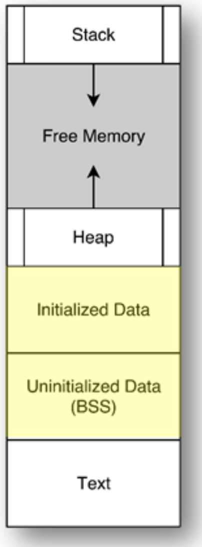
Memory & Addresses

One of the principles behind the design of C++ is that programmers should have **as much access as possible** to the underlying hardware. For this reason, C++ makes **memory addresses** explicitly visible to the programmer. An object whose value is an address in memory is called **a pointer**.



The illustration on the right provides a rough sketch as to how memory is organized in a typical C++ program when it is loaded from disk and run. The **machine instructions** are put into the **text** or (**code**) section. This section of memory is read-only and protected by the operating system.

Global variables and `static` variables are stored in **the static area** (which, in the illustration is marked as Initialized Data and Uninitialized Data). You can read and write data to this area of memory, but variables stored here **don't move around**. They are stored when the **program loads** and before it starts executing.

At the opposite end of memory is **the stack**. Each time your program calls a function the computer creates a new **stack frame** in this memory region, containing parameters, local variables and the return address. When that function returns, the stack frame is discarded leaving the memory free to be used for the subsequent calls.

The region between the end of the program data and the stack is called **the heap** which is used for dynamically allocated memory. We'll study dynamic memory allocation a little later in this course. It will be a big part of CS 250.

Where a variable is stored depends on **how** the variable is defined. Click the "Running Man" on the left to visualize three variables.

Naming Concepts

Three terms are used to describe the characteristics of a variable or function name:

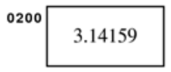
1. **Scope**: where the name is **visible**.
    a. Variables with **block scope** are visible from the point of their declaration to the end of the block where they are declared. Local variables and parameters have block scope.
    b. Functions and global variables have **file scope**; they are visible from the point of declaration to the end of the file in which they are declared.
2. **Storage** and **duration**: where a variable is located, and how long it stays there.
    a. Variables in **static storage** are placed there when the program starts running and stay at the same address until the program is finished. Global variables and local `static` variables have static storage class.
    b. Variables with **automatic storage**, are placed on the stack when they are defined, and then destroyed when the block they are defined in ends; automatic variables always have block scope. Local variables and parameters have automatic storage.
    c. **Dynamic storage** is determined by the programmer; dynamic variables are placed on the heap and removed from the heap in response to specific programmer commands, such as `new` and `delete`.
3. **Linkage**: how variables and functions can be shared between different files.
    a. **External linkage** means that a global variable or function can be used from other files. This is the normal case with global variables.
    b. **Internal linkage** means that a "global" variable or function is only visible to other functions in the same file. This is indicated by placing the keyword `static` before the definition of the variable or function.
    c. **No linkage** means that a variable cannot be used inside any other function. All local variables and parameters have no linkage.

Global Variables

Global variables—usually constants in this class—are allocated in the static storage area. Thus, if the compiler sees the definition below (outside of any function), it reserves eight bytes in the static area, and stores the literal value when the **program is compiled.**

```
const double kPi = 3.14159;
```

As a programmer, you have no idea **what** address the compiler will choose, but it often helps you to visualize what is happening if you make up an address and use that in a diagram. Here you might imagine that the constant `kPi` is stored in the address `0200`.



Most platforms support a much more accurate value for `PI`. We can calculate that value using the expression `acos(-1.0)` **at run-time**.

```
const auto kPi = acos(-1.0);
```

This produces the following output when printed with 16 digits of precision:

```
kPI->3.1415926535897931
```

Local Variables

Parameter variables, and variables created inside a function, are local variables, **allocated on the stack** in a block of memory called a **stack frame**. Internally, these variables are **pushed onto the top of the stack** at the time of each function call.

The same local variable may be stored at a different address each time the function is called. In fact, when we covered **recursion** earlier in the semester, we saw that there may be **multiple copies** of the **same local variable**, each stored at a different location on the stack. This is what makes recursion possible.

Local static Variables

A local variable that uses the `static` modifier is not stored on the stack, but **in the static storage area**, like a global variable. As far as its storage class goes, it is a global variable, but as far as its scope and linkage goes, it is a local variable.

Characteristics of Variables

Every variable has at least three characteristics.

- **Name**: used to access the data in your code.
- **Type**: used to determine the amount of memory required to store the variable, the representation or interpretation of the bits stored in memory at that location, and the operations that are legal on that memory location.
- **Value**: the **meaning** of the bits stored at the memory location selected by the compiler, when interpreted according to its type.

When you define a variable in a C++ program, the compiler makes sure that the variable is **allocated enough memory** to hold a value of that type. Here's an example:

```
int a = 3;          // name->a, type->int, value->3
auto b = 3.14159;   // name->b, type->double, value->pi
cout << a << endl;  // print value
cout << b << endl;  // print value
```

The sizeof Operator

The `sizeof` operator returns the amount of memory allocated for a variable. The operator takes a single operand, which must be **either an expression**, such as the name of a variable or a **type name**. Type names must be enclosed in parentheses.

If used with a variable or an expression, the `sizeof` operator returns the **number of bytes** required to store the value of that expression. If used with a **type**, `sizeof` returns the number of bytes required to store a value **of that type**.

```
int a = 42;
cout << sizeof a << endl;        // 4 (on our platform)
cout << sizeof(double) << endl; // 8
cout << sizeof 7LL + 4 << endl; // 12-WHY?
```

The first line prints the bytes required to store the `int` variable `a`; the second line prints the number of bytes required to store **any value** of type `double`. The third is more confusing. On our platform, a `long long` should take `8` bytes, but this prints `12`. **Why?**

Simple: `sizeof` is a **unary** operator. That means that the expression shown here reads as `sizeof(7LL) + 4` which is `8 + 4` or `12`. The fix is equally simple: **always parenthesize arguments** to the `sizeof` operator, **even when they are not needed**.

The Address Operator

The address operator, `&`, when placed in front of a variable, returns the address where the variable is stored in memory.

```
cout << "&a->" << &a << endl;
cout << "&b->" << &b << endl;
```

Addresses are normally printed in hexadecimal, and depend on the size of the pointer. Here's the output from this fragment of code when run on two platforms:

```
&a->009CF808 - Visual C++ 19 (Windows)
&b->009CF7F8
&a->0x7fff448e448c - G++ & Clang (Unix)
&b->0x7fff448e4490
```

Notice that Visual Studio has 32-bit addresses, while Unix uses 64-bit. Of course, the actual values of the addresses printed on your machine will be entirely different. You can take the address of a variable, such as `&a` or `&b`, **but not a type**. Writing `&int` is nonsensical and illegal.

You also cannot take the address of a literal or expression: `&12` is meaningless.

Pointers

A pointer is variable that contains the address of another variable. In languages like Java, C# and Python, pointers are **hidden** from the programmer, and used only by the runtime system. In C++, understanding pointers is necessary for understanding how C++ programs work.

An expression that refers to an object in memory is an ***lvalue***. Variables are ***lvalues*** because you can store data in them. A named constant is a **non-modifiable *lvalue***. Many values in C++ **are not *lvalues***; the result of an expression **is a temporary value**, but it **is not** an ***lvalue***, because you cannot assign a new value it.

The following properties apply to modifiable ***lvalues*** in C++:

- Every ***lvalue*** is stored somewhere in memory; thus it **has an address**.
- **The address of an *lvalue* never changes**, even though the contents of those memory locations may change.
- The address of an ***lvalue*** is a **pointer or address value**, which can be stored in memory and manipulated as data.

To store an address value in memory, you create a **pointer variable**. Thus, a pointer is simply a variable that stores the address of some object in memory.

Defining Pointers

To define a pointer, add an asterisk (`*`) between the variable type and the variable name in the variable definition. Here, `p` is a pointer variable that "points to" an `int`; its type is **pointer-to-int**.

```
int *p;
```

In this context, `*` is the **pointer declarator operator**. It turns the name on its right into a pointer to the type on its left. The line below defines `cptr`, a **pointer-to-char**.

```
char *cptr;
```

Even though `p` and `cptr` are both **pointers**, **each is a distinct type**; pointers are very strongly typed and there are **no implicit conversions between pointer types**.

A pointer **belongs syntactically with the variable name** and not with the base type.

```
int* p1, p2;      // p1 is a pointer, p2 an int
int *p3, *p4;     // Both are pointers
```

If you use the same declaration to define two pointers of the same type, you need to mark each of the variables with an asterisk.

Initializing Pointers

A pointer can be in one of four states:[1]

1. It can point to a **valid object**.
2. It can point **one-past** a valid object (in an array or `vector` for instance.
3. It can contain the value `nullptr` to indicate it points to "nothing", or is unused.
4. It can **be invalid**, such as an uninitialized pointer.

You can **initialize a pointer** in several ways.

- With the address of another object obtained from the **address operator.**
- With the address of an object **created on the heap** with the `new` operator.
- With the **name of a previously defined array**.
- By using **pointer assignment** to copy the address from another pointer

If you don't initialize a pointer, **it is invalid**. Here are examples of each of these:

```
int x{42}, y{0}, a[10]; // x->int, y->int, a->array

int *p1{&y};          // points to y
int *p2{&x};          // points to x
int *p3{new int{3}}; // points to int on heap
int *p4{a};          // points to first element of a
int *p5{a+10};       // points to "one past" the array a
int *p6{nullptr};    // points to "nothing"
int *p7;             // uninitialzed (invalid)
```

[1] Lippmann, C++ Primer, 5th Edition, Page 52, Section 3.3.2

Dereferencing Pointers

Let's look at the list of pointers on the previous page again.

```
int x{42}, y{0}, a[10]; // x->int, y->int, a->array

int *p1{&y};          // points to y
int *p2{&x};          // points to x
int *p3{new int{3}}; // points to int on heap
int *p4{a};           // points to first element of a
int *p5{a+10};        // points "one past" the array a
int *p6{nullptr};     // points to "nothing"
int *p7;              // uninitialzed (invalid)
```

The `*` **dereferencing operator** returns the value that a pointer points to, **provided that** the pointer points to a **valid object**, such as `p1` and `p2`. Using the dereferencing operator on `p5`, `p6` or `p7` produces **undefined behavior**. The value that a pointer "points to" is called its **indirect value**.

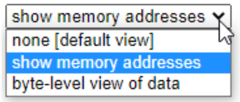
Since `p1` is a pointer to `int`, the compiler "knows" that `*p1` **must be an integer object**. Thus `*p1` turns out to be **another name (or alias) for the variable** `y`. Like the simple name `y`, `*p1` is an ***lvalue*** , and you can assign new values to it.

```
int x{42}, y{0};
int *p1{&y};         // points to y
int *p2{&x};         // points to x
*p1 = 17;            // assign to indirect value
```

This last statement changes the value in the variable `y` because that is the target of the pointer `p1`. `p1` **is unaffected** by this assignment; it continues to point to the variable `y`. Click the little running-man on the left to see this animated in a new window.

Pointer Assignment

It is also possible to assign new values to the pointer variables themselves. Look at this animation. Before you do, change the drop-down list so that it says "show memory addresses".



Line 6 makes a copy of the **direct value** (that is the address) stored in `p2` and copies it into the variable `p1`. Afterwards, both variables now point to the same location.

If you draw your diagrams using arrows, keep in mind that copying a pointer replaces the destination pointer with a new arrow that points to the same location as the old one. Thus `p1` = `p2` changes the arrow leading from `p1` so it points to the same location as the arrow originating from `p2`.

It is important to distinguish the assignment of a pointer from that of a value. ***Pointer assignment****,* `p1 = p2`*, makes* `p1` *and* `p2` *point to the same location. By contrast,* ***value assignment****,* `*p1 = *p2`*, copies the value from the location pointed to by* `p2` *into the location pointed to by* `p1`*.*

The "null" Pointer

The value that indicates that a pointer is not being used is called the null pointer. It is represented internally by `0`. While you **cannot assign an arbitrary integer** to a pointer variable, you **can assign the value `0`**.

Using a literal `0`, however, makes it hard to find all of the null pointers in your code. C++11 introduced **an actual null pointer constant** named `nullptr`. You should use that instead `0`. Do not use the C language value `NULL`.

It is **illegal to dereference** a null pointer. In UNIX, it usually results in a **segmentation fault**, but that is **not guaranteed**. Some machines return the contents of address `0000`. As a result, **this is undefined behavior**, as in the case of uninitialized pointers.

If you declare a pointer but **fail to initialize it**, the computer tries to interpret the contents of that pointer as an address and tries to read that region of memory. Such programs can fail in ways that are extremely difficult to detect. Again, this is **undefined behavior**.

Call By Pointer

When working with functions, you can simulate call by reference by using explicit pointer parameters. Many programmers prefer to do this because it is obvious that you are passing by pointer instead of by value at the function call site.

Let's look at a `swap()` function that **exchanges the values** contained in two integers. The algorithm is simple, assuming the two parameters are `a` and `b`. Using reference parameters, we write it like this:

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

When you **call the function**, there is **no indication** that `x` and `y` will be changed.

```
swap(x, y);     // does this change x and y? Can't tell here!
```

To use **explicit pointer parameters**, change the implementation of `swap()` like this:

```
void swap(int *a, int *b) {
    int temp{*a};
    *a = *b;
    *b = temp;
}
```

Then, **call the function** like this. The effect is the same as pass-by-reference, but you can tell at the call-site that `x` and `y` may be changed in the function.

```
int x{3}, y{4};
swap(&x, &y);     // explicit address passing
cout << "x->" << x << ", y->" << y << endl;
// x->4, y->3
```

Pointers & const

Pointers have two values: an indirect and an explicit (or direct) value. Either (or both) may be `const`. Consider this code.

```
string a{"A"}, b{"B"}, c{"C"};
const string *ps1 = &a;
string * const ps2 = &b;
const string * const ps3 = &c;
```

Note the word `const` in the declaration of `p1`, `p2` and `p3`.

- Prevent **writing to the pointer's indirect value**, by putting the const **before the type** (`ps1`). Thus `*ps1 = "x";` is illegal.
- When `const` comes **after the star**, (`ps2`), it means that the pointer itself cannot be changed; you **cannot make it point to a different location**. It would be illegal to write `ps2 = &a;`
- Prevent changing either the pointer, **or** what it points to, by using **both** versions of `const` (`ps3`).

When you write a function which **should not** change the item that it points to, make sure you define it as a "pointer to `const`. For instance, consider this template function which prints both the address and value of any variable:

```
template <typename T>
void printData(const T* p)
{
    cout << "Pointer: " << p << "->";
    if (p != nullptr) cout << *p << endl;
    else cout << "nullptr" << endl;
}
```

Because the parameter is a "pointer to `const`, you can pass the function **both** `const` and non-`const` variables. It works with everything because it **guarantees** that it won't inadvertently change the object that p points to.