CSCI 520
Information Structures

Review

- Programming Style
- Variables
- Arrays
- Pointers
- Structures
- Dynamic memory allocation
Programming Style

- Indentation
  - Good

```cpp
if (nInput < nMax)
{
    if (nInput > nMin)
        if (nInput > nMedian)
            cout << nInput;
    else
    {
        cout << nInput;
    }
}
```

Programming Style

- Indentation
  - Wrong

```cpp
if (nInput < nMax)
{
    if (nInput > nMin)
        if (nInput > nMedian)
            cout << nInput;
    else
    {
        cout << nInput;
    }
```
Programming Style

- Comments
- Coding Style Conventions
- Hungarian Notation
  - E.g., nInput, cType, szName[], *pName
- Blank space both size of operators
  - E.g., int a = 2; (0)
    int a = 2; (X)

Variables

- A name used to refer to a certain location in memory
- C programming requires that variables must be declared with the variable type
  - int: an integer, typically reflecting the natural size of integers on the host machine
  - char: a single byte, capable of holding one character
  - float: single-precision floating point
  - double: double-precision floating point

Variables

- **Primitive Data Types in C**
  - short (unsigned short)
  - int (unsigned int)
  - long (unsigned long)
  - char
  - float
  - double

Fundamentals of Computer Systems

- **Bit: zero or one**
  - Morse Code, Punched Card, Electrical Switch, Two distinct Voltage or current levels

- **1 Byte = 8 bits**
  - historically for a single character
  - A unit to access a memory
  - Data can be accessed 8 bits or multiples of 8 bits at a time

- **16 bit vs 32 bits vs 64 bits computers?**
  - Processor registers (small amount of storage in a CPU)
    - Data, Address, General purpose, Conditional, and so on..
Data representation

- **Character (1 byte)**
  - **ASCII code** (American Standard Code for Information Interchange)

<table>
<thead>
<tr>
<th>DEC</th>
<th>OCT</th>
<th>HEX</th>
<th>BIN</th>
<th>Symbol</th>
<th>HTML Number</th>
<th>HTML Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>040</td>
<td>20</td>
<td>00100000</td>
<td>Space</td>
<td>&amp;#32;</td>
<td></td>
<td>Space</td>
</tr>
<tr>
<td>33</td>
<td>041</td>
<td>21</td>
<td>00100001</td>
<td>!</td>
<td>&amp;#33;</td>
<td></td>
<td>Exclamation mark</td>
</tr>
<tr>
<td>35</td>
<td>043</td>
<td>23</td>
<td>00100011</td>
<td>#</td>
<td>&amp;#35;</td>
<td></td>
<td>Number</td>
</tr>
<tr>
<td>36</td>
<td>044</td>
<td>24</td>
<td>00100100</td>
<td>$</td>
<td>&amp;#36;</td>
<td></td>
<td>Dollar</td>
</tr>
<tr>
<td>37</td>
<td>045</td>
<td>25</td>
<td>00100101</td>
<td>%</td>
<td>&amp;#37;</td>
<td></td>
<td>Procenttecken</td>
</tr>
</tbody>
</table>

ASCII code Table: [http://www.ascii-code.com/](http://www.ascii-code.com/)

- **Integer**
  - 2 bytes in 16 bits systems
  - 4 bytes in 32/64 bits systems
  - Unsigned integer: $0 \sim 2^n - 1$
  - Signed integer (two’s complement): $-2^{n-1} \sim 2^{n-1} - 1$

Two’s complement: [http://en.wikipedia.org/wiki/Two%27s_complement](http://en.wikipedia.org/wiki/Two%27s_complement)
Arrays

- Multiple values of the same data type can be stored with one variable name.
- Arrays are used to store collections of related data
- Can be multidimensional arrays

Vartype VariableName[Size1][Size2]...[SizeN]

The sizeof() an Array

- The total amount of memory used by the array

Example)

```c
int rgnExample[6];
sizeof(rgnExample) \rightarrow returns 24
//since six integers occupy 6 * 4 = 24 bytes
sizeof(rgnExample) / sizeof(int) \rightarrow returns 6
```
Initialization of Arrays

- **Initialize Arrays**

  \[
  \text{Type name}[N] = \{\text{Val1, Val2, …, ValN}\};
  \]

  **E.g.**

  int rgnIntegers[5] = {0, 1, 2, 3, 4};
  float rgfExample[3] = {0.11, 0.152, 2.4};
  int rgnIntegers[] = {0, 1, 2, 3, 4, 5, 6};

Arrays as Parameters to Functions

- An array can be passed as a parameter to a function
- The array must be declared in the function header

  \[
  \text{VarType function_name}(…, \text{type variable}[], …)
  \]
Arrays as Parameters to Functions

Example)

```cpp
void print_array(int rgnExample[])
{
    int i;
    for(i = 0; i < sizeof(rgnExample)/sizeof(int); i++)
        cout << rgnExample[i] << " ";
}

int main(void)
{
    int rgnExample[] = {101, 102, 103, 104};
    print_array(rgnExample);
    return 0;
}
```

Arrays as Parameters to Functions

Example)

```cpp
void print_array(int rgnExample[], int nSize)
{
    int i;
    for(i = 0; i < nSize; i++)
        cout << rgnExample[i] << " ";
}

int main(void)
{
    int rgnExample[] = {101, 102, 103, 104};
    print_array(rgnExample, 4);
    return 0;
}
```
Multi-Dimensional Arrays

- Supports multidimensional arrays.
- Two-Dimensional Arrays

```c
type name[N1][N2];
```

<table>
<thead>
<tr>
<th></th>
<th>Column 0</th>
<th>Column 1</th>
<th>Column 2</th>
<th>Column N2-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 0</td>
<td>name[0][0]</td>
<td>name[0][1]</td>
<td>name[0][2]</td>
<td>...</td>
</tr>
<tr>
<td>name[0][0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row 1</td>
<td>name[1][0]</td>
<td>name[1][1]</td>
<td>name[1][2]</td>
<td>...</td>
</tr>
<tr>
<td>name[1][0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row N1-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Initialization

```c
int rgnA[3][4] = {
    {0, 1, 2, 3}, /* initializers for row indexed by 0 */
    {4, 5, 6, 7}, /* initializers for row indexed by 1 */
    {8, 9, 10, 11} /* initializers for row indexed by 2 */
};
```

Equivalently,

```c
int rgnA[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```
Pointers

- A pointer variable stores the address of a memory location.
- In C/C++, pointers are considered to be separate data types.
  - Each of the data types has a corresponding pointer data type.
  - E.g.,
    
    ```
    char → char *
    int → int *
    ```

Pointers

- Declaration

  ```
  Data_type *Variable_Name
  ```

- Example:
  ```
  int *pnInput;
  char *pcInput;
  float *pfVariable;
  ```
Pointers

```c
int nInt;
int *pnInt = &nInt;
// &nInt refers to the address of the variable nInt
cout << "The values of nInt are stored in the memory, " << endl;
cout << "location " << &nInt << endl;
cout << "pnInt points " << pnInt << endl;
cout << "sizeof(pnInt)=" << sizeof(pnInt) << endl;
```

Output:
The values of nInt are stored in the memory location 0022FF18.
pnInt points 0022FF18.
sizeof(pnInt)=4.    // 4 on 32bits, 8 on 64bits

Tip

- How to compile codes for a 32 bit or 64 bit environment in C
  - `gcc -m64 c_file_name -o object_file_name`    // 64 bits
  - `gcc -m32 c_file_name -o object_file_name`    // 32 bits

- Check file type
  - `file file_name`
Value of a pointer variable

- When retrieving or changing values in the memory space that a pointer is pointing, we use dereference operator `*`.

```
char cInput, *pcInput = &cInput;
cInput = 'Q';
cout << "The value of cInput is " << cInput << endl;
cout << "The character " << cInput << " is store in hex "
    << (void*)pcInput << endl;
cout << "The value store in " << (void*)pcInput << " is "
    << *pcInput << endl;

*pcInput = 'R';
cout << "The value of cInput is " << cInput << endl;
cout << "The character " << cInput << " is store in hex "
    << (void*)pcInput << endl;
cout << "The value store in " << (void*)pcInput << " is "
    << *pcInput << endl;
```
Pointer Initialization

- **Initialization with assignment statement**
  - E.g.
    ```c
    int nVar, *pnVar, *pnVar2;
    pnVar = &nVar;
    pnVar2 = pnVar;
    pnVar = (int *) 1000;
    ```

- The identifier NULL (defined to be zero) is used to indicate that the pointer does not point at a legal memory address.

  ```c
  int *pnVar = NULL;
  if (pnVar != NULL)
      do_something();
  if (pnVar)
      do_something();
  ```
An Array as a Pointer

- A variable of an array points the first cell of the memory location.

- E.g.,
  
  ```
  int rgnArray[10], *pnArray = NULL;
  pnArray = rgnArray;
  ```

Pointer Arithmetic

- Pointer +(or -) an integer (N)
  - Move to the next pointer
  - The next address is Pointer + (or -) sizeof(type of the pointer) * N

```
int rgnArray = {10, 11, 12};
int *pnArray = rgnArray;
```
Pointer Arithmetic

```cpp
char szString[] = "This is a String."
char *pszString = szString;
char *pszPointer = szString;
int i = 0;
for (i = 0; i < strlen(szString); i++)
{
    cout << szString[i] << " " << *(pszString + i) << " " << *pszPointer << endl;
    pszPointer++;
}
```

Pointer Arrays

- Can store multiple pointers by using an array

```
data_type *variable_name[Size];
```

- e.g.,

```
int *pnName[10];
char *pszName[10];
```
**Pointer Arrays**

```c
const int MAX = 3;
int main ()
{
    int var[] = {10, 100, 200};
    int i, *pnVar[MAX];
    for ( i = 0; i < MAX; i++)
    {
        pnVar[i] = &var[i]; /* assign the address of integer. */
    }
    for ( i = 0; i < MAX; i++)
    {
        cout << "Value of var[" << i << "] = " << *pnVar[i] << endl;
    }
    return 0;
}
```

---

**Pointer Arrays**

```c
#include <stdio.h>
const int MAX = 4;

int main ()
{
    char *names[] = {
        "Zara John",
        "Hina Ali",
        "Nuha Thomson",
        "Sara William"};
    int i = 0;
    for ( i = 0; i < MAX; i++)
    {
        cout << "Value of names[" << i << "] = " << names[i] << endl;
    }
    return 0;
}
```
Passing Pointer Parameters to Functions

- Parameters could be passed by reference or passed by value.
- Pointer Parameters will be call by reference approach.

```c
void Add1(int nA, int *pnB)
{
    nA = nA + 1;
    *pnB = *pnB + 1;
}
int main(void)
{
    int nValueA = 5, nValueB = 10;
    int *pnValueB = &nValueB;
    cout << nValueA << " " << nValueB << endl;
    Add1(nValueA, &nValueB);
    cout << nValueA << " " << nValueB << endl;
    Add1(nValueA, pnValueB);
    cout << nValueA << " " << nValueB << endl;
    return 0;
}
```
Structures

- In contrast to an array, a structure can concurrently hold multiple data values of different types.

- For example,
  - Student: Last Name, First Name, DOB, Gender,…
  - Car: Model Name, Manufacturer, Year, Price, …. 

Structures

- Declaration

```c
struct struct_name
{
    type1 type_name1;
    type2 type_name2;
    ...
} variable_name1, ..., variable_nameN;
```
Structures

Example:

```c
struct student
{
    int nID;
    char szName[100];
    int nBirthYear;
    int nGrade;
} student1, student2;
```

Structures

- A variable of a structure type
  - `struct structure_name  variable_name;`

- Initialization
  - `struct student student1 = {101, "John", 1995, 3};`
The individual fields of a structure can be accessed with the syntax

```
variable_name.member_name
```

```c
struct student
{
    int nID;
    char szName[100];
    int nBirthYear;
    int nGrade;
};

int main(void)
{
    struct student student1 = {101, "John", 1995, 3};
    printf("ID: %d\nName:%s\nYear:%d\nGrade:%d\n", student1.nID,
            student1.szName, student1.nBirthYear,student1.nGrade);
    return 0;
}
```
Structures

- `sizeof(structure_variable)`
  - Returns amount of the memory space that the structure allocates

```c
printf("%d", sizeof(student1));
```

See Structures1.c

Structures as Parameters to Functions

- A structure can be passed as a parameter to a function
- Passing structures to functions follows Call by a Value
Structures as Parameters to Functions

Void TryToChangeValue(struct student student1)
{
    student1.nBirthYear = 2000;
}

int main(void){
    struct student student1 = {101, "John", 1995, 3};
    printf("Before: \nID: %d \nName: %s \nYear: %d \nGrade: %d \n", student1.nID, 
            student1.szName, student1.nBirthYear, student1.nGrade);
    TryToChangeValue(student1);
    printf("\nAfter: \nID: %d \nName: %s \nYear: %d \nGrade: %d \n", student1.nID, 
            student1.szName, student1.nBirthYear, student1.nGrade);
    return 0;
}

See Structures2.c

Returning a structure in a function

- A structure can be a returning type in a function.

struct student AddStudent(int nID, char szName[],
                           int nBirthYear, int nGrade)
{
    struct student student1;
    ...
    return student1;
}

See Structures3.c
Structure Pointer

```c
struct student
{
    int nID;
    char szName[100];
    int nBirthYear;
    int nGrade;
};
struct student student1;
struct student *pStudent1 = &student1;
printf("ID is %d", pStudent1->nID);
printf("ID is %d", *(pStudent1->nID));
```

See Structures4.c

Structure Pointer in a Function

```c
void StructPointerinFunction(struct student *student1)
{
    student1->nBirthYear = 2000;
}
int main(void)
{
    struct student student1 = {101, "John", 1995, 3};
    printf("Before: \nID: %d \nName:%s \nYear:%d \nGrade:%d \n", 
           student1.nID, student1.szName, 
           student1.nBirthYear, student1.nGrade);
    StructPointerinFunction(&student1);
    printf("\nAfter: \nID: %d \nName:%s \nYear:%d \nGrade:%d \n", 
           student1.nID, student1.szName, 
           student1.nBirthYear, student1.nGrade);
}
```

See Structures5.c
Structures with Arrays

- A array of a structure

```c
struct student StudentCSCI520[50];

StudentCSCI520[0].szName
StudentCSCI520[0].nBirthYear
```

---

Structures with Arrays

- Initialization

```c
struct student StudentCSCI520[3] = {
    {102, “Mary”, 1990, 4},
    {103, “Tom”, 1992, 1}};
```
Structures with Arrays

```c
struct student StudentCSCI520[3] = {
    {101, "John", 1995, 3},
    {102, "Mary", 1990, 4},
    {103, "Tom", 1992, 1}};

int i;

for (i = 0; i < 3; i++)
{
    printf("ID: %d
Name:%s
Year:%d
Grade:%d"
, 
StudentCSCI520[i].nID, StudentCSCI520[i].szName, 
StudentCSCI520[i].nBirthYear, StudentCSCI520[i].nGrade);
}
```

See Structures6.c