Computer Programming

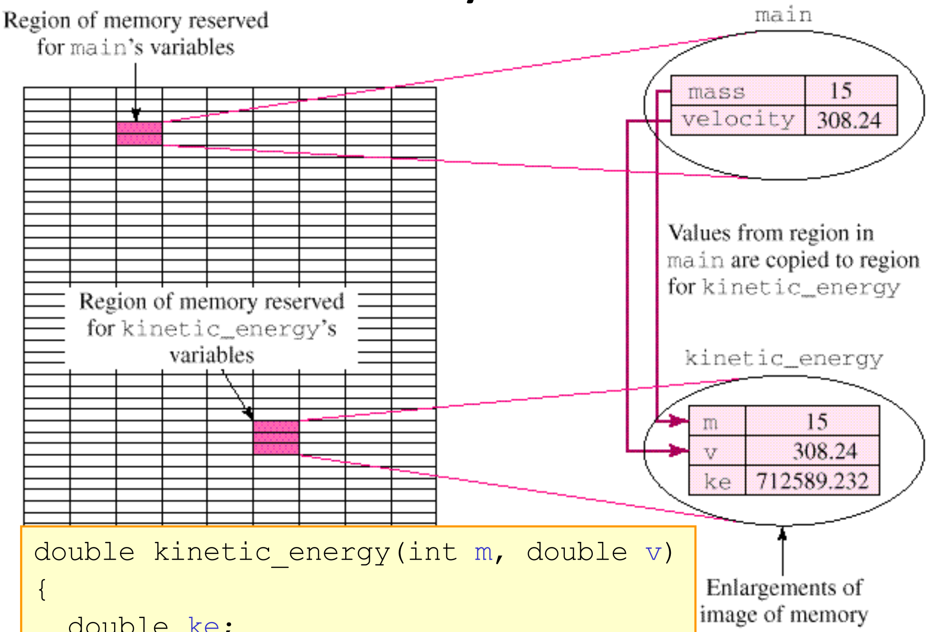
More on Arguments

```
int main( )
{
  int mass=1
  double vel
  double ene
    …
}
```

```
double kinetic_energy(int m, double v)
{
  double ke;
  ke = 0.5*m*v*v;
  return ke;
}
```

Passing Arguments

- Memory allocation
  - When a function is called, a stack frame is allocated for variables in the function's argument list and the local variables declared in the function's body
  - When the function completes execution, the stack frame is freed up
  - ☞ Call (pass) by value
    - Values of variables in the argument list in the calling function are passed to the called function


Region of memory reserved for main's variables
main
mass
15
velocity
308.24
Values from region in main are copied to region for kinetic_energy
Region of memory reserved for kinetic_energy's variables
kinetic_energy
m
15
v
308.24
ke
712589.232
Enlargements of image of memory

```
double kinetic_energy(int m, double v)
{
  double ke;
  ke = 0.5*m*v*v;
  return ke;
}
```

Call by Value Example

```
#include <iostream>
using namespace std;
void cube_vol_area(int, double, double, double, double, double);

int main( )
{
    int id=5;
    double s=3, v=10, x=6.3, y=7.2, z=1.5;
    cout<<"cube surface area="<<s<<" cube volume="<<v<<endl;
    cube_vol_area(id, x, y, z, s, v);
    cout<<"cube surface area="<<s<<" cube volume="<<v<<endl;
}

void cube_vol_area (int id, double width, double length, double height,
     double surface, double volume)
{
    surface = 2*width*height+2*length*height+2*width*length;
    volume  = width*length*height;
    cout<<"cube surface area="<<surface<<" cube volume="<<volume<<endl;
}
```

Call by Value Revisited

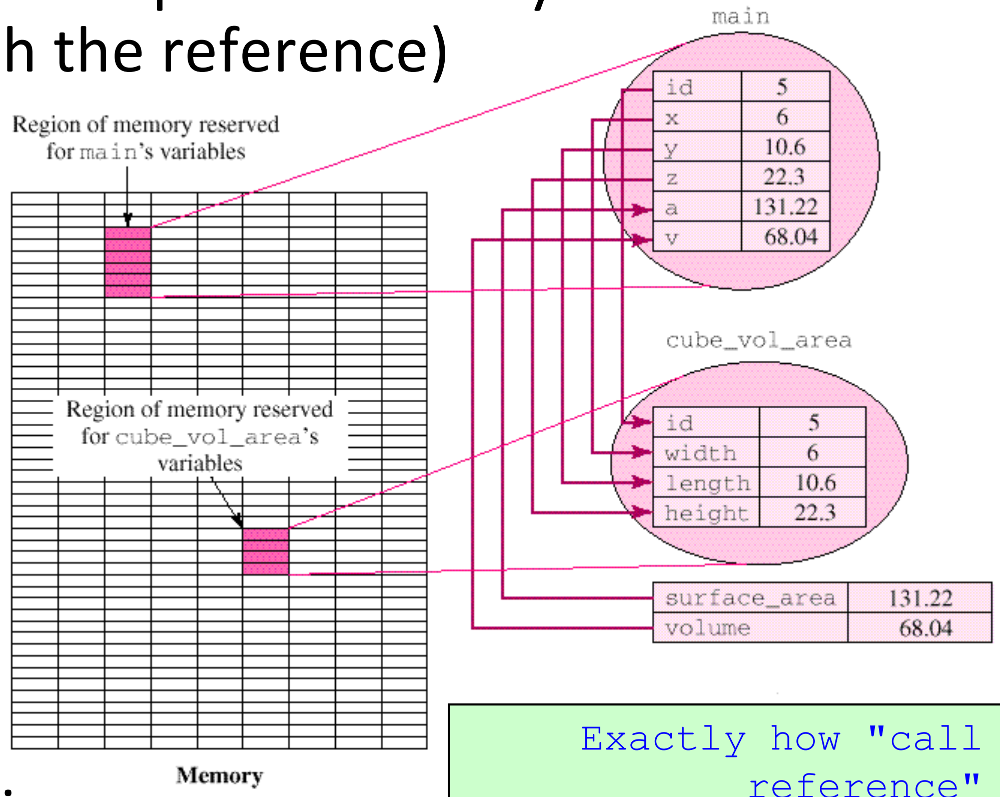
- Need for modifying variables
  - Sometimes it is desired to modify the values of the variables in the argument list that are passed between the caller and the callee
  - ☞ The effect is that the function "returns" multiple new values to the caller as a result of the function call
- Overhead in call by value
  - The overhead incurred in creating (allocating) new variables in the argument lists can be non-negligible when the size of the variable is large
  - ☞ A different way of passing arguments in the function call from "call by value"

Call by Reference

```
void func(int &cref);
      func(count);
```

```
int &cref = count;
```

- Call by reference
  - The called function can operate directly on the variables passed to it (through the reference)
  - Variables can be modified by the called function
  - No new memory is allocated for variables passed by reference
    - The two variables bind together
  - ☞ Add & both in function definition and declaration



Exactly how "call by reference" is implemented is unspecified in C++

Call by Reference Example

```
#include <iostream>
using namespace std;
void cube_vol_area(int, double, double, double, double&, double&);

int main( )
{
    int id=5;
    double s=3, v=10, x=6.3, y=7.2, z=1.5;
    cout<<"cube surface area="<<s<<" cube volume="<<v<<endl;
    cube_vol_area(id, x, y, z, s, v);
    cout<<"cube surface area="<<s<<" cube volume="<<v<<endl;
}

void cube_vol_area (int id, double width, double length, double height,
     double& surface, double& volume)
{
    surface = 2*width*height+2*length*height+2*width*length;
    volume  = width*length*height;
    cout<<"cube surface area="<<surface<<" cube volume="<<volume<<endl;
}
```

Note the location of &

Only at declaration and definition

Not needed for function call

Pointer Argument

- Passing arguments to a function
  - Call by value
  - Call by reference

- Call by (value of) pointer (or call by address)
  - ☞ Can simulate call-by-reference
  - Use pointers and indirection (dereferencing) operator
  - Pass the *address of the argument* using the & operator

  - Note that still the *value of the variable in the function call* is passed to the variable inside the function (call by value)

An Example

Size of the array does not need to be passed as an argument due to '\0'

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

void copy1(char*s1, char*s2) {for (int i=0; (s1[i]=s2[i])!='\0'; i++);}
void copy2(char*s1, char*s2) {for (; (*s1=*s2)!='\0'; s1++, s2++);}

int main( )
{
    char string1[10], string3[10];
    char *string2  = "Hello";
    char string4[] = "Good bye";

    copy1(string1, string2);
    copy2(string3, string4);

    cout << "string1 = " << string1 << endl;
    cout << "string3 = " << string3 << endl;
}
```

```
void copy1(char *s1,
           const char *s2);
```

Point to the starting address of the string literal "Hello" (in DATA)

Initialize the array with characters in string "Good bye"

```
const char
*string2
= "Hello";
```

Pointer vs. Reference

Call by reference is not supported in all languages (e.g. in C++ but not in C)

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

void cubeByReference(int *);

int main( )
{
    int number = 5;
    cout << "Old value is " << number
         << endl;
    cubeByReference(&number);

    cout << "New value is " << number
         << endl;
}

void cubeByReference(int *nPtr)
{
    *nPtr = *nPtr * *nPtr * *nPtr;
}
```

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

void cubeByReference(int &);

int main( )
{
    int number = 5;
    cout … << number
           << endl;
    cubeByReference(number);

    cout … << number
           << endl;
}

void cubeByReference(int &nPtr)
{
    nPtr = nPtr * nPtr * nPtr;
}
```

Pointer vs. Array

As function arguments, the array notation and pointer notation are *equivalent for 1-D array* (e.g. `int a[]`== `int *a`)

```
#include <iostream>
using namespace std;
void printArray(int[], int);

int main( )
{
    const int N = 5;
    int a[N] = {1, 2, 3, 4, 5};

    cout << "Values in a: " << endl;
    printArray(a, N);
}

void printArray(int x[], int size)
{
    for (int i=0;i<size;i++)
    cout << x[i] << ' ';

    cout << endl;
}
```

```
#include <iostream>
using namespace std;
void printArray(int*, int);

int main( )
{
    const int N = 5;
    int a[N] = {1, 2, 3, 4, 5};

    cout << "Values in a: " <<endl;
    printArray(a, N);
}

void printArray(int *x, int size)
{
    for (int i=0;i<size;i++)
    cout << x[i] << ' ';

    cout << endl;
}
```

Array Argument

The value of the name of the array is the memory address of the first element of the array

- Passing arrays to functions
  - Specify the name of the array without any bracket

```
int theArray[24];
modifyArray(theArray, 24);
```

Optional; it helps the function knows the size of the array

- Function to receive an array as an argument
  - Specify the bracket in declaration & definition

```
void modifyArray(int temp[], int size)
{
…
}
```

Empty [ ] means it is an array (not an integer)

☞ The elements in the array can be modified by the function (similar to "call by reference")

Example

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

void printArray(int[], int);

int main( )
{
    const int N = 5;
    int a[N] = {1, 2, 3, 4, 5};

    cout << "Values in array a by row are:" << endl;
    printArray(a, N);
}

void printArray(int a[], int size)
{
    for (int i=0;i<size;i++) cout << a[i] << ' ';

    cout << endl;
}
```

Note that the whole array a can be modified by the function since the latter has access to the original array through the argument (providing address of the data)

```
void printArray(const int a[],
int size);
```

const means that the array elements cannot be modified

Multi-Dimensional Array Argument

a	a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]

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

void printArray(const int[][3], int size);

int main( )
{
    int a[2][3] = {{1, 2, 3}, {4, 5, 6}};
    int b[2][3] = {1, 2, 3, 4, 5};
    int c[2][3] = {{1, 2}, {4}};
    cout << "Values in array a by row are:" << endl;
    printArray(a, 2);
}
void printArray(const int a[][3], int size)
{
    for (int i=0;i<size;i++) {
        for (int j=0;j<3;j++) cout << a[i][j] << ' ';
    }
    cout << endl;
}
```

Only the size of the first dimension is not required; subsequent dimension sizes must be included in declaration to help the compiler know array structure

Optional but helpful

```
void printArray(const int (*a)[3], int size)
```

a is of the type `int (*)[3]`

```
int (*p)[3] = a;
```

p is of a pointer to an array of 3 integers

Default Argument

- Function declaration
  - It is possible not to explicitly pass argument value for the argument that is *infrequently specified different values*
  - Default argument values must be specified once with the first occurrence of the function name (in the declaration *or* definition)

```
void commute_time(double v, double d=25, int n=5);
```

  - No ordinary argument can follow default arguments
    - Default argument(s) must be the trailing argument(s) in the list

```
void commute_time(double v=60, double d, int n=5);

void commute_time(double v, double d=25, int n);
```

wrong!

Using Default Arguments

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

void commute_time (double, double=25, int=5);

int main( )
{
    commute_time(40);
    commute_time(30, 20);
    commute_time(35, 30, 8);
}

void commute_time(double velocity, double distance, int num_lights)
{
    cout<<"The commute time is " <<(distance/velocity + num_lights*0.01)
        <<" hours."<<endl;
}
```

It is good to set the default arguments in the declaration for users to know

Function Overloading

Internally, the compiler will encode each function identifier differently to distinguish overloaded functions

Name mangling

- Function overloading
  - Define two or more functions with the same name
  - ☞ Similar functionality, but different in the argument list

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

```
int maximum(int x, int y)
    {return x>y?x:y;}

int maximum(int x, int y, int z)
    {return x>y&&x>z?x:y>z?y:z;}
```

  - ☞ Function signature: function name & parameter types
  - ☞ Return type difference does not constitute function overloading (name mangling)
  - ☞ Function overloading can make function naming easier, but it can be *confusing* as to which function is called

An Example (1/2)

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

void rotate(int&, int&);
void rotate(int&, int&, int&);

int main( )
{
    int a, b, c, d;
    a=1; b=2;
    rotate(a, b);
    cout<<"a="<<a<<" b="<<b<<endl;

    a=1; b=2; c=3;
    rotate(a, b, c);
    cout<<"a="<<a<<" b="<<b<<" c="<<c<<endl;
}
```

a	b	c
b	c	a

An Example (2/2)

```
void rotate(int& a, int& b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}

void rotate(int& a, int& b, int& c)
{
    int temp;
    temp = a;
    a = b;
    b = c;
    c = temp;
}
```

What happens if the reference is removed?

a	b	c
b	c	a

Function Template

- Dealing with different types of arguments
  - Function overloading: similar operations
  - Function template: same operations
- Automatic code generation
  - A single function template definition is needed
  - Separate function template specializations are *generated by C++ (compiler)* to handle each type of call appropriately
  - ☞ *Compile on demand*
- Defining a function template

Okay to replace `typename` with `class`

```
template <typename type> function_declaration;
```

Function Template (1/2)

```
template <typename T>
T maximum(T x, T y, T z)
{
    T maxvalue = x;

    if (y > maxvalue) maxvalue = y;

    if (z > maxvalue) maxvalue = z;

    return maxvalue;
}
```

Name this file as "maximum.h"

```
int maximum(int x, int y, int z)
{
    int maxvalue = x;
    if (y>maxvalue) maxvalue = y;
    if (z>maxvalue) maxvalue = z;

    return maxvalue;
}
```

- ☞ It is possible to have mixed types

```
template <typename T, typename U>
T maximum (T a, U b) {return (a>b?a:b);}
```

Function Template (2/2)

The number of functions to generate (instantiate) by the compiler depends on the function call

```
#include <iostream>
#include "maximum.h"
using namespace std;

int main( )
{
    int int1, int2, int3;

    cout << "Input three integer value: ";
    cin >> int1 >> int2 >> int3;
    cout << "Maximum integer is " << maximum(int1, int2, int3) << endl;

    double double1, double2, double3;

    cout << "Input three double value: ";
    cin >> double1 >> double2 >> double3;
    cout << "Maximum double is " << maximum(double1, double2, double3);
}
```

Type Conversion

Some prefer the C++-style conversion since it is easier to find such explicit conversion in your code for debugging

- Conversion between data types
  - Explicit conversion

```
static_cast< type >( value )
```

Safer for conversion involving classes

```
( type ) value   or   type( value )
```

```
int i=10, j;
double k, m, n, p;
k = static_cast<double>(i);
m = (double) i;
n = double(i);
```

  - Implicit conversion (mixed-type expression)

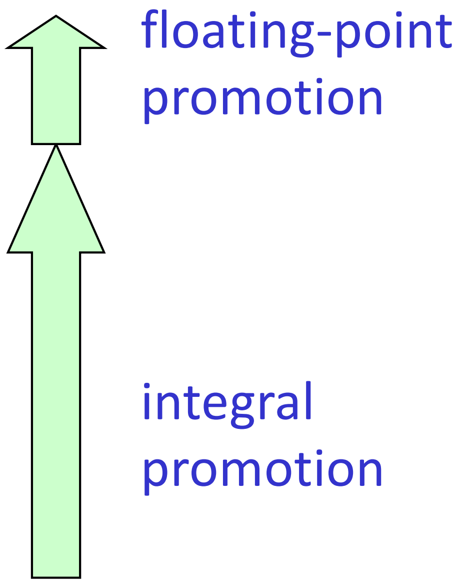
```
p = i;
j = 23.3 / i;
```

Argument Coercion

```
      int i = 1;
cout << atan(i);
```

- Implicit conversion of function arguments
  - Argument values given may not correspond precisely to the data types in the function prototype
  - The compiler performs "argument coercion" to convert arguments to proper types before the function is called

Promotion Rule	
`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`	



Resolving Function Calls

- Finding the right function to call
  - With *default argument*, *function overloading*, *function template*, and *argument coercion*, multiple candidate functions could exist to match the specified function call

```
int maximum(int x, int y) {return x>y?x:y;}
double maximum(double x, double y){return x>y?x:y;}
maximum(3.0, 7);
```

  - The compiler has some rules for finding the "best match" among candidate functions
  - The compiler may complain about *ambiguous* function call if there is no clear winner to match the call

☞ *Avoid writing ambiguous codes by yourself*