module, we ask whether our tasks belong to that module: a reason to include a task is its relation to the other tasks within the module.  A reason to exclude a task is its independence from other tasks within the module.

For example, the following tasks are related

* receives a date and the number of days to add
* converts the date into a format for adding days
* adds the number of days received
* converts the result to a new date
* returns the new date

The following tasks are unrelated

* calculates Federal tax on bi-weekly payroll
* calculates the value of π
* outputs an integer in hexadecimal format

We allocate unrelated tasks to separate modules in the program design.

Coupling

*Coupling* describes the degree of interrelatedness of a module with other modules.  The less information that passes between the module and the other modules the better the design.  We prefer designs in which each module completely control its own computations and avoids transferring control data to any other module.

Consider a module that receives a flag from another module and performs a calculation based on that flag.  Such a module is highly coupled: another module controls its execution.  To improve our design, we transfer data to the module and let it create its own flags before completing its task.

1. Why might you use const in a function parameter list? (Week 8 – Modularity: Functions, Arrays and Structures)

Answer [see below]:

Barring Changes

To prevent a function from changing any element of an array identified by a function parameter, we qualify the parameter as const.  The function header takes the form

*type function\_identifier*(const *type array\_identifier*[], ... )

or

*type function\_identifier*(const *type \*array\_identifier*, ... )

For example

|  |  |
| --- | --- |
| void display(const int g[], int n)  {  for(i = 0; i < n; i++)  printf("%d ", g[i]);  } | void display(const int \*g, int n)  {  for(i = 0; i < n; i++)  printf("%d ", g[i]);  } |

Any attempt to modify the value of an element of g will generate a compiler error.  Without the const keyword, we could reset the value of the first element to 10 by adding a statement like g[0] = 10;.

1. What is printed out in the following example if the user enters ‘3’?

What might the return value of factorial() represent?

#define \_CRT\_SECURE\_NO\_WARNINGS

#include<stdio.h>

int factorial(int, double \*);

main() {

int num;

double res;

printf("Enter a number : ");

scanf("%d", &num);

int retVal = factorial(num, &res);

if(retVal == 0) {

printf("The factorial of %d is %.2lf\n", num, res);

}

return 0;

}

int factorial(int number, double \*result) {

\*result = 1;

for (int i = 1; i <= number; ++i) {

\*result = \*result \* (double)i;

}

return 0;

}

Answer [see also Review1.c]:

The address of *res* is passed to factorial, therefore factorial can change the value of *res*. In factorial:

|  |  |
| --- | --- |
| i | \*result |
| 1 | 1\*1 = 1 |
| 2 | 2\*1 = 2 |
| 3 | 3\*2 = 6 |

The variable *double \*result* in factorial points to the variable *double res* in main. At the end, res has the value 6. The printout is therefore:

The factorial of 3 is 6.00

1. What does the following program print out:

#define \_CRT\_SECURE\_NO\_WARNINGS

#define CURRENT\_YEAR 2019

#define NUM 3

#include <stdio.h>

struct Car {

char brand[31];

int year;

double purchasePrice;

double currentValue;

};

int GetValue(struct Car \*);

int main() {

struct Car car[NUM] = {"Toyota Corolla", 2008, 21000.00, 21000.00,

"Mazda 3", 2013, 24000.00, 24000.00,

"Honda Accord", 2016, 30000.00, 30000.00 };

for (int i = 0; i < NUM; ++i) {

int err = GetValue(&car[i]);

if (err == 0) {

printf("The %d %s was bought for %.2lf but is now worth %.2lf\n",

car[i].year, car[i].brand,

car[i].purchasePrice, car[i].currentValue);

}

else {

printf("Your %d %s is worthless!\n", car[i].year, car[i].brand);

}

}

return 0;

}

int GetValue(struct Car \*car) {

int retVal=0;

double depreciation = 3000.00 \* (CURRENT\_YEAR - car->year);//Loses $3000.00/year

car->currentValue = car->purchasePrice - depreciation;

if (car->currentValue <= 0) {

car->currentValue = 0;

retVal = -1;

}

return retVal;

}

Answer [see also Review2.c]:

The structure struct Car car[NUM] represents an array of structures. We pass the address of each element of the array to GetValue by address. car by itself represents an address of the entire array, but &car[i] represents the address of each element. We are passing each element, one at a time, to GetValue() therefore we use &car[i]. See next page:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| i | Struct Car \*car | CURRENT  \_YEAR | car->year | depre-ciation | car->purchasePrice | car->currentValue | retVal |
| 0 | &car[0] | 2019 | 2008 | 33000 | 21000 | -12000(becomes 0) | -1 |
| 1 | &car[1] | 2019 | 2013 | 18000 | 24000 | 6000 | 0 |
| 2 | &car[2] | 2019 | 2016 | 9000 | 30000 | 21000 | 0 |

Your 2008 Toyota Corolla is worthless!

The 2013 Mazda 3 was bought for 24000.00 but it is now worth 6000.00

The 2016 Honda Accord was bought for 30000.00 but is now worth 21000.00

1. Write a function called CurveMarks that takes the array of students and multiplies each mark by CURVE\_RATIO. If a mark is over 100, set the mark to 100

#define \_CRT\_SECURE\_NO\_WARNINGS

#define CURVE\_RATIO 1.2

#define NUM 5

#define NUM\_MARKS 5

#include <stdio.h>

struct Student {

char name[31];

int marks[NUM\_MARKS];

int studentNumber;

};

void CurveMarks(struct Student st[]);

int main(void) {

struct Student student[NUM] = { "Bill", 45, 57, 78, 62, 56, 10000,

"Tomasz", 66,67,78,44,59, 10001,

"Anusha", 72, 76, 74, 81, 69, 10002,

"Peng", 61, 62, 47, 56, 59, 10003,

"Erika", 77, 72, 42, 66, 79, 10004 };

CurveMarks(student);

for (int i = 0; i < NUM; ++i) {

printf("%s's marks are now", student[i].name);

for (int j = 0; j < NUM\_MARKS; ++j) printf(" %d", student[i].marks[j]);

printf("\n");

}

return 0;

}

Answer [see also Review3.c]:

Note that *student* is passed through to CurveMarks. This is the address of the entire array itself. Remember that *&student[0]* represents the address of one element in the array (the first), but *student* represents the address of the entire array.

First of all, you have to add a function prototype above main(), which takes as an argument the address of the entire array, perhaps put this prototype below the declaration of struct Student.

[…]

struct Student {

char name[31];

int marks[5];

int studentNumber;

};

void CurveMarks(struct Student st[]);//You could also use pointer notation

int main(void) {

[…]

CurveMarks will go through the entire array of students, so we start with a for-next loop from index 0 to NUM-1. Note the scope of NUM, NUM\_MARKS and CURVE\_RATIO. Since these variables are global in scope, we do not have to pass them through to CurveMarks(). CurveMarks() sees these variables.

For each student, we have to curve each mark. So for each student, we have to calculate a new mark for NUM\_MARKS marks. Therefore we have a nested for-next loop inside that goes from mark 0 to NUM\_MARKS-1.

For each mark, we test against 100. If over, set the new mark to 100:

void CurveMarks(struct Student st[]) {

for (int i = 0; i < NUM; ++i) {

for (int j = 0; j < NUM\_MARKS; ++j) {

int newMark = st[i].marks[j] \* CURVE\_RATIO;

if (newMark > 100) newMark = 100;

st[i].marks[j] = newMark;

}

}

}

The question did not ask for a print-out, but here it is anyway:

**Bill's marks are now 54 68 93 74 67**

**Tomasz's marks are now 79 80 93 52 70**

**Anusha's marks are now 86 91 88 97 82**

**Peng's marks are now 73 74 56 67 70**

**Erika's marks are now 92 86 50 79 94**

1. There are five errors in the following code. Can you find them?

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

int ReturnGrade(const double mark);

int main(void) {

double markInPercent;

do {

printf("Enter your mark in percent (enter -1 to quit): ");

scanf("%lf", markInPercent);

if (markInPercent >= '0') {

char grade[] = ReturnGrade(markInPercent);

switch (grade) {

case 'A':

printf("You got an A. You win a scholarship of $5000\n");

case 'B':

printf("You got a B. You win a scholarship of $1000\n");

case 'C':

case 'D'://fall-through

printf("You got a %c. You passed\n", grade);

break;

case 'F':

printf("You have failed.\n");

}

printf("\n");

}

} while (markInPercent >= 0);

}

char ReturnGrade(const double mark) {

char grade;

mark += 2;//One lab was not fair, add 2 percent to each mark

if (mark >= 80) grade = 'A';

else if (mark >= 70) grade = 'B';

else if (mark >= 60) grade = 'C';

else if (mark >= 50) grade = 'D';

else grade = 'F';

return grade;

}

Answer: [see also commented out code in Errors.c]

I will display the corrected code on the next page with comments showing where the errors are. You could put the comments directly on the original code on the final exam.

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

//int ReturnGrade(const double mark);

char ReturnGrade(double mark);//Return char, Remove the const

int main(void) {

double markInPercent;

do {

printf("Enter your mark in percent (enter -1 to quit): ");

// scanf("%lf", markInPercent);

scanf("%lf", &markInPercent);//pass address of markInPercent

// if (markInPercent >= '0') {

if (markInPercent >= 0) {//Compare again a number and not a character

// char grade[] = ReturnGrade(markInPercent);

char grade = ReturnGrade(markInPercent);//ReturnGrade returns a char

// so get rid of the array

switch (grade) {

case 'A':

printf("You got an A. You win a scholarship of $5000\n");

break;//Add a break here

case 'B':

printf("You got a B. You win a scholarship of $1000\n");

break;//Add a break here

case 'C'://No break is required because a fallthrough is intended

case 'D'://fall-through

printf("You got a %c. You passed\n", grade);

break;

case 'F':

printf("You have failed.\n");

}

printf("\n");

}

} while (markInPercent >= 0);

}

//char ReturnGrade(const double mark) {

char ReturnGrade(double mark) {//Remove the const

char grade;

mark += 2;//One lab was not fair, add 2 percent to each mark

if (mark >= 80) grade = 'A';

else if (mark >= 70) grade = 'B';

else if (mark >= 60) grade = 'C';

else if (mark >= 50) grade = 'D';

else grade = 'F';

return grade;

}

1. Write a program that stores geological data for an earthquake. Create a structure called GeoData which stores the following elements:

Name of the city [30 characters]

Latitude of the city [double float]

Longitude of the city [double float]

Wave amplitude of the tremours [double float]

Richter Scale of the earthquake[double float]

The main program will store geological data for three cities of type *GeoData*. The main program will prompt the user to enter the following for each of the three cities:

The city name,

The latitude and longitude separated by a space on the same line

The amplitude of the wave in millimeters

The main program will calculate the richter scale for each entry by calling a function CalculateRichterScale which will return an integer and will accept a pointer to a GeoData structure.

The main function will determine which city has the greatest tremor. The main function will then print out which city has the greatest tremor, its latitude and longitude, the amplitude of the shock waves, and the richter scale of the earthquake.

The function CalculateRichterScale return an integer (0:no error, -1: error). It takes the wave amplitude of the tremor and calculates the richter scale according to the following formula:

richterScale = sqrt(amplitude);

To use the sqrt() function, you will have to include the math library in math.h.

Answer [see also Review4.c]:

Create a structure according to the specifications:

struct GeoData {

char name[31];//Add one character for the null terminator

double latitude;

double longitude;

double waveAmplitude;

double richterScale;

};

There are three cities of type GeoData in main. Let us define a constant NUM of value three:

#define NUM 3

[…]

struct GeoData city[NUM];

Information has to be entered for each of the three which implies a for-next loop from 0 to 3 (NUM). Also, the loop will keep track of the largest amplitude and the index where it occurs. In the loop the user will be asked for name, latitude and longitude and wave amplitude:

[…]

double largestAmplitude = 0;

int index = -1;

for (int i = 0; i < NUM; ++i) {

printf("Enter the city name: ");

scanf("%30s", city[i].name);

printf("Enter the latitude and longitude separated by a space: ");

scanf("%lf %lf", &city[i].longitude, &city[i].latitude);

printf("Enter the wave Amplitude in millimeters: ");

scanf("%lf", &city[i].waveAmplitude);

[…]

In the loop the function CalculateRichterScale will be called passing a city through as an argument:

int err = CalculateRichterScale(&city[i]);

If there is no error, we will test the amplitude of the shock of that city with the largest shock. If it is bigger, we update the largest amplitude and the index in the array where the largest amplitude occurs.

if (err==0 && city[i].waveAmplitude > largestAmplitude) {

largestAmplitude = city[i].waveAmplitude;

index = i;

}

At the end we report the city with the largest tremor according to spec:

printf("The city with the largest tremor is %s at latitude %.2lf and longitude %.2lf\n",

city[index].name, city[index].latitude, city[index].longitude);

printf("The tremor measured %.2lfmm which is %.2lf on the Richter scale\n",

city[index].waveAmplitude, city[index].richterScale);

For the function CalculateRichterScale, we do a validity test on the wave amplitude then calculate the Richter scale. For sqrt() we have to include “math.h”. We also need to include stdio.h for all the input/output. Don’t forget the function prototype at the top. The function will return an error status:

#include "math.h"

#include <stdio.h>

int CalculateRichterScale(struct GeoData \*geo);

[…]

int CalculateRichterScale(struct GeoData \*geo) {

int retVal = 0;

if (geo->waveAmplitude < 0) {

geo->richterScale = 0;

retVal = -1;

}

else {

geo->richterScale = sqrt(geo->waveAmplitude);

}

return retVal;

}

Putting it all together we have:

#define \_CRT\_SECURE\_NO\_WARNINGS

#define NUM 3

#include "math.h"

#include <stdio.h>

struct GeoData {

char name[31];

double latitude;

double longitude;

double waveAmplitude;

double richterScale;

};

int CalculateRichterScale(struct GeoData \*geo);

int main(void) {

struct GeoData city[NUM];

double largestAmplitude = 0;

int index = -1;

for (int i = 0; i < NUM; ++i) {

printf("Enter the city name: ");

scanf("%s", city[i].name);

printf("Enter the latitude and longitude separated by a space: ");

scanf("%lf %lf", &city[i].longitude, &city[i].latitude);

printf("Enter the wave Amplitude in millimeters: ");

scanf("%lf", &city[i].waveAmplitude);

int err = CalculateRichterScale(&city[i]);

if (err==0 && city[i].waveAmplitude > largestAmplitude) {

largestAmplitude = city[i].waveAmplitude;

index = i;

}

}

printf("The city with the largest tremor is %s at latitude %.2lf and longitude %.2lf\n",

city[index].name, city[index].latitude, city[index].longitude);

printf("The tremor measured %.2lfmm which is %.2lf on the Richter scale\n",

city[index].waveAmplitude, city[index].richterScale);

}

int CalculateRichterScale(struct GeoData \*geo) {

int retVal = 0;

if (geo->waveAmplitude < 0) {

geo->richterScale = 0;

retVal = -1;

}

else {

geo->richterScale = sqrt(geo->waveAmplitude);

}

return retVal;

}