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

Const variables need Pointers to a Const

```
const int SIZE = 3;

const double payRates[SIZE] = {12.11, 20.34, 34.32};


displayPayRates(payRates, SIZE);
```

Prototype:

```
void displayPayRates(const double *rates, int size);
```

Not necessary. Why?

Const variables need Pointers to a Const

```
const int SIZE = 3;

const double payRates[SIZE] = {12.11, 20.34, 34.32};


displayPayRates(payRates, SIZE);
```

```
void displayPayRates(const double *rates, int size);
```

Write the code to display the Pay Rates in Dollars ($)
Using POINTERS (*) AND NOT ARRAYS ([])

Constant Pointers

A pointer ITSELF can be a constant.

```
int valueA = 29;

int valueB = 35;

int * const ptr = &valueA;


ptr = &valueB;    // error: assignment of read-only variable 'ptr'

*ptr = valueB;    // OK!


ptr++;            // error: increment of read-only variable 'ptr'

(*ptr)++;           // OK!
```

What is this?

```
int value = 29;
const int * const ptr = &value;

cout << *ptr << endl;

ptr++;

(*ptr)++;
```

What is this?

```
int value = 29;
```

```
const int * const ptr = &value;
```

A constant pointer to a constant variable.

(Note that the variable itself can be a non-constant, but the `const int` protects it)

```
ptr++;
```

Error!

```
(*ptr)++;
```

Error!

Exercises

```
int array[5] = {1,2,3,4,5};
```

```
//A
if(array < &array[1])
    cout << "True";
else
    cout << "False";
```

```
//B
if(&array[4] < &array[1])
    cout << "True";
else
    cout << "False";
```

```
//C
if(array != &array[2])
    cout << "True";
else
    cout << "False";
```

```
//D
if(array != &array[0])
    cout << "True";
else
    cout << "False";
```

Exercises

```
int array[5] = {1,2,3,4,5};
```

```
//A
if(array < &array[1])
    cout << "True";
else
    cout << "False";
```

True

```
//B
if(&array[4] < &array[1])
    cout << "True";
else
    cout << "False";
```

False

```
//C
if(array != &array[2])
    cout << "True";
else
    cout << "False";
```

True

```
//D
if(array != &array[0])
    cout << "True";
else
    cout << "False";
```

False

Exercises

Write a function that takes a pointer to an integer as its input and converts it to a negative integer (only if it is positive, however)!

Here is the prototype to help you get started:

```
void makeNegative(int *val);
```

Write some code the demonstrates the use of this function.

Exercises

```
void makeNegative(int *val)
{
   if(*val > 0)
      *val = -(*val);
}
```

Write some code the demonstrates the use of this function.

```
int a = 1;
int b = -1;
makeNegative(&a);
makeNegative(&b);
cout << a << " " << b << endl;
```

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.

```
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:

```
int * ptr;

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

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

```
for(int i = 0; i < 100; i++)

   ptr[i] = 1;
```

Dynamically Allocating Arrays

Not limited to only using integers.

```
char * cPtr;

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

What is stored in the cPtr array?

```
for(int i = 0; i < 26; i++)

   cPtr[i] = 'a' + i;

cPtr[26] = '\0';
```

Dynamically Allocating Arrays

Not limited to only using integers.

```
char * cPtr;

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

What is stored in the cPtr array?

```
for(int i = 0; i < 26; i++)

   cPtr[i] = 'a' + i;

cPtr[26] = '\0';
```

NULL byte at the end

```
abcdefghijklmnopqrstuvwxyz
```

Memory is finite

```
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**)

```
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.

```
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 resued

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

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

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

Always free the memory that you dynamically allocate.

C++ does not do garbage collection

Deleting Arrays

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

```
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.

```
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).

```
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).

`name` is a local variable to the function.

Exists only within the scope of the function.

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

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.

```
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.

```
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;
```

Exercises

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

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

Exercises

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

```
int * ip;

ip = new int;

*ip = 255;

delete ip;
```

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

Exercises

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

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

```
int * ip;

ip = new int[500];

for(int i = 0; i < 500; i++)

  *(ip + i) = 255;

delete [] ip;
```

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:

`bool`	`int`	`unsigned long int`
`char`	`long int`	`float`
`unsigned char`	`unsigned short int`	`double`
`short int`	`unsigned int`	`long double`

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 it's limitation?

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;

};



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;

};



Time now;
```

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

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;
};
```

The *tag* (i.e. name of the new structured data-type you are defining).

```
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;
};


Time now;
```

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

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;
};
```

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

```
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;
};
```

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

```
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;
};
```

Variables are declared like in any function.

```
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;
};
```

```
Time now;
```

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`).

```
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?

```
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.

```
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**.

```
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?

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

int main() {
   Date current;
   cout << “Enter the current date: “;
   cin >> current;
   cout << current << endl;
}
```