# Functions Review

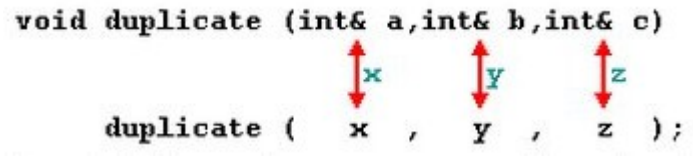
```
// function example
#include <iostream>
using namespace std;

int subtraction (int a, int b)
{
  int r;
  r=a-b;
  return (r);
}

int main ()
{
  int x=5, y=3, z;
  z = subtraction (7,2);
  cout << "The first result is " << z << '\n';
  cout << "The second result is " << subtraction (7,2) << '\n';
  cout << "The third result is " << subtraction (x,y) << '\n';
  z= 4 + subtraction (x,y);
  cout << "The fourth result is " << z << '\n';
  return 0;
}
```

```
The first result is 5
The second result is 5
The third result is 2
The fourth result is 6
```

http://cplusplus.com/doc/tutorial/functions/




```
1 // passing parameters by reference
2 #include <iostream>
3 using namespace std;
4
5 void duplicate (int& a, int& b, int& c)
6 {
7   a*=2;
8   b*=2;
9   c*=2;
10 }
11
12 int main ()
13 {
14   int x=1, y=3, z=7;
15   duplicate (x, y, z);
16   cout << "x=" << x << ", y=" << y << ", z=" << z;
17   return 0;
18 }
```

x=2, y=6, z=14

# Overloaded functions

```
// overloaded function
#include <iostream>
using namespace std;

int operate (int a, int b)
{
  return (a*b);
}

float operate (float a, float b)
{
  return (a/b);
}

int main ()
{
  int x=5,y=2;
  float n=5.0,m=2.0;
  cout << operate (x,y);
  cout << "\n";
  cout << operate (n,m);
  cout << "\n";
  return 0;
}
```

```
10
2.5
```

Recursion

```
// factorial calculator
#include <iostream>
using namespace std;

long factorial (long a)
{
  if (a > 1)
   return (a * factorial (a-1));
  else
   return (1);
}

int main ()
{
  long number;
  cout << "Please type a number: ";
  cin >> number;
  cout << number << "! = " << factorial (number);
  return 0;
}
```

```
Please type a number: 9
9! = 362880
```

# Function Prototypes

```
Type a number (0 to exit): 9
Number is odd.
Type a number (0 to exit): 6
Number is even.
Type a number (0 to exit): 1030
Number is even.
Type a number (0 to exit): 0
Number is even.
```

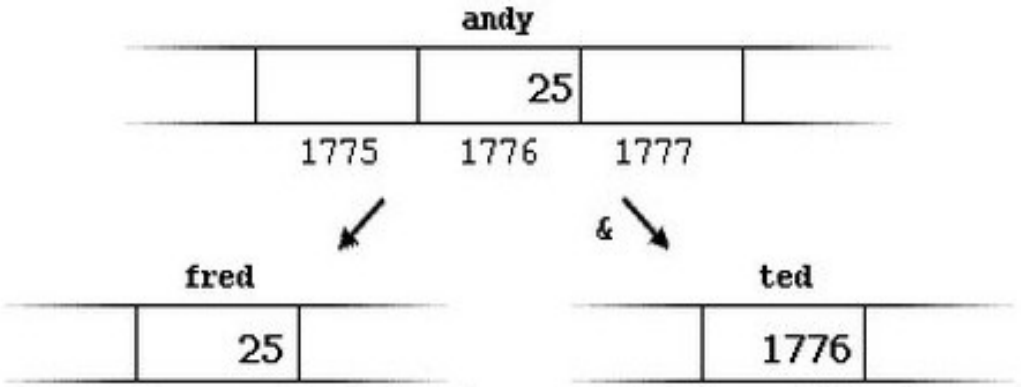
```
1  // declaring functions prototypes
2  #include <iostream>
3  using namespace std;
4
5  void odd (int a);
6  void even (int a);
7
8  int main ()
9  {
10   int i;
11   do {
12     cout << "Type a number (0 to exit): ";
13     cin >> i;
14     odd (i);
15   } while (i!=0);
16   return 0;
17 }
18
19 void odd (int a)
20 {
21   if ((a%2)!=0) cout << "Number is odd.\n";
22   else even (a);
23 }
24
25 void even (int a)
26 {
27   if ((a%2)==0) cout << "Number is even.\n";
28   else odd (a);
29 }
```

# Reference operator

Consider the following code fragment:

```
1 andy = 25;
2 fred = andy;
3 ted = &andy;
```

The values contained in each variable after the execution of this, are shown in the following diagram:



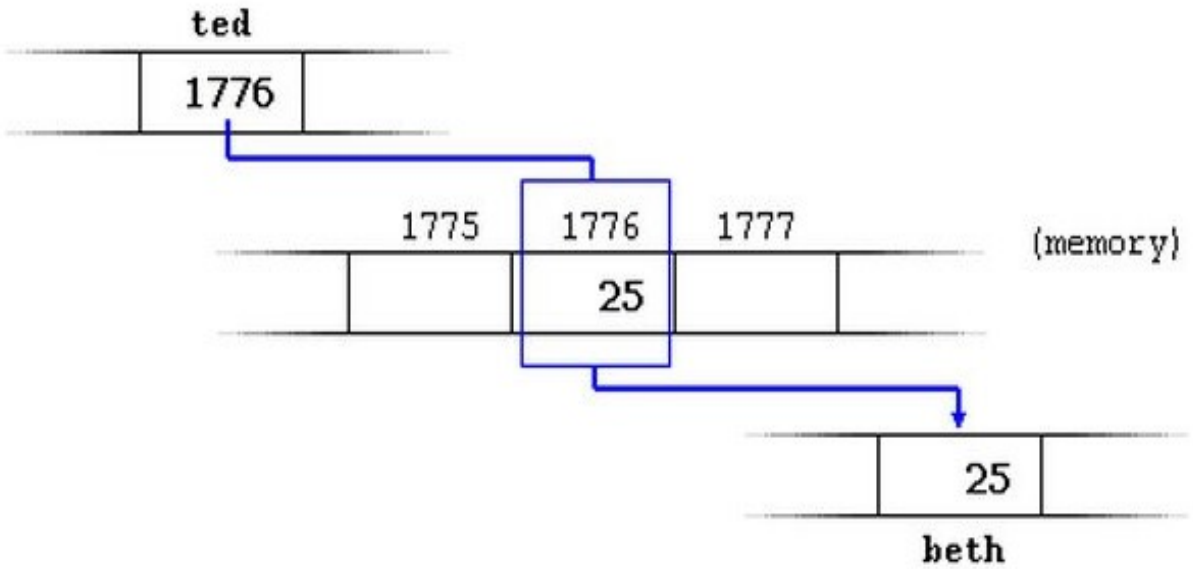

First, we have assigned the value 25 to andy (a variable whose address in memory we have assumed to be 1776).

The second statement copied to `fred` the content of variable andy (which is 25). This is a standard assignment operation, as we have done so many times before.

# Dereference operator

```
beth = *ted;
```

(that we could read as: "beth equal to value pointed by ted") beth would take the value 25, since ted is 1776, and the value pointed by 1776 is 25.




You must clearly differentiate that the expression ted refers to the value 1776, while *ted (with an asterisk * preceding the identifier) refers to the value stored at address 1776, which in this case is 25. Notice the difference of including or not including the dereference operator (I have included an explanatory commentary of how each of these two expressions could be read):

# Using Pointers

```
1   // using pointers
2   #include <iostream>
3   using namespace std;
4
5   int main ()
6   {
7     int firstvalue = 5, secondvalue = 15;
8     int * p1, * p2;
9
10    p1 = &firstvalue;  // p1 = address of firstvalue
11    p2 = &secondvalue; // p2 = address of secondvalue
12    *p1 = 10;          // value pointed by p1 = 10
13    *p2 = *p1;         // value pointed by p2 = value pointed by p1
14    p1 = p2;           // p1 = p2 (value of pointer is copied)
15    *p1 = 20;          // value pointed by p1 = 20
16
17    cout << "firstvalue is " << firstvalue << endl;
18    cout << "secondvalue is " << secondvalue << endl;
19    return 0;
20  }
```

```
firstvalue is 10
secondvalue is 20
```

10, 20, 30, 40, 50,

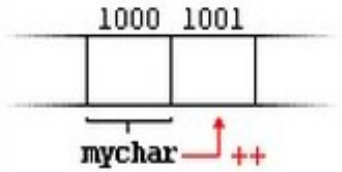
```
1   // more pointers
2   #include <iostream>
3   using namespace std;
4
5   int main ()
6   {
7     int numbers[5];
8     int * p;
9     p = numbers;  *p = 10;
10     p++;  *p = 20;
11     p = &numbers[2];  *p = 30;
12     p = numbers + 3;  *p = 40;
13     p = numbers;  *(p+4) = 50;
14     for (int n=0; n<5; n++)
15       cout << numbers[n] << ", ";
16     return 0;
17  }
```

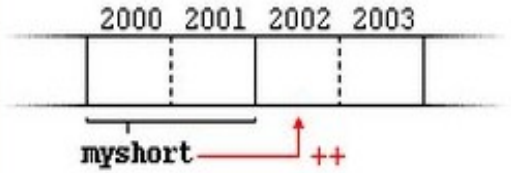
# Pointer Arithmetic

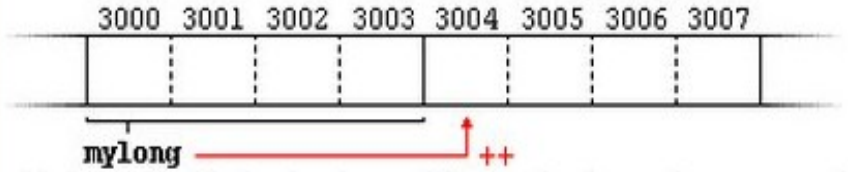
Increment is sizeof(type)

```
1 mychar++;
2 myshort++;
3 mylong++;
```

mychar, as you may expect, would contain the value 1001. But not so obviously, myshort would contain the value 2002, and mylong would contain 3004, even though they have each been increased only once. The reason is that when adding one to a pointer we are making it to point to the following element of the same type with which it has been defined, and therefore the size in bytes of the type pointed is added to the pointer.




This is applicable both when adding and subtracting any number to a pointer. It would happen exactly the same if we

## void pointers

The void type of pointer is a special type of pointer. In C++, void represents the absence of type, so void pointers are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereference properties).

This allows void pointers to point to any data type, from an integer value or a float to a string of characters. But in exchange they have a great limitation: the data pointed by them cannot be directly dereferenced (which is logical, since we have no type to dereference to), and for that reason we will always have to cast the address in the void pointer to some other pointer type that points to a concrete data type before dereferencing it.

```
1   // increaser
2   #include <iostream>
3   using namespace std;
4
5   void increase (void* data, int psize)
6   {
7     if ( psize == sizeof(char) )
8     { char* pchar; pchar=(char*)data; ++(*pchar); }
9     else if (psize == sizeof(int) )
10    { int* pint; pint=(int*)data; ++(*pint); }
11  }
12
13  int main ()
14  {
15    char a = 'x';
16    int b = 1602;
17    increase (&a,sizeof(a));
18    increase (&b,sizeof(b));
19    cout << a << ", " << b << endl;
20    return 0;
21  }
```

y, 1603

# Pointers to functions

```
1   // pointer to functions
2   #include <iostream>
3   using namespace std;
4
5   int addition (int a, int b)
6   { return (a+b); }
7
8   int subtraction (int a, int b)
9   { return (a-b); }
10
11  int operation (int x, int y, int (*functocall)(int,int))
12  {
13    int g;
14    g = (*functocall)(x,y);
15    return (g);
16  }
17
18  int main ()
19  {
20    int m,n;
21    int (*minus)(int,int) = subtraction;
22
23    m = operation (7, 5, addition);
24    n = operation (20, m, minus);
25    cout <<n;
26    return 0;
27  }
```

8

C++ allows operations with pointers to functions.

The use of this is for passing a function as an argument to another function, since these cannot be passed dereferenced.

In order to declare a pointer to a function we have to declare it like the prototype of the function except that the name of the function is enclosed between parentheses () and an asterisk (*) is inserted before the name:

# Data Structures

A data structure creates a new type: Once a data structure is declared, a new type with the identifier specified as structure_name is created and can be used in the rest of the program as if it was any other type.

Once declared, product has become a new valid type name like the fundamental ones int, char or short and from that point on we are able to declare objects (variables) of this compound new type, like we have done with apple, banana and melon

```
1. struct product {
2.   int weight;
3.   float price;
4. } ;
5.
6. product apple;
7. product banana, melon;
```

or

```
1.  struct product {
2.    int weight;
3.    float price;
4.  } apple, banana, melon;
```

```
apple.weight
apple.price
banana.weight
banana.price
melon.weight
melon.price
```

Pointers to Structures

```
Enter title: Invasion of the body snatchers
Enter year: 1978

You have entered:
Invasion of the body snatchers (1978)
```

```
1.  // pointers to structures
2.  #include <iostream>
3.  #include <string>
4.  #include <sstream>
5.  using namespace std;
6.
7.  struct movies_t {
8.    string title;
9.    int year;
10. };
11.
12. int main ()
13. {
14.   string mystr;
15.
16.   movies_t amovie;
17.   movies_t * pmovie;
18.   pmovie = &amovie;
19.
20.   cout << "Enter title: ";
21.   getline (cin, pmovie->title);
22.   cout << "Enter year: ";
23.   getline (cin, mystr);
24.   (stringstream) mystr >> pmovie->year;
25.
26.   cout << "\nYou have entered:\n";
27.   cout << pmovie->title;
28.   cout << " (" << pmovie->year << ")\n";
29.
30.   return 0;
31. }
```

# Defining data types (typedef)

C++ allows the definition of our own types based on other existing data types. We can do this using the keyword typedef

```
1. typedef char C;
2. typedef unsigned int WORD;
3. typedef char * pChar;
4. typedef char field [50];
```

In this case we have defined four data types:
C,  is char
WORD,  is unsigned int
pChar  is char pointer
field as  array of char

that we could perfectly use in declarations later as any other valid type:

```
1. C mychar, anotherchar, *ptc1;
2. WORD myword;
3. pChar ptc2;
4. field name;
```

Enumerations

```
enum enumeration_name {
  value1,
  value2,
  value3,
  .
  .
} object_names;
```

```
enum colors_t {black, blue, green, cyan, red, purple, yellow, white};
```

```
1. colors_t mycolor;
2.
3. mycolor = blue;
4. if (mycolor == green) mycolor = red;
```

We can explicitly specify an integer value for any of the constant values that our enumerated type can take. If the constant value that follows it is not given an integer value, it is automatically assumed the same value as the previous one plus one. For example:

```
1 enum months_t { january=1, february, march, april,
2                 may, june, july, august,
3                 september, october, november, december} y2k;
```

In this case, variable `y2k` of enumerated type `months_t` can contain any of the 12 possible values that go from `january` to `december` and that are equivalent to values between `1` and `12` (not between `0` and `11`, since we have made `january` equal to `1`).