Course Notes Set 4:

COMP1200-001

Introduction to Computing for Engineers and Scientists

C Programming

Modularity

Computer Science and Software Engineering

Auburn University

Modularity

```
#include <stdio.h>
int main(void)
{
  int age;
  printf("Enter your age: ");
  scanf("%d",&age);
  printf("Your age in dog years is:%d\n",age*7);
  return 0;
}
```

What if the *printf* and *scanf* commands did not exist?

Modularity

Without these commands, we'd have to write the code to load data/print formatted output to the screen each and every time we had something to load/print in the program. Our program would look like this:


```
int main(void)
{
    int age;
    /* start the print code */
    print code line 1;
    print code line 2;
    ...
    print code line n;
    /* start the scan code */
    scan code line 1;
    scan code line 2;
    ...
    scan code line k;
    /* start the print code */
    print code line 1;
    print code line 2;
    ...
    print code line n;
    return 0;
}
```

Code to print to screen

Code to read from keyboard

Code to print to screen

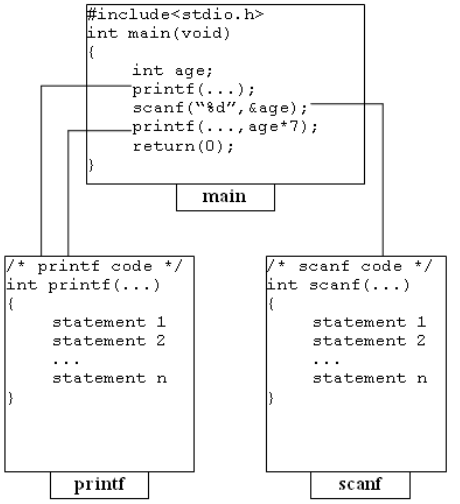
Modularity

Let's suppose it takes 500 lines of code to do a print, and 750 lines of code to do a scan.

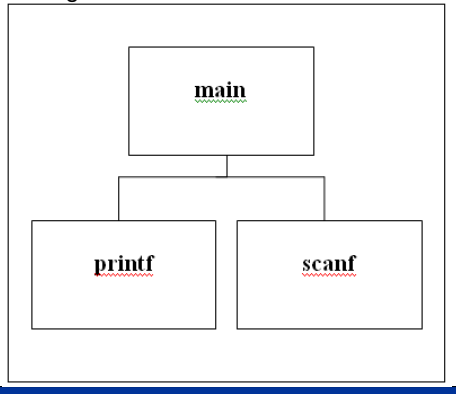
Our 9 line program is suddenly over 1750 lines long!

Obviously nobody wants to write programs that long to do such simple tasks. This is why the compiler vendor supplies us with ***functions*** such as `printf` and `scanf` so we can ***modularize*** our code

Think of our original program looking like this:



Here we show the program in modular form, with the `printf` and `scanf` functions broken away from our code. We can make this a bit easier to read by just not showing the code, and only showing the functions that our ***main*** uses:



Modularity

This type of chart is called a ***structure chart***. It tells us how a program is broken up in terms of modules.

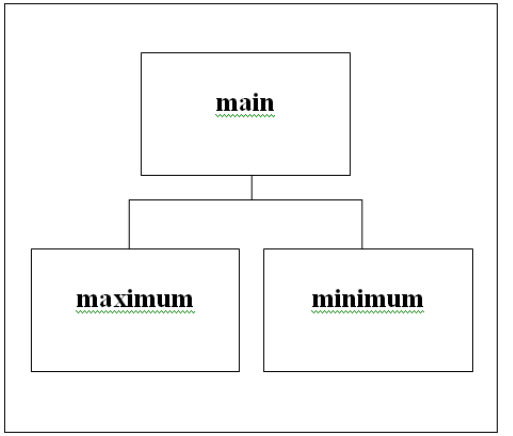
Making programs modular has several advantages:

1. Easier testing.
2. Reduced code length.
3. Reusability.
4. Abstraction.

Modularity

Traditionally, functions such as `printf`, `scanf`, etc. are not shown in structure charts because they are so commonly used. However, it is often the case that we will write code that is used frequently in a program. In that case, we may want to break the code out into a module so we don't have to keep writing it over and over (similar to the `printf/scanf` example).

Our **structure chart** for a main program that used these two functions would look like this:



Functions

We've already seen examples of functions in the stdio library:
`printf, fprintf, scanf, fscanf.`

The C runtime math library contains many other functions:

```
sin(x)
cos(x)
tan(x)
pow(x,y)
...
```

These functions are already written for us. But what if we have a module of code (such as **maximum**) that we would like to put in a function for our use?

Functions

Let's take an example:

There are many cases where we might want to compare two numbers and return the larger of the two. Since this type of activity is so common, we can break it out into a separate module called **`maximum`**. We might have a similar module for finding the smaller of two numbers, called **`minimum`**.

A program that finds the **maximum**.

```
#include<stdio.h>
int main()
{
   int n1, n2, theMax;
   printf("Enter two integers: ");
   scanf("%d %d",&n1,&n2);
   if (n1 > n2)
   {
      theMax = n1;
   }
   else
   {
      theMax = n2;
   }
   printf("Max = %d\n",theMax);
   return 0;
}
```


Functions

Now let's take the part that figures out the maximum:

```
if (n1 > n2)
{
    theMax = n1;
}
else
{
    theMax = n2;
}
```


Functions

What we want is to just be able to say something like this in our program:

```
theMax = maximum(n1,n2);
```

This is an example of a ***function call***. We'll encase our code in a function called **`maximum`** and then give it two numbers to compare. We'll want it to return the largest of the numbers and put it in **`theMax`**.

The basic parts of a function look like this:

```
<return_type> <name>(<parameters>)
{
   <declarations>
   <statements>
}
```

Where **`<return_type>`** is the type of value that will be passed back to the code that called this function.

`<name>` is the name of the function, it must conform to normal variable naming conventions.

`<parameters>` is a list of information that will be passed via variables to the function. If no information is passed, then **`(void)`** or **`()`** should be used for **`<parameters>.`**

Let's look at an example using our **maximum** function:

```
int maximum(int num1, int num2)
{
   int maxNum;
   if (num1 > num2)
   {
      maxNum = num1;
   }
   else
   {
      maxNum = num2;
   }
   return maxNum;
}
```


Functions

<return_type> = int
This indicates that this function will return an integer value to whoever calls it.

<name> = maximum
The name of the function is maximum.

```
int maximum(int num1, int num2)
```

<parameters> = (int num1, int num2)
This function needs two pieces of integer information passed to it. Think of these parameters (called *formal parameters*) as variable declarations.

Functions

<declarations> → `int maxNum ;`

We declare a variable to hold the maximum of num1 and num2.

`return maxNum ;`

The statements determine which is larger, num1 or num2. This is assigned to `maxNum` . The return statement is used to send this value back to the calling program.

Because the **<return_type>** → `int`,
the variable with `return` must be an **integer**.

Calling Functions

Now let's suppose we want to use our function. Any time we **use a function**, it is known as ***calling*** it.

Let's call our function:

<name> of the function

(<arguments>)

```
int m, n1=3, n2=1;
. . .
m = maximum(10,20);
m = maximum(pow(2,5),33);
m = maximum(n1,n2);
m = maximum(n1 + 10, n2);
printf("Max = %d\n",maximum(n1,n2));
```

Calling Functions

What happens when these calls occur?

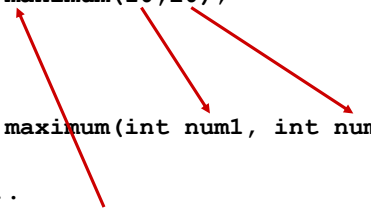
The C runtime system will find the function you are calling. It then takes the *actual arguments* you give it and assign them to the *formal parameters* (just like an internal assignment statement had occurred).

Calling Functions

So if we have:

```
m = maximum(10,20);


int maximum(int num1, int num2)
{
   ...
   return maxNum;
}
```

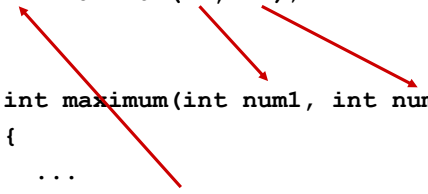


Calling Functions

Or if we have:

```
m = maximum(n1, n2);


int maximum(int num1, int num2)
{
   ...
   return maxNum;
}
```



Parameter Lists

When we define a function:

```
int maximum(int num1, int num2)
{
...
```

The parameters num1, and num2 are called the ***formal parameters.***

Parameter Lists

When we call a function:

```
largest = maximum(x,y);
```

The parameters x and y are called the ***actual arguments***.

Calling Functions

Or if we have:

```
m = maximum(n1, n2);
```

Actual arguments

```
int maximum(int num1, int num2)
{
   ...
   return maxNum;
}
```

Formal parameters

Number and Order of Arguments

Because the runtime environment goes through each actual parameter in the call and assigns it to the corresponding formal parameter,

order and number are very important.

Number and Order of Arguments

For example:

```
largest = maximum(x);
```

is wrong!

Why?

```
int maximum(int num1, int num2)
{
   ...
   return maxNum;
}
```

?????

Number and Order of Arguments

For example:

```
largest = maximum(x,y,z);
```

is wrong!

Why?

?????

```
int maximum(int num1, int num2)
{
  ...
  return maxNum;
}
```


Number and Order of Arguments

Consider what would happen if we did the following:

```
largest = maximum(3.4,7.8);
```

Remember that our formal parameters are **int** not **float**.

```
int maximum(int num1, int num2)
{
  ...
  return maxNum;
}
```

7

Void Functions

Some functions do not need to return a value.

For example, if we wrote a function to print a row of asterisks, we wouldn't need it to return anything. To indicate this, we use a special type called void.

```
void PrintAsterisks()
{
  printf("**********************\n");
}
```

Void functions do not require a return statement.

Parameter Lists

When we call this function, we supply no parameters and do not put it inside an assignment:

Empty ( )
Because
No parameters

```
int main(void)
{
  printAsterisks();
  printf("Asterisks above and below.\n");
  PrintAsterisks();
  return 0;
}
```

Prototype

A function must be declared before it can be referenced.

Prototypes

At this point, a logical question to ask is:

"Where do I put the functions I create?"

The usual place to put them is ***after the main function***. The problem with doing this is that the main function may call the our function. Since the compiler hasn't seen that function yet, it won't know anything about it (such as the parameters needed) and will probably complain.

We solve this problem by putting ***a prototype before the main function***. A prototype says to the compiler:

"I'm going to be using this function;
here's what it returns and the parameters it requires."

Prototypes

To create a prototype for a function, we just take the **declaration portion** of a function and put a **semi-colon** after the closing parenthesis:

<return type> <name> (<parameters>);

A prototype for maximum would look like:

```
int maximum(int num1, int num2);
```

Note the "**;**" A "**;**" is at the end of a prototype, but **not** at the end of the definition.

So our file organization looks like:

```
#include statements

prototypes
int maximum(int num1, int num2);
int minimum(int num1, int num2);

main function
int main(void)
{
   ...
}

function
int maximum(int num1, int num2)
{
...
}

function
int minimum(int num1, int num2)
{
...
}
```

Scope

The scope of a name refers to the area in a program in which that name is visible and can be used.

Scope

- The scope of a defined constant name begins at its definition and continues to the end of the source file.
- The scope of a function subprogram name begins with its prototype and continues to the end of the source file.
- All formal parameters and local variables are visible only from their declaration to the closing bracket of the function in which they are declared.

Call by Value

Send a copy of argument values to function parameters.

Return only one value.

Call By Value

This method of calling functions with parameters is called **call by value**. In this method, we pass a value to the function and **a copy is made** for the function to use. The **function can change its copy** of the value ... but the changes are not reflected back to the calling program.

```
largest = maximum(x,y);
```

The variables x and y cannot be changed by maximum. Even if maximum changes the values of num1 and num2, there is no effect on x and y.

Call By Value

```
                          After call
main                      to maximum
 int x          FFF8  4     4
 int y          FFFA  9     9
 int largest    FFFC  ?     9


maximum
 int num1       FFE0  4
 int num2       FFE2  9
 int maxnum     FFE4  9
```

The `FFF_`s are made up memory addresses in hexdecimal.

Call By Value

```
int main()                          After call
{                                   to addOne()
  int x=34;            FFF4  34      34
  int y=0;             FFF6  0       35
  y = addOne(x);
  return 0;
}

int addOne(int num1)
{
  num1 = num1 + 1;     FF4A 34       35
  return num1;
}
```


Call by Reference

Pass the addresses of the arguments to the function's pointer type parameters.

Pass values back to calling function by writing at the argument's address.

Functions that "return" more that one value

The previous use of parameters have been **call by value**.

In this method, we pass a value to the function and **a copy is made** for the function to use. The **function can change its copy** of the value ... but the **changes are not reflected back to the calling program.**

Also, the previous examples have provided the calling program with **only one value using the return**.

But what if we need to send more than one value to the calling program.

Call By Value

```
int main()                          After call
{                                   to addOne()
  int x=34;            FFF4  34       34
  int y=0;             FFF6  0        35
  y = addOne(x);
  return 0;
}

int addOne(int num1)
{
  num1 = num1 + 1;     FF4A  34       35
  return num1;
}
```


Call by reference

Another method of using parameters is **call by reference**. In this method, the **address of the variable** is use to share information between the calling program and the function.

A example of this is using a function to input data.

Look at the function prototype

```
void getInput(*float c1, *float c2, *float c3);
```

In main we use

```
float a, b, c;
getInput(&a, &b, &c);
```


Call by reference

```
void getInput(float *c1, float *c2, float *c3);
```

In the function prototype, ***** means that the variable following the ***** will hold an address.

The variables **c1, c2,** and **c3** for function **getInput** do not hold floats; they hold addresses of float values.

Call by reference

```
void getInput(float *c1, float *c2, float *c3);

float a, b, c;
getInput(&a, &b, &c);
```

Because **&** is the "**address of**" operator, a copy of the *addresses* of **a, b,** and **c** are put into the memory cells reserved for **c1, c2,** and **c2** in the memory region reserved for the variables of **getInput**.

Call by reference

```
main                                After getInput
  float a;       FFF8    ?      3.4
  float b;       FFFA    ?      2.2
  float c;       FFFC    ?      4.0
-------------------------------------------------------------
getInput
  float *c1;     FFE0   FFF8
  float *c2;     FFE2   FFFA
  float *c3;     FFE4   FFFC
  ....
  scanf("%f %f %f",c1,c2,c3);
-------------------------------------------------------------
From the keyboard we read  3.4 2.2 4.0
```

Notice
No "&"
because
c1, c2, c3
are addresses.