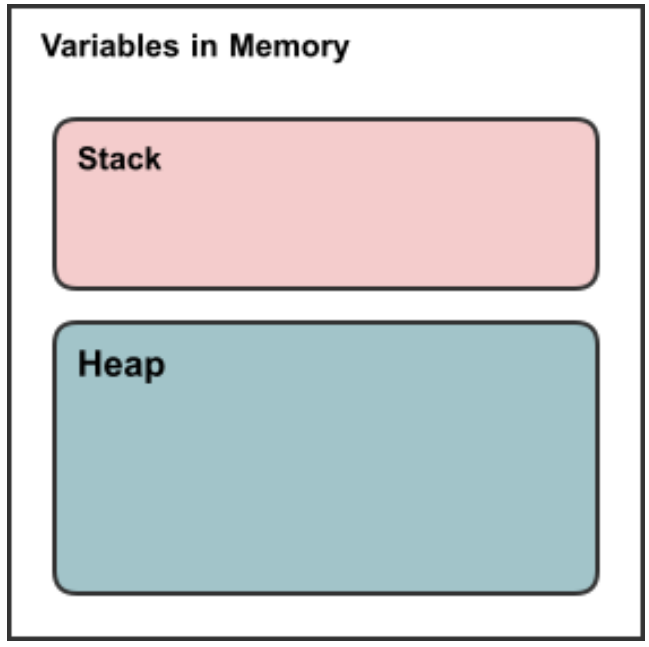
Memory Layout

Memory Layout



Variables on Stack Memory

Variables on Stack Memory

The following variables are allocated on the stack

```
char x = 's';

float pi = 3.1415;

int k = 0;

int j = k;

double e {2.71828};
```

Variables on Stack Memory

The following variables are allocated on the stack

```
char x = 's';

float pi = 3.1415;

int k = 0;

int j = k;

double e {2.71828};
```

- Automatically deleted after going out of their scope.

Variables on Stack Memory

The following variables are allocated on the stack

```
char x = 's';

float pi = 3.1415;

int k = 0;

int j = k;

double e {2.71828};
```

- Automatically deleted after going out of their scope.
- Very simple.

Address of a Variable in the Memory

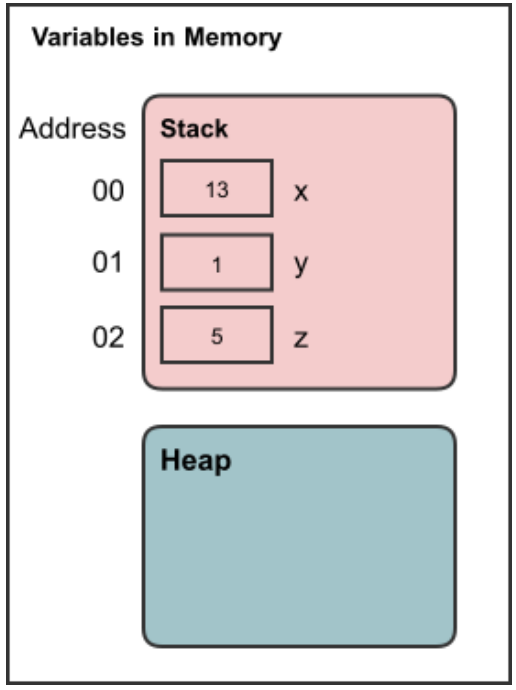
- Variables exist in memory.
- A variable in memory has an address.

```
// Declare integer x and initialize it with 13.
int x = 13;

// Declare integer y and initialize it with 1.
int y = 1;

// Declare integer z and initialize it with 5.
int z = 5;
```

Their physical presence in memory looks like this:



Address of a variable in C++

Address of a variable in C++

But how to get (retrieve) the address of a variable in C++?

Address of a variable in C++

But how to get (retrieve) the address of a variable in C++?

- By using & (ampersand operator).

Address of a variable in C++

But how to get (retrieve) the address of a variable in C++?

- By using & (ampersand operator).

```
int x = 5;

std::cout << &x << "\n";
// Prints: the location of x in memory
```

Pointers

Pointers

We store the address of a variable in a special type called **pointer**.

- **Pointer** is a primitive data type.
- **Pointer** type occupies **8 bytes** (64-bit machines).
- **Pointer** is declared using the syntax: `int *` for pointer to integer, `double *` for pointer to doubles, ...etc.

Pointers

We store the address of a variable in a special type called **pointer**.

- **Pointer** is a primitive data type.
- **Pointer** type occupies **8 bytes** (64-bit machines).
- **Pointer** is declared using the syntax: `int *` for pointer to integer, `double *` for pointer to doubles, ...etc.

```
int x = 13;
int y = 1;
int z = 5;
// Declare 'pointer to integer' px and
// initialize with address of x.
int *px = &x;

// Declare 'pointer to integer' py and
// initialize with address of y.
int *py = &y;

// Declare 'pointer to integer' pz and
// initialize with address of z.
int *pz = &z;
```

Primitive Data Types in C++ (Revisited)

Primitive Data Types (PDT) in C++

- `bool`: holds logical value, occupies **1 byte** of memory.
- `char`: a character, occupies **1 byte** of memory.
- `int`: an integer, occupies **4 bytes** of memory.
- `float`: a real-number-like, occupies **4 bytes** of memory.
- `double`: like float, but higher precision, occupies **8 bytes** of memory.

Primitive Data Types in C++ (Revisited)

Primitive Data Types (PDT) in C++

- `bool`: holds logical value, occupies **1 byte** of memory.
- `char`: a character, occupies **1 byte** of memory.
- `int`: an integer, occupies **4 bytes** of memory.
- `float`: a real-number-like, occupies **4 bytes** of memory.
- `double`: like float, but higher precision, occupies **8 bytes** of memory.
- pointer: holds the location of a variable in memory, occupies **8 bytes** of memory.

Why using Address?

Why using Address?

Flexibility

Addresses gives a great flexibility to control variables. For example, you can modify a variable value if you have its address.

```
int x = 9;

std::cout << x << std::endl; // prints: 9

int *px = &x ;

// Derefrencing px to access x.
*px = 13;

std::cout << x << std::endl; // prints 13
```

Passing arguments by pointer

Passing arguments by pointer

You can pass a **pointer to variable** as argument to a function.

Passing arguments by pointer

You can pass a **pointer to variable** as argument to a function.

```
void max( double a , double b , double *presults )
{
    // Dereference the presults to access the underlying variable.
    if( a > b ) *presults = a;
    else *presults = b;
}

int main()
{
    double results = 0; double x = 0; double y = 0;
    std::cin >> x >> y;
    max( x , y , &results ); // Now results has new value.
    std::cout << results << "\n";
}
```

Passing arguments by pointer

You can pass a **pointer to variable** as argument to a function.

```
void max( double a , double b , double *presults )
{
    // Dereference the presults to access the underlying variable.
    if( a > b ) *presults = a;
    else *presults = b;
}

int main()
{
    double results = 0, x = 0, y = 0;
    std::cin >> x >> y;
    max( x , y , &results ); // Now results has new value.
    std::cout << results << "\n";
}
```

Passing arguments by pointer

You can pass a **pointer to variable** as argument to a function.

```
void max( double a , double b , double *presults )
{
    // Dereference the presults to access the underlying variable.
    if( a > b ) *presults = a;
    else *presults = b;
}

int main()
{
    double results = 0, x = 0, y = 0;
    std::cin >> x >> y;
    max( x , y , &results ); // Now results has new value.
    std::cout << results << "\n";
}
```

- this style acceptable in C language.
- not preferred in C++, and always prefer to return the results.

Cont'd

Which is better?

This?

```
void max( double a , double b , double *presults )
{
    if( a > b ) *presults = a;
    else *presults = b;
}

int main()
{
    double results = 0, x = 0; y = 0;
    std::cin >> x >> y;
    max( x , y , &results );
    std::cout << results << "\n";
}
```

Cont'd

Which is better?

Or this?

```
double max( double a , double b )
{
    if( a > b) return a;
    else return b;
}

int main()
{
    double x = 0; y = 0;
    std::cin >> x >> y;
    double results = max( x , y );
    std::cout << results << "\n";
}
```

Stack Memory vs. Heap Memory

Stack Memory	Heap Memory
Limited capacity	Large capacity for scalable structures
Automatic memory management	Manual memory management

Variables on Heap Memory

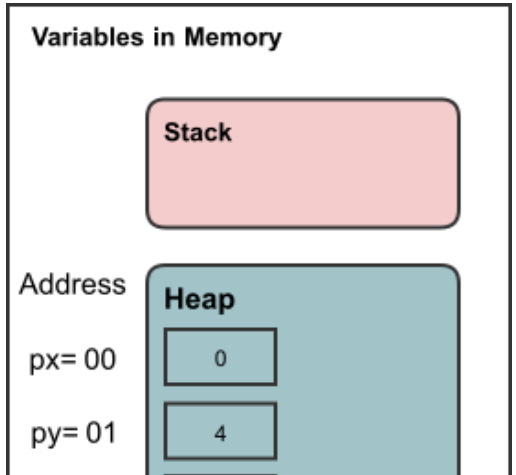
Variables can also be created on heap.

```
// Allocate integer with initializing to zero
// on heap memory, and save the address in px.
int *px = new int{0};

// Allocate integer with initializing to 4
// on heap memory, and save the address in py.
int *py = new int(4);


int *pz = new int(8);
```

Physically, they would look like this:



Memory Management

Memory Management

- Variables created on heap memory (using `new` operator), should be deleted manually when they are no longer used.
- Otherwise, you will allocate a lot of space that will become unusable.

Memory Management

- Variables created on heap memory (using `new` operator), should be deleted manually when they are no longer used.
- Otherwise, you will allocate a lot of space that will become unusable.

```
int *px = new int{0};

int *py = new int(4);

int *pz = new int(8);
```

Memory Management

- Variables created on heap memory (using `new` operator), should be deleted manually when they are no longer used.
- Otherwise, you will allocate a lot of space that will become unusable.

```
int *px = new int{0};

int *py = new int(4);

int *pz = new int(8);
```

- After making some prcessing on `px`, `py`, and `pz`

```
delete px;
delete py;
delete pz;
```

Important rule for memory management

Important rule for memory management

- To avoid memory leaks, make sure that allocations/deallocations are balanced in the en.

Important rule for memory management

- To avoid memory leaks, make sure that allocations/deallocations are balanced in the en.
- # `new` = # `delete`.

Reference types

Reference types

- Very important type in C++,

Reference types

- Very important type in C++,
- Using it in the right way makes your program very efficient.

Reference types

- Very important type in C++,
- Using it in the right way makes your program very efficient.
- **References** are alternative for pointers to enhance the readability of your code.

Reference types

- Very important type in C++,
- Using it in the right way makes your program very efficient.
- **References** are alternative for pointers to enhance the readability of your code.
- When you make a reference to a variable, you actually making an alias to that variable.

Reference types

- Very important type in C++,
- Using it in the right way makes your program very efficient.
- **References** are alternative for pointers to enhance the readability of your code.
- When you make a reference to a variable, you actually making an alias to that variable.
- In other words, you are making another name for the same variable.

Primitive Data Types in C++ (Revisited 2)

Primitive Data Types (PDT) in C++

- `bool`: holds logical value, occupies **1 byte** of memory.
- `char`: a character, occupies **1 byte** of memory.
- `int`: an integer, occupies **4 bytes** of memory.
- `float`: a real-number-like, occupies **4 bytes** of memory.
- `double`: like float, but higher precision, occupies **8 bytes** of memory.

Primitive Data Types in C++ (Revisited 2)

Primitive Data Types (PDT) in C++

- `bool`: holds logical value, occupies **1 byte** of memory.
- `char`: a character, occupies **1 byte** of memory.
- `int`: an integer, occupies **4 bytes** of memory.
- `float`: a real-number-like, occupies **4 bytes** of memory.
- `double`: like float, but higher precision, occupies **8 bytes** of memory.
- **pointer**: holds the location of a variable in memory, occupies **8 bytes** of memory.

Primitive Data Types in C++ (Revisited 2)

Primitive Data Types (PDT) in C++

- `bool`: holds logical value, occupies **1 byte** of memory.
- `char`: a character, occupies **1 byte** of memory.
- `int`: an integer, occupies **4 bytes** of memory.
- `float`: a real-number-like, occupies **4 bytes** of memory.
- `double`: like float, but higher precision, occupies **8 bytes** of memory.
- **pointer**: holds the location of a variable in memory, occupies **8 bytes** of memory.
- **reference**: an alias to an existing variable, occupies **8 bytes** of memory.

References in C++

References in C++

```cpp
// Declaration of integer x and initializing with zero.
int x = 0;

// Declaration of reference y and to be reference for x.
int &y = x;

// Now x and y, are the same variable, but with different name.

// Chaning y value, will also affect x, and vice versa.
y = 10;

std::cout << x << "\n"; // prints: 10
```

Cont'd

Recall the example of passing pointer as argument:

```
void max( double a , double b , double *presults )
{
    if( a > b ) *presults = a;
    else *presults = b;
}

int main()
{
    double results = 0, x = 0, y = 0;
    std::cin >> x >> y;
    max( x , y , &results );
    std::cout << results << "\n";
}
```

This can be written in more elegant way using references:

```
void max( double a , double b , double &results )
{
    // No need for dereference as we did in pointers, like it is a real v
    if( a > b ) results = a;
    else results = b;
}

int main()
{
    double results = 0;

    // No need to pass the address explicitly.
    max( 13 , 5 , results );

    std::cout << results << "\n";
}
```

Rule: Keep it simple, stupid (KISS)

More about {KISS} principle.

But again, it is very preferred to use the simplest form when possible!

```
double max( double a , double b )
{
    return (a > b)? a : b;
}

int main()
{
    double results = max( 13 , 5 );
}
```

We used pointer and references in previous examples just for explanations!

Thank you