Functions – Intro

Function- a collection of statements that perform a specific task
- separately defined and named

Modular Programming Principle:
- a program can be logically broken down into manageable units (functions, modules, code blocks ...)

Modular programming style permits more effective debugging, because each module can be unit-tested and debugged separately.

Debugging = during testing, when find that the program departs from the required behaviour, determine how to amend the code so it conforms to requirements, and making and testing those code changes.

Functions – Intro

Declaring and defining your own

```
#include <iostream>
using namespace std;

float average2( float x, float y)
{
      float sum = x + y;
      return ( sum /2);
 }

int main()
{
    float a, b, avg;
    cout << " Enter a floating point number: ";
    cin >> a;
    cout << " Enter a floating point number: ";
    cin >> b;
    avg = (a+b)/2;
    cout << " The average is " << avg << endl;
```

Functions – Intro

Declaring and defining your own functions:

```
#include <iostream>
using namespace std;

float average2( float x, float y)
{
      float sum = x + y;
      return ( sum /2);
 }

int main()
{
    float a, b, avg;
    cout << " Enter a floating point number: ";
    cin >> a;
    cout << " Enter a floating point number: ";
    cin >> b;
    avg = (a+b)/2;
    cout << " The average is " << avg << endl;
    return 0;
}
```

Functions - Intro

Functions - Built into C++

arithmetic functions

+ - * /

x+y means plus(x,y)

Functions that come into our programs through library inclusion:

sizeof (int)

These functions are more complex than what we will write

Functions - Intro

Functions - Built into C++

arithmetic functions

+ - * /

flexible about type!

x+y means plus(x,y)

Functions that come into our programs through library inclusion:

sizeof(int)

← takes a type not a variable!

These functions are more complex than what we will write

Functions - Intro

Functions - Built into C++

arithmetic functions

+ - * /

x+y means plus(x,y)

Functions that come into our programs through library inclusion:

sizeof (int)

These functions are more complex than what we will write; we will be more rigid about

- need our arguments to be variables or literals
- need the type of the arguments to be clear in the declaration (prototype) and the definition of the function.

Functions – How deep can we go?

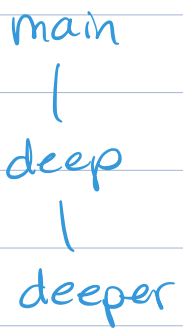
```
// This program has 3 functions: main, deep, deeper
#include <iostream>
using namespace

void deep()
{
    cout << "In deep...\n";
    deeper();
    cout << "Back in deep.\n";
}

void deepe()
{   cout << "In deeper!\n";
}

int main()
{
    cout << "In main, about to call deep.\n";
    deep();
    cout << "Back in main! whew.\n";
}
```

Calls hierarchy:



Lots more to learn about functions!

Back to functions later...

char type inputs

```
// This program reads a single character into a
// char variable
#include <iostream>
using namespace std;
int main()
{
    char ch;
    cout << "Type a single character and press Enter: ";
    cin >> ch;
    cout << "You entered " << ch << endl;
    return 0;
}
```

cin - ignores white space
- often, that is what you want
- used white space as delimiters

so you can cin multiple entities at once.

```
cin >> age >> initial >> gpa
```

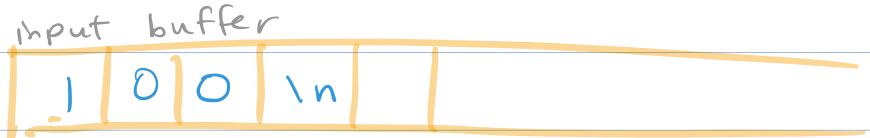
age: declared int
initial: declared char
gpa: declared float

But what if we want to read a white space character? (space bar, Tab, Enter)

Char type inputs

```
// This program demonstrates how a program can be
// paused until the user hits Enter
#include <iostream>
using namespace std;
int main()
{
    char ch;
    cout << "This program has paused. Press Enter to continue.";
    cin.get(ch);
    cout << "It has paused a second time. Press Enter
            to continue.";
    ch = cin.get();
    cout << "It has paused a third time. Press Enter
            to continue.";
    cin.get();
    cout << "Cheers!\n";
    return 0;
}
```

There are subtleties to the mixed use of `cin <<` and `cin.get()`



If - else Statements – Basics

```
if (hours ≤ 40)
{
}
        pay = 40 * payrate
else
{
        regPay = 40 * payrate;
        overtimePay = (hours-40) * 2 * payRate;
        pay = regPay + overtimePay
}
```

There need not be an "else" clause.

```
mark = computeGrade(test, a1, a2, a3, final);
if (mark ≥ 95)
{
    cout << "Wow! ";
}
cout << "Your grade is " << mark;
```

If-else Statements – Basics

The compiler will let you leave off the { , }
BUT this can change the meaning.

```
if (hours ≤ 40)
        pay = 40 * payrate

else
        regPay = 40 * payrate;
        overtimePay = (hours-40) * 2 * payRate
        pay = regPay + overtimePay
```

If-else Statements – Basics

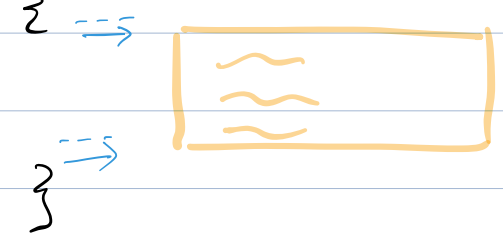
The compiler will let you leave off the { , }
BUT this can change the meaning.

```
if (hours ≤ 40)
        pay = 40 * payrate
else
        regPay = 40 * payrate;

overtimePay = (hours-40) * 2 * payRate
pay = regPay + overtimePay
```

White space doesn't matter (much) to the logic of the program (unlike Python)

Our 159 style sheet says always block off your if-else code blocks with { } That is safer



If statements can be chained or embedded within one-another... be thoughtful about the logic,

```
if (mark >= 80)
{
    if (mark >= 90)
    {
        if (mark >= 95)
        {
            cout << "Wow! ";
        }
        else
        {
            cout << "Excellent! ";
        }
    }
    else
    {
        cout << "Very good. ";
    }
}
cout << "Your mark is " << mark << endl;
```

If statements can be chained or embedded within one-another... be thoughtful about the logic,

```
if (mark >= 80)
{
    if (mark >= 90)
    {
        if (mark >= 95)
        {
            cout << "Wow! ";
        }
        else
        {
            cout << "Excellent! ";
        }
    }
    else
    {
        cout << "Very good. ";
    }
}
cout << "Your mark is " << mark << endl;
```