Goals for This Lecture:

• Understand the use of predefined functions in C++
• Understand user defined functions in C++
Predefined functions in C++

- Most C++ compilers can access precompiled libraries of functions to accomplish a wide variety of basic tasks.
- We refer to these types of functions as predefined functions.
- Many math basic mathematical functions are contained in the cmath library.
- Example:

```cpp
#include <iostream>
#include <cmath>     // include the math library
using namespace std;

main()
{
    double x=2.0;
    cout << "sqrt(x) = " << sqrt(x) << endl;
    cout << "sin(x) = " << sin(x) << endl;
    return(0);
}
```
Rules of predefined function use in C++

- Functions are accessed through a function call or function invocation.
- Many C++ predefined functions return values.
  - Those that don't are called void functions.
- Functions that return a value can be used anywhere that it would be legal to use a value of that type.
- Example:

```cpp
#include <iostream>
#include <cmath>      // include the math library
using namespace std;
main()
{
    double a=1.0, b=2.0, c= 0.5, root;
    root = (-b + sqrt(b*b-4.0*a*c))/(2.0*a);
    cout << "root = " << root << endl;
    return(0);
}
```

- For a list of useful functions see appendix 4 of Savitch.
Predefined void functions in C++

• Some functions return no value whatsoever
  – These are called void functions
• They are accessed through a function call or function invocation
  – They cannot be used in expressions
• Essentially they act like FORTRAN subroutines
• An example is the `exit` function
• Example:

```cpp
#include <iostream>
#include <cstdlib>      // include the cstdlib library
using namespace std;

int main()
{
    double a=1.0, b=2.0, c = 0.5;
    if(b*b-4.0*a*c < 0.0) exit(1);  // indicate an error
    cout << " root is real!" << endl;
    return(0);
}
```

• For a discussion of `exit` see page 102 of Savitch
Defining your own functions in C++

• Like FORTRAN, C++ allows you to create your own functions

• In C++ there are two steps to doing this:
  1. Creation of a function declaration, a.k.a. a function prototype, in the calling program
  2. Creation of a function definition

• The function declaration tells the compiler about the form of the function to that the calling program can reference it

• It is similar to an interface block in FORTRAN

• The function definition defines the actions of the function
Defining your own functions in C++

- Example:

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

// function declarations (note the semicolons)
double real_part(double ac, double bc, double cc);
double imag_part(double ac, double bc, double cc);
main()
{
    double a=1.0, b=2.0, c= 0.5, real_p, imag_p;
    if(b*b-4.0*a*c < 0.0)
    {
        real_p = real_part(a,b,c); // calc. the real part
        imag_p = imag_part(a,b,c); // calc. the imag. part
        cout << " root = " << real_p << "+" <<imag_p<< "i"<< endl;
    }
    return(0);
}
Function definitions in C++

// real_part function definition (note: no semicolons)
double real_part(double a, double b, double c)
    // a, b, & c are called the formal
    // parameters, or parameters for short,
    // of the function
{
    return( -b/(2.0*a));
}

// imag_part function definition
double imag_part(double a, double b, double c)
{
    return( sqrt(4.0*a*c-b*b)/(2.0*a) ) ;
}
Defining your own void functions in C++

• Functions can be defined of almost any type
• This includes type **void**
  – These return no value
• Both the declaration (prototype) and the definition must be declared as void
• Example:

```cpp
#include <iostream>
#include <cstdlib>  // include the cstdlib library
using namespace std;
void warn_user() ;  // function declaration
main()
{
  double a=1.0, b=2.0, c= 0.5;
  if(b*b-4.0*a*c < 0.0) warn_user();  // issue warning
  return(0);
}
```
The void-type function

```cpp
void warn_user() ; // function definition
{
    cout << " warning: root is complex!" << endl;
}
```

- Note: functions of type void need not have a `return` statement in them

- Function execution ends at the `}` brace and execution returns to the calling point
Local Variables & Scope

• Just like with FORTRAN variables declared inside the braces of a function definition are local to the function, i.e. they are local variables
• We say the scope of the variables lies within the function
• Local variables are unknown outside of the function

#include <iostream>
using namespace std;
double add_two(double z);  // Function declaration
main(){
double x=1.0, y;
y = add_two(x);
cout << " warning: root is complex!" << endl;
return(0); }

double add_two(double w)  // Function definition
{ double u;
u = w+2.0;    // Local variable u is unknown outside of
return(u);}   // the function add_two
Global Variables & Constants

- Variables or constants declared outside the braces of any function are globally accessible to all functions defined within that file.

```cpp
#include <iostream>

using namespace std;

const double PI = 3.1415927; // Global constant
double x_global;             // Global variable

double add_things(double z); // Function declaration

main(){
  double x = 1.0, y;
  x_global = 1.0;            // Set the global variable
  y = add_things(x);
  cout << "warning: root is complex!" << endl;
  return(0); }

double add_things(double w) // Function definition
{ double u;
  u = w + PI + x_global;    // access a global variable & constant
  return(u); }
```
Call-by-value argument passing

• C++ differs from FORTRAN in the way it passes arguments by default
• The default argument passing scheme is referred to as call-by-value
• The values of the arguments are plugged into the parameters of the function
• The parameters are not associated with the arguments
• Therefore when using the call-by-value argument passing scheme changes to the parameters in the function are not manifested outside of the function
• C++ has a call-by-reference scheme akin to that of FORTRAN but it must be invoked explicitly
Call-by-reference argument passing

• Call-by-reference argument passing in C++ is done through the use of the referencing operator & in conjunction with the type definition
• Also known as the addressing operator
• The parameters in both the function declaration and function definition must be declared as call be reference parameters
• Warning: lots of potential for errors here
• Example:

```c++
double add_two(double& x) ; // function declaration
double add_two(double& y)    // function definition
{ y = y+2.0; return(y); }
```

• Note: y will be changed in the calling program
• Argument in function invocation must be a variable (not an expression)
Assignment

– Read sections 4.1 & 5.1 of Savitch