# Chapter 9 References & more

CS185

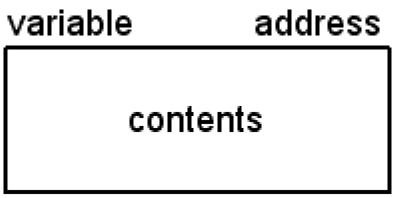
# Brief Pointer and Memory Review

Here's a brief review of pointers and memory:

We can declare pointer variables easily:

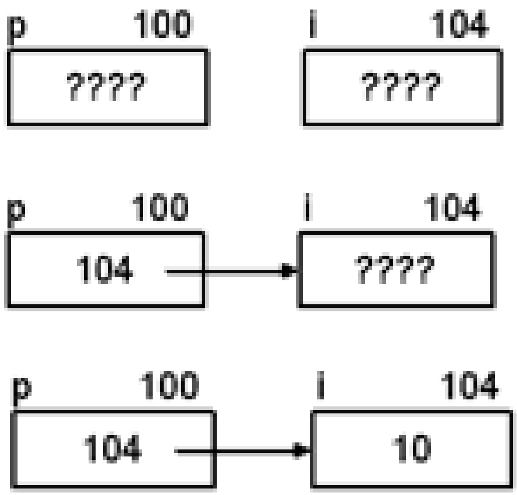
```
void foo(void)
{
       int i;  /* i can only store integer values            */
                     /* The value of i is undefined at this point  */

       int *p; /* p can only store the address of an integer */
                     /* The value of p is undefined at this point  */

       p = &i; /* The value of p is now the address of i     */
       i = 10; /* The value of i is now 10                   */
}
```

This is the notation that will be used when talking about variables in memory:



- **variable** - The name of the variable
- **address** - The arbitrary address (the actual values are meaningless, but are good for discussion purposes)
- **contents** - The value stored at this location. ???? means it is undefined.

Visualizing the code above:



***One important thing to realize is that once you name a memory location, that name cannot be used for another memory location (in the same scope). In other words, once you bind a name to a memory location, you can't unbind it:***

```
int i;   /* i is the name of this memory location               */
float i; /* i is now attempting to name this memory location (not legal) */
```

In the diagrams above, you can see that it is possible to modify i's value in two different ways.

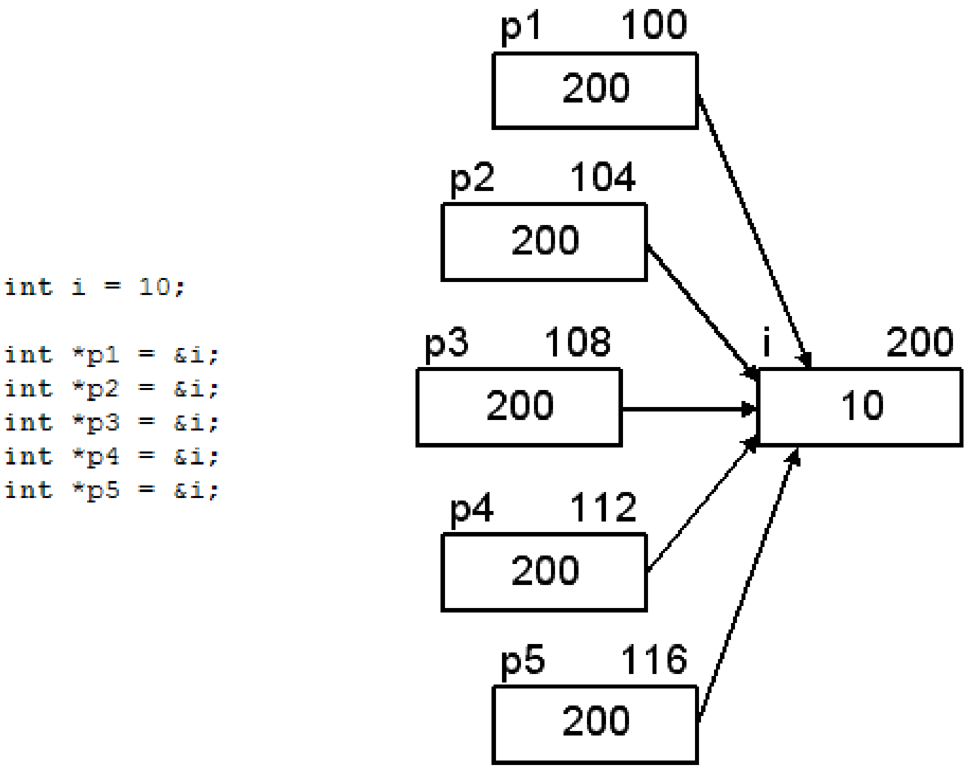
- Through the variable `i` itself:

```
i = 20;  /* The value of i is now 20 */
```

- Through the pointer `p`:

```
*p = 30; /* The value of i is now 30 */
```

In fact, we can have any number of pointers pointing to `i`:



```
int i = 10;

int *p1 = &i;
int *p2 = &i;
int *p3 = &i;
int *p4 = &i;
int *p5 = &i;
```

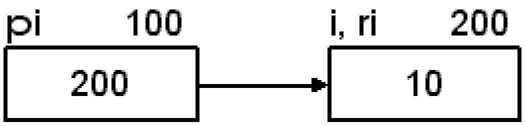
- Each of these pointers can be used to modify `i`.
- Each pointer itself can be modified to point at something else.
- Each pointer requires additional 4-bytes (size of a 32-bit pointer) in the program.
- Pointers are useful when we need to pass something to a function and have that function *modify the original*.

## References

In some way, *references* are similar to pointers, but with a cleaner syntax. A reference can be thought of as an *alias* for another variable (i.e. a memory location). An example with a diagram will make it clearer:

```
int i = 10;   // i represents an address, requires 4 bytes, holds the value 10
int *pi = &i; // pi is a pointer, requires 4 bytes, holds the address of i
int &ri = i;  // ri is an alias (another name) for i, requires no storage
```

Diagram:



- The symbol `ri` is not a new variable or a new storage location, it is another name for `i`.
- The diagram shows how `ri` is really just another name for the memory also known as `i`.
- What we have done was to *bind* the name `ri` to the memory location also known as `i`.
- `ri` can be used anywhere that `i` is used and in the same exact way.
- You do not (and cannot) dereference `ri`, since it is not a pointer.
- All of these statements modify the value stored at `i` (a.k.a. `ri`):

```
i = 20;   // i (and ri) is now 20
ri = 30;  // i (and ri) is now 30
*pi = 40; // i (and ri) is now 40
```

You can see that `i` and `ri` do, in fact, represent the same piece of memory:

**Code**

```
std::cout << " i is " << i << std::endl;
std::cout << "ri is " << ri << std::endl;

std::cout << "address of  i is " << &i << std::endl;
std::cout << "address of ri is " << &ri << std::endl;
```

**Output**

```
i is 40
ri is 40
address of  i is 0012FE00
address of ri is 0012FE00
```

Compare that with `pi`, which is a separate entity in the program:

**Code**

```
std::cout << "pi is " << pi << std::endl;
std::cout << "*pi is " << *pi << std::endl;
std::cout << "address of pi is " << &pi << std::endl;
```

**Output**

```
pi is 0012FE00
*pi is 40
address of pi is 0012FDF4
```

When you declare a reference, you must initialize it. You can't have any unbound references (or variables for that matter). In this respect, it is much like a constant pointer that must point to something when it is declared:

**Code**

```
int i;          // i is bound to a memory location by the compiler
int &r1 = i;    // r1 is bound to the same memory location as i
int &r2;        // error: r2 is not bound to anything

int * const p1 = &i; // Ok, p1 points to i
int * const p2;      // error, p2 must be initialized
p1 = &i;             // error, p1 is a constant so you can't modify it
```

**Error**

| Severity | Code | Description |
|---|---|---|
| Error | C2530 | 'r2': references must be initialized |
| Error | C2734 | 'p2': 'const' object must be initialized if not 'extern' |
| Error | C3892 | 'p1': you cannot assign to a variable that is const |

Some more subtle issues with references, mostly pertaining to constant references:

```
int i = 10;
int j = 20;

int &r1 = 5;       // Error. How do you change '5'?
const int &r2 = 5; // Ok, r2 is const (5 is put into a temporary by the compiler)

int &r3 = i;      // Ok
int &r4 = i + j;  // Error, i + j is in temporary (probably a register on the CPU)
const int &r5 = i + j; // Ok, i + j is in temp but r5 is const
```

We will see more of these issues when dealing with functions and reference parameters.
Of course, just like when you first learned about pointers, your response was: *"Yeah, so what?"*

## Reference Parameters

We don't often create a reference (alias) for another variable since it rarely provides any benefit. The real benefit comes from using references as parameters to functions.

We know that, by default, parameters are passed by value. If we want the function to modify the parameters, we need to pass the address of the data we want modified.

A classic example is the `swap` function. This function is "broken", because it passes by value.

**Code**

```
#include <iostream> // cout, endl

/* Exchanges the values of the parameters */
void swap(int a, int b)
{
  int temp = a; /* Save a for later      */
  a = b;        /* a gets value of b     */
  b = temp;     /* b gets old value of a */
}

int main(void)
{
  int x = 10;
  int y = 20;

  std::cout << "Before: x = " << x << ", y = " << y << std::endl;
  swap(x, y);
  std::cout << "After: x = " << x << ", y = " << y << std::endl;

  system("pause");
  return 0;
}
```

**Output**

```
Before: x = 10, y = 20
After: x = 10, y = 20
```

We fixed this by passing the address:

Code

```
#include <iostream> // cout, endl

/* Exchanges the values of the parameters */
void swap(int *a, int *b)
{
  int temp = *a; /* Save a for later      */
  *a = *b;       /* a gets value of b     */
  *b = temp;     /* b gets old value of a */
}

int main(void)
{
  int x = 10;
  int y = 20;

  std::cout << "Before: x = " << x << ", y = " << y << std::endl;
  swap(&x, &y);
  std::cout << "After: x = " << x << ", y = " << y << std::endl;

  system("pause");
  return 0;
}
```

Output

```
Before: x = 10, y = 20
After: x = 20, y = 10
```

In C++, we can pass by reference, which acts kind of like pass by address. The only thing that has changed between this and the first pass-by-value function is the **&** in the parameters to the `swap` function.

Code

```
#include <iostream> // cout, endl

/* Exchanges the values of the parameters */
void swap(int &a, int &b)
{
  int temp = a; /* Save a for later      */
  a = b;       /* a gets value of b     */
  b = temp;    /* b gets old value of a */
}

int main(void)
{
  int x = 10;
  int y = 20;

  std::cout << "Before: x = " << x << ", y = " << y << std::endl;
  swap(x, y);
  std::cout << "After: x = " << x << ", y = " << y << std::endl;

  system("pause");
  return 0;
}
```

| Output | Before: x = 10, y = 20<br>After: x = 20, y = 10 |
|---|---|

***Note:*** ***When you pass a parameter by reference, you are actually passing an address to the function. Roughly speaking, you get pass-by-address semantics with pass-by-value syntax. The compiler is doing all of the necessary dereferencing for you behind the scenes.***

## Another example:

Using values:

```
#include <iostream> // cout, endl

/* Assumes there is at least one element in the array  */
int find_largest1(int a[], int size)
{
       int i;
       int max = a[0]; /* assume 1st is largest */

       for (i = 1; i < size; i++)
       {
              if (a[i] > max)
              {
                     max = a[i]; /* found a larger one */
              }
       }

       return max;     /* max is the largest */
}

int main(void)
{
       int a[] = { 4, 5, 3, 9, 5, 2, 7, 6 };
       int size = sizeof(a) / sizeof(*a);

       int largest = find_largest1(a, size);
       std::cout << "Largest value is " << largest << std::endl;


       system("pause");
       return 0;
}
```

Using pointers:

```
#include <iostream> // cout, endl

/* Assumes there is at least one element in the array  */
int* find_largest2(int a[], int size)
{
       int i;
       int max = 0;  /* assume 1st is largest */

       for (i = 1; i < size; i++)
       {
              if (a[i] > a[max])
              {
                     max = i;   /* found a larger one */
              }
       }

       return &a[max]; /* return the largest */
}

int main(void)
{
       int a[] = { 4, 5, 3, 9, 5, 2, 7, 6 };
       int size = sizeof(a) / sizeof(*a);

       // Have to dereference the returned pointer
       int largest = *find_largest2(a, size);
       std::cout << "Largest value is " << largest << std::endl;

       system("pause");
       return 0;
}
```

Using references:

```
#include <iostream> // cout, endl

/* Assumes there is at least one element in the array  */
int& find_largest3(int a[], int size)
{
       int i;
       int max = 0;  /* assume 1st is largest */

       for (i = 1; i < size; i++)
       {
              if (a[i] > a[max])
              {
                     max = i;   /* found a larger one */
              }
       }

       return a[max]; /* return the largest */
}
```

```
int main(void)
{
        int a[] = { 4, 5, 3, 9, 5, 2, 7, 6 };
        int size = sizeof(a) / sizeof(*a);

        int largest = find_largest3(a, size);
        std::cout << "Largest value is " << largest << std::endl;

        system("pause");
        return 0;
}
```

Notes:

- With small data objects (like integers), passing by reference is not a huge win.
- When the data objects are large (like structures), references can be a significant improvement. References also give you the original data, not a copy, which may be required.
- Sometimes a reference is needed as an l-value and a copy (r-value) won't work.

## Some Issues with References

- When looking at your code, it is not immediately obvious that you are actually changing the original value of a parameter.
    - It's pretty obvious with pointers. (You have to dereference pointers to get at the data.)
    - The syntax for references and values is the same. (This is on purpose and is the whole point of references.)

- References must bind to an address. Consider these calls to `swapr`:

```
        swapr(a, b);      // Swap a and b
        swapr(a + 3, b);  // ???
        swapr(3, 5);      // ???
```

- Passing a large amount of data:

```
// On-screen graphic
struct Sprite
{
       double x, y;
       int weight;
       int level;
       char name[20];
};

// Pass by value
void DisplaySprite(Sprite sprite)
{
       std::cout << "x position: " << sprite.x << std::endl;
       std::cout << "y position: " << sprite.y << std::endl;
       std::cout << "    weight: " << sprite.weight << std::endl;
       std::cout << "     level: " << sprite.level << std::endl;
       std::cout << "      name: " << sprite.name << std::endl;
}

// Pass by reference
void DisplaySprite(Sprite &sprite)
{
       // Other code...
       sprite.level = 100; // Oops.
}
// Pass by const reference
void DisplaySprite(const Sprite &sprite)
{
       // Other code...
       sprite.level = 100; // Error. Compiler will catch this now.
}
```

- Returning a reference to local data:

```
// Return by value (OK)
Sprite MakeSprite(void)
{
        Sprite s = { 5.25, 2.8, 50, 1, "Pebbles" };
        return s;
}
```

```
// Return by reference (Not OK)
Sprite& MakeSprite2(void)
{
        Sprite s = { 15.0, 2.9, 100, 6, "BamBam" };
        return s;
}
```

- Warning/Error? (What does the compiler say about returning the local reference?)

```
warning C4172: returning address of local variable or temporary: s
```

**Q: Why would you use pass-by-reference instead of pass-by-address?**
**A: When we start working with classes and objects, we'll see that references are much more natural than pointers.**

**Also, with references, the caller can't tell the difference between passing by value and passing by reference. This allows the caller to *always* use the same syntax and let the function decide the optimal way to receive the data.**

# Default Parameters

Examples:

**Code**

```
#include <iostream> // cout, endl

void print_array(int a[], int size)
{
    for (int i = 0; i < size; i++)
    {
        std::cout << a[i];
        if (i < size - 1)
        {
            std::cout << ", ";
        }
    }
    std::cout << std::endl;
}

int main(void)
{
    int a[] = { 4, 5, 3, 9, 5, 2, 7, 6 };
    int size = sizeof(a) / sizeof(*a);

    print_array(a, size);

    return 0;
}
```

**Output**

```
4, 5, 3, 9, 5, 2, 7, 6
```

Change the formatting:

**Code**

```
#include <iostream> // cout, endl

void print_array2(int a[], int size)
{
        for (int i = 0; i < size; i++)
        {
                std::cout << a[i] << std::endl;
        }
}

int main(void)
{
        int a[] = { 4, 5, 3, 9, 5, 2, 7, 6 };
        int size = sizeof(a) / sizeof(*a);

        print_array2(a, size);

        return 0;
}
```

**Output**

```
4
5
3
9
5
2
7
6
```

Adding a default parameter to the function:

Code

```cpp
#include <iostream> // cout, endl

void print_array(int a[], int size, bool newlines = false)
{
        for (int i = 0; i < size; i++)
        {
                std::cout << a[i];
                if (i < size - 1)
                {
                        if (newlines == true)
                        {
                                std::cout << std::endl;
                        }
                        else
                        {
                                std::cout << ", ";
                        }
                }
        }
        std::cout << std::endl;
}

int main(void)
{
        int a[] = { 4, 5, 3, 9, 5, 2, 7, 6 };
        int size = sizeof(a) / sizeof(*a);

        // Calls with (a, size, false)
        print_array(a, size);
        print_array(a, size, false);

        // Calls with (a, size, true)
        print_array(a, size, true);


        system("pause");
        return 0;
}
```

Output

```
4, 5, 3, 9, 5, 2, 7, 6
4, 5, 3, 9, 5, 2, 7, 6
4
5
3
9
5
2
7
6
```

Another example:

**Code**

```
#include <iostream> // cout, endl

int& Inc(int& value, int amount = 1)
{
       value += amount;
       return value;
}


int main(void)
{
       int i = 10;
       std::cout << Inc(i) << std::endl;
       std::cout << Inc(i) << std::endl;
       std::cout << Inc(i, 2) << std::endl;
       std::cout << Inc(i, 4) << std::endl;
       std::cout << Inc(i, 5) << std::endl;

       return 0;
}
```

**Output**

```
11
12
14
18
23
```

You can have multiple default parameters:

```
void foo(int a, int b, int c = 10);

foo(1, 2);    // foo(1, 2, 10)
foo(1, 2, 9); // foo(1, 2, 9)


void foo(int a, int b = 8, int c = 10);

foo(1);       // foo(1, 8, 10)
foo(1, 2);    // foo(1, 2, 10)
foo(1, 2, 9); // foo(1, 2, 9)


void foo(int a = 5, int b = 8, int c = 10);

foo();        // foo(5, 8, 10)
foo(1);       // foo(1, 8, 10)
foo(1, 2);    // foo(1, 2, 10)
foo(1, 2, 9); // foo(1, 2, 9)
```

These two functions are illegal because of the ordering of the default parameters:

```
void foo(int a, int b = 8, int c);
void foo(int a = 5, int b, int c = 10);
```

Notes:

- Functions that use default parameters will **always** get parameters. (They will never be "missing".)
- If the programmer doesn't pass them (by explicitly putting them in the function call), the compiler will pass them. Period.
- Your code will never need to worry about "Oh, no! What happens if they don't pass me anything?!?!?! OMGWTFBBQ!!!".
- When using default parameters, you can only use defaults from right to left (as shown above).
- You need to put the default parameters in the prototype, not the implementation. (The compiler needs them, not the linker)

# Overloaded Functions

```
#include <iostream> // cout, endl

int cube(int n)
{
       return n * n * n;
}

int main(void)
{
       int i = 8;
       long l = 50L;
       float f = 2.5F;
       double d = 3.14;

       // Works fine: 512
       std::cout << cube(i) << std::endl;

       // May or may not work: 125000
       std::cout << cube(l) << std::endl;

       // Not quite what we want: 8
       std::cout << cube(f) << std::endl;

       // Not quite what we want: 27
       std::cout << cube(d) << std::endl;


       system("pause");
       return 0;
}
```

First attempt, "old school" fix in C:

```
int cube_int(int n)
{
        return n * n * n;
}
```

```
double cube_double(double n)
{
        return n * n * n;
}
```

```
float cube_float(float n)
{
        return n * n * n;
}
```

```
long cube_long(long n)
{
        return n * n * n;
}
```

It will work as expected:

```
// Works fine: 512
std::cout << cube_int(i) << std::endl;

// Works fine: 125000
std::cout << cube_long(l) << std::endl;

// Works fine: 15.625
std::cout << cube_float(f) << std::endl;

// Works fine: 30.9591
std::cout << cube_double(d) << std::endl;
```

This quickly becomes tedious and unmanageable as we write other functions to handle other types such as **unsigned int, unsigned long, char**, as well as user-defined types that might come along.

Using overloaded functions in C++:

```
int cube(int n)
{
        return n * n * n;
}
```

```
double cube(double n)
{
        return n * n * n;
}
```

```
float cube(float n)
{
        return n * n * n;
}
```

```
long cube(long n)
{
        return n * n * n;
}
```

It will also work as expected without the user needing to choose the right function:

```
// Works fine, calls cube(int): 512
std::cout << cube(i) << std::endl;

// Works fine, calls cube(long): 125000
std::cout << cube(l) << std::endl;

// Works fine, calls cube(float): 15.625
std::cout << cube(f) << std::endl;

// Works fine, calls cube(double): 30.9591
std::cout << cube(d) << std::endl;
```

Now, if we decide we need to handle another data type, we simply *overload* the **cube** function to handle the new type. The users (clients) of our code have no idea that we implement the cube function as separate functions.

More example uses:

```
int i = cube(2);           // calls cube(int), i is 8
long l = cube(100L);       // calls cube(long), l is 1000000L
float f = cube(2.5f);      // calls cube(float), f is 15.625f
double d = cube(2.34e25);  // calls cube(double), d is 1.2812904e+76
```

Notes

- Overloaded functions have the same name but different parameters.
- There are three attributes to the parameters: type, order, and number. These are all valid overloads:

```
void foo(double);
void foo(int);
void foo(int, int);
void foo(int, double);
void foo(double, int);
void foo(void);
```

- The return value is **not** used to distinguish between overloaded functions.

```
int foo(double);
float foo(double); // Ambiguous
```

An error message:

```
error: new declaration 'float foo(double)'
error: ambiguates old declaration 'int foo(double)'
```

- Ambiguity can be a problem:

| | |
|---|---|
| **Code** | `void foo(double);`<br>`void foo(float);`<br><br>`foo(1.0F); // Calls foo(float)`<br>`foo(1.0);  // Calls foo(double)`<br>`foo(1);    // Which one?` |
| **Error** | `error: call of overloaded 'foo(int)' is ambiguous`<br>`note: candidates are: void foo(double)`<br>`note:  void foo(float)` |

These are not technically ambiguous. The problem is that the second one is a *redefinition* of the first one. (They are considered the same.)

```
void foo(const int);
void foo(int);
```

The compiler can distinguish between these two:

```
void foo(int&);
void foo(const int&);
```

Try your compiler to answer these questions:

```
void foo(int&)
{
       std::cout << "int&\n";
}

void foo(const int&)
{
       std::cout << "const int&\n";
}

int main(void)
{
       int i = 1;
       const int j = 2;

       foo(5);       // Which one?
       foo(i);       // Which one?
       foo(j);       // Which one?
       foo(i + j);   // Which one?
       foo((int&)j); // Which one?

       return 0;
}
```

These are also different:

```
void foo(int&) { std::cout << "int&\n"; }
void foo(int) { std::cout << "int\n"; }
```

Finally, we can have problems when mixing overloading and defaults:

```
void foo(int a, int b = 10);
void foo(int a);

foo(5, 6); // Ok
foo(5);    // Ambiguous
```

## Understanding the Big Picture™

What problems are solved by

1. references?
2. reference parameters?
3. default parameters?
4. overloaded functions?