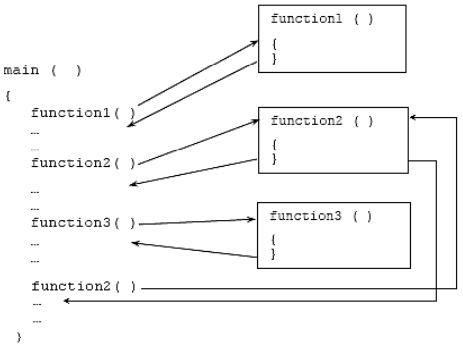
261102
Computer Programming

Lecture 6: Functions

Divide and conquer

- Construct a program from smaller pieces or components (modules)
- Each piece more manageable than the original program
- Modules in C++: Functions and Classes
- Programs use new and "prepackaged" modules
  - New: programmer-defined functions, classes
  - Prepackaged: from the standard library

Functions



- Modularize a program
- Boss to worker analogy
  - A boss (the calling function or caller) asks a worker (the called function) to perform a task and return (i.e., report back) the results when the task is done.
- Functions invoked by function call
  - Function name and information (arguments) it needs
- Software reusability
  - Call as many time as needed (with different arguments)

Functions

- Function definitions
  - Only written once
  - Hidden from other functions
- Local variables
  - Known only in the function in which they are defined
  - All variables declared in function definitions are local variables
- Parameters
  - Local variables passed to function when called
  - Provide outside information

Function Definition

return-value-type *function-name* **(** *parameter-list* **)**
{
declarations and statements
}

– Parameter list
- Comma separated list of arguments
  – Data type needed for each argument
- If no arguments, use **`void`** or leave blank

– Return-value-type
- Data type of result returned (use **`void`** if nothing returned)

Function Definition

- **`return`** keyword
  – Returns data, and control goes to function's caller
    - If no data to return, use **`return;`**
  – Function ends when reaches right brace
    - Control goes to caller
- Functions cannot be defined inside other functions
- Example function

```
int square( int y )
{
   return y * y;
}
```


Function Calling

function-name **(** *argument-list* **)**

- Pass the input arguments
- Function gets its own copy of arguments
- After finished, passes back result (**`return value;`**)
- Example function calling

```
square(2);
z = square(x)+square(y);
cout << square(x);
```


Example 6-A: Rock Paper Scissors

```
1  #include <iostream>
2  using namespace std;
3
4  int gameJudge(char x,char y){
5      if(x == y) return 0;
6      else if(x == 'P' && y == 'R') return 1;
7      else if(x == 'P' && y == 'S') return 2;
8      else if(x == 'R' && y == 'P') return 2;
9      else if(x == 'R' && y == 'S') return 1;
10     else if(x == 'S' && y == 'P') return 1;
11     else if(x == 'S' && y == 'R') return 2;
12     else return 0;
13 }
14
15 int main()
16 {
17     char P1,P2;
18     cout << "Input for player 1 (P,R,S):";
19     cin >> P1;
20     cout << "Input for player 2 (P,R,S):";
21     cin >> P2;
22     int winner = gameJudge(P1,P2);
23     if(winner == 0) cout << "Draw";
24     else cout << "Player " << winner << " wins";
25     return 0;
26 }
```

User-defined Function
- Functions must be defined before used

main Function

Function Calling

Example 6-A: Rock Paper Scissors

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

int gameJudge(char x,char y){
    if(x == y) return 0;
    else if(x == 'P' && y == 'R') return 1;
    else if(x == 'P' && y == 'S') return 2;
    else if(x == 'R' && y == 'P') return 2;
    else if(x == 'R' && y == 'S') return 1;
    else if(x == 'S' && y == 'P') return 1;
    else if(x == 'S' && y == 'R') return 2;
    else return 0;
}

int main()
{
    char P1,P2;
    cout << "Input for player 1 (P,R,S):";
    cin >> P1;
    cout << "Input for player 2 (P,R,S):";
    cin >> P2;
    int winner = gameJudge(P1,P2);
    if(winner == 0) cout << "Draw";
    else cout << "Player " << winner << " wins";
    return 0;
}
```

```
Input for player 1 (P,R,S):R
Input for player 2 (P,R,S):S
Player 1 wins
```

```
Input for player 1 (P,R,S):P
Input for player 2 (P,R,S):P
Draw
```

```
Input for player 1 (P,R,S):R
Input for player 2 (P,R,S):P
Player 2 wins
```


Example 6-A: Rock Paper Scissors

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

int gameJudge(char x,char y){
    if(x == y) return 0;
    else if(x == 'P' && y == 'R') return 1;
    else if(x == 'P' && y == 'S') return 2;
    else if(x == 'R' && y == 'P') return 2;
    else if(x == 'R' && y == 'S') return 1;
    else if(x == 'S' && y == 'P') return 1;
    else if(x == 'S' && y == 'R') return 2;
    else return 0;
}

int main()
{
    char P1,P2;
    cout << "Input for player 1 (P,R,S):";
    cin >> P1;
    cout << "Input for player 2 (P,R,S):";
    cin >> P2;
    int winner = gameJudge(P1,P2);
    if(winner == 0) cout << "Draw";
    else cout << "Player " << winner << " wins";
    return 0;
}
```

Function name: **gameJudge**
- 2 input arguments
- Both inputs are character type
- This function returns integer
- Return-value is different depending on values of the input arguments
- **x,y** are local variables

Call function **gameJudge**
- Transfer values of **P1** and **P2** to function variables **x** and **y** (**x,y** are copies of **P1,P2**)
- The return-value is stored in variable **winner**

Argument Coercion

- Force arguments to be of proper type
- Conversion rules
  - Arguments usually converted automatically
  - Changing from **double** to **int** can truncate data
    - 3.4 to 3
- Mixed type goes to highest type (promotion)
  - **Int * double**

Argument Coercion

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

double half(double x){
    double r = x/2;
    return r;
}

int main(){
    int a = 5;
    double b = half(a);
    cout << b;
    return 0;
}
```

2.5

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

double half(int x){
    double r = x/2;
    return r;
}

int main(){
    double a = 5.5;
    double b = half(a);
    cout << b;
    return 0;
}
```

2

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

double half(int x){
    double r = x/2.0;
    return r;
}

int main(){
    double a = 5.5;
    double b = half(a);
    cout << b;
    return 0;
}
```

2.5

Argument Coercion

Data types	
long double	
double	
float	
unsigned long int	(synonymous with unsigned long)
long int	(synonymous with long)
unsigned int	(synonymous with unsigned)
int	
unsigned short int	(synonymous with unsigned short)
short int	(synonymous with short)
unsigned char	
char	
bool	(false becomes 0, true becomes 1)

Function that Returns No Value

Use **void** as a return-value type

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

void printSOTUS(int N){
    for(int i = 1; i<=N; i++){
        cout << "SOTUS\n";
    }
}

int main()
{
    cout << "I LOVE\n";
    printSOTUS(1);
    cout << "YOU LOVE\n";
    printSOTUS(1);
    cout << "ALL WE NEED IS\n";
    printSOTUS(4);
    return 0;
}
```

```
void function-name( parameter-list )
{
    declarations and statements
}
```

Output

```
I LOVE
SOTUS
YOU LOVE
SOTUS
ALL WE NEED IS
SOTUS
SOTUS
SOTUS
SOTUS
```


Function with Empty Parameter Lists

void or leave parameter list empty indicates function takes no arguments

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

void printSmile(){
    for(int i = 1; i<=69; i++){
        cout << ":)";
        if(i%23==0) cout << '\n';
    }
}

int main()
{
    printSmile();
    return 0;
}
```

```
return-value-type function-name( void )
{
    declarations and statements
}
```

```
return-value-type function-name( )
{
    declarations and statements
}
```

```
:):):):):):):):):):):):):):):):):):):):):):):)
:):):):):):):):):):):):):):):):):):):):):):):)
:):):):):):):):):):):):):):):):):):):):):):):)
```


Function with Empty Parameter Lists

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
using namespace std;

float findMean(){
    int count;
    float sum = 0;
    ifstream source;
    source.open("mydata.txt");
    string textline;
    while (getline(source,textline))
    {
      sum += atof(textline.c_str());
      count++;
    }
    source.close();

    return sum/count;
}
```

```
int main () {

    cout << "Avg = "<< findMean() << '\n';

    return 0;
}
```

Input arguments of findMean = ?

findMean returns ?

Default Arguments

- Function call with omitted parameters
  - If not enough parameters, rightmost go to their defaults
  - Default values
    - Can be constants, global variables, or function calls

```
1  #include <iostream>
2  using namespace std;
3
4  void printSOTUS(int N = 1){
5      for(int i = 1; i<=N; i++){
6          cout << "SOTUS\n";
7      }
8  }
9
10 int main()
11 {
12     cout << "I LOVE\n";
13     printSOTUS();
14     cout << "YOU LOVE\n";
15     printSOTUS();
16     cout << "ALL WE NEED IS\n";
17     printSOTUS(4);
18     return 0;
19 }
```

Set Default Values

Use Default Values

Use N = 4

Default Arguments

Output = ?

```
1  #include <iostream>
2  using namespace std;
3
4  int myFunction(int x=1,int y=1, int z=1){
5      return x+2*y+3*z;
6  }
7
8  int main()
9  {
10     cout << myFunction() << "\n";
11     cout << myFunction(69) << "\n";
12     cout << myFunction(69,10) << "\n";
13     cout << myFunction(69,10,-1) << "\n";
14 }
```


Function Prototype

```
1  #include <iostream>
2  using namespace std;
3
4  void printPrint(char,int,int);
5
6  int main()
7  {
8      printPrint('#',2,2);
9      printPrint('-',2,9);
10     printPrint('#',2,2);
11 }
12
13 void printPrint(char x,int y,int z){
14     for(int i = 1;i<=y;i++){
15         for(int j = 1;j<=z;j++){
16             cout << x;
17         }
18         cout << '\n';
19     }
20 }
```

Function Prototype
(Declare before calling)

Function Calling

Output = ?

Function Definition
(Declare after calling)

Function Prototype

return-value-type *function-name*(*parameter-type-list*);

- Tells compiler argument type and return type of function
- Explained in more detail (function definition) later
- Only needed if function definition after function call
- Function signature = Part of prototype with name and parameters

```
int myFunction(int,int,int);
```

Function signature

Function Prototype

- Prototype must match function definition
  - Function prototype

```
double maximum( double, double, double );
```

  - Definition

```
double maximum( double x, double y, double z )
{
 …
}
```


Function Prototype

```
1  #include <iostream>
2  using namespace std;
3
4  void printPrint(char,int,int);
5
6  int main()
7  {
8      printPrint('#',2,2);
9      printPrint('-',2,9);
10     printPrint('#',2,2);
11 }
12
13 void printPrint(char x,int y,int z){
14     for(int i = 1;i<=y;i++){
15         for(int j = 1;j<=z;j++){
16             cout << x;
17         }
18         cout << '\n';
19     }
20 }
```

```
4  void printPrint(char x,int y,int z);
4  void printPrint(char a,int b,int c);
4  void printPrint(char a,int b=1,int c=1);
```

Parameters' name can be different from function definition

- Default value must be set in the prototype before function call with omitted parameters

Function Overloading

- Function overloading
  - Functions with same name and different parameters
  - Should perform similar tasks
    - i.e., function to square **int**s and function to square **float**s

```
int square(int x) {return x * x;}
float square(float x) { return x * x; }
```

- Overloaded functions distinguished by signature
  - Based on name and parameter types (order matters)
  - Name mangling
    - Encodes function identifier with parameters
  - Type-safe linkage
    - Ensures proper overloaded function called

Function Overloading

```
1  #include <iostream>
2  using namespace std;
3
4  int square(int x){
5      cout << "int version called\n";
6      return x*x;
7  }
8
9  float square(float x){
10     cout << "float version called\n";
11     return x*x;
12 }
13
14 double square(double x){
15     cout << "double version called\n";
16     return x*x;
17 }
18
19 int main()
20 {
21     cout << "result = " << square(2) << "\n";
22     cout << "result = " << square(2.0f) << "\n";
23     cout << "result = " << square(2.0) << "\n";
24     return 0;
25 }
```

```
int version called
result = 4
float version called
result = 4
double version called
result = 4
```

Function Overloading

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int square(int x){
 5      return x*x;
 6  }
 7
 8  float square(float x){
 9      return x*x;
10  }
11
12  int main(){
13      cout << square(2.0);
14      return 0;
15  }
```

```
 In function 'int main()':
13:23: error: call of overloaded 'square(double)' is ambiguous
13:23: note: candidates are:
4:5: note: int square(int)
8:7: note: float square(float)
```


Function Templates

- Compact way to make overloaded functions
  - Generate separate function for different data types
- Format
  - Begin with keyword **`template`**
  - Formal type parameters in brackets **`<>`**
    - Every type parameter preceded by **`typename`** or **`class`** (synonyms)
    - Placeholders for built-in types (i.e., **`int`**) or user-defined types
    - Specify arguments types, return types, declare variables
  - Function definition like normal, except formal types used

Function Templates

- Example

```
template < class T > // or template <typename T>
T square( T value1 )
{
   return value1 * value1;
}
```

  - T is a formal type, used as parameter type
    - Above function returns variable of same type as parameter
  - In function call, T replaced by real type
    - If **`int`**, all **`T`**'s become **`int`**s

```
int x;
int y = square(x);
```


Function Templates

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  template <typename currentType>
 5  currentType justOneBefore(currentType x){
 6      currentType y = x-1;
 7      return y;
 8  }
 9
10  int main()
11  {
12      cout << "result = " << justOneBefore('Z') << "\n";
13      cout << "result = " << justOneBefore(0u) << "\n";
14      cout << "result = " << justOneBefore(-68) << "\n";
15      cout << "result = " << justOneBefore(6.55555) << "\n";
16      return 0;
17  }
```

```
result = Y
result = 4294967295
result = -69
result = 5.55555
```

Function Templates

```
4  template <typename T>
5  T justOneBefore(T);

6
7  int main()
8  {
9      cout << "result = " << justOneBefore('Z') << "\n";
10     cout << "result = " << justOneBefore(0u) << "\n";
11     cout << "result = " << justOneBefore(-68) << "\n";
12     cout << "result = " << justOneBefore(6.55555) << "\n";
13     return 0;
14 }
15
16 template <typename currentType>
17 currentType justOneBefore(currentType x){
18     currentType y = x-1;
19     return y;
20 }
```

Prototype

Different typename can be used in prototype and definition

```
result = Y
result = 4294967295
result = -69
result = 5.55555
```


Function Call by Reference

function-name(*argument-list*)

- Call by value
  - Copy of data passed to function
  - Changes to copy do not change original
  - Prevent unwanted side effects

square(*x*)

function-name(**&***argument-list*)

- Call by reference
  - Function can directly access data
  - Changes affect original

square(**&***x*)

Function Call by Reference

- Reference parameter
  - Alias for argument in function call
    - Passes parameter by reference
  - Use **&** after data type in prototype

```
void myFunction( int &data )
```

    - Read "**data** is a reference to an **int**"
  - Function call format the same
    - However, original can now be changed

Function Call by Value

```
1  #include <iostream>
2  using namespace std;
3
4  int square(int);
5
6  int main()
7  {
8    int num = 5;
9    cout << "Input argument = " << num << "\n";
10   cout << "Return value = " << square(num) << "\n";
11   cout << "Input argument = " << num << "\n";
12 }
13
14 int square(int x){
15     return x*x;
16 }
```

```
Input argument = 5
Return value = 25
Input argument = 5
```

Function Call by Reference

```
1  #include <iostream>
2  using namespace std;
3
4  void square(int &);
5
6  int main()
7  {
8    int num = 5;
9    cout << "Input argument = " << num << "\n";
10   square(num);
11   cout << "Input argument = " << num << "\n";
12 }
13
14 void square(int &x){
15     x = x*x;
16 }
```

```
Input argument = 5
Input argument = 25
```


Function Call by Reference

```
1  #include <iostream>
2  using namespace std;
3
4  int c = 2;
5  void square(int &x = c);
6
7  int main()
8  {
9    cout << "Default argument = " << c << "\n";
10   square();
11   cout << "Default argument = " << c << "\n";
12 }
13
14 void square(int &x){
15     x = x*x;
16 }
```

⇦ Set default reference parameter to variable **c**

```
Default argument = 2
Default argument = 4
```


Recursion (Recursive Function)

- Recursive functions
  - Functions that call themselves
  - Can only solve a base case
- If not base case
  - Break problem into smaller problem(s)
  - Launch new copy of function to work on the smaller problem (recursive call/recursive step)
    - Slowly converges towards base case
    - Function makes call to itself inside the return statement
  - Eventually base case gets solved
    - Answer works way back up, solves entire problem

Recursion (Recursive Function)

- Example: factorial

  $n! = n * (n - 1) * (n - 2) * \ldots * 1$

  - Recursive relationship (n! = n * (n – 1)!)

  $5! = 5 * 4!$

  $4! = 4 * 3! \ldots$

  - Base case (1! = 0! = 1)

Example 6-B: Factorial (recursive ver.)

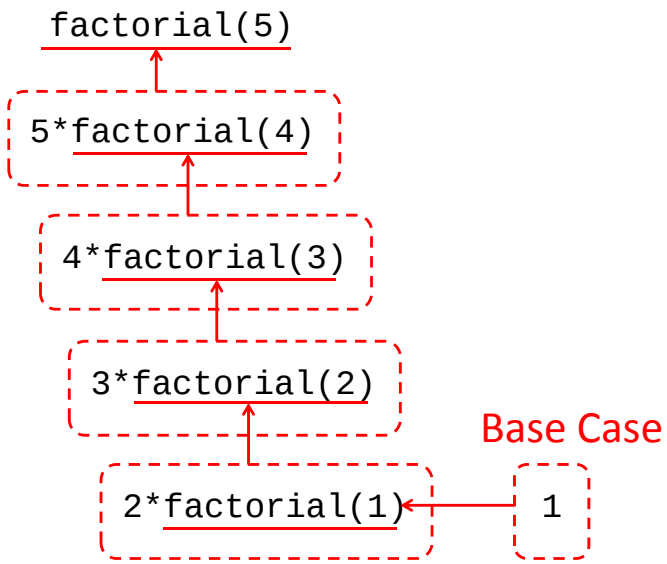
```
1  #include <iostream>
2  using namespace std;
3
4
5  int factorial (int x)
6  {
7      if(x <= 1){
8          return 1;
9      }else{
10         return x*factorial(x-1);
11     }
12
13 }
14
15 int main(){
16     cout << "5! = " << factorial(5);
17 }
```

Base case for 1! or 0! = 1

This function call itself inside
Factorial of x can be obtained
from factorial of smaller number
(x-1)

```
5! = 120
```

Example 6-B: Factorial (recursive ver.)



Example 6-B: Factorial (iteration ver.)

```
1  #include <iostream>
2  using namespace std;
3
4
5  int factorial(int x)
6  {
7      int y = 1;
8      for(;x>=2;x--){
9          y *= x;
10     }
11     return y;
12 }
13
14 int main(){
15     cout << "5! = " << factorial(5);
16 }
```

Use for-loop

```
5! = 120
```


Recursion vs. Iteration

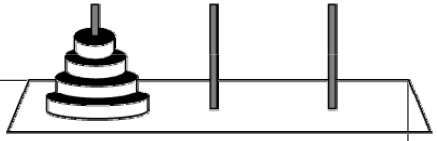
- Repetition
  - Iteration: explicit loop
  - Recursion: repeated function calls
- Termination
  - Iteration: loop condition fails
  - Recursion: base case recognized
- Both can have infinite loops
- Balance between performance (iteration) and good software engineering (recursion)

Recursion vs. Iteration

จงหา 1+2+3+...+N

สิ่งที่ Iteration เห็น	สิ่งที่ Recursionเห็น	สิ่งที่นักศึกษาบางคนเห็น
$f(N)=\sum_{x=1}^{N} x$	$f(x)=x+f(x-1)$ $f(1)=1$	*FFFFFFFFFFFF* *FFFFFFFFFFF* *FFF* *FFFFFFFFF* *FFF* *FFF*

Example 6-C: Towers of Hanoi



```
1  #include <iostream>
2  using namespace std;
3
4  void hanoi(int N,int s=1, int d=3)
5  {
6      if(N == 1) cout << "Move from " << s << " to " << d << '\n';
7      else{
8          int buffer = 6-(s+d);
9          hanoi(N-1,s,buffer);
10         cout << "Move from " << s << " to " << d << '\n';
11         hanoi(N-1,buffer,d);
12     }
13 }
14
15 int main(){
16     hanoi(3);
17 }
```

```
Move from 1 to 3
Move from 1 to 2
Move from 3 to 2
Move from 1 to 3
Move from 2 to 1
Move from 2 to 3
Move from 1 to 3
```