ECE 220 Final Sample

Student Name (print): ______________________________

Student ID (print): ______________________________

Student Section (print): ______________________________

Exam Room: ______________________________

This exam contains **23 pages** (including this cover page & references) and **5 questions**. The total number of possible points is **100**. Always enter your answers in the space provided. For multiple choice questions, clearly and unambiguously indicate your choices. All pages must be turned in at the end of the exam.

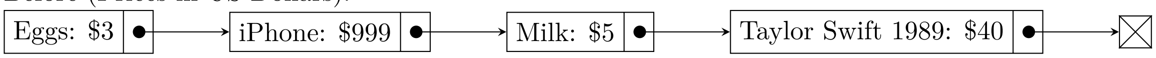
- This is a closed-book, closed-notes exam.
- You may **not** use any personal electronic devices such as cellphone and calculator.
- Absolutely **no interaction** is allowed between students – the exam hall may be electronically monitored/recorded for ensuring exam integrity.
- Answers must be written neatly. **Illegible handwriting will be graded as incorrect.**

Do not write in the table to the right.

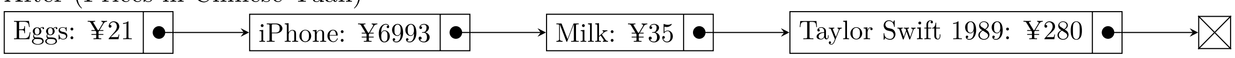
Question	Points	Score
1	20	
2	30	
3	20	
4	20	
5	10	
Total:	100	

1. (20 points) **Currency Exchange**
Asher just bought some groceries from Target. His mother called from China and asked how much everything costs. Help him write a program that converts all the prices into another currency given the exchange rate. In the following example, 1 US Dollar ($) ≈ 7 Chinese Yuan (¥). Assume all prices and exchange rates are positive integers.

Before (Prices in US Dollars):



After (Prices in Chinese Yuan)




```
typedef struct product {
    char *product_name;
    unsigned int price;
    struct product * next;
} product;


void convert_currency(product* head_ptr, int exchange_rate) {
    if (head_ptr == NULL) return;
    head_ptr->price *= exchange_rate;
    convert_currency(head_ptr->next, exchange_rate);
}
```

Recall that a function's activation record has the following format:

Local Variables
Caller's Frame Pointer
Return Address
Return Value
Arguments

Additional Notes:

- R6 - Stack top pointer (top most occupied location)
- R5 - Frame pointer

Implement `convert_currency` in LC-3 with the use of Run-Time-Stack. **You must conform to the C to LC-3 calling convention covered in lectures.**

```
convert_currency
    ; Callee Setup
    ; Reserve space on the stack for book keeping info
    (1) ___________________________________________
    ; Store return address on the stack
    (2) ___________________________________________
    ; Store caller's frame pointer on the stack
    (3) ___________________________________________
    ; Set new frame pointer
    (4) ___________________________________________

    ; Function logic
    ; Load argument head_ptr into R0 using R5
    (5) ___________________________________________
    ; Check if head_ptr is NULL, if so, go to callee teardown
    (6) ___________________________________________
    ; Load head_ptr->price into R1
    (7) ___________________________________________
    ; load argument exchange_rate into R2 using R5
    (8) ___________________________________________
    ; Mutiply head_ptr->price by exchange_rate
    ; Set R3 to equal R1
    (9) ___________________________________________
    ; Clear R1
    (10) ___________________________________________
    ; Continuously add R2 to R1 until R3 is 0
    MULTIPLY_LOOP
    ; Check if R3 is 0
    (11) ___________________________________________
    ; If R3 is 0, exit loop
    (12) ___________________________________________
```

 UIN: ____________________

```
; ADD R2 to R1
(13)____________________________________________
; Decrement R3
(14)____________________________________________
; Continue loop
(15)____________________________________________
MULTIPLY_DONE
; Store the result in R1 back into head_ptr->price
(16)____________________________________________
; Move stack pointer to reserve space for arguments
(17)____________________________________________
; Load head_ptr->next into R1
(18)____________________________________________
; Put exchange_rate onto the stack as argument using R5
(19)____________________________________________
; Put head_ptr->next onto the stack as argument using R5
(20)____________________________________________
; recursively call convert_currency
(21)____________________________________________
; Pop return value and arguments off the stack
(22)____________________________________________

CALLEE_TEARDOWN
; Callee teardown
; Restore caller's frame pointer using R6
(23)____________________________________________f
; Restore return address
(24)____________________________________________
; Pop book keeping info
(25)____________________________________________
; Return to caller
RET
```

2. (30 points) **C++ Object Oriented Programming**

   Description: Grids, or structured meshes, are used in many scientific and engineering applications. They are used to represent physical domains and to discretize partial differential equations to have solutions approximated by computers. This problem will focus on sampling a scalar field at points specified by the grids implemented in C++. In this problem, you will implement multiple classes that represent different types of grids. The abstract class `Grid` is provided for you. You will implement the derived classes, as well as member functions for the `Position` class.

   Position:

   1. Represents a 2D position in space.
   2. The default constructor no arguments and defaults to the point (0,0).
   3. The second constructor takes the `x` and `y` coordinates as parameters.
   4. The `getX` and `getY` methods return the `x` and `y` coordinates, respectively.
   5. The operator `+` is overloaded to perform component-wise addition on two `Position` objects.
   6. The operator `-` is overloaded to perform component-wise subtraction on two `Position` objects.

   UniformGrid:

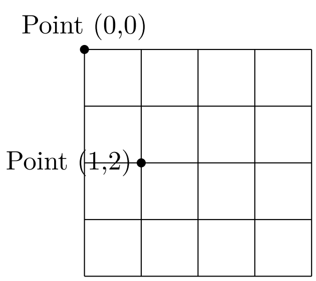
   1. Represents a uniform grid where all cells are of the same size.
   2. The constructor takes the number of rows, columns, spacing between grid points, and the origin point as parameters and will need to allocate memory used by the grid.
   3. The `setValue` method assigns a value to a specific cell based on its row and column indices.
   4. The `getValue` method retrieves the value from a specific cell.
   5. The `getPosition` method calculates the position of a grid point based on its row and column indices.

   CurvilinearGrid:

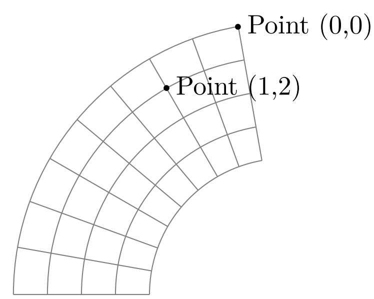
   1. Represents a curvilinear grid where cells can have arbitrary shapes.
   2. The constructor takes the number of rows, columns, and a list of `Position` objects that define the grid points.
   3. The `setValue` method sets the value of a specific cell based on its row and column indices.
   4. The `getValue` method retrieves the value from a specific cell.
   5. The `getPosition` method returns the position object for a specific cell.

The below figures show examples of Uniform and Curvilinear grids.

Uniform Grid



Curvilinear Grid



```
// imports omitted, assume correct
// 2d position class
class Position {
public:
  Position() {
    // initialize x and y to 0, omitted
  }
  Position(double x, double y) {
    // initialize x and y to the given values
    (1) ________________________;
    (2) ________________________;
  }
  double getX() const { return x_; }
  double getY() const { return y_; }
  Position operator+(const Position &p) const {
    Position ret = (3) _________________;
    return ret;
  }
  Position operator-(const Position &p) const {
    Position ret = (4) _________________;
    return ret;
  }
private:
  double x_, y_;
};

// abstract class for grids
class Grid {
public:
  virtual int getRows() const = 0;
  virtual int getCols() const = 0;
  virtual double getValue(int row, int col) const = 0;
  virtual Position getPosition(int row, int col) const = 0;
  virtual void setValue(int row, int col, double value) = 0;
  void print() { // omitted }
```

```
  virtual ~Grid() {}
};

// 2-d uniform grid class
// An axis-aligned uniform grid is a grid where all the cells have
// the same size and type.
class UniformGrid final : public Grid {
public:
  UniformGrid(int rows, int cols, double cellSize, Position p0) {
    rows_ = rows;
    cols_ = cols;
    cellSize_ = cellSize;
    p0_ = p0;
    grid = (5) __________________;
  }
  ~UniformGrid() { delete[] grid; }
  int getRows() const { return rows_; }
  int getCols() const { return cols_; }
  double getValue(int row, int col) const {return (6) ______________;}
  Position getPosition(int row, int col) const {
    // use operator overloading
    Position p1 = (7) __________________;
    return (8) __________________;
  }
  void setValue(int row, int col, double value) {
    (9) __________________;
  }
private:
  int rows_, cols_;  // number of rows and columns
  double *grid;    // 1-d array to store the 2-d grid
  double cellSize_; // size of each cell
  Position p0_;     // position of the top-left corner
};

// 2-d cuvilinear grid class
```

```
// A curvilinear grid is a grid where the cell verticies
// are freely specifiable.
class CurvilinearGrid final : public Grid {
public:
  CurvilinearGrid(const int rows, const int cols,
                  const std::list<Position> positions) {
    rows_ = rows;
    cols_ = cols;
    // initialize grid
    grid = (10) __________________;
    for (size_t i = 0; i < rows * cols; i++) {
      grid[i] = 0;
    }
    // initialize positions
    this->positions = (11) __________________;
    // copy positions argument to the grid class
    int i = 0;
    for ((12) __________; (13) __________; (14) __________) {
      (15) __________________;
    }
  }
  ~CurvilinearGrid() {
    delete[] grid;
    delete[] positions;
  }
  int getRows() const { return rows_; }
  int getCols() const { return cols_; }
  double getValue(int row, int col) const { (16) __________________; }
  Position getPosition(int row, int col) const {
    return (17) __________________;
  }
  void setValue(int row, int col, double value) {
    (18) __________________;
  }
private:
```

```
  int rows_, cols_;
  double *grid;
  Position *positions;
};

// create list of positions from csv file
std::list<Position> readPositions(const char *filename) {
  // omitted... assume no error
}

// evaluate scalar field at a position
// f(x, y) = x^2 + y^2
double eval_function(const Position &p) {
  return (19) __________________;
}

int main() {
  // create a uniform grid
  UniformGrid grid1(5, 5, 2.0, Position(-5, -5));

  // create a curvilinear grid
  std::list<Position> positions =
      readPositions("positions.csv"); // assume introduces no error
  CurvilinearGrid grid2(5, 5, positions);

  // create list of grids
  std::list<Grid *> grids;
  grids.push_back(&grid1);
  grids.push_back(&grid2);

  // sample scalar field on the grids
  for (Grid *grid : grids) {
    for (int i = 0; i < grid->getRows(); i++) {
      for (int j = 0; j < grid->getCols(); j++) {
        Position p = (20) __________________;
```

```
        double value = eval_function(p);
        // set value
        (21) __________________;
      }
    }
  }

  return 0;
}
```

Ben Bitdiddle is a student who is attempting to use the classes you have implemented above. He does not know what kind of grid he will be using, so he attempts to create a new `Grid` with the following code. What will happen when he tries to compile his code and why?

```
Grid *sampler = new Grid()
```

3. (20 points) **Find Professor**

You and the Professor were working late in the ECE220 lab. Suddenly, a quantum anomaly transported both of you into a mysterious maze. To escape, you must first find the Professor within the maze. Luckily, you have a rope that you can tie to the starting point to prevent getting lost. However, the rope has a limited length, which restricts how far you can explore the maze.

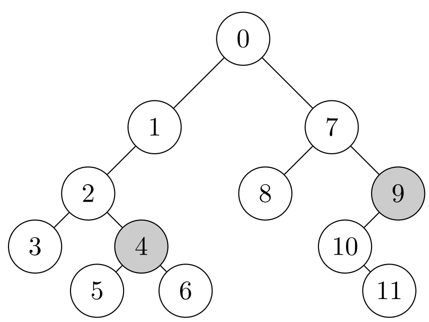
The maze is represented as a **binary tree**, where each node represents a room or passage. You start at a specific node in the tree and can explore the maze by moving to connected rooms (nodes). The rope length limits the maximum distance you can travel from your starting position.

Your task is to determine whether you can reach the Professor within the given rope length. Each node in the tree has the following attributes:

```
typedef struct Node {
  /* 1 (your location), 2 (professor location), otherwise 0 */
  int status;
  int visited; /* 1 if the node is visited, otherwise 0 */
  struct Node *left;    /* Left child  */
  struct Node *right;   /* Right child */
  struct Node * parent; /* Parent of node */
} Node;
```

In the tree, the node's `status` is 2 where professor is located, is 1 where you are located, and 0 otherwise. Your algorithm will traverse a node until it reached the limit of the rope or you find the Professor by setting `visited` attribute to 1 if you already visited.

Example:
In the following example, Node 4 is Your location and Node 9 is where Professor located. If we have a rope of length greater than or equal to 5 (≥ 5), then we can find Professor since the distance between Node4 and Node9 is 5.



Function to implement:
Implement the following function which traverses the binary tree and sets the `visited` attribute and distance as appropriate:

```
int findProfessor(Node *node, int *distance, int limit)
```

- `node`: the current node.
- `distance`: pointer to an integer holding the distance from the start node.
- `limit`: the length of the rope.

Algorithm details:

- Before the recursive calls, the function increases the distance.
- The return value of `findProfessor` is 1 if you find the professor and 0 otherwise

```
int findProfessor(Node *node, int *distance, int limit) {
    /* Base case: return 0 if it is not matched */
    /* (1) Reached a null node or
    *  (2) Already visited or (3) distance > limit */
    if ((1)________________________________________ ||
        (2)________________________________________ ||
        (3)________________________________________)
    {
        (4)________________________________________;
    }

    /* Found professor */
    if ((5)________________________________________) {
        (6)________________________________________;
    }
    /* Set visited */
    (7)________________________________________;

    /* Search in the left subtree
    * First increase distance */
    (8)________________________________________;

    /* Return 1 on success */
    if ((9)________________________________________) {
        return 1;
    }

    /*Search in the right subtree */
    if ((10)________________________________________) {
```

```
        (11)________________________________________;
    }


    /* Search upper part of the node */
    if ((12)________________________________________) {
        (13)________________________________________;
    }


    /* Undo increasing distance since ...  */
    (14)________________________________________;


    /* ... target was not found. */
    (15)________________________________________;
}


int find (Node *node, int limit) {
    /* ASSUME TREE IS ALREADY SETUP */
    int distance = 0;
    return findProfessor(node,
    (16)________________________________________, limit)
}
```


UIN: ____________________

4. (20 points) **Linked Lists**

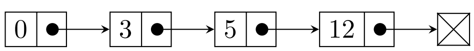
Asher has been assigned the task of rotating **k-length segments** in a linked list. In essence, the linked list is partitioned into sublists of k nodes and each sublist is rotated. Rotation means that the first node in that segment is moved to the end of the segment and other nodes are shifted appropriately to make space for it. He needs your help with writing the functions `rotate`, which rotates a linked list, and `subK_rotate`, which does the operation described.

Here is the algorithm for `subK_rotate`:

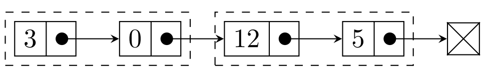
1. Check if there are no nodes, or if $k <= 1$. This means we do not need to run `subK_rotate` on the list.
2. Find the kth node in the list and separate the first k nodes from the rest of the list.
3. Call `rotate` on the first k nodes.
4. Recursively run `subK_rotate` on the remaining nodes and attach to the end of the rotated segment.
5. Return the new head of the list.

Here are some examples of what `subK_rotate` would do to a linked list. The dashed boxes show the rotated segments.

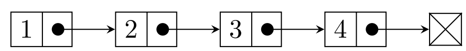
Example 1:



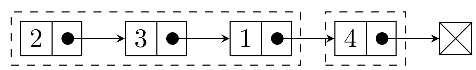
When calling `subK_rotate` with $k = 2$:



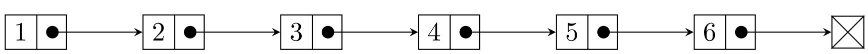
Example 2:



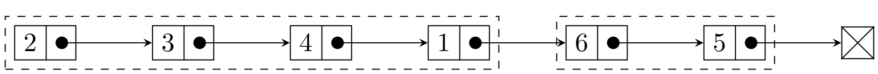
When calling `subK_rotate` with $k = 3$:



Example 3:



When calling `subK_rotate` with $k = 4$:



Note: Although $6 \rightarrow 5$ is not length k, it still gets rotated.

Complete the functions `rotate` and `subK_rotate`. The node struct is given on the next page.

Note: It is assumed that the caller of the `subK_rotate` function will allocate and free the memory for the nodes.

```
typedef struct node_struct{
  int data ;
  struct node_struct *next;
} node;
```

Part 1: Fill in the following code.

```
node* rotate(node* old_head) {
    node* temp = old_head;

    /* Set temp to the end of the list */
    while ((1)________________________________) {
           (2)________________________________;
    }
    /* Set the new end of the list to be old_head */
    (3)__________________________________________;
    /* Set temp to be the new start of the list */
    temp = (4)__________________________________;
    /* Clear old_head's next pointer */
    (5)_________________________________________;
    return temp;
}


/* Recursive function to rotate a linked list in k-length segments */
node* subK_rotate(node* head, int k) {
    int i;
    node* currNode = head;
    node* nextNode;


    /* No reversal is needed */
```

```
    if ((6)______________________ || (7)_____________________){
        return head;
    }

    /* Set currNode to the end of the segment to rotate */
    i = (8)________________;
    while ((9)__________________ && (10)_________________) {
        currNode = currNode->next;
        i++;
    }

    /* Detach the segment to be reversed from the rest of the list,
        keeping track of the rest of the list using nextNode */
    nextNode = currNode->next;
    (11)___________________________;

    /* Call the rotate function */
    head = (12)_________________________;

    /* If currNode is not the end of the segment */
    if((13)_______________________________){
    /* Set currNode to the end of the rotated segment */
       (14)________________________________;
    }
    /* Add the rest of the list recursively*/
    (15)__________________________________;

    return head;
}
```

Part 2: Shown below are two versions of a function to delete the first node of a linked list. Which version (A or B) is correct and why?

A

```
void del_first(node* head){
    node* temp;
    if(head == NULL){
        return;
    }

    temp = head->next;
    free(head);
    head = temp;
    return;
}
```

B

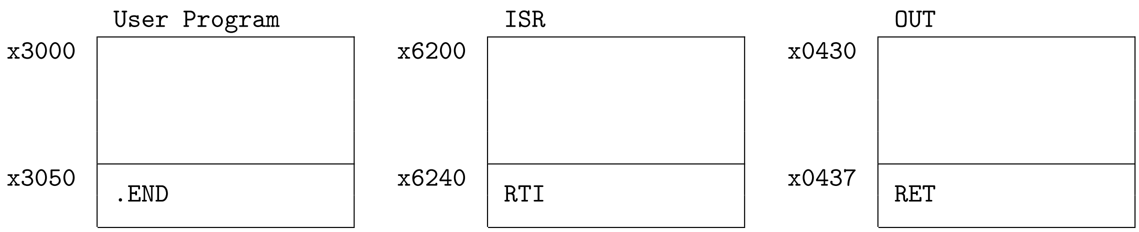
```
void del_first(node** head){
    node* temp;
    if(head == NULL
         || *head == NULL){
        return;
    }
    temp = (*head)->next;
    free(*head);
    *head = temp;
    return;
}
```

Which version (A or B) is correct? ____________

Why? (no more than 20 words):

5. (10 points) Concepts

   (a) The figures below shows the start and end of a user program, an interrupt service routine (`ISR`), and the `OUT` `TRAP` service routine.



The four figures below show snapshots of the values in `PC` (program counter), `R6` (Stack Pointer) and `R7` during the execution.

Snapshot 1	
PC	x3005
R6	x4007
R7	Irrelevant

Snapshot 2	
PC	x6210
R6	x2FF0
R7	Irrelevant

Snapshot 3	
PC	x4302
R6	x2FF0
R7	Irrelevant

Snapshot 4	
PC	x3010
R6	x4007
R7	Irrelevant

1. An interrupt occurred between Snapshot 1 and Snapshot 2. Why did the value in `R6` change? Limit your answer to no more than 20 words.

2. In Snapshot 4, how did `R6` recover its original value `x4007`? Limit your answer to no more than 20 words.

 UIN: ____________________

(b) Consider the following C program snippet.

```
int num = 10;

int main(){
    int value[2] = {15, 20};
    int* ptr_arr[2];
    ptr_arr[0] = (int*)malloc(sizeof(int)*num);
    ptr_arr[1] = (int*)malloc(sizeof(int)*num*2);

    /* assume both allocations are successful */
    *(ptr_arr[0]) = value[1];

    /* remaining code omitted for simplicity */
    return 0;
}
```

Identify the location in memory (program text, data section, heap, run-time stack) where these variables are being stored.

1. `num` Your answer: ______________________
2. `value` Your answer: ______________________
3. `ptr_arr` Your answer: ______________________
4. `*(ptr_arr[0])` Your answer: ______________________

Assume the code is running on a 32-bit system, and an integer is 32-bit long. Determine the size of the following variables when using `sizeof()` in `main`.

1. `sizeof(value)` = ______________ Bytes
2. `sizeof(ptr_arr)` = ______________ Bytes

 UIN: ______________________

LC-3 Instructions

NOTES: RTL corresponds to execution (after fetch!); JSRR not shown

Instruction	Encoding	Assembly	RTL
ADD	0001 | DR | SR1 | 0 | 00 | SR2	ADD DR, SR1, SR2	DR ← SR1 + SR2, Setcc
ADD	0001 | DR | SR1 | 1 | imm5	ADD DR, SR1, *imm5*	DR ← SR1 + SEXT(imm5), Setcc
AND	0101 | DR | SR1 | 0 | 00 | SR2	AND DR, SR1, SR2	DR ← SR1 AND SR2, Setcc
AND	0101 | DR | SR1 | 1 | imm5	AND DR, SR1, *imm5*	DR ← SR1 AND SEXT(imm5), Setcc
BR	0000 | n | z | p | PCoffset9	BR{nzp} *PCoffset9*	((n AND N) OR (z AND Z) OR (p AND P)): PC ← PC + SEXT(PCoffset9)
JMP	1100 | 000 | BaseR | 000000	JMP BaseR	PC ← BaseR
JSR	0100 | 1 | PCoffset11	JSR *PCoffset11*	R7 ← PC, PC ← PC + SEXT(PCoffset11)
TRAP	1111 | 0000 | trapvect8	TRAP *trapvect8*	R7 ← PC, PC ← M[ZEXT(trapvect8)]
LD	0010 | DR | PCoffset9	LD DR, *PCoffset9*	DR ← M[PC + SEXT(PCoffset9)], Setcc
LDI	1010 | DR | PCoffset9	LDI DR, *PCoffset9*	DR ← M[M[PC + SEXT(PCoffset9)]], Setcc
LDR	0110 | DR | BaseR | offset6	LDR DR, BaseR, *offset6*	DR ← M[BaseR + SEXT(offset6)], Setcc
LEA	1110 | DR | PCoffset9	LEA DR, *PCoffset9*	DR ← PC + SEXT(PCoffset9), Setcc
NOT	1001 | DR | SR | 111111	NOT DR, SR	DR ← NOT SR, Setcc
ST	0011 | SR | PCoffset9	ST SR, *PCoffset9*	M[PC + SEXT(PCoffset9)] ← SR
STI	1011 | SR | PCoffset9	STI SR, *PCoffset9*	M[M[PC + SEXT(PCoffset9)]] ← SR
STR	0111 | SR | BaseR | offset6	STR SR, BaseR, *offset6*	M[BaseR + SEXT(offset6)] ← SR

UIN: ____________________

width=!,height=!,scale=0.9, pages=-,pagecommand=

C++ Reference Card

C++ Data Types

Data Type	Description
`bool`	boolean (true or false)
`char`	character ('a', 'b', etc.)
`char[]`	character array (C-style string if `null` terminated)
`string`	C++ string (from the STL)
`int`	integer (1, 2, -1, 1000, etc.)
`long int`	long integer
`float`	single precision floating point
`double`	double precision floating point

These are the most commonly used types; this is not a complete list.

Operators

The most commonly used operators in order of precedence:

1	`++` (post-increment), `--` (post-decrement)
2	`!` (not), `++` (pre-increment), `--` (pre-decrement)
3	`*`, `/`, `%` (modulus)
4	`+`, `-`
5	`<`, `<=`, `>`, `>=`
6	`==` (equal-to), `!=` (not-equal-to)
7	`&&` (and)
8	`||` (or)
9	`=` (assignment), `*=`, `/=`, `%=`, `+=`, `-=`

Console Input/Output

`cout <<`	console out, printing to screen
`cin >>`	console in, reading from keyboard
`cerr <<`	console error

Example:
```
cout << "Enter an integer: ";
cin >> i;
cout << "Input: " << i << endl;
```

File Input/Output

Example (input):
```
ifstream inputFile;
inputFile.open("data.txt");
inputFile >> inputVariable;
// you can also use get (char) or
// getline (entire line) in addition to >>
  ...
inputFile.close();
```

Example (output):
```
ofstream outFile;
outfile.open("output.txt");
outFile << outputVariable;
  ...
outFile.close();
```

Decision Statements

if
```
if (expression)
    statement;
```
Example
```
if (x < y)
    cout << x;
```

if / else
```
if (expression)
    statement;
else
    statement;
```
Example
```
if (x < y)
    cout << x;
else
    cout << y;
```

switch / case
```
switch(int expression)
{
  case int-constant:
    statement(s);
    break;
  case int-constant:
    statement(s);
    break;
  default:
    statement;
}
```
Example
```
switch(choice)
{
  case 0:
    cout << "Zero";
    break;
  case 1:
    cout << "One";
    break;
  default:
    cout << "What?";
}
```

Looping

while Loop
```
while (expression)
    statement;

while (expression)
{
    statement;
    statement;
}
```
Example
```
while (x < 100)
    cout << x++ << endl;

while (x < 100)
{
    cout << x << endl;
    x++;
}
```

do-while Loop
```
do
    statement;
while (expression);

do
{
    statement;
    statement;
}
while (expression);
```
Example
```
do
    cout << x++ << endl;
while (x < 100);

do
{
    cout << x << endl;
    x++;
}
while (x < 100);
```

for Loop
```
for (initialization; test; update)
    statement;

for (initialization; test; update)
{
    statement;
    statement;
}
```

Example
```
for (count = 0; count < 10; count++)
{
    cout << "count equals: ";
    cout << count << endl;
}
```

Functions

Functions return at most one value. A function that does not return a value has a return type of `void`. Values needed by a function are called parameters.

```
return_type function(type p1, type p2, ...)
{
    statement;
    statement;
    ...
}
```

Examples
```
int timesTwo(int v)
{
    int d;
    d = v * 2;
    return d;
}

void printCourseNumber()
{
    cout << "CSE1284" << endl;
    return;
}
```

Passing Parameters by Value
```
return_type function(type p1)
```
Variable is passed into the function but changes to *p1* are not passed back.

Passing Parameters by Reference
```
return_type function(type &p1)
```
Variable is passed into the function and changes to *p1* are passed back.

Default Parameter Values
```
return_type function(type p1=val)
```
val is used as the value of *p1* if the function is called without a parameter.

Pointers

A pointer variable (or just pointer) is a variable that stores a memory address. Pointers allow the indirect manipulation of data stored in memory.

Pointers are declared using `*`. To set a pointer's value to the address of another variable, use the `&` operator.

Example
```
char c = 'a';
char* cPtr;
cPtr = &c;
```

Use the indirection operator (`*`) to access or change the value that the pointer references.

Example
```
// continued from example above
*cPtr = 'b';
cout << *cPtr << endl;  // prints the char b
cout << c << endl;      // prints the char b
```

Array names can be used as constant pointers, and pointers can be used as array names.

Example
```
int numbers[]={10, 20, 30, 40, 50};
int* numPtr = numbers;
cout << numbers[0] << endl;     // prints 10
cout << *numPtr << endl;        // prints 10
cout << numbers[1] << endl;     // prints 20
cout << *(numPtr + 1) << endl;  // prints 20
cout << numPtr[2] << endl;      // prints 30
```

Dynamic Memory

Allocate Memory
```
ptr = new type;

ptr = new type[size];
```
Examples
```
int* iPtr;
iPtr = new int;

int* intArray;
intArray = new int[5];
```

Deallocate Memory
```
delete ptr;
delete [] ptr;
```
Examples
```
delete iPtr;
delete [] intArray;
```

Once a pointer is used to allocate the memory for an array, array notation can be used to access the array locations.

Example
```
int* intArray;
intArray = new int[5];
intArray[0] = 23;
intArray[1] = 32;
```

Structures

Declaration
```
struct name
{
  type1 element1;
  type2 element2;
};
```
Example
```
struct Hamburger
{
  int patties;
  bool cheese;
};
```

Definition
```
name varName;

name* ptrName;
```
Example
```
Hamburger h;

Hamburger* hPtr;
hPtr = &h;
```

Accessing Members
```
varName.element=val;

ptrName->element=val;
```
Example
```
h.patties = 2;
h.cheese = true;

hPtr->patties = 1;
hPtr->cheese = false;
```

Structures can be used just like the built-in data types in arrays.

Classes

Declaration

```
class classname
{
public:
  classname(params);
  ~classname();
  type member1;
  type member2;
protected:
  type member3;
private:
  type member4;
};
```

Example

```
class Square
{
public:
  Square();
  Square(float w);
  void setWidth(float w);
  float getArea();
private:
  float width;
};
```

public members are accessible from anywhere the class is visible.

private members are only accessible from the same class or a friend (function or class).

protected members are accessible from the same class, derived classes, or a friend (function or class).

constructors may be overloaded just like any other function. You can define two or more constructors as long as each constructor has a different parameter list.

Definition of Member Functions

```
return_type classname::functionName(params)
{
    statements;
}
```

Examples

```
Square::Square()
{
    width = 0;
}

void Square::setWidth(float w)
{
    if (w >= 0)
      width = w;
    else
      exit(-1);
}

float Square::getArea()
{
    return width*width;
}
```

Definition of Instances

```
classname varName;
classname* ptrName;
```

Example

```
Square s1();
Square s2(3.5);

Square* sPtr;
sPtr=new Square(1.8);
```

Accessing Members

```
varName.member=val;
varName.member();

ptrName->member=val;
ptrName->member();
```

Example

```
s1.setWidth(1.5);
cout << s.getArea();

cout<<sPtr->getArea();
```

Inheritance

Inheritance allows a new class to be based on an existing class. The new class inherits all the member variables and functions (except the constructors and destructor) of the class it is based on.

Example

```
class Student
{
public:
  Student(string n, string id);
  void print();
protected:
  string name;
  string netID;
};

class GradStudent : public Student
{
public:
  GradStudent(string n, string id,
               string prev);
  void print();
protected:
  string prevDegree;
};
```

Visibility of Members after Inheritance

Inheritance Specification	Access Specifier in Base Class: private	protected	public
private	-	*private*	*private*
protected	-	*protected*	*protected*
public	-	*protected*	*public*

Operator Overloading

C++ allows you to define how standard operators (+, -, *, etc.) work with classes that you write. For example, to use the operator + with your class, you would write a function named `operator+` for your class.

Example

Prototype for a function that overloads + for the `Square` class:

```
Square operator+ (const Square &);
```

If the object that receives the function call is not an instance of a class that you wrote, write the function as a friend of your class. This is standard practice for overloading << and >>.

Example

Prototype for a function that overloads << for the `Square` class:

```
friend ostream & operator<<
         (ostream &, const Square &);
```

Make sure the return type of the overloaded function matches what C++ programmers expect. The return type of relational operators (<, >, ==, etc.) should be `bool`, the return type of << should be `ostream &`, etc.

Exceptions

Example

```
try
{
  // code here calls functions that might
  // throw exceptions
  quotient = divide(num1, num2);

  // or this code might test and throw
  // exceptions directly
  if (num3 < 0)
    throw -1;  // exception to be thrown can
               // be a value or an object
}
catch (int)
{
  cout << "num3 can not be negative!";
  exit(-1);
}
catch (char* exceptionString)
{
  cout << exceptionString;
  exit(-2);
}
//  add more catch blocks as needed
```

Function Templates

Example

```
template <class T>
T getMax(T a, T b)
{
  if (a>b)
    return a;
  else
    return b;
}

// example calls to the function template
int a=9, b=2, c;
c = getMax(a, b);

float f=5.3, g=9.7, h;
h = getMax(f, g);
```

Class Templates

Example

```
template <class T>
class Point
{
public:
  Point(T x, T y);
  void print();
  double distance(Point<T> p);
private:
  T x;
  T y;
};

// examples using the class template
Point<int> p1(3, 2);
Point<float> p2(3.5, 2.5);
p1.print();
p2.print();
```

Suggested Websites

C++ Reference: http://www.cppreference.com/ http://www.informit.com/guides/guide.aspx?g=cplusplus

C++ Tutorial: http://www.cplusplus.com/doc/tutorial/ http://www.sparknotes.com/cs/

C++ Examples: http://www.fredosaurus.com/notes-cpp/

Gaddis Textbook:

Video Notes http://media.pearsoncmg.com/aw/aw_gaddis_sowcso_6/videos

Source Code ftp://ftp.aw.com/cseng/authors/gaddis/CCSO5 (5th edition)

Developed for Mississippi State University's CSE1284 and CSE1384 courses February 17, 2009

Download the latest version from http://cse.msstate.edu/~crumpton/reference

UIN: ______________________