Structured Data

CIS 15 : Spring 2007
Functionalia

HW4 Part A due this SUNDAY April 1st: 11:59pm
Reminder: I do NOT accept LATE HOMEWORK.

Today:

• Dynamic Memory Allocation
• Allocating Arrays
• Returning Pointers from Functions
• Structured Data
Variables can be created and destroyed

While a program is running, variables (and arrays of variables) can be created on the fly through dynamic memory allocation.

The program asks the computer for an unused chunk of memory with the size of the variable requested.

The computer returns the starting address of the chunk of memory. This starting address is stored in a pointer.

Use the new operator along with the type of the variable you want.

```cpp
int * ptr;
ptr = new int;
*ptr = 25;
cout << *ptr;
cin >> *ptr;
(*ptr)++;
```
Dynamically Allocating Arrays

Dynamically Allocating Arrays is the more common use of the new operator:

```cpp
int * ptr;
p = new int[100]; // creates a 100 integer sized array
```

Use `ptr` the same way one would use an integer array name.

```cpp
for(int i = 0; i < 100; i++)
    ptr[i] = 1;
```
Dynamically Allocating Arrays

Not limited to only using integers.

```c
char * cPtr;
cPtr = new char[27]; // creates a 27 character sized array
```

What is stored in the cPtr array?

```c
for(int i = 0; i < 26; i++)
    cPtr[i] = 'a' + i;
cPtr[26] = '\0';
```
Dynamically Allocating Arrays

Not limited to only using integers.

```c
char * cPtr;
cPtr = new char[27]; // creates a 27 character sized array
```

**What is stored in the cPtr array?**

```c
for(int i = 0; i < 26; i++)
    cPtr[i] = 'a' + i;
cPtr[26] = '\0';
```

NULL byte at the end

`abcdefghijklmnopqrstuvwxyz`
Memory is finite

```cpp
int * wholeLottaMemory = new int[1000000000000000000000000];
```

What happens when the computer runs out of memory?

1. Throws an Exception (Error Handling in C++)
2. Returns memory address 0 (also known as `NULL`)

```cpp
int * ptr = new int[100];
if(ptr == NULL)
{
    cout << "Error allocating memory\n";
    return;
}
```
What is created must be deleted!

When your program is finished using dynamically allocated memory, it must **free** the memory for future use.

```cpp
int * ptr;
ptr = new int;
*ptr = 5;
*ptr = *ptr + 2;
...
delete ptr;
```

Free the memory by using the **delete** operator
Pointers are able to be reused

`delete` does not remove the pointer. It only FREES the memory that it points to.

```cpp
int * ptr;
ptr = new int;
*ptr = 5;
*ptr = *ptr + 2;
delete ptr;
```

Always free the memory that you dynamically allocate.

```cpp
ptr = new int;
*ptr = 3;
delete ptr;
```

C++ does not do garbage collection.
Deleting Arrays

To delete dynamically allocated arrays, need to add the `[]` symbol.

```c
char * cPtr;
cPtr = new char[27]; // creates a 27 character sized array
...
delete [] cPtr;
```

Not deleting dynamically allocated memory creates *memory leaks*. (And results in sluggish and failing programs).
Always check for **NULL**

NULL points to memory address 0.

Not a usable address. Operating system data is stored in the lower memory address space.

Always check if a pointer is pointing to NULL.

When a pointer is not being used any more. Set it to NULL.

```cpp
char * cPtr;
cPtr = new char[27]; // creates a 27 character sized array
...

delete [] cPtr;
cPtr = NULL;
```
What's wrong?

**Note:** Functions can return pointers (Take a look at all C String Functions).

```cpp
char * getName()
{
    char name[81];
    cout << "Enter your name: ";
    cin.getline(name, 81);
    return name;
}
```
What's wrong?

**Note**: Functions can return pointers (Take a look at all C String Functions).

```cpp
char * getName()
{
    char name[81];
    cout << "Enter your name: ";
    cin.getline(name, 81);
    return name;
}
```

(name is a local variable to the function. Exists only within the scope of the function.)
Dynamically Allocate the Memory

Functions themselves can dynamically allocate memory, return a pointer to the memory, and the memory sticks around beyond the scope of the function.

```cpp
char * getName()
{
    char * name;
    name = new char[81];
    cout << "Enter your name: ";
    cin.getline(name, 81);
    return name;
}

char * yourName = getName();
cout << yourName << endl;
delete [] yourName;
```
Dynamically Allocate the Memory

This is how MEMORY LEAKS can happen. When you lose track of memory and forget to free it.

```cpp
char * getName()
{
    char * name;
    name = new char[81];
    cout << "Enter your name: ";
    cin.getline(name, 81);
    return name;
}
```

```cpp
char * yourName = getName();
cout << yourName << endl;
delete [] yourName;
```
Exercises

1. Assume that \( \text{ip} \) is a pointer to an \( \text{int} \). Write a statement that will dynamically allocate an integer variable and store its address in \( \text{ip} \). Write a statement that will free the memory allocated to \( \text{ip} \).

2. Assume \( \text{ip} \) is a pointer to an \( \text{int} \). Then, write a statement that will dynamically allocate an array of 500 integers and store its address in \( \text{ip} \). Write a statement that will free the memory allocated in the statement you just wrote.
1. Assume that \texttt{ip} is a pointer to an \texttt{int}. Write a statement that will dynamically allocate an integer variable and store its address in \texttt{ip}. Write a statement that will free the memory allocated to \texttt{ip}.

\begin{verbatim}
int * ip;
ip = new int;
*ip = 255;
delete ip;
\end{verbatim}

2. Assume \texttt{ip} is a pointer to an \texttt{int}. Then, write a statement that will dynamically allocate an array of 500 integers and store its address in \texttt{ip}. Write a statement that will free the memory allocated in the statement you just wrote.
Exercises

1. Assume that \texttt{ip} is a pointer to an \texttt{int}. Write a statement that will dynamically allocate an integer variable and store its address in \texttt{ip}. Write a statement that will free the memory allocated to \texttt{ip}.

2. Assume \texttt{ip} is a pointer to an \texttt{int}. Then, write a statement that will dynamically allocate an array of 500 integers and store its address in \texttt{ip}. Write a statement that will free the memory allocated in the statement you just wrote.

\begin{verbatim}
int * ip;
ip = new int[500];
for(int i = 0; i < 500; i++)
    *(ip + i) = 255;
delete [] ip;
\end{verbatim}
## Primitive Data Types

So far (with the exception of learning a little bit of Classes in 1.5), the data types you’re accustomed to are:

<table>
<thead>
<tr>
<th></th>
<th>bool</th>
<th>int</th>
<th>unsigned long int</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>long int</td>
<td></td>
<td></td>
<td>float</td>
</tr>
<tr>
<td>unsigned char</td>
<td>unsigned short int</td>
<td></td>
<td>double</td>
</tr>
<tr>
<td>short int</td>
<td></td>
<td>unsigned int</td>
<td>long double</td>
</tr>
</tbody>
</table>
Structured Data

To provide a level of **Abstraction**, C++ allows you to group several variables together into a single item known as structure.

**What is Abstraction?**

A struct is similar to a class, but more simple, in that it abstracts only **data**, and not **functions**.

An **array** allows one to package variables and data together, but what is its limitation?
Here is a structure (called Time) that contains 3 integers (hour, minutes, and seconds).

```c
struct Time
{
    int hour;
    int minutes;
    int seconds;
};

Time now;
```
Anatomy of a struct

Here is a structure (called `Time`) that contains 3 integers (hour, minutes, and seconds).

```
struct Time {
    int hour;
    int minutes;
    int seconds;
};
```

Keyword `struct` to indicate that what follows is a struct.

Time now;
Anatomy of a struct

Here is a structure (called \texttt{Time}) that contains 3 integers (hour, minutes, and seconds).

\begin{verbatim}
struct Time {
    int hour;
    int minutes;
    int seconds;
};
\end{verbatim}

\texttt{Time} now;
Here is a structure (called Time) that contains 3 integers (hour, minutes, and seconds).

```c
struct Time {
    int hour;
    int minutes;
    int seconds;
};
```

It is customary to always Capitalize the first letter of the name of a structure.

Time now;
Anatomy of a struct

Here is a structure (called `Time`) that contains 3 integers (hour, minutes, and seconds).

```c
struct Time {
    int hour;
    int minutes;
    int seconds;
};
```

The member data types are contained in curly braces (just like functions)

```c
Time now;
```
Anatomy of a struct

Here is a structure (called `Time`) that contains 3 integers (hour, minutes, and seconds).

```c
struct Time {
    int hour;
    int minutes;
    int seconds;
};
```

NOTE! There is a semi-colon at the end of the struct (unlike functions)

```c
Time now;
```
Anatomy of a struct

Here is a structure (called `Time`) that contains 3 integers (`hour`, `minutes`, and `seconds`).

```c
struct Time {
    int hour;
    int minutes;
    int seconds;
};
```

Variables are declared like in any function.

```c
Time now;
```
Here is a structure (called Time) that contains 3 integers (hour, minutes, and seconds).

```c
struct Time
{
    int hour;
    int minutes;
    int seconds;
};
```

Your structure can be declared now as any other variable.
Anatomy of a struct

Here is a structure (called `Time`) that contains 3 integers (`hour`, `minutes`, and `seconds`).

```c
struct Time {
    int hour;
    int minutes;
    int seconds;
};
```

This is the **definition**.
(Only one).

This is the **instantiation**.
(Can be many).

```
Time now;
Time later;
```
Definition of structs

Typically struct’s are defined outside of any functions, and at the top of the program (i.e. near the prototypes). Why?

```c
struct Date
{
    int day, month, year;
    char longName[255];
};

int main() {
    Date current;
}
```
Definition of structs

Note the mixture of data-types, and using one line for the integers.

```c
struct Date
{
    int day, month, year;
    char longName[255];
};

int main()
{
    Date current;
}
```
Accessing the members of a struct

You can access the members of a struct variable through **dot-notation**.

```c
struct Date
{
    int day, month, year;
    char longName[255];
};

int main() {
    Date current;
    current.day = 26;
    current.month = 3;
    current.year = 2007;
    strcpy(current.longName, "Bangladesh - Independence Day");
}
```
Accessing the members of a struct

Why won’t this work?

```cpp
struct Date {
    int day, month, year;
    char longName[255];
};

int main() {
    Date current;
    cout << "Enter the current date: ";
    cin >> current;
    cout << current << endl;
}
```
struct CityInfo
{
    char cityName[30];
    char state[3];
    long population;
    int distance;
};

CityInfo location = {“Asheville”, “NC”, 50000, 28};
Initializing Data in Structs

```c
struct CityInfo
{
    char cityName[30];
    char state[3];
    long population;
    int distance;
};

CityInfo location = {"Asheville", "NC", 50000, 28};
```
struct CityInfo
{
    char cityName[30];
    char state[3];
    long population;
    int distance;
};

CityInfo location = {"Tampa"];
Initializing Data in Structs

```c
struct CityInfo
{
    char cityName[30];
    char state[3];
    long population;
    int distance;
};

CityInfo location = {"Atlanta", "GA"};
```
struct CityInfo
{
    char cityName[30];
    char state[3];
    long population;
    int distance;
};

CityInfo location = {"Knoxville", "TN", , 90};
Initializing Data in Structs

```c
struct CityInfo
{
    char cityName[30];
    char state[3];
    long population;
    int distance;
};

CityInfo location = {"Knoxville", "TN", , 90};
Illegal!!!
```
Arrays of Structs

```cpp
struct BookInfo
{
    char title[50];
    char author[30];
    char publisher[25];
    double price;
};

BookInfo bookList[20]; // creates 20 BookInfo’s

cout << bookList[10].title << endl;
cout << bookList[10].title[0] << endl;
```

What’s the difference?
Structs can be nested

```c
struct Rectangle
{
    int length;
    int width;
};

struct House
{
    int height;
    Rectangle footprint;
};

House mine;
mine.footprint.length = 20;
mine.footprint.width = 40;
```